HARMONISED EUROPEAN STANDARD
CYBER; CRA; Essential cybersecurity requirements for Browsers

Reference

<Workitem>

Keywords

<keywords>

ETSI

650 Route des Lucioles

F-06921 Sophia Antipolis Cedex - FRANCE

Tel.: +33 4 92 94 42 00 Fax: +33 4 93 65 47 16

Siret N° 348 623 562 00017 - APE 7112B

Association à but non lucratif enregistrée à la

Sous-préfecture de Grasse (06) N° w061004871

Important notice

Notice of disclaimer & limitation of liability

The information provided in the present deliverable is directed solely to professionals who have the appropriate degree of experience to understand and interpret its content in accordance with generally accepted engineering or

other professional standard and applicable regulations.

No recommendation as to products and services or vendors is made or should be implied.

No representation or warranty is made that this deliverable is technically accurate or sufficient or conforms to any law and/or governmental rule and/or regulation and further, no representation or warranty is made of merchantability or fitness for any particular purpose or against infringement of intellectual property rights.

In no event shall ETSI be held liable for loss of profits or any other incidental or consequential damages.

Any software contained in this deliverable is provided "AS IS" with no warranties, express or implied, including but not limited to, the warranties of merchantability, fitness for a particular purpose and non-infringement of intellectual property rights and ETSI shall not be held liable in any event for any damages whatsoever (including, without limitation, damages for loss of profits, business interruption, loss of information, or any other pecuniary loss) arising out of or related to the use of or inability to use the software.

Contents

Contents

Intellectual Property Rights

Essential patents

IPRs essential or potentially essential to normative deliverables may have been declared to ETSI. The declarations pertaining to these essential IPRs, if any, are publicly available for **ETSI members and non-members** , and can be found in ETSI SR 000 314: *"Intellectual Property Rights (IPRs); Essential, or potentially Essential, IPRs notified to ETSI in respect of ETSI standards"* , which is available from the ETSI Secretariat. Latest updates are available on the ETSI IPR online database.

Pursuant to the ETSI Directives including the ETSI IPR Policy, no investigation regarding the essentiality of IPRs, including IPR searches, has been carried out by ETSI. No guarantee can be given as to the existence of other IPRs not referenced in ETSI SR 000 314 (or the updates on the ETSI Web server) which are, or may be, or may become, essential to the present document.

Trademarks

The present document may include trademarks and/or tradenames which are asserted and/or registered by their owners. ETSI claims no ownership of these except for any which are indicated as being the property of ETSI, and conveys no right to use or reproduce any trademark and/or tradename. Mention of those trademarks in the present document does not constitute an endorsement by ETSI of products, services or organizations associated with those trademarks.

DECT™, **PLUGTESTS™**, **UMTS™** and the ETSI logo are trademarks of ETSI registered for the benefit of its Members. **3GPP™**, **LTE™** and **5G™** logo are trademarks of ETSI registered for the benefit of its Members and of the 3GPP Organizational Partners. **oneM2M™** logo is a trademark of ETSI registered for the benefit of its Members and of the oneM2M Partners. **GSM®** and the GSM logo are trademarks registered and owned by the GSM Association.

Foreword

This Group Report (GR) has been produced by ETSI Industry Specification Group <long ISGname> (<short ISGname>).

Modal verbs terminology

In the present document "**should**", "**should not**", "**may**", "**need not**", "**will**", "**will not**", "**can**" and "**cannot**" are to be interpreted as described in clause 3.2 of the ETSI Drafting Rules (Verbal forms for the expression of provisions).

"**must**" and "**must not**" are **NOT** allowed in ETSI deliverables except when used in direct citation.

Executive summary

Browsers represent one of the most complex and security-critical software products in modern computing, serving as the primary gateway between users and internet resources while processing untrusted content from millions of sources daily. The browser's architecture encompasses multiple interconnected subsystems—including rendering engines, JavaScript/WebAssembly execution environments, network stacks, and extension frameworks, each presenting distinct attack surfaces that shall be defended while maintaining performance, compatibility with legacy web content, and user autonomy.

Unlike traditional security products that can enforce restrictive controls, browsers shall balance protection against an evolving threat landscape with respect for user choice, creating unique challenges where users may deliberately choose to visit malicious sites, install risky extensions, or disable security features. The browser's multi-layered trust model, spanning from the highly privileged browser core through semi-trusted extensions to completely untrusted web content, requires sophisticated isolation mechanisms, granular permission systems, and careful mediation of system resource access.

Given browsers' ubiquitous deployment across consumer, enterprise, and specialized environments, their role as platforms for Progressive Web Applications, and their position as primary targets for nation-state and criminal actors, establishing proportionate security requirements under the Cyber Resilience Act demands careful consideration of the inherent tensions between security, functionality, performance, and user agency that define the modern web browsing experience.

Introduction

This European harmonised standard defines cybersecurity requirements applicable to browsers.

This document will provide security requirements and assessment criteria covering all elements defined in CRA Annex I Part 1 and Part 2 for stand alone browsers, as mentioned in CRA Annex III Class I important products. This work item intends to produce an EN as candidate for harmonisation, under the standardisation request in support of the implementation of the CRA (M/606).

1 Scope

This standard focuses on browsers, both standalone and embedded. Browsers are software products with digital elements that enable end users to access and interact with web content hosted on servers that are connected to local and remote networks.

Within the context of an operating system, browsers are user-applications with a primary function and probable daily use. They are often leveraged as means of accessing remote authentication (single-sign-on) or even as a bridge (deep-link) to another application that has already been installed. In both cases, all systems have the notion of a "default browser" that can then be instrumented by other applications to navigate to a website or perform such an activity.

The activity of browsing can be defined in the following steps:

1. A machine accesses remote resources and source code, such as HTML, JavaScript/WebAssembly, and CSS.
2. This source is represented visually, acoustically, or in some other form.
3. The user interacts with the rendered representation through input and output interfaces, including visual observation, text entry, pointer interaction, or other supported input modalities.

1.1 Browser

1.1.1 Standalone

Standalone browsers are applications that fulfil the functions of browsing.

Web browsers are software applications that access, retrieve, and interact with information and resources addressed by URLs. A standalone browser may be used for everyday tasks such as reading email, managing a calendar, or consuming the news.

Such programs commonly have interfaces for managing multiple websites, browsing history, bookmarks, user identities, passwords, and other settings.

They can commonly be extended with browser extensions, which are products with digital elements that have the ability to read, store, and modify the websites that users interact with.

1.1.2 Embedded

Embedded browsers are browsing services that are integrated into another system or application.

As such, they are programs using the same baseline technology of browsing but are commonly used for "single purpose" browsing. This means that instead of opening the user's preferred standalone browser, the hosting application will open an embedded browser to keep the user's attention. It is not common for a user to be able to change the configuration of an embedded browser.

1.1.3 Progressive Web Apps (PWA)

Progressive Web Apps are web applications that can be installed to a user's device from a standalone browser and subsequently operate in a dedicated application-like context.

PWAs leverage browser capabilities including service workers, application manifests, and isolated storage to provide offline functionality, push notifications, and integration with operating system features such as the application launcher and task switcher. When installed, they execute within the browser's process architecture but present themselves to the user as distinct applications with their own windows, icons, and settings.

Unlike traditional web pages, installed PWAs maintain separate configuration contexts from the main browser, including distinct storage partitions, permission grants, and display modes. They may register custom protocol handlers, manage their own cache strategies through service workers, and receive operating system events such as share targets or file handlers. Despite this application-like presentation, PWAs remain fundamentally web applications subject to the same security boundaries and web platform APIs as content rendered in standard browser tabs.

1.1.4 Browser Extensions

Browser extensions are third-party software components that integrate with and extend the functionality of standalone browsers.

Extensions operate with elevated privileges compared to standard web content, enabling them to intercept and modify network requests, inject scripts into web pages, access cross-origin resources, interact with browser APIs, and persist data across browsing sessions. They are distributed through vendor-operated extension stores or side-loaded through developer modes, and are subject to varying degrees of review, validation, and ongoing monitoring depending on the browser vendor's policies.

Unlike web applications that execute within the constraints of the same-origin policy, extensions declare their required permissions through manifest files and, once granted, operate with capabilities that span multiple origins and browser contexts. They may consist of background scripts or service workers for persistent logic, content scripts that execute within web page contexts, popup interfaces, options pages, and other components. The security model of extensions creates a unique trust boundary where extensions act as intermediaries between the browser core and web content, requiring careful permission management, isolation mechanisms, and code signing to prevent abuse while enabling legitimate functionality enhancements.

1.2 Derivative Browsers and Manufacturer Obligations

A significant proportion of browsers placed on the market are derivative products based on open source browser engines or substantially complete browser implementations. Understanding the manufacturer's obligations for such derivative products is essential to the proportionate application of this standard.

1.2.1 Open Source Browser Engines and Derivative Products

Open source browser projects such as Chromium, Gecko (Firefox), and WebKit provide complete or near-complete browser implementations that serve as the foundation for derivative products. These upstream projects are stewarded by organizations that maintain the core rendering engines, JavaScript execution environments, network stacks, and security architectures, but the projects themselves do not constitute products placed on the EU market with CE marking.

When an economic operator takes such an open source project, applies modifications (whether substantial or minor), and places the resulting browser on the market under their own brand or distribution channel, that operator becomes a manufacturer under the Cyber Resilience Act [i.1]. This classification applies regardless of the extent of modification—from minor branding and default configuration changes to substantial feature additions, custom user interfaces, or integration of proprietary services.

1.2.2 Spectrum of Derivative Modifications

Derivative browsers exist along a spectrum of modification, each with implications for conformity assessment:

Minor Modifications: Browsers that modify only branding elements, default search providers, homepage settings, bundled bookmarks, or visual themes while maintaining the upstream codebase's security architecture, update mechanisms, and core functionality. Examples include rebranded releases for specific markets or partnerships.

Configuration-Level Modifications: Browsers that alter default privacy settings, tracking protection levels, extension policies, or feature flags to differentiate the product while preserving the underlying implementation. Such modifications may strengthen or weaken security postures relative to the upstream project.

Feature Additions: Browsers that integrate additional capabilities such as built-in VPN services, cryptocurrency wallets, AI assistants, proprietary synchronization services, or vertical-specific toolbars. These additions create new attack surfaces and data processing considerations beyond those present in the upstream project.

Architectural Modifications: Browsers that modify process architecture, sandbox implementations, network request routing, certificate validation logic, or other security-critical components. Such changes may fundamentally alter the security properties inherited from the upstream project.

1.2.3 Manufacturer Responsibilities for Derivative Products

Economic operators placing derivative browsers on the market bear full manufacturer obligations under the CRA, regardless of their reliance on upstream security implementations. These obligations include:

Security Requirement Compliance: Demonstrating that the derivative product, in its modified form, satisfies the essential cybersecurity requirements of Annex I of the CRA. While manufacturers may rely on the security properties of unmodified upstream components, any modifications should be assessed for their impact on those security properties.

Vulnerability Management: Establishing processes to monitor both upstream security advisories and vulnerabilities specific to the manufacturer's modifications. Timely integration of upstream security patches is a critical manufacturer responsibility, as delays in patch integration extend the exposure window for known vulnerabilities affecting end users.

Conformity Assessment: Conducting or commissioning technical assessments that address both the inherited security properties from the upstream project and the security implications of the manufacturer's specific modifications. The assessment should consider whether modifications have weakened, maintained, or strengthened the security posture.

Technical Documentation: Maintaining documentation that clearly delineates which components are inherited from the upstream project versus manufacturer modifications, security reviews conducted on modifications, processes for integrating upstream updates, and any divergences from upstream security defaults.

Update Delivery: Ensuring that security updates reach end users in a timely manner. For derivative browsers, this includes both the integration of upstream security patches into the manufacturer's codebase and the delivery of updated builds to end users through the manufacturer's distribution and update infrastructure.

1.2.4 Trust in Upstream Security Implementations

Manufacturers of derivative browsers commonly rely on the security implementations provided by upstream projects for foundational requirements such as TLS protocol implementation, cryptographic library usage, certificate validation, same-origin policy enforcement, and sandbox architecture. This reliance is reasonable provided that:

Upstream Security Processes are Verifiable: The upstream project demonstrates transparent security practices including public vulnerability disclosure, security-focused development processes, regular security audits, and timely patch releases.

Modifications Do Not Undermine Upstream Security: The manufacturer's changes do not bypass, weaken, or interfere with the security mechanisms inherited from the upstream project. For example, modifications that disable certificate validation, weaken content security policies, or reduce sandbox restrictions would constitute substantial security regressions requiring additional justification and compensating controls.

Integration Timeliness is Maintained: The manufacturer maintains a process to integrate upstream security patches within a reasonable timeframe. Extended delays between upstream patch availability and manufacturer distribution create unnecessary risk exposure for end users.

Deviation Points are Documented and Assessed: Where the manufacturer intentionally diverges from upstream security defaults (e.g., enabling features disabled upstream for security reasons, or modifying cryptographic configurations), these deviations are documented with security rationale and risk assessment.

1.2.5 Application of This Standard to Derivative Browsers

When applying the requirements of this standard to derivative browsers, manufacturers and assessors should consider:

Inherited vs. Modified Components: Requirements addressing components that remain unmodified from the upstream project may be satisfied by demonstrating that the upstream implementation meets the requirement, provided the manufacturer's integration does not interfere with that implementation.

Modification-Specific Assessment: Requirements addressing areas where the manufacturer has made modifications require direct assessment of those modifications. This includes manufacturer-added features, modified defaults, integrated services, and any changes to security-critical code paths.

Update Mechanism Obligations: Even where a manufacturer relies on the upstream project's update mechanism architecture, the manufacturer remains responsible for ensuring that updates reach end users. This includes operating update servers, signing update packages, managing update channels, and ensuring update delivery reliability.

Use Case Alignment: Derivative browsers should be assessed against the use cases (Chapter 4.4) that align with their intended deployment contexts. A derivative browser marketed for general consumer use would align with UC-B1, while one marketed for enterprise deployment with proprietary features would align with UC-B7, regardless of their shared upstream heritage.

Derivative browsers represent a practical and economically significant category of products within the browser market. This standard recognizes that reliance on well-maintained upstream security implementations is a valid engineering approach, while maintaining that manufacturers placing derivative products on the market retain full responsibility for the security properties of the products they distribute.

1.2.6 State of the Art: Industry Testing and Security Practices

The state of the art for browser development and security validation encompasses organizational practices, industry standards, and comprehensive testing regimes that manufacturers should demonstrate to establish the quality and security of their browser implementations, whether original or derivative.

Organizational Practices and Resources:

Reputable browser manufacturers, both upstream projects and derivative product vendors, demonstrate their commitment to security through:

- **Adequate staffing**: Employment of sufficient numbers of developers and security personnel with expertise in browser architecture, web standards, cryptography, and vulnerability research
- **Security-focused development**: Dedicated security teams, secure development lifecycle practices, code review processes, and security architecture oversight
- **Transparency and communication**: Public disclosure of security policies, vulnerability handling procedures, and regular security bulletins
- **User commitment**: Published statements of commitment to user security and privacy, including privacy policies, data handling practices, and user control mechanisms
- **Update cadence**: Regular release schedules for security updates and patches, with clear timelines for critical vulnerability remediation

Industry Standards Compliance:

Browsers are expected to comply with applicable industry standards including but not limited to:

- **Web standards**: W3C specifications (HTML, CSS, JavaScript/ECMAScript, DOM, Fetch, etc.), WHATWG living standards
- **Security standards**: IETF RFCs for TLS, HTTP, WebAuthn, and related protocols; CA/Browser Forum Baseline Requirements
- **Accessibility standards**: WCAG (Web Content Accessibility Guidelines), ARIA (Accessible Rich Internet Applications)
- **Privacy standards**: Do Not Track, Global Privacy Control, tracking protection standards

Industry-Recognized Testing Frameworks:

Browsers should undergo testing using recognized industry test suites and frameworks:

Standards Conformance Tests: - **Web Platform Tests (WPT)**: Comprehensive cross-browser test suite maintained by W3C and browser vendors, with public dashboard available at https://wpt.fyi/ showing conformance across implementations - **Test262**: Official ECMAScript conformance test suite maintained by Ecma TC39, verifying JavaScript/ECMAScript specification compliance - **W3C Test Suites**: Individual test suites for specific W3C specifications (CSS Working Group tests, HTML5 tests, etc.) - **Acid Tests**: Historical but influential browser standards compliance tests (Acid1, Acid2, Acid3) developed by the Web Standards Project

Functional and Compatibility Testing: - **Selenium**: Open-source testing framework for browser automation, widely used for functional testing and regression testing across browsers - **BrowserStack**: Cloud-based cross-browser testing platform enabling compatibility verification across browser versions, operating systems, and devices - **Playwright/Puppeteer**: Modern browser automation and testing frameworks providing programmatic control for automated testing

Security-Specific Testing and Validation: - **CA/Browser Forum participation**: Engagement with the CA/Browser Forum (https://cabforum.org/working-groups/server/charter/) which establishes baseline requirements for certificate authorities and browser trust store policies - **TLS/Certificate validation testing**: Testing against standard certificate validation scenarios, revocation checking, and TLS protocol compliance - **Vulnerability disclosure programs**: Participation in responsible disclosure programs, bug bounty programs, and coordinated vulnerability disclosure processes - **Penetration testing**: Regular security assessments by internal security teams or external security researchers

Standards Body Participation:

Active participation in standards bodies demonstrates commitment to interoperability and security best practices:

- **W3C (World Wide Web Consortium)**: Participation in working groups defining web standards
- **WHATWG (Web Hypertext Application Technology Working Group)**: Collaboration on living standards for HTML, DOM, and related specifications

- **IETF (Internet Engineering Task Force)**: Engagement in protocol standardization (TLS, HTTP, WebRTC, etc.)
- **Ecma International**: Participation in ECMAScript (JavaScript) standardization through TC39
- **CA/Browser Forum**: Involvement in establishing requirements for certificate authorities and browser root programs
- **FIDO Alliance**: Participation in authentication standards development (WebAuthn, FIDO2)

Implications for Derivative Browsers:

Derivative browser manufacturers should demonstrate:

1. **Upstream testing inheritance**: Evidence that the upstream project undergoes comprehensive testing via WPT, Test262, and other industry-standard test suites, with results publicly available
2. **Modification testing**: Testing of manufacturer-specific modifications using appropriate test frameworks to ensure modifications do not introduce regressions or security vulnerabilities
3. **Integration testing**: Validation that the integration of upstream components with manufacturer additions maintains standards compliance and security properties
4. **Security review process**: Documentation of security review procedures for modifications, including code review, security testing, and vulnerability assessment
5. **Update testing**: Verification that upstream updates are tested before distribution to ensure compatibility with manufacturer modifications

Manufacturers may demonstrate compliance with industry testing practices by referencing: - Publicly available test results on wpt.fyi or similar dashboards - Participation in open-source testing efforts - Documentation of testing methodologies and results - Third-party security assessments or certifications - Membership in relevant standards bodies and working groups

The state of the art represents a comprehensive approach to browser quality and security, combining organizational commitment, standards compliance, extensive automated testing, security-focused practices, and community engagement. Derivative browser manufacturers should demonstrate that their products meet or exceed these industry norms, either through inheritance from well-maintained upstream projects or through their own testing and validation processes.

2 References

2.1 Normative references

In Harmonised Standards these references shall be specific *(identified by date of publication and/or edition number or version number)* ***publicly available and in English, except in exceptional circumstances making sure that impacts have been evaluated and explanations have been given on how any negative implications should be avoided*** *. See clauses 2.10.1 and 8.4 of the EDRs and the communiqué on "References in ETSI Deliverables".*

Guidance for selecting normative references in harmonised standards is given in clause 2.8.3 of the Vademecum on European standardisation. Please ***systematically consult with your Technical Officer*** *for the latest guidance on normative references other than to ENs, ISO/IEC standards, notably to prevent the risk of non-acceptance.*

Legal acts can never be used as normative references.

It is recommended that the number of references be limited to the minimum needed for the implementation/application of the ETSI Deliverables. References not directly concerned with the implementation/application/understanding of the ETSI Deliverable shall be listed in the Bibliography annex.

References are either specific (identified by date of publication and/or edition number or version number) or non-specific. For specific references, only the cited version applies. For non-specific references, the latest version of the referenced document (including any amendments) applies.

Referenced documents which are not found to be publicly available in the expected location might be found in the ETSI docbox.

> NOTE: While any hyperlinks included in this clause were valid at the time of publication, ETSI cannot guarantee their long-term validity.

The following referenced documents are necessary for the application of the present document.

- [1] <Standard Organization acronym> <document number> (<version number>): "<Title>".

2.2 Informative references

References are either specific (identified by date of publication and/or edition number or version number) or nonspecific. For specific references, only the cited version applies. For non-specific references, the latest version of the referenced document (including any amendments) applies.

> NOTE: While any hyperlinks included in this clause were valid at the time of publication, ETSI cannot guarantee their long-term validity.

The following referenced documents may be useful in implementing an ETSI deliverable or add to the reader's understanding but are not required for conformance to the present document.

- [i.1] Regulation (EU) 2024/2847 of the European Parliament and of the Council of 23 October 2024 on horizontal cybersecurity requirements for products with digital elements and amending Regulations (EU) No 168/2013 and (EU) 2019/1020 and Directive (EU) 2020/1828 (Cyber Resilience Act).
- [i.2] NIST SP 800-128 (2011) Guide for Security-Focused Configuration Management of Information Systems
- [i.x] <Standard Organization acronym> <document number> (<version number>): "<Title>".

3 Definition of terms, symbols and abbreviations

3.1 Terms

The terms below are important for understanding the purpose and usage of browsers. For the purposes of the present document, the following terms apply:

Term	Definition
Access	The capability to retrieve, load, and display web content from servers through network protocols, including establishing connections, downloading resources, and rendering content for user consumption.
Accessing Web Content	The complete process by which browsers retrieve, process, and present web resources to end users, encompassing network communication, content parsing, rendering, and user interface presentation.
Browser Extensions	Software modules that augment browser functionality by adding features, modifying behavior, or enhancing user experience beyond the browser's core capabilities, typically installed and managed through the browser's extension system.
Browsers	Software products with digital elements that enable end users to access and interact with web content hosted on servers that are connected to local and remote networks. *Note: Expert group definition - In the context of this category of products, browsers are software products with digital elements that enable end users to access and interact with web content hosted on servers that are connected to networks such as the Internet.*

Term	Definition
Custom Protocol	Non-standard or application-specific communication protocols that browsers may support for specialized content access or functionality, extending beyond traditional web protocols like HTTP/HTTPS.
Embedded Browsers	Browsers that are intended for integration into another system or application.
End Users	Natural persons who utilize browsers to access web content for personal, professional, or other purposes, including but not limited to browsing, reading, viewing multimedia content, and interacting with web applications.
Interact	The critical activity that defines browsing, encompassing user actions such as clicking hyperlinks, submitting forms, executing scripts, manipulating page elements, and engaging with dynamic web content through input devices.
Networks	Communication infrastructures that enable data transmission between browsers and servers, encompassing local area networks (LANs), wide area networks (WANs), and the global Internet.
Progressive Web Applications	Web-based applications that operate within the browser environment, leveraging advanced browser APIs and capabilities to provide enhanced functionality including offline operation, background synchronization, push notifications, and device hardware access, while remaining fundamentally dependent on the browser's runtime and security model for execution and user interaction.
Raw Content	Unprocessed source code and data formats delivered by servers, including but not limited to XML, JSON, JavaScript, HTML, CSS, and other markup or programming languages before browser interpretation.
Servers	Computer systems or software applications that store, process, and deliver web content to browsers via network protocols, responding to browser requests with appropriate resources and data.
Standalone Browsers	Standalone applications that fulfil the functions of browsers.
Web Content	The displayed and rendered representation of raw content, transformed by browsers into human-perceivable formats including text, images, videos, interactive elements, and structured layouts as intended by content creators.

3.2 Symbols

For the purposes of the present document, the [following] symbols [given in ... and the following] apply:

[to be added]

4 Product Context

4.1 General

4.2 Out of scope use/environments

List uses/environments covered by other legislation or standards (critical, industrial, medical, etc.). Hoping to have a reusable generic list of these soon.

The types of product with digital elements listed in the section do not fall within the scope of the Cyber Resilience Act [i.1], and are not covered by this standard:

1. Services, except for the remote data processing solutions for a covered product as defined in CRA recitals 11-12; article 3, 2 [i.1];
2. Products specifically designed or procured for national security and defence purpose as defined in CRA recitals 14 and 26; article 2, 7-8 [i.1];
3. Products developed for or used exclusively for internal use by public administration as defined in CRA recital 16; article 5, 2 [i.1];
4. Non-commercial free and open source software as defined in CRA recitals 17-21; article 13, 5 [i.1];
5. Medical Devices and Software as defined in CRA recital 25; article 2, 2 [a-b] [i.1];
6. Vehicles, including aviation and marine equipment as defined in CRA recital 27; article 2, 2.c "vehicles"; recital 27; article 2, 3 "aviation"; article 2, 4 "marine equipment" [i.1];
7. Spare and used parts as defined in CRA recital 29; article 2, 6 [i.1];
8. Refurbished, repaired, and upgraded products that have not been substantially modifiedas defined in recitals 39 - 42 [i.1];

The following types of products have reduced or varied requirements under the Cyber Resilience Act [i.1] and can only be partially covered by this standard.

1. High Risk AI as defined in CRA recital 51; article 12 [i.1];
2. Testing and unfinished versions as defined in recital 37; Article 4, 2-3 [i.1];
3. Products Placed on the Market Prior to December 11, 2027 as defined in CRA article 69 [i.1].

4.3 In-Scope Components

4.3.1 In-Scope components standalone browser

For the purposes of this standard, a standalone browser consists of the following in-scope security-relevant components:

Core Browser Components:

1. **Rendering Engine**: HTML parser, CSS engine, layout system, and DOM implementation responsible for processing and displaying web content.
2. **JavaScript Engine**: JavaScript runtime, JIT compiler, garbage collector, and execution context management providing the environment for web application code execution.
3. **Network Stack**: HTTP/HTTPS client implementation, certificate validation, connection management, caching subsystem, and protocol handlers (WebSocket, WebRTC, etc.).
4. **Process Architecture**: Multi-process isolation model including browser process, renderer processes, GPU process, network process, and inter-process communication (IPC) mechanisms.
5. **Storage Subsystem**: Cookie management, localStorage, sessionStorage, IndexedDB, Cache API, origin-partitioned storage, and persistent storage quota management.
6. **Permission System**: Runtime permission prompts, permission state management, permission policy enforcement, and user consent UI for sensitive capabilities (camera, microphone, location, notifications, etc.).
7. **Sandbox Implementation**: Operating system-level process sandboxing, seccomp/AppContainer restrictions, filesystem access controls, and system call filtering.
8. **Security Policy Engines**: Same-Origin Policy enforcement, Cross-Origin Resource Sharing (CORS) validation, Content Security Policy (CSP) parser and enforcer, and Mixed Content blocking.

Extension System Components (if present):

9. **Extension Runtime**: Extension process management, manifest validation, permission enforcement for extension APIs, and content script injection mechanism.
10. **Extension API Layer**: Browser APIs exposed to extensions (webRequest, tabs, storage, etc.), permission-based access controls, and extension-to-browser IPC.

Update and Maintenance Components:

11. **Update System**: Automatic update mechanism, update signature verification, update rollback capability, and background update process.
12. **Diagnostic and Telemetry**: Crash reporting, error logging, usage metrics collection (where implemented with user consent), and debug logging infrastructure.

User Interface Components:

13. **Security Indicators**: HTTPS lock icon, certificate viewer, permission indicators, malicious site warnings, and phishing/malware protection UI.
14. **User Consent UI**: Permission prompts, download confirmations, external protocol handler registration prompts, and security warnings.

Certificate and Trust Components:

15. **Certificate Management**: Root certificate store, certificate validation logic, OCSP/CRL checking, Certificate Transparency verification, and certificate pinning.
16. **Trust Decisions**: Safe Browsing integration, malicious site detection, phishing protection, download scanning integration, and security warnings.

Out-of-Scope Components:

The following components are explicitly excluded from the security requirements of this standard:

- Third-party websites and web applications accessed through the browser
- Server-side infrastructure operated by the browser manufacturer (sync services, account systems) except where they deliver security-critical updates
- Operating system components and system libraries not distributed as part of the browser package
- Third-party extensions and plugins developed outside the browser manufacturer's control
- User-generated bookmarks, preferences, and configuration data
- Remote attestation or DRM modules that operate under separate security models

4.3.2 In-Scope components embedded browser

For the purposes of this standard, an embedded browser (WebView component or integrated browser engine) consists of the following in-scope security-relevant components in addition to or as variations of the standalone browser components listed in 4.3.1:

Embedded Browser Core Components:

1. **WebView Engine**: The embedded rendering engine (e.g., Android WebView, iOS WKWebView, Electron, CEF, WebView2) including HTML/CSS/JavaScript processing capabilities adapted for host application integration.
2. **Host Application Boundary**: Security boundary enforcement between web content running in the WebView and the native host application code, including context isolation and privilege separation.
3. **JavaScript Bridge**: Native-to-web and web-to-native communication interface allowing controlled interaction between JavaScript running in web content and native application APIs, including message passing and function exposure mechanisms.
4. **Custom URL Scheme Handlers**: Registration and handling of application-specific URL schemes (e.g., app://, custom-protocol://) that trigger native code execution or data retrieval from web content.
5. **WebView Configuration API**: Host application APIs for configuring WebView security properties (JavaScript enablement, file access permissions, content security settings, network access controls).

Embedded-Specific Security Components:

6. **Content Source Policy**: Allowlisting and trust management for content sources loaded into the WebView (local files, remote URLs, bundled assets), including validation of content origin and integrity.
7. **Storage Isolation**: Separation of WebView storage (cookies, localStorage, IndexedDB) from both the host application's native storage and from other WebView instances or standalone browsers on the same device.
8. **Permission Delegation**: Mechanism for handling web API permission requests (camera, microphone, location, etc.) where the WebView delegates permission decisions to the host application.
9. **Navigation Controls**: Host application controls over WebView navigation including URL allowlisting, navigation interception, redirect validation, and prevention of unintended navigation to external sites.
10. **Script Injection Controls**: Security mechanisms governing the injection of JavaScript into web content by the host application, including timing, isolation, and privilege level of injected scripts.

Host Integration Components:

11. **Native API Exposure Management**: Controlled exposure of native device capabilities (filesystem, camera, sensors, biometrics, secure storage) to web content through the JavaScript bridge with appropriate authorization and validation.
12. **Data Sharing Controls**: Mechanisms for secure data transfer between web content and native application context, including input validation, output encoding, and data sanitization at the boundary.
13. **Event Handling**: WebView event system for navigation events, load events, error events, and security-relevant events (certificate errors, SSL warnings, mixed content detection) with propagation to host application.
14. **Native UI Integration**: Security considerations for WebView rendering within native UI containers, including overlay protection, screenshot prevention for sensitive content, and secure display of trust indicators.

Update and Maintenance Components (Embedded-Specific):

15. **WebView Update Mechanism**: System for updating the embedded browser engine independently of the host application (platform-provided WebView updates) or bundled with application updates (Electron, CEF).
16. **Compatibility Validation**: Testing and validation that WebView security properties are maintained across engine updates and remain compatible with host application security requirements.

Additional Embedded Browser Considerations:

17. **Process Architecture**: Whether the WebView runs in-process with the host application or in a separate process, and the IPC mechanisms used for communication if separated.
18. **TLS/Certificate Management**: Handling of TLS connections including whether the WebView uses system certificate stores or custom certificate validation, and integration with certificate pinning.
19. **Debugging Interfaces**: Security controls around WebView debugging capabilities (Chrome DevTools, Safari Web Inspector) to prevent unauthorized debugging of production applications.

Differences from Standalone Browsers:

Unlike standalone browsers where all web content is considered untrusted, embedded browsers shall:

- Establish selective trust relationships with certain content sources (bundled HTML, local files, specific remote origins) while maintaining security boundaries
- Mediate permissions through the host application rather than direct user prompts
- Share or isolate storage and state management with the host application
- Navigate the security boundary between web content privileges and native application privileges
- Consider the combined attack surface of both the WebView engine and the host application

Out-of-Scope Components for Embedded Browsers:

The following components are explicitly excluded from the security requirements for embedded browsers:

- Host application code outside the WebView integration points
- Server-side infrastructure serving content to the WebView (unless operated by the WebView engine provider for security updates)
- Third-party web content loaded into the WebView beyond the control of the host application developer
- Platform WebView implementations provided by the operating system (Android System WebView, iOS WKWebView) where the host application developer has no control over the engine implementation
- Native dependencies and libraries used by the host application but not related to WebView functionality

4.4 Use Cases

This clause defines representative use cases that illustrate the diverse operational contexts in which browsers and embedded browser components are deployed. These use cases serve multiple purposes within the conformity assessment framework:

1. **Risk Contextualization**: Each use case is associated with a risk level (Standard, High, or Critical) that reflects the potential impact of security failures in that deployment context, derived from the risk assessment methodology detailed in Annex A.
2. **Requirement Selection Guidance**: The use cases inform the selection of appropriate security capability levels from Chapter 5, helping manufacturers determine which condition levels are suitable for their intended deployment contexts.
3. **Proportionality Principle**: In accordance with the CRA's proportionality principle, these use cases demonstrate how security requirements scale with risk, ensuring that security controls are commensurate with the threats and impacts relevant to each deployment scenario.
4. **Shared Understanding**: The use cases provide a common vocabulary for manufacturers, assessors, and regulators to discuss browser security requirements in relation to real-world deployment contexts.

Scope and Applicability:

The use cases defined in this clause encompass both standalone browsers and embedded browser components (WebViews). While the fundamental security capabilities defined in Chapter 5 apply to both categories, the specific threats, deployment environments, and risk profiles differ:

- **Standalone browsers** (UC-B1 through UC-B8) operate as independent applications with direct user interaction and comprehensive security controls managed by the browser itself.
- **Embedded browsers** are represented in UC-B10 and in aspects of other use cases where browser engines are integrated into host applications, requiring consideration of host-web security boundaries and trust relationships.
- **Progressive Web Applications (PWAs)**, while representing an important deployment model, inherit their security properties from the underlying browser (standalone or embedded) in which they execute, and thus are covered by the applicable use case for that browser deployment.

Use Case Structure:

Each use case provides:

- **Description**: The primary purpose and scope of the deployment
- **Typical workflows**: Common user interactions and usage patterns
- **Typical environments**: Physical and network contexts in which the browser operates
- **Security considerations**: Key threats, vulnerabilities, and security controls relevant to the use case
- **Risk level**: Overall risk classification (Standard, High, or Critical)
- **Rationale**: Justification for the assigned risk level based on threat landscape and potential impact

Risk Levels Explained:

- **Standard Risk**: General-purpose deployments where security failures primarily affect individual users, with limited financial or societal impact. Standard security capabilities are appropriate.
- **High Risk**: Deployments involving sensitive personal data, financial transactions, health information, organizational data, or authenticated access to critical services. Enhanced security capabilities and stricter condition levels are recommended.
- **Critical Risk**: Deployments where security failures could result in significant physical harm, disruption of essential services, compromise of critical infrastructure, or large-scale societal impact. Maximum security capabilities with the strictest condition levels are required.

4.4.1 Application to Conformity Assessment

Manufacturers shall use these use cases as follows:

1. **Identify Applicable Use Cases**: Determine which use case(s) best represent the intended purpose and deployment context of the browser or embedded browser component.
2. **Assess Risk Level**: Evaluate whether the assigned risk level (Standard, High, Critical) aligns with the manufacturer's risk assessment for their specific deployment. Where multiple use cases apply, the highest applicable risk level should be considered.
3. **Select Security Capabilities**: Use Annex B to identify the recommended security capability condition levels for the applicable use case(s). These recommendations represent typical configurations; manufacturers may select stricter conditions based on their risk assessment or specific deployment requirements.
4. **Document Rationale**: In conformity assessment documentation, manufacturers should clearly identify which use case(s) apply to their product and provide justification for any deviations from recommended capability levels.
5. **Consider Use Case Combinations**: Many deployments will span multiple use cases (e.g., a browser used for both general web browsing and enterprise applications). In such cases, manufacturers should satisfy requirements for all applicable use cases or implement use-case-specific profiles.

Note: These use cases are representative and not exhaustive. Manufacturers deploying browsers in contexts not explicitly covered by these use cases should conduct a detailed risk assessment per Annex A and select security capability levels appropriate to the identified risks.

4.4.2 Use Cases for Browsers

UC-B1: General Purpose Web Browsing

- Description: Browsing of public websites for news, social media, entertainment, shopping, streaming, and general information. Excludes authenticated access to sensitive personal or organizational systems.
- Typical workflows: High tab count with frequent context switching; passive consumption (reading/watching); form fills; long idle sessions.
- Typical environments: Personal devices used in homes, cafes, transit, or public spaces. No physical security controls; high exposure to shoulder surfing, untrusted networks, or device theft.
- Security considerations: Tracking protection; HTTPS-only mode; blocking of malicious sites; access to camera/microphone/location; exposure to browser extensions; integrated password management; profile separation.
- Risk level: Standard
- Rationale: Primary threats originate from web content (malware, trackers, phishing) rather than targeted compromise.

UC-B2: Development and Testing Environments

- Description: Browser usage by developers, QA, and testers for building, debugging, and validating web applications, including compatibility testing with pre-release (canary/nightly/beta) browser builds.

- Typical workflows: Frequent navigation to localhost/internal IPs; manual and automated interaction with untrusted or malformed code; usage of developer tools; HAR/network capture; testing auth flows.
- Typical environments: Developer workstations in office or home offices; may be BYOD or corporate-managed; moderate physical security.
- Security considerations: Isolation using ephemeral profiles or web browser private mode, allowlists for extensions, supply chain auditing, curation and anonymization of test data.
- Risk level: High
- Rationale: Exposure to untrusted code, experimental browser features, and misconfiguration increases likelihood of exploit execution or leakage of credentials.

UC-B3: Kiosks and Shared Terminals

- Description: Multi-user public or semi-public devices for check-in, customer service, library access, clinic intake, or retail assistance.
- Typical workflows: Short, single-purpose sessions (check-in, lookup, form submission); no authentication or credential entry.
- Typical environments: Fixed-location terminals in lobbies, libraries, clinics, classrooms or stores.
- Security considerations: Strict domain allowlist; no access to camera/mic/location/notifications; block credential saving and autofill; remote wipe and health monitoring; encryption of cached assets.
- Risk level: High
- Rationale: High turnover of untrusted users combined with potential for physical tampering.

UC-B4: Financial Services Access

- Description: Access to online banking, brokerage, payments, crypto exchanges, wallets, and insurance portals involving monetary transactions or sensitive financial data.
- Typical workflows: Daily logins; balance checks; initiating transfers/payments; uploading documents.
- Typical environments: Primarily personal devices in uncontrolled locations; corporate devices in secure facilities
- Security considerations: HTTPS-only mode; credential monitoring for breaches.
- Risk level: High
- Rationale: Exposure to financial fraud and account takeover; session hijacking; MITM;

UC-B5: Healthcare and Medical Systems

- Description: Browser access to EHRs, telemedicine platforms, patient portals, prescription systems or health insurance.
- Typical workflows: Patient record access/modification; remote consultations; e-prescribing; uploading/downloading diagnostic files.
- Typical environments: #TODO: Similar to home + hospital workstations?
- Security considerations: Session re-auth for sensitive actions, auto-timeout after inactivity, data encryption at rest/in transit.
- Risk level: High
- Rationale: Regulatory penalties, reputational damage, and potential patient safety risks.

UC-B6: E-Government Services Access

- Description: Access to citizen-facing or administrative government portals for taxes, benefits, licenses, identity verification, legal filings, or voting systems.
- Typical workflows: Infrequent but critical sessions, form filling, document uploads, digital signature application.
- Typical environments: #TODO: Personal devices or public terminals.
- Security considerations: Strong authentication; digital signature validation; certificate management.
- Risk level: High
- Rationale: Compromise can lead to identity theft, benefit fraud, election interference, or erosion of civic trust.

UC-B7: Enterprise Applications

- Description: Internal browser-based tools for CRM, ERP, HRMS, document collaboration, project management and BI,
- Typical workflows: Daily CRUD operations of records, document management,
- Typical environments: Corporate laptops, desktops, or BYOD devices used remotely or on-premise.
- Security considerations: SSO, DLP (control copy/paste/print/download actions), allowlist for extensions, integration with SIEM, containerization of BYOD.
- Risk level: High
- Rationale: Exposure of intellectual property, customer data, or internal operations to exfiltration or insider threat.

UC-B8: Critical Infrastructure

- Description: Web interfaces for SCADA, energy grid, water treatment, transportation control, or emergency dispatch.
- Typical workflows: Administered access, scoped to individual user accounts.
- Typical environments: Secure control rooms, data centers, or field stations - physically access-controlled, often air-gapped or operating within segmented network.
- Security considerations: Certificate management; zero trust architecture; mTLS; RBAC; supply chain controls; immutable logging.
- Risk level: Critical
- Rationale: Successful attack could disrupt essential services, cause physical damage, or endanger human life.

UC-B9: Security Research #TODO: Perhaps this one is too much

- Description: Intentional browsing to analyze phishing pages, malware or malicious extensions.
- Typical workflows: Automated or manual navigation to malicious URLs; downloading/executing payloads in sandbox; DOM inspection; screenshot/HAR capture; behavioral logging.
- Typical environments: Dedicated lab machines or air-gapped workstations, often behind shielded network zones.
- Security considerations: Isolation using disposable VMs, capture of all artifacts and network traffic;
- Risk level: Critical
- Rationale: Deliberate exposure to live threats; failure can lead to host compromise, lateral movement, or data exfiltration.

UC-B10: Adapted Browser with Modified Features

- Description: A manufacturer creates a product based on an existing open-source browser (e.g., Chromium, Firefox) by adding custom features, modifying default configurations, integrating proprietary services, or tailoring the browser for specific market segments, enterprise deployments, or regional requirements.
- Typical workflows: Standard web browsing workflows similar to general-purpose browsers, but with manufacturer-specific features such as custom home pages, integrated search providers, proprietary sync services, enhanced privacy controls, vertical-specific toolbars, or pre-configured extension bundles.
- Typical environments: All environments applicable to the underlying browser engine (personal devices, corporate workstations, mobile devices, kiosks), but with manufacturer-controlled default configurations and update channels.
- Security considerations: Inheritance of upstream browser vulnerabilities; security review of added features and modifications; validation that customizations do not weaken existing security controls; secure management of manufacturer-operated services (sync, accounts, analytics); timely integration of upstream security patches; transparency regarding data collection by added features; supply chain security for bundled extensions or services; verification that modifications maintain sandboxing and isolation properties; compliance with baseline browser security requirements while accounting for manufacturer additions.
- Risk level: Standard to High (depends on extent of modifications and deployment context)
- Rationale: The security posture depends on both the upstream browser's security and the manufacturer's implementation quality. Added features introduce additional attack surface and potential vulnerabilities. Delayed or incomplete integration of upstream patches can extend exposure to known vulnerabilities.

Manufacturer-operated services create additional trust dependencies and data processing considerations. However, when properly implemented, adapted browsers can maintain equivalent security to their upstream base while providing differentiated user value. The risk level increases when modifications are extensive, when manufacturer services handle sensitive data, or when the browser is deployed in high-risk contexts (UC-B4 through UC-B8).

4.5 Product overview and architecture

4.5.1 Product Definition

A standalone browser is a software application that enables users to access, retrieve, and interact with content on the World Wide Web and other network resources. Unlike embedded browsers or WebView components integrated into other applications, standalone browsers operate as independent applications with direct user interfaces, comprehensive feature sets, and autonomous update mechanisms.

4.5.2 Architectural Overview

4.5.2.1 Core Architecture Components

The browser architecture consists of several interconnected subsystems:

Rendering Engine: Processes HTML, CSS, and JavaScript to display web content. This includes the DOM parser, CSS engine, and layout system that transforms web resources into visual representations.

JavaScript Engine: Executes JavaScript code in isolated contexts, providing the runtime environment for dynamic web applications while maintaining security boundaries between different execution contexts.

Network Stack: Manages all network communications including HTTP/HTTPS requests, WebSocket connections, and other protocols. Implements connection pooling, caching, and security features such as certificate validation.

Process Architecture: Modern browsers employ multi-process architectures where the main browser process is separated from renderer processes, plugin processes, and GPU processes. This isolation prevents compromise of one process from affecting others.

Storage Subsystem: Manages various forms of local data storage including cookies, localStorage, IndexedDB, and cache storage, each with distinct security properties and access controls.

4.5.2.2 Security Architecture

Sandbox Model: Web content executes within restricted sandboxes that limit access to system resources. The sandbox is enforced at multiple levels - process isolation at the OS level, and API restrictions at the browser level.

Permission System: Browsers mediate access to sensitive capabilities (camera, microphone, location, notifications) through a permission system that requires explicit user consent and provides ongoing usage indicators.

Content Security Boundaries: The Same-Origin Policy (SOP) forms the fundamental security boundary, ensuring that content from one origin cannot access resources from another origin without explicit permission.

Certificate and Trust Management: Browsers maintain root certificate stores and implement certificate validation, including mechanisms for certificate pinning, HSTS (HTTP Strict Transport Security), and Certificate Transparency.

4.5.2.3 Extension Architecture

Browser extensions operate in a unique position within the security model:

- **Privileged Execution Context**: Extensions run with elevated privileges compared to web content, able to intercept and modify network requests, access cross-origin resources, and interact with browser APIs.
- **Manifest-Based Permissions**: Extensions declare required permissions in manifests, which are presented to users during installation. However, the granularity and understandability of these permissions remain challenges.
- **Content Script Injection**: Extensions can inject scripts into web pages, creating a three-way trust relationship between the browser, the extension, and the web content.

4.5.3 Trust Boundaries and Threat Model

4.5.3.1 Trust Zones

1. **Browser Core** (Highest Trust): The browser executable and core libraries, typically signed and verified by the operating system.
2. **Browser Extensions** (Elevated Trust): Third-party code with significant privileges, operating between web content and browser core.
3. **Web Applications** (Limited Trust): JavaScript applications running within the browser sandbox with restricted capabilities.
4. **Web Content** (Untrusted): General web content including advertisements, user-generated content, and third-party resources.
5. **Network** (Untrusted): All network communications are considered potentially hostile, requiring encryption and validation.

4.5.3.2 Attack Surface

The browser presents multiple attack surfaces:

- **Network Interface**: Exposed to malicious servers, man-in-the-middle attacks, and protocol-level exploits
- **Rendering Engine**: Subject to parsing vulnerabilities, memory corruption, and logic bugs
- **JavaScript Engine**: Target for JIT compiler bugs, type confusion, and sandbox escapes
- **Extension APIs**: Abused by malicious extensions or compromised legitimate extensions
- **User Interface**: Social engineering through spoofing, phishing, and permission fatigue
- **Local Storage**: Cross-site tracking, data exfiltration, and persistence mechanisms

4.5.4 Deployment Contexts

4.5.4.1 Consumer Environment

Individual users installing browsers on personal devices, typically with:

- Direct internet connectivity
- Mixed personal and professional usage
- User-managed security decisions
- Varied technical expertise levels

4.5.4.2 Enterprise Environment

Organizational deployments requiring:

- Centralized policy management
- Compliance with regulatory requirements
- Integration with security infrastructure (proxies, SIEM systems)

- Controlled update cycles
- Restriction of certain features or extensions

4.5.4.3 Specialized Environments

- **Kiosk Mode**: Public-facing browsers with locked-down configurations
- **Development Mode**: Browsers with relaxed security for testing
- **Privacy-Focused**: Configurations emphasizing anonymity and tracking prevention
- **Isolated Browsing**: Air-gapped or heavily restricted network access

4.5.5 Security-Relevant Characteristics

4.5.5.1 Dynamic Threat Landscape

Browsers face constantly evolving threats as new web standards introduce new capabilities, zero-day vulnerabilities are discovered, and attack techniques advance. The browser serves as the primary interface between users and potentially hostile internet content.

4.5.5.2 Compatibility Requirements

Browsers shall maintain backward compatibility with legacy web content while implementing new security features, creating tension between security and functionality. This includes supporting older TLS versions, maintaining compatibility with enterprise applications, and handling non-standard web content. Should the usage of legacy web content impact the security of the browser, then the content shall be ignored.

4.5.5.3 Performance Constraints

Security measures shall be balanced against performance requirements. Users expect instantaneous page loads and smooth interactions, limiting the computational overhead available for security checks. This affects decisions around sandboxing granularity, encryption methods, and validation procedures.

4.5.5.4 User Agency and Autonomy

Unlike many security products, browsers shall respect user choice while protecting against threats. Users may choose to:

- Visit dangerous websites despite warnings
- Install risky extensions
- Disable security features for compatibility
- Share sensitive information voluntarily

This requirement for user agency fundamentally shapes the browser security model, requiring informed consent mechanisms rather than purely restrictive controls.

4.6 Essential functions

The essential functions of a browser, as defined for the purposes of this standard, are those capabilities that shall remain operational to fulfill the browser's primary purpose of enabling secure access to web content and services. These functions form the baseline against which security requirements are assessed.

This section addresses both **standalone browsers** (independent applications with direct user interfaces) and **embedded browsers** (browser engines integrated into host applications, commonly referred to as WebViews, browser components, or embedded web rendering engines).

4.6.1 Core Essential Functions

Content Retrieval and Rendering: The browser shall retrieve web resources via network protocols (primarily HTTP/HTTPS) and render them into a visual and interactive presentation for the user. This includes:

- HTML parsing and DOM construction
- CSS styling and layout computation
- JavaScript and WebAssembly execution within secure sandboxes
- Rendering of images, media, and other embedded content

For embedded browsers: The host application may provide additional constraints on content sources, implement content filtering, or restrict certain rendering features. However, the fundamental capability to securely parse and render web content remains essential. See reference: WebView Security Best Practices.

Navigation and Session Management: The browser shall enable navigation between web resources and maintain session state including navigation history and form data.

For standalone browsers: This includes managing multiple concurrent browsing contexts (tabs, windows) with independent session states.

For embedded browsers: Navigation may be constrained by the host application to approved domains or URL patterns. The host application is responsible for implementing navigation controls and may delegate or restrict back/forward navigation. Session management may be simplified or controlled by the host application. See reference: Android WebView Security.

Cryptographic Communication: The browser shall establish encrypted connections to web servers using TLS/SSL protocols (TLS 1.2 minimum, TLS 1.3 recommended), validate server certificates, and provide indicators of connection security status.

For standalone browsers: Visual security indicators (padlock icons, address bar coloring) shall be prominently displayed in the user interface.

For embedded browsers: Security indicators may be delegated to the host application's UI. The host application shall be responsible for surfacing certificate validation status to end users when handling sensitive data. Certificate pinning may be implemented at the host application level. See references: Certificate Pinning, TLS Best Practices.

Data Storage: The browser shall provide controlled mechanisms for web applications to store data locally, including cookies, localStorage, IndexedDB, and cache storage, subject to security policies and user controls.

For embedded browsers: Storage isolation from the host application is critical. The embedded browser shall maintain separate storage contexts that are not directly accessible to host application code without explicit bridging. Storage may be partitioned per embedded browser instance to prevent data leakage between different uses within the same application. See reference: WebView Data Isolation.

User Input and Interaction: The browser shall accept and process user inputs (keyboard, mouse, touch) and deliver them securely to web content, while protecting against input injection attacks and unauthorized access to input devices.

For embedded browsers: Input handling shall prevent the host application from injecting synthetic events that could bypass user consent (e.g., programmatically triggering clicks on permission prompts). The boundary between host-provided input and user-generated input shall be clearly maintained. See reference: Input Validation in WebViews.

4.6.2 Security-Related Essential Functions

Isolation and Sandboxing: The browser shall maintain security boundaries between different origins, processes, and execution contexts to prevent unauthorized access and limit the impact of compromised content.

For embedded browsers: A critical additional boundary exists between the embedded browser content and the host application. This boundary shall prevent:

- Web content from accessing host application memory, files, or system resources without explicit bridging
- Web content from executing arbitrary code in the host application context
- Host application code from arbitrarily accessing or manipulating web content internal state

The host application shall implement a secure bridge or message-passing interface for controlled communication between web content and native code. See references: WebView Isolation, iOS WKWebView Security.

Permission Management: The browser shall mediate access to sensitive capabilities (camera, microphone, location, notifications, clipboard) through a permission system that requires user consent and provides ongoing visibility.

For embedded browsers: Permission requests may be delegated to the host application, which is then responsible for obtaining appropriate user consent and respecting platform-level permission grants. The host application shall not automatically grant permissions to embedded web content without user awareness. Permissions granted to the host application shall not automatically extend to all web content loaded in embedded browsers. See reference: WebView Permissions Model.

Update Mechanism: The browser shall maintain the capability to receive, verify, and apply security updates to address vulnerabilities and maintain security effectiveness over the product lifecycle.

For standalone browsers: Direct update mechanisms with user notification and control.

For embedded browsers: Updates are typically delivered as part of operating system updates (for system WebViews) or application updates (for bundled browser engines). The host application developer is responsible for:

- Using up-to-date versions of embedded browser components
- Monitoring security advisories for the embedded browser engine
- Deploying application updates that include patched browser components
- Not pinning to outdated versions of browser engines with known vulnerabilities

See references: Android System WebView Updates, Electron Security Updates.

Certificate Validation: The browser shall validate server certificates against trusted root certificate authorities and enforce certificate transparency requirements to detect mis-issued certificates.

For embedded browsers: Certificate validation shall use the platform's trusted certificate store unless explicit certificate pinning is implemented by the host application. The host application shall not disable certificate validation except in clearly documented development/testing scenarios that are disabled in production builds. See reference: Certificate Validation Best Practices.

4.6.3 Embedded Browser-Specific Security Functions

JavaScript Bridge Security: For embedded browsers that expose native functionality to web content through JavaScript bridges (e.g., `addJavascriptInterface` in Android WebView, `WKScriptMessageHandler` in iOS), the following are essential:

- **Input Validation**: All data passed from web content to native code shall be validated and sanitized to prevent injection attacks
- **Output Encoding**: Data passed from native code to web content shall be properly encoded to prevent XSS
- **Minimal Exposed Surface**: Only necessary functionality shall be exposed; avoid exposing powerful or sensitive APIs
- **Authentication**: Bridge calls that perform sensitive operations shall verify origin and intent
- **Intent Validation**: For URL schemes and intent handlers, validate and sanitize all parameters

See references: Android WebView JavaScript Interface Security, iOS JavaScript Bridge Security.

Content Source Validation: Embedded browsers shall validate and restrict content sources:

- **Allowlist Enforcement**: Maintain allowlists of approved domains/URLs when loading sensitive functionality
- **Local Content Handling**: When loading local HTML/JS resources, ensure they cannot be overwritten or manipulated by untrusted code
- **Deep Link Validation**: Validate and sanitize deep links that cause content to load in embedded browsers

See reference: Mobile Application Security Testing Guide - WebViews.

Host Application Data Exposure Prevention: The embedded browser shall prevent web content from accessing host application data through:

- **File System Isolation**: Preventing JavaScript file:// access to arbitrary application files
- **Database Isolation**: Preventing web content from accessing native application databases
- **Shared Preferences/User Defaults Isolation**: Protecting native application settings from web access
- **Intent/URL Scheme Filtering**: Validating and restricting URL scheme invocations that could manipulate the host application

See reference: OWASP Mobile Security Testing Guide.

4.6.4 Functions NOT Considered Essential

The following functions, while commonly present in browsers, are not considered essential for the core purpose and may be disabled or restricted in high-security configurations without fundamentally impairing browser functionality:

For standalone browsers:

- Extension/add-on support
- Synchronization of browsing data across devices
- Built-in password management
- Autofill of forms
- Access to hardware devices beyond display and basic input
- Developer tools and debugging interfaces (except in designated development builds)
- Support for legacy protocols or deprecated web standards
- Integration with external applications or services beyond standard web APIs

For embedded browsers (in addition to the above):

- Multiple tab/window management (may be simplified to single content view)
- Browser history UI (may be delegated to host application)
- Download management UI (host application typically handles downloads)
- Print functionality (host application may provide)
- Find-in-page functionality (host application may provide)
- Reader mode or content simplification features

4.7 Operational Environment

The operational environment encompasses the technical, physical, and organizational contexts in which browsers are deployed and operated. Understanding these environments is essential for appropriate security configuration and risk assessment.

4.7.1 Technical Environment

Platform Diversity: Browsers operate across heterogeneous platforms including:

- Desktop operating systems (Windows, macOS, Linux)

- Mobile operating systems (iOS, Android)
- Embedded systems and IoT devices
- Virtualized and containerized environments

Each platform provides different security primitives (process isolation, memory protection, file system controls) that browsers leverage for security enforcement.

Network Conditions: Browsers shall operate across varying network environments:

- Trusted corporate networks with security controls and monitoring
- Home networks with varying security configurations
- Public Wi-Fi networks with potential adversarial presence
- Mobile networks with carrier-level controls
- Air-gapped or isolated networks in high-security environments

Network conditions directly impact threat exposure, with untrusted networks requiring enhanced protection against network-level attacks (MITM, eavesdropping, traffic manipulation).

Hardware Capabilities: The operational environment includes diverse hardware with varying security features:

- Devices with Trusted Platform Modules (TPM) or secure enclaves
- Systems with hardware-enforced memory protection (DEP, ASLR)
- Devices with biometric authentication capabilities
- Systems with varying performance characteristics affecting security feature feasibility

4.7.2 Physical Environment

Physical Security Controls: Browser deployment contexts vary significantly in physical security:

- **Controlled Environments**: Corporate offices, data centers, and secure facilities with physical access controls, surveillance, and authorized personnel only
- **Semi-Controlled Environments**: Home offices, shared workspaces with moderate physical security
- **Uncontrolled Environments**: Public spaces, cafes, transit where device theft, shoulder surfing, and physical tampering are realistic threats
- **Hostile Environments**: Border crossings, adversarial jurisdictions where targeted physical attacks or device seizure may occur

Device Ownership and Control:

- **Corporate-Owned Devices**: Centrally managed with enforced policies, monitoring, and remote management capabilities
- **Bring Your Own Device (BYOD)**: Personal devices used for work purposes with limited organizational control
- **Shared Devices**: Multi-user systems (kiosks, family computers) where isolation between users is required
- **Personal Devices**: Individually owned and managed with user-determined security configurations

4.7.3 Organizational Environment

Governance and Compliance: Organizations deploying browsers may be subject to:

- Regulatory requirements (GDPR, EU Data Act, PCI-DSS, sector-specific regulations)
- Industry standards and certification requirements
- Internal security policies and acceptable use policies
- Contractual obligations to customers or partners

Security Maturity: Organizations vary in security capabilities:

- **Advanced**: Dedicated security teams, SIEM integration, threat intelligence, security orchestration
- **Intermediate**: Basic security tools, patch management, logging and monitoring

- **Basic**: Default configurations, reactive security posture, limited security resources

User Training and Awareness: The effectiveness of browser security controls depends on user security awareness:

- Ability to recognize phishing attempts and malicious websites
- Understanding of permission prompts and security indicators
- Adherence to security policies and best practices
- Reporting of security incidents

4.7.4 Threat Environment

The operational environment includes varying threat actor capabilities and motivations:

Opportunistic Threats: Automated attacks, commodity malware, and broad-spectrum phishing affecting general internet users

Targeted Threats: Attacks directed at specific organizations, industries, or individuals including:

- Corporate espionage and intellectual property theft
- Financial fraud and account takeover
- Nation-state surveillance and intelligence gathering
- Ransomware and extortion campaigns

Insider Threats: Risks from authorized users with legitimate access who may act maliciously or negligently

Supply Chain Threats: Compromise of browser components, extensions, or integrated services during development, distribution, or update processes

4.7.5 Lifecycle Environment

Development and Testing: Browsers used in development environments encounter untrusted code, pre-release features, and non-standard configurations that increase risk exposure.

Production Deployment: Operational browsers serving end-users with expected availability, performance, and security requirements.

Decommissioning: End-of-life browsers that may still contain sensitive data requiring secure disposal or migration processes.

Update and Maintenance: Browsers require ongoing updates that shall be delivered securely, tested for stability, and deployed without service interruption.

4.8 Users

Browser users encompass a diverse population with varying technical expertise, security awareness, and usage patterns. Understanding user characteristics is essential for designing effective security controls that users can understand and properly utilize.

4.8.1 User Categories

General Consumers: The largest user population, including:

- Minimal to moderate technical expertise
- Primary activities: web browsing, social media, shopping, entertainment, personal email
- Security awareness varies widely; may not recognize threats
- Expect seamless user experience; security friction reduces adoption
- Limited ability to troubleshoot security issues independently

Professional Users: Individuals using browsers for work-related activities:

- Moderate technical expertise with domain-specific knowledge
- Activities: SaaS applications, web-based productivity tools, professional research
- Subject to organizational security policies
- Generally higher security awareness due to training
- Balance productivity needs with security requirements

Developers and Technical Users: Advanced users with deep technical knowledge:

- High technical expertise; understand browser internals and web standards
- Activities: web development, testing, debugging, performance analysis
- Require access to developer tools and advanced features
- Higher tolerance for security friction; may disable controls intentionally
- Capable of understanding and responding to complex security warnings

Enterprise Administrators: IT personnel managing browser deployments:

- High technical expertise in systems administration
- Responsible for configuring, deploying, and managing browsers at scale
- Implement group policies and security configurations
- Monitor browser usage and security events
- Balance security requirements with user productivity

Children and Protected Users: Vulnerable populations requiring enhanced protection:

- Limited technical expertise and security awareness
- Susceptible to social engineering and inappropriate content
- Require parental controls and content filtering
- May not understand consequences of security decisions
- Need simplified security interfaces with reduced decision-making burden

4.8.2 User Behavior Patterns

Security Decision-Making: Users exhibit common patterns in security decisions:

- **Alert Fatigue**: Repeated security prompts lead to automatic dismissal without reading
- **Optimism Bias**: Underestimation of personal risk ("it won't happen to me")
- **Immediate Gratification**: Preference for immediate access over security delay
- **Trust Misplacement**: Difficulty distinguishing legitimate from malicious content
- **Habituation**: Security behaviors become routine without conscious evaluation

Permission Granting: When presented with permission requests, users typically:

- Grant permissions to accomplish immediate tasks without evaluating necessity
- Lack understanding of implications of granted permissions
- Rarely review or revoke previously granted permissions
- May not notice when permissions are being actively used

Update Behavior: User approaches to browser updates vary:

- Some users apply updates immediately when prompted
- Others defer updates indefinitely to avoid interruption
- Many users unaware of update status or importance
- Corporate environments often enforce centralized update management

4.8.3 User Needs and Expectations

Usability: Users expect browsers to be:

- Intuitive and easy to use without extensive training
- Responsive and fast, with minimal performance impact from security features
- Compatible with their required websites and applications

- Consistent across sessions and devices

Privacy: Users increasingly expect:

- Control over personal data collection and usage
- Protection from tracking by advertisers and third parties
- Transparency about data practices
- Compliance with privacy regulations

Security: While security awareness varies, users generally expect:

- Protection from malware and phishing without constant manual intervention
- Clear warnings when accessing dangerous sites
- Secure handling of passwords and payment information
- Prevention of unauthorized access to sensitive data

Transparency: Users need:

- Understandable explanations of security features and warnings
- Visibility into what data is being collected and shared
- Clear indication of security status (encrypted connections, permissions in use)
- Accessible security settings and controls

4.8.4 User Assistance and Responsibilities

User Responsibilities: Effective browser security requires users to:

- Keep browsers updated to address vulnerabilities
- Exercise caution with extension installation
- Recognize and avoid phishing attempts
- Use strong, unique passwords for web accounts
- Respond appropriately to security warnings
- Report security incidents when encountered

Browser Responsibilities to Users: Browsers shall support users by:

- Providing clear, actionable security warnings
- Using understandable language avoiding technical jargon
- Offering contextual help and explanations
- Defaulting to secure configurations
- Making security controls discoverable and accessible
- Minimizing security decision burden through sound defaults

4.8.5 Accessibility Considerations

Security features shall be accessible to users with disabilities:

- **Visual Impairments**: Security indicators shall be perceivable by screen readers; warnings shall not rely solely on color
- **Cognitive Disabilities**: Security messages shall be simple and clear; complex security decisions should be minimized
- **Motor Impairments**: Security controls shall be operable with alternative input methods
- **Hearing Impairments**: Audio-based security notifications shall have visual alternatives

Accessibility requirements shall not compromise security effectiveness, but security features should not exclude users with disabilities from safe browser usage.

5 Browser-Specific Risk Factors

5.1 Isolation Mechanisms

5.1.1 Domain and Origin Isolation **[DOM]** The manufacturer shall ensure that execution contexts belonging to different origins are securely isolated to prevent unauthorized data access, code execution, or state manipulation across boundaries. Isolation shall include process separation, independent storage and cache spaces, and validation of all cross-origin communication through standardized, browser-mediated mechanisms such as Cross-Origin Resource Sharing (CORS), which allows controlled sharing of resources between origins through validated HTTP headers, and postMessage, which provides a secure message-passing interface between isolated contexts (e.g. frames or windows). Any relaxation of isolation shall be explicitly authorized, documented, and monitored to prevent data leakage or privilege escalation.

Capability: Browser enforces isolation between domains and origins (defined by scheme, host, and port) to protect integrity and confidentiality of data and execution.

Conditions:

- DOM-0: Full isolation: Each origin is strictly separated. No mechanism exists for cross-origin access or relaxation.
- DOM-1: Controlled isolation: Isolation is enforced by default but may be selectively relaxed through standardized, browser-mediated mechanisms (e.g. CORS or postMessage) with explicit validation.
- DOM-2: Configurable isolation: Isolation is enforced by default, but users or administrators can define exceptions via explicit configuration or policy.
- DOM-3: Integrated isolation: Isolation remains in place, but third-party integrations, compatibility modes, or embedded components may introduce controlled exceptions under defined policies.

Threats:

- Cross-site scripting (XSS) attacks
- Cross-site request forgery (CSRF)
- Data exfiltration between origins
- Session hijacking
- Clickjacking

Risk: HIGH - Failure of isolation mechanisms enables immediate compromise of user data across multiple web applications

Requirements:

DOM-0 Requirements (Full isolation)

- **DOM-0-REQ-1**: Browser shall implement process-per-site isolation for all origins → Assessment: DOM-REQ-1
- **DOM-0-REQ-2**: Browser shall enforce Cross-Origin Read Blocking (CORB) for all cross-origin resource loads → Assessment: DOM-REQ-2
- **DOM-0-REQ-3**: Browser shall prevent all cross-origin DOM access without exception → Assessment: DOM-REQ-3
- **DOM-0-REQ-4**: Browser shall isolate localStorage and IndexedDB per origin with no sharing mechanism → Assessment: DOM-REQ-6
- **DOM-0-REQ-5**: Browser shall treat all sandboxed and data: origins as opaque with no relaxation → Assessment: DOM-REQ-8
- **DOM-0-REQ-6**: Browser shall enforce document.domain restrictions without any override mechanism → Assessment: DOM-REQ-12

DOM-1 Requirements (Controlled isolation)

- **DOM-1-REQ-1**: Browser shall implement process-per-site isolation → Assessment: DOM-REQ-1

- **DOM-1-REQ-2**: Browser shall enforce CORB with exceptions only for properly configured CORS headers → Assessment: DOM-REQ-2
- **DOM-1-REQ-3**: Browser shall prevent cross-origin DOM access except via postMessage → Assessment: DOM-REQ-3
- **DOM-1-REQ-4**: Browser shall enforce CORS preflight for all non-simple cross-origin requests → Assessment: DOM-REQ-4
- **DOM-1-REQ-5**: Browser shall enforce SameSite cookie attribute with Lax as default → Assessment: DOM-REQ-5
- **DOM-1-REQ-6**: Browser shall isolate storage per origin → Assessment: DOM-REQ-6
- **DOM-1-REQ-7**: Browser shall support iframe sandbox attribute with granular tokens → Assessment: DOM-REQ-7
- **DOM-1-REQ-8**: Browser shall treat sandboxed and data: origins as opaque → Assessment: DOM-REQ-8
- **DOM-1-REQ-9**: Browser shall restrict document.domain setter by default (require Origin-Agent-Cluster opt-out) → Assessment: DOM-REQ-12

DOM-2 Requirements (Configurable isolation)

- **DOM-2-REQ-1**: Browser shall implement process-per-site isolation → Assessment: DOM-REQ-1
- **DOM-2-REQ-2**: Browser shall enforce CORB → Assessment: DOM-REQ-2
- **DOM-2-REQ-3**: Browser shall prevent cross-origin DOM access except via CORS and postMessage → Assessment: DOM-REQ-3
- **DOM-2-REQ-4**: Browser shall enforce CORS preflight → Assessment: DOM-REQ-4
- **DOM-2-REQ-5**: Browser shall enforce SameSite cookie policies → Assessment: DOM-REQ-5
- **DOM-2-REQ-6**: Browser shall isolate storage per origin → Assessment: DOM-REQ-6
- **DOM-2-REQ-7**: Browser shall support iframe sandboxing → Assessment: DOM-REQ-7
- **DOM-2-REQ-8**: Browser shall support Cross-Origin-Resource-Policy (CORP) header → Assessment: DOM-REQ-9
- **DOM-2-REQ-9**: Browser shall enforce Cross-Origin-Opener-Policy (COOP) → Assessment: DOM-REQ-10
- **DOM-2-REQ-10**: Browser shall enforce Cross-Origin-Embedder-Policy (COEP) → Assessment: DOM-REQ-11
- **DOM-2-REQ-11**: Administrators shall be able to configure origin isolation policies via enterprise policy
- **DOM-2-REQ-12**: Browser shall log all policy-based isolation exceptions

DOM-3 Requirements (Integrated isolation)

- **DOM-3-REQ-1**: Browser shall implement baseline process-per-site isolation → Assessment: DOM-REQ-1
- **DOM-3-REQ-2**: Browser shall enforce CORB with documented exceptions for compatibility → Assessment: DOM-REQ-2
- **DOM-3-REQ-3**: Browser shall enforce CORS policies → Assessment: DOM-REQ-4
- **DOM-3-REQ-4**: Browser shall enforce SameSite cookie policies → Assessment: DOM-REQ-5
- **DOM-3-REQ-5**: Browser shall isolate storage per origin → Assessment: DOM-REQ-6
- **DOM-3-REQ-6**: Browser shall support CORP, COOP, and COEP headers → Assessments: DOM-REQ-9, DOM-REQ-10, DOM-REQ-11
- **DOM-3-REQ-7**: Compatibility modes shall not weaken core isolation boundaries
- **DOM-3-REQ-8**: Third-party integrations shall be subject to same origin isolation policies
- **DOM-3-REQ-9**: All isolation exceptions for compatibility shall be documented and logged
- **DOM-3-REQ-10**: Embedded components shall maintain storage isolation from embedding context

5.2 Extension System Security

5.2.1 Third-Party Code Execution [**EXT**] The manufacturer shall implement controls for third-party extensions that can modify browser behavior, considering that certified extensions may still exhibit malicious behavior.

Capability: Browser extension system with APIs for third-party code augmentation

Conditions:

- EXT-0: No extension support
- EXT-1: Curated extension store only
- EXT-2: Curated store with developer mode
- EXT-3: Unrestricted extension installation

Threats:

- Malicious extensions harvesting user data
- Extension-based cryptomining
- Browser fingerprinting
- Privilege escalation through extension APIs
- Supply chain attacks on popular extensions

Risk: HIGH - Extensions operate as semi-trusted code with significant access to browser functionality and user data

Requirements:

EXT-0 Requirements (No extension support)

- **EXT-0-REQ-1**: Browser shall not provide any extension installation or execution capability
- **EXT-0-REQ-2**: Browser shall block all attempts to load extension code
- **EXT-0-REQ-3**: Browser build shall not include extension subsystem components

EXT-1 Requirements (Curated extension store only)

- **EXT-1-REQ-1**: Browser shall implement granular permission model for extensions → Assessment: EXT-REQ-1
- **EXT-1-REQ-2**: Browser shall isolate extension content scripts from page scripts → Assessment: EXT-REQ-2
- **EXT-1-REQ-3**: Browser shall enforce extension API access control based on declared permissions → Assessment: EXT-REQ-3
- **EXT-1-REQ-4**: Browser shall validate extension manifests before installation → Assessment: EXT-REQ-4
- **EXT-1-REQ-5**: Browser shall sandbox extension execution environments → Assessment: EXT-REQ-5
- **EXT-1-REQ-6**: Browser shall isolate extensions from each other → Assessment: EXT-REQ-6
- **EXT-1-REQ-7**: Browser shall validate host permissions against declared patterns → Assessment: EXT-REQ-7
- **EXT-1-REQ-8**: Browser shall enforce Content Security Policy for extensions → Assessment: EXT-REQ-8
- **EXT-1-REQ-9**: Browser shall verify extension update signatures from official store → Assessment: EXT-REQ-10
- **EXT-1-REQ-10**: Browser shall enforce Manifest V3 compliance for new extensions → Assessment: EXT-REQ-13
- **EXT-1-REQ-11**: Browser shall validate extension signatures before installation → Assessment: EXT-REQ-17
- **EXT-1-REQ-12**: Browser shall provide transparent permissions UI during installation → Assessment: EXT-REQ-18
- **EXT-1-REQ-13**: Browser shall only allow installation from official curated store

- **EXT-1-REQ-14**: Extensions shall undergo security review before store publication

EXT-2 Requirements (Curated store with developer mode)

- **EXT-2-REQ-1**: Browser shall implement all EXT-1 requirements for store-installed extensions
- **EXT-2-REQ-2**: Browser shall isolate extension storage per extension → Assessment: EXT-REQ-11
- **EXT-2-REQ-3**: Browser shall restrict background script capabilities → Assessment: EXT-REQ-12
- **EXT-2-REQ-4**: Browser shall enforce WebRequest API security controls → Assessment: EXT-REQ-9
- **EXT-2-REQ-5**: Browser shall monitor extension-controlled web content → Assessment: EXT-REQ-15
- **EXT-2-REQ-6**: Developer mode shall require explicit user activation with security warnings
- **EXT-2-REQ-7**: Developer mode extensions shall display persistent visual indicators
- **EXT-2-REQ-8**: Developer mode shall disable automatic extension updates
- **EXT-2-REQ-9**: Browser shall log all developer mode extension activities
- **EXT-2-REQ-10**: Enterprise policies shall be able to disable developer mode

EXT-3 Requirements (Unrestricted extension installation)

- **EXT-3-REQ-1**: Browser shall implement baseline extension permission model → Assessment: EXT-REQ-1
- **EXT-3-REQ-2**: Browser shall enforce content script isolation → Assessment: EXT-REQ-2
- **EXT-3-REQ-3**: Browser shall control extension API access → Assessment: EXT-REQ-3
- **EXT-3-REQ-4**: Browser shall validate extension manifests → Assessment: EXT-REQ-4
- **EXT-3-REQ-5**: Browser shall sandbox extensions → Assessment: EXT-REQ-5
- **EXT-3-REQ-6**: Browser shall isolate extensions from each other → Assessment: EXT-REQ-6
- **EXT-3-REQ-7**: Browser shall enforce native messaging security controls → Assessment: EXT-REQ-14
- **EXT-3-REQ-8**: Browser shall monitor extension telemetry with privacy protections → Assessment: EXT-REQ-16
- **EXT-3-REQ-9**: Sideloaded extensions shall display prominent security warnings
- **EXT-3-REQ-10**: Browser shall provide user controls to review and revoke extension permissions
- **EXT-3-REQ-11**: Browser shall scan sideloaded extensions for known malware signatures
- **EXT-3-REQ-12**: All extension security events shall be logged for review

5.3 Encryption Implementation

5.3.1 Data Protection Layers **[ENC]** The manufacturer shall implement comprehensive encryption across all data states and communication channels.

Capability: Multi-layer encryption for data in transit, at rest, and during synchronization

Conditions:

- ENC-0: Full encryption with hardware security module support
- ENC-1: Standard encryption with software-based key management
- ENC-2: Selective encryption based on data sensitivity
- ENC-3: Basic encryption with user-optional enhanced protection
- ENC-4: Full

Threats:

- Man-in-the-middle attacks
- Data breach of stored credentials
- Synchronization data interception
- Update channel compromise
- Certificate validation bypass

Risk: CRITICAL - Encryption failures expose all user data and communications

Requirements:

ENC-0 Requirements (Full encryption with HSM support)

- **ENC-0-REQ-1**: Browser shall support TLS 1.3 or higher exclusively for all network communications → Assessment: ENC-REQ-1
- **ENC-0-REQ-2**: Browser shall perform complete certificate chain validation including expiry, revocation, and trust anchor verification → Assessment: ENC-REQ-2
- **ENC-0-REQ-3**: Browser shall enforce HTTP Public Key Pinning (HPKP) or alternative certificate pinning mechanisms → Assessment: ENC-REQ-3
- **ENC-0-REQ-4**: Browser shall enforce HTTP Strict Transport Security (HSTS) with preload support → Assessment: ENC-REQ-4
- **ENC-0-REQ-5**: Browser shall block all mixed content without exception → Assessment: ENC-REQ-5
- **ENC-0-REQ-6**: Browser shall enforce Certificate Transparency requirements for all certificates → Assessment: ENC-REQ-6
- **ENC-0-REQ-7**: Browser shall support and validate OCSP stapling responses → Assessment: ENC-REQ-7
- **ENC-0-REQ-8**: Browser shall restrict cipher suites to strong, modern algorithms only → Assessment: ENC-REQ-8
- **ENC-0-REQ-9**: Browser shall enforce perfect forward secrecy for all TLS connections → Assessment: ENC-REQ-9
- **ENC-0-REQ-10**: Browser shall perform real-time certificate revocation checking → Assessment: ENC-REQ-10
- **ENC-0-REQ-11**: Browser shall implement Web Crypto API with full W3C compliance → Assessment: ENC-REQ-11
- **ENC-0-REQ-12**: Browser shall provide cryptographically secure random number generation → Assessment: ENC-REQ-12
- **ENC-0-REQ-13**: Browser shall enforce SubResource Integrity (SRI) validation → Assessment: ENC-REQ-13
- **ENC-0-REQ-14**: Browser shall support Encrypted SNI (ESNI) or Encrypted Client Hello (ECH) → Assessment: ENC-REQ-14
- **ENC-0-REQ-15**: Browser shall display prominent, non-bypassable certificate error UI → Assessment: ENC-REQ-15
- **ENC-0-REQ-16**: Browser shall operate in HTTPS-first mode with automatic upgrade → Assessment: ENC-REQ-16
- **ENC-0-REQ-17**: Browser shall detect and alert on certificate pinning bypass attempts → Assessment: ENC-REQ-17
- **ENC-0-REQ-18**: Browser shall protect against TLS downgrade attacks → Assessment: ENC-REQ-18
- **ENC-0-REQ-19**: Browser shall deprecate and disable legacy cryptographic protocols (SSL, TLS 1.0/1.1) → Assessment: ENC-REQ-19
- **ENC-0-REQ-20**: Browser shall isolate cryptographic keys in hardware security modules when available → Assessment: ENC-REQ-20
- **ENC-0-REQ-21**: Browser shall implement secure certificate store with integrity protection → Assessment: ENC-REQ-21
- **ENC-0-REQ-22**: Browser shall not allow user override of certificate pinning failures
- **ENC-0-REQ-23**: All cryptographic operations shall be performed in hardware-backed secure enclaves when available

ENC-1 Requirements (Standard encryption with software key management)

- **ENC-1-REQ-1**: Browser shall support TLS 1.3 or higher for all network communications → Assessment: ENC-REQ-1
- **ENC-1-REQ-2**: Browser shall perform complete certificate chain validation → Assessment: ENC-REQ-2
- **ENC-1-REQ-3**: Browser shall support certificate pinning mechanisms → Assessment: ENC-REQ-3
- **ENC-1-REQ-4**: Browser shall enforce HSTS to prevent protocol downgrade attacks → Assessment: ENC-REQ-4

- **ENC-1-REQ-5**: Browser shall block mixed content with opt-in for passive content → Assessment: ENC-REQ-5
- **ENC-1-REQ-6**: Browser shall validate Certificate Transparency logs → Assessment: ENC-REQ-6
- **ENC-1-REQ-7**: Browser shall support OCSP stapling → Assessment: ENC-REQ-7
- **ENC-1-REQ-8**: Browser shall restrict cipher suites to industry-standard secure algorithms → Assessment: ENC-REQ-8
- **ENC-1-REQ-9**: Browser shall prefer perfect forward secrecy cipher suites → Assessment: ENC-REQ-9
- **ENC-1-REQ-10**: Browser shall perform certificate revocation checking (OCSP or CRLSets) → Assessment: ENC-REQ-10
- **ENC-1-REQ-11**: Browser shall implement Web Crypto API → Assessment: ENC-REQ-11
- **ENC-1-REQ-12**: Browser shall provide secure random number generation → Assessment: ENC-REQ-12
- **ENC-1-REQ-13**: Browser shall support SubResource Integrity validation → Assessment: ENC-REQ-13
- **ENC-1-REQ-14**: Browser shall display clear certificate error UI with bypass warnings → Assessment: ENC-REQ-15
- **ENC-1-REQ-15**: Browser shall support HTTPS-first mode as user option → Assessment: ENC-REQ-16
- **ENC-1-REQ-16**: Browser shall protect against TLS downgrade attacks → Assessment: ENC-REQ-18
- **ENC-1-REQ-17**: Browser shall disable legacy cryptographic protocols by default → Assessment: ENC-REQ-19
- **ENC-1-REQ-18**: Browser shall isolate cryptographic keys in process-level sandboxes → Assessment: ENC-REQ-20
- **ENC-1-REQ-19**: Browser shall implement secure certificate store → Assessment: ENC-REQ-21

ENC-2 Requirements (Selective encryption)

- **ENC-2-REQ-1**: Browser shall support TLS 1.3 and TLS 1.2 → Assessment: ENC-REQ-1
- **ENC-2-REQ-2**: Browser shall perform certificate chain validation → Assessment: ENC-REQ-2
- **ENC-2-REQ-3**: Browser shall support certificate pinning for critical domains → Assessment: ENC-REQ-3
- **ENC-2-REQ-4**: Browser shall enforce HSTS when declared by server → Assessment: ENC-REQ-4
- **ENC-2-REQ-5**: Browser shall warn about mixed content → Assessment: ENC-REQ-5
- **ENC-2-REQ-6**: Browser shall support Certificate Transparency → Assessment: ENC-REQ-6
- **ENC-2-REQ-7**: Browser shall support OCSP stapling → Assessment: ENC-REQ-7
- **ENC-2-REQ-8**: Browser shall support standard cipher suites → Assessment: ENC-REQ-8
- **ENC-2-REQ-9**: Browser shall support forward secrecy cipher suites → Assessment: ENC-REQ-9
- **ENC-2-REQ-10**: Browser shall perform revocation checking with soft-fail option → Assessment: ENC-REQ-10
- **ENC-2-REQ-11**: Browser shall implement Web Crypto API → Assessment: ENC-REQ-11
- **ENC-2-REQ-12**: Browser shall provide secure random number generation → Assessment: ENC-REQ-12
- **ENC-2-REQ-13**: Browser shall support SubResource Integrity → Assessment: ENC-REQ-13
- **ENC-2-REQ-14**: Browser shall display certificate errors with bypass option → Assessment: ENC-REQ-15
- **ENC-2-REQ-15**: Browser shall provide HTTPS upgrade suggestions → Assessment: ENC-REQ-16
- **ENC-2-REQ-16**: Browser shall maintain secure certificate store → Assessment: ENC-REQ-21
- **ENC-2-REQ-17**: Users shall be able to configure encryption strictness levels
- **ENC-2-REQ-18**: Browser shall provide visual indicators for connection security status

ENC-3 Requirements (Basic encryption with optional enhanced protection)

- **ENC-3-REQ-1**: Browser shall support TLS 1.2 or higher → Assessment: ENC-REQ-1
- **ENC-3-REQ-2**: Browser shall perform basic certificate validation → Assessment: ENC-REQ-2
- **ENC-3-REQ-3**: Browser shall enforce HSTS when enabled → Assessment: ENC-REQ-4

- **ENC-3-REQ-4**: Browser shall notify users of mixed content → Assessment: ENC-REQ-5
- **ENC-3-REQ-5**: Browser shall support common cipher suites → Assessment: ENC-REQ-8
- **ENC-3-REQ-6**: Browser shall support certificate revocation checking → Assessment: ENC-REQ-10
- **ENC-3-REQ-7**: Browser shall implement Web Crypto API → Assessment: ENC-REQ-11
- **ENC-3-REQ-8**: Browser shall provide secure random number generation → Assessment: ENC-REQ-12
- **ENC-3-REQ-9**: Browser shall display certificate errors with clear bypass options → Assessment: ENC-REQ-15
- **ENC-3-REQ-10**: Browser shall support basic certificate store operations → Assessment: ENC-REQ-21
- **ENC-3-REQ-11**: Users shall have full control over encryption settings
- **ENC-3-REQ-12**: Browser shall provide encryption status indicators in UI
- **ENC-3-REQ-13**: Legacy protocol support may be enabled for compatibility with user consent

ENC-4 Requirements (Minimal encryption)

- **ENC-4-REQ-1**: Browser shall support TLS 1.2 → Assessment: ENC-REQ-1
- **ENC-4-REQ-2**: Browser shall perform basic certificate validation → Assessment: ENC-REQ-2
- **ENC-4-REQ-3**: Browser shall display certificate warnings → Assessment: ENC-REQ-15
- **ENC-4-REQ-4**: Browser shall implement Web Crypto API for web applications → Assessment: ENC-REQ-11
- **ENC-4-REQ-5**: Browser shall provide secure random number generation → Assessment: ENC-REQ-12
- **ENC-4-REQ-6**: Users shall have complete control over all encryption policies
- **ENC-4-REQ-7**: Browser shall support legacy protocols when explicitly enabled by user

5.4 Diagnostic and Monitoring Systems

5.4.1 Logging and Crash Reporting [**LOG**] The manufacturer shall balance diagnostic capabilities with privacy protection in logging and monitoring systems.

Capability: Crash dumps, audit trails, activity logs, and enterprise integration capabilities

Conditions:

- LOG-0: No logging or local-only logging
- LOG-1: Opt-in telemetry with anonymization
- LOG-2: Default telemetry with opt-out
- LOG-3: Mandatory telemetry for enterprise management

Threats:

- Information disclosure through logs
- Privacy violations via telemetry
- Log tampering or deletion
- Unauthorized access to diagnostic data
- Correlation attacks using telemetry data

Risk: MEDIUM - Diagnostic systems may leak sensitive information while being necessary for security monitoring

Requirements:

LOG-0 Requirements (No logging or local-only)

- **LOG-0-REQ-1**: Browser shall not transmit any telemetry or diagnostic data to remote servers
- **LOG-0-REQ-2**: Browser shall maintain local security event logs for audit purposes → Assessment: LOG-REQ-1
- **LOG-0-REQ-3**: Local logs shall be stored with integrity protection → Assessment: LOG-REQ-11
- **LOG-0-REQ-4**: Local logs shall implement data minimization principles → Assessment: LOG-REQ-7

- **LOG-0-REQ-5**: Browser shall provide local security dashboard for log review → Assessment: LOG-REQ-13
- **LOG-0-REQ-6**: Local logs shall support forensic export for security analysis → Assessment: LOG-REQ-17
- **LOG-0-REQ-7**: Browser shall enforce strict access controls on local log files → Assessment: LOG-REQ-20
- **LOG-0-REQ-8**: All logging shall be disabled by default with opt-in for local logging
- **LOG-0-REQ-9**: Users shall be able to view and delete all local logs at any time

LOG-1 Requirements (Opt-in telemetry with anonymization)

- **LOG-1-REQ-1**: Browser shall log security events locally → Assessment: LOG-REQ-1
- **LOG-1-REQ-2**: Browser shall log certificate errors and validation failures → Assessment: LOG-REQ-2
- **LOG-1-REQ-3**: Browser shall log extension security events → Assessment: LOG-REQ-3
- **LOG-1-REQ-4**: Browser shall support CSP violation reporting → Assessment: LOG-REQ-4
- **LOG-1-REQ-5**: Browser shall support Network Error Logging (NEL) → Assessment: LOG-REQ-5
- **LOG-1-REQ-6**: Browser shall implement crash reporting with user consent → Assessment: LOG-REQ-6
- **LOG-1-REQ-7**: Browser shall minimize data collection in logs → Assessment: LOG-REQ-7
- **LOG-1-REQ-8**: Browser shall anonymize telemetry data before transmission → Assessment: LOG-REQ-8
- **LOG-1-REQ-9**: Browser shall require explicit user consent for all telemetry → Assessment: LOG-REQ-9
- **LOG-1-REQ-10**: Browser shall transmit logs over secure channels only → Assessment: LOG-REQ-10
- **LOG-1-REQ-11**: Browser shall protect log integrity with cryptographic signatures → Assessment: LOG-REQ-11
- **LOG-1-REQ-12**: Browser shall enforce log retention policies → Assessment: LOG-REQ-12
- **LOG-1-REQ-13**: Browser shall provide security dashboard for log review → Assessment: LOG-REQ-13
- **LOG-1-REQ-14**: Browser shall use privacy-preserving analytics techniques → Assessment: LOG-REQ-18
- **LOG-1-REQ-15**: Browser shall enforce access controls on diagnostic data → Assessment: LOG-REQ-20
- **LOG-1-REQ-16**: Telemetry shall be disabled by default and require explicit opt-in
- **LOG-1-REQ-17**: Users shall have granular control over telemetry categories
- **LOG-1-REQ-18**: Browser shall provide clear documentation of all collected data

LOG-2 Requirements (Default telemetry with opt-out)

- **LOG-2-REQ-1**: Browser shall log security events → Assessment: LOG-REQ-1
- **LOG-2-REQ-2**: Browser shall log certificate errors → Assessment: LOG-REQ-2
- **LOG-2-REQ-3**: Browser shall log extension security events → Assessment: LOG-REQ-3
- **LOG-2-REQ-4**: Browser shall support CSP violation reporting → Assessment: LOG-REQ-4
- **LOG-2-REQ-5**: Browser shall support Network Error Logging → Assessment: LOG-REQ-5
- **LOG-2-REQ-6**: Browser shall implement crash reporting → Assessment: LOG-REQ-6
- **LOG-2-REQ-7**: Browser shall minimize logged data → Assessment: LOG-REQ-7
- **LOG-2-REQ-8**: Browser shall anonymize telemetry data → Assessment: LOG-REQ-8
- **LOG-2-REQ-9**: Browser shall provide clear opt-out mechanism during first run → Assessment: LOG-REQ-9
- **LOG-2-REQ-10**: Browser shall transmit logs securely → Assessment: LOG-REQ-10
- **LOG-2-REQ-11**: Browser shall protect log integrity → Assessment: LOG-REQ-11
- **LOG-2-REQ-12**: Browser shall enforce retention policies → Assessment: LOG-REQ-12
- **LOG-2-REQ-13**: Browser shall provide security dashboard → Assessment: LOG-REQ-13
- **LOG-2-REQ-14**: Browser shall support incident detection → Assessment: LOG-REQ-14
- **LOG-2-REQ-15**: Browser shall maintain complete audit trail → Assessment: LOG-REQ-15
- **LOG-2-REQ-16**: Browser shall use privacy-preserving analytics → Assessment: LOG-REQ-18
- **LOG-2-REQ-17**: Browser shall support compliance logging → Assessment: LOG-REQ-19

- **LOG-2-REQ-18**: Browser shall enforce log access controls → Assessment: LOG-REQ-20
- **LOG-2-REQ-19**: Users shall be able to disable telemetry at any time
- **LOG-2-REQ-20**: Browser shall display telemetry status in settings UI

LOG-3 Requirements (Mandatory telemetry for enterprise)

- **LOG-3-REQ-1**: Browser shall log all security events → Assessment: LOG-REQ-1
- **LOG-3-REQ-2**: Browser shall log certificate errors and security warnings → Assessment: LOG-REQ-2
- **LOG-3-REQ-3**: Browser shall log all extension security events → Assessment: LOG-REQ-3
- **LOG-3-REQ-4**: Browser shall support CSP violation reporting → Assessment: LOG-REQ-4
- **LOG-3-REQ-5**: Browser shall support Network Error Logging → Assessment: LOG-REQ-5
- **LOG-3-REQ-6**: Browser shall implement comprehensive crash reporting → Assessment: LOG-REQ-6
- **LOG-3-REQ-7**: Browser shall log data with minimal redaction for forensics → Assessment: LOG-REQ-7
- **LOG-3-REQ-8**: Browser shall transmit logs securely to enterprise SIEM → Assessment: LOG-REQ-10
- **LOG-3-REQ-9**: Browser shall protect log integrity with cryptographic controls → Assessment: LOG-REQ-11
- **LOG-3-REQ-10**: Browser shall enforce enterprise-defined retention policies → Assessment: LOG-REQ-12
- **LOG-3-REQ-11**: Browser shall provide comprehensive security dashboard → Assessment: LOG-REQ-13
- **LOG-3-REQ-12**: Browser shall support real-time incident detection → Assessment: LOG-REQ-14
- **LOG-3-REQ-13**: Browser shall maintain complete audit trail of all security events → Assessment: LOG-REQ-15
- **LOG-3-REQ-14**: Browser shall support real-time security alerts → Assessment: LOG-REQ-16
- **LOG-3-REQ-15**: Browser shall support forensic log export in standard formats → Assessment: LOG-REQ-17
- **LOG-3-REQ-16**: Browser shall implement compliance logging for regulatory requirements → Assessment: LOG-REQ-19
- **LOG-3-REQ-17**: Browser shall enforce role-based access controls for logs → Assessment: LOG-REQ-20
- **LOG-3-REQ-18**: Enterprise policies shall prevent users from disabling mandatory logging
- **LOG-3-REQ-19**: Browser shall support integration with enterprise monitoring systems
- **LOG-3-REQ-20**: Browser shall provide tamper-evident logging mechanisms

5.5 Update Delivery Mechanisms

5.5.1 Security Update Management **[UPD]** The manufacturer shall implement secure update mechanisms that balance security needs with user autonomy.

Capability: Automatic security updates with user notification and control options

Conditions:

- UPD-0: Forced automatic updates without user control
- UPD-1: Automatic updates with postponement options
- UPD-2: Optional automatic updates (user should enable)
- UPD-3: Manual updates only

Threats:

- Exploitation of unpatched vulnerabilities
- Update channel compromise
- Malicious update injection
- Denial of service through forced updates
- Zero-day exploitation window

Risk: HIGH - Delayed or compromised updates leave browsers vulnerable to known exploits

Requirements:

UPD-0 Requirements (Forced automatic updates)

- **UPD-0-REQ-1**: Browser shall implement automatic update mechanism without user control → Assessment: UPD-REQ-1
- **UPD-0-REQ-2**: Browser shall verify digital signatures on all updates → Assessment: UPD-REQ-2
- **UPD-0-REQ-3**: Browser shall deliver updates exclusively over HTTPS → Assessment: UPD-REQ-3
- **UPD-0-REQ-4**: Browser shall validate update manifest integrity → Assessment: UPD-REQ-4
- **UPD-0-REQ-5**: Browser shall implement rollback protection to prevent downgrade attacks → Assessment: UPD-REQ-5
- **UPD-0-REQ-6**: Browser shall isolate update channels (stable, beta, dev) → Assessment: UPD-REQ-6
- **UPD-0-REQ-7**: Browser shall support component-level updates for security patches → Assessment: UPD-REQ-7
- **UPD-0-REQ-8**: Browser shall support emergency update capability for zero-day threats → Assessment: UPD-REQ-8
- **UPD-0-REQ-9**: Browser shall verify updates before installation → Assessment: UPD-REQ-9
- **UPD-0-REQ-10**: Browser shall implement failure recovery for failed updates → Assessment: UPD-REQ-10
- **UPD-0-REQ-11**: Browser shall log all updates to transparency log → Assessment: UPD-REQ-11
- **UPD-0-REQ-12**: Browser shall validate delta update security → Assessment: UPD-REQ-12
- **UPD-0-REQ-13**: Browser shall authenticate update servers with certificate pinning → Assessment: UPD-REQ-13
- **UPD-0-REQ-14**: Browser shall implement timing jitter to prevent fingerprinting → Assessment: UPD-REQ-14
- **UPD-0-REQ-15**: Browser shall enforce background update installation → Assessment: UPD-REQ-15
- **UPD-0-REQ-16**: Browser shall force critical security updates immediately → Assessment: UPD-REQ-17
- **UPD-0-REQ-17**: Browser shall verify complete update verification chain → Assessment: UPD-REQ-18
- **UPD-0-REQ-18**: Browser shall implement update source pinning → Assessment: UPD-REQ-19
- **UPD-0-REQ-19**: Browser shall verify update integrity with cryptographic hashes → Assessment: UPD-REQ-20
- **UPD-0-REQ-20**: Browser shall support staged rollout for risk mitigation → Assessment: UPD-REQ-21
- **UPD-0-REQ-21**: Browser shall validate update domain authenticity → Assessment: UPD-REQ-22
- **UPD-0-REQ-22**: Browser shall support binary reproducibility verification → Assessment: UPD-REQ-23
- **UPD-0-REQ-23**: Updates shall be applied without user intervention or postponement
- **UPD-0-REQ-24**: Browser shall restart automatically after critical updates when safe

UPD-1 Requirements (Automatic with postponement)

- **UPD-1-REQ-1**: Browser shall implement automatic update mechanism → Assessment: UPD-REQ-1
- **UPD-1-REQ-2**: Browser shall verify update signatures → Assessment: UPD-REQ-2
- **UPD-1-REQ-3**: Browser shall deliver updates over HTTPS only → Assessment: UPD-REQ-3
- **UPD-1-REQ-4**: Browser shall validate update manifest integrity → Assessment: UPD-REQ-4
- **UPD-1-REQ-5**: Browser shall implement rollback protection → Assessment: UPD-REQ-5
- **UPD-1-REQ-6**: Browser shall isolate update channels → Assessment: UPD-REQ-6
- **UPD-1-REQ-7**: Browser shall support component updates → Assessment: UPD-REQ-7
- **UPD-1-REQ-8**: Browser shall support emergency updates → Assessment: UPD-REQ-8
- **UPD-1-REQ-9**: Browser shall verify updates before installation → Assessment: UPD-REQ-9
- **UPD-1-REQ-10**: Browser shall implement update failure recovery → Assessment: UPD-REQ-10
- **UPD-1-REQ-11**: Browser shall log updates to transparency log → Assessment: UPD-REQ-11
- **UPD-1-REQ-12**: Browser shall secure delta updates → Assessment: UPD-REQ-12
- **UPD-1-REQ-13**: Browser shall authenticate update servers → Assessment: UPD-REQ-13
- **UPD-1-REQ-14**: Browser shall implement update timing jitter → Assessment: UPD-REQ-14

- **UPD-1-REQ-15**: Browser shall enforce background updates → Assessment: UPD-REQ-15
- **UPD-1-REQ-16**: Browser shall display clear update notifications → Assessment: UPD-REQ-16
- **UPD-1-REQ-17**: Browser shall force critical updates with limited postponement → Assessment: UPD-REQ-17
- **UPD-1-REQ-18**: Browser shall verify update verification chain → Assessment: UPD-REQ-18
- **UPD-1-REQ-19**: Browser shall implement update source pinning → Assessment: UPD-REQ-19
- **UPD-1-REQ-20**: Browser shall verify update integrity → Assessment: UPD-REQ-20
- **UPD-1-REQ-21**: Browser shall support staged rollouts → Assessment: UPD-REQ-21
- **UPD-1-REQ-22**: Browser shall validate update domains → Assessment: UPD-REQ-22
- **UPD-1-REQ-23**: Users shall be able to postpone non-critical updates for limited time (max 7 days)
- **UPD-1-REQ-24**: Critical security updates shall not be postponable beyond 24 hours
- **UPD-1-REQ-25**: Browser shall notify users of pending updates with severity indication

UPD-2 Requirements (Optional automatic updates)

- **UPD-2-REQ-1**: Browser shall support automatic update mechanism when enabled → Assessment: UPD-REQ-1
- **UPD-2-REQ-2**: Browser shall verify update signatures → Assessment: UPD-REQ-2
- **UPD-2-REQ-3**: Browser shall deliver updates over HTTPS → Assessment: UPD-REQ-3
- **UPD-2-REQ-4**: Browser shall validate update manifest integrity → Assessment: UPD-REQ-4
- **UPD-2-REQ-5**: Browser shall implement rollback protection → Assessment: UPD-REQ-5
- **UPD-2-REQ-6**: Browser shall isolate update channels → Assessment: UPD-REQ-6
- **UPD-2-REQ-7**: Browser shall support component updates → Assessment: UPD-REQ-7
- **UPD-2-REQ-8**: Browser shall verify updates before installation → Assessment: UPD-REQ-9
- **UPD-2-REQ-9**: Browser shall implement update failure recovery → Assessment: UPD-REQ-10
- **UPD-2-REQ-10**: Browser shall authenticate update servers → Assessment: UPD-REQ-13
- **UPD-2-REQ-11**: Browser shall display update notifications → Assessment: UPD-REQ-16
- **UPD-2-REQ-12**: Browser shall verify update integrity → Assessment: UPD-REQ-20
- **UPD-2-REQ-13**: Browser shall validate update domains → Assessment: UPD-REQ-22
- **UPD-2-REQ-14**: Automatic updates shall be disabled by default
- **UPD-2-REQ-15**: Browser shall prominently recommend enabling automatic updates
- **UPD-2-REQ-16**: Browser shall display security warnings when updates are available
- **UPD-2-REQ-17**: Browser shall provide easy mechanism to check for and install updates
- **UPD-2-REQ-18**: Users shall have full control over update timing and installation

UPD-3 Requirements (Manual updates only)

- **UPD-3-REQ-1**: Browser shall verify update signatures when manually triggered → Assessment: UPD-REQ-2
- **UPD-3-REQ-2**: Browser shall deliver updates over HTTPS → Assessment: UPD-REQ-3
- **UPD-3-REQ-3**: Browser shall validate update manifest integrity → Assessment: UPD-REQ-4
- **UPD-3-REQ-4**: Browser shall implement rollback protection → Assessment: UPD-REQ-5
- **UPD-3-REQ-5**: Browser shall verify updates before installation → Assessment: UPD-REQ-9
- **UPD-3-REQ-6**: Browser shall implement update failure recovery → Assessment: UPD-REQ-10
- **UPD-3-REQ-7**: Browser shall authenticate update servers → Assessment: UPD-REQ-13
- **UPD-3-REQ-8**: Browser shall display update availability notifications → Assessment: UPD-REQ-16
- **UPD-3-REQ-9**: Browser shall verify update integrity → Assessment: UPD-REQ-20
- **UPD-3-REQ-10**: Browser shall validate update domains → Assessment: UPD-REQ-22
- **UPD-3-REQ-11**: Browser shall provide manual update check mechanism
- **UPD-3-REQ-12**: Browser shall display security warnings for outdated versions
- **UPD-3-REQ-13**: Browser shall provide clear indication of available security updates
- **UPD-3-REQ-14**: All updates shall require explicit user initiation
- **UPD-3-REQ-15**: Browser shall display update changelog and security impact

5.6 Protocol Handler Security

5.6.1 Custom Protocol Management **[PRO]** The manufacturer shall securely handle various communication protocols beyond standard HTTP/HTTPS.

Capability: Support for custom schemes, WebSocket, WebRTC, and emerging web standards

Conditions:

- PRO-0: HTTP/HTTPS only
- PRO-1: Standard web protocols with strict validation
- PRO-2: Custom protocols with registration system
- PRO-3: Unrestricted protocol handler registration

Threats:

- Protocol confusion attacks
- Scheme hijacking
- Bypass of security controls via custom protocols
- Data leakage through protocol handlers
- Local application exploitation via URL schemes

Risk: MEDIUM - Custom protocols can bypass standard web security controls

Requirements:

PRO-0 Requirements (HTTP/HTTPS only)

- **PRO-0-REQ-1**: Browser shall only support HTTP and HTTPS protocols
- **PRO-0-REQ-2**: Browser shall reject all custom protocol handler registration attempts
- **PRO-0-REQ-3**: Browser shall block access to non-standard URL schemes (file://, data://, javascript://, etc.)
- **PRO-0-REQ-4**: Browser shall not provide registerProtocolHandler() API or equivalent functionality
- **PRO-0-REQ-5**: Browser shall reject navigation to any non-HTTP/HTTPS protocols

PRO-1 Requirements (Standard web protocols with strict validation)

- **PRO-1-REQ-1**: Browser shall validate all custom protocol handler registrations → Assessment: PRO-REQ-1
- **PRO-1-REQ-2**: Browser shall obtain explicit user consent before activating custom protocol handlers → Assessment: PRO-REQ-2
- **PRO-1-REQ-3**: Browser shall enforce protocol allowlists that restrict which custom schemes can be registered → Assessment: PRO-REQ-3
- **PRO-1-REQ-4**: Browser shall prevent scheme hijacking attacks → Assessment: PRO-REQ-4
- **PRO-1-REQ-5**: Browser shall sanitize protocol URL parameters before passing to handlers → Assessment: PRO-REQ-5
- **PRO-1-REQ-6**: Browser shall implement security controls for external protocol handlers → Assessment: PRO-REQ-6
- **PRO-1-REQ-7**: Browser shall provide transparent UI indicating protocol handler registration and invocation → Assessment: PRO-REQ-7
- **PRO-1-REQ-8**: Browser shall prevent protocol downgrade attacks → Assessment: PRO-REQ-8
- **PRO-1-REQ-9**: Browser shall log protocol handler registration, modification, and invocation events → Assessment: PRO-REQ-9
- **PRO-1-REQ-10**: Browser shall support web+custom scheme conventions → Assessment: PRO-REQ-10
- **PRO-1-REQ-11**: Browser shall enforce handler capability restrictions → Assessment: PRO-REQ-13
- **PRO-1-REQ-12**: Browser shall provide protocol handler revocation mechanisms → Assessment: PRO-REQ-14
- **PRO-1-REQ-13**: Browser shall enforce cross-origin protocol restrictions → Assessment: PRO-REQ-15

PRO-2 Requirements (Custom protocols with registration system)

- **PRO-2-REQ-1**: Browser shall implement all PRO-1 requirements
- **PRO-2-REQ-2**: Browser shall validate protocol handler persistence across sessions → Assessment: PRO-REQ-11
- **PRO-2-REQ-3**: Browser shall mitigate protocol confusion attacks → Assessment: PRO-REQ-12
- **PRO-2-REQ-4**: Browser shall validate protocol handler manifests → Assessment: PRO-REQ-16
- **PRO-2-REQ-5**: Browser shall integrate protocol handlers with Content Security Policy → Assessment: PRO-REQ-20
- **PRO-2-REQ-6**: Browser shall maintain audit trail for handler registrations → Assessment: PRO-REQ-21
- **PRO-2-REQ-7**: Browser shall enforce secure handler update mechanisms → Assessment: PRO-REQ-22
- **PRO-2-REQ-8**: Browser shall enforce handler isolation between origins → Assessment: PRO-REQ-23
- **PRO-2-REQ-9**: Browser shall support Intent URL security on Android platforms → Assessment: PRO-REQ-17
- **PRO-2-REQ-10**: Browser shall support Universal Links security on iOS platforms → Assessment: PRO-REQ-18
- **PRO-2-REQ-11**: Browser shall validate mobile deep linking security → Assessment: PRO-REQ-19
- **PRO-2-REQ-12**: Enterprise administrators shall be able to configure protocol handler allowlists and blocklists

PRO-3 Requirements (Unrestricted protocol registration)

- **PRO-3-REQ-1**: Browser shall implement baseline protocol handler validation → Assessment: PRO-REQ-1
- **PRO-3-REQ-2**: Browser shall obtain user consent for protocol handler activation → Assessment: PRO-REQ-2
- **PRO-3-REQ-3**: Browser shall sanitize protocol parameters → Assessment: PRO-REQ-5
- **PRO-3-REQ-4**: Browser shall implement external handler security controls → Assessment: PRO-REQ-6
- **PRO-3-REQ-5**: Browser shall provide handler management UI → Assessment: PRO-REQ-7
- **PRO-3-REQ-6**: Browser shall log protocol handler security events → Assessment: PRO-REQ-9
- **PRO-3-REQ-7**: Browser shall allow registration of custom schemes without web+ prefix
- **PRO-3-REQ-8**: Browser shall display security warnings for non-standard protocol handlers
- **PRO-3-REQ-9**: Browser shall provide user-accessible handler revocation controls → Assessment: PRO-REQ-14
- **PRO-3-REQ-10**: Users shall be able to review all registered protocol handlers in browser settings
- **PRO-3-REQ-11**: Browser shall scan custom handlers for known security vulnerabilities
- **PRO-3-REQ-12**: All protocol handler security exceptions shall be logged and auditable

5.7 Core Component Security

5.7.1 System Resource Access **[SYS]** The manufacturer shall implement secure boundaries between web content and system resources through Hardware Abstraction Layers and API mediation.

Capability: HAL implementation, PWA support, accessibility features, and local storage mechanisms

Conditions:

- SYS-0: Fully sandboxed with no system access
- SYS-1: Limited system access with strict permissions
- SYS-2: Extended system access for PWAs
- SYS-3: Native-equivalent system access

Threats:

- Sandbox escape attacks
- Unauthorized file system access

- Hardware fingerprinting
- Resource exhaustion attacks
- Accessibility feature abuse for screen reading

Risk: CRITICAL - System access breaches can compromise the entire host environment

Requirements:

SYS-0 Requirements (Fully sandboxed, no system access)

- **SYS-0-REQ-1**: Browser shall enforce process-level sandboxing for all web content → Assessment: SYS-REQ-1
- **SYS-0-REQ-2**: Browser shall isolate renderer processes from each other and browser core → Assessment: SYS-REQ-2
- **SYS-0-REQ-3**: Browser shall isolate GPU rendering in separate sandboxed process → Assessment: SYS-REQ-3
- **SYS-0-REQ-4**: Browser shall isolate network operations in separate process or service → Assessment: SYS-REQ-4
- **SYS-0-REQ-5**: Browser shall block all filesystem access from web content → Assessment: SYS-REQ-5
- **SYS-0-REQ-6**: Browser shall block all device API access (no camera, microphone, location, etc.)
- **SYS-0-REQ-7**: Browser shall enforce strict hardware resource limits → Assessment: SYS-REQ-20
- **SYS-0-REQ-8**: Browser shall enforce memory isolation between processes → Assessment: SYS-REQ-21
- **SYS-0-REQ-9**: Browser shall implement sandbox escape prevention mechanisms → Assessment: SYS-REQ-26
- **SYS-0-REQ-10**: Browser shall implement Spectre/Meltdown mitigations → Assessment: SYS-REQ-27
- **SYS-0-REQ-11**: Browser shall implement side-channel attack mitigations → Assessment: SYS-REQ-28
- **SYS-0-REQ-12**: Browser shall not expose any native messaging interfaces
- **SYS-0-REQ-13**: Browser shall block all host OS integration features

SYS-1 Requirements (Limited system access with strict permissions)

- **SYS-1-REQ-1**: Browser shall enforce process-level sandboxing → Assessment: SYS-REQ-1
- **SYS-1-REQ-2**: Browser shall isolate renderer processes → Assessment: SYS-REQ-2
- **SYS-1-REQ-3**: Browser shall isolate GPU process → Assessment: SYS-REQ-3
- **SYS-1-REQ-4**: Browser shall isolate network service → Assessment: SYS-REQ-4
- **SYS-1-REQ-5**: Browser shall enforce strict filesystem access controls → Assessment: SYS-REQ-5
- **SYS-1-REQ-6**: Browser shall implement permission controls for device hardware APIs → Assessment: SYS-REQ-6
- **SYS-1-REQ-7**: Browser shall enforce geolocation permission requirements → Assessment: SYS-REQ-8
- **SYS-1-REQ-8**: Browser shall enforce camera/microphone access controls with visible indicators → Assessment: SYS-REQ-9
- **SYS-1-REQ-9**: Browser shall restrict clipboard access to require user interaction → Assessment: SYS-REQ-10
- **SYS-1-REQ-10**: Browser shall enforce notification permission management → Assessment: SYS-REQ-11
- **SYS-1-REQ-11**: Browser shall restrict Sensor API access with permissions → Assessment: SYS-REQ-18
- **SYS-1-REQ-12**: Browser shall restrict Battery Status API to prevent fingerprinting → Assessment: SYS-REQ-19
- **SYS-1-REQ-13**: Browser shall enforce hardware resource limits → Assessment: SYS-REQ-20
- **SYS-1-REQ-14**: Browser shall enforce memory isolation → Assessment: SYS-REQ-21
- **SYS-1-REQ-15**: Browser shall implement CPU resource quotas → Assessment: SYS-REQ-22
- **SYS-1-REQ-16**: Browser shall enforce network bandwidth limits → Assessment: SYS-REQ-23
- **SYS-1-REQ-17**: Browser shall enforce storage quota limits → Assessment: SYS-REQ-24
- **SYS-1-REQ-18**: Browser shall implement process priority management → Assessment: SYS-REQ-25
- **SYS-1-REQ-19**: Browser shall prevent sandbox escapes → Assessment: SYS-REQ-26

- **SYS-1-REQ-20**: Browser shall implement speculative execution mitigations → Assessment: SYS-REQ-27
- **SYS-1-REQ-21**: Browser shall implement side-channel mitigations → Assessment: SYS-REQ-28
- **SYS-1-REQ-22**: Browser shall implement accessibility API security controls → Assessment: SYS-REQ-30

SYS-2 Requirements (Extended system access for PWAs)

- **SYS-2-REQ-1**: Browser shall implement all SYS-1 requirements
- **SYS-2-REQ-2**: Browser shall enforce equivalent permission controls for PWAs → Assessment: SYS-REQ-7
- **SYS-2-REQ-3**: Browser shall implement USB device access security → Assessment: SYS-REQ-12
- **SYS-2-REQ-4**: Browser shall enforce Bluetooth permission controls → Assessment: SYS-REQ-13
- **SYS-2-REQ-5**: Browser shall implement File System Access API security → Assessment: SYS-REQ-14
- **SYS-2-REQ-6**: Browser shall enforce WebUSB security controls → Assessment: SYS-REQ-15
- **SYS-2-REQ-7**: Browser shall enforce WebBluetooth security → Assessment: SYS-REQ-16
- **SYS-2-REQ-8**: Browser shall implement WebNFC permission management → Assessment: SYS-REQ-17
- **SYS-2-REQ-9**: Browser shall implement hardware token security → Assessment: SYS-REQ-29
- **SYS-2-REQ-10**: Browser shall enforce restricted native messaging with security controls → Assessment: SYS-REQ-31
- **SYS-2-REQ-11**: PWA permissions shall not exceed web context permissions
- **SYS-2-REQ-12**: PWA installation shall not auto-grant extended permissions
- **SYS-2-REQ-13**: PWA uninstallation shall revoke all granted permissions
- **SYS-2-REQ-14**: Browser shall maintain audit log of all PWA permission grants and revocations
- **SYS-2-REQ-15**: Enterprise administrators shall be able to configure PWA permission policies

SYS-3 Requirements (Native-equivalent system access)

- **SYS-3-REQ-1**: Browser shall implement baseline sandboxing for renderer processes → Assessment: SYS-REQ-1
- **SYS-3-REQ-2**: Browser shall enforce process isolation → Assessment: SYS-REQ-2
- **SYS-3-REQ-3**: Browser shall implement baseline permission controls for device APIs → Assessment: SYS-REQ-6
- **SYS-3-REQ-4**: Browser shall enforce filesystem access controls → Assessment: SYS-REQ-5
- **SYS-3-REQ-5**: Browser shall implement resource limits to prevent exhaustion → Assessment: SYS-REQ-20
- **SYS-3-REQ-6**: Browser shall implement memory isolation → Assessment: SYS-REQ-21
- **SYS-3-REQ-7**: Browser shall implement CPU quotas → Assessment: SYS-REQ-22
- **SYS-3-REQ-8**: Browser shall enforce storage quotas → Assessment: SYS-REQ-24
- **SYS-3-REQ-9**: Browser shall implement speculative execution mitigations → Assessment: SYS-REQ-27
- **SYS-3-REQ-10**: Browser shall implement side-channel mitigations → Assessment: SYS-REQ-28
- **SYS-3-REQ-11**: Browser shall enforce host OS integration security → Assessment: SYS-REQ-32
- **SYS-3-REQ-12**: Browser shall allow native-equivalent API access with user consent
- **SYS-3-REQ-13**: Browser shall provide transparent UI for all native integration features
- **SYS-3-REQ-14**: Browser shall display security warnings for privileged API access
- **SYS-3-REQ-15**: Browser shall log all extended system access for security auditing
- **SYS-3-REQ-16**: Users shall be able to review and revoke all system permissions
- **SYS-3-REQ-17**: Enterprise administrators shall be able to restrict native-equivalent features
- **SYS-3-REQ-18**: All native integration exceptions shall be documented and auditable

5.8 Embedded Browser Security

5.8.1 Overview

Embedded browsers (WebView components, browser engines integrated into native applications) present unique security challenges distinct from standalone browsers. While standalone browsers operate as independent applications with their own security boundaries, embedded browsers exist within a host application context where they navigate the security boundary between web content and the host application.

Key Security Challenges:

1. **Host-Code Injection**: Malicious web content may attempt to inject code into or manipulate the host application through JavaScript bridge interfaces, custom URL scheme handlers, or exploitation of WebView API vulnerabilities.
2. **User Data Exfiltration**: Web content loaded in an embedded browser may access sensitive data from the host application through insecure bridge configurations, shared storage, or insufficient isolation between web and native contexts.
3. **Insufficient Isolation**: Unlike standalone browsers where all web content is untrusted, embedded browsers often establish trust relationships with certain content sources while maintaining security boundaries, creating complex policy enforcement challenges.

Risk Classification: CRITICAL - Vulnerabilities in embedded browser security can lead to complete compromise of the host application and exfiltration of all user data accessible to the host.

Applicable Use Cases: UC-B08 (Embedded Browser Component), UC-B09 (WebView Component in Native Applications), and aspects of UC-B10 (Adapted Browser with Modified Features) when the adaptation includes native component integration.

References:

- OWASP Mobile Security Testing Guide - WebViews: https://mas.owasp.org/MASTG/tests/android/MASVS-PLATFORM/MASTG-TEST-0028/
- Android WebView Security Best Practices: https://developer.android.com/develop/ui/views/layout/webapps/webview
- iOS WKWebView Security: https://developer.apple.com/documentation/webkit/wkwebview
- Chromium Embedded Framework Security: https://bitbucket.org/chromiumembedded/cef/wiki/GeneralUsage#markdow header-security
- Electron Security Best Practices: https://www.electronjs.org/docs/latest/tutorial/security

5.8.2 Host Application Boundary Security

[**EMB**] The manufacturer shall implement secure isolation boundaries between embedded browser content and host application code, data, and resources.

Capability: JavaScript bridge security, native API exposure control, host data protection, and context isolation

Conditions:

- **EMB-0**: No JavaScript bridge or native API exposure (isolated WebView)
- **EMB-1**: Limited JavaScript bridge with explicit allowlist of safe APIs
- **EMB-2**: Extended JavaScript bridge with bidirectional communication
- **EMB-3**: Full integration with access to native capabilities and host data

Threats:

- JavaScript injection into host application context
- Unauthorized access to native APIs through bridge exploitation
- Code execution in host process via WebView vulnerabilities
- Cross-context data leakage between web and native layers
- Exploitation of insecure JavaScript bridge configurations

- Bypass of content security policies via native bridges
- Host application credential theft via bridge access
- Native code injection via crafted web content

Risk: CRITICAL - Compromise of the host-browser boundary can lead to complete application takeover and data exfiltration

Requirements:

EMB-0 Requirements (No JavaScript bridge or native API exposure)

- **EMB-0-REQ-1**: Embedded browser shall implement complete context isolation → Assessment: EMB-REQ-4
- **EMB-0-REQ-2**: Embedded browser shall isolate all storage from host application → Assessment: EMB-REQ-12
- **EMB-0-REQ-3**: Embedded browser shall enforce CSP for all web content → Assessment: EMB-REQ-13
- **EMB-0-REQ-4**: Embedded browser shall prevent all web content access to host application objects
- **EMB-0-REQ-5**: No JavaScript bridge or native API exposure mechanism shall exist
- **EMB-0-REQ-6**: Embedded browser shall treat all loaded content as completely untrusted
- **EMB-0-REQ-7**: Host application credentials shall be completely isolated from web context → Assessment: EMB-REQ-8

EMB-1 Requirements (Limited JavaScript bridge with explicit allowlist)

- **EMB-1-REQ-1**: JavaScript bridge shall implement explicit API allowlists with per-API access controls → Assessment: EMB-REQ-1
- **EMB-1-REQ-2**: All bridge data shall be validated, sanitized, and type-checked on native side → Assessment: EMB-REQ-2
- **EMB-1-REQ-3**: JavaScript bridge communications shall be logged → Assessment: EMB-REQ-3
- **EMB-1-REQ-4**: Embedded browser shall implement context isolation → Assessment: EMB-REQ-4
- **EMB-1-REQ-5**: Sensitive native APIs shall require explicit user consent → Assessment: EMB-REQ-5
- **EMB-1-REQ-6**: System-level APIs shall not be exposed without additional security controls → Assessment: EMB-REQ-6
- **EMB-1-REQ-7**: JavaScript bridge configuration shall be immutable after initialization → Assessment: EMB-REQ-7
- **EMB-1-REQ-8**: Web content shall not access host credentials, tokens, or keys → Assessment: EMB-REQ-8
- **EMB-1-REQ-9**: Bridge implementations shall be reviewed for injection vulnerabilities → Assessment: EMB-REQ-9
- **EMB-1-REQ-10**: Host shall implement rate limiting on bridge API calls → Assessment: EMB-REQ-10
- **EMB-1-REQ-11**: Bridge shall support granular capability-based permissions → Assessment: EMB-REQ-11
- **EMB-1-REQ-12**: Embedded browser shall isolate storage from host → Assessment: EMB-REQ-12
- **EMB-1-REQ-13**: Host shall enforce CSP for all embedded content → Assessment: EMB-REQ-13
- **EMB-1-REQ-14**: Web content shall not trigger misleading native UI → Assessment: EMB-REQ-15
- **EMB-1-REQ-15**: Host shall implement allowlists rather than denylists for bridge APIs → Assessment: EMB-REQ-16
- **EMB-1-REQ-16**: Only explicitly allowlisted APIs shall be accessible from web content
- **EMB-1-REQ-17**: Bridge API allowlist shall be minimal and documented

EMB-2 Requirements (Extended JavaScript bridge with bidirectional communication)

- **EMB-2-REQ-1**: All EMB-1 requirements shall be implemented
- **EMB-2-REQ-2**: Bridge communications crossing process boundaries shall be encrypted → Assessment: EMB-REQ-14

- **EMB-2-REQ-3**: Bridge shall implement granular permissions → Assessment: EMB-REQ-11
- **EMB-2-REQ-4**: Host shall implement comprehensive rate limiting → Assessment: EMB-REQ-10
- **EMB-2-REQ-5**: All bridge communications shall be logged with full audit trails → Assessment: EMB-REQ-3
- **EMB-2-REQ-6**: Bidirectional bridge calls shall maintain same security controls in both directions
- **EMB-2-REQ-7**: Web content callback handlers shall be validated before invocation
- **EMB-2-REQ-8**: Bridge shall implement message queuing with integrity protection
- **EMB-2-REQ-9**: Host shall monitor bridge traffic for anomalies
- **EMB-2-REQ-10**: Enterprise administrators shall be able to configure bridge API policies

EMB-3 Requirements (Full integration with native capabilities)

- **EMB-3-REQ-1**: Baseline EMB-1 security controls shall be maintained
- **EMB-3-REQ-2**: Bridge shall implement comprehensive input validation → Assessment: EMB-REQ-2
- **EMB-3-REQ-3**: Bridge shall log all operations with security context → Assessment: EMB-REQ-3
- **EMB-3-REQ-4**: Sensitive native operations shall require user consent → Assessment: EMB-REQ-5
- **EMB-3-REQ-5**: System-level APIs shall have strict additional controls → Assessment: EMB-REQ-6
- **EMB-3-REQ-6**: Bridge implementations shall undergo security review → Assessment: EMB-REQ-9
- **EMB-3-REQ-7**: Rate limiting shall prevent bridge API abuse → Assessment: EMB-REQ-10
- **EMB-3-REQ-8**: Full integration shall not bypass core security boundaries
- **EMB-3-REQ-9**: User shall be informed of all native capabilities granted to web content
- **EMB-3-REQ-10**: User shall be able to review and revoke native API access
- **EMB-3-REQ-11**: All native integrations shall be documented and auditable
- **EMB-3-REQ-12**: Enterprise policies shall be able to restrict native integration scope

5.8.3 Content Source Trust Management

[**EMB**] The manufacturer shall implement mechanisms to establish and enforce trust relationships between embedded browser content sources and the host application.

Capability: Content source validation, certificate pinning for embedded content, subresource integrity, and trust boundary enforcement

Conditions:

- **EMB-0**: All content treated as untrusted (public internet)
- **EMB-1**: Trusted domains with certificate validation
- **EMB-2**: Certificate pinning for specific trusted origins
- **EMB-3**: Local/bundled content with cryptographic verification

Threats:

- Man-in-the-middle attacks against trusted content sources
- Loading of malicious content from compromised trusted domains
- Bypass of trust boundaries through redirect chains
- Subresource substitution attacks
- DNS hijacking of embedded content sources
- Certificate authority compromise affecting trusted origins
- Mixed content attacks (trusted page loading untrusted resources)
- Cache poisoning affecting embedded content

Risk: CRITICAL - Compromise of trusted content sources can lead to injection of malicious code with elevated privileges

Requirements:

EMB-0 Requirements (All content treated as untrusted - public internet)

- **EMB-0-REQ-1**: All content shall be treated as completely untrusted
- **EMB-0-REQ-2**: Embedded browser shall validate SSL/TLS certificates for all remote content → Assessment: EMB-REQ-17
- **EMB-0-REQ-3**: Embedded browser shall prevent all mixed content → Assessment: EMB-REQ-21
- **EMB-0-REQ-4**: Certificate validation failures shall block content loading → Assessment: EMB-REQ-26
- **EMB-0-REQ-5**: Network security configuration shall prevent cleartext traffic → Assessment: EMB-REQ-27
- **EMB-0-REQ-6**: Trust boundary violations shall trigger security events → Assessment: EMB-REQ-32
- **EMB-0-REQ-7**: No content origin shall have privileged access
- **EMB-0-REQ-8**: All CSP policies shall be strictly enforced without exceptions

EMB-1 Requirements (Trusted domains with certificate validation)

- **EMB-1-REQ-1**: Embedded browser shall validate SSL/TLS certificates for all remote content → Assessment: EMB-REQ-17
- **EMB-1-REQ-2**: Host shall implement allowlist of trusted content origins → Assessment: EMB-REQ-18
- **EMB-1-REQ-3**: Embedded browser shall prevent mixed content → Assessment: EMB-REQ-21
- **EMB-1-REQ-4**: Trust decisions shall be logged with full context → Assessment: EMB-REQ-22
- **EMB-1-REQ-5**: Host shall implement redirect chain validation → Assessment: EMB-REQ-24
- **EMB-1-REQ-6**: Embedded browser shall enforce HSTS for trusted origins → Assessment: EMB-REQ-25
- **EMB-1-REQ-7**: Certificate failures shall trigger immediate notification and blocking → Assessment: EMB-REQ-26
- **EMB-1-REQ-8**: Network security config shall prevent cleartext to trusted domains → Assessment: EMB-REQ-27
- **EMB-1-REQ-9**: Trusted content shall not load untrusted third-party content without CSP → Assessment: EMB-REQ-28
- **EMB-1-REQ-10**: Trust boundary violations shall trigger security events → Assessment: EMB-REQ-32
- **EMB-1-REQ-11**: Only explicitly allowlisted origins shall be considered trusted
- **EMB-1-REQ-12**: Trust allowlist shall be immutable by web content

EMB-2 Requirements (Certificate pinning for specific trusted origins)

- **EMB-2-REQ-1**: All EMB-1 requirements shall be implemented
- **EMB-2-REQ-2**: Embedded browser shall implement certificate pinning for critical origins → Assessment: EMB-REQ-17
- **EMB-2-REQ-3**: Embedded browser shall enforce SRI for external scripts from trusted content → Assessment: EMB-REQ-19
- **EMB-2-REQ-4**: Certificate pinning shall include backup pins and rotation mechanisms → Assessment: EMB-REQ-20
- **EMB-2-REQ-5**: Trust policies shall be configurable per browser instance → Assessment: EMB-REQ-29
- **EMB-2-REQ-6**: Embedded browser shall implement certificate transparency verification → Assessment: EMB-REQ-30
- **EMB-2-REQ-7**: Host shall detect and prevent DNS rebinding attacks → Assessment: EMB-REQ-31
- **EMB-2-REQ-8**: Pin configuration shall be immutable after initialization
- **EMB-2-REQ-9**: Pinning violations shall immediately block content loading
- **EMB-2-REQ-10**: Pin rotation procedures shall be documented and tested

EMB-3 Requirements (Local/bundled content with cryptographic verification)

- **EMB-3-REQ-1**: Baseline EMB-1 certificate validation shall apply to all remote content
- **EMB-3-REQ-2**: Embedded browser shall verify cryptographic signatures for local/bundled content → Assessment: EMB-REQ-23

- **EMB-3-REQ-3**: Embedded browser shall enforce SRI for all external scripts → Assessment: EMB-REQ-19
- **EMB-3-REQ-4**: Certificate pinning shall be enforced for remote trusted origins → Assessment: EMB-REQ-17, EMB-REQ-20
- **EMB-3-REQ-5**: Trust decisions shall be logged comprehensively → Assessment: EMB-REQ-22
- **EMB-3-REQ-6**: Trust policies shall be configurable per instance → Assessment: EMB-REQ-29
- **EMB-3-REQ-7**: DNS rebinding prevention shall be enforced → Assessment: EMB-REQ-31
- **EMB-3-REQ-8**: Trust boundary violations shall trigger detailed security events → Assessment: EMB-REQ-32
- **EMB-3-REQ-9**: Local content signature verification shall use secure algorithms (RSA-2048+, ECDSA P-256+)
- **EMB-3-REQ-10**: Modified local content shall fail signature verification and be rejected
- **EMB-3-REQ-11**: Signing keys for local content shall be protected from extraction
- **EMB-3-REQ-12**: Hybrid deployments (local + remote) shall maintain strictest security controls for each content type

References:

- OWASP Mobile Top 10 - M1: Improper Platform Usage: https://owasp.org/www-project-mobile-top-10/
- CWE-749: Exposed Dangerous Method or Function: https://cwe.mitre.org/data/definitions/749.html
- CWE-940: Improper Verification of Source of a Communication Channel: https://cwe.mitre.org/data/definitions/940.html
- Android Network Security Configuration: https://developer.android.com/training/articles/security-config
- iOS App Transport Security: https://developer.apple.com/documentation/security/preventing_insecure_network_conne
- Certificate Pinning Best Practices: https://owasp.org/www-community/controls/Certificate_and_Public_Key_Pinning
- Electron Context Isolation: https://www.electronjs.org/docs/latest/tutorial/context-isolation

6 Technical Security Assessments

This chapter provides detailed technical assessment procedures for verifying conformance with the security requirements specified in Chapter 5. Each assessment follows a structured methodology designed to produce objective, repeatable results suitable for third-party conformity assessment.

6.1 Domain and Origin Isolation Assessments

This section covers assessment procedures for requirements DOM-REQ-1 through DOM-REQ-12, addressing site isolation, origin-based security boundaries, cross-origin resource sharing, and related isolation mechanisms.

Assessment: DOM-REQ-1 (Process-per-site isolation)

Reference: DOM-REQ-1 - Browser shall implement process-per-site isolation

Given: A conformant browser with site isolation capabilities (DOM-1 or higher)

Task: Verify that the browser enforces operating system-level process isolation between distinct sites to prevent compromise of one site from affecting other sites, and to enable OS-level security mechanisms (ASLR, sandboxing, memory protection) to provide defense-in-depth against web-based attacks.

Verification:

1. Open the browser and navigate to three distinct origins: https://example.com, https://test.com, and https://example.org
2. In each origin, open the browser's task manager or use platform process monitoring tools (Process Explorer on Windows, Activity Monitor on macOS, ps/top on Linux)
3. Identify the renderer processes associated with each origin
4. Record the Process IDs (PIDs) for each origin's renderer process

5. Navigate from https://example.com to https://example.com/page2 and verify the PID remains the same
6. Navigate from https://example.com to https://different.example.com (different subdomain, same site) and verify the PID remains the same or changes according to documented site isolation policy
7. Create cross-origin iframes (example.com embedding test.com) and verify they run in separate processes
8. Confirm that each distinct site (eTLD+1) is allocated a separate renderer process
9. Verify that same-site navigations reuse the same renderer process
10. Check that cross-origin iframes are isolated in separate processes from their embedder
11. Validate that process separation is maintained throughout the browser session
12. Review browser task manager or process monitoring tools to confirm distinct processes for distinct sites

Pass Criteria: All distinct sites use separate renderer processes AND cross-origin iframes are isolated in separate processes

Fail Criteria: Any two distinct sites share a renderer process OR cross-origin iframe runs in the same process as embedder

Evidence: Screenshots of browser task manager showing distinct PIDs for distinct sites, process tree diagrams, video recordings of process creation during navigation

References:

- Chromium Site Isolation Architecture: https://www.chromium.org/Home/chromium-security/site-isolation/
- Process Models for Web Browsers: https://www.chromium.org/developers/design-documents/process-models/
- Firefox Project Fission: https://wiki.mozilla.org/Project_Fission

Assessment: DOM-REQ-2 (Cross-origin read blocking)

Reference: DOM-REQ-2 - Browser shall enforce Cross-Origin Read Blocking (CORB)

Given: A conformant browser with DOM-1 or higher capability

Task: Verify that the browser prevents attackers from using script/image/stylesheet tags to read sensitive cross-origin data (JSON, HTML, XML) that lacks CORS headers, protecting against Spectre-style attacks and cross-site data leakage where malicious pages attempt to load victim resources into contexts that could expose response data through side channels.

Verification:

1. Set up a test web server serving resources with various MIME types (HTML, JSON, XML, images, scripts)
2. Create a test page at https://attacker.com that attempts to load resources from https://victim.com using various methods:
   - `<script src="https://victim.com/sensitive.json">`
   - `<img src="https://victim.com/sensitive.json">`
   - `<link rel="stylesheet" href="https://victim.com/sensitive.html">`
   - `fetch()` requests without CORS headers
3. Configure https://victim.com to serve JSON/HTML/XML resources without CORS headers
4. Monitor network traffic using browser DevTools to capture CORB decisions
5. Verify that cross-origin requests for JSON/HTML/XML are blocked when served with incorrect MIME types
6. Verify that legitimate cross-origin image/script/CSS loads still succeed
7. Test edge cases: nosniff headers, ambiguous MIME types, empty responses
8. Confirm that JSON, HTML, and XML responses without CORS headers are blocked from cross-origin script contexts
9. Verify that browser console shows CORB warning messages with details of blocked resources
10. Check that network panel shows resources blocked with CORB designation

11. Validate that legitimate cross-origin subresources (images, scripts with correct MIME types) load successfully
12. Confirm that `X-Content-Type-Options: nosniff` is respected in CORB decisions

Pass Criteria: All cross-origin HTML/JSON/XML loads without CORS headers are blocked from script contexts AND browser logs CORB blocking events

Fail Criteria: Any JSON/HTML/XML resource is readable cross-origin without CORS headers OR no CORB enforcement is observable

Evidence: Browser console screenshots showing CORB warnings, network panel captures showing blocked resources, packet captures demonstrating data was fetched but not exposed to scripts, test server logs

References:

- CORB Specification: https://chromium.googlesource.com/chromium/src/+/master/services/network/cross_origin_read
- Fetch Metadata Request Headers: https://www.w3.org/TR/fetch-metadata/
- MIME Sniffing Standard: https://mimesniff.spec.whatwg.org/

Assessment: DOM-REQ-3 (Strict origin policy enforcement)

Reference: DOM-REQ-3 - Browser shall prevent cross-origin DOM access without explicit consent

Given: A conformant browser with DOM-0 or higher capability

Task: Verify that the browser enforces the Same-Origin Policy to prevent scripts from one origin from reading or manipulating the DOM of another origin, protecting against cross-site scripting attacks where malicious sites attempt to steal sensitive data or hijack user sessions by accessing cross-origin window objects, documents, or storage.

Verification:

1. Create two test pages: https://site-a.com/test.html and https://site-b.com/test.html
2. From site-a, open site-b in a new window using `window.open()`
3. Attempt cross-origin DOM access from site-a to site-b window:
   - `otherWindow.document` - attempt to access document
   - `otherWindow.localStorage` - attempt to access storage
   - `otherWindow.location.href = "javascript:..."` - attempt navigation hijacking
   - `otherWindow.frames[0]` - attempt to access frames
4. Verify that all cross-origin DOM access attempts throw SecurityError exceptions
5. Test that same-origin window access succeeds
6. Test `window.postMessage()` as the legitimate cross-origin communication channel
7. Verify that `Location` object allows limited cross-origin access (href setter only) but not href getter
8. Confirm that all cross-origin DOM access attempts throw SecurityError exceptions
9. Verify that browser console logs SecurityError with clear origin mismatch messages
10. Check that same-origin window access succeeds without errors
11. Validate that postMessage() works correctly for cross-origin communication
12. Confirm that Location.href can be set cross-origin but not read

Pass Criteria: All cross-origin DOM property access attempts throw SecurityError AND postMessage provides functional cross-origin communication

Fail Criteria: Any cross-origin DOM property is readable/writable OR no exception is thrown

Evidence: Browser console screenshots showing SecurityError exceptions, video demonstration of test execution, automated test results from Web Platform Tests

References:

- HTML Standard - Cross-origin objects: https://html.spec.whatwg.org/multipage/browsers.html#cross-origin-objects

- Same-Origin Policy (MDN): https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy
- Web Platform Tests - Origin isolation: https://github.com/web-platform-tests/wpt/tree/master/html/browsers/origin

Assessment: DOM-REQ-4 (CORS preflight enforcement)

Reference: DOM-REQ-4 - Browser shall enforce CORS preflight for non-simple requests

Given: A conformant browser with DOM-1 or higher capability

Task: Verify that the browser enforces CORS preflight checks for potentially dangerous cross-origin requests (non-simple methods, custom headers) to prevent attackers from triggering unauthorized state-changing operations on victim servers, ensuring that servers have an opportunity to reject requests before they execute and protecting against CSRF-style attacks that bypass simple request restrictions.

Verification:

1. Set up a test server that logs all incoming requests including OPTIONS requests
2. Create test pages that make various fetch() requests to cross-origin servers:
   - Simple requests (GET with simple headers)
   - Non-simple requests (PUT, DELETE, PATCH methods)
   - Requests with custom headers (X-Custom-Header)
   - Requests with credentials (cookies)
3. Monitor network traffic to verify preflight OPTIONS requests are sent before non-simple requests
4. Configure the server to respond with various CORS header combinations:
   - Correct CORS headers (Access-Control-Allow-Origin, Allow-Methods, Allow-Headers)
   - Missing CORS headers
   - Incorrect origin in CORS headers
   - Expired preflight cache (Access-Control-Max-Age: 0)
5. Verify that actual requests only proceed after successful preflight
6. Test preflight caching behavior
7. Confirm that OPTIONS preflight requests are sent before all non-simple cross-origin requests
8. Verify that actual requests only proceed after successful preflight response with matching CORS headers
9. Check that browser blocks requests when preflight fails or returns incorrect headers
10. Validate that preflight responses are cached according to Access-Control-Max-Age
11. Confirm that simple requests proceed without preflight but are still subject to CORS checks on response

Pass Criteria: All non-simple requests are preceded by OPTIONS preflight AND requests fail when preflight response lacks appropriate CORS headers

Fail Criteria: Non-simple requests proceed without preflight OR requests succeed despite missing CORS headers

Evidence: Network panel screenshots showing OPTIONS requests before actual requests, server logs demonstrating preflight sequence, packet captures with timing analysis

References:

- Fetch Standard - CORS protocol: https://fetch.spec.whatwg.org/#http-cors-protocol
- CORS (MDN): https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS
- Preflight request specification: https://fetch.spec.whatwg.org/#cors-preflight-request

Assessment: DOM-REQ-5 (Cookie SameSite attribute enforcement)

Reference: DOM-REQ-5 - Browser shall enforce SameSite cookie attribute

Given: A conformant browser with DOM-1 or higher capability

Task: Verify that the browser enforces SameSite cookie restrictions to prevent Cross-Site Request Forgery (CSRF) attacks where malicious sites trigger authenticated requests to victim applications by controlling

when cookies are sent in cross-site contexts, with Strict preventing all cross-site transmission, Lax allowing safe top-level navigations, and None requiring explicit Secure flag.

Verification:

1. Set up two test domains: https://site-a.com and https://site-b.com
2. Configure site-a to set cookies with various SameSite attributes:
   - `Set-Cookie: session=abc123; SameSite=Strict; Secure`
   - `Set-Cookie: tracking=xyz789; SameSite=Lax; Secure`
   - `Set-Cookie: legacy=old; Secure` (no SameSite)
   - `Set-Cookie: none=test; SameSite=None; Secure`
3. From site-b, perform various cross-site requests to site-a:
   - Top-level navigation (clicking link)
   - Embedded resources (images, iframes)
   - JavaScript fetch() POST request
   - Form submission (GET and POST)
4. Monitor network traffic to verify which cookies are sent in each scenario
5. Test the default SameSite behavior for cookies without explicit attribute (should be Lax)
6. Verify that SameSite=None requires Secure attribute
7. Confirm that SameSite=Strict cookies are never sent in cross-site contexts
8. Verify that SameSite=Lax cookies are sent only in top-level navigation (GET)
9. Check that SameSite=None cookies are sent in all contexts but require Secure attribute
10. Validate that cookies without SameSite attribute default to Lax behavior
11. Confirm that browser rejects SameSite=None cookies without Secure attribute

Pass Criteria: Cookie transmission matches SameSite attribute policy for all test cases AND default behavior is Lax

Fail Criteria: Any cookie is sent in violation of its SameSite policy OR SameSite=None works without Secure

Evidence: Network panel screenshots showing Cookie headers in different contexts, DevTools Application tab showing cookie attributes, test server logs of received cookies

References:

- RFC 6265bis - SameSite Cookies: https://datatracker.ietf.org/doc/html/draft-ietf-httpbis-rfc6265bis
- SameSite Cookies Explained (web.dev): https://web.dev/samesite-cookies-explained/
- Cookie SameSite attribute (MDN): https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Set-Cookie/SameSite

Assessment: DOM-REQ-6 (Origin-bound storage isolation)

Reference: DOM-REQ-6 - Browser shall isolate localStorage and IndexedDB per origin

Given: A conformant browser with DOM-0 or higher capability

Task: Verify that the browser enforces complete storage isolation between origins to prevent malicious sites from reading sensitive user data (session tokens, personal information) stored by other applications, ensuring that origin boundaries (scheme, host, port) create impermeable barriers for localStorage and IndexedDB access.

Verification:

1. Create test pages at multiple origins: https://example.com, https://example.org, http://example.com, https://sub.example.com
2. In each origin, write distinct data to localStorage:

```
localStorage.setItem('origin-test', location.origin);
localStorage.setItem('timestamp', Date.now());
```

3. In each origin, create an IndexedDB database with a distinct name and store origin-specific data
4. Attempt to read localStorage and IndexedDB from each origin
5. Verify that each origin only sees its own storage
6. Test subdomain isolation (example.com vs sub.example.com)
7. Test protocol isolation (https vs http)
8. Test port isolation (example.com:443 vs example.com:8080)
9. Clear storage for one origin and verify other origins' storage is unaffected
10. Confirm that each origin has completely isolated localStorage namespace
11. Verify that each origin has completely isolated IndexedDB namespace
12. Check that different subdomains cannot access each other's storage
13. Validate that different protocols (http vs https) have separate storage
14. Confirm that different ports have separate storage
15. Verify that storage clearing is scoped to single origin

Pass Criteria: No origin can read another origin's localStorage or IndexedDB AND all origin components (scheme, host, port) contribute to isolation boundary

Fail Criteria: Any cross-origin storage access succeeds OR incomplete origin matching (e.g., ignoring port)

Evidence: Browser DevTools Application tab screenshots showing storage contents per origin, console logs demonstrating isolation, automated test results

References:

- Web Storage API specification: https://html.spec.whatwg.org/multipage/webstorage.html
- IndexedDB API specification: https://w3c.github.io/IndexedDB/
- Origin definition: https://html.spec.whatwg.org/multipage/origin.html#concept-origin

Assessment: DOM-REQ-7 (Frame sandboxing support)

Reference: DOM-REQ-7 - Browser shall support iframe sandbox attribute

Given: A conformant browser with DOM-1 or higher capability

Task: Verify that the browser implements iframe sandbox restrictions to mitigate risks from untrusted content by allowing developers to apply least-privilege principles to embedded frames, preventing malicious iframes from executing scripts, accessing parent windows, navigating the top frame, or abusing other dangerous capabilities unless explicitly permitted.

Verification:

1. Create test pages with iframes using various sandbox configurations:

```
<iframe sandbox src="test.html"></iframe>
<iframe sandbox="allow-scripts" src="test.html"></iframe>
<iframe sandbox="allow-scripts allow-same-origin" src="test.html"></iframe>
<iframe sandbox="allow-forms allow-popups" src="test.html"></iframe>
```

2. In each sandboxed iframe, attempt various actions:
   - JavaScript execution (alert, console.log)
   - Form submission
   - Opening popups (window.open)
   - Accessing parent window

- Accessing localStorage
- Top navigation (top.location = ...)

3. Verify that only explicitly allowed capabilities work
4. Test that sandbox="" (empty) applies strictest restrictions
5. Test CSP sandbox directive equivalence
6. Verify unique origin treatment for sandboxed iframes without allow-same-origin
7. Confirm that sandbox attribute restricts capabilities according to specified tokens
8. Verify that scripts are blocked unless allow-scripts is present
9. Check that same-origin access is blocked unless allow-same-origin is present
10. Validate that sandboxed frames without allow-same-origin have unique opaque origin
11. Confirm that browser console logs security errors for blocked actions
12. Verify that CSP sandbox directive provides equivalent restrictions

Pass Criteria: All tested restrictions are enforced according to sandbox tokens AND browser logs security errors for blocked actions

Fail Criteria: Any capability works without corresponding allow-* token OR no restrictions are observed

Evidence: Browser console screenshots showing blocked actions, DevTools showing unique origin for sandboxed frames, test results demonstrating each sandbox token

References:

- HTML Standard - Sandboxing: https://html.spec.whatwg.org/multipage/iframe-embed-object.html#attr-iframe-sandbox
- CSP sandbox directive: https://www.w3.org/TR/CSP3/#directive-sandbox
- iframe sandbox (MDN): https://developer.mozilla.org/en-US/docs/Web/HTML/Element/iframe#attr-sandbox

Assessment: DOM-REQ-8 (Opaque origin handling)

Reference: DOM-REQ-8 - Browser shall treat sandboxed and data: origins as opaque

Given: A conformant browser with DOM-0 or higher capability

Task: Verify that the browser treats sandboxed iframes and data: URLs as having opaque (unique, unguessable) origins that cannot access storage or credentials, preventing untrusted content from stealing sensitive data or establishing persistent state, while ensuring that each opaque origin is internally unique to prevent two untrusted contexts from communicating even though both serialize as "null".

Verification:

1. Create test scenarios for opaque origins:
   - Sandboxed iframe without allow-same-origin: `<iframe sandbox="allow-scripts" src="...">`
   - data: URL navigation: `window.open('data:text/html,<h1>Test</h1>')`
   - Blob URL: `URL.createObjectURL(new Blob(['...'], {type: 'text/html'}))`
2. In each opaque origin context, attempt to:
   - Access localStorage/sessionStorage (should throw SecurityError)
   - Access IndexedDB (should throw SecurityError)
   - Make fetch() requests (should succeed but not send credentials)
   - Access parent window (should be blocked for sandboxed frames)
3. Verify that opaque origins serialize as "null" in `window.origin`
4. Test that two distinct opaque origins cannot access each other even though both serialize as "null"
5. Verify that cookies are not sent/received from opaque origins

6. Confirm that opaque origins report window.origin as “null”
7. Verify that opaque origins cannot access localStorage, sessionStorage, or IndexedDB
8. Check that opaque origins do not send or receive cookies
9. Validate that each opaque origin is unique and cannot access other opaque origins
10. Confirm that fetch requests from opaque origins work but without credentials

Pass Criteria: All storage APIs throw SecurityError in opaque origins AND window.origin reports “null” AND cookies are not sent

Fail Criteria: Any storage access succeeds from opaque origin OR cookies are sent/received

Evidence: Console screenshots showing SecurityError exceptions, network captures showing missing cookies, DevTools Application tab showing empty storage

References:

- HTML Standard - Opaque origins: https://html.spec.whatwg.org/multipage/origin.html#concept-origin-opaque
- data: URL security: https://datatracker.ietf.org/doc/html/rfc2397
- Fetch - CORS and credentials: https://fetch.spec.whatwg.org/#http-cors-protocol

Assessment: DOM-REQ-9 (CORP for cross-origin isolation)

Reference: DOM-REQ-9 - Browser shall support Cross-Origin-Resource-Policy header

Given: A conformant browser with DOM-2 or higher capability

Task: Verify that the browser enforces Cross-Origin-Resource-Policy (CORP) headers to allow servers to protect their resources from being loaded by cross-origin pages, defending against Spectre-style attacks where malicious sites embed victim resources to leak data through side channels, and enabling servers to opt into stronger isolation guarantees for sensitive content.

Verification:

1. Set up test servers on multiple origins (site-a.com, site-b.com, cdn.example.com)
2. Configure site-a to serve resources with various CORP headers:
   - `Cross-Origin-Resource-Policy: same-origin`
   - `Cross-Origin-Resource-Policy: same-site`
   - `Cross-Origin-Resource-Policy: cross-origin`
3. From site-b, attempt to load resources from site-a:

```
<img src="https://site-a.com/image-same-origin.png">
<script src="https://site-a.com/script-same-site.js"></script>
<iframe src="https://site-a.com/frame-cross-origin.html"></iframe>
```

4. Verify blocking behavior based on CORP header
5. Test CORP interaction with CORS headers
6. Verify that CORP applies to all resource types (images, scripts, frames, fetch)
7. Test CORP enforcement in cross-origin isolated contexts (COOP+COEP)
8. Confirm that resources with CORP: same-origin are blocked from cross-origin loads
9. Verify that resources with CORP: same-site are blocked from cross-site loads but allowed same-site
10. Check that resources with CORP: cross-origin load from any origin
11. Validate that browser console shows CORP blocking errors with clear messages
12. Confirm that CORP is enforced for all resource types

13. Verify that CORP is enforced even for opaque responses (no-cors mode)

Pass Criteria: All CORP policies are enforced according to specification AND browser logs blocking errors

Fail Criteria: Any resource loads in violation of its CORP header OR no CORP enforcement is observable

Evidence: Network panel showing blocked resources, console screenshots showing CORP errors, test results demonstrating same-origin/same-site/cross-origin behavior

References:

- CORP Specification: https://fetch.spec.whatwg.org/#cross-origin-resource-policy-header
- Cross-Origin Isolation guide: https://web.dev/coop-coep/
- CORP (MDN): https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Cross-Origin-Resource-Policy

Assessment: DOM-REQ-10 (COOP enforcement)

Reference: DOM-REQ-10 - Browser shall enforce Cross-Origin-Opener-Policy

Given: A conformant browser with DOM-2 or higher capability

Task: Verify that the browser enforces Cross-Origin-Opener-Policy (COOP) to prevent cross-origin documents from sharing browsing context groups and accessing each other through window.opener references, protecting against Spectre-style attacks by enabling process isolation and allowing sites to opt into cross-origin isolation that grants access to powerful features like SharedArrayBuffer.

Verification:

1. Create test pages with various COOP headers:
   - `Cross-Origin-Opener-Policy: same-origin`
   - `Cross-Origin-Opener-Policy: same-origin-allow-popups`
   - `Cross-Origin-Opener-Policy: unsafe-none` (default)
2. Test window.opener relationships:
   - Page A (COOP: same-origin) opens Page B (no COOP) → opener should be null
   - Page A (no COOP) opens Page B (COOP: same-origin) → opener should be null
   - Page A (COOP: same-origin) opens Page B (same-origin with COOP) → opener should work
3. Verify browsing context group isolation
4. Test that cross-origin-isolated pages cannot be in the same browsing context group as non-isolated pages
5. Verify SharedArrayBuffer availability in cross-origin isolated contexts
6. Test COOP reporting endpoint functionality
7. Confirm that COOP: same-origin severs opener relationship with cross-origin pages
8. Verify that COOP: same-origin-allow-popups preserves opener for popups but not navigations
9. Check that cross-origin isolated pages (COOP + COEP) get access to high-resolution timers and SharedArrayBuffer
10. Validate that browser process allocation reflects browsing context group isolation
11. Confirm that violation reports are sent to reporting endpoint when configured

Pass Criteria: Opener relationship is severed as specified by COOP policy AND cross-origin isolation enables SharedArrayBuffer

Fail Criteria: Opener relationship persists in violation of COOP policy OR SharedArrayBuffer unavailable in properly isolated context

Evidence: Console logs showing null window.opener, DevTools showing browsing context groups, demonstration of SharedArrayBuffer availability, network captures of violation reports

References:

- COOP Specification: https://html.spec.whatwg.org/multipage/origin.html#cross-origin-opener-policies

- Cross-Origin Isolation guide: https://web.dev/coop-coep/
- SharedArrayBuffer and cross-origin isolation: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Glob

Assessment: DOM-REQ-11 (COEP enforcement)

Reference: DOM-REQ-11 - Browser shall enforce Cross-Origin-Embedder-Policy

Given: A conformant browser with DOM-2 or higher capability

Task: Verify that the browser enforces Cross-Origin-Embedder-Policy (COEP) to ensure that all cross-origin resources loaded by a document have explicitly opted in via CORP or CORS headers, preventing the document from inadvertently loading attacker-controlled resources that could be used in Spectre-style side-channel attacks, and enabling cross-origin isolation when combined with COOP.

Verification:

1. Create a test page with COEP header: `Cross-Origin-Embedder-Policy: require-corp`
2. From this page, attempt to load various cross-origin resources:
   - Image without CORP/CORS: `<img src="https://cross-origin.com/image.png">`
   - Image with CORP: `<img src="https://cross-origin.com/image-with-corp.png">` (CORP: cross-origin)
   - Script without CORS: `<script src="https://cross-origin.com/script.js">`
   - Script with CORS: `<script src="..." crossorigin="anonymous">` (with proper CORS headers)
   - Iframe without COEP: `<iframe src="https://cross-origin.com/page.html">`
   - Iframe with COEP: `<iframe src="...">` (page has COEP header)
3. Verify blocking of resources without CORP/CORS
4. Test COEP: credentialless as alternative to require-corp
5. Verify that combining COOP + COEP enables cross-origin isolation
6. Test COEP violation reporting
7. Confirm that resources without CORP or CORS are blocked when page has COEP: require-corp
8. Verify that resources with CORP: cross-origin load successfully
9. Check that resources with CORS and crossorigin attribute load successfully
10. Validate that iframes without COEP are blocked from embedding in COEP page
11. Confirm that browser console shows COEP blocking errors
12. Verify that COOP + COEP combination enables self.crossOriginIsolated === true

Pass Criteria: All resources without CORP/CORS are blocked AND cross-origin isolation is achieved with COOP+COEP

Fail Criteria: Any resource without CORP/CORS loads successfully OR cross-origin isolation not achieved despite proper headers

Evidence: Network panel showing blocked resources, console screenshots showing COEP errors, JavaScript console showing self.crossOriginIsolated === true

References:

- COEP Specification: https://html.spec.whatwg.org/multipage/origin.html#coep
- Cross-Origin Isolation: https://web.dev/coop-coep/
- COEP (MDN): https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Cross-Origin-Embedder-Policy

Assessment: DOM-REQ-12 (Document.domain deprecation)

Reference: DOM-REQ-12 - Browser shall restrict or remove document.domain setter

Given: A conformant browser with DOM-1 or higher capability

Task: Verify that the browser restricts or removes the document.domain setter to eliminate a legacy same-origin policy relaxation mechanism that undermines site isolation, prevents origin-keyed agent clustering for performance, and creates security risks by allowing subdomains to arbitrarily merge their security boundaries, requiring sites to explicitly opt-out of modern isolation to use this deprecated feature.

Verification:

1. Create test pages on related subdomains: https://sub1.example.com and https://sub2.example.com
2. Attempt to relax same-origin policy using document.domain:

   ```
   // On both sub1.example.com and sub2.example.com
   document.domain = 'example.com';
   ```

3. Verify that document.domain setter is either:
   - Removed entirely (throws error or no-op)
   - Gated behind Origin-Agent-Cluster: ?0 header
4. Test that pages with `Origin-Agent-Cluster: ?1` (default) cannot set document.domain
5. Verify that failing to set document.domain prevents cross-subdomain DOM access
6. Test browser console warnings about document.domain deprecation
7. Confirm that document.domain setter is unavailable or no-op by default
8. Verify that pages can only set document.domain if they explicitly opt-out via Origin-Agent-Cluster: ?0
9. Check that browser console shows deprecation warnings when document.domain is accessed
10. Validate that cross-subdomain access fails when document.domain is disabled
11. Confirm that Origin-Agent-Cluster header controls document.domain availability

Pass Criteria: document.domain is restricted by default (requires explicit opt-out) AND browser shows deprecation warnings

Fail Criteria: document.domain works unconditionally OR no deprecation warnings shown

Evidence: Console screenshots showing errors/warnings, test results showing cross-subdomain access blocked, DevTools showing Origin-Agent-Cluster header processing

References:

- document.domain deprecation: https://developer.chrome.com/blog/immutable-document-domain/
- Origin-Agent-Cluster header: https://html.spec.whatwg.org/multipage/origin.html#origin-agent-cluster
- HTML Standard - Origin-keyed agent clusters: https://html.spec.whatwg.org/multipage/origin.html#origin-keyed-agent-clusters

6.2 Extension System Security Assessments

This section covers assessment procedures for requirements EXT-REQ-1 through EXT-REQ-18, addressing browser extension security including permissions, content script isolation, extension API access control, manifest validation, and extension update security.

Assessment: EXT-REQ-1 (Permission model for extensions)

Reference: EXT-REQ-1 - Browser shall implement a permission model that restricts extension capabilities to explicitly declared permissions

Given: A conformant browser with EXT-1 or higher capability

Task: Verify that the browser implements a least-privilege permission model for extensions to prevent malicious or compromised extensions from accessing sensitive APIs and user data beyond their declared functionality, protecting users from over-privileged extensions that could steal credentials, intercept network traffic, or exfiltrate browsing history without explicit user consent.

Verification:

1. Create a test extension with a minimal manifest.json declaring only basic permissions (e.g., "storage", "tabs")
2. Attempt to use APIs that require undeclared permissions:
   - chrome.cookies API (requires "cookies" permission)
   - chrome.webRequest API (requires "webRequest" permission)
   - chrome.downloads API (requires "downloads" permission)
   - Access to specific host patterns not declared in host_permissions
3. Verify that the browser blocks API access and throws exceptions for undeclared permissions
4. Monitor browser console for permission-related error messages
5. Add the required permissions to manifest.json and reload the extension
6. Verify that previously blocked APIs now function correctly
7. Test that permission requests at install time accurately reflect manifest declarations
8. Verify that extensions cannot dynamically request permissions not declared in optional_permissions
9. Test that host permissions are enforced for content script injection and webRequest interception
10. Confirm that extensions cannot access APIs without corresponding manifest permissions
11. Verify that browser throws clear error messages when undeclared APIs are accessed (e.g., "Cannot access chrome.cookies without 'cookies' permission")
12. Check that permission prompts at install time accurately reflect all requested permissions
13. Validate that host permissions restrict content script injection to declared patterns
14. Confirm that optional permissions can only be requested if declared in manifest
15. Verify that runtime permission requests are properly gated by manifest declarations

Pass Criteria: All API access is blocked when permissions are not declared AND clear error messages are shown AND permission grants are persistent across sessions

Fail Criteria: Any API access succeeds without declared permission OR permission system can be bypassed OR no error messages are shown

Evidence: Console screenshots showing permission errors, test results demonstrating API blocking, DevTools Extension panel showing active permissions, permission prompt screenshots

References:

- Chrome Extension Permissions: https://developer.chrome.com/docs/extensions/mv3/declare_permissions/
- Mozilla WebExtensions Permissions: https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/manifest.json/permissions
- Manifest V3 Permissions: https://developer.chrome.com/docs/extensions/mv3/permission_warnings/
- CWE-250: Execution with Unnecessary Privileges: https://cwe.mitre.org/data/definitions/250.html
- Optional Permissions API: https://developer.chrome.com/docs/extensions/reference/permissions/
- WebExtensions Security Best Practices: https://extensionworkshop.com/documentation/develop/build-a-secure-extension/

Assessment: EXT-REQ-2 (Content script isolation)

Reference: EXT-REQ-2 - Browser shall isolate content scripts from web page JavaScript contexts

Given: A conformant browser with EXT-1 or higher capability

Task: Verify that the browser isolates content scripts in a separate JavaScript execution world from the host web page to prevent malicious pages from stealing extension secrets, intercepting extension message passing, or using prototype pollution to compromise the extension's security, while allowing content scripts to safely manipulate the DOM for legitimate functionality.

Verification:

1. Create a test extension with a content script that:
   - Defines a global variable: `var extensionSecret = "sensitive_data"`
   - Attempts to access variables defined by the web page
   - Uses messaging to communicate with the background script
2. Create a test web page that:
   - Defines its own global variables: `var pageVariable = "page_data"`
   - Attempts to access variables defined by the content script
   - Attempts to intercept or modify content script message passing
3. Verify that the content script cannot directly access web page variables and vice versa
4. Test that content scripts run in an isolated JavaScript world with separate global scope
5. Verify that DOM modifications are visible to both contexts but JavaScript objects are not shared
6. Test that the web page cannot intercept chrome.runtime.sendMessage() calls from content scripts
7. Verify that content scripts cannot access page's inline event handlers or Function.prototype modifications
8. Test protection against prototype pollution attacks from page context to content script
9. Confirm that content scripts and web page JavaScript execute in separate JavaScript worlds
10. Verify that variables and functions defined in one context are not accessible from the other
11. Check that content scripts can manipulate the DOM but cannot access JavaScript objects from page context
12. Validate that message passing between content scripts and extension background is not interceptable by web page
13. Confirm that prototype modifications in page context do not affect content script execution
14. Verify that content scripts have access to clean browser APIs unmodified by page JavaScript

Pass Criteria: Complete JavaScript isolation between content script and web page contexts AND secure message passing AND protection from prototype pollution

Fail Criteria: Any JavaScript objects/variables leak between contexts OR message passing can be intercepted OR prototype pollution succeeds

Evidence: Console logs showing undefined variables across contexts, test results demonstrating isolation, browser DevTools showing separate execution contexts, security test results showing prototype pollution protection

References:

- Chrome Content Script Isolated Worlds: https://developer.chrome.com/docs/extensions/mv3/content_scripts/#isolated_
- Mozilla Content Script Execution Environment: https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/Content_scripts#execution_environment
- WebExtensions Content Script Security: https://developer.chrome.com/docs/extensions/mv3/security/
- CWE-501: Trust Boundary Violation: https://cwe.mitre.org/data/definitions/501.html
- Prototype Pollution Prevention: https://portswigger.net/web-security/prototype-pollution
- Content Script Communication: https://developer.chrome.com/docs/extensions/mv3/messaging/

Assessment: EXT-REQ-3 (Extension API access control)

Reference: EXT-REQ-3 - Browser shall enforce access control for sensitive extension APIs based on manifest declarations

Given: A conformant browser with EXT-1 or higher capability

Task: Verify that the browser restricts access to powerful extension APIs (debugger, privacy settings, system information) based on explicit manifest permissions to prevent malicious extensions from escalating privileges, debugging other extensions/tabs to steal data, modifying privacy settings, or accessing system information without user awareness and consent.

Verification:

1. Identify sensitive extension APIs that require specific permissions:
   - chrome.debugger (requires "debugger" permission)
   - chrome.management (requires "management" permission)
   - chrome.privacy (requires "privacy" permission)
   - chrome.system.cpu/memory/storage (requires "system.*" permissions)
   - chrome.desktopCapture (requires "desktopCapture" permission)
2. Create test extensions with varying permission sets
3. Attempt to access each sensitive API without the required permission
4. Verify that access is denied with clear error messages
5. Test that powerful APIs like debugger and management show enhanced warnings during install
6. Verify that certain APIs (e.g., debugger) cannot be used in published extensions on Chrome Web Store
7. Test that API access control cannot be bypassed through indirect means (eval, dynamic code loading)
8. Verify access control is enforced consistently across background scripts, content scripts, and popup contexts
9. Confirm that sensitive APIs are blocked without appropriate permissions
10. Verify that browser throws descriptive errors when API access is denied
11. Check that permission warnings during extension install clearly communicate sensitive capabilities
12. Validate that API access control is enforced uniformly across all extension contexts
13. Confirm that no bypasses exist through code evaluation or dynamic loading
14. Verify that debugger and management APIs show enhanced security warnings

Pass Criteria: All sensitive APIs are access-controlled based on manifest permissions AND appropriate warnings are shown AND no bypass mechanisms exist

Fail Criteria: Any sensitive API accessible without permission OR warnings are insufficient OR bypass mechanisms exist

Evidence: Console screenshots showing API access errors, permission prompt screenshots showing warnings, test results across multiple extension contexts, Chrome Web Store policy enforcement verification

References:

- Chrome Extension API Reference: https://developer.chrome.com/docs/extensions/reference/
- Sensitive Permissions in Chrome: https://developer.chrome.com/docs/extensions/mv3/permission_warnings/#permissio
- Mozilla WebExtensions API: https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/API
- Chrome Debugger API Restrictions: https://developer.chrome.com/docs/extensions/reference/debugger/
- CWE-306: Missing Authentication for Critical Function: https://cwe.mitre.org/data/definitions/306.html

Assessment: EXT-REQ-4 (Manifest validation)

Reference: EXT-REQ-4 - Browser shall validate extension manifests and reject extensions with invalid or malicious manifest declarations

Given: A conformant browser with EXT-1 or higher capability

Task: Verify that the browser strictly validates extension manifest files to prevent installation of malformed or malicious extensions that declare invalid permissions, overly broad access patterns, insecure content security policies, or deprecated features, protecting users from extensions that attempt to bypass security controls through manifest manipulation.

Verification:

1. Create test extensions with various manifest violations:
   - Missing required fields (name, version, manifest_version)
   - Invalid JSON syntax
   - Unsupported manifest_version (e.g., manifest_version: 1)
   - Invalid permission names
   - Malformed host_permissions patterns
   - Content security policy violations

- Invalid web_accessible_resources declarations

2. Attempt to load each malformed extension through chrome://extensions in developer mode
3. Verify that the browser rejects invalid manifests with clear error messages
4. Test manifest schema validation for all fields (permissions, content_scripts, background, etc.)
5. Verify that overly broad host permissions trigger warnings (e.g., , *:///**)
6. Test validation of content_security_policy field for Manifest V3 requirements
7. Verify rejection of deprecated Manifest V2 fields in Manifest V3 extensions
8. Test that manifest changes require extension reload and revalidation
9. Confirm that extensions with invalid manifests are rejected at load time
10. Verify that clear, actionable error messages describe manifest violations
11. Check that manifest schema is strictly enforced for all fields
12. Validate that overly broad permissions trigger user-visible warnings
13. Confirm that Manifest V3 CSP restrictions are enforced (no unsafe-eval, no remote code)
14. Verify that deprecated Manifest V2 features are rejected in Manifest V3
15. Check that manifest validation occurs on every extension load/reload

Pass Criteria: All manifest violations are detected and rejected AND clear error messages guide developers AND dangerous patterns trigger warnings

Fail Criteria: Invalid manifests are accepted OR error messages are unclear OR dangerous patterns have no warnings

Evidence: Screenshots of manifest validation errors, test results with various malformed manifests, warning dialogs for broad permissions, browser console logs during extension load

References:

- Chrome Manifest File Format: https://developer.chrome.com/docs/extensions/mv3/manifest/
- Manifest V3 Migration Guide: https://developer.chrome.com/docs/extensions/mv3/intro/mv3-migration/
- Mozilla Manifest.json Documentation: https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/manifest.json
- Extension Content Security Policy: https://developer.chrome.com/docs/extensions/mv3/manifest/content_security_poli
- CWE-20: Improper Input Validation: https://cwe.mitre.org/data/definitions/20.html
- WebExtensions Manifest Validation: https://extensionworkshop.com/documentation/develop/manifest-v3-migration-guide/

Assessment: EXT-REQ-5 (Extension sandboxing)

Reference: EXT-REQ-5 - Browser shall sandbox extension processes to prevent system-level access and privilege escalation

Given: A conformant browser with EXT-1 or higher capability

Task: Verify that the browser runs extension processes in an operating system sandbox with reduced privileges to prevent malicious extensions from accessing the file system directly, executing arbitrary system commands, creating processes, or escalating privileges to compromise the user's system, limiting extensions to controlled APIs and native messaging for necessary system interactions.

Verification:

1. Load a test extension and identify its background service worker process using browser task manager or process explorer
2. Attempt to execute operations that require system-level privileges from the extension:
   - File system access outside of extension storage APIs
   - Network operations outside of allowed extension APIs
   - Process creation or system command execution
   - Access to other processes' memory
3. Use platform-specific tools to verify sandbox restrictions:

- Windows: Process Explorer to check process token and integrity level
- macOS: Activity Monitor and sandbox-exec to check sandbox profile
- Linux: /proc filesystem and seccomp to verify syscall restrictions

4. Verify that extension processes run with reduced privileges (low integrity on Windows, restricted sandbox on macOS/Linux)
5. Test that native messaging hosts are the only permitted mechanism for system access
6. Verify that extension APIs providing system access (chrome.system.*) are themselves sandboxed and permission-gated
7. Test that renderer process crashes are isolated and don't affect browser stability
8. Confirm that extension processes run in a restricted sandbox with limited system access
9. Verify that direct file system access outside storage APIs is blocked
10. Check that system command execution is prevented
11. Validate that extension processes have reduced privilege levels observable via platform tools
12. Confirm that native messaging is the only controlled pathway to system-level functionality
13. Verify that process isolation prevents extensions from affecting each other or the browser
14. Check that sandbox violations trigger security errors and process termination

Pass Criteria: Extension processes are sandboxed with restricted system access AND privilege level is verifiably reduced AND native messaging is the only system access pathway

Fail Criteria: Extensions can perform system-level operations directly OR process privilege levels are not reduced OR sandbox can be escaped

Evidence: Process Explorer/Activity Monitor screenshots showing sandbox status, test results demonstrating blocked system operations, platform-specific sandbox verification (integrity levels, sandbox profiles), crash logs showing isolation

References:

- Chrome Sandbox Architecture: https://chromium.googlesource.com/chromium/src/+/master/docs/design/sandbox.md
- Extension Process Model: https://developer.chrome.com/docs/extensions/mv3/architecture-overview/
- Native Messaging: https://developer.chrome.com/docs/extensions/mv3/nativeMessaging/
- CWE-269: Improper Privilege Management: https://cwe.mitre.org/data/definitions/269.html
- Browser Sandbox Comparison: https://wiki.mozilla.org/Security/Sandbox
- Chrome Extension Security Architecture: https://www.chromium.org/Home/chromium-security/extension-content-script-fetches/

Assessment: EXT-REQ-6 (Cross-extension isolation)

Reference: EXT-REQ-6 - Browser shall isolate extensions from each other to prevent unauthorized inter-extension communication

Given: A conformant browser with EXT-1 or higher capability

Task: Verify that the browser enforces complete isolation between different extensions to prevent malicious or compromised extensions from stealing data, intercepting communications, or exploiting capabilities of other installed extensions, ensuring that each extension operates within its own security boundary and can only interact with others through explicitly configured and consent-based channels.

Verification:

1. Install two test extensions (Extension A and Extension B) with different extension IDs
2. From Extension A, attempt to access Extension B's resources:
   - Try to load Extension B's background page scripts
   - Attempt to access Extension B's storage (chrome.storage.local)
   - Try to send messages to Extension B using chrome.runtime.sendMessage(extensionBId, message)
3. Verify that direct access to another extension's resources is blocked
4. Test that only explicitly externally_connectable extensions can receive messages from other extensions
5. Create Extension B with externally_connectable manifest key allowing Extension A

6. Verify that messaging now works but only in the declared direction
7. Test that extensions cannot inject content scripts into each other's extension pages
8. Verify that web_accessible_resources from one extension cannot be accessed by another extension's content scripts without explicit configuration
9. Test that extension storage is strictly isolated between extensions
10. Confirm that extensions cannot access each other's background pages, storage, or internal resources by default
11. Verify that cross-extension messaging only works when explicitly configured via externally_connectable
12. Check that content script injection into other extensions' pages is blocked
13. Validate that extension storage is completely isolated per extension ID
14. Confirm that web-accessible resources have controlled access between extensions
15. Verify that attempts to access other extensions' resources throw security errors

Pass Criteria: Complete isolation between extensions by default AND externally_connectable controls messaging AND storage is isolated

Fail Criteria: Any unauthorized access between extensions OR messaging works without externally_connectable OR storage isolation fails

Evidence: Console screenshots showing cross-extension access errors, test results with and without externally_connectable, storage isolation verification, messaging test results

References:

- Chrome Extensions Externally Connectable: https://developer.chrome.com/docs/extensions/mv3/manifest/externally_c
- Extension Messaging: https://developer.chrome.com/docs/extensions/mv3/messaging/
- Web Accessible Resources: https://developer.chrome.com/docs/extensions/mv3/manifest/web_accessible_resources/
- CWE-668: Exposure of Resource to Wrong Sphere: https://cwe.mitre.org/data/definitions/668.html
- Extension Security Model: https://developer.chrome.com/docs/extensions/mv3/security/
- Mozilla Extension Communication: https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/Content_scripts#communicating_with_background_scripts

Assessment: EXT-REQ-7 (Host permissions validation)

Reference: EXT-REQ-7 - Browser shall validate and enforce host permissions declared in extension manifests

Given: A conformant browser with EXT-1 or higher capability

Task: Verify that the browser strictly enforces host permission declarations to prevent extensions from accessing arbitrary websites beyond their stated functionality, protecting users from extensions that attempt to steal credentials, intercept traffic, or exfiltrate data from unrelated sites, while ensuring that wildcard patterns and permission grants are clearly communicated and user-controllable.

Verification:

1. Create a test extension with specific host_permissions: `["https://example.com/*", "https://*.test.com/*"]`
2. Attempt to inject content scripts into various URLs:
   - Allowed: https://example.com/page, https://sub.test.com/page
   - Blocked: https://example.org/, https://different.com/, https://test.com.evil.com/
3. Test chrome.webRequest API interception with various URL patterns
4. Verify that fetch() and XMLHttpRequest from extension contexts respect host permissions
5. Test that host permission grants are persistent and revocable by users
6. Verify that optional host permissions require user interaction to grant
7. Test wildcard host permission patterns (, *:///*) and verify appropriate warnings
8. Test that activeTab permission provides temporary access to current tab without broad host permissions
9. Verify that host permissions are enforced consistently across all extension APIs (tabs, webRequest, scripting, etc.)
10. Confirm that content scripts can only inject into URLs matching host_permissions patterns
11. Verify that host permission patterns are correctly parsed and enforced (wildcards, subdomains, paths)

12. Check that network requests from extensions are blocked to non-permitted hosts
13. Validate that users can view and revoke host permissions through extension management UI
14. Confirm that optional host permissions require explicit user grant
15. Verify that broad permissions like show prominent warnings during install
16. Check that activeTab provides scoped, temporary permissions without persistent broad access

Pass Criteria: Host permissions are enforced across all APIs AND wildcard patterns work correctly AND users can control permissions AND activeTab provides limited scope

Fail Criteria: Extensions access hosts beyond declared permissions OR permission patterns are incorrectly parsed OR user controls are ineffective

Evidence: Test results showing blocked/allowed access by host, permission management UI screenshots, installation warning screenshots for broad permissions, activeTab test results

References:

- Chrome Host Permissions: https://developer.chrome.com/docs/extensions/mv3/match_patterns/
- ActiveTab Permission: https://developer.chrome.com/docs/extensions/mv3/manifest/activeTab/
- Optional Permissions: https://developer.chrome.com/docs/extensions/reference/permissions/
- User Control of Extension Permissions: https://developer.chrome.com/docs/extensions/mv3/permission_warnings/
- CWE-284: Improper Access Control: https://cwe.mitre.org/data/definitions/284.html
- Mozilla Match Patterns: https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/Match_patterns

Assessment: EXT-REQ-8 (CSP for extensions)

Reference: EXT-REQ-8 - Browser shall enforce Content Security Policy for extension pages and prevent unsafe code execution

Given: A conformant browser with EXT-1 or higher capability

Task: Verify that the browser enforces strict Content Security Policy on extension pages to prevent malicious or compromised extensions from executing arbitrary remote code, using eval-based code injection, or loading scripts from attacker-controlled servers, mitigating the risk of extensions becoming vectors for code injection attacks or post-install malicious behavior updates.

Verification:

1. Create a test extension with default CSP (Manifest V3 default: `script-src 'self'; object-src 'self'`)
2. In the extension's popup or background page, attempt to:
   - Execute inline scripts: `<script>alert('test')</script>`
   - Use eval(): `eval("alert('test')")`
   - Use Function constructor: `new Function("alert('test')")`
   - Load external scripts from CDNs: `<script src="https://cdn.example.com/lib.js">`
   - Use inline event handlers: `<button onclick="handleClick()">Click</button>`
3. Verify that all unsafe code execution attempts are blocked with CSP violations
4. Test that extension CSP cannot be relaxed to allow unsafe-eval or unsafe-inline in Manifest V3
5. Verify that remote code loading is blocked in Manifest V3
6. Test that sandboxed pages in extensions can have relaxed CSP but remain isolated
7. Monitor browser console for CSP violation reports
8. Test WASM execution with wasm-unsafe-eval directive requirements
9. Confirm that extension pages enforce strict CSP by default
10. Verify that inline scripts, eval(), and Function constructor are blocked
11. Check that remote code loading from CDNs is blocked
12. Validate that CSP violations are logged to console with clear messages
13. Confirm that Manifest V3 prevents CSP relaxation to unsafe-eval or unsafe-inline
14. Verify that sandboxed extension pages can have different CSP but remain isolated
15. Check that only local scripts from the extension package can execute

Pass Criteria: Strict CSP is enforced on all extension pages AND remote code is blocked AND CSP cannot be weakened in Manifest V3

Fail Criteria: Unsafe code execution succeeds OR remote scripts load OR CSP can be bypassed

Evidence: Console screenshots showing CSP violations, test results for eval/inline scripts/remote code, manifest CSP configuration examples, sandboxed page test results

References:

- Extension Content Security Policy: https://developer.chrome.com/docs/extensions/mv3/manifest/content_security_poli
- Manifest V3 CSP Changes: https://developer.chrome.com/docs/extensions/mv3/intro/mv3-migration/#content-security-policy
- CSP Specification: https://www.w3.org/TR/CSP3/
- Sandboxed Pages in Extensions: https://developer.chrome.com/docs/extensions/mv3/manifest/sandbox/
- CWE-94: Improper Control of Generation of Code: https://cwe.mitre.org/data/definitions/94.html
- Mozilla Extension CSP: https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/Content_Security_

Assessment: EXT-REQ-9 (WebRequest API security)

Reference: EXT-REQ-9 - Browser shall secure the WebRequest API to prevent malicious request interception and modification

Given: A conformant browser with EXT-2 or higher capability

Task: Verify that the browser secures the WebRequest API to prevent malicious extensions from performing man-in-the-middle attacks, stealing authentication tokens, modifying banking transactions, or blocking security updates, while ensuring that Manifest V3's declarativeNetRequest provides necessary functionality with reduced attack surface through rule validation, limits, and sensitive header protections.

Verification:

1. Create test extensions using declarativeNetRequest API (Manifest V3):
   - Rules to block specific URLs
   - Rules to redirect requests
   - Rules to modify request headers
   - Rules to modify response headers
2. Verify that declarativeNetRequest rules are validated before installation
3. Test that rule limits are enforced (static rules, dynamic rules, session rules)
4. Attempt to create overly broad rules that would affect all network traffic
5. Verify that sensitive headers (Cookie, Authorization) have restricted modification capabilities
6. Test that extensions cannot intercept requests to chrome:// or chrome-extension:// schemes
7. Verify that declarativeNetRequest is more restrictive than legacy webRequest blocking
8. Test that multiple extensions with conflicting rules have predictable precedence
9. Verify that users are warned about extensions with webRequest permissions (Manifest V2) or declarativeNetRequest (Manifest V3)
10. Confirm that declarativeNetRequest rules are validated and enforced
11. Verify that rule count limits prevent resource exhaustion
12. Check that sensitive headers have protection against modification
13. Validate that browser internal URLs are protected from interception
14. Confirm that Manifest V3 declarativeNetRequest is more restrictive than Manifest V2 webRequest
15. Verify that rule conflicts between extensions are resolved predictably
16. Check that permission warnings clearly communicate network interception capabilities

Pass Criteria: DeclarativeNetRequest enforces rule validation and limits AND sensitive contexts are protected AND permission warnings are clear

Fail Criteria: Invalid rules are accepted OR no rule limits OR internal URLs can be intercepted OR no permission warnings

Evidence: Test results showing rule validation, screenshots of rule limits enforcement, header modification test results, permission warning screenshots, conflict resolution examples

References:

- Chrome DeclarativeNetRequest API: https://developer.chrome.com/docs/extensions/reference/declarativeNetRequest/
- Manifest V3 WebRequest Changes: https://developer.chrome.com/docs/extensions/mv3/intro/mv3-migration/#modifying-network-requests
- DeclarativeNetRequest Rule Limits: https://developer.chrome.com/docs/extensions/reference/declarativeNetRequest/#l
- Mozilla WebRequest API: https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/API/webRequest
- CWE-300: Channel Accessible by Non-Endpoint: https://cwe.mitre.org/data/definitions/300.html
- Extension Network Request Security: https://developer.chrome.com/docs/extensions/mv3/security/#network-requests

Assessment: EXT-REQ-10 (Extension update verification)

Reference: EXT-REQ-10 - Browser shall cryptographically verify extension updates before installation

Given: A conformant browser with EXT-1 or higher capability

Task: Verify that the browser cryptographically verifies all extension updates to prevent attackers from distributing malware through compromised update servers, man-in-the-middle attacks, or extension ID hijacking, ensuring that only authentic updates signed by the original developer can replace installed extensions and protecting users from supply chain attacks.

Verification:

1. Install a test extension from the Chrome Web Store or addons.mozilla.org
2. Monitor the extension update process using browser internals (chrome://extensions, about:debugging)
3. Configure a local web server to serve a modified version of the extension with:
   - Invalid signature
   - Mismatched extension ID
   - Tampered CRX/XPI file
4. Attempt to force the browser to update from the malicious server
5. Verify that the browser rejects the tampered update
6. Test that update_url in manifest should point to official web stores for published extensions
7. Verify that self-hosted extensions require proper signatures
8. Test CRX3 signature verification (Chrome) or signed XPI verification (Firefox)
9. Monitor network traffic to verify updates use HTTPS
10. Test that update checks include extension ID, version, and signature verification
11. Confirm that all extension updates are downloaded over HTTPS
12. Verify that CRX/XPI file signatures are cryptographically verified before installation
13. Check that tampered updates are rejected with error messages
14. Validate that extension ID shall match between installed extension and update
15. Confirm that update URLs should be HTTPS and point to trusted sources
16. Verify that self-hosted extensions require proper code signing
17. Check that update process cannot be MITM attacked to install malicious code

Pass Criteria: All updates are signature-verified AND tampered updates are rejected AND HTTPS is enforced AND extension ID matching is enforced

Fail Criteria: Unsigned or tampered updates install successfully OR HTTP update URLs work OR extension ID mismatch is allowed

Evidence: Network captures showing HTTPS update checks, logs of rejected tampered updates, signature verification test results, CRX/XPI file inspection showing signatures

References:

- Chrome CRX3 Extension Format: https://chromium.googlesource.com/chromium/src/+/master/components/crx_file/cr

- Chrome Extension Update Mechanism: https://developer.chrome.com/docs/extensions/mv3/linux_hosting/#update
- Mozilla Extension Signing: https://extensionworkshop.com/documentation/publish/signing-and-distribution-overview/
- CWE-494: Download of Code Without Integrity Check: https://cwe.mitre.org/data/definitions/494.html
- Extension Package Security: https://www.chromium.org/Home/chromium-security/extension-content-script-fetches/

Assessment: EXT-REQ-11 (Extension storage isolation)

Reference: EXT-REQ-11 - Browser shall isolate extension storage to prevent unauthorized access between extensions and web pages

Given: A conformant browser with EXT-1 or higher capability

Task: Verify that the browser isolates extension storage to prevent malicious extensions from stealing API keys, authentication tokens, or user data stored by other extensions, and to prevent web pages from accessing extension storage to exfiltrate sensitive information, ensuring that each extension's storage remains private and accessible only within its own security context.

Verification:

1. Create Extension A that stores sensitive data in chrome.storage.local and chrome.storage.sync:

   ```
   chrome.storage.local.set({secret: "extension_A_secret"});
   chrome.storage.sync.set({syncData: "extension_A_sync"});
   ```

2. Create Extension B that attempts to read Extension A's storage:

   ```
   chrome.storage.local.get("secret", (result) => console.log(result));
   ```

3. Verify that Extension B cannot access Extension A's storage
4. Create a web page that attempts to access extension storage using various methods:
   - Direct chrome.storage access
   - IndexedDB inspection for extension storage
   - File system access to extension storage location
5. Verify that web pages have no access to extension storage
6. Test that extension storage persists across browser restarts
7. Verify that uninstalling an extension removes its storage
8. Test chrome.storage quota limits and enforcement
9. Verify that chrome.storage.sync has appropriate sync limits and encryption
10. Confirm that each extension has isolated storage inaccessible to other extensions
11. Verify that web pages cannot access any extension storage APIs or data
12. Check that extension storage persists across sessions but is removed on uninstall
13. Validate that storage quota limits are enforced per extension
14. Confirm that chrome.storage.sync uses encrypted sync when user is signed in
15. Verify that no file system or database tools can access extension storage from outside the browser
16. Check that storage access from wrong context results in clear error messages

Pass Criteria: Complete storage isolation between extensions AND no web page access AND proper quota enforcement AND sync encryption

Fail Criteria: Cross-extension storage access succeeds OR web pages can read extension storage OR no quota enforcement

Evidence: Test results showing isolation between extensions, console errors from web pages attempting access, persistent storage verification, quota limit test results, sync encryption verification

References:

- Chrome Storage API: https://developer.chrome.com/docs/extensions/reference/storage/
- Storage Quota Limits: https://developer.chrome.com/docs/extensions/reference/storage/#property-sync-QUOTA_BYTES
- Mozilla Storage API: https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/API/storage
- Storage Sync Security: https://developer.chrome.com/docs/extensions/reference/storage/#property-sync
- CWE-552: Files or Directories Accessible to External Parties: https://cwe.mitre.org/data/definitions/552.html
- Extension Data Security: https://developer.chrome.com/docs/extensions/mv3/security/#data

Assessment: EXT-REQ-12 (Background script restrictions)

Reference: EXT-REQ-12 - Browser shall enforce restrictions on background scripts including service worker lifecycle and capabilities

Given: A conformant browser with EXT-2 or higher capability (Manifest V3)

Task: Verify that the browser enforces service worker lifecycle restrictions for background scripts to reduce resource consumption, prevent persistent background pages that could perform long-term surveillance or crypto-mining, and ensure that extensions cannot maintain always-on processes that degrade performance or bypass browser power management, while maintaining security through strict CSP enforcement.

Verification:

1. Create a Manifest V3 extension with a background service worker
2. Verify that background pages (persistent background scripts) are not allowed in Manifest V3
3. Test service worker lifecycle:
    - Service worker starts on extension events (installation, message, alarm)
    - Service worker terminates after idle timeout (~30 seconds)
    - Service worker restarts when needed for events
4. Attempt to use browser APIs not available in service workers:
    - DOM APIs (document, window)
    - Synchronous storage APIs
    - XMLHttpRequest (should use fetch instead)
5. Verify that service workers cannot use eval() or other dynamic code execution
6. Test that long-running operations should use alarms API or native messaging
7. Verify that service worker registration is automatic and cannot be modified
8. Test that service worker cannot be kept alive artificially
9. Verify CSP enforcement in service worker context
10. Confirm that Manifest V3 extensions use service workers, not persistent background pages
11. Verify that service workers terminate after idle timeout and restart on events
12. Check that DOM APIs are unavailable in service worker context
13. Validate that dynamic code execution is blocked in service workers
14. Confirm that service worker lifecycle is managed by the browser, not the extension
15. Verify that long-running tasks should use appropriate APIs (alarms, native messaging)
16. Check that service worker CSP is strict and enforced

Pass Criteria: Service worker lifecycle is enforced AND unavailable APIs throw errors AND CSP is enforced AND artificial keep-alive is prevented

Fail Criteria: Persistent background pages work in Manifest V3 OR service worker doesn't terminate OR restricted APIs are available

Evidence: Test results showing service worker termination, console errors for unavailable APIs, lifecycle event logs, CSP violation logs, timing tests showing automatic termination

References:

- Manifest V3 Service Workers: https://developer.chrome.com/docs/extensions/mv3/migrating_to_service_workers/
- Service Worker Lifecycle: https://developer.chrome.com/docs/extensions/mv3/service_workers/
- Background Script Migration: https://developer.chrome.com/docs/extensions/mv3/intro/mv3-migration/#background-service-workers
- Service Workers in Extensions: https://developer.chrome.com/docs/extensions/mv3/service_workers/basics/
- CWE-405: Asymmetric Resource Consumption: https://cwe.mitre.org/data/definitions/405.html
- Extension Service Worker Events: https://developer.chrome.com/docs/extensions/mv3/service_workers/events/

Assessment: EXT-REQ-13 (Manifest V3 compliance)

Reference: EXT-REQ-13 - Browser shall enforce Manifest V3 security requirements for new and updated extensions

Given: A conformant browser with EXT-2 or higher capability

Task: Verify that the browser enforces Manifest V3 security requirements to prevent extensions from using deprecated, insecure patterns such as persistent background pages that enable surveillance, blocking webRequest that enables man-in-the-middle attacks, relaxed CSP that allows remote code execution, and callback-based APIs prone to timing attacks, ensuring all new extensions benefit from modern security architecture.

Verification:

1. Verify that browser supports Manifest V3 extensions
2. Test that new extension submissions require Manifest V3 (check store policies)
3. Create test extensions demonstrating Manifest V3 security improvements:
   - Service workers instead of persistent background pages
   - DeclarativeNetRequest instead of blocking webRequest
   - Strict CSP with no unsafe-eval
   - No remote code execution
   - Promises-based APIs instead of callbacks
4. Attempt to use deprecated Manifest V2 features in Manifest V3 extension:
   - background.persistent
   - background.page
   - webRequest blocking with broad host permissions
   - Relaxed CSP with unsafe-eval
5. Verify that browser rejects or warns about Manifest V2 features in Manifest V3 context
6. Test that Manifest V2 extensions show deprecation warnings
7. Verify manifest_version field validation (should be 2 or 3, with 3 preferred)
8. Test migration path from V2 to V3 with breaking changes properly surfaced
9. Confirm that browser fully supports Manifest V3 specification
10. Verify that Manifest V2 deprecated features are rejected in Manifest V3 extensions
11. Check that security improvements (service workers, declarativeNetRequest, strict CSP) are enforced
12. Validate that clear error messages guide developers away from deprecated patterns
13. Confirm that Manifest V2 extensions show deprecation warnings to users
14. Verify that extension stores enforce Manifest V3 for new submissions
15. Check that migration tooling and documentation are available

Pass Criteria: Manifest V3 security requirements are fully enforced AND deprecated V2 features are rejected AND clear migration guidance exists

Fail Criteria: V2 deprecated features work in V3 extensions OR no enforcement of V3 security model OR no deprecation warnings

Evidence: Manifest validation test results, deprecation warning screenshots, test extensions demonstrating V3 features, store policy documentation, migration guide references

References:

- Manifest V3 Overview: https://developer.chrome.com/docs/extensions/mv3/intro/
- Manifest V3 Migration Guide: https://developer.chrome.com/docs/extensions/mv3/intro/mv3-migration/
- Manifest V3 Platform Vision: https://developer.chrome.com/docs/extensions/mv3/intro/platform-vision/
- Mozilla Manifest V3: https://blog.mozilla.org/addons/2022/05/18/manifest-v3-in-firefox-recap-next-steps/
- CWE-477: Use of Obsolete Function: https://cwe.mitre.org/data/definitions/477.html
- Manifest V3 Security Improvements: https://developer.chrome.com/docs/extensions/mv3/intro/mv3-overview/#security

Assessment: EXT-REQ-14 (Native messaging security)

Reference: EXT-REQ-14 - Browser shall secure native messaging to prevent unauthorized native application access

Given: A conformant browser with EXT-1 or higher capability

Task: Verify that the browser secures native messaging to prevent malicious extensions from escaping the browser sandbox by connecting to arbitrary native applications, installing native malware disguised as messaging hosts, or using native messaging as a privilege escalation vector, ensuring that only explicitly whitelisted extensions can communicate with user-installed, validated native applications.

Verification:

1. Create a native messaging host application with a manifest file
2. Register the native messaging host according to platform requirements:
   - Windows: Registry entry under HKEY_CURRENT_USER or HKEY_LOCAL_MACHINE
   - macOS/Linux: JSON manifest in specified directories
3. Create an extension with nativeMessaging permission and allowed_origins in native host manifest
4. Test communication between extension and native host using chrome.runtime.connectNative()
5. Verify that only extensions listed in native host manifest's allowed_origins can connect
6. Attempt connection from an unlisted extension and verify rejection
7. Test that native host path validation prevents directory traversal
8. Verify that native messaging requires user-installed native applications (not downloadable by extensions)
9. Test message size limits and validation
10. Verify that native host processes run with appropriate user privileges (not elevated)
11. Confirm that native messaging requires explicit nativeMessaging permission
12. Verify that native host manifest should acceptlist extension IDs in allowed_origins
13. Check that unlisted extensions cannot connect to native hosts
14. Validate that native host executable path is validated against tampering
15. Confirm that extensions cannot download or install native hosts programmatically
16. Verify that message passing is properly sandboxed and size-limited
17. Check that native hosts run without elevated privileges
18. Validate that connection attempts from unauthorized extensions fail with clear errors

Pass Criteria: Native messaging requires permission and allowed_origins whitelisting AND path validation prevents tampering AND extensions cannot install hosts

Fail Criteria: Any extension can connect to native hosts OR path validation is bypassable OR extensions can install native hosts programmatically

Evidence: Test results showing connection rejection for unlisted extensions, native host manifest examples, registry/filesystem inspection showing host registration, message passing test results, privilege level verification

References:

- Chrome Native Messaging: https://developer.chrome.com/docs/extensions/mv3/nativeMessaging/
- Native Messaging Host Protocol: https://developer.chrome.com/docs/extensions/mv3/nativeMessaging/#native-messaging-host-protocol
- Mozilla Native Messaging: https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/Native_messagin
- Native Host Manifest Format: https://developer.chrome.com/docs/extensions/mv3/nativeMessaging/#native-messaging-host
- CWE-494: Download of Code Without Integrity Check: https://cwe.mitre.org/data/definitions/494.html
- Extension-Native App Security: https://developer.chrome.com/docs/extensions/mv3/security/#native-messaging

Assessment: EXT-REQ-15 (Extension-controlled web content)

Reference: EXT-REQ-15 - Browser shall prevent extensions from injecting malicious content or deceptively modifying web pages

Given: A conformant browser with EXT-1 or higher capability

Task: Verify that the browser prevents extensions from injecting malicious content into web pages to conduct phishing attacks, overlaying fake login forms to steal credentials, intercepting form submissions to capture banking details, or deceptively modifying trusted website content, while ensuring that legitimate content script functionality is appropriately permission-gated and user-visible through DevTools attribution.

Verification:

1. Create test extensions that attempt various forms of web content modification:
   - Injecting scripts into web pages via content scripts
   - Modifying DOM to overlay fake UI elements (e.g., fake login forms)
   - Intercepting and modifying form submissions
   - Replacing legitimate content with malicious content
2. Verify that content script capabilities are limited by Content Security Policy
3. Test that extensions should declare host permissions for content script injection
4. Create a test extension that attempts to:
   - Inject content scripts into browser UI pages (chrome://, edge://)
   - Inject into other extensions' pages
   - Inject into local file:// URLs without explicit permission
5. Verify that such injections are blocked
6. Test that users are warned about extensions with broad host permissions during installation
7. Verify that content script modifications are visible in DevTools with extension attribution
8. Test that extensions cannot inject content into incognito mode without explicit permission
9. Verify CSP restrictions prevent content scripts from loading remote code
10. Confirm that content scripts require declared host permissions
11. Verify that browser UI pages and other extensions are protected from content script injection
12. Check that broad host permissions trigger prominent installation warnings
13. Validate that content script modifications are attributable in DevTools
14. Confirm that incognito mode requires explicit extension permission
15. Verify that CSP prevents content scripts from loading remote malicious code
16. Check that file:// access requires explicit user permission

Pass Criteria: Content scripts are permission-gated AND sensitive contexts are protected AND user warnings are clear AND DevTools attribution works

Fail Criteria: Content scripts inject without permissions OR browser pages are injectable OR no user warnings OR no attribution

Evidence: Permission prompt screenshots, test results showing blocked injections, DevTools showing extension attribution, CSP enforcement test results, incognito permission tests

References:

- Content Scripts Security: https://developer.chrome.com/docs/extensions/mv3/content_scripts/#security
- Extension Content Script Injection: https://developer.chrome.com/docs/extensions/mv3/content_scripts/#programmati
- Mozilla Content Scripts: https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/Content_scripts
- Incognito Mode for Extensions: https://developer.chrome.com/docs/extensions/mv3/manifest/incognito/
- CWE-79: Cross-site Scripting (XSS): https://cwe.mitre.org/data/definitions/79.html
- Extension Security Best Practices: https://extensionworkshop.com/documentation/develop/build-a-secure-extension/

Assessment: EXT-REQ-16 (Extension telemetry privacy)

Reference: EXT-REQ-16 - Browser shall ensure extension telemetry and error reporting respect user privacy

Given: A conformant browser with EXT-1 or higher capability

Task: Verify that the browser ensures extension telemetry and error reporting respect user privacy to prevent extensions from exfiltrating browsing history, capturing personally identifiable information, or using analytics to track users across the web without consent, while ensuring that extension developers disclose data collection practices and that sensitive information is not inadvertently leaked through error reports or console logs.

Verification:

1. Install a test extension with error reporting or analytics code
2. Monitor network traffic from the extension using browser DevTools or external proxy (e.g., Burp Suite, mitmproxy)
3. Verify that extensions cannot access browser telemetry or crash reporting APIs directly
4. Test that extension storage sync does not leak data to unauthorized parties
5. Verify that error reporting from extensions requires user consent if it includes:
    - URLs visited by the user
    - Personal identifiable information
    - Browsing history or patterns
6. Test that chrome.storage.sync encryption prevents extension from reading other extensions' sync data
7. Verify that extensions should declare and justify data collection in privacy policies
8. Test that extensions cannot access browser's own telemetry data
9. Monitor for sensitive data leakage in extension console logs (passwords, tokens, PII)
10. Verify that extension store policies require privacy disclosures for data collection
11. Confirm that extensions have no access to browser telemetry APIs
12. Verify that extension-to-server communications are visible and monitorable by users
13. Check that sensitive user data is not included in extension error reports without consent
14. Validate that chrome.storage.sync data is encrypted and isolated
15. Confirm that extension developers shall disclose data collection practices
16. Verify that console logs do not inadvertently expose sensitive data
17. Check that users can review extension permissions related to data collection

Pass Criteria: Extensions cannot access browser telemetry AND user consent is required for sensitive data collection AND privacy policies are required

Fail Criteria: Extensions access browser telemetry OR sensitive data is transmitted without consent OR no privacy policy requirements

Evidence: Network traffic captures showing extension communications, test results showing blocked telemetry API access, privacy policy examples from extension stores, storage sync encryption verification

References:

- Chrome Extension Privacy Practices: https://developer.chrome.com/docs/extensions/mv3/user_privacy/
- Extension Privacy Policy Requirements: https://developer.chrome.com/docs/webstore/program-policies/privacy/

- Mozilla Data Collection Guidelines: https://extensionworkshop.com/documentation/publish/add-on-policies/#data-disclosure-collection-and-management
- GDPR Compliance for Extensions: https://gdpr.eu/what-is-gdpr/
- CWE-359: Exposure of Private Personal Information: https://cwe.mitre.org/data/definitions/359.html

Assessment: EXT-REQ-17 (Extension signature validation)

Reference: EXT-REQ-17 - Browser shall validate cryptographic signatures of extensions before installation and during runtime

Given: A conformant browser with EXT-1 or higher capability

Task: Verify that the browser validates cryptographic signatures of extensions to prevent installation of tampered, backdoored, or malicious extensions distributed through compromised channels, ensuring that only extensions authenticated by trusted authorities or the original developer can be installed, and that signature validation provides defense-in-depth against supply chain attacks and extension package manipulation.

Verification:

1. Download a legitimate signed extension from the Chrome Web Store or addons.mozilla.org
2. Inspect the extension package (CRX for Chrome, XPI for Firefox) to verify signature presence
3. Use tools to verify the signature:
    - Chrome: Examine CRX3 header with crx3 tools
    - Firefox: Verify XPI signature with Mozilla's signing infrastructure
4. Attempt to install a modified extension with:
    - Invalid signature
    - No signature
    - Expired signature
    - Signature from untrusted authority
5. Verify that the browser rejects all invalid signatures with clear error messages
6. Test that extensions loaded in developer mode can bypass signature requirements (but with clear warnings)
7. Verify that production extensions require valid signatures from trusted authorities
8. Test that signature validation occurs at both install time and runtime
9. Verify that browser checks certificate revocation for extension signatures
10. Test that extension updates have been signed by the same key as the original
11. Confirm that all production extensions have valid cryptographic signatures
12. Verify that signatures are verified at installation time
13. Check that invalid, expired, or missing signatures prevent installation
14. Validate that developer mode allows unsigned extensions with prominent warnings
15. Confirm that signature validation uses trusted certificate authorities
16. Verify that certificate revocation is checked during validation
17. Check that extension updates require signature continuity (same signing key)
18. Validate that clear error messages explain signature validation failures

Pass Criteria: All production extensions require valid signatures AND signature validation is comprehensive AND update signature continuity is enforced

Fail Criteria: Unsigned extensions install in production mode OR signature validation is bypassable OR no certificate revocation checking

Evidence: CRX/XPI file inspection showing signatures, test results with tampered signatures, installation error screenshots, certificate chain verification, developer mode warning screenshots

References:

- Chrome CRX3 Format and Signing: https://chromium.googlesource.com/chromium/src/+/master/components/crx_file/
- Mozilla Extension Signing: https://extensionworkshop.com/documentation/publish/signing-and-distribution-overview/

- Extension Package Format: https://developer.chrome.com/docs/extensions/mv3/linux_hosting/#packaging
- Code Signing Best Practices: https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/Distribution
- CWE-345: Insufficient Verification of Data Authenticity: https://cwe.mitre.org/data/definitions/345.html
- Chrome Web Store Developer Policies: https://developer.chrome.com/docs/webstore/program-policies/

Assessment: EXT-REQ-18 (Extension permissions UI transparency)

Reference: EXT-REQ-18 - Browser shall provide clear, understandable UI for extension permissions allowing informed user consent

Given: A conformant browser with EXT-1 or higher capability

Task: Verify that the browser provides clear, understandable permission UI to enable informed user consent, preventing extensions from deceiving users about their capabilities through technical jargon, hidden permission requests, or unclear warnings, ensuring that users understand what access they're granting and can make security-conscious decisions about extension installation, especially for high-risk permissions that could enable data theft or surveillance.

Verification:

1. Create test extensions with various permission combinations:
   - Low-risk permissions (storage, alarms)
   - Medium-risk permissions (tabs, activeTab)
   - High-risk permissions (webRequest, , cookies)
   - Sensitive permissions (debugger, management, privacy)
2. Install each extension and observe permission prompts
3. Verify that permission prompts:
   - Use clear, non-technical language
   - Group permissions by risk level
   - Explain what each permission allows
   - Show prominent warnings for dangerous permissions
4. Test that users can view current permissions in extension management UI
5. Verify that users can revoke permissions without uninstalling extension
6. Test optional permissions flow where extensions request additional permissions at runtime
7. Verify that permission grant/revocation is persistent across sessions
8. Test that permission changes trigger re-prompting for consent
9. Verify that extensions cannot request permissions programmatically without user interaction
10. Test that broad permissions like have prominent, scary warnings
11. Confirm that permission prompts use clear, user-friendly language (not technical jargon)
12. Verify that high-risk permissions have prominent warnings with specific explanations
13. Check that users can view all extension permissions in management UI
14. Validate that users can revoke individual permissions without uninstalling
15. Confirm that optional permissions require explicit user interaction to grant
16. Verify that permission changes trigger new consent prompts
17. Check that broad permissions have special warnings about privacy implications
18. Validate that permission UI is consistent across installation and runtime requests

Pass Criteria: Permission UI is clear and understandable AND risk-appropriate warnings are shown AND users have granular control AND persistent consent tracking

Fail Criteria: Permission language is too technical OR no risk differentiation OR users cannot revoke permissions OR no warnings for dangerous permissions

Evidence: Screenshots of permission prompts at various risk levels, extension management UI screenshots, optional permission flow recordings, user testing results demonstrating comprehension, revocation test results

References:

- Chrome Permission Warnings: https://developer.chrome.com/docs/extensions/mv3/permission_warnings/

- Mozilla Permission Requests: https://extensionworkshop.com/documentation/develop/request-the-right-permissions/
- Extension Permission UX Best Practices: https://developer.chrome.com/docs/extensions/mv3/permission_warnings/#pe
- CWE-276: Incorrect Default Permissions: https://cwe.mitre.org/data/definitions/276.html
- Designing User-Friendly Permission Systems: https://extensionworkshop.com/documentation/develop/build-a-secure-extension/#request-permissions-at-runtime

6.3 Cryptographic Security Assessments

This section covers assessment procedures for requirements ENC-REQ-1 through ENC-REQ-21, addressing TLS/SSL configuration, certificate validation, cryptographic API usage, secure random number generation, and encryption key management.

Assessment: ENC-REQ-1 (TLS 1.3+ support)

Reference: ENC-REQ-1 - Browser shall support TLS 1.3 or higher for all network communications

Given: A conformant browser with encryption capability (ENC-1 or higher)

Task: Verify that the browser supports modern TLS 1.3 protocol to protect network communications from downgrade attacks, reduce latency through improved handshake efficiency, and provide forward secrecy through ephemeral key exchange, ensuring that user traffic benefits from the latest cryptographic protections against eavesdropping and man-in-the-middle attacks.

Verification:

1. Configure a test web server to support multiple TLS versions: TLS 1.0, TLS 1.1, TLS 1.2, and TLS 1.3
2. Navigate the browser to each server endpoint and capture the TLS handshake using packet capture tools (Wireshark, tcpdump)
3. Verify that the browser successfully negotiates TLS 1.3 when available
4. Configure the server to offer only TLS 1.3 and verify successful connection
5. Test TLS 1.3 features: 0-RTT resumption, encrypted SNI (ESNI/ECH), and post-handshake authentication
6. Use browser DevTools Security tab to verify TLS version for the connection
7. Test that browser prefers TLS 1.3 over older versions when both are available
8. Verify that TLS 1.3 cipher suites are properly supported (TLS_AES_128_GCM_SHA256, TLS_AES_256_GCM_SHA384, TLS_CHACHA20_POLY1305_SHA256)
9. Confirm that browser successfully negotiates TLS 1.3 when server supports it
10. Verify that TLS 1.3 is preferred over TLS 1.2 when both are available
11. Check that browser Security panel shows "TLS 1.3" as the protocol version
12. Validate that packet captures confirm TLS 1.3 handshake structure (single round-trip)
13. Confirm that TLS 1.3-specific features (0-RTT, encrypted extensions) function correctly
14. Verify that browser supports all mandatory TLS 1.3 cipher suites

Pass Criteria: Browser successfully negotiates TLS 1.3 connections AND prefers TLS 1.3 over older versions when available

Fail Criteria: Browser cannot establish TLS 1.3 connections OR prefers older TLS versions when TLS 1.3 is available

Evidence: Wireshark packet captures showing TLS 1.3 handshake, DevTools Security tab screenshots, server logs showing negotiated protocol version, cipher suite listings

References:

- RFC 8446 - TLS 1.3 Specification: https://datatracker.ietf.org/doc/html/rfc8446
- Mozilla SSL Configuration Generator: https://ssl-config.mozilla.org/
- NIST SP 800-52 Rev. 2 - TLS Guidelines: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-52r2.pdf

- Qualys SSL Labs - SSL/TLS Best Practices: https://github.com/ssllabs/research/wiki/SSL-and-TLS-Deployment-Best-Practices
- Chrome Network Security: https://www.chromium.org/Home/chromium-security/education/tls/

Assessment: ENC-REQ-2 (Certificate validation)

Reference: ENC-REQ-2 - Browser shall perform complete certificate chain validation including expiry, revocation, and trust anchor verification

Given: A conformant browser with encryption capability (ENC-1 or higher)

Task: Verify that the browser performs comprehensive certificate chain validation to prevent man-in-the-middle attacks using expired, self-signed, or fraudulent certificates, ensuring that only connections to servers with valid certificates from trusted certificate authorities are established without warnings, protecting users from impersonation attacks and ensuring transport layer authentication.

Verification:

1. Set up test scenarios with various certificate configurations:
   - Valid certificate from trusted CA
   - Expired certificate
   - Self-signed certificate
   - Certificate with wrong hostname (CN/SAN mismatch)
   - Certificate with incomplete chain
   - Certificate from untrusted CA
   - Revoked certificate
2. Navigate to each test endpoint and observe browser behavior
3. Verify that browser displays appropriate security warnings for invalid certificates
4. Test that users cannot bypass critical certificate errors without explicit acknowledgment
5. Verify certificate chain validation using browser DevTools Security tab
6. Test extended validation (EV) certificate display
7. Capture and analyze the certificate validation process
8. Test certificate validation with intermediate CA certificates
9. Confirm that valid certificates from trusted CAs are accepted without warnings
10. Verify that expired certificates trigger security warnings and connection blocks
11. Check that self-signed certificates are rejected with clear error messages
12. Validate that hostname mismatches are detected and reported
13. Confirm that incomplete certificate chains are rejected
14. Verify that certificates from untrusted CAs trigger warnings
15. Check that users should explicitly acknowledge risk to bypass warnings
16. Validate that EV certificates display organization name in browser UI

Pass Criteria: All invalid certificates are detected and reported with appropriate warnings AND users cannot silently bypass critical errors

Fail Criteria: Any invalid certificate is accepted without warning OR users can bypass critical errors without explicit acknowledgment

Evidence: Screenshots of certificate warning dialogs, DevTools Security tab showing certificate validation status, error console logs, packet captures showing certificate exchange

References:

- RFC 5280 - X.509 Certificate and CRL Profile: https://datatracker.ietf.org/doc/html/rfc5280
- CA/Browser Forum Baseline Requirements: https://cabforum.org/baseline-requirements-documents/
- OWASP Transport Layer Protection Cheat Sheet: https://cheatsheetseries.owasp.org/cheatsheets/Transport_Layer_Prot
- Mozilla Root Store Policy: https://www.mozilla.org/en-US/about/governance/policies/security-group/certs/policy/
- Chrome Certificate Transparency Policy: https://github.com/chromium/ct-policy

Assessment: ENC-REQ-3 (Certificate pinning support)

Reference: ENC-REQ-3 - Browser shall support HTTP Public Key Pinning (HPKP) or alternative certificate pinning mechanisms

Given: A conformant browser with encryption capability (ENC-2 or higher)

Task: Verify that the browser enforces certificate pinning to prevent man-in-the-middle attacks against critical domains by rejecting connections to pinned sites that present certificates from unexpected certificate authorities, even if those CAs are otherwise trusted, ensuring that even with a compromised certificate authority, attackers cannot impersonate pinned origins through fraudulent certificate issuance.

Verification:

1. Configure a test server to send Public-Key-Pins or Expect-CT headers
2. Navigate to the test server and verify that pins are stored by the browser
3. Attempt to connect with a certificate that violates the pin (different public key)
4. Verify that the browser rejects the connection due to pin validation failure
5. Test pin expiration and max-age directive behavior
6. Test includeSubDomains directive for pin inheritance
7. Test report-uri directive for pin violation reporting
8. Examine browser's HPKP/pin storage (chrome://net-internals/#hsts for Chromium)
9. Test static pins for built-in domains (e.g., Google properties)
10. Confirm that browser correctly stores certificate pins from HTTP headers
11. Verify that connections with mismatched certificates are rejected
12. Check that pin validation errors display clear security warnings
13. Validate that pins expire according to max-age directive
14. Confirm that includeSubDomains directive applies pins to subdomains
15. Verify that pin violations are reported to specified report-uri
16. Check that browser maintains persistent pin storage across sessions
17. Validate that static pins for built-in domains are enforced

Pass Criteria: Browser enforces certificate pins correctly AND rejects connections that violate pin requirements

Fail Criteria: Browser ignores certificate pins OR allows connections that violate pin requirements

Evidence: Screenshots of pin validation errors, browser internal state showing stored pins, network logs showing rejected connections, pin violation reports

References:

- RFC 7469 - HTTP Public Key Pinning (HPKP): https://datatracker.ietf.org/doc/html/rfc7469
- Expect-CT Header: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Expect-CT
- OWASP Certificate Pinning: https://owasp.org/www-community/controls/Certificate_and_Public_Key_Pinning
- Chrome HPKP Guidance: https://www.chromium.org/hsts/

Assessment: ENC-REQ-4 (HSTS enforcement)

Reference: ENC-REQ-4 - Browser shall enforce HTTP Strict Transport Security (HSTS) to prevent protocol downgrade attacks

Given: A conformant browser with encryption capability (ENC-1 or higher)

Task: Verify that HSTS enforcement prevents protocol downgrade attacks where attackers strip TLS to intercept credentials, ensuring all connections to HSTS-enabled domains use HTTPS even if users or applications attempt insecure HTTP connections. This protection is critical for preventing man-in-the-middle attacks that exploit the initial HTTP connection window before users type "https://" or before redirects to HTTPS occur.

Verification:

1. Configure a test server to send Strict-Transport-Security header: `Strict-Transport-Security: max-age=31536000; includeSubDomains; preload`
2. Navigate to https://test.example.com and verify HSTS header is received
3. Attempt to navigate to http://test.example.com (non-HTTPS) and verify automatic upgrade to HTTPS
4. Test that browser stores HSTS policy persistently across sessions
5. Verify includeSubDomains directive applies HSTS to all subdomains
6. Test HSTS policy expiration by setting short max-age values
7. Examine browser's HSTS storage (chrome://net-internals/#hsts)
8. Test HSTS preload list for well-known domains
9. Attempt to bypass HSTS with user override and verify it's prevented
10. Browser receives and processes Strict-Transport-Security header
11. HTTP requests are automatically upgraded to HTTPS for HSTS domains
12. HSTS policy persists across browser sessions
13. includeSubDomains directive correctly applies to subdomains
14. HSTS policies expire according to max-age directive
15. Preloaded domains enforce HTTPS even on first visit
16. Users cannot bypass HSTS enforcement
17. Browser maintains HSTS state in persistent storage

Pass Criteria: All HTTP requests to HSTS domains are automatically upgraded to HTTPS AND HSTS policies are enforced persistently

Fail Criteria: HTTP requests are not upgraded OR HSTS policies can be bypassed OR policies do not persist

Evidence: Network logs showing HTTP to HTTPS upgrades, browser HSTS storage screenshots, packet captures showing only HTTPS connections, console logs

References:

- RFC 6797 - HTTP Strict Transport Security (HSTS): https://datatracker.ietf.org/doc/html/rfc6797
- HSTS Preload List: https://hstspreload.org/
- OWASP HSTS Cheat Sheet: https://cheatsheetseries.owasp.org/cheatsheets/HTTP_Strict_Transport_Security_Cheat_
- MDN HSTS Documentation: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Strict-Transport-Security
- Chromium HSTS Implementation: https://www.chromium.org/hsts/

Assessment: ENC-REQ-5 (Mixed content blocking)

Reference: ENC-REQ-5 - Browser shall block mixed content (HTTP resources on HTTPS pages) to prevent downgrade attacks

Given: A conformant browser with encryption capability (ENC-1 or higher)

Task: Verify that mixed content blocking prevents protocol downgrade attacks where attackers inject HTTP resources into HTTPS pages to intercept data or execute malicious scripts. Blocking mixed content maintains the security guarantees of HTTPS by preventing unencrypted channels that could be exploited by network-level attackers to compromise the page's integrity and confidentiality.

Verification:

1. Create an HTTPS test page (https://test.example.com) that loads various HTTP resources:
   - HTTP scripts: `<script src="http://example.com/script.js">`
   - HTTP stylesheets: `<link href="http://example.com/style.css">`
   - HTTP iframes: `<iframe src="http://example.com/page.html">`
   - HTTP images: `<img src="http://example.com/image.jpg">`
   - HTTP XHR/fetch requests
   - HTTP form submission actions
2. Navigate to the test page and observe browser behavior

3. Verify that active mixed content (scripts, stylesheets, iframes) is blocked
4. Test browser's handling of passive mixed content (images, audio, video)
5. Check browser console for mixed content warnings
6. Test Content-Security-Policy: block-all-mixed-content directive
7. Verify that upgrade-insecure-requests CSP directive upgrades HTTP to HTTPS
8. Test mixed content indicator in browser address bar
9. Active mixed content (scripts, stylesheets, iframes) is completely blocked
10. Browser console displays clear mixed content blocking messages
11. Passive mixed content may load with warnings (or be blocked in strict mode)
12. Mixed content indicator appears in browser address bar/security indicator
13. CSP block-all-mixed-content directive blocks all mixed content
14. CSP upgrade-insecure-requests upgrades HTTP resources to HTTPS
15. Form submissions to HTTP URLs are blocked or warned
16. Users are warned about security implications of mixed content

Pass Criteria: All active mixed content is blocked AND clear warnings are displayed in console and UI

Fail Criteria: Active mixed content loads successfully OR no warnings are displayed

Evidence: Browser console screenshots showing blocked resources, network panel showing blocked requests, security indicator screenshots, DevTools showing CSP violations

References:

- W3C Mixed Content Specification: https://w3c.github.io/webappsec-mixed-content/
- MDN Mixed Content Documentation: https://developer.mozilla.org/en-US/docs/Web/Security/Mixed_content
- Chrome Mixed Content Policy: https://blog.chromium.org/2019/10/no-more-mixed-messages-about-https.html
- Upgrade Insecure Requests: https://www.w3.org/TR/upgrade-insecure-requests/

Assessment: ENC-REQ-6 (Certificate Transparency)

Reference: ENC-REQ-6 - Browser shall enforce Certificate Transparency (CT) requirements for all publicly trusted certificates

Given: A conformant browser with encryption capability (ENC-2 or higher)

Task: Verify that Certificate Transparency enforcement detects rogue certificates issued by compromised or malicious certificate authorities, preventing attackers from using fraudulently issued certificates for man-in-the-middle attacks. CT ensures all publicly trusted certificates are logged in append-only public ledgers, making unauthorized certificate issuance detectable and accountable.

Verification:

1. Set up test servers with certificates that have different CT compliance levels:
   - Certificate with embedded SCTs (Signed Certificate Timestamps)
   - Certificate with SCTs delivered via TLS extension
   - Certificate with SCTs delivered via OCSP stapling
   - Certificate without any SCTs
2. Navigate to each test server and observe browser behavior
3. Verify that browser requires CT for publicly trusted certificates
4. Check DevTools Security tab for CT information
5. Test that certificates without valid SCTs are rejected
6. Verify that browser validates SCT signatures
7. Test CT enforcement for different certificate types (DV, OV, EV)
8. Examine browser's CT policy compliance
9. Browser accepts certificates with valid SCTs
10. Certificates without SCTs are rejected with security errors
11. DevTools Security tab displays CT compliance status

12. Browser validates SCT signatures from known logs
13. CT is enforced for all publicly trusted certificates
14. Multiple SCT delivery mechanisms are supported
15. Browser maintains list of trusted CT log servers
16. CT violations trigger certificate errors

Pass Criteria: Browser enforces CT requirements AND rejects certificates without valid SCTs

Fail Criteria: Browser accepts certificates without SCTs OR does not validate SCT signatures

Evidence: DevTools Security tab screenshots showing CT status, certificate error dialogs for non-CT certificates, network logs showing SCT validation, browser CT policy configuration

References:

- RFC 6962 - Certificate Transparency: https://datatracker.ietf.org/doc/html/rfc6962
- Chrome CT Policy: https://github.com/chromium/ct-policy/blob/master/ct_policy.md
- Certificate Transparency Overview: https://certificate.transparency.dev/
- Google CT Log List: https://www.gstatic.com/ct/log_list/v3/all_logs_list.json
- CT Monitoring: https://developers.google.com/web/updates/2017/10/certificate-transparency

Assessment: ENC-REQ-7 (OCSP stapling)

Reference: ENC-REQ-7 - Browser shall support OCSP stapling for efficient certificate revocation checking

Given: A conformant browser with encryption capability (ENC-1 or higher)

Task: Verify that OCSP stapling support enables efficient certificate revocation checking without privacy leaks, preventing attackers from using revoked certificates while avoiding OCSP responder queries that reveal user browsing patterns. Stapling moves the revocation check to the server, improving performance and privacy while maintaining security against compromised certificates.

Verification:

1. Configure a test server to support OCSP stapling (include OCSP response in TLS handshake)
2. Navigate to the test server and capture the TLS handshake using packet capture tools
3. Verify that browser receives and validates the stapled OCSP response
4. Test server without OCSP stapling and verify browser falls back to OCSP responder query
5. Configure OCSP response indicating certificate is revoked and verify browser rejection
6. Test OCSP response validation including signature verification
7. Measure performance difference between stapled and non-stapled OCSP
8. Test OCSP Must-Staple certificate extension enforcement
9. Verify handling of expired or invalid OCSP responses
10. Browser successfully receives and validates stapled OCSP responses
11. Packet captures show OCSP response in TLS handshake (CertificateStatus message)
12. Browser validates OCSP response signatures
13. Revoked certificates are rejected based on OCSP response
14. Browser falls back to direct OCSP query when stapling unavailable
15. OCSP Must-Staple extension is enforced (if present)
16. Performance metrics show faster connection with stapling
17. Invalid OCSP responses trigger appropriate error handling

Pass Criteria: Browser correctly validates stapled OCSP responses AND rejects revoked certificates

Fail Criteria: Browser ignores OCSP responses OR accepts revoked certificates

Evidence: Packet captures showing stapled OCSP in TLS handshake, certificate error screenshots for revoked certificates, performance measurements, DevTools Security tab showing OCSP status

References:

- RFC 6066 - TLS Extension: OCSP Status Request: https://datatracker.ietf.org/doc/html/rfc6066#section-8
- RFC 6960 - Online Certificate Status Protocol (OCSP): https://datatracker.ietf.org/doc/html/rfc6960
- OCSP Stapling Explained: https://www.digicert.com/kb/ssl-support/what-is-ocsp-stapling.htm
- OCSP Must-Staple: https://scotthelme.co.uk/ocsp-must-staple/

Assessment: ENC-REQ-8 (Cipher suite restrictions)

Reference: ENC-REQ-8 - Browser shall support only secure cipher suites and disable weak or deprecated algorithms

Given: A conformant browser with encryption capability (ENC-1 or higher)

Task: Verify that cipher suite restrictions prevent cryptographic weaknesses from being exploited by attackers to break TLS encryption, ensuring only modern algorithms resistant to known attacks are used. Rejecting weak ciphers like RC4, DES, and export-grade algorithms prevents downgrade attacks and ensures forward security against future cryptanalysis.

Verification:

1. Configure test servers with various cipher suites:
   - Strong: TLS_AES_256_GCM_SHA384, TLS_CHACHA20_POLY1305_SHA256
   - Acceptable: TLS_AES_128_GCM_SHA256, ECDHE-RSA-AES128-GCM-SHA256
   - Weak: RC4, DES, 3DES, NULL ciphers
   - Export-grade ciphers
2. Test browser connection to each server configuration
3. Verify that browser refuses weak cipher suites
4. Capture cipher suite negotiation using packet analysis
5. Test browser's cipher suite preference order
6. Verify that AEAD (Authenticated Encryption with Associated Data) ciphers are preferred
7. Test that CBC-mode ciphers are avoided or deprioritized
8. Examine browser's supported cipher suite list
9. Test cipher suite restrictions for different TLS versions
10. Browser successfully negotiates strong cipher suites
11. Weak cipher suites (RC4, DES, 3DES, NULL) are rejected
12. Connection fails when only weak ciphers are available
13. Browser prefers AEAD ciphers over CBC-mode ciphers
14. Cipher suite negotiation follows secure ordering
15. Export-grade ciphers are completely disabled
16. DevTools shows negotiated cipher suite
17. Browser supports all modern recommended cipher suites

Pass Criteria: Browser only accepts strong cipher suites AND rejects all weak or deprecated ciphers

Fail Criteria: Browser accepts any weak cipher suite OR fails to negotiate strong ciphers when available

Evidence: Packet captures showing cipher suite negotiation, DevTools Security tab showing negotiated cipher, server logs, connection error screenshots for weak cipher servers

References:

- Mozilla SSL Configuration Generator: https://ssl-config.mozilla.org/
- NIST SP 800-52 Rev. 2 - TLS Guidelines: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-52r2.pdf
- RFC 8446 - TLS 1.3 Cipher Suites: https://datatracker.ietf.org/doc/html/rfc8446#appendix-B.4
- IANA TLS Cipher Suite Registry: https://www.iana.org/assignments/tls-parameters/tls-parameters.xhtml#tls-parameters-4
- Qualys SSL Labs - Cipher Suite Guidance: https://github.com/ssllabs/research/wiki/SSL-and-TLS-Deployment-Best-Practices#23-use-secure-cipher-suites

Assessment: ENC-REQ-9 (Perfect forward secrecy)

Reference: ENC-REQ-9 - Browser shall support and prefer cipher suites with perfect forward secrecy (PFS)

Given: A conformant browser with encryption capability (ENC-2 or higher)

Task: Verify that perfect forward secrecy prevents retrospective decryption attacks where adversaries record encrypted traffic and later compromise server private keys to decrypt historical sessions. PFS ensures that even if long-term keys are compromised, past TLS sessions remain secure because session keys were ephemeral and never stored.

Verification:

1. Configure test servers with different key exchange mechanisms:
   - ECDHE (Elliptic Curve Diffie-Hellman Ephemeral) - provides PFS
   - DHE (Diffie-Hellman Ephemeral) - provides PFS
   - RSA key exchange - no PFS
2. Test browser connection to each server configuration
3. Verify that browser prefers ECDHE/DHE over static RSA key exchange
4. Capture TLS handshake to confirm ephemeral key exchange
5. Test that browser supports multiple elliptic curves (X25519, P-256, P-384)
6. Verify proper DH parameter size enforcement (minimum 2048 bits)
7. Test TLS 1.3 behavior (all cipher suites provide PFS)
8. Examine cipher suite preference order for PFS priority
9. Test connection failure when only non-PFS ciphers are available
10. Browser successfully negotiates ECDHE/DHE cipher suites
11. Browser prefers PFS cipher suites over non-PFS alternatives
12. Packet captures show ephemeral key exchange in handshake
13. Browser supports modern elliptic curves (X25519 preferred)
14. DHE parameters meet minimum security requirements (2048+ bits)
15. TLS 1.3 connections always provide PFS
16. DevTools Security tab indicates PFS status
17. Browser may reject or deprioritize non-PFS cipher suites

Pass Criteria: Browser prefers PFS cipher suites when available AND successfully negotiates ephemeral key exchange

Fail Criteria: Browser prefers non-PFS ciphers over PFS alternatives OR fails to negotiate PFS key exchange

Evidence: Packet captures showing ECDHE/DHE key exchange, DevTools Security tab showing PFS cipher suite, connection test results, cipher suite preference analysis

References:

- RFC 8446 - TLS 1.3 (all cipher suites provide PFS): https://datatracker.ietf.org/doc/html/rfc8446
- Perfect Forward Secrecy Explained: https://scotthelme.co.uk/perfect-forward-secrecy/
- OWASP TLS Cheat Sheet - Forward Secrecy: https://cheatsheetseries.owasp.org/cheatsheets/Transport_Layer_Protectio secrecy
- Mozilla Server Side TLS Guidelines: https://wiki.mozilla.org/Security/Server_Side_TLS

Assessment: ENC-REQ-10 (Revocation checking)

Reference: ENC-REQ-10 - Browser shall perform certificate revocation checking via OCSP and/or CRL mechanisms

Given: A conformant browser with encryption capability (ENC-1 or higher)

Task: Verify that certificate revocation checking prevents attackers from using compromised or fraudulently obtained certificates that have been revoked by the issuing CA. Checking revocation status ensures browsers

reject certificates known to be untrustworthy, protecting users from man-in-the-middle attacks using stolen or misused certificates even after they've been marked as invalid.

Verification:

1. Set up test certificates with different revocation configurations:
   - Valid certificate with OCSP responder URL
   - Valid certificate with CRL distribution point
   - Revoked certificate (marked as revoked in OCSP/CRL)
   - Certificate without revocation information
2. Navigate to servers using each certificate type
3. Verify that browser checks certificate revocation status
4. Test OCSP responder queries (use network monitoring)
5. Test CRL download and parsing
6. Configure OCSP responder to return "revoked" status and verify browser rejection
7. Test soft-fail vs hard-fail behavior when revocation check is unavailable
8. Measure performance impact of revocation checking
9. Test browser's revocation checking cache behavior
10. Browser performs OCSP queries for certificates with OCSP URLs
11. Browser downloads and processes CRLs when available
12. Revoked certificates are rejected with clear error messages
13. Browser caches revocation check results appropriately
14. OCSP responses are validated (signature, freshness)
15. CRL validity period is checked
16. Browser handles revocation check failures gracefully
17. Network panel shows OCSP/CRL requests

Pass Criteria: Browser performs revocation checks AND rejects revoked certificates with clear error messages

Fail Criteria: Browser skips revocation checking OR accepts revoked certificates

Evidence: Network logs showing OCSP/CRL requests, certificate error screenshots for revoked certificates, packet captures, DevTools showing revocation status, performance measurements

References:

- RFC 6960 - Online Certificate Status Protocol (OCSP): https://datatracker.ietf.org/doc/html/rfc6960
- RFC 5280 - Certificate and CRL Profile: https://datatracker.ietf.org/doc/html/rfc5280
- Chrome CRLSet Mechanism: https://www.imperialviolet.org/2012/02/05/crlsets.html
- OWASP Certificate Revocation: https://owasp.org/www-community/controls/Certificate_and_Public_Key_Pinning

Assessment: ENC-REQ-11 (Web Crypto API compliance)

Reference: ENC-REQ-11 - Browser shall implement W3C Web Cryptography API for JavaScript cryptographic operations

Given: A conformant browser with encryption capability (ENC-1 or higher)

Task: Verify that Web Crypto API compliance provides secure, browser-native cryptographic operations preventing developers from implementing insecure custom cryptography that could be vulnerable to timing attacks, weak random number generation, or algorithmic flaws. Restricting the API to secure contexts ensures cryptographic keys and operations are only available over HTTPS, preventing network attackers from intercepting keys.

Verification:

1. Create test web page accessing window.crypto.subtle API
2. Test key generation for various algorithms:
   - RSA-OAEP, RSA-PSS (2048, 3072, 4096 bits)
   - ECDSA, ECDH (P-256, P-384, P-521 curves)

    - AES-GCM, AES-CBC (128, 256 bits)
    - HMAC with various hash functions
3. Test encryption/decryption operations with generated keys
4. Test signing and verification operations
5. Test key derivation functions (PBKDF2, HKDF)
6. Test key import/export in various formats (JWK, PKCS8, SPKI, raw)
7. Verify that subtle crypto operations are only available in secure contexts (HTTPS)
8. Test that cryptographic keys can be marked as non-extractable
9. Test key usage constraints (encrypt, decrypt, sign, verify, etc.)
10. Verify proper error handling for invalid parameters
11. window.crypto.subtle is available in secure contexts only
12. All mandatory algorithms are supported
13. Key generation produces keys of requested type and size
14. Encrypt/decrypt operations work correctly with proper padding
15. Sign/verify operations validate signatures correctly
16. Key derivation functions produce deterministic results
17. Non-extractable keys cannot be exported
18. Key usage constraints are enforced
19. Operations fail appropriately in insecure contexts (HTTP)
20. Error messages are clear and informative

Pass Criteria: All Web Crypto API operations function correctly AND API is restricted to secure contexts

Fail Criteria: Required algorithms are missing OR API is available in insecure contexts OR operations produce incorrect results

Evidence: Console logs showing successful crypto operations, test result screenshots, code demonstrating API usage, error messages for insecure context access

References:

- W3C Web Cryptography API Specification: https://www.w3.org/TR/WebCryptoAPI/
- MDN Web Crypto API Documentation: https://developer.mozilla.org/en-US/docs/Web/API/Web_Crypto_API
- Web Crypto API Examples: https://github.com/diafygi/webcrypto-examples
- OWASP Cryptographic Storage Cheat Sheet: https://cheatsheetseries.owasp.org/cheatsheets/Cryptographic_Storage_Ch
- Chrome Web Crypto Implementation: https://www.chromium.org/blink/webcrypto/

Assessment: ENC-REQ-12 (Secure random number generation)

Reference: ENC-REQ-12 - Browser shall provide cryptographically secure random number generation via Crypto.getRandomValues()

Given: A conformant browser with encryption capability (ENC-1 or higher)

Task: Verify that cryptographically secure random number generation prevents predictability attacks where weak randomness allows attackers to guess session tokens, cryptographic keys, or authentication nonces. High-quality entropy ensures generated values are unpredictable and cannot be reproduced, which is essential for secure key generation, token creation, and cryptographic operations.

Verification:

1. Create test page calling window.crypto.getRandomValues() with various typed arrays:
    - Uint8Array, Uint16Array, Uint32Array
    - Different array sizes (1 byte to 65536 bytes)
2. Generate large samples of random data (millions of bytes)
3. Test statistical properties of generated random data:
    - Frequency test (chi-square)
    - Runs test
    - Entropy analysis

4. Verify that repeated calls produce different values
5. Test that maximum size limit (65536 bytes) is enforced
6. Verify proper error handling for invalid parameters
7. Test that random values are available in both secure and insecure contexts
8. Compare randomness quality across multiple browser sessions
9. Test performance of random number generation
10. window.crypto.getRandomValues() is available and functional
11. Generated values pass statistical randomness tests
12. Repeated calls produce different, unpredictable values
13. All typed array types are supported
14. Maximum size limit (65536 bytes) is enforced
15. QuotaExceededError thrown for oversized requests
16. Random values have sufficient entropy (>= 7.9 bits per byte)
17. Performance is acceptable (can generate MBs per second)
18. API is available in all contexts (secure and insecure)

Pass Criteria: Generated random values pass statistical tests AND have sufficient entropy for cryptographic use

Fail Criteria: Random values show statistical bias OR have insufficient entropy OR values are predictable

Evidence: Statistical test results, entropy analysis graphs, performance benchmarks, console logs showing API usage, test code with results

References:

- W3C Web Cryptography API - getRandomValues: https://www.w3.org/TR/WebCryptoAPI/#Crypto-method-getRandomValues
- MDN Crypto.getRandomValues(): https://developer.mozilla.org/en-US/docs/Web/API/Crypto/getRandomValues
- NIST SP 800-22 - Statistical Tests for Random Number Generators: https://csrc.nist.gov/publications/detail/sp/800-22/rev-1a/final
- Random.org - Statistical Tests: https://www.random.org/analysis/
- OWASP Cryptographic Storage - Random Number Generation: https://cheatsheetseries.owasp.org/cheatsheets/Cryptogra random-number-generation

Assessment: ENC-REQ-13 (SubResource Integrity)

Reference: ENC-REQ-13 - Browser shall enforce Subresource Integrity (SRI) for externally loaded scripts and stylesheets

Given: A conformant browser with encryption capability (ENC-1 or higher)

Task: Verify that Subresource Integrity enforcement prevents supply chain attacks where compromised CDNs or third-party servers deliver malicious modified scripts or stylesheets. SRI ensures that external resources match their expected cryptographic hashes, detecting unauthorized modifications that could inject malware, steal credentials, or compromise user data even if the hosting server is compromised.

Verification:

1. Create test page loading external resources with integrity attributes:

```
<script src="https://cdn.example.com/lib.js"
        integrity="sha384-oqVuAfXRKap7fdgcCY5uykM6+R9GqQ8K/uxy9rx7HNQlGYl1kPzQho1wx4JwY8wC"
        crossorigin="anonymous"></script>
<link href="https://cdn.example.com/style.css"
      integrity="sha384-T3c6CoIi6uLrA9TneNEoa7RxnatzjcDSCmG1MXxSR1GAsXEV/Dwwykc2MPK8M2HN"
      crossorigin="anonymous">
```

2. Verify that resources with correct integrity hashes load successfully

3. Modify resource content to mismatch integrity hash and verify blocking
4. Test multiple hash algorithms: sha256, sha384, sha512
5. Test multiple integrity values (fallback hashes)
6. Test CORS requirement for cross-origin SRI resources
7. Verify browser console errors for SRI failures
8. Test SRI with different resource types (scripts, stylesheets, preload links)
9. Test that SRI failures prevent script execution/style application
10. Resources with matching integrity hashes load and execute successfully
11. Resources with mismatched hashes are blocked from loading
12. Browser console displays clear SRI violation errors
13. Multiple hash algorithms (sha256, sha384, sha512) are supported
14. Multiple integrity values allow fallback verification
15. Cross-origin resources require proper CORS headers for SRI
16. SRI failures prevent resource execution/application
17. Network panel shows blocked resources with SRI violations
18. Page functionality degrades gracefully when SRI blocks resources

Pass Criteria: Resources with correct integrity hashes load successfully AND resources with mismatched hashes are blocked with clear errors

Fail Criteria: Resources with mismatched hashes load and execute OR no error messages displayed

Evidence: Browser console screenshots showing SRI errors, network panel showing blocked resources, test page demonstrating SRI enforcement, DevTools showing integrity attribute validation

References:

- W3C Subresource Integrity Specification: https://www.w3.org/TR/SRI/
- MDN Subresource Integrity: https://developer.mozilla.org/en-US/docs/Web/Security/Subresource_Integrity
- SRI Hash Generator: https://www.srihash.org/
- OWASP SRI Guidance: https://cheatsheetseries.owasp.org/cheatsheets/Third_Party_Javascript_Management_Cheat_integrity
- Chrome SRI Implementation: https://www.chromium.org/Home/chromium-security/education/tls/

Assessment: ENC-REQ-14 (Encrypted SNI)

Reference: ENC-REQ-14 - Browser shall support Encrypted Server Name Indication (ESNI/ECH) to prevent SNI-based censorship and surveillance

Given: A conformant browser with encryption capability (ENC-3 or higher)

Task: Verify that Encrypted SNI support prevents network surveillance and censorship where adversaries monitor TLS handshakes to identify which websites users visit, enabling targeted blocking or surveillance. Encrypting SNI protects user privacy by hiding the destination hostname from network observers, preventing traffic analysis attacks that reveal browsing patterns even when connections use HTTPS.

Verification:

1. Configure a test server supporting Encrypted Client Hello (ECH) or ESNI
2. Publish ECH configuration in DNS (TLS HTTPS record type 65)
3. Navigate browser to the test server

4. Capture TLS handshake using packet analysis tools
5. Verify that SNI extension is encrypted in ClientHello message
6. Test fallback behavior when ECH is unavailable
7. Test that cleartext SNI is not visible in packet captures
8. Verify browser DNS-over-HTTPS (DoH) or DNS-over-TLS (DoT) usage for ECH config retrieval
9. Test ECH with split-horizon DNS configurations
10. Examine browser settings for ECH/ESNI enablement
11. Browser successfully negotiates ECH/ESNI when available
12. SNI extension is not visible in cleartext in packet captures
13. ClientHello contains encrypted_client_hello extension
14. Browser retrieves ECH configuration via DNS
15. Fallback to cleartext SNI works when ECH unavailable
16. DNS queries for ECH config are encrypted (DoH/DoT)
17. Browser DevTools or internal pages show ECH status
18. Connection succeeds with ECH-enabled servers

Pass Criteria: Browser encrypts SNI when ECH is available AND cleartext SNI is not visible in packet captures

Fail Criteria: SNI is transmitted in cleartext when ECH is available OR browser doesn't support ECH

Evidence: Packet captures showing encrypted ClientHello, Wireshark analysis showing absence of cleartext SNI, DNS query logs showing ECH config retrieval, browser configuration screenshots

References:

- RFC 8744 - Encrypted Server Name Indication (ESNI): https://datatracker.ietf.org/doc/html/rfc8744
- Encrypted Client Hello (ECH) Draft: https://datatracker.ietf.org/doc/html/draft-ietf-tls-esni
- Cloudflare ECH Announcement: https://blog.cloudflare.com/encrypted-client-hello/
- Mozilla ECH Implementation: https://support.mozilla.org/en-US/kb/understand-encrypted-client-hello
- Chrome ECH Status: https://chromestatus.com/feature/6196703843581952

Assessment: ENC-REQ-15 (Certificate error UI)

Reference: ENC-REQ-15 - Browser shall display clear, prominent warnings for certificate errors with appropriate risk communication

Given: A conformant browser with encryption capability (ENC-1 or higher)

Task: Verify that certificate error UI effectively warns users about man-in-the-middle attacks or compromised servers before they transmit sensitive data to untrusted connections. Clear, prominent warnings with appropriate risk communication prevent users from inadvertently trusting malicious certificates, while making bypass actions deliberately difficult discourages risky behavior that could expose credentials or personal information.

Verification:

1. Create test scenarios for various certificate errors:
   - Expired certificate
   - Untrusted CA
   - Hostname mismatch
   - Self-signed certificate
   - Revoked certificate
   - Weak signature algorithm (SHA-1)
   - Invalid certificate chain
2. Navigate to each error scenario and document browser UI response
3. Verify that warning messages clearly communicate security risk
4. Test that warnings are difficult to bypass (require explicit user action)
5. Evaluate warning message clarity for non-technical users

6. Test that technical details are available (certificate viewer)
7. Verify that bypass actions are clearly labeled with risk warnings
8. Test mobile and desktop warning UI differences
9. Verify that errors are logged in browser console
10. Certificate errors trigger full-page interstitial warnings
11. Warning messages clearly explain the security risk
12. Users should take explicit action to bypass (not easy clickthrough)
13. Technical details are accessible via "Advanced" or similar link
14. Certificate details can be viewed and inspected
15. Warning UI uses appropriate visual indicators (red, warning icons)
16. Bypass options are clearly labeled with risk warnings
17. Error types are distinguishable in UI messages
18. Console logs provide technical error details

Pass Criteria: All certificate errors display prominent warnings AND bypass requires explicit user acknowledgment of risk

Fail Criteria: Certificate errors can be bypassed silently OR warnings are unclear or easily dismissed

Evidence: Screenshots of warning UI for each error type, user testing feedback on clarity, console error logs, comparison with browser security UI guidelines

References:

- Google Transparency Report - HTTPS Security: https://transparencyreport.google.com/https/overview

Assessment: ENC-REQ-16 (HTTPS-first mode)

Reference: ENC-REQ-16 - Browser shall implement HTTPS-first mode to automatically upgrade HTTP connections to HTTPS when available

Given: A conformant browser with encryption capability (ENC-2 or higher)

Task: Verify that HTTPS-first mode protects users from accidental insecure connections where attackers perform SSL-stripping attacks to downgrade HTTPS to HTTP, enabling interception of credentials and session tokens. Automatic HTTPS upgrade eliminates the window of vulnerability before secure connections are established, while user warnings for HTTP-only sites ensure informed consent before transmitting data over insecure channels.

Verification:

1. Enable HTTPS-first mode in browser settings
2. Navigate to HTTP URLs of sites that support HTTPS (http://example.com)
3. Verify that browser automatically upgrades to HTTPS
4. Test fallback behavior when HTTPS is unavailable:
    - Site doesn't support HTTPS
    - HTTPS connection fails/times out
    - Certificate error on HTTPS version
5. Verify user is prompted before loading HTTP-only sites
6. Test HTTPS upgrade for embedded resources (images, scripts, iframes)
7. Test interaction with HSTS and upgrade-insecure-requests
8. Measure performance impact of HTTPS upgrade attempts
9. Test HTTPS-first with different types of navigation (typed URL, bookmarks, links)
10. Verify browser remembers HTTP-only sites to avoid repeated upgrade attempts
11. HTTP URLs are automatically upgraded to HTTPS
12. Address bar shows HTTPS protocol after upgrade
13. Fallback to HTTP works with user consent when HTTPS unavailable
14. User is warned before loading HTTP-only sites
15. Browser remembers HTTP-only sites (cache/allowlist)

16. Embedded resources are also upgraded
17. HTTPS-first works alongside HSTS
18. Network panel shows upgrade attempts
19. Performance impact is minimal (parallel attempts)

Pass Criteria: HTTP connections are automatically upgraded to HTTPS when available AND users are warned before HTTP-only sites load

Fail Criteria: HTTP URLs are not upgraded OR no warnings for HTTP-only sites OR fallback doesn't work

Evidence: Network logs showing HTTP to HTTPS upgrades, address bar screenshots, warning dialog screenshots, performance measurements, browser settings showing HTTPS-first configuration

References:

- Chrome HTTPS-First Mode: https://blog.chromium.org/2021/07/increasing-https-adoption.html
- Firefox HTTPS-Only Mode: https://support.mozilla.org/en-US/kb/https-only-prefs
- HTTPS Upgrade Mechanisms: https://www.w3.org/TR/upgrade-insecure-requests/
- EFF HTTPS Everywhere: https://www.eff.org/https-everywhere
- OWASP Transport Layer Protection: https://cheatsheetseries.owasp.org/cheatsheets/Transport_Layer_Protection_Chea

Assessment: ENC-REQ-17 (Certificate pinning bypass detection)

Reference: ENC-REQ-17 - Browser shall detect and prevent attempts to bypass certificate pinning protections

Given: A conformant browser with encryption capability (ENC-2 or higher)

Task: Verify that certificate pinning bypass detection prevents attackers or malware from installing rogue root certificates to perform man-in-the-middle attacks against pinned domains. Detecting bypass attempts protects browser vendor properties and high-security sites from SSL interception, while logging and user warnings ensure transparency when certificate validation is weakened by enterprise policies or security tools.

Verification:

1. Configure test environment with certificate pinning enabled
2. Attempt various bypass techniques:
   - Installing custom root CA certificates
   - Using SSL/TLS interception proxies (corporate MITM)
   - Modifying browser certificate store
   - Using browser extensions to disable pinning
   - Command-line flags to disable certificate validation
3. Test browser's built-in static pins (Google, Mozilla properties)
4. Verify that pin bypass attempts are detected and logged
5. Test enterprise policy controls for pinning exceptions
6. Verify user notifications for certificate store modifications
7. Test that developer tools can't silently bypass pinning
8. Examine browser internal state for pin enforcement
9. Test interaction between pin bypass and security indicators
10. Static pins for built-in domains cannot be bypassed
11. Custom root CA installation triggers user warnings
12. SSL interception is detected and indicated in UI
13. Browser logs pin bypass attempts
14. Certificate store modifications are visible to users
15. Enterprise policies can override pins with explicit configuration
16. Developer tools respect pinning (or show clear bypass warnings)
17. Security indicators reflect weakened security when pins bypassed
18. Console logs show certificate validation details

Pass Criteria: Static certificate pins cannot be bypassed without explicit user/admin action AND pin bypass attempts are logged and indicated

Fail Criteria: Pins can be silently bypassed OR no indication when certificate validation is weakened

Evidence: Console logs showing pin enforcement, certificate store modification warnings, test results from bypass attempts, enterprise policy documentation, security indicator screenshots

References:

- Chrome Certificate Pinning: https://www.chromium.org/Home/chromium-security/education/tls/
- OWASP Certificate Pinning: https://owasp.org/www-community/controls/Certificate_and_Public_Key_Pinning
- SSL Interception Detection: https://blog.mozilla.org/security/2015/04/30/deprecating-non-secure-http/
- Mozilla Pin Override: https://wiki.mozilla.org/SecurityEngineering/Public_Key_Pinning

Assessment: ENC-REQ-18 (TLS downgrade protection)

Reference: ENC-REQ-18 - Browser shall implement protections against TLS version and cipher suite downgrade attacks

Given: A conformant browser with encryption capability (ENC-1 or higher)

Task: Verify that TLS downgrade protection prevents man-in-the-middle attackers from forcing browsers to use older TLS versions or weaker cipher suites with known vulnerabilities like POODLE or BEAST. Downgrade protection ensures that even when attackers intercept and modify handshake messages, the browser detects the manipulation and aborts the connection rather than proceeding with weakened cryptographic parameters.

Verification:

1. Set up test environment capable of simulating man-in-the-middle attacks
2. Configure server supporting TLS 1.3 and TLS 1.2
3. Attempt to force downgrade from TLS 1.3 to TLS 1.2 by manipulating ClientHello
4. Test TLS_FALLBACK_SCSV signaling value (RFC 7507)
5. Attempt downgrade attacks during connection:
    - Version rollback to older TLS versions
    - Cipher suite downgrade to weaker algorithms
    - Extension stripping attacks
6. Verify browser detects and rejects downgrade attempts
7. Test that Finished message MAC includes all handshake messages
8. Verify TLS 1.3 downgrade protection sentinel values in ServerHello.random
9. Test protection against truncation attacks
10. Browser signals maximum supported TLS version correctly
11. TLS_FALLBACK_SCSV is included in fallback connections
12. Version rollback attacks are detected and connection aborted
13. Cipher suite downgrade attempts trigger handshake failure
14. Browser validates ServerHello.random for downgrade sentinels
15. Extension stripping is detected through transcript hash validation
16. Finished message properly authenticates handshake
17. Console shows error messages for detected downgrade attempts
18. Connection fails securely rather than completing with weakened security

Pass Criteria: All TLS downgrade attempts are detected AND connections fail rather than proceed with weakened security

Fail Criteria: Any downgrade attack succeeds OR browser accepts weakened connection parameters

Evidence: Packet captures showing downgrade attempts and rejection, Wireshark showing TLS_FALLBACK_SCSV, console error logs, test scripts demonstrating attack attempts

References:

- RFC 7507 - TLS Fallback SCSV: https://datatracker.ietf.org/doc/html/rfc7507
- RFC 8446 - TLS 1.3 Downgrade Protection: https://datatracker.ietf.org/doc/html/rfc8446#section-4.1.3
- POODLE Attack and Downgrade Prevention: https://www.openssl.org/~bodo/ssl-poodle.pdf
- Chrome TLS Implementation: https://www.chromium.org/Home/chromium-security/education/tls/

Assessment: ENC-REQ-19 (Legacy crypto deprecation)

Reference: ENC-REQ-19 - Browser shall deprecate and remove support for legacy cryptographic algorithms and protocols

Given: A conformant browser with encryption capability (ENC-1 or higher)

Task: Verify that legacy crypto deprecation prevents attackers from exploiting known cryptographic weaknesses in outdated algorithms like SHA-1 collision attacks, RC4 biases, or short RSA key factorization. Progressive deprecation with clear timelines gives organizations migration paths while ensuring browsers eventually reject severely compromised cryptography that no longer provides meaningful security guarantees.

Verification:

1. Test browser behavior with legacy cryptographic elements:
   - TLS 1.0 and TLS 1.1 protocols
   - SHA-1 certificates
   - 1024-bit RSA keys
   - MD5-based signatures
   - RC4 cipher suite
   - CBC-mode cipher suites
   - DSA certificates
2. Verify that legacy protocols/algorithms are rejected or trigger warnings
3. Test deprecation timeline (when were features removed)
4. Verify that browser update notes document deprecated features
5. Test enterprise policy overrides for legacy support (temporary exceptions)
6. Check browser developer documentation for deprecation roadmap
7. Test fallback behavior when modern crypto unavailable
8. Verify that critical errors can't be bypassed for severely deprecated crypto
9. TLS 1.0 and 1.1 connections are rejected or show warnings
10. SHA-1 certificates trigger security errors
11. 1024-bit RSA keys are rejected
12. MD5 and RC4 are completely disabled
13. Legacy crypto rejections show clear error messages
14. Browser documentation lists deprecated features with timelines
15. Enterprise policies can temporarily enable legacy support (if necessary)
16. No silent fallback to insecure legacy protocols
17. Console logs indicate when legacy crypto is encountered

Pass Criteria: All severely deprecated cryptographic elements are rejected AND users are warned about moderately deprecated features

Fail Criteria: Legacy crypto is accepted without warnings OR deprecated features work without indication

Evidence: Connection error screenshots for legacy servers, browser release notes documenting deprecations, console error logs, test results across browser versions showing deprecation timeline

References:

- Chrome Deprecation Timeline: https://www.chromium.org/Home/chromium-security/education/tls/
- Mozilla Security Roadmap: https://wiki.mozilla.org/Security/Server_Side_TLS

- RFC 8996 - Deprecating TLS 1.0 and TLS 1.1: https://datatracker.ietf.org/doc/html/rfc8996
- CA/B Forum - SHA-1 Deprecation: https://cabforum.org/2014/10/16/ballot-118-sha-1-sunset/
- NIST Cryptographic Algorithm Deprecation: https://csrc.nist.gov/projects/hash-functions

Assessment: ENC-REQ-20 (Cryptographic key isolation)

Reference: ENC-REQ-20 - Browser shall isolate cryptographic keys and prevent cross-origin key access

Given: A conformant browser with encryption capability (ENC-2 or higher)

Task: Verify that cryptographic key isolation enforces same-origin policy for Web Crypto API keys, preventing malicious cross-origin scripts from accessing or exfiltrating cryptographic keys generated by other origins. Key isolation ensures that even if an attacker compromises one origin, they cannot steal cryptographic keys belonging to other origins to impersonate users or decrypt sensitive data protected by those keys.

Verification:

1. Create test scenarios with Web Crypto API key generation on different origins
2. Generate cryptographic keys on https://origin-a.com using Web Crypto API
3. Attempt to access keys from https://origin-b.com
4. Test IndexedDB key storage isolation across origins
5. Verify that keys marked as non-extractable cannot be extracted
6. Test key export restrictions based on key usage
7. Test that key handles cannot be passed between origins via postMessage
8. Verify key isolation in browser's internal key storage
9. Test hardware-backed key storage (if available, e.g., WebAuthn)
10. Test key isolation for different user profiles/contexts
11. Keys generated on one origin cannot be accessed from another origin
12. IndexedDB key storage respects same-origin policy
13. Non-extractable keys cannot be exported or extracted
14. Key usage restrictions are enforced (keys can't be used for unauthorized operations)
15. Key handles are opaque and cannot be transferred cross-origin
16. Browser internal key storage is isolated per origin
17. Hardware-backed keys are protected by platform security
18. Different user profiles have separate key storage
19. Attempts to access cross-origin keys throw SecurityError

Pass Criteria: Cryptographic keys are strictly isolated by origin AND non-extractable keys cannot be exported

Fail Criteria: Keys can be accessed across origins OR key usage restrictions can be bypassed

Evidence: Console logs showing SecurityError for cross-origin key access, test code demonstrating isolation, browser internal state showing key storage separation, WebAuthn test results

References:

- W3C Web Cryptography API - Key Storage: https://www.w3.org/TR/WebCryptoAPI/#concepts-key-storage
- MDN CryptoKey: https://developer.mozilla.org/en-US/docs/Web/API/CryptoKey
- Chrome Web Crypto Key Isolation: https://chromium.googlesource.com/chromium/src/+/master/components/webcrypt
- OWASP Key Management Cheat Sheet: https://cheatsheetseries.owasp.org/cheatsheets/Key_Management_Cheat_Shee
- WebAuthn Specification - Credential Storage: https://www.w3.org/TR/webauthn-2/#credential-storage

Assessment: ENC-REQ-21 (Certificate store security)

Reference: ENC-REQ-21 - Browser shall maintain secure certificate store with integrity protection and auditing

Given: A conformant browser with encryption capability (ENC-1 or higher)

Task: Verify that certificate store security prevents attackers from installing rogue root certificates to enable man-in-the-middle attacks against all TLS connections. Protected certificate stores with integrity checking and audit trails detect unauthorized modifications, while requiring explicit user consent for root CA changes prevents malware from silently compromising the foundation of TLS trust.

Verification:

1. Examine browser's certificate store location and access controls
2. Test that certificate store is protected with appropriate file system permissions
3. Attempt to modify certificate store while browser is running
4. Verify that certificate store changes are logged/auditable
5. Test root CA certificate management:
   - View installed root certificates
   - Add custom root CA (with user consent)
   - Remove root CA (with user consent)
   - Verify certificate trust settings
6. Test that system certificate store is used appropriately
7. Verify that enterprise-managed certificates are clearly indicated
8. Test certificate store integrity verification mechanisms
9. Test that certificate store updates are secure and authenticated
10. Verify user notifications for certificate store modifications
11. Certificate store files have restrictive permissions (not world-readable)
12. Certificate store cannot be modified without appropriate privileges
13. Browser detects and handles certificate store corruption
14. Root CA additions/removals require explicit user consent
15. Certificate store UI shows all installed root certificates
16. System certificate store integration works correctly
17. Enterprise-managed certificates are visibly marked
18. Certificate store modifications are logged
19. Users are notified of certificate store changes
20. Certificate store updates are signed and verified

Pass Criteria: Certificate store is protected with appropriate access controls AND modifications require user consent and are logged

Fail Criteria: Certificate store can be modified without user knowledge OR no audit trail for modifications

Evidence: File system permission analysis, certificate store UI screenshots, audit log samples, test results from modification attempts, enterprise policy documentation

References:

- Mozilla Root Store Program: https://www.mozilla.org/en-US/about/governance/policies/security-group/certs/
- Chrome Root Store: https://chromium.googlesource.com/chromium/src/+/main/net/data/ssl/chrome_root_store/
- Microsoft Trusted Root Program: https://learn.microsoft.com/en-us/security/trusted-root/program-requirements
- Apple Root Certificate Program: https://www.apple.com/certificateauthority/ca_program.html
- OWASP Certificate Management: https://cheatsheetseries.owasp.org/cheatsheets/Transport_Layer_Protection_Cheat_and-public-key-pinning

6.4 Security Event Logging Assessments

This section covers assessment procedures for requirements LOG-REQ-1 through LOG-REQ-20, addressing security event logging, audit trails, privacy-preserving telemetry, log retention, and security monitoring capabilities.

Assessment: LOG-REQ-1 (Security event logging)

Reference: LOG-REQ-1 - Browser shall implement comprehensive security event logging for security-relevant events

Given: A conformant browser with LOG-1 or higher capability

Task: Verify that comprehensive security event logging enables detection and investigation of attacks, providing audit trails for security incidents like XSS attempts, CSP violations, and CORS bypasses. Complete logging with timestamps, origins, and outcomes allows security teams to identify attack patterns, correlate events across multiple targets, and respond to threats effectively while supporting compliance with security monitoring requirements.

Verification:

1. Configure the browser to enable security event logging in developer or enterprise mode
2. Access the browser's internal logging interfaces (chrome://net-internals/#events, about:networking, or equivalent)
3. Navigate to a test page that triggers multiple security events (mixed content, certificate errors, XSS attempts)
4. Attempt cross-origin requests that violate CORS policies
5. Load test pages with Content Security Policy violations
6. Trigger sandbox violations by attempting to access restricted APIs
7. Review the logged events to verify all security-relevant actions are captured
8. Export the security event log to verify it contains timestamps, event types, origins, and outcomes
9. Verify that security events include sufficient context for investigation (URL, origin, error type, timestamp)
10. Test that security events persist across browser restarts if configured for persistent logging
11. All security policy violations are logged with accurate timestamps
12. Logs include sufficient context to identify the origin and nature of security events
13. Security events include: CSP violations, CORS failures, mixed content blocks, certificate errors, sandbox violations
14. Logs distinguish between blocked and allowed actions with clear outcomes
15. Event log format is structured and machine-readable (JSON or similar)
16. Logs can be exported for analysis or forwarding to external systems

Pass Criteria: All tested security events are captured in logs with complete context (timestamp, origin, event type, outcome) AND logs are exportable in a structured format

Fail Criteria: Any security event fails to be logged OR logs lack critical context (timestamp, origin, or outcome) OR logs are not exportable

Evidence: Screenshots of security event logs showing various event types, exported log files in JSON/structured format, video recordings of security events being triggered and logged, comparison matrices showing event coverage

References:

- W3C Reporting API: https://www.w3.org/TR/reporting-1/
- OWASP Logging Cheat Sheet: https://cheatsheetseries.owasp.org/cheatsheets/Logging_Cheat_Sheet.html
- NIST SP 800-92 Guide to Computer Security Log Management: https://csrc.nist.gov/publications/detail/sp/800-92/final

Assessment: LOG-REQ-2 (Certificate error logging)

Reference: LOG-REQ-2 - Browser shall log all certificate validation failures with detailed error information

Given: A conformant browser with LOG-1 or higher capability

Task: Verify that certificate error logging enables detection of man-in-the-middle attacks and certificate misconfigurations, providing detailed information for security investigations. Comprehensive certificate

logs with chain details, error types, and revocation status help identify patterns of TLS interception, rogue certificates, or systematic validation failures that could indicate ongoing attacks against users or infrastructure.

Verification:

1. Set up a test web server with various certificate issues (expired, self-signed, wrong hostname, revoked)
2. Navigate to https://expired.badssl.com/ and verify the certificate error is logged
3. Navigate to https://wrong.host.badssl.com/ and verify hostname mismatch is logged
4. Navigate to https://self-signed.badssl.com/ and verify self-signed certificate is logged
5. Navigate to https://revoked.badssl.com/ and verify revocation status is logged
6. Review the certificate error logs to verify they include: certificate chain, error type, validation date, origin
7. Test certificate pinning failures by creating a pinning policy and violating it
8. Verify that certificate transparency failures are logged when CT enforcement is enabled
9. Test that HSTS violations involving certificates are properly logged
10. All certificate validation failures are logged with specific error codes
11. Logs include certificate subject, issuer, validity period, and error reason
12. Certificate chain information is captured in logs
13. Revocation check results (OCSP/CRL) are included
14. Certificate pinning violations are logged separately
15. Logs distinguish between hard failures (blocked) and soft failures (warnings)

Pass Criteria: All certificate validation failures are logged with complete certificate details AND error reasons are specific and actionable

Fail Criteria: Any certificate error is not logged OR logs lack certificate details OR error reasons are generic/unhelpful

Evidence: Certificate error log entries showing various failure types, screenshots of BadSSL.com test results with corresponding logs, certificate chain dumps from logs, network traces showing certificate validation process

References:

- RFC 5280 X.509 Certificate Validation: https://www.rfc-editor.org/rfc/rfc5280
- Certificate Transparency RFC 6962: https://www.rfc-editor.org/rfc/rfc6962
- Chrome Certificate Error Logging: https://www.chromium.org/Home/chromium-security/certificate-transparency/
- OWASP Transport Layer Protection Cheat Sheet: https://cheatsheetseries.owasp.org/cheatsheets/Transport_Layer_Prot

Assessment: LOG-REQ-3 (Extension security events)

Reference: LOG-REQ-3 - Browser shall log extension-related security events including installation, permission changes, and security violations

Given: A conformant browser with EXT-1 or higher capability and LOG-1 or higher capability

Task: Verify that extension security event logging enables detection of malicious extensions that abuse permissions, access sensitive APIs without authorization, or attempt to bypass security policies. Detailed extension logs with lifecycle events, permission changes, and security violations help identify compromised or rogue extensions that could exfiltrate data, inject malicious scripts, or escalate privileges beyond their declared capabilities.

Verification:

1. Install a test extension and verify the installation event is logged with extension ID, name, and permissions
2. Modify extension permissions and verify permission changes are logged
3. Create a test extension that attempts to access APIs without proper permissions
4. Trigger extension content script injection and verify it's logged

5. Test extension network requests to verify they are logged separately from normal browsing
6. Uninstall the extension and verify the removal event is logged
7. Test developer mode extension loading and verify it's flagged in logs
8. Simulate an extension attempting to bypass CSP and verify the violation is logged
9. Test extension update events and verify version changes are logged
10. Extension lifecycle events (install, update, uninstall) are logged with metadata
11. Permission requests and grants are logged with timestamp and user action
12. Extension security violations are logged separately from web page violations
13. Extension API access attempts are logged with success/failure status
14. Developer mode extensions are clearly marked in logs
15. Extension-injected content is distinguishable in logs from page content

Pass Criteria: All extension lifecycle and security events are logged with complete metadata AND extension actions are distinguishable from page actions

Fail Criteria: Any extension security event is not logged OR extension actions cannot be distinguished from page actions OR permission changes are not logged

Evidence: Extension event logs showing lifecycle events, permission change logs, screenshots of extension security violations with corresponding log entries, comparison of extension vs. page event logs

References:

- Chrome Extension Security Architecture: https://developer.chrome.com/docs/extensions/mv3/security/
- WebExtensions API Security: https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/API
- Extension Permission Model: https://www.chromium.org/Home/chromium-security/extension-content-script-fetches/
- OWASP Browser Extension Security: https://owasp.org/www-community/vulnerabilities/Unsafe_Mobile_Code

Assessment: LOG-REQ-4 (CSP violation reporting)

Reference: LOG-REQ-4 - Browser shall implement Content Security Policy violation reporting and logging

Given: A conformant browser with LOG-1 or higher capability

Task: Verify that CSP violation reporting enables detection of XSS attempts and policy bypasses, providing actionable intelligence about attacks targeting web applications. Automated reporting with complete violation context allows security teams to identify attack vectors, adjust CSP policies, and detect systematic attempts to inject malicious scripts or load unauthorized resources that could compromise user data.

Verification:

1. Create a test page with a strict CSP policy: `Content-Security-Policy: default-src 'self'; report-uri /csp-report`
2. Attempt to load external scripts that violate the CSP policy
3. Verify CSP violation reports are sent to the report-uri endpoint
4. Check browser console for CSP violation messages
5. Test inline script violations and verify they are reported
6. Test eval() violations and verify they are blocked and reported
7. Configure CSP with report-to directive and Report-To header for modern reporting
8. Test CSP reporting with report-only mode using Content-Security-Policy-Report-Only header
9. Verify that violation reports include: violated-directive, blocked-uri, source-file, line-number
10. All CSP violations trigger console warnings in developer tools
11. Violation reports are sent to configured report-uri endpoints
12. Reports include complete context: violated directive, blocked resource, source location
13. Report-only mode generates reports without blocking resources
14. Modern Reporting API (report-to) is supported for CSP reporting
15. Reports are generated in standard JSON format per W3C specification

Pass Criteria: All CSP violations are reported to configured endpoints AND console warnings are displayed AND reports contain complete violation context

Fail Criteria: Any CSP violation is not reported OR reports lack critical information OR report-uri/report-to mechanisms don't function

Evidence: CSP violation reports in JSON format, server logs showing received reports, browser console screenshots with CSP warnings, network traces showing report transmission, comparison of report-only vs. enforce mode

References:

- Content Security Policy Level 3: https://www.w3.org/TR/CSP3/
- CSP Violation Reports: https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP#violation_reports
- W3C Reporting API: https://www.w3.org/TR/reporting-1/
- CSP Evaluator Tool: https://csp-evaluator.withgoogle.com/

Assessment: LOG-REQ-5 (Network Error Logging - NEL)

Reference: LOG-REQ-5 - Browser shall support Network Error Logging (NEL) for monitoring network failures

Given: A conformant browser with LOG-1 or higher capability

Task: Verify that Network Error Logging enables detection of network-level attacks and infrastructure issues, monitoring connection failures that could indicate DNS hijacking, TLS interception, or targeted denial of service. NEL provides visibility into network failures that occur before HTTP layer, helping identify systematic attacks, CDN failures, or connectivity issues affecting user security and availability.

Verification:

1. Set up a test server that sends NEL policy header: `NEL: {"report_to":"network-errors","max_age":86400,"succes`
2. Configure Report-To header: `Report-To: {"group":"network-errors","max_age":86400,"endpoints":[{"url":"h`
3. Trigger DNS resolution failures by attempting to navigate to non-existent domains
4. Trigger connection timeout errors by connecting to a filtered port
5. Trigger TLS handshake failures by using misconfigured certificates
6. Trigger HTTP protocol errors by sending malformed responses
7. Verify that NEL reports are generated and sent to the configured endpoint
8. Check that success sampling works correctly (success_fraction parameter)
9. Verify NEL reports include: type, url, server-ip, protocol, status-code, elapsed-time, phase
10. NEL policy is correctly parsed from HTTP headers
11. Network failures trigger NEL report generation
12. Reports are sent to configured endpoints asynchronously
13. Sampling fractions are respected for success/failure events
14. NEL reports include detailed failure context (phase, type, status)
15. Reports distinguish between DNS, connection, TLS, and HTTP errors
16. NEL reports are batched and sent efficiently

Pass Criteria: NEL policy is respected AND all configured failure types generate reports AND reports include complete network error context

Fail Criteria: NEL policy is ignored OR network failures don't generate reports OR reports lack critical failure information

Evidence: NEL policy headers from server, collected NEL reports in JSON format, server logs showing received reports, network traces demonstrating various failure types, NEL report timing analysis

References:

- Network Error Logging Specification: https://www.w3.org/TR/network-error-logging/
- Reporting API Specification: https://www.w3.org/TR/reporting-1/

- NEL Deployment Guide: https://developers.google.com/web/updates/2018/09/reportingapi
- MDN Network Error Logging: https://developer.mozilla.org/en-US/docs/Web/HTTP/Network_Error_Logging

Assessment: LOG-REQ-6 (Crash reporting)

Reference: LOG-REQ-6 - Browser shall implement privacy-preserving crash reporting with user consent

Given: A conformant browser with LOG-1 or higher capability

Task: Verify that privacy-preserving crash reporting enables browser vendors to identify and fix security vulnerabilities without exposing user browsing history or personal information. User consent requirements ensure transparency while crash report anonymization prevents data leakage, balancing the need for diagnostic information to address exploitable bugs with user privacy rights and regulatory compliance.

Verification:

1. Review browser settings for crash reporting consent options
2. Enable crash reporting and verify user consent is obtained before activation
3. Force a controlled browser crash using debugging tools or crash test pages
4. Verify a crash report is generated locally
5. Check that crash reports are anonymized and don't contain browsing history or personal data
6. Verify crash reports include: crash signature, stack trace (symbolicated), browser version, OS version
7. Test that crash reports are only uploaded after user consent
8. Verify users can view and delete crash reports before submission
9. Test opt-out functionality and verify no reports are sent when opted out
10. Verify renderer process crashes are reported separately from browser process crashes
11. User consent is required before crash reporting is enabled
12. Crash reports are generated for browser and renderer crashes
13. Reports include technical diagnostics (stack traces, crash signatures) but no personal data
14. Users can review crash reports before submission
15. Crash reporting can be disabled and re-enabled in settings
16. Crash reports are transmitted securely (HTTPS) to vendor endpoints
17. Local crash report storage has size/age limits to prevent disk exhaustion

Pass Criteria: User consent is obtained before crash reporting AND crash reports exclude personal data AND users can review/delete reports

Fail Criteria: Crash reports are sent without consent OR reports contain personal/browsing data OR users cannot control crash reporting

Evidence: Crash report consent dialogs, sanitized crash reports showing included data, settings screenshots showing crash reporting controls, privacy policy documentation, crash report upload network traces

References:

- Breakpad Crash Reporting: https://chromium.googlesource.com/breakpad/breakpad/
- Firefox Crash Reporter: https://support.mozilla.org/en-US/kb/mozilla-crash-reporter
- GDPR Crash Reporting Compliance: https://gdpr.eu/what-is-gdpr/

Assessment: LOG-REQ-7 (Log data minimization)

Reference: LOG-REQ-7 - Browser shall minimize data collection in logs, collecting only information necessary for security purposes

Given: A conformant browser with LOG-1 or higher capability

Task: Verify that log data minimization prevents excessive collection of personal information, balancing security monitoring needs with privacy rights and regulatory compliance. Collecting only necessary security-relevant data with automatic retention limits reduces the risk of data breaches exposing user browsing history,

credentials, or sensitive personal information while still enabling effective threat detection and incident response.

Verification:

1. Review all log categories to identify what data is collected (security events, network, performance, etc.)
2. Examine security event logs to verify they don't contain unnecessary personal information
3. Check that URLs in logs are sanitized (query parameters removed or hashed)
4. Verify that user credentials are never logged, even in error conditions
5. Test that cookies and authentication tokens are redacted from network logs
6. Review crash reports to ensure they exclude browsing history and form data
7. Verify that IP addresses in logs are anonymized (last octet removed) or hashed
8. Test that logs have automatic retention limits (time-based and size-based)
9. Verify that sensitive form fields (passwords, credit cards) are never logged
10. Check that telemetry aggregates data rather than logging individual user actions
11. Logs contain only security-relevant events, not general browsing activity
12. Personal identifiable information (PII) is redacted or hashed in logs
13. URL parameters that may contain session tokens are removed
14. Credentials, cookies, and authentication headers are never logged
15. IP addresses are anonymized or removed from logs
16. Logs automatically expire based on retention policies
17. Form input data is excluded from all logs
18. Aggregated metrics replace individual event logging where possible

Pass Criteria: All logs demonstrate data minimization (no unnecessary PII) AND sensitive data is consistently redacted AND retention limits are enforced

Fail Criteria: Logs contain unnecessary PII OR credentials/tokens appear in logs OR no retention limits exist

Evidence: Log samples showing redaction of sensitive data, privacy analysis of log contents, retention policy documentation, comparison of logged vs. available data showing minimization, code review of logging implementations

References:

- OWASP Logging Cheat Sheet - Data Sanitization: https://cheatsheetseries.owasp.org/cheatsheets/Logging_Cheat_Sheet
- NIST Privacy Framework: https://www.nist.gov/privacy-framework
- W3C Privacy Principles: https://www.w3.org/TR/privacy-principles/

Assessment: LOG-REQ-8 (Log anonymization)

Reference: LOG-REQ-8 - Browser shall implement anonymization techniques for logs that require user-related data

Given: A conformant browser with LOG-1 or higher capability

Task: Verify that log anonymization prevents re-identification of users from telemetry data, protecting user privacy while maintaining the ability to detect security incidents and diagnose technical issues. Proper anonymization defends against correlation attacks where adversaries combine multiple log entries to de-anonymize users, as well as against data breaches where stolen logs could reveal sensitive user information.

Verification:

1. Review telemetry logs to identify user-related fields (user ID, device ID, session ID)
2. Verify that user identifiers are hashed with secure cryptographic hash functions
3. Check that hash salts are rotated periodically to prevent correlation
4. Test that IP addresses are anonymized using techniques like IP truncation or hashing
5. Verify that timestamps are rounded to reduce precision (hour or day level) where appropriate
6. Test that geographic data is generalized (city level rather than GPS coordinates)

7. Review aggregation techniques to ensure k-anonymity (minimum group size) is maintained
8. Verify differential privacy techniques are applied to statistical queries on logs
9. Test that user fingerprints cannot be reconstructed from anonymized logs
10. Check that pseudonymous identifiers change across different log contexts
11. User identifiers are consistently hashed with strong cryptographic algorithms
12. Hash salts are documented and rotated on a defined schedule
13. IP addresses are truncated or hashed before storage
14. Timestamps are appropriately rounded to reduce granularity
15. Geographic data is generalized to prevent precise location tracking
16. Aggregated data maintains k-anonymity with k >= 5
17. Differential privacy noise is added to prevent individual identification
18. Cross-log correlation attacks are prevented through identifier rotation

Pass Criteria: All user-related data is anonymized using documented techniques AND re-identification is demonstrably prevented AND k-anonymity is maintained

Fail Criteria: User data is logged in plaintext OR anonymization is reversible OR re-identification is possible through correlation

Evidence: Anonymized log samples with hash values, salt rotation policy documentation, privacy analysis showing k-anonymity, differential privacy parameters, re-identification attack test results (negative results expected)

References:

- Differential Privacy: https://www.microsoft.com/en-us/research/publication/differential-privacy/
- K-Anonymity Model: https://epic.org/privacy/reidentification/Sweeney_Article.pdf
- NIST De-Identification Guidelines: https://nvlpubs.nist.gov/nistpubs/ir/2015/NIST.IR.8053.pdf
- Google's Privacy-Preserving Techniques: https://policies.google.com/technologies/anonymization

Assessment: LOG-REQ-9 (User consent for telemetry)

Reference: LOG-REQ-9 - Browser shall obtain explicit user consent before collecting and transmitting telemetry data

Given: A conformant browser with LOG-1 or higher capability

Task: Verify that explicit user consent for telemetry protects user privacy rights and complies with data protection regulations including GDPR and CPRA. Without proper consent mechanisms, browsers may violate privacy laws by collecting personal data without permission, and users are deprived of control over their information. Consent must be freely given, specific, informed, and revocable to meet legal and ethical standards.

Verification:

1. Perform a fresh installation of the browser and observe the first-run experience
2. Verify that a clear consent prompt is displayed for telemetry collection
3. Check that the consent prompt explains what data is collected and why
4. Verify users can decline telemetry without affecting core browser functionality
5. Test that declining telemetry prevents all non-essential data collection
6. Navigate to browser settings and verify telemetry preferences are accessible
7. Verify users can change their consent choice at any time in settings
8. Test that telemetry settings are granular (separate controls for crash reports, usage stats, etc.)
9. Verify that consent choices persist across browser sessions and updates
10. Check that consent is re-requested when telemetry data types or purposes change significantly
11. First-run consent prompt is clear, prominent, and explains data collection
12. Users can freely choose to accept or decline without dark patterns
13. Declining telemetry doesn't degrade core browser functionality
14. Telemetry settings are easily accessible in preferences/settings

15. Consent choices are persistent and respected across updates
16. Granular controls allow users to consent to specific telemetry types
17. Changes to data collection practices trigger new consent requests
18. Consent records are maintained for compliance auditing

Pass Criteria: Explicit consent is obtained before telemetry collection AND users can easily manage consent preferences AND browser functions normally when telemetry is declined

Fail Criteria: Telemetry starts without consent OR consent cannot be withdrawn OR declining breaks browser functionality OR dark patterns are used

Evidence: Screenshots of consent prompts and settings UI, network traces showing no telemetry when declined, functional testing with telemetry disabled, consent flow video recordings, privacy policy documentation

References:

- GDPR Consent Requirements: https://gdpr.eu/gdpr-consent-requirements/
- ePrivacy Directive: https://eur-lex.europa.eu/legal-content/EN/ALL/?uri=CELEX:32002L0058
- W3C Privacy Principles - User Control: https://www.w3.org/TR/privacy-principles/#user-control

Assessment: LOG-REQ-10 (Secure log transmission)

Reference: LOG-REQ-10 - Browser shall transmit logs securely using encrypted channels with certificate validation

Given: A conformant browser with LOG-1 or higher capability

Task: Verify that secure log transmission prevents interception or modification of telemetry and crash reports in transit, protecting sensitive diagnostic data from network attackers. Without encrypted transmission and certificate validation, adversaries can eavesdrop on log data to gain insights into user behavior, browser vulnerabilities, or enterprise configurations, or perform man-in-the-middle attacks to inject false telemetry data.

Verification:

1. Enable telemetry and crash reporting in browser settings
2. Trigger events that generate log transmissions (crash, CSP violation, NEL error)
3. Use network monitoring tools (Wireshark, mitmproxy) to capture log transmission traffic
4. Verify all log transmissions use HTTPS (TLS 1.2 or higher)
5. Verify certificate validation is performed for log collection endpoints
6. Test that log transmission fails if the server certificate is invalid
7. Check that certificate pinning is used for log collection endpoints if available
8. Verify log data is not transmitted over insecure protocols (HTTP, FTP, unencrypted sockets)
9. Test that log transmission includes retry logic for temporary network failures
10. Verify log transmission is batched and rate-limited to prevent network abuse
11. All log transmissions use TLS 1.2 or higher encryption
12. Certificate validation is enforced for log collection servers
13. Invalid or expired certificates prevent log transmission
14. Certificate pinning is applied to log endpoints where supported
15. No log data is ever transmitted in plaintext
16. Connection failures trigger retry with exponential backoff
17. Log batching reduces network overhead and improves privacy
18. Rate limiting prevents log transmission from consuming excessive bandwidth

Pass Criteria: All log transmissions use TLS 1.2+ with certificate validation AND transmission fails for invalid certificates AND no plaintext transmission occurs

Fail Criteria: Any logs transmitted over plaintext protocols OR certificate validation is not enforced OR invalid certificates are accepted

Evidence: Network packet captures showing TLS-encrypted log traffic, certificate validation test results, failed transmission logs for invalid certificates, retry mechanism testing, bandwidth usage analysis

References:

- TLS 1.3 Specification RFC 8446: https://www.rfc-editor.org/rfc/rfc8446
- Certificate Pinning: https://owasp.org/www-community/controls/Certificate_and_Public_Key_Pinning
- OWASP Transport Layer Protection: https://cheatsheetseries.owasp.org/cheatsheets/Transport_Layer_Protection_Chea
- Mozilla TLS Configuration: https://wiki.mozilla.org/Security/Server_Side_TLS

Assessment: LOG-REQ-11 (Log integrity protection)

Reference: LOG-REQ-11 - Browser shall implement integrity protection for locally stored logs to prevent tampering

Given: A conformant browser with LOG-1 or higher capability

Task: Verify that log integrity protection prevents attackers from covering their tracks after compromising a system by tampering with security logs. Without integrity protection, malicious actors who gain local access can modify or delete log entries to hide evidence of their activities, making incident response and forensic investigation impossible. Cryptographic integrity mechanisms ensure that any tampering is detected.

Verification:

1. Enable local security logging in browser configuration or enterprise policy
2. Generate security events that create local log entries
3. Locate the local log storage files in the browser's data directory
4. Verify that log files include cryptographic signatures or message authentication codes (MACs)
5. Attempt to modify a log entry manually and verify the tampering is detected
6. Check that log files use append-only mechanisms where supported by the OS
7. Verify log rotation maintains integrity chains between rotated files
8. Test that the browser detects and alerts on corrupted or tampered logs
9. Verify enterprise-mode logs support additional integrity mechanisms (digital signatures)
10. Test that log integrity is checked before logs are exported or transmitted
11. Local logs include integrity protection mechanisms (signatures, MACs, or hashes)
12. Tampering with log contents is detected by the browser
13. Log files use OS-level protection where available (append-only, immutable flags)
14. Log rotation preserves integrity chains across files
15. Corrupted logs trigger alerts or warnings
16. Enterprise deployments support strong integrity mechanisms (digital signatures)
17. Integrity checks occur before log export or transmission
18. Integrity metadata is stored separately from log content for additional protection

Pass Criteria: Logs include integrity protection (signatures/MACs/hashes) AND tampering is detected AND alerts are generated for integrity violations

Fail Criteria: Logs lack integrity protection OR tampering is not detected OR no alerts for integrity violations

Evidence: Log file analysis showing integrity mechanisms, tampering test results demonstrating detection, alert screenshots for corrupted logs, documentation of integrity algorithms used, enterprise policy configurations

References:

- NIST FIPS 180-4 Secure Hash Standard: https://csrc.nist.gov/publications/detail/fips/180/4/final
- Log Integrity and Non-Repudiation: https://www.nist.gov/publications/guide-computer-security-log-management
- Merkle Tree for Log Integrity: https://en.wikipedia.org/wiki/Merkle_tree
- OWASP Logging Guide - Integrity: https://cheatsheetseries.owasp.org/cheatsheets/Logging_Cheat_Sheet.html

Assessment: LOG-REQ-12 (Log retention policies)

Reference: LOG-REQ-12 - Browser shall implement and enforce log retention policies that balance security needs with privacy requirements

Given: A conformant browser with LOG-1 or higher capability

Task: Verify that log retention policies balance security investigation needs against privacy rights by limiting how long personal data is stored. Excessive retention violates privacy regulations like GDPR which mandate data minimization, while insufficient retention hampers security incident investigation. Proper retention policies ensure logs are available for legitimate security purposes without becoming an indefinite privacy liability.

Verification:

1. Review browser documentation for default log retention policies
2. Examine local log storage to identify retention periods for different log types
3. Verify that security logs have appropriate retention (30-90 days typical)
4. Test that crash dumps are automatically deleted after retention period
5. Verify that telemetry data has shorter retention than security logs
6. Check that enterprise mode supports configurable retention policies
7. Test that log rotation occurs based on size and time criteria
8. Verify that users can manually clear logs before retention period expires
9. Test that retention policies are enforced even when browser is closed
10. Verify that regulatory compliance requirements (GDPR, etc.) are considered in retention
11. Default retention periods are documented for each log type
12. Security logs are retained longer than general telemetry (30-90 days vs. 7-30 days)
13. Automatic deletion occurs when retention period expires
14. Log rotation prevents disk exhaustion (size-based limits)
15. Enterprise policies allow customization of retention periods
16. Users can manually clear logs through settings or clear browsing data
17. Retention enforcement continues even when browser is not running
18. GDPR/privacy compliance is demonstrated through retention limits

Pass Criteria: Documented retention policies exist for all log types AND automatic deletion enforces retention AND policies comply with privacy regulations

Fail Criteria: No retention policies OR logs grow unbounded OR retention periods violate privacy regulations (too long)

Evidence: Retention policy documentation, log file age analysis, storage usage over time, automatic deletion test results, enterprise policy configuration examples, GDPR compliance analysis

References:

- NIST SP 800-92 Log Retention: https://csrc.nist.gov/publications/detail/sp/800-92/final
- ISO 27001 Log Management: https://www.iso.org/standard/54534.html
- PCI DSS Logging Requirements: https://www.pcisecuritystandards.org/

Assessment: LOG-REQ-13 (Security dashboard)

Reference: LOG-REQ-13 - Browser shall provide a security dashboard that presents security events and status to users

Given: A conformant browser with LOG-1 or higher capability

Task: Verify that a security dashboard empowers users to understand their security posture and respond to threats by providing clear visibility into security events and protection status. Without a dashboard, users remain unaware of ongoing attacks, misconfigurations, or compromised security settings, leaving them vulnerable. Transparent security status information enables informed security decisions and builds user trust.

Verification:

1. Access the browser's security dashboard (e.g., chrome://settings/security, about:preferences#privacy)
2. Verify the dashboard displays current security status (safe/warning/critical)
3. Check that recent security events are listed with timestamps and descriptions
4. Trigger a security event (certificate error, malware warning, etc.) and verify it appears in the dashboard
5. Test that the dashboard categorizes events by severity (critical, warning, info)
6. Verify the dashboard shows security settings status (HTTPS-only, Safe Browsing, etc.)
7. Test that clicking on security events provides detailed information and remediation steps
8. Verify the dashboard updates in real-time or near-real-time when security events occur
9. Check that the dashboard is accessible from the main browser settings menu
10. Test that the dashboard supports filtering and searching of security events
11. Security dashboard is easily accessible from main settings
12. Current security status is clearly displayed with visual indicators
13. Recent security events are listed chronologically with timestamps
14. Events are categorized by severity level with appropriate visual coding
15. Each event includes actionable information and remediation guidance
16. Dashboard updates when new security events occur
17. Users can filter events by type, severity, or time period
18. Dashboard shows overall security posture (enabled protections)
19. Interface is user-friendly and avoids excessive technical jargon

Pass Criteria: Security dashboard is accessible AND displays recent security events with severity AND provides actionable remediation guidance

Fail Criteria: No security dashboard exists OR dashboard doesn't show events OR events lack context/remediation info

Evidence: Screenshots of security dashboard showing various states, video walkthrough of dashboard features, security event listings, user interface usability assessment, comparison with security best practices

References:

- Chrome Security Settings: https://support.google.com/chrome/answer/114836
- NIST Cybersecurity Framework - Detect: https://www.nist.gov/cyberframework
- User-Centered Security Design: https://www.usenix.org/conference/soups2019

Assessment: LOG-REQ-14 (Incident detection)

Reference: LOG-REQ-14 - Browser shall implement automated incident detection based on security event patterns

Given: A conformant browser with LOG-2 or higher capability

Task: Verify that automated incident detection identifies active attacks by correlating security event patterns that indicate malicious activity, enabling rapid response before significant damage occurs. Manual log review alone cannot detect sophisticated attacks that span multiple events or occur at scale. Automated detection using heuristics and pattern matching provides early warning of credential stuffing, reconnaissance, malware distribution, and other attack campaigns.

Verification:

1. Configure the browser for enhanced security monitoring (enterprise mode if required)
2. Access browser's internal incident detection interfaces or logs
3. Simulate a credential stuffing attack by repeatedly entering wrong passwords
4. Verify that repeated authentication failures trigger an incident alert
5. Simulate a port scanning attack by navigating to many sequential ports on localhost
6. Verify that unusual network activity patterns are detected
7. Trigger multiple CSP violations in rapid succession and verify pattern detection

8. Test that suspicious extension behavior (excessive API calls) triggers alerts
9. Verify that malware download attempts are detected and blocked
10. Test that correlation of multiple minor events escalates to incident status
11. Automated detection identifies suspicious patterns (credential stuffing, scanning, etc.)
12. Incident detection uses heuristics and machine learning where appropriate
13. Multiple low-severity events can aggregate to trigger incident alerts
14. False positive rates are managed through tuning and whitelisting
15. Incidents are logged with detailed context for investigation
16. Users or administrators receive notifications for detected incidents
17. Incident severity is calculated based on event type and frequency
18. Detection rules are updated regularly to address new attack patterns

Pass Criteria: Automated detection identifies at least 3 attack patterns (credential stuffing, scanning, malware) AND incidents are logged with context AND alerts are generated

Fail Criteria: No automated detection occurs OR fewer than 3 attack patterns detected OR no alerts generated

Evidence: Incident detection logs showing various attack patterns, alert notifications, false positive analysis, detection rule documentation, test results for simulated attacks, tuning methodology

References:

- MITRE ATT&CK Framework: https://attack.mitre.org/
- NIST Incident Response Guide SP 800-61: https://csrc.nist.gov/publications/detail/sp/800-61/rev-2/final
- Browser Security Indicators: https://www.w3.org/TR/security-privacy-questionnaire/

Assessment: LOG-REQ-15 (Audit trail completeness)

Reference: LOG-REQ-15 - Browser shall maintain complete audit trails for security-relevant administrative actions

Given: A conformant browser with LOG-2 or higher capability (enterprise mode)

Task: Verify that complete audit trails for administrative actions enable accountability and investigation of security policy changes, preventing unauthorized or malicious modifications from going unnoticed. Without comprehensive audit logging, insider threats or compromised administrator accounts can weaken security settings without detection. Complete audit trails create accountability and support forensic investigations when security incidents occur.

Verification:

1. Enable enterprise policy management for the browser
2. Change a security-critical setting (e.g., disable Safe Browsing, modify HTTPS-only mode)
3. Verify the change is logged with: timestamp, user/admin identity, setting name, old value, new value
4. Install or remove a browser extension and verify the action is logged
5. Modify certificate trust settings and verify the change is logged
6. Change cookie or site permission policies and verify logging
7. Modify content security policies and verify logging
8. Test that policy enforcement (GPO, MDM) actions are logged
9. Verify that failed administrative actions (insufficient permissions) are also logged
10. Export the audit log and verify it includes all tested actions with complete metadata
11. All security-relevant configuration changes are logged
12. Logs include: timestamp, user/admin identity, action type, object affected, before/after values
13. Both successful and failed administrative actions are logged
14. Extension lifecycle events (install/update/remove) are included
15. Certificate and trust anchor modifications are logged
16. Policy enforcement events are captured

17. Audit logs are tamper-evident and include integrity protection
18. Logs are exportable in standard formats (JSON, CSV, syslog)

Pass Criteria: All security-relevant administrative actions are logged with complete metadata AND failed actions are logged AND logs are exportable

Fail Criteria: Any security configuration change is not logged OR logs lack critical metadata OR logs are not exportable

Evidence: Audit log exports showing various administrative actions, log completeness analysis, integrity verification results, enterprise policy documentation, screenshots of logged events

References:

- NIST SP 800-53 Audit and Accountability: https://csrc.nist.gov/publications/detail/sp/800-53/rev-5/final
- ISO 27001 Audit Logging: https://www.iso.org/standard/54534.html
- CIS Controls - Audit Log Management: https://www.cisecurity.org/controls/

Assessment: LOG-REQ-16 (Real-time security alerts)

Reference: LOG-REQ-16 - Browser shall provide real-time security alerts for critical security events

Given: A conformant browser with LOG-1 or higher capability

Task: Verify that real-time security alerts prevent users from inadvertently exposing themselves to immediate threats by providing prominent warnings before dangerous actions occur. Delayed or passive alerts allow users to proceed with risky actions like visiting malware sites or ignoring certificate errors. Immediate, blocking alerts with clear threat information enable users to make informed security decisions and avoid common attack vectors.

Verification:

1. Navigate to a site with a revoked certificate and verify immediate alert is displayed
2. Navigate to a known malware site (using Safe Browsing test URLs) and verify blocking alert
3. Trigger a password breach detection (if supported) and verify immediate notification
4. Install a malicious extension (test extension) and verify warning is displayed
5. Attempt to download a known malicious file and verify real-time blocking alert
6. Test that alerts are displayed before allowing dangerous actions (not after)
7. Verify alerts are prominent, modal, and cannot be easily dismissed accidentally
8. Test that alerts provide clear information about the threat and recommended actions
9. Verify enterprise mode supports additional real-time alerting (admin notifications)
10. Test that alert severity levels affect presentation (critical vs. warning vs. info)
11. Critical security events trigger immediate, modal alerts
12. Alerts are displayed before dangerous actions are allowed
13. Alert content is clear, specific, and actionable
14. Users should explicitly acknowledge alerts to proceed
15. Alerts distinguish between critical threats (malware) and warnings (certificate issues)
16. Visual design makes alerts prominent and attention-getting
17. Enterprise mode supports admin notifications for critical events
18. Alert fatigue is avoided through appropriate severity calibration
19. Alerts include context and remediation guidance

Pass Criteria: Critical security events trigger immediate modal alerts AND alerts provide clear threat information AND users should acknowledge before proceeding

Fail Criteria: No real-time alerts for critical events OR alerts are easily dismissed OR alerts lack actionable information

Evidence: Screenshots of various security alerts, video recordings of alert timing, user studies on alert comprehensibility, enterprise admin notification examples, alert frequency analysis

References:

- NIST Usable Security: https://www.nist.gov/programs-projects/usable-cybersecurity
- Google Safe Browsing: https://safebrowsing.google.com/
- Security Warning Design: https://www.usenix.org/conference/soups2019
- Alert Fatigue Research: https://www.ndss-symposium.org/ndss-paper/auto-draft-188/

Assessment: LOG-REQ-17 (Forensic log export)

Reference: LOG-REQ-17 - Browser shall support forensic-quality log export for security investigations

Given: A conformant browser with LOG-2 or higher capability

Task: Verify that forensic log export enables detailed security investigations by providing complete, integrity-protected logs in standard formats that can be analyzed with industry-standard tools. Without proper export capabilities, security teams cannot perform comprehensive incident response or forensic analysis, limiting their ability to understand attack vectors, determine scope of compromise, or provide evidence for legal proceedings.

Verification:

1. Generate various security events across multiple sessions (certificate errors, CSP violations, etc.)
2. Access browser log export functionality (may require developer or enterprise mode)
3. Export security logs in multiple formats (JSON, CSV, syslog)
4. Verify exported logs include all events from the specified time period
5. Check that exported logs maintain chronological ordering
6. Verify exported logs include complete metadata (timestamps in ISO 8601 format, event IDs, etc.)
7. Test that log export includes integrity information (signatures or hashes)
8. Verify sensitive information is appropriately redacted in exported logs
9. Test that exported logs are in formats compatible with SIEM tools (Splunk, ELK, etc.)
10. Verify that export process itself is logged for audit purposes
11. Log export is available through settings or developer tools
12. Multiple export formats are supported (JSON, CSV, syslog, CEF)
13. Exported logs are complete and chronologically ordered
14. Timestamps use standardized formats (ISO 8601, Unix epoch)
15. Event identifiers are included for correlation
16. Integrity information accompanies exports (checksums or signatures)
17. Sensitive data is redacted appropriately
18. Exported formats are compatible with common SIEM platforms
19. Export actions are logged for accountability

Pass Criteria: Log export functionality exists AND multiple standard formats supported AND exported logs include complete metadata with integrity protection

Fail Criteria: No export functionality OR only proprietary formats OR exported logs lack metadata OR no integrity protection

Evidence: Exported log files in various formats, SIEM import test results, log completeness verification, integrity validation results, format specification documentation, screenshots of export interface

References:

- Common Event Format (CEF): https://www.microfocus.com/documentation/arcsight/arcsight-smartconnectors-8.3/cef-implementation-standard/
- Syslog Protocol RFC 5424: https://www.rfc-editor.org/rfc/rfc5424
- ELK Stack Log Analysis: https://www.elastic.co/what-is/elk-stack
- NIST SP 800-92 Log Management: https://csrc.nist.gov/publications/detail/sp/800-92/final

Assessment: LOG-REQ-18 (Privacy-preserving analytics)

Reference: LOG-REQ-18 - Browser shall use privacy-preserving techniques for analytics and aggregate reporting

Given: A conformant browser with LOG-1 or higher capability

Task: Verify that privacy-preserving analytics techniques enable browsers to gather valuable usage insights and improve security without compromising individual user privacy. Traditional analytics create re-identification risks by collecting detailed individual behavior. Differential privacy, local noise injection, and k-anonymity allow aggregated insights while mathematically guaranteeing that individual users cannot be identified or their specific behaviors revealed.

Verification:

1. Review browser telemetry documentation for privacy-preserving techniques
2. Verify that differential privacy is used for usage statistics aggregation
3. Check that local differential privacy (LDP) adds noise before data leaves the device
4. Test that RAPPOR (Randomized Aggregatable Privacy-Preserving Ordinal Response) or similar is used
5. Verify that aggregated metrics cannot be de-aggregated to identify individuals
6. Test that feature usage statistics use k-anonymity (minimum group size)
7. Verify that privacy budgets limit information disclosure over time
8. Check that federated learning is used where applicable (e.g., next-word prediction)
9. Test that aggregate reporting APIs (Attribution Reporting) use noise injection
10. Verify that privacy parameters (epsilon, delta) are documented and justified
11. Differential privacy is applied to aggregate statistics
12. Local differential privacy adds noise on-device before transmission
13. RAPPOR or equivalent techniques are used for categorical data
14. Privacy budgets limit cumulative information disclosure
15. K-anonymity ensures minimum group sizes (k >= 5)
16. Federated learning keeps training data local
17. Attribution Reporting API uses noise and aggregation
18. Privacy parameters (epsilon, delta, k) are publicly documented
19. Regular privacy audits verify techniques are correctly implemented

Pass Criteria: Differential privacy or equivalent techniques are used AND privacy parameters are documented AND individual users cannot be identified from aggregates

Fail Criteria: No privacy-preserving techniques used OR aggregate data allows individual identification OR privacy parameters undocumented

Evidence: Privacy technique documentation, epsilon/delta parameter specifications, de-identification attack test results (negative), differential privacy implementation code review, aggregate report samples, federated learning architecture diagrams

References:

- Differential Privacy: https://www.microsoft.com/en-us/research/publication/differential-privacy/
- RAPPOR: Randomized Aggregatable Privacy-Preserving Ordinal Response: https://research.google/pubs/pub42852/
- Attribution Reporting API: https://github.com/WICG/attribution-reporting-api
- Federated Learning: https://ai.googleblog.com/2017/04/federated-learning-collaborative.html
- Apple Differential Privacy: https://www.apple.com/privacy/docs/Differential_Privacy_Overview.pdf
- W3C Privacy Principles: https://www.w3.org/TR/privacy-principles/

Assessment: LOG-REQ-19 (Compliance logging)

Reference: LOG-REQ-19 - Browser shall provide logging capabilities to support regulatory compliance requirements (GDPR etc.)

Given: A conformant browser with LOG-2 or higher capability (enterprise mode)

Task: Verify that compliance logging enables organizations to demonstrate adherence to privacy regulations by maintaining comprehensive records of data processing activities, consent, and data subject rights fulfillment. Without proper compliance logging, organizations cannot prove they honor user rights, track data processing lawfulness, or respond to regulatory audits, leading to significant legal and financial penalties under GDPR and similar laws.

Verification:

1. Review browser documentation for compliance logging capabilities
2. Verify that data processing activities are logged (collection, storage, transmission)
3. Test that user consent events are logged with timestamps and scope
4. Verify that data deletion requests are logged and honored
5. Check that data subject access requests can be fulfilled from logs
6. Test that cross-border data transfers are logged with destination regions
7. Verify that third-party data sharing events are logged
8. Test that data breach detection events are logged with required metadata
9. Verify that retention policies align with regulatory requirements
10. Check that logs can demonstrate compliance during audits
11. Data processing activities are comprehensively logged
12. Consent events capture: timestamp, user ID, data types, purposes, duration
13. Data deletion events are logged with completion verification
14. Access request fulfillment is possible from log data
15. Cross-border transfers are logged with legal basis
16. Third-party data sharing is logged with recipient and purpose
17. Breach detection and notification events are logged
18. Retention aligns with GDPR (no longer than necessary) and other regulations
19. Compliance reports can be generated from logs

Pass Criteria: Compliance-relevant activities are logged (consent, deletion, access) AND logs support audit requirements AND retention aligns with regulations

Fail Criteria: Compliance activities not logged OR logs insufficient for audits OR retention violates regulations

Evidence: Compliance log exports, sample audit reports generated from logs, consent event logs, deletion request logs, data processing records, legal basis documentation, retention policy compliance analysis

References:

- GDPR Requirements: https://gdpr.eu/
- ISO 27001 Compliance Auditing: https://www.iso.org/standard/54534.html
- NIST Privacy Framework: https://www.nist.gov/privacy-framework
- ePrivacy Directive: https://eur-lex.europa.eu/legal-content/EN/ALL/?uri=CELEX:32002L0058

Assessment: LOG-REQ-20 (Log access controls)

Reference: LOG-REQ-20 - Browser shall implement access controls to protect logs from unauthorized access or modification

Given: A conformant browser with LOG-2 or higher capability (enterprise mode)

Task: Verify that log access controls protect sensitive security and diagnostic information from unauthorized disclosure or tampering, preserving both user privacy and forensic integrity. Unprotected logs can be read by malware or local attackers to gather intelligence about system configuration, security events, or user activities. Without write protection, attackers can tamper with logs to hide evidence of compromise.

Verification:

1. Review log storage locations and verify they use appropriate OS-level permissions
2. Test that log files are readable only by the browser process and authorized users

3. Verify that unprivileged processes cannot access browser log files
4. Test that log files use OS access control mechanisms (file permissions, ACLs, encryption)
5. Verify that logs stored in user profile directories are protected from other users
6. Test that remote log access (enterprise SIEM integration) requires authentication
7. Verify that log export functionality requires user confirmation or admin privileges
8. Test that log modification is prevented through append-only modes or immutable flags
9. Verify that log access attempts are themselves logged for audit
10. Check that encryption at rest is available for sensitive logs
11. Log files have restrictive OS permissions (user-only or admin-only read)
12. File ACLs prevent unauthorized access on multi-user systems
13. Logs in user profiles are isolated from other user accounts
14. Remote log transmission uses authenticated, encrypted channels
15. Log export requires explicit user action or administrative privileges
16. Log files use append-only or immutable attributes where supported
17. Log access attempts are recorded in audit logs
18. Encryption at rest protects logs on disk
19. Enterprise mode supports centralized access control policies

Pass Criteria: Log files have restrictive permissions AND remote access requires authentication AND log modification is prevented AND access is audited

Fail Criteria: Logs are world-readable OR no access controls on remote access OR logs can be modified OR access not audited

Evidence: File permission analysis (ls -l, icacls), ACL configurations, multi-user access testing, remote access authentication tests, append-only flag verification, access audit log samples, encryption at rest verification

References:

- NIST SP 800-92 Log Protection: https://csrc.nist.gov/publications/detail/sp/800-92/final
- Linux File Permissions Best Practices: https://www.redhat.com/sysadmin/linux-file-permissions-explained
- Windows ACL Security: https://docs.microsoft.com/en-us/windows/security/identity-protection/access-control/access-control
- macOS File System Security: https://developer.apple.com/documentation/security
- OWASP Logging Guide - Protection: https://cheatsheetseries.owasp.org/cheatsheets/Logging_Cheat_Sheet.html
- ISO 27001 Log Access Controls: https://www.iso.org/standard/54534.html

6.5 Update Mechanism Security Assessments

This section covers assessment procedures for requirements UPD-REQ-1 through UPD-REQ-23, addressing secure update delivery, signature verification, rollback protection, update channels, and update transparency.

Assessment: UPD-REQ-1 (Automatic update mechanism)

Reference: UPD-REQ-1 - Browser shall implement an automatic update mechanism that checks for and applies security updates without user intervention

Given: A conformant browser with UPD-1 or higher capability

Task: Verify that automatic updates prevent users from running vulnerable browser versions by ensuring security patches are promptly applied without requiring user awareness or action. Manual update processes fail because users often ignore update notifications or delay patching, leaving exploitable vulnerabilities active for extended periods. Automatic updates close this security gap by applying critical fixes as soon as they become available.

Verification:

1. Install the browser and configure it to allow automatic updates (verify this is the default setting)

2. Check the browser's update settings to confirm automatic updates are enabled
3. Access the browser's internal update interface (chrome://settings/help, about:preferences#general, or edge://settings/help)
4. Monitor the browser's update check schedule by reviewing internal logs or using network monitoring tools
5. Simulate an available update by configuring a test update server or using the browser's internal testing mechanisms
6. Verify that the browser automatically checks for updates at the configured interval (typically daily)
7. Confirm that updates are downloaded in the background without blocking browser operation
8. Verify that updates are applied either at browser restart or in the background without user intervention
9. Test that the browser prompts for restart only when necessary for update installation
10. Verify that the automatic update mechanism continues to function across browser restarts and system reboots
11. Automatic update checks occur at regular intervals without user action
12. Updates are downloaded in the background without interrupting browsing
13. Update installation is automated with minimal or no user interaction required
14. Browser update status is clearly displayed to the user
15. Update checks occur even when the browser is running in background mode
16. Failed update attempts are retried automatically

Pass Criteria: Browser automatically checks for updates at least daily AND downloads and applies updates without mandatory user intervention AND update status is visible to users

Fail Criteria: Automatic updates require manual user action to initiate OR updates fail to check automatically OR update mechanism can be permanently disabled by users

Evidence: Screenshots of automatic update settings and status pages, network packet captures showing update check requests, browser internal logs showing update schedule, time-stamped video of automatic update process, configuration file exports showing update settings

References:

- Chrome Update Architecture: https://chromium.googlesource.com/chromium/src/+/main/docs/updater/functional_spe
- Firefox Update System: https://wiki.mozilla.org/Software_Update
- Microsoft Edge Update Policies: https://docs.microsoft.com/en-us/deployedge/microsoft-edge-update-policies
- NIST SP 800-40 Guide to Enterprise Patch Management: https://csrc.nist.gov/publications/detail/sp/800-40/rev-4/final
- CIS Browser Security Benchmarks: https://www.cisecurity.org/benchmark/google_chrome

Assessment: UPD-REQ-2 (Update signature verification)

Reference: UPD-REQ-2 - Browser shall verify cryptographic signatures of all updates before installation using trusted public keys

Given: A conformant browser with UPD-1 or higher capability

Task: Verify that update signature verification prevents distribution of malicious browser versions by ensuring only authentic vendor-signed packages can be installed, even if update infrastructure is compromised or man-in-the-middle attacks occur. Without cryptographic signature verification, attackers who compromise update servers or intercept update traffic can inject backdoored browser binaries, resulting in complete system compromise for all affected users.

Verification:

1. Obtain a legitimate browser update package from the official distribution channel
2. Extract the update package and locate the cryptographic signature file or embedded signature
3. Verify the signature algorithm used (should be RSA-4096, ECDSA-P384, or stronger)
4. Attempt to modify the update package contents and observe that signature verification fails

5. Create a test update package signed with an untrusted key and attempt to install it
6. Monitor the browser's update process using system call tracing tools to verify signature verification occurs
7. Check that the browser's trusted public keys are embedded in the binary or stored in a protected location
8. Verify that signature verification failures prevent update installation and generate error logs
9. Test that the browser rejects updates with missing signatures
10. Confirm that signature verification occurs before any update content is executed
11. All update packages include valid cryptographic signatures
12. Signature verification uses strong cryptographic algorithms (RSA-4096, ECDSA-P384, or better)
13. Modified update packages are rejected due to signature verification failure
14. Updates signed with untrusted keys are rejected
15. Signature verification occurs before any update code execution
16. Verification failures are logged and reported to the user
17. Trusted public keys are protected from tampering

Pass Criteria: All updates are cryptographically signed AND signatures are verified before installation AND modified or incorrectly signed updates are rejected

Fail Criteria: Updates can be installed without valid signatures OR signature verification can be bypassed OR weak cryptographic algorithms are used (RSA-2048 or weaker)

Evidence: Update package signature files, signature verification logs, network traces of update downloads showing signature transmission, system call traces showing verification process, test results from modified update packages, cryptographic algorithm analysis

References:

- Mozilla Code Signing Policy: https://wiki.mozilla.org/Security/Binary_Transparency
- NIST FIPS 186-5 Digital Signature Standard: https://csrc.nist.gov/publications/detail/fips/186/5/final
- Authenticode Code Signing (Windows): https://docs.microsoft.com/en-us/windows-hardware/drivers/install/authenticod
- Apple Code Signing Guide: https://developer.apple.com/library/archive/documentation/Security/Conceptual/CodeSigni
- OWASP Code Signing Best Practices: https://cheatsheetseries.owasp.org/cheatsheets/Third_Party_Javascript_Manager

Assessment: UPD-REQ-3 (HTTPS-only update delivery)

Reference: UPD-REQ-3 - Browser shall download all updates exclusively over HTTPS with certificate validation and pinning

Given: A conformant browser with UPD-1 or higher capability

Task: Verify that HTTPS-only update delivery protects update packages from interception and modification during transit, preventing network attackers from injecting malicious code even before signature verification. While signature verification provides end-to-end authenticity, transport encryption prevents metadata leakage about which browser versions are vulnerable and adds defense-in-depth against implementation flaws in signature verification.

Verification:

1. Configure a network proxy to intercept browser update traffic
2. Monitor network traffic during an update check to verify all connections use HTTPS
3. Attempt to redirect update requests to HTTP endpoints using DNS spoofing or proxy manipulation
4. Verify that the browser rejects HTTP update URLs and only accepts HTTPS
5. Test certificate validation by presenting an invalid certificate for the update server
6. Check if the browser implements certificate pinning for update domains
7. Attempt a man-in-the-middle attack on the update channel using a rogue certificate
8. Verify that TLS 1.2 or higher is used for all update connections
9. Test that the browser verifies the update server's hostname matches the certificate

10. Confirm that update requests include proper TLS configurations (strong cipher suites, no deprecated protocols)
11. All update downloads occur exclusively over HTTPS connections
12. HTTP update URLs are rejected or automatically upgraded to HTTPS
13. TLS 1.2 or TLS 1.3 is enforced for all update traffic
14. Certificate validation is performed with proper hostname verification
15. Certificate pinning is implemented for update domains (Chrome, Edge)
16. Man-in-the-middle attacks on update channels are detected and blocked

- Strong cipher suites are negotiated (AES-GCM, ChaCha20-Poly1305)
- Deprecated protocols (TLS 1.0, TLS 1.1, SSLv3) are rejected

Pass Criteria: All update traffic uses HTTPS with TLS 1.2+ AND certificate validation is enforced AND HTTP update URLs are rejected AND certificate pinning is implemented for critical update domains

Fail Criteria: Updates can be downloaded over HTTP OR TLS 1.1 or earlier is accepted OR certificate validation can be bypassed OR no certificate pinning is implemented

Evidence: Network packet captures showing HTTPS-only update traffic, TLS handshake analysis, certificate chain validation logs, test results from HTTP redirect attempts, man-in-the-middle attack test results, cipher suite negotiation logs

References:

- Chrome Update Server Pinning: https://www.chromium.org/Home/chromium-security/security-faq/#TOC-How-does-key-pinning-work-in-Chrome-
- Mozilla Update Server Security: https://wiki.mozilla.org/Security/Server_Side_TLS
- NIST SP 800-52 TLS Guidelines: https://csrc.nist.gov/publications/detail/sp/800-52/rev-2/final
- IETF RFC 8446 TLS 1.3: https://www.rfc-editor.org/rfc/rfc8446
- OWASP Transport Layer Security: https://cheatsheetseries.owasp.org/cheatsheets/Transport_Layer_Security_Cheat_S
- Certificate Pinning Best Practices: https://owasp.org/www-community/controls/Certificate_and_Public_Key_Pinning

Assessment: UPD-REQ-4 (Update manifest integrity)

Reference: UPD-REQ-4 - Browser shall verify the integrity of update manifests containing version information, file hashes, and metadata

Given: A conformant browser with UPD-1 or higher capability

Task: Verify that update manifest integrity protects the update metadata from tampering, ensuring that version information, file hashes, and download URLs cannot be manipulated by attackers to redirect updates to malicious files or bypass security checks. Compromised manifests can trick browsers into downloading wrong versions or accepting manipulated update files even when signature verification is implemented.

Verification:

1. Capture an update manifest file during a browser update check (typically JSON or XML format)
2. Examine the manifest structure to verify it includes: version number, file hashes, file sizes, signature, and metadata
3. Verify that the manifest itself is signed and the signature is validated before processing
4. Attempt to modify the manifest file (change version number or file hashes) and verify it is rejected
5. Test that each file listed in the manifest includes a cryptographic hash (SHA-256 or stronger)
6. Verify that downloaded update files are checked against the manifest hashes before installation
7. Test that mismatched file hashes prevent installation and trigger error handling
8. Check that the manifest includes timestamp information to prevent replay attacks
9. Verify that the manifest specifies minimum supported versions for rollback protection
10. Confirm that manifest validation failures are logged and prevent update installation
11. Update manifests are cryptographically signed and signatures are verified
12. Manifests include complete metadata: version, file hashes (SHA-256+), sizes, timestamps

13. Modified manifests are rejected due to signature verification failures
14. Each update file is verified against manifest hashes before installation
15. Hash mismatches prevent installation and generate error logs
16. Manifests include rollback protection information (minimum version)
17. Timestamp validation prevents replay attacks
18. Manifest structure follows a well-defined schema

Pass Criteria: Update manifests are signed and verified AND contain file hashes (SHA-256+) for all components AND hash verification is enforced before installation AND manifest tampering is detected and rejected

Fail Criteria: Manifests are not signed OR file hashes are missing or not verified OR modified manifests are accepted OR weak hash algorithms (MD5, SHA-1) are used

Evidence: Update manifest files in JSON/XML format, manifest signature verification logs, test results from modified manifests, hash verification logs, network captures showing manifest downloads, schema validation results

References:

- Chrome Update Manifest Format: https://chromium.googlesource.com/chromium/src/+/main/docs/updater/protocol_3
- The Update Framework (TUF) Specification: https://theupdateframework.github.io/specification/latest/
- NIST SP 800-107 Hash Function Recommendations: https://csrc.nist.gov/publications/detail/sp/800-107/rev-1/final
- Google Omaha Protocol: https://github.com/google/omaha/blob/main/doc/ServerProtocolV3.md
- OWASP Software Supply Chain Security: https://owasp.org/www-project-software-component-verification-standard/

Assessment: UPD-REQ-5 (Rollback protection)

Reference: UPD-REQ-5 - Browser shall implement rollback protection to prevent installation of older versions with known vulnerabilities

Given: A conformant browser with UPD-1 or higher capability

Task: Verify that rollback protection prevents attackers from downgrading browsers to older versions containing known exploitable vulnerabilities, blocking a common attack vector where adversaries force installation of vulnerable software to exploit publicly disclosed flaws. Without rollback protection, attackers who compromise update infrastructure or perform man-in-the-middle attacks can systematically downgrade browser installations to versions with known, weaponized exploits.

Verification:

1. Identify the currently installed browser version number
2. Obtain an older version of the browser update package (at least 2-3 versions older)
3. Attempt to install the older update package through the automatic update mechanism
4. Verify that the browser rejects the installation with an appropriate error message
5. Check the browser's update configuration for minimum version enforcement policies
6. Test that the browser maintains a record of the highest version ever installed
7. Attempt to manually downgrade the browser by installing an older version package
8. Verify that critical security updates cannot be rolled back even by administrators
9. Test that the rollback protection persists across browser reinstallations (if applicable)
10. Confirm that rollback protection errors are logged with version information
11. Browser rejects installation of older versions through automatic update mechanism
12. Minimum version numbers are enforced based on security requirements
13. Manual downgrade attempts are blocked or generate security warnings
14. Browser maintains persistent version history to detect rollback attempts
15. Critical security milestone versions cannot be downgraded
16. Rollback protection applies to both full updates and component updates

17. Clear error messages indicate rollback protection enforcement
18. Version comparison logic handles all version number formats correctly

Pass Criteria: Browser prevents installation of older versions through automatic updates AND maintains version history to detect rollbacks AND critical security versions cannot be downgraded

Fail Criteria: Older versions can be installed through update mechanism OR no minimum version enforcement exists OR rollback protection can be easily bypassed

Evidence: Update rejection logs showing version mismatch errors, test results from downgrade attempts, version history configuration files, error messages from rollback protection, policy configuration showing minimum version requirements

References:

- Chrome Version Enforcement: https://chromeenterprise.google/policies/#TargetVersionPrefix
- Android Rollback Protection: https://source.android.com/docs/security/features/verifiedboot/verified-boot#rollback-protection
- TUF Rollback Attack Prevention: https://theupdateframework.io/docs/security/
- NIST Cybersecurity Framework - Update Management: https://www.nist.gov/cyberframework
- Mozilla Update Security Guidelines: https://wiki.mozilla.org/Security/Server_Side_TLS
- Microsoft Update Rollback Protection: https://docs.microsoft.com/en-us/windows/deployment/update/waas-restart

Assessment: UPD-REQ-6 (Update channel isolation)

Reference: UPD-REQ-6 - Browser shall maintain isolation between update channels (stable, beta, dev) with independent signature verification

Given: A conformant browser with UPD-1 or higher capability supporting multiple channels

Task: Verify that update channel isolation prevents attackers from injecting unstable or experimental browser versions into production deployments by ensuring strict separation between stable, beta, and development update streams. Without channel isolation, compromised development infrastructure or testing servers could be leveraged to push untested code to stable channel users, introducing vulnerabilities or instability into production environments.

Verification:

1. Identify the available update channels for the browser (stable, beta, dev, canary)
2. Install the browser in stable channel mode and verify the update channel configuration
3. Examine the update server URLs for each channel to verify they are distinct
4. Attempt to inject a beta or dev channel update into a stable channel installation
5. Verify that each channel uses separate signature verification keys or key policies
6. Test that switching between channels requires explicit user action and confirmation
7. Check that channel metadata is included in update manifests and verified
8. Attempt to downgrade from stable to beta channel and verify the security implications are communicated
9. Verify that each channel has independent version numbering and rollback protection
10. Confirm that cross-channel update injection attempts are logged and rejected
11. Each update channel has distinct update server endpoints
12. Channel-specific signature verification prevents cross-channel update injection
13. Update manifests include channel identification that is verified
14. Channel switching requires explicit user consent and configuration changes
15. Downgrading channels (stable to beta/dev) triggers security warnings
16. Channel isolation prevents malicious update server redirection
17. Each channel maintains independent version history and rollback protection
18. Cross-channel update attempts are detected and logged

Pass Criteria: Update channels use separate server endpoints AND channel-specific signature verification prevents injection AND channel switching requires explicit user action AND cross-channel updates are rejected

Fail Criteria: Update channels share signature keys OR cross-channel updates can be injected OR no channel verification in update manifests OR channel switching occurs without user consent

Evidence: Update server URL configurations for each channel, signature verification key policies, channel metadata in update manifests, test results from cross-channel injection attempts, channel switching logs, network traces showing channel-specific endpoints

References:

- Chrome Release Channels: https://www.chromium.org/getting-involved/dev-channel/
- Firefox Release Management: https://wiki.mozilla.org/Release_Management/Release_Process
- Microsoft Edge Channels: https://docs.microsoft.com/en-us/deployedge/microsoft-edge-channels
- Google Chrome Enterprise Channel Management: https://support.google.com/chrome/a/answer/9982578
- NIST Secure Software Development Framework: https://csrc.nist.gov/Projects/ssdf

Assessment: UPD-REQ-7 (Component update support)

Reference: UPD-REQ-7 - Browser shall support independent security updates for components (rendering engine, JavaScript engine, libraries) without full browser updates

Given: A conformant browser with UPD-2 or higher capability

Task: Verify that component-level updates enable rapid patching of critical subsystems like JavaScript engines or rendering engines without waiting for full browser release cycles, reducing the window of exposure for component-specific vulnerabilities. Monolithic update systems delay security fixes because all components must be tested together, while independent component updates allow targeted, accelerated security patching for high-risk subsystems.

Verification:

1. Identify the browser's major components that support independent updates (e.g., V8, Chromium base, libraries)
2. Monitor the update mechanism to detect component-specific update packages
3. Verify that component updates include version information and dependency specifications
4. Test that component updates are applied without requiring a full browser restart when possible
5. Attempt to install incompatible component versions and verify dependency checking prevents installation
6. Check that component updates follow the same security verification as full updates (signatures, HTTPS)
7. Verify that component update manifests specify compatibility with browser versions
8. Test that critical component vulnerabilities can be patched independently of the full release schedule
9. Confirm that component updates maintain rollback protection independently
10. Verify that component update status is visible in the browser's update interface
11. Individual components can be updated independently of full browser updates
12. Component updates include version and dependency metadata
13. Component updates are signed and verified like full browser updates
14. Incompatible component versions are rejected based on dependency checking
15. Component updates can be applied with minimal or no browser restart
16. Critical security components can be updated on accelerated schedules
17. Component update history is tracked separately from full browser versions
18. Update interface displays component-level version information

Pass Criteria: Browser supports independent component updates with signature verification AND dependency checking prevents incompatible installations AND component updates follow same security verification as full updates

Fail Criteria: No support for component-level updates OR component updates bypass security verification OR no dependency checking for component compatibility

Evidence: Component update manifest files, component version listings from browser internals, update logs showing component-specific updates, dependency verification results, signature verification for component updates, test results from incompatible component installations

References:

- Chrome Component Updates: https://chromium.googlesource.com/chromium/src/+/main/components/component_upd
- V8 Engine Versioning: https://v8.dev/docs/version-numbers
- WebRTC Component Updates: https://webrtc.github.io/webrtc-org/release-notes/
- NIST Software Component Verification: https://csrc.nist.gov/Projects/cyber-supply-chain-risk-management
- OWASP Dependency Check: https://owasp.org/www-project-dependency-check/

Assessment: UPD-REQ-8 (Emergency update capability)

Reference: UPD-REQ-8 - Browser shall support emergency update mechanism for critical zero-day vulnerabilities with accelerated deployment

Given: A conformant browser with UPD-1 or higher capability

Task: Verify that emergency update mechanisms enable rapid deployment of critical zero-day patches to all users before exploit code becomes widely available, minimizing the window of vulnerability exploitation. Standard rollout schedules of days or weeks are unacceptable for actively exploited vulnerabilities, requiring accelerated deployment paths that bypass normal staged rollouts while maintaining security verification integrity.

Verification:

1. Review the browser's update documentation for emergency or critical update procedures
2. Examine update check frequency configuration and verify it can be increased for critical updates
3. Test that the browser can be configured to prioritize critical security updates over feature updates
4. Simulate an emergency update scenario by configuring a test update with high-priority flags
5. Verify that emergency updates bypass normal staged rollout delays
6. Test that users receive prominent notifications for critical security updates requiring immediate action
7. Check that emergency updates can trigger forced restarts with appropriate user warnings
8. Verify that emergency update flags cannot be spoofed by malicious update servers
9. Test that emergency updates are logged with high-priority markers for audit purposes
10. Confirm that emergency update mechanisms include additional verification to prevent abuse
11. Browser supports accelerated update checks for critical security updates
12. Emergency updates include priority flags in update manifests
13. Critical updates bypass staged rollout mechanisms for faster deployment
14. Users receive prominent notifications for emergency security updates
15. Emergency updates can trigger forced restarts with clear security justification
16. Emergency update flags are authenticated and cannot be spoofed
17. Audit logs distinguish emergency updates from regular updates
18. Emergency update mechanism includes safeguards against abuse

Pass Criteria: Browser supports emergency update mechanism with accelerated deployment AND emergency updates bypass normal rollout delays AND priority flags are authenticated AND users are clearly notified of critical updates

Fail Criteria: No emergency update mechanism exists OR emergency updates follow normal rollout schedule OR priority flags can be spoofed OR users cannot distinguish critical from regular updates

Evidence: Emergency update configuration documentation, test results from simulated critical updates, update priority flags in manifests, notification screenshots for critical updates, audit logs showing emergency update markers, rollout bypass verification

References:

- Chrome Critical Update Deployment: https://chromereleases.googleblog.com/
- Firefox Critical Updates: https://www.mozilla.org/en-US/security/advisories/
- NIST SP 800-40 Emergency Patching: https://csrc.nist.gov/publications/detail/sp/800-40/rev-4/final
- CERT Vulnerability Disclosure: https://vuls.cert.org/confluence/display/Wiki/Vulnerability+Disclosure+Policy
- Microsoft Security Response Center: https://www.microsoft.com/en-us/msrc

Assessment: UPD-REQ-9 (Update verification before installation)

Reference: UPD-REQ-9 - Browser shall perform complete integrity verification of all update files before beginning installation process

Given: A conformant browser with UPD-1 or higher capability

Task: Verify that update verification prevents installation of compromised or tampered browser builds, ensuring only authentic vendor-signed updates can be applied even if the update server or distribution network is compromised. This prevents attackers from injecting malicious code through man-in-the-middle attacks, compromised CDNs, or corrupted downloads that bypass cryptographic verification.

Verification:

1. Monitor the browser update process to identify the verification phase before installation
2. Capture update files during download and verify that hashes match the update manifest
3. Corrupt an update file after download but before installation to test verification
4. Verify that corrupted files are detected and re-downloaded before installation proceeds
5. Test that partial downloads are detected and not installed
6. Check that all update files are verified against the signed manifest before any installation steps
7. Verify that installation only begins after all files pass integrity checks
8. Test that verification failures trigger error handling and logging
9. Confirm that failed verifications do not leave the browser in an unstable state
10. Verify that the verification process cannot be bypassed or interrupted
11. All update files are verified against cryptographic hashes before installation
12. Hash algorithms used are SHA-256 or stronger
13. Corrupted or modified files are detected and rejected
14. Partial downloads are detected and completed before installation
15. Installation begins only after complete verification of all update components
16. Verification failures trigger automatic re-download or error reporting
17. Failed verifications are logged with specific error information
18. Browser remains in stable state if verification fails

Pass Criteria: All update files are verified with cryptographic hashes (SHA-256+) before installation AND corrupted files are detected and re-downloaded AND installation only proceeds after complete verification

Fail Criteria: Update files are installed without verification OR weak hash algorithms are used OR corrupted files can be installed OR verification can be bypassed

Evidence: Update verification logs showing hash checks, test results from corrupted file installations, network traces showing re-download behavior, hash algorithm analysis, installation process flow documentation, error handling logs

References:

- Chrome Update Verification: https://chromium.googlesource.com/chromium/src/+/main/docs/updater/functional_spec
- Mozilla Update Integrity Checks: https://wiki.mozilla.org/Software_Update#Integrity_Checks
- NIST Hash Function Security: https://csrc.nist.gov/projects/hash-functions
- TUF Consistent Snapshot Protection: https://theupdateframework.github.io/specification/latest/#consistent-snapshots
- ISO/IEC 29147 Vulnerability Disclosure: https://www.iso.org/standard/72311.html

Assessment: UPD-REQ-10 (Update failure recovery)

Reference: UPD-REQ-10 - Browser shall implement robust failure recovery mechanisms to restore functionality if update installation fails

Given: A conformant browser with UPD-1 or higher capability

Task: Verify that update failure recovery mechanisms maintain browser availability and security posture even when updates fail, preventing denial-of-service scenarios where failed updates render the browser unusable or leave it in a vulnerable state. This ensures users retain access to a functional, secure browser and prevents attackers from exploiting update failures to create persistent availability issues.

Verification:

1. Create a test scenario where update installation fails (disk full, permission denied, file corruption)
2. Trigger an update and simulate installation failure at various stages (download, verification, installation)
3. Verify that the browser rolls back to the previous working version after installation failure
4. Test that the browser remains functional and can be launched after a failed update
5. Verify that failed update state is detected and reported to the user with clear messaging
6. Test that the browser automatically retries failed updates with exponential backoff
7. Simulate power loss or system crash during update installation
8. Verify that the browser can recover from interrupted updates on next launch
9. Test that update failure recovery preserves user data and settings
10. Confirm that recovery processes are logged for troubleshooting
11. Failed updates do not prevent browser from launching or functioning
12. Browser automatically rolls back to previous version after installation failures
13. User data and settings are preserved through failed update attempts
14. Clear error messages indicate update failure causes
15. Automatic retry mechanisms with exponential backoff are implemented
16. Interrupted updates (power loss, crash) are detected and recovered on next launch
17. Recovery processes maintain browser stability and security
18. All recovery actions are logged with timestamps and error codes

Pass Criteria: Browser remains functional after update failures AND automatically rolls back to working version AND retries failed updates automatically AND preserves user data through failures

Fail Criteria: Failed updates prevent browser launch OR no rollback mechanism exists OR user data is corrupted by failed updates OR no automatic retry for failed updates

Evidence: Update failure logs with error codes, rollback process documentation, test results from simulated failures (disk full, crash, corruption), user data integrity verification, retry attempt logs with backoff timing, recovery process screenshots

References:

- Chrome Update Recovery: https://chromium.googlesource.com/chromium/src/+/main/docs/updater/functional_spec.m handling
- NIST Resilience Engineering: https://csrc.nist.gov/glossary/term/resilience
- Google Omaha Error Handling: https://github.com/google/omaha/blob/main/doc/ServerProtocolV3.md#error-codes

Assessment: UPD-REQ-11 (Update transparency logging)

Reference: UPD-REQ-11 - Browser shall implement update transparency logging to create auditable records of all update activities

Given: A conformant browser with UPD-1 or higher capability and LOG-1 or higher capability

Task: Verify that update transparency logging creates auditable records of all update activities, enabling detection of compromised update infrastructure, supply chain attacks, or unauthorized modifications to the

browser. This provides forensic evidence for security incident investigation and enables organizations to verify that only legitimate, authorized updates were applied to their browser fleet.

Verification:

1. Enable update logging through browser configuration or developer tools
2. Perform a complete update cycle and capture all generated logs
3. Verify that logs include: update check timestamps, available versions, download start/completion, verification results, installation status
4. Test that update logs include cryptographic hashes of installed components
5. Verify that signature verification results are logged with key identifiers
6. Check that update server URLs and responses are logged for audit purposes
7. Test that all update failures are logged with specific error codes and context
8. Verify that logs include user actions related to updates (manual checks, deferrals, channel changes)
9. Confirm that update logs can be exported for external analysis or compliance reporting
10. Test that update logs are protected from tampering and include integrity verification
11. All update activities are logged with timestamps and version information
12. Logs include download sources, file hashes, and signature verification results
13. Update failures are logged with detailed error information
14. User actions related to updates are captured in logs
15. Logs are structured and machine-readable (JSON or similar format)
16. Update logs can be exported for compliance and audit purposes
17. Logs include sufficient detail for security incident investigation
18. Log integrity is protected through checksums or signing

Pass Criteria: All update activities are logged with complete details AND logs include hashes and verification results AND logs are exportable in structured format AND log integrity is protected

Fail Criteria: Update activities are not logged OR logs lack critical information (timestamps, versions, hashes) OR logs cannot be exported OR logs can be tampered with

Evidence: Complete update log files showing full update cycle, exported logs in JSON/structured format, log integrity verification results, test results showing failure logging, screenshots of update log interfaces, compliance report samples

References:

- Chrome Update Logging: https://chromium.googlesource.com/chromium/src/+/main/docs/updater/functional_spec.md
- Binary Transparency: https://wiki.mozilla.org/Security/Binary_Transparency
- NIST SP 800-92 Log Management: https://csrc.nist.gov/publications/detail/sp/800-92/final
- Google Binary Authorization: https://cloud.google.com/binary-authorization/docs
- Certificate Transparency RFC 6962: https://www.rfc-editor.org/rfc/rfc6962
- OWASP Logging Cheat Sheet: https://cheatsheetseries.owasp.org/cheatsheets/Logging_Cheat_Sheet.html

Assessment: UPD-REQ-12 (Delta update security)

Reference: UPD-REQ-12 - Browser shall securely implement delta updates (patches) with same security verification as full updates

Given: A conformant browser with UPD-2 or higher capability supporting delta updates

Task: Verify that delta update security prevents attackers from exploiting differential patching mechanisms to inject malicious code through crafted patch files that bypass full binary verification. Delta updates introduce unique attack vectors where malicious patches could transform a legitimate binary into a compromised one if the patch itself and the resulting binary are not both cryptographically verified against known-good hashes.

Verification:

1. Monitor the browser update mechanism to detect delta update packages (smaller than full updates)
2. Verify that delta updates are offered only when the current version is compatible

3. Capture a delta update package and examine its structure and signature
4. Verify that delta updates are cryptographically signed independently from full updates
5. Test that applying a delta update includes verification of the resulting files against expected hashes
6. Attempt to apply a delta update to an incompatible base version and verify it is rejected
7. Test that delta update application includes atomic operations (all-or-nothing installation)
8. Verify that failed delta updates can fall back to full update downloads
9. Check that delta updates include integrity checks for both the patch and the result
10. Confirm that delta update security matches or exceeds full update security
11. Delta updates are cryptographically signed and verified before application
12. Source version verification ensures delta is compatible with installed version
13. Resulting files after delta application are verified against target hashes
14. Incompatible delta updates are rejected with fallback to full updates
15. Delta update application is atomic (complete or rollback)
16. Both patch integrity and result integrity are verified
17. Delta updates use same or stronger cryptography as full updates
18. Delta update failures trigger automatic fallback mechanisms

Pass Criteria: Delta updates are signed and verified independently AND source version compatibility is checked AND resulting files are verified against target hashes AND fallback to full updates on failure

Fail Criteria: Delta updates bypass security verification OR no source version checking OR result files not verified OR no fallback mechanism for failed deltas

Evidence: Delta update package analysis showing signatures, delta vs full update size comparisons, verification logs for delta application, test results from incompatible delta applications, fallback mechanism demonstrations, atomic operation verification

References:

- Google Courgette (Delta Compression): https://www.chromium.org/developers/design-documents/software-updates-courgette/
- Binary Diff Security: https://theupdateframework.github.io/specification/latest/#targets-metadata
- Mozilla MAR Format (Mozilla Archive): https://wiki.mozilla.org/Software_Update:MAR
- Microsoft Delta Updates: https://docs.microsoft.com/en-us/windows/deployment/update/psfxwhitepaper
- NIST Software Patch Security: https://csrc.nist.gov/glossary/term/patch

Assessment: UPD-REQ-13 (Update server authentication)

Reference: UPD-REQ-13 - Browser shall authenticate update servers using certificate validation, pinning, and domain verification

Given: A conformant browser with UPD-1 or higher capability

Task: Verify that update server authentication prevents man-in-the-middle attacks and DNS hijacking attempts that could redirect browsers to malicious update servers distributing compromised builds. Certificate pinning and domain verification ensure that even if an attacker obtains a valid TLS certificate through a compromised Certificate Authority or DNS spoofing, the browser will reject connections to fraudulent update servers.

Verification:

1. Capture network traffic during update checks to identify update server domains
2. Examine TLS certificates presented by update servers for validity and chain of trust
3. Verify that update server domains match expected values hardcoded in browser or configuration
4. Test certificate pinning by attempting to present a valid but unpinned certificate for update domain
5. Attempt DNS spoofing to redirect update requests to malicious servers
6. Verify that update server certificate validation includes OCSP/CRL checks
7. Test that expired or revoked certificates for update servers are rejected
8. Check that update server authentication occurs before any update data is processed

9. Verify that update server domains use HSTS to prevent protocol downgrade attacks
10. Confirm that failed authentication prevents update checks and is logged
11. Update servers present valid TLS certificates with complete chain of trust
12. Certificate pinning is implemented for update server domains
13. Update server domains are verified against expected values
14. DNS spoofing attempts are detected through certificate pinning
15. Certificate revocation status is checked via OCSP or CRL
16. Expired or revoked certificates cause update failures
17. HSTS is enforced for update server domains
18. Authentication failures are logged and prevent update downloads

Pass Criteria: Update servers are authenticated with certificate validation AND certificate pinning is implemented AND domain verification prevents spoofing AND revocation checking is performed

Fail Criteria: No certificate pinning for update servers OR domain verification can be bypassed OR revocation checking is not performed OR expired certificates are accepted

Evidence: Update server TLS certificates and chains, certificate pinning configurations, network traces showing authentication, test results from DNS spoofing attempts, OCSP/CRL check logs, HSTS policy verification, authentication failure logs

References:

- Chrome Certificate Pinning: https://www.chromium.org/Home/chromium-security/security-faq/#TOC-How-does-key-pinning-work-in-Chrome-
- Certificate Transparency Monitoring: https://certificate.transparency.dev/
- IETF RFC 7469 Public Key Pinning: https://www.rfc-editor.org/rfc/rfc7469
- OCSP Stapling: https://www.rfc-editor.org/rfc/rfc6066#section-8
- Mozilla Update Server Security: https://wiki.mozilla.org/Security/Server_Side_TLS
- OWASP Certificate Pinning: https://owasp.org/www-community/controls/Certificate_and_Public_Key_Pinning

Assessment: UPD-REQ-14 (Update timing jitter)

Reference: UPD-REQ-14 - Browser shall implement randomized timing jitter for update checks to prevent server load spikes and timing analysis

Given: A conformant browser with UPD-1 or higher capability

Task: Verify that update timing jitter prevents thundering herd problems that could enable denial-of-service attacks against update infrastructure or timing analysis attacks that reveal browser deployment patterns. Synchronized update checks from millions of browsers could overwhelm update servers or allow attackers to identify organizational update policies through traffic analysis, making randomized jitter a critical availability and privacy protection.

Verification:

1. Configure multiple browser instances (at least 10) with synchronized clocks
2. Monitor update check timing for each instance over a 24-hour period
3. Calculate the distribution of update check times relative to the scheduled interval
4. Verify that update checks are not synchronized across all instances
5. Measure the jitter range (randomization window) applied to update checks
6. Test that jitter is applied even when manual update checks are performed
7. Verify that jitter does not delay critical security updates beyond acceptable windows
8. Check that jitter implementation prevents timing analysis attacks
9. Test that jitter ranges are appropriate (e.g., ±1-2 hours for daily checks)
10. Confirm that jitter is cryptographically random, not predictable
11. Update checks include randomized timing jitter to distribute server load
12. Jitter range is appropriate for the update check frequency (typically ±10-20%)
13. Multiple instances do not synchronize update checks at same time

14. Jitter uses cryptographically secure random number generation
15. Critical updates can bypass jitter for immediate deployment when needed
16. Jitter prevents timing analysis attacks on update behavior
17. Jitter does not excessively delay important security updates
18. Update check distribution follows expected random distribution

Pass Criteria: Update checks include random jitter of at least ±10% of check interval AND jitter uses cryptographic randomness AND multiple instances show distributed check times AND critical updates can bypass jitter

Fail Criteria: No timing jitter implemented OR jitter is predictable OR all instances synchronize checks OR critical updates are delayed by jitter

Evidence: Update check timing logs from multiple instances, statistical analysis of check time distribution, jitter range configuration, randomness quality analysis, test results showing de-synchronized checks, critical update bypass demonstrations

References:

- Chrome Update Timing: https://chromium.googlesource.com/chromium/src/+/main/docs/updater/functional_spec.md checks
- Thundering Herd Problem: https://en.wikipedia.org/wiki/Thundering_herd_problem
- NIST Randomness Recommendations: https://csrc.nist.gov/projects/random-bit-generation
- Google Omaha Protocol Timing: https://github.com/google/omaha/blob/main/doc/ServerProtocolV3.md#update-check-timing
- Load Balancing Best Practices: https://aws.amazon.com/architecture/well-architected/

Assessment: UPD-REQ-15 (Background update enforcement)

Reference: UPD-REQ-15 - Browser shall enforce background update processes that continue even when browser is not actively running

Given: A conformant browser with UPD-2 or higher capability

Task: Verify that background update enforcement ensures security updates are applied even when users rarely launch the browser, preventing scenarios where unpatched browsers accumulate critical vulnerabilities. Without background updates, attackers can target users who infrequently use their browsers but still have them installed, exploiting the extended window of vulnerability between releases and actual patching.

Verification:

1. Close all browser windows completely to ensure browser is not running
2. Monitor system processes to verify background update service remains active
3. Wait for the scheduled update check interval with browser closed
4. Verify that update checks occur even when browser is not running
5. Test that background update service starts automatically at system boot
6. Simulate an available update and verify it downloads in background while browser is closed
7. Test that background updates can wake the system from sleep if configured (platform-dependent)
8. Verify that background update service has appropriate system permissions but runs with minimal privileges
9. Check that background update process is resistant to termination by users or malware
10. Confirm that background updates respect system resource constraints (network metering, battery status)
11. Background update service remains active when browser is closed
12. Update checks occur on schedule regardless of browser running state
13. Background service starts automatically at system boot
14. Updates can download and install without browser being open
15. Background service runs with minimal necessary privileges
16. Service cannot be easily disabled by users or malicious software

17. Resource-aware update behavior (respects metered connections, battery)
18. Background service restarts automatically if terminated abnormally

Pass Criteria: Background update service runs independently of browser AND performs update checks on schedule when browser closed AND downloads updates in background AND restarts automatically if terminated

Fail Criteria: Update service requires browser to be running OR no background update checks when closed OR service can be easily disabled OR does not restart after termination

Evidence: System process listings showing background service, update logs showing checks while browser closed, service configuration and permissions, automatic restart verification, resource usage monitoring, test results from service termination attempts

References:

- Chrome Background Updates: https://chromium.googlesource.com/chromium/src/+/main/docs/updater/functional_spe mode
- Windows Update Service Architecture: https://docs.microsoft.com/en-us/windows/deployment/update/how-windows-update-works
- macOS Launch Agents: https://developer.apple.com/library/archive/documentation/MacOSX/Conceptual/BPSystemSta
- Linux systemd Services: https://www.freedesktop.org/software/systemd/man/systemd.service.html
- Firefox Background Update Service: https://support.mozilla.org/en-US/kb/enable-background-updates-firefox-windows
- NIST Automated Patch Management: https://csrc.nist.gov/publications/detail/sp/800-40/rev-4/final

Assessment: UPD-REQ-16 (Update notification UI)

Reference: UPD-REQ-16 - Browser shall provide clear, user-friendly notifications about available updates and security status without enabling user suppression of critical updates

Given: A conformant browser with UPD-1 or higher capability

Task: Verify that update notification UI balances security enforcement with user experience, ensuring critical security updates cannot be suppressed while avoiding notification fatigue that causes users to ignore all update prompts. This prevents social engineering attacks where users habitually dismiss security notifications and ensures that truly critical updates demanding immediate attention are distinguished from routine maintenance updates.

Verification:

1. Trigger an available update and observe the notification mechanism
2. Verify that update notifications are visible but non-intrusive (e.g., menu icon, subtle indicator)
3. Test that critical security updates generate more prominent notifications than feature updates
4. Attempt to dismiss or suppress update notifications and verify critical updates cannot be permanently suppressed
5. Verify that update notifications include clear information about update type (security vs feature)
6. Test that users can defer non-critical updates but not critical security updates
7. Check that update status is always accessible through browser settings or help menu
8. Verify that notifications include estimated update size and installation time
9. Test that update notifications are accessible (screen reader compatible, high contrast support)
10. Confirm that update UI clearly distinguishes between "check for updates" and current update status
11. Update notifications are displayed prominently but do not block browsing
12. Critical security updates have more prominent notifications than feature updates
13. Update type and importance are clearly communicated to users
14. Critical security updates cannot be permanently dismissed or ignored
15. Non-critical updates can be deferred by users with clear re-notification
16. Update status is always visible in browser settings/help menu
17. Notifications include helpful details (update size, type, installation requirements)

18. Update UI is accessible to users with disabilities
19. Clear distinction between available updates and current version status

Pass Criteria: Update notifications are clear and accessible AND critical updates cannot be permanently suppressed AND update type and importance are communicated AND users can defer non-critical updates

Fail Criteria: No update notifications OR critical updates can be permanently suppressed OR update type unclear OR notifications block browser usage OR inaccessible UI

Evidence: Screenshots of update notifications for various update types, test results from notification dismissal attempts, accessibility testing results (screen reader, high contrast), user flow documentation, notification timing and frequency logs

References:

- Chrome Update UI: https://support.google.com/chrome/answer/95414
- Firefox Update Preferences: https://support.mozilla.org/en-US/kb/update-firefox-latest-release
- WCAG 2.1 Accessibility Guidelines: https://www.w3.org/WAI/WCAG21/quickref/
- Microsoft UI Design Principles: https://docs.microsoft.com/en-us/windows/apps/design/
- NIST Usability and Security: https://www.nist.gov/itl/applied-cybersecurity/tig/back-basics-multi-factor-authentication

Assessment: UPD-REQ-17 (Forced update for critical vulnerabilities)

Reference: UPD-REQ-17 - Browser shall support forced update mechanisms for critical vulnerabilities that require immediate patching

Given: A conformant browser with UPD-1 or higher capability

Task: Verify that forced update mechanisms can override user preferences when actively exploited critical vulnerabilities require immediate patching, preventing scenarios where user inaction leaves browsers vulnerable to widespread attacks. This emergency response capability must be protected against abuse through cryptographic verification while ensuring that users running vulnerable versions are protected even if they attempt to defer updates.

Verification:

1. Review browser documentation for forced update or killswitch mechanisms
2. Simulate a critical vulnerability scenario requiring immediate update
3. Verify that forced updates can override user deferral preferences
4. Test that forced updates trigger mandatory browser restart with clear security messaging
5. Check that forced update status is indicated in update manifests with verifiable flags
6. Attempt to bypass or cancel a forced update and verify it cannot be avoided
7. Verify that forced updates include grace periods with countdown timers before enforcement
8. Test that forced updates can disable the browser if update fails or is unavailable
9. Check that forced update mechanism includes safeguards against abuse (signed manifests, trusted channels)
10. Confirm that forced update events are logged with justification and admin override options (if applicable)
11. Browser supports forced update mechanism for critical security issues
12. Forced updates override user preferences and deferral settings
13. Clear security messaging explains necessity of forced updates to users
14. Forced update flags in manifests are cryptographically verified
15. Grace periods provide countdown timers before mandatory restart
16. Forced updates cannot be bypassed or permanently cancelled
17. Browser may be disabled if critical update cannot be applied
18. Forced update mechanism is protected against spoofing and abuse
19. Enterprise environments may have limited admin override with logging

Pass Criteria: Forced update mechanism exists for critical vulnerabilities AND overrides user deferrals AND provides clear security messaging AND forced update flags are cryptographically verified AND includes grace periods

Fail Criteria: No forced update mechanism OR can be bypassed by users OR no verification of forced update flags OR no security messaging explaining necessity

Evidence: Forced update configuration documentation, test results from simulated critical updates, user notification screenshots, forced update manifest flags, grace period timer demonstrations, bypass attempt results, audit logs for forced updates

References:

- Chrome Component Killswitch: https://www.chromium.org/administrators/policy-list-3#ComponentUpdatesEnabled
- Firefox Blocklist System: https://wiki.mozilla.org/Blocklisting
- Microsoft Forced Update Policies: https://docs.microsoft.com/en-us/deployedge/microsoft-edge-update-policies#updatedefault
- CVE Critical Severity Guidelines: https://www.first.org/cvss/
- NIST Critical Patch Management: https://csrc.nist.gov/publications/detail/sp/800-40/rev-4/final

Assessment: UPD-REQ-18 (Update verification chain)

Reference: UPD-REQ-18 - Browser shall implement complete chain-of-trust verification from update manifest through component signatures to final installation

Given: A conformant browser with UPD-1 or higher capability

Task: Verify that complete chain-of-trust verification prevents sophisticated supply chain attacks where attackers compromise individual components of the update distribution system. By requiring cryptographic validation at every level from root keys through manifests to individual component signatures, the browser ensures that compromise of any single element in the distribution chain cannot result in installation of malicious updates.

Verification:

1. Trace the complete update verification chain from initial update check to installation
2. Verify that update manifest is signed by trusted root key embedded in browser
3. Examine the signature chain: root key → manifest → component hashes → component signatures
4. Test that each level of the verification chain is validated before proceeding
5. Attempt to break the chain by presenting valid components with invalid manifest
6. Verify that intermediate certificate revocation breaks the verification chain
7. Test that the root keys used for update verification are embedded in browser binary and protected
8. Check that chain-of-trust validation occurs even for component updates
9. Verify that the entire verification chain is logged for audit purposes
10. Confirm that any break in verification chain prevents installation and triggers errors
11. Complete chain-of-trust from root keys to installed components
12. Root keys for update verification are embedded in browser binary
13. Update manifests are signed and verified against trusted root keys
14. Component signatures are verified against manifest hashes
15. Each verification step is performed in sequence with no shortcuts
16. Any break in verification chain prevents installation
17. Intermediate certificate revocation is detected and enforced
18. Verification chain is logged with details at each step
19. Chain-of-trust applies to both full and component updates

Pass Criteria: Complete verification chain from root keys to components AND each level is validated before proceeding AND breaks in chain prevent installation AND root keys are embedded and protected

Fail Criteria: Incomplete verification chain OR steps can be skipped OR root keys not embedded OR chain breaks do not prevent installation

Evidence: Verification chain documentation and diagrams, root key extraction from browser binary, signature verification logs at each chain level, test results from chain break attempts, intermediate certificate revocation tests, audit logs showing complete chain verification

References:

- Chrome Root Certificate Program: https://www.chromium.org/Home/chromium-security/root-ca-policy/
- Code Signing Certificate Chains: https://docs.microsoft.com/en-us/windows-hardware/drivers/install/digital-signatures
- X.509 Certificate Path Validation: https://www.rfc-editor.org/rfc/rfc5280#section-6
- Mozilla Root Store Policy: https://www.mozilla.org/en-US/about/governance/policies/security-group/certs/policy/
- TUF Root of Trust: https://theupdateframework.github.io/specification/latest/#root-metadata
- NIST Trust Anchor Management: https://csrc.nist.gov/publications/detail/sp/800-57-part-1/rev-5/final

Assessment: UPD-REQ-19 (Update source pinning)

Reference: UPD-REQ-19 - Browser shall implement update source pinning to prevent malicious redirection to unauthorized update servers

Given: A conformant browser with UPD-1 or higher capability

Task: Verify that update source pinning prevents malicious redirection attacks that could deliver compromised updates through DNS hijacking, BGP routing attacks, or compromised enterprise proxies. Hardcoded, pinned update server domains ensure that even if network infrastructure is compromised, browsers will only accept updates from authentic vendor-controlled servers validated through certificate pinning.

Verification:

1. Identify the configured update server URLs embedded in browser binary or configuration
2. Verify that update server domains are pinned and cannot be modified through configuration files
3. Attempt to modify system DNS or hosts file to redirect update domains to malicious servers
4. Test that certificate pinning for update domains prevents MITM attacks
5. Verify that update server URLs use multiple trusted domains with fallback mechanisms
6. Attempt to inject malicious update server URLs through registry/preference modifications
7. Test that update source pinning is enforced even for enterprise-managed deployments
8. Check that hardcoded update domains cannot be overridden by network-based attacks
9. Verify that fallback update servers are also pinned and verified
10. Confirm that update source pinning violations are logged and reported
11. Update server domains are hardcoded in browser binary
12. Update source URLs cannot be modified through configuration or registry
13. DNS redirection attacks are prevented through certificate pinning
14. Multiple trusted update domains with verified fallback mechanisms
15. Enterprise policies cannot override update source pinning for security updates
16. Network-based redirection attempts are detected and blocked
17. Fallback servers are subject to same pinning requirements
18. Source pinning violations are logged as security events

Pass Criteria: Update server domains are hardcoded AND cannot be modified by configuration OR DNS attacks AND certificate pinning prevents redirection AND fallback servers are also pinned

Fail Criteria: Update servers can be modified through configuration OR DNS redirection succeeds OR no certificate pinning OR enterprise policies can override source pinning

Evidence: Decompiled browser binary showing hardcoded update URLs, test results from DNS redirection attempts, certificate pinning verification, configuration modification tests, enterprise policy override tests, source pinning violation logs

References:

- Chrome Update Server Infrastructure: https://chromium.googlesource.com/chromium/src/+/main/docs/updater/protoc server
- Certificate Pinning Implementation: https://owasp.org/www-community/controls/Certificate_and_Public_Key_Pinnin
- DNS Rebinding Protection: https://en.wikipedia.org/wiki/DNS_rebinding
- Mozilla Update Server Security: https://wiki.mozilla.org/Software_Update#Security
- Google Omaha Server Protocol: https://github.com/google/omaha/blob/main/doc/ServerProtocolV3.md
- NIST Supply Chain Risk Management: https://csrc.nist.gov/Projects/cyber-supply-chain-risk-management

Assessment: UPD-REQ-20 (Update integrity verification)

Reference: UPD-REQ-20 - Browser shall verify integrity of installed components after update application to detect corruption or tampering

Given: A conformant browser with UPD-1 or higher capability

Task: Verify that post-installation integrity verification detects tampering or corruption that occurs after updates are applied, protecting against attacks where malware modifies browser components between installation and execution. This closes the security window where installed files are vulnerable to modification before integrity checks run, ensuring that any tampering triggers immediate remediation.

Verification:

1. Perform a complete browser update and monitor the post-installation verification phase
2. Verify that installed files are checked against expected cryptographic hashes after installation
3. Attempt to modify an installed file immediately after update completion
4. Test that modified files are detected on next browser launch
5. Verify that integrity verification occurs both after installation and at browser startup
6. Check that integrity verification covers all critical components (binaries, libraries, resources)
7. Test that integrity failures trigger repair or re-installation mechanisms
8. Verify that integrity verification results are logged with specific failure details
9. Test that browser refuses to launch if critical component integrity verification fails
10. Confirm that integrity verification uses same cryptographic strength as update verification (SHA-256+)
11. Post-installation integrity verification checks all installed components
12. Cryptographic hashes (SHA-256+) are used for integrity verification
13. Modified files are detected immediately after installation
14. Startup integrity checks detect tampering between sessions
15. Critical components are verified (binaries, libraries, key resources)
16. Integrity failures trigger automatic repair or update mechanisms
17. Browser refuses to launch if critical components fail verification
18. Integrity verification results are logged with specific error details
19. Verification coverage includes all security-critical components

Pass Criteria: Post-installation integrity verification is performed AND uses strong cryptographic hashes (SHA-256+) AND modified files are detected AND integrity failures trigger repair OR prevent launch

Fail Criteria: No post-installation verification OR weak hash algorithms OR modified files not detected OR integrity failures do not trigger corrective actions

Evidence: Post-installation verification logs, hash algorithm analysis, test results from file modification attacks, startup integrity check logs, repair mechanism demonstrations, critical component verification coverage analysis

References:

- Chrome Component Integrity Verification: https://www.chromium.org/Home/chromium-security/education/tls#TOC-Certificate-Verification
- Windows Code Integrity: https://docs.microsoft.com/en-us/windows/security/threat-protection/windows-defender-application-control/windows-defender-application-control
- macOS Gatekeeper and Code Signing: https://developer.apple.com/documentation/security/notarizing_macos_software_
- Linux Integrity Measurement Architecture: https://sourceforge.net/p/linux-ima/wiki/Home/
- NIST File Integrity Monitoring: https://csrc.nist.gov/glossary/term/file_integrity_monitoring
- OWASP Software Component Verification: https://owasp.org/www-project-software-component-verification-standard/

Assessment: UPD-REQ-21 (Staged rollout support)

Reference: UPD-REQ-21 - Browser shall support staged rollout mechanisms to gradually deploy updates to user populations with rollback capability

Given: A conformant browser with UPD-2 or higher capability

Task: Verify that staged rollout mechanisms limit the blast radius of defective updates while maintaining rapid response capability for critical security fixes. Gradual deployment with rollback capability prevents scenarios where buggy updates simultaneously impact millions of users, while emergency bypass ensures that actively exploited vulnerabilities can still be patched immediately across the entire user base.

Verification:

1. Review browser update architecture for staged rollout or canary deployment support
2. Verify that update manifests include rollout percentage or cohort targeting information
3. Test that updates are delivered to progressively larger user populations over time
4. Check that rollout stages are controlled by server-side configuration, not client manipulation
5. Verify that users cannot force-join or opt-out of rollout cohorts
6. Test that rollout can be paused or rolled back if issues are detected
7. Monitor multiple browser instances to observe different rollout timing
8. Verify that critical security updates can bypass staged rollout for immediate deployment
9. Test that rollout cohorts are assigned using stable user identifiers (not random per check)
10. Confirm that staged rollout status is logged for monitoring and analysis
11. Update system supports staged rollout with progressive deployment
12. Rollout percentages are specified in update manifests and enforced server-side
13. Users cannot manipulate rollout cohort assignment
14. Rollout can be paused or reversed if issues are detected
15. Critical security updates can bypass staged rollout
16. Cohort assignment is stable (same user gets consistent rollout timing)
17. Multiple instances show varied rollout timing based on cohort
18. Rollout status and cohort information are logged
19. Rollout progression follows defined stages (e.g., 1%, 10%, 50%, 100%)

Pass Criteria: Staged rollout mechanism is implemented AND rollout percentages are server-controlled AND users cannot manipulate cohorts AND critical updates can bypass rollout AND rollback capability exists

Fail Criteria: No staged rollout support OR users can manipulate rollout OR all users receive updates simultaneously OR no rollback capability OR critical updates follow slow rollout

Evidence: Update manifest files showing rollout percentages, server-side rollout configuration documentation, test results from multiple instances showing varied timing, rollout logs with cohort information, critical update bypass demonstrations, rollback process documentation

References:

- Chrome Staged Rollout: https://chromium.googlesource.com/chromium/src/+/main/docs/updater/functional_spec.md#rollout
- Google Omaha Rollout Mechanisms: https://github.com/google/omaha/blob/main/doc/ServerProtocolV3.md#rollout
- Canary Deployments: https://martinfowler.com/bliki/CanaryRelease.html
- Progressive Delivery Patterns: https://www.split.io/glossary/progressive-delivery/
- NIST Phased Deployment: https://csrc.nist.gov/publications/detail/sp/800-40/rev-4/final

Assessment: UPD-REQ-22 (Update domain validation)

Reference: UPD-REQ-22 - Browser shall validate that update requests and responses originate from authorized domains with strict certificate requirements

Given: A conformant browser with UPD-1 or higher capability

Task: Verify that update domain validation prevents server-side request forgery and domain confusion attacks where attackers redirect update requests to malicious servers through HTTP redirects, DNS manipulation, or compromised network infrastructure. Strict domain allowlisting ensures that browsers only download updates from explicitly authorized vendor domains, rejecting any deviation regardless of certificate validity.

Verification:

1. Capture update network traffic to identify all domains involved in update process
2. Verify that all update domains are whitelisted or hardcoded in browser configuration
3. Examine TLS certificates for update domains to verify they meet strict requirements (EV or equivalent)
4. Attempt to redirect update requests to unauthorized domains using DNS attacks
5. Test that update domain acceptlist cannot be modified by users or malware
6. Verify that update responses from unauthorized domains are rejected
7. Check that update domain validation includes subdomain restrictions (no wildcards)
8. Test that HTTP redirects to unauthorized domains during update process are rejected
9. Verify that update domain validation occurs for all update-related requests (manifests, downloads, telemetry)
10. Confirm that domain validation failures are logged with domain details
11. All update domains are explicitly whitelisted or hardcoded
12. TLS certificates for update domains meet strict requirements
13. Unauthorized domains are rejected even with valid TLS certificates
14. Domain acceptlist cannot be modified through configuration or registry
15. Subdomain validation is strict (no wildcard matching)
16. HTTP redirects to unauthorized domains are blocked
17. Domain validation applies to all update-related traffic
18. Validation failures are logged with attempted domain information
19. Update domain list is protected and verified at startup

Pass Criteria: Update domains are whitelisted/hardcoded AND unauthorized domains are rejected AND domain list cannot be modified AND validation applies to all update traffic AND redirects to unauthorized domains are blocked

Fail Criteria: No domain acceptlist OR unauthorized domains accepted OR domain list can be modified OR validation can be bypassed with redirects OR logging insufficient

Evidence: Network traffic captures showing update domains, domain acceptlist extraction from browser binary, test results from unauthorized domain redirects, TLS certificate analysis, domain validation logs, configuration modification attempt results

References:

- Chrome Update Domain Security: https://chromium.googlesource.com/chromium/src/+/main/docs/updater/protocol_3
- Mozilla Update Domain Policies: https://wiki.mozilla.org/Software_Update#Security
- DNS Security Extensions (DNSSEC): https://www.icann.org/resources/pages/dnssec-what-is-it-why-important-2019-03-05-en

- Extended Validation Certificates: https://en.wikipedia.org/wiki/Extended_Validation_Certificate
- OWASP Server-Side Request Forgery Prevention: https://cheatsheetseries.owasp.org/cheatsheets/Server_Side_Request_
- NIST Domain Validation Guidelines: https://csrc.nist.gov/publications/detail/sp/800-63/3/final

Assessment: UPD-REQ-23 (Update binary reproducibility)

Reference: UPD-REQ-23 - Browser shall support mechanisms to enable verification of binary reproducibility for update transparency and supply chain security

Given: A conformant browser with UPD-2 or higher capability

Task: Verify that binary reproducibility enables independent verification of update authenticity, protecting against sophisticated supply chain attacks where build infrastructure is compromised to inject backdoors. Reproducible builds allow security researchers and organizations to verify that distributed binaries match published source code, detecting unauthorized modifications introduced during compilation or packaging.

Verification:

1. Review browser build and release documentation for reproducible build support
2. Verify that update packages include build metadata (compiler version, build timestamp, source commit)
3. Attempt to reproduce an official update binary from published source code using documented build process
4. Compare reproduced binary hash with official release hash to verify reproducibility
5. Check that build process documentation includes all dependencies and toolchain versions
6. Verify that update transparency logs include binary hashes for independent verification
7. Test that multiple independent parties can reproduce identical binaries from same source
8. Check for availability of build attestations or signed build manifests
9. Verify that non-reproducible elements (timestamps, randomness) are minimized or eliminated
10. Confirm that reproducibility documentation and verification tools are publicly available
11. Browser supports reproducible builds with documented build process
12. Update packages include complete build metadata
13. Independent verification of binary reproducibility is possible
14. Build process documentation includes all dependencies and toolchain versions
15. Multiple builds from same source produce bit-identical binaries
16. Update transparency logs enable third-party verification
17. Non-deterministic build elements are minimized or documented
18. Build attestations or signed manifests are available
19. Public documentation and tools support reproducibility verification

Pass Criteria: Reproducible build process is documented AND build metadata is included in updates AND independent parties can verify binary reproducibility AND update transparency supports third-party verification

Fail Criteria: No reproducible build support OR build process not documented OR independent verification not possible OR build metadata missing OR excessive non-deterministic elements

Evidence: Build process documentation, reproduced binary hash comparisons, build metadata extraction from update packages, transparency log entries, independent build verification results, build attestation signatures, reproducibility verification tool outputs

References:

- Reproducible Builds Project: https://reproducible-builds.org/
- Chromium Build Documentation: https://chromium.googlesource.com/chromium/src/+/main/docs/linux/build_instruct
- Debian Reproducible Builds: https://wiki.debian.org/ReproducibleBuilds
- NIST Secure Software Development Framework: https://csrc.nist.gov/Projects/ssdf
- SLSA Supply Chain Security Framework: https://slsa.dev/

6.6 Protocol Handler Security Assessments

This section covers assessment procedures for requirements PRO-REQ-1 through PRO-REQ-23, addressing custom protocol handler registration, scheme hijacking prevention, protocol allowlisting, URL scheme security, mobile deep linking, and handler isolation.

Assessment: PRO-REQ-1 (Protocol handler registration validation)

Reference: PRO-REQ-1 - Browser shall validate all custom protocol handler registrations to ensure they meet security requirements before registration

Given: A conformant browser with PRO-1 or higher capability

Task: Verify that protocol handler registration validation prevents malicious websites from hijacking critical URL schemes or injecting XSS vectors through handler templates. Without strict validation, attackers could register handlers for privileged schemes like javascript: or craft handler URLs containing script injection, enabling cross-site scripting attacks when victims click protocol links.

Verification:

1. Navigate to a test page that attempts to register a custom protocol handler using registerProtocolHandler()
2. Test registration with a valid web+custom scheme (e.g., web+music, web+calendar)
3. Attempt to register a handler for a dangerous built-in scheme (http, https, file, javascript)
4. Verify that the browser rejects registration attempts for privileged schemes
5. Test registration with malformed URL templates (missing %s placeholder, invalid URL format)
6. Verify that only HTTPS origins can register protocol handlers (HTTP origins should be rejected)
7. Test that the handler URL template is properly validated for XSS vectors
8. Attempt to register handlers with data: or blob: URLs and verify rejection
9. Test registration from an iframe and verify same-origin policy enforcement
10. Verify that successful registrations require user permission or confirmation
11. Valid web+custom scheme registrations are accepted after validation
12. Built-in privileged schemes (http, https, file, javascript, data, blob) cannot be registered
13. Only HTTPS origins can register protocol handlers
14. URL templates shall include %s placeholder and be valid URLs
15. Malformed registrations are rejected with appropriate error messages
16. XSS vectors in handler URLs are sanitized or rejected
17. Registration from iframes follows same-origin policy
18. User permission is required before handler activation

Pass Criteria: Browser validates protocol registrations AND rejects privileged schemes AND requires HTTPS origins AND validates URL templates AND requires user permission

Fail Criteria: Privileged schemes can be registered OR HTTP origins accepted OR invalid URL templates accepted OR no user permission required OR XSS vectors not sanitized

Evidence: Screenshots of registration attempts for various schemes, console error messages for rejected registrations, permission prompt screenshots, network traces showing HTTPS enforcement, XSS test results, developer console outputs

References:

- WHATWG HTML Standard - registerProtocolHandler: https://html.spec.whatwg.org/multipage/system-state.html#custom-handlers
- MDN registerProtocolHandler API: https://developer.mozilla.org/en-US/docs/Web/API/Navigator/registerProtocolHan
- Custom URL Scheme Security: https://tools.ietf.org/html/rfc8252#section-7.1
- Protocol Handler Security Best Practices: https://www.chromium.org/developers/design-documents/create-amazing-password-forms/
- OWASP URL Validation: https://cheatsheetseries.owasp.org/cheatsheets/Input_Validation_Cheat_Sheet.html

- Web Application Security Working Group: https://www.w3.org/2011/webappsec/

Assessment: PRO-REQ-2 (User consent for custom protocols)

Reference: PRO-REQ-2 - Browser shall obtain explicit user consent before activating custom protocol handlers

Given: A conformant browser with PRO-1 or higher capability

Task: Verify that protocol handler consent requirements prevent malicious websites from silently registering handlers that could launch local applications or exfiltrate data through custom protocols without user knowledge. Explicit consent ensures users understand which websites can intercept specific protocol schemes, protecting against protocol handler hijacking where attackers register handlers to intercept sensitive protocol activations.

Verification:

1. Register a custom protocol handler (web+test) on a test page
2. Verify that registration triggers a permission prompt to the user
3. Test that the permission prompt clearly identifies the scheme and handling origin
4. Click a link with the custom protocol (web+test:example) and verify activation prompt
5. Test that users can allow, deny, or remember the choice for the handler
6. Verify that denying handler activation falls back to default behavior or shows error
7. Test that remembered choices persist across browser sessions
8. Navigate to browser settings and verify users can revoke protocol handler permissions
9. Test that each origin requires separate consent (cross-origin isolation)
10. Verify that permission prompts cannot be spoofed or triggered without user action
11. Protocol handler registration shows clear permission prompt
12. Permission prompt identifies the custom scheme and requesting origin
13. Activation of custom protocol shows confirmation before launching handler
14. Users can allow, deny, or set persistent preferences
15. Permission choices persist across sessions
16. Settings provide UI to view and revoke handler permissions
17. Each origin requires independent user consent
18. Permission prompts are genuine browser UI (not web content)
19. Consent is required for each distinct protocol scheme

Pass Criteria: Explicit user consent required for registration AND activation prompts shown before launching AND permissions are manageable in settings AND cross-origin isolation enforced

Fail Criteria: No consent prompts displayed OR handlers activate without user permission OR permissions cannot be revoked OR cross-origin handlers share permissions

Evidence: Screenshots of permission prompts (registration and activation), settings UI showing handler permissions, video recording of consent flow, persistent permission test results, cross-origin permission isolation tests, prompt timing analysis

References:

- WHATWG HTML Standard - User Activation: https://html.spec.whatwg.org/multipage/interaction.html#tracking-user-activation
- Permissions API Specification: https://www.w3.org/TR/permissions/
- User Consent Best Practices: https://www.w3.org/TR/security-privacy-questionnaire/
- GDPR Consent Requirements: https://gdpr.eu/gdpr-consent-requirements/
- Chrome Permission UX Guidelines: https://developer.chrome.com/docs/extensions/mv3/permission_warnings/
- Mozilla Permission Prompts: https://support.mozilla.org/en-US/kb/permissions-manager-give-ability-store-passwords-set-cookies-more

Assessment: PRO-REQ-3 (Protocol allowlist enforcement)

Reference: PRO-REQ-3 - Browser shall enforce protocol allowlists that restrict which custom schemes can be registered

Given: A conformant browser with PRO-1 or higher capability

Task: Verify that protocol allowlist enforcement prevents registration of dangerous schemes that could enable code execution, data exfiltration, or privilege escalation through protocol handlers. Strict allowlisting blocks attackers from registering handlers for privileged schemes like vbscript: or shell: that could execute arbitrary code when activated.

Verification:

1. Review browser documentation for allowed custom protocol scheme patterns
2. Attempt to register a handler for web+validname and verify acceptance
3. Test registration with schemes that don't follow web+ prefix convention
4. Verify that safelisted schemes (mailto, tel, sms) can be registered with special handling
5. Attempt to register handlers for blocklisted dangerous schemes (vbscript, shell, etc.)
6. Test enterprise policy controls for custom protocol allowlists
7. Verify that scheme names follow DNS label conventions (alphanumeric, hyphens)
8. Test that scheme names are case-insensitive during registration and matching
9. Attempt registration with excessively long scheme names and verify length limits
10. Verify that only approved safelist schemes bypass the web+ prefix requirement
11. Custom schemes use web+ prefix unless on explicit safelist
12. Safelisted schemes (mailto, tel, sms) can be registered without web+ prefix
13. Dangerous schemes (javascript, data, vbscript, shell) are blocklisted
14. Scheme names follow DNS label conventions (alphanumeric, hyphens, no spaces)
15. Scheme matching is case-insensitive
16. Length limits prevent excessively long scheme names (e.g., 64 character limit)
17. Enterprise policies can extend or restrict allowlists
18. Invalid scheme patterns are rejected with clear error messages

Pass Criteria: web+ prefix required for custom schemes AND safelist exceptions work correctly AND blocklist prevents dangerous schemes AND scheme validation follows standards

Fail Criteria: web+ prefix not enforced OR dangerous schemes accepted OR invalid scheme patterns allowed OR no length limits

Evidence: Registration test results for various scheme patterns, error messages for rejected schemes, enterprise policy configuration examples, scheme validation test matrix, documentation of allowlist and blocklist

References:

- WHATWG URL Standard - Schemes: https://url.spec.whatwg.org/#schemes
- RFC 3986 URI Generic Syntax: https://www.rfc-editor.org/rfc/rfc3986#section-3.1
- Custom URL Scheme Guidelines: https://www.iana.org/assignments/uri-schemes/uri-schemes.xhtml
- Chromium URL Scheme List: https://source.chromium.org/chromium/chromium/src/+/main:url/url_constants.cc
- Safari Custom Protocol Handlers: https://developer.apple.com/documentation/xcode/defining-a-custom-url-scheme-for-your-app
- Mozilla Protocol Handler Allowlist: https://searchfox.org/mozilla-central/source/dom/base/nsContentUtils.cpp

Assessment: PRO-REQ-4 (Scheme hijacking prevention)

Reference: PRO-REQ-4 - Browser shall prevent scheme hijacking attacks where malicious handlers override legitimate protocol handlers

Given: A conformant browser with PRO-1 or higher capability

Task: Verify that scheme hijacking prevention protects users from malicious websites that attempt to override legitimate protocol handlers to intercept sensitive protocol activations. Without protection, attackers could register handlers for schemes like mailto: or custom banking protocols to capture credentials, redirect users, or exfiltrate data when victims click protocol links.

Verification:

1. Register a legitimate protocol handler for web+test from https://trusted.example.com
2. Attempt to register a competing handler for web+test from https://malicious.example.com
3. Verify that the browser either prevents the override or prompts user for choice
4. Test that the most recently used or user-preferred handler takes precedence
5. Verify that built-in handlers (mailto, tel) cannot be completely overridden without permission
6. Test that unregistering a handler doesn't automatically activate an attacker's handler
7. Attempt to register a handler that impersonates a well-known service
8. Verify that handler selection UI clearly shows the origin of each handler
9. Test that handlers registered in private/incognito mode don't persist
10. Verify that resetting browser settings revokes all custom protocol handlers
11. Multiple handlers for same scheme trigger user choice rather than silent override
12. User can see and select from all registered handlers for a scheme
13. Built-in handlers maintain priority or require explicit user override
14. Handler selection UI clearly displays origin and scheme information
15. Private/incognito mode handlers are session-only
16. Browser reset revokes all custom handlers
17. No automatic activation of handlers after unregistration
18. Handler precedence is deterministic and user-controllable

Pass Criteria: Multiple handlers for same scheme require user selection AND origins clearly displayed AND built-in handlers protected AND private mode isolation enforced

Fail Criteria: Silent override of existing handlers OR origins not displayed OR built-in handlers easily hijacked OR private mode handlers persist

Evidence: Screenshots of handler selection UI, multi-handler registration test results, private mode isolation verification, browser reset test results, handler precedence documentation, user choice recording

References:

- Same-Origin Policy for Handlers: https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy
- URL Scheme Hijacking Prevention: https://portswigger.net/web-security/dom-based/open-redirection
- OWASP URL Redirection Cheat Sheet: https://cheatsheetseries.owasp.org/cheatsheets/Unvalidated_Redirects_and_For
- Browser Handler Precedence: https://html.spec.whatwg.org/multipage/system-state.html#concept-handler-precedence

Assessment: PRO-REQ-5 (Protocol parameter sanitization)

Reference: PRO-REQ-5 - Browser shall sanitize protocol URL parameters before passing them to protocol handlers

Given: A conformant browser with PRO-1 or higher capability

Task: Verify that protocol parameter sanitization prevents injection attacks where malicious protocol URLs containing special characters could exploit vulnerabilities in handler implementations. Without proper encoding, protocol parameters containing script tags, SQL commands, or command injection payloads could execute in the context of the handling website or application, enabling XSS, SQLi, or remote code execution.

Verification:

1. Register a test protocol handler with URL template: https://handler.example.com/handle?url=%s
2. Create a test link with XSS payload: Click

3. Click the link and verify that the script payload is URL-encoded when passed to handler
4. Test with SQL injection characters in protocol URL and verify proper encoding
5. Attempt protocol URL with newline/CRLF characters and verify sanitization
6. Test with null bytes and other control characters in protocol URL
7. Verify that the %s placeholder is properly substituted with encoded parameter
8. Test with extremely long protocol parameters and verify truncation or rejection
9. Attempt to include additional parameters beyond the defined template
10. Verify that Unicode characters are properly normalized and encoded
11. All special characters in protocol URLs are properly URL-encoded
12. Script tags and JavaScript code are encoded, not executed
13. SQL injection characters are escaped/encoded
14. CRLF and newline characters are stripped or encoded
15. Null bytes and control characters are removed or encoded
16. Parameter substitution uses safe encoding (encodeURIComponent or equivalent)
17. Extremely long parameters are truncated or rejected
18. Template structure is enforced (no parameter injection)
19. Unicode is normalized (NFC) and safely encoded

Pass Criteria: All special characters properly URL-encoded AND XSS payloads neutralized AND injection attacks prevented AND template structure enforced

Fail Criteria: Special characters not encoded OR XSS possible through protocol parameters OR injection successful OR template can be modified

Evidence: XSS test results showing encoding, SQL injection test results, CRLF injection tests, parameter encoding examples, extremely long parameter handling, Unicode normalization tests, network traces showing encoded parameters

References:

- OWASP XSS Prevention Cheat Sheet: https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention
- URL Encoding RFC 3986: https://www.rfc-editor.org/rfc/rfc3986#section-2.1
- JavaScript URL Encoding: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/encod
- Parameter Injection Attacks: https://owasp.org/www-community/attacks/Command_Injection
- Content Security Policy: https://www.w3.org/TR/CSP3/

Assessment: PRO-REQ-6 (External protocol handler security)

Reference: PRO-REQ-6 - Browser shall implement security controls when launching external (OS-level) protocol handlers

Given: A conformant browser with PRO-1 or higher capability

Task: Verify that external protocol handler security prevents malicious websites from silently launching local applications with crafted parameters that could enable command injection or arbitrary code execution. Confirmation prompts and parameter sanitization ensure users understand what application is being launched and prevent attackers from exploiting vulnerable OS-level handlers through injection attacks.

Verification:

1. Configure an OS-level protocol handler for a custom scheme (e.g., myapp://)
2. Create a web page with a link to the custom protocol: Launch
3. Click the link and verify that a confirmation prompt appears before launching
4. Test that the confirmation clearly identifies the external application to be launched
5. Verify that parameters passed to external handlers are sanitized
6. Test launching external handlers with malicious parameters (command injection attempts)
7. Attempt to launch external handlers from iframes and verify restrictions
8. Test that user should click or interact to trigger external handler (no automatic launch)
9. Verify that external handler launches respect user preferences (allow/deny lists)

10. Test that repeated external handler launches don't bypass confirmation prompts
11. Confirmation prompt appears before launching any external protocol handler
12. Prompt identifies the target application and protocol scheme
13. User shall explicitly approve each launch (or set persistent preference)
14. Parameters are sanitized to prevent command injection
15. Automatic launch is prevented (requires user gesture)
16. Iframe restrictions prevent silent external handler launches
17. User preferences for external handlers are persistent and accessible
18. Allow/deny lists work correctly for external protocols
19. Repeated launches maintain security checks

Pass Criteria: Confirmation required for external handler launches AND application identified in prompt AND parameters sanitized AND user gesture required

Fail Criteria: External handlers launch without confirmation OR application not identified OR command injection possible OR automatic launch allowed

Evidence: Screenshots of external handler confirmation prompts, command injection test results, parameter sanitization verification, user gesture requirement tests, iframe restriction tests, preference persistence verification

References:

- External Protocol Handler Security: https://textslashplain.com/2019/08/28/browser-architecture-web-to-app-communication-overview/
- macOS URL Scheme Handling: https://developer.apple.com/documentation/xcode/defining-a-custom-url-scheme-for-your-app
- Linux Desktop Entry Specification: https://specifications.freedesktop.org/desktop-entry-spec/desktop-entry-spec-latest.html

Assessment: PRO-REQ-7 (Protocol handler UI transparency)

Reference: PRO-REQ-7 - Browser shall provide transparent UI that clearly indicates when protocol handlers are registered or invoked

Given: A conformant browser with PRO-1 or higher capability

Task: Verify that protocol handler UI transparency prevents deceptive handler registrations where users unknowingly grant protocol handling permissions without understanding the security implications. Transparent UI ensures users can identify, review, and revoke protocol handlers, protecting against social engineering attacks where malicious sites silently register handlers to intercept sensitive protocol activations.

Verification:

1. Navigate to a page and register a custom protocol handler
2. Verify that a clear notification or permission prompt appears during registration
3. Check that the browser UI shows an indicator when handlers are registered (e.g., icon in address bar)
4. Click a custom protocol link and verify that the handler invocation is visible to user
5. Test that handler management UI is accessible from browser settings
6. Verify that the settings UI lists all registered handlers with origins and schemes
7. Test that users can easily identify which handler will be invoked for a scheme
8. Verify that handler removal is straightforward from the settings UI
9. Test that the browser provides clear feedback when handler invocation fails
10. Verify that developer tools show protocol handler events for debugging
11. Registration triggers visible notification or permission request
12. Browser UI indicates when handlers are registered for current origin
13. Handler invocation shows clear user feedback (dialog, notification, or status)
14. Settings provide comprehensive handler management interface
15. All registered handlers listed with scheme, origin, and URL template

16. Handler selection and removal are user-friendly
17. Failed handler invocations show error messages
18. Developer console logs handler events
19. No silent or hidden handler operations

Pass Criteria: Registration and invocation visible to user AND settings provide handler management AND all handlers listed with details AND clear feedback for all operations

Fail Criteria: Silent handler operations OR no settings UI for management OR handlers hidden from user OR no feedback on invocation

Evidence: Screenshots of registration notifications, address bar indicators, settings UI showing handler list, handler invocation UI, error messages, developer console logs, video walkthrough of handler lifecycle

References:

- User Interface Security Principles: https://www.w3.org/TR/security-privacy-questionnaire/#questions
- Chrome Protocol Handler Settings: https://support.google.com/chrome/answer/114662
- Firefox Protocol Handler Management: https://support.mozilla.org/en-US/kb/change-program-used-open-email-links
- WCAG Accessibility Guidelines: https://www.w3.org/WAI/WCAG21/quickref/
- User-Centered Security Design: https://www.usenix.org/conference/soups2019

Assessment: PRO-REQ-8 (Protocol downgrade protection)

Reference: PRO-REQ-8 - Browser shall prevent protocol downgrade attacks where HTTPS handlers are replaced with HTTP handlers

Given: A conformant browser with PRO-1 or higher capability

Task: Verify that protocol downgrade protection prevents attackers from replacing secure HTTPS handlers with insecure HTTP handlers to enable man-in-the-middle attacks on protocol activations. Without downgrade protection, attackers could register HTTP handlers to intercept sensitive data passed through protocol URLs that were previously handled securely over HTTPS.

Verification:

1. Register a protocol handler from an HTTPS origin: https://secure.example.com/handler?url=%s
2. Attempt to register a handler for the same scheme from an HTTP origin: http://insecure.example.com/handler?url=%s
3. Verify that the HTTP handler registration is rejected or requires explicit user confirmation
4. Test that handler URL templates are HTTPS if registered from HTTPS origins
5. Attempt to update an existing HTTPS handler to use an HTTP URL template
6. Verify that downgrades are prevented or trigger security warnings
7. Test that HSTS policies affect protocol handler registration
8. Verify that mixed content policies apply to handler URL templates
9. Test that handler invocation follows upgrade-insecure-requests policy
10. Verify that downgrade attempts are logged for security monitoring
11. HTTP origins cannot register handlers if HTTPS origin has already registered
12. Handler URL templates are HTTPS for security-sensitive schemes
13. Existing HTTPS handlers cannot be downgraded to HTTP
14. HSTS policies prevent HTTP handler registration for HSTS hosts
15. Mixed content blocking applies to handler URLs
16. Upgrade-insecure-requests policy upgrades handler URLs when applicable
17. Downgrade attempts generate security warnings or errors
18. Security events are logged for downgrade attempts

Pass Criteria: HTTP handler registration blocked when HTTPS exists AND URL templates are HTTPS AND HSTS enforced for handlers AND downgrade attempts logged

Fail Criteria: HTTP handlers can override HTTPS handlers OR HTTP URL templates accepted OR HSTS not enforced OR no logging of downgrades

Evidence: Downgrade test results, HSTS enforcement verification, mixed content policy tests, security warning screenshots, event logs showing downgrade attempts, upgrade-insecure-requests verification

References:

- HTTP Strict Transport Security (HSTS) RFC 6797: https://www.rfc-editor.org/rfc/rfc6797
- Upgrade Insecure Requests: https://www.w3.org/TR/upgrade-insecure-requests/
- Mixed Content Specification: https://www.w3.org/TR/mixed-content/
- Chrome HTTPS Best Practices: https://developers.google.com/web/fundamentals/security/encrypt-in-transit/why-https
- Mozilla Web Security Guidelines: https://infosec.mozilla.org/guidelines/web_security

Assessment: PRO-REQ-9 (Protocol handler logging)

Reference: PRO-REQ-9 - Browser shall log protocol handler registration, modification, and invocation events for security auditing

Given: A conformant browser with PRO-1 or higher capability and LOG-1 or higher capability

Task: Verify that protocol handler logging creates comprehensive audit trails of handler lifecycle events, enabling detection of malicious handler registrations, unauthorized invocations, or suspicious patterns that could indicate compromise. Complete logging supports security investigations by providing forensic evidence of when handlers were registered, by whom, and how they were used.

Verification:

1. Enable security event logging in browser configuration
2. Register a custom protocol handler and verify the event is logged
3. Check that the log entry includes: timestamp, origin, scheme, handler URL template, user decision
4. Invoke a registered protocol handler and verify the invocation is logged
5. Modify an existing handler (if supported) and verify the change is logged
6. Unregister a protocol handler and verify the removal is logged
7. Test that failed registration attempts are logged with error reasons
8. Verify that external protocol handler launches are logged
9. Test that logs include sufficient context for security analysis
10. Export protocol handler logs and verify they are in structured format
11. All handler lifecycle events are logged (registration, invocation, modification, removal)
12. Log entries include complete metadata: timestamp, origin, scheme, URL template, user action
13. Failed registration attempts are logged with error details
14. External handler invocations are logged separately from web handlers
15. Logs distinguish between user-initiated and script-initiated events
16. Log format is structured (JSON or similar) for analysis
17. Logs can be exported for security monitoring
18. Log retention follows security event retention policies
19. Sensitive parameters are redacted from logs

Pass Criteria: All handler lifecycle events logged with complete metadata AND failed attempts logged AND logs exportable in structured format

Fail Criteria: Handler events not logged OR logs lack critical metadata OR failed attempts not logged OR logs not exportable

Evidence: Log exports showing handler events, log entry examples with metadata, failed registration logs, external handler invocation logs, log format documentation, retention policy verification

References:

- NIST SP 800-92 Log Management: https://csrc.nist.gov/publications/detail/sp/800-92/final
- W3C Reporting API: https://www.w3.org/TR/reporting-1/
- OWASP Logging Cheat Sheet: https://cheatsheetseries.owasp.org/cheatsheets/Logging_Cheat_Sheet.html
- Audit Trail Requirements: https://www.iso.org/standard/54534.html

Assessment: PRO-REQ-10 (Web+custom scheme support)

Reference: PRO-REQ-10 - Browser shall support web+custom scheme format for custom protocol handlers as specified by WHATWG

Given: A conformant browser with PRO-1 or higher capability

Task: Verify that web+custom scheme support enforces the WHATWG-specified naming convention that prevents namespace collisions with system-level or IANA-registered schemes. The web+ prefix requirement ensures custom web handlers are clearly distinguished from OS-level protocol handlers, preventing malicious sites from hijacking system protocols while enabling safe custom protocol functionality.

Verification:

1. Register a protocol handler using web+ prefix: registerProtocolHandler('web+music', 'https://handler.example.com/play?
2. Verify that the registration succeeds for properly formatted web+ schemes
3. Test that web+ schemes are case-insensitive (web+music equals WEB+MUSIC)
4. Create a link with web+music:track123 and verify it invokes the handler
5. Test that schemes without web+ prefix (except safelisted) are rejected
6. Verify that web+ is followed by at least one alphanumeric character
7. Test that web+ alone (without suffix) is rejected
8. Verify that web+ schemes follow DNS label rules (no spaces, special chars limited)
9. Test registration of multiple different web+ schemes from same origin
10. Verify that web+ handlers work correctly across different browser contexts
11. web+ prefix is recognized and properly handled
12. Schemes are case-insensitive during registration and matching
13. web+ shall be followed by valid scheme name (alphanumeric, hyphens)
14. web+ alone without suffix is invalid
15. Scheme names follow DNS label conventions
16. Multiple web+ schemes can be registered from same origin
17. Links with web+ schemes correctly invoke registered handlers
18. Error messages guide developers on correct web+ format
19. Cross-context invocation works (different tabs, windows)

Pass Criteria: web+ prefix required and recognized AND case-insensitive matching AND DNS label rules enforced AND multiple schemes supported per origin

Fail Criteria: web+ prefix not recognized OR case-sensitive matching OR invalid scheme names accepted OR only one scheme per origin

Evidence: Registration test results for various web+ formats, case sensitivity tests, scheme validation test matrix, multi-scheme registration examples, error message documentation, cross-context invocation tests

References:

- WHATWG HTML Standard - web+ Schemes: https://html.spec.whatwg.org/multipage/system-state.html#normalize-protocol-handler-parameters
- RFC 3986 URI Scheme Syntax: https://www.rfc-editor.org/rfc/rfc3986#section-3.1
- Custom Protocol Handler Specification: https://html.spec.whatwg.org/multipage/system-state.html#custom-handlers
- MDN web+ Scheme Documentation: https://developer.mozilla.org/en-US/docs/Web/API/Navigator/registerProtocolHa
- DNS Label Syntax RFC 1035: https://www.rfc-editor.org/rfc/rfc1035
- Browser Protocol Handler Implementation: https://www.chromium.org/developers/design-documents/create-amazing-password-forms/

Assessment: PRO-REQ-11 (Protocol handler persistence)

Reference: PRO-REQ-11 - Browser shall persist protocol handler registrations across sessions while respecting privacy modes

Given: A conformant browser with PRO-1 or higher capability

Task: Verify that protocol handler persistence balances usability with privacy by maintaining registrations across normal browser sessions while ensuring private mode handlers do not leak across sessions. This prevents privacy violations where private browsing handlers could reveal user activity history while maintaining functionality for legitimate persistent handler registrations.

Verification:

1. Register a protocol handler in normal browsing mode
2. Close the browser and reopen it
3. Verify that the protocol handler registration persists and is still functional
4. Click a custom protocol link and verify the handler still works after restart
5. Register a protocol handler in private/incognito mode
6. Verify that the handler works during the private session
7. Close the private/incognito window and open a new one
8. Verify that the handler registered in private mode does NOT persist
9. Test that clearing browsing data removes protocol handler registrations
10. Verify that exported browser profiles include protocol handler settings
11. Handler registrations persist across normal browser restarts
12. Registered handlers remain functional after session closure
13. Private/incognito mode handlers are session-only (do not persist)
14. Private mode handlers do not leak to normal mode or vice versa
15. Clearing browsing data removes handler registrations
16. Handler persistence respects user privacy preferences
17. Profile export/import includes handler configurations
18. Handler storage is properly synchronized in multi-device scenarios

Pass Criteria: Normal mode handlers persist across sessions AND private mode handlers are session-only AND data clearing removes handlers AND profile export includes handlers

Fail Criteria: Normal handlers don't persist OR private handlers persist OR data clearing doesn't remove handlers OR handlers not in profile export

Evidence: Persistence test results across restarts, private mode isolation verification, data clearing test results, profile export/import examples, multi-device sync verification (if applicable)

References:

- Browser Storage Persistence: https://storage.spec.whatwg.org/
- Private Browsing Mode Specification: https://www.w3.org/TR/tracking-dnt/#private-browsing
- Clear Browsing Data Specification: https://www.w3.org/TR/clear-site-data/
- Firefox Private Browsing: https://support.mozilla.org/en-US/kb/private-browsing-use-firefox-without-history
- Chrome Incognito Mode: https://support.google.com/chrome/answer/95464

Assessment: PRO-REQ-12 (Protocol confusion mitigation)

Reference: PRO-REQ-12 - Browser shall mitigate protocol confusion attacks where similar-looking schemes are used to deceive users

Given: A conformant browser with PRO-1 or higher capability

Task: Verify that protocol confusion mitigation prevents homograph attacks where visually similar scheme names using Unicode characters deceive users into trusting malicious handlers. Attackers could register

handlers for schemes that look identical to legitimate ones using Cyrillic or other non-Latin characters, tricking users into believing they're using trusted handlers while data is actually sent to attacker-controlled servers.

Verification:

1. Attempt to register a protocol handler using homoglyphs (e.g., web+test with Cyrillic 'e' instead of Latin 'e')
2. Verify that the browser normalizes Unicode in scheme names or rejects homoglyphs
3. Test registration with mixed scripts in scheme names (Latin + Cyrillic)
4. Attempt to register schemes that visually resemble built-in schemes (e.g., web+https with Cyrillic characters)
5. Verify that scheme names are restricted to ASCII alphanumeric and hyphens
6. Test that UI clearly displays scheme names without ambiguity
7. Verify that punycode or IDN homographs are not allowed in scheme names
8. Test registration of schemes with confusable characters (0 vs O, 1 vs l)
9. Verify that handler selection UI disambiguates similar schemes
10. Test that schemes with different Unicode normalization forms are treated as different
11. Scheme names are restricted to ASCII alphanumeric characters and hyphens
12. Unicode homoglyphs in scheme names are rejected
13. Mixed script scheme names are not allowed
14. Schemes visually similar to built-in schemes are flagged or rejected
15. UI displays scheme names in monospace or disambiguating font
16. Punycode/IDN encoding not allowed in scheme names
17. Confusable character combinations are prevented or warned
18. Different Unicode normalizations are properly handled
19. User is warned when schemes are potentially confusing

Pass Criteria: Scheme names restricted to ASCII AND homoglyphs rejected AND confusable schemes flagged AND UI clearly displays schemes

Fail Criteria: Unicode homoglyphs accepted OR mixed scripts allowed OR confusing schemes not flagged OR ambiguous UI display

Evidence: Homoglyph test results, Unicode normalization tests, confusable character test matrix, UI screenshots showing scheme display, rejection error messages, scheme validation code review

References:

- Unicode Security Considerations: https://www.unicode.org/reports/tr36/
- Homograph Attacks: https://en.wikipedia.org/wiki/IDN_homograph_attack
- ASCII URI Schemes RFC 3986: https://www.rfc-editor.org/rfc/rfc3986#section-3.1
- Chrome IDN Spoof Protection: https://chromium.googlesource.com/chromium/src/+/master/docs/idn.md
- OWASP Unicode Security: https://owasp.org/www-community/attacks/Unicode_Encoding

Assessment: PRO-REQ-13 (Handler capability restrictions)

Reference: PRO-REQ-13 - Browser shall restrict capabilities available to protocol handlers based on context and permissions

Given: A conformant browser with PRO-1 or higher capability

Task: Verify that handler capability restrictions prevent privilege escalation where malicious handlers exploit protocol invocation to gain unauthorized access to APIs or bypass security policies. Handlers must operate within their origin's security context without inheriting privileges from the protocol scheme itself, preventing attackers from using custom protocols to circumvent same-origin policy or Content Security Policy restrictions.

Verification:

1. Register a protocol handler and invoke it with a custom protocol URL

2. Verify that the handler page inherits security context from its origin, not protocol URL
3. Test that handler cannot access privileged APIs without proper permissions
4. Verify that protocol parameters don't grant additional capabilities
5. Test that handlers follow Content Security Policy of their origin
6. Attempt to access local files from handler and verify blocking
7. Test that handlers cannot bypass same-origin policy using protocol parameters
8. Verify that handlers in iframes have restricted capabilities
9. Test that handler invocation doesn't grant automatic permission escalation
10. Verify that handlers respect Permissions Policy (formerly Feature Policy)
11. Handler security context is based on handler URL origin, not protocol
12. Privileged APIs require explicit permissions (not granted by handler status)
13. CSP of handler origin is enforced
14. Protocol parameters cannot inject capabilities or bypass security
15. Local file access is blocked unless explicitly permitted
16. Same-origin policy is enforced for handlers
17. Iframe handlers have sandboxed capabilities
18. No automatic permission grants upon handler invocation
19. Permissions Policy restrictions are enforced

Pass Criteria: Handler capabilities based on origin AND CSP enforced AND no privilege escalation via protocol AND same-origin policy maintained

Fail Criteria: Protocol grants additional capabilities OR CSP bypassed OR privilege escalation possible OR same-origin policy violated

Evidence: Capability restriction test results, CSP enforcement verification, permission escalation tests (negative), same-origin policy tests, iframe sandbox verification, Permissions Policy tests

References:

- Content Security Policy Level 3: https://www.w3.org/TR/CSP3/
- Permissions Policy: https://www.w3.org/TR/permissions-policy-1/
- Same-Origin Policy: https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy
- HTML Iframe Sandbox: https://html.spec.whatwg.org/multipage/iframe-embed-object.html#attr-iframe-sandbox
- Web API Permissions: https://www.w3.org/TR/permissions/
- Browser Security Model: https://www.chromium.org/Home/chromium-security/

Assessment: PRO-REQ-14 (Protocol handler revocation)

Reference: PRO-REQ-14 - Browser shall provide mechanisms to revoke protocol handler registrations

Given: A conformant browser with PRO-1 or higher capability

Task: Verify that protocol handler revocation mechanisms enable users to remove unwanted or malicious handlers that may have been registered through social engineering or deceptive UIs. Without effective revocation, users would be permanently stuck with handlers they didn't intend to register, allowing attackers to maintain persistent control over protocol scheme handling even after users discover the deception.

Verification:

1. Register multiple protocol handlers from different origins
2. Navigate to browser settings and locate protocol handler management section
3. Verify that all registered handlers are listed with scheme, origin, and URL template
4. Revoke a specific handler using the settings UI
5. Verify that the revoked handler no longer appears in the handler list
6. Test that clicking custom protocol links no longer invokes the revoked handler
7. Verify that unregisterProtocolHandler() API can programmatically revoke handlers
8. Test that only the registering origin can programmatically unregister its handlers

9. Verify that clearing site data revokes all handlers from that origin
10. Test that browser reset removes all custom protocol handlers
11. Settings UI provides clear handler management interface
12. All registered handlers are visible with complete details
13. Individual handlers can be selectively revoked
14. Revoked handlers stop functioning immediately
15. unregisterProtocolHandler() API works correctly
16. Cross-origin revocation is prevented (only origin can unregister its handlers)
17. Site data clearing includes handler revocation
18. Browser reset removes all handlers
19. Revocation events are logged for audit purposes

Pass Criteria: Settings UI allows handler revocation AND unregisterProtocolHandler() API works AND cross-origin revocation prevented AND site clearing removes handlers

Fail Criteria: No revocation mechanism OR handlers persist after revocation OR cross-origin revocation possible OR site clearing doesn't affect handlers

Evidence: Settings UI screenshots showing handler management, revocation test results, unregisterProtocolHandler() examples, cross-origin revocation prevention tests, site data clearing verification, browser reset verification

References:

- WHATWG unregisterProtocolHandler API: https://html.spec.whatwg.org/multipage/system-state.html#dom-navigator-unregisterprotocolhandler
- Clear Site Data Specification: https://www.w3.org/TR/clear-site-data/
- Browser Settings Best Practices: https://www.w3.org/TR/security-privacy-questionnaire/
- Chrome Protocol Handler Settings: https://support.google.com/chrome/answer/114662
- Firefox Handler Management: https://support.mozilla.org/en-US/kb/change-program-used-open-email-links
- User Control and Transparency: https://www.w3.org/TR/privacy-principles/#user-control

Assessment: PRO-REQ-15 (Cross-origin protocol restrictions)

Reference: PRO-REQ-15 - Browser shall enforce cross-origin restrictions for protocol handler registration and invocation

Given: A conformant browser with PRO-1 or higher capability

Task: Verify that cross-origin protocol restrictions enforce same-origin policy for handler registration and management, preventing malicious iframes from registering handlers on behalf of parent frames or vice versa. This isolation ensures that each origin maintains independent control over its handlers, preventing cross-origin attacks where embedded content could hijack the parent's protocol handling or manipulate handlers from other origins.

Verification:

1. Create a page at https://origin-a.example.com that registers a protocol handler
2. Create an iframe at https://origin-b.example.com embedded in origin-a
3. Attempt to register a protocol handler from the iframe
4. Verify that the handler is attributed to origin-b (iframe origin), not origin-a (parent)
5. Test that handlers registered by origin-a cannot be unregistered by origin-b
6. Create a cross-origin link to custom protocol and verify handler invocation
7. Test that handler URL templates match the registering origin
8. Verify that postMessage cannot be used to bypass handler origin restrictions
9. Test that CORS policies don't affect protocol handler registration
10. Verify that subdomains are treated as separate origins for handler registration
11. Protocol handlers are attributed to the registering origin (not parent frame)

12. Cross-origin iframes register handlers under their own origin
13. Handlers can only be modified/removed by their registering origin
14. Handler URL templates are same-origin with registration origin
15. Cross-origin invocation works but respects security boundaries
16. postMessage doesn't bypass origin restrictions
17. CORS policies are orthogonal to handler registration
18. Subdomain isolation is enforced (sub.example.com != example.com)
19. Public suffix list respected for origin determination

Pass Criteria: Handlers attributed to registering origin AND cross-origin modification prevented AND URL templates same-origin AND subdomain isolation enforced

Fail Criteria: Handler origin attribution incorrect OR cross-origin modification allowed OR URL templates can be cross-origin OR subdomain isolation violated

Evidence: Cross-origin registration tests, iframe attribution tests, cross-origin modification prevention tests, URL template origin validation, subdomain isolation tests, public suffix list verification

References:

- Same-Origin Policy: https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy
- HTML Origin Specification: https://html.spec.whatwg.org/multipage/origin.html
- CORS Specification: https://fetch.spec.whatwg.org/#http-cors-protocol
- Public Suffix List: https://publicsuffix.org/
- Iframe Security: https://html.spec.whatwg.org/multipage/iframe-embed-object.html
- Subdomain Security: https://tools.ietf.org/html/rfc6265#section-5.1.3

Assessment: PRO-REQ-16 (Protocol handler manifest validation)

Reference: PRO-REQ-16 - Browser shall validate protocol handler manifests in Progressive Web Apps and installed web applications

Given: A conformant browser with PRO-1 or higher capability and support for Web App Manifests

Task: Progressive Web Apps can declare protocol handlers in their manifest files, creating a persistent attack surface that bypasses runtime API restrictions. Malicious PWAs could register handlers for privileged schemes, escape manifest scope restrictions, or persist handlers after uninstallation to maintain access to user data. Rigorous manifest validation ensures PWA protocol handlers meet the same security standards as API-registered handlers while preventing manifest-based handler abuse.

Verification:

1. Create a PWA with a manifest file declaring protocol handlers in protocol_handlers field
2. Install the PWA and verify that declared protocol handlers are registered
3. Test manifest protocol handlers with invalid URL templates and verify rejection
4. Verify that manifest handlers require user permission before activation
5. Test that manifest protocol handlers follow same security rules as registerProtocolHandler
6. Attempt to declare privileged schemes (http, https) in manifest and verify rejection
7. Test that manifest handlers are unregistered when app is uninstalled
8. Verify that manifest scope restrictions apply to protocol handlers
9. Test manifest handler updates when app manifest is updated
10. Verify that manifest validation errors are reported to developer console
11. protocol_handlers field in manifest is parsed and validated
12. Declared handlers are registered upon app installation with user permission
13. Invalid handlers in manifest are rejected with clear error messages
14. Same security validations apply as registerProtocolHandler API
15. Privileged schemes cannot be registered via manifest
16. Handlers are removed when app is uninstalled
17. Handlers point to URLs within app scope

18. Manifest updates trigger handler re-validation
19. Developer console shows validation errors
20. OS-level protocol handler registration occurs for installed apps

Pass Criteria: Manifest protocol_handlers validated AND user permission required AND same security rules as API AND uninstalled app removes handlers

Fail Criteria: Invalid manifests accepted OR no user permission OR weaker security than API OR handlers persist after uninstall

Evidence: Manifest examples with protocol handlers, installation flow screenshots, validation error messages, uninstallation verification, scope restriction tests, OS protocol handler registration proof, update handling tests

References:

- Web App Manifest Specification: https://www.w3.org/TR/appmanifest/
- PWA Protocol Handlers: https://web.dev/url-protocol-handler/
- URL Handlers in PWA: https://github.com/WICG/pwa-url-handler/blob/main/explainer.md
- Chrome PWA Installation: https://web.dev/install-criteria/
- MDN Web App Manifest: https://developer.mozilla.org/en-US/docs/Web/Manifest
- App Scope Specification: https://www.w3.org/TR/appmanifest/#scope-member

Assessment: PRO-REQ-17 (Intent URL security - mobile)

Reference: PRO-REQ-17 - Browser shall implement security controls for Android Intent URLs and prevent intent scheme attacks

Given: A conformant mobile browser with PRO-1 or higher capability running on Android

Task: Android Intent URLs allow web pages to launch native applications with arbitrary parameters, creating a powerful attack vector for launching privileged system components, bypassing app permissions, or exfiltrating sensitive data through maliciously crafted intent extras. Intent scheme attacks can trigger dangerous actions like sending SMS, making calls, or accessing sensitive device features without proper authorization. Comprehensive intent URL validation prevents these attacks while maintaining legitimate app integration functionality.

Verification:

1. Create a test page with an intent URL: Click
2. Verify that clicking the intent link triggers appropriate security checks
3. Test that malicious intent URLs cannot launch privileged system components
4. Attempt intent URL with dangerous actions (SEND_SMS, CALL, etc.) and verify blocking or permission prompts
5. Test intent URLs with arbitrary extras and verify sanitization
6. Verify that intent URLs cannot bypass app permissions
7. Test that browser validates package names in intent URLs
8. Attempt to use intent URLs to exfiltrate data and verify prevention
9. Test that intent URLs from iframes have additional restrictions
10. Verify that intent URL invocation is logged for security auditing
11. Intent URLs are parsed and validated before invocation
12. Privileged system components cannot be launched via intent URLs
13. Dangerous actions trigger user permission prompts or are blocked
14. Intent extras are sanitized to prevent injection attacks
15. Intent URLs cannot bypass app permission model
16. Package name validation prevents targeting of unintended apps
17. Data exfiltration via intent URLs is prevented
18. Iframe intent URLs have restricted capabilities
19. Intent invocations are logged with target package and action

20. User confirmation required for sensitive intent actions

Pass Criteria: Intent URLs validated before launch AND privileged actions blocked AND extras sanitized AND user confirmation for sensitive actions AND invocations logged

Fail Criteria: Intent URLs launch without validation OR privileged actions allowed OR no sanitization OR no confirmation for sensitive actions OR not logged

Evidence: Intent URL test results, privileged action blocking verification, extras sanitization tests, permission prompt screenshots, iframe restriction tests, security log entries, data exfiltration prevention tests

References:

- Android Intent Specification: https://developer.android.com/reference/android/content/Intent
- Intent URL Scheme Security: https://tools.ietf.org/html/rfc8252#appendix-B.2
- Chrome Intent URLs: https://developer.chrome.com/docs/multidevice/android/intents/
- Android App Links: https://developer.android.com/training/app-links
- OWASP Mobile Security: https://owasp.org/www-project-mobile-security-testing-guide/
- Intent Security Best Practices: https://developer.android.com/guide/components/intents-filters#SafeIntent

Assessment: PRO-REQ-18 (Universal Links security - iOS)

Reference: PRO-REQ-18 - Browser shall implement security controls for iOS Universal Links and prevent link hijacking

Given: A conformant mobile browser with PRO-1 or higher capability running on iOS

Task: iOS Universal Links enable seamless web-to-app transitions but create risks of link hijacking, phishing through malicious association files, and privacy violations through link interception tracking. Attackers can serve fraudulent apple-app-site-association files over compromised connections or use HTTP downgrade attacks to bypass domain validation. Strict validation of association files over HTTPS with certificate verification prevents unauthorized app launches and protects user privacy while maintaining legitimate app integration.

Verification:

1. Set up a test website with an apple-app-site-association file for Universal Links
2. Create a link to a URL that matches a Universal Link pattern
3. Verify that the browser validates the apple-app-site-association file before allowing app opening
4. Test that HTTPS is required for Universal Links (HTTP domains should be rejected)
5. Verify that the association file is fetched over HTTPS with certificate validation
6. Test that only domains listed in the association file can trigger the app
7. Attempt to use a malicious association file and verify rejection
8. Test that users can choose to open in app or browser
9. Verify that Universal Link preferences persist across sessions
10. Test that Universal Links respect user privacy (no tracking via link interception)
11. apple-app-site-association file is validated before Universal Link activation
12. HTTPS required for both the association file and triggering URLs
13. Certificate validation enforced when fetching association file
14. Domain validation prevents unauthorized app openings
15. Malformed or malicious association files are rejected
16. User choice between app and browser is preserved
17. User preferences for Universal Links persist
18. No user tracking via Universal Link interception
19. Association file caching follows secure practices
20. Link preview shows correct destination before opening

Pass Criteria: Association file validated over HTTPS AND certificate validation enforced AND user choice respected AND preferences persist

Fail Criteria: No association file validation OR HTTP accepted OR certificate errors ignored OR no user choice OR preferences don't persist

Evidence: Association file validation tests, HTTPS enforcement verification, certificate validation tests, malformed file rejection tests, user choice UI screenshots, preference persistence tests, privacy analysis

References:

- Apple Universal Links Documentation: https://developer.apple.com/ios/universal-links/
- Supporting Associated Domains: https://developer.apple.com/documentation/xcode/supporting-associated-domains
- apple-app-site-association Format: https://developer.apple.com/documentation/bundleresources/applinks
- Universal Links Security: https://developer.apple.com/videos/play/wwdc2019/717/
- Deep Linking Security: https://tools.ietf.org/html/rfc8252
- iOS Security Guide: https://support.apple.com/guide/security/welcome/web

Assessment: PRO-REQ-19 (Deep linking validation)

Reference: PRO-REQ-19 - Browser shall validate deep links before navigation to prevent deep link hijacking and phishing

Given: A conformant browser with PRO-1 or higher capability

Task: Deep links enable direct navigation to specific app content but create attack vectors for parameter injection, XSS payload delivery, privilege escalation, and phishing through misleading app names or domains. Unvalidated deep links can bypass app sandboxes, exfiltrate data through malicious parameters, or launch unverified apps without user awareness. Comprehensive deep link validation with parameter sanitization, domain verification, and user confirmation for unverified apps prevents these attacks while maintaining legitimate app deep linking functionality.

Verification:

1. Create a deep link that targets a mobile app: myapp://user/profile?id=123
2. Verify that the browser validates the deep link format before allowing navigation
3. Test that parameters in deep links are sanitized to prevent injection
4. Attempt to use a deep link with XSS payload and verify sanitization
5. Test that deep links to unverified apps trigger user confirmation
6. Verify that deep link domains are validated against app claims (App Links/Universal Links)
7. Test that deep links cannot bypass app sandbox or permissions
8. Attempt to use deep links for phishing (misleading app names) and verify warnings
9. Verify that HTTPS fallback URLs are used when app is not installed
10. Test that deep link invocation is logged for security monitoring
11. Deep link format and structure validated before processing
12. Parameters in deep links are URL-encoded and sanitized
13. XSS payloads in deep links are neutralized
14. User confirmation required for unverified deep links
15. Domain validation confirms app ownership of deep link domains
16. Deep links cannot escalate privileges or bypass sandbox
17. Phishing detection identifies misleading app names or domains
18. HTTPS fallback works when target app not installed
19. Deep link invocations logged with destination and parameters
20. Rate limiting prevents deep link abuse

Pass Criteria: Deep link format validated AND parameters sanitized AND user confirmation for unverified links AND domain validation enforced AND invocations logged

Fail Criteria: Invalid deep links accepted OR no parameter sanitization OR no confirmation for unverified links OR no domain validation OR not logged

Evidence: Deep link validation tests, parameter sanitization examples, XSS prevention tests, user confirmation screenshots, domain validation verification, phishing detection tests, fallback URL tests, security logs

References:

- Android App Links Verification: https://developer.android.com/training/app-links/verify-site-associations
- iOS Universal Links Validation: https://developer.apple.com/documentation/xcode/supporting-associated-domains
- Deep Link Security: https://tools.ietf.org/html/rfc8252#section-7.11
- OWASP Mobile Deep Link Security: https://owasp.org/www-project-mobile-security-testing-guide/
- Deep Link Phishing Prevention: https://attack.mitre.org/techniques/T1660/
- URL Scheme Best Practices: https://www.rfc-editor.org/rfc/rfc8252#appendix-B

Assessment: PRO-REQ-20 (Protocol handler CSP integration)

Reference: PRO-REQ-20 - Browser shall integrate protocol handlers with Content Security Policy to prevent handler-based attacks

Given: A conformant browser with PRO-1 or higher capability

Task: Protocol handlers can be weaponized to bypass Content Security Policy restrictions if not properly integrated with CSP directives, allowing attackers to navigate to restricted destinations, execute blocked scripts, or submit forms to prohibited targets through custom protocol invocations. Without CSP integration, handlers create a side channel that circumvents navigate-to, form-action, and script-src protections. Enforcing CSP directives on protocol handler operations ensures handlers cannot be used to violate the page's security policy.

Verification:

1. Create a page with a strict CSP that includes navigate-to directive
2. Register a protocol handler from this page
3. Verify that protocol handler invocation respects the navigate-to CSP directive
4. Test that CSP violations during handler registration are reported
5. Create a handler URL template that would violate CSP and verify blocking
6. Test that the script-src directive affects protocol handler registration scripts
7. Verify that form-action CSP directive applies to forms targeting custom protocols
8. Test that connect-src doesn't restrict protocol handler registration but affects handler page
9. Verify that CSP inheritance works correctly for handler pages
10. Test that CSP reports include protocol handler context when violations occur
11. navigate-to CSP directive restricts protocol handler destinations
12. CSP violations during handler operations are properly reported
13. Handler URL templates that violate CSP are rejected
14. script-src controls JavaScript that registers handlers
15. form-action applies to forms with custom protocol actions
16. connect-src applies to handler page, not registration
17. Handler pages inherit appropriate CSP from their origin
18. CSP violation reports include handler-specific context
19. Unsafe-inline and unsafe-eval restrictions apply to handlers
20. CSP nonces and hashes work with handler registration

Pass Criteria: navigate-to restricts handler destinations AND violations reported AND handler URLs validated against CSP AND form-action enforced

Fail Criteria: CSP not enforced for handlers OR violations not reported OR handler URLs bypass CSP OR form-action ignored

Evidence: CSP policy examples with handlers, navigate-to enforcement tests, violation reports with handler context, handler URL validation tests, form-action tests, CSP inheritance verification

References:

- Content Security Policy Level 3: https://www.w3.org/TR/CSP3/
- CSP navigate-to Directive: https://www.w3.org/TR/CSP3/#directive-navigate-to
- CSP form-action Directive: https://www.w3.org/TR/CSP3/#directive-form-action
- CSP Violation Reporting: https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP#violation_reports
- CSP Integration: https://w3c.github.io/webappsec-csp/
- Protocol Handler CSP Considerations: https://www.chromium.org/Home/chromium-security/

Assessment: PRO-REQ-21 (Handler registration audit trail)

Reference: PRO-REQ-21 - Browser shall maintain a complete audit trail of protocol handler registration and modification events

Given: A conformant browser with PRO-2 or higher capability (enterprise mode)

Task: Protocol handler registration and modification events create security-critical audit trails essential for detecting handler-based attacks, insider threats, and policy violations in enterprise environments. Without comprehensive auditing, attackers can register malicious handlers, modify existing handlers, or abuse handler permissions without detection, leaving no forensic evidence for incident response. Tamper-evident audit trails with complete lifecycle tracking enable security teams to detect, investigate, and respond to handler abuse while meeting compliance requirements.

Verification:

1. Enable enterprise audit logging for protocol handlers
2. Register a protocol handler and verify the registration is logged
3. Check that audit log includes: timestamp, origin, scheme, handler URL, user/admin identity
4. Modify handler permissions and verify the change is audited
5. Unregister a handler and verify the removal is audited
6. Test that failed registration attempts are logged with error reasons
7. Verify that permission grants/denials for handlers are audited
8. Test that enterprise policy changes affecting handlers are logged
9. Export the audit trail and verify it's in a tamper-evident format
10. Verify that audit logs can be forwarded to enterprise SIEM systems
11. Complete audit trail for all handler lifecycle events
12. Audit entries include: timestamp, identity, action, object, before/after state, outcome
13. Failed operations logged with error details
14. Permission changes audited with user decision
15. Enterprise policy enforcement events included
16. Audit logs are tamper-evident (signed or chained hashes)
17. Logs exportable in standard formats (JSON, CEF, syslog)
18. SIEM integration supported for centralized logging
19. Audit trail completeness can be verified
20. Log retention aligns with compliance requirements

Pass Criteria: All handler events audited with complete metadata AND failed attempts included AND logs tamper-evident AND SIEM integration supported

Fail Criteria: Incomplete audit trail OR missing metadata OR logs can be tampered OR no SIEM integration

Evidence: Audit log exports, log completeness analysis, tamper-evidence verification, SIEM integration examples, failed operation logs, enterprise policy audit examples, retention policy documentation

References:

- NIST SP 800-53 Audit and Accountability: https://csrc.nist.gov/publications/detail/sp/800-53/rev-5/final
- ISO 27001 Audit Trail Requirements: https://www.iso.org/standard/54534.html
- Common Event Format (CEF): https://www.microfocus.com/documentation/arcsight/arcsight-smartconnectors-8.3/cef-implementation-standard/
- Syslog Protocol RFC 5424: https://www.rfc-editor.org/rfc/rfc5424
- CIS Audit Log Management: https://www.cisecurity.org/controls/

Assessment: PRO-REQ-22 (Protocol handler update security)

Reference: PRO-REQ-22 - Browser shall securely handle updates to protocol handler registrations and prevent malicious modifications

Given: A conformant browser with PRO-1 or higher capability

Task: Protocol handler updates create opportunities for malicious modification attacks where attackers hijack existing trusted handlers by changing their URL templates to point to attacker-controlled destinations, dangerous URL schemes, or downgraded HTTP endpoints. Cross-origin handler updates could allow one domain to subvert another's handlers, while unvalidated PWA manifest updates could silently redirect protocol traffic. Secure update mechanisms with same-origin enforcement, dangerous scheme blocking, and downgrade protection prevent handler hijacking while maintaining legitimate update functionality.

Verification:

1. Register a protocol handler from https://trusted.example.com
2. Attempt to update the handler URL template from the same origin
3. Verify that the update requires user confirmation or follows original permissions
4. Attempt to update the handler from a different origin (https://attacker.example.com)
5. Verify that cross-origin updates are prevented
6. Test that handler updates during active use trigger security warnings
7. Verify that automatic handler updates (e.g., PWA manifest updates) are validated
8. Test that handler URL changes are logged in audit trail
9. Attempt to update handler to point to data: or javascript: URLs and verify blocking
10. Verify that downgrade protection applies to handler updates
11. Handler updates require same-origin or user permission
12. Cross-origin handler modification is prevented
13. Active handler updates trigger user notification
14. PWA manifest handler updates follow secure update process
15. Handler URL changes are audited with before/after values
16. Updates to dangerous URL schemes (data:, javascript:) are blocked
17. HTTPS-to-HTTP downgrades in handler URLs are prevented
18. Update frequency is rate-limited to prevent abuse
19. Failed update attempts are logged
20. Users can review and approve pending handler updates

Pass Criteria: Same-origin update restriction enforced AND cross-origin updates prevented AND dangerous schemes blocked AND updates audited

Fail Criteria: Cross-origin updates allowed OR dangerous schemes accepted OR no audit trail OR downgrade attacks possible

Evidence: Update test results showing same-origin enforcement, cross-origin prevention tests, dangerous scheme blocking verification, audit log entries for updates, downgrade prevention tests, PWA manifest update security verification

References:

- WHATWG HTML Handler Updates: https://html.spec.whatwg.org/multipage/system-state.html#custom-handlers
- Web App Manifest Updates: https://www.w3.org/TR/appmanifest/#updating
- Secure Software Updates: https://www.rfc-editor.org/rfc/rfc8240
- OWASP Secure Update Guidelines: https://cheatsheetseries.owasp.org/cheatsheets/Vulnerable_Dependency_Manageme
- Same-Origin Policy for Updates: https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy

Assessment: PRO-REQ-23 (Handler isolation enforcement)

Reference: PRO-REQ-23 - Browser shall enforce process isolation for protocol handlers to limit impact of handler compromise

Given: A conformant browser with PRO-2 or higher capability

Task: Protocol handlers execute in the browser context with potential access to sensitive user data and system resources, making them high-value targets for exploitation. Without process isolation, a compromised handler could access data from other origins, bypass sandbox restrictions, read cross-process memory through Spectre-type attacks, or crash the entire browser. Handler isolation through Site Isolation architecture with sandboxing, IPC validation, and Spectre mitigations contains the impact of handler compromise and prevents privilege escalation attacks.

Verification:

1. Register and invoke a protocol handler
2. Use browser internals (chrome://process-internals or equivalent) to verify handler process isolation
3. Verify that handler pages run in separate renderer processes from the triggering page
4. Test that compromised handler cannot access other origin's data
5. Verify that Site Isolation applies to protocol handler pages
6. Test that handler processes have appropriate sandbox restrictions
7. Verify that IPC between handler and browser process is validated
8. Test that handler crash doesn't affect other tabs or browser stability
9. Verify that handler processes don't have elevated privileges
10. Test that memory isolation prevents handler from reading other process memory
11. Protocol handlers run in isolated renderer processes
12. Handler isolation follows Site Isolation architecture
13. Cross-origin data is not accessible from handler process
14. Handler processes are sandboxed with restricted capabilities
15. IPC messages from handlers are validated and sanitized
16. Handler crashes are isolated and don't affect browser stability
17. No elevated privileges granted to handler processes
18. Memory isolation prevents cross-process memory access
19. Spectre/Meltdown mitigations apply to handler processes
20. Process limit enforced to prevent resource exhaustion

Pass Criteria: Handler process isolation enforced AND Site Isolation applied AND sandboxing active AND IPC validated AND crashes isolated

Fail Criteria: Handlers in same process as other content OR no sandboxing OR IPC not validated OR crashes affect browser OR cross-process memory access possible

Evidence: Process isolation verification using browser internals, Site Isolation tests, sandbox capability analysis, IPC validation tests, crash isolation tests, memory isolation verification, Spectre mitigation confirmation

References:

- Chromium Site Isolation: https://www.chromium.org/Home/chromium-security/site-isolation/

- Browser Process Architecture: https://www.chromium.org/developers/design-documents/multi-process-architecture/
- Sandbox Architecture: https://chromium.googlesource.com/chromium/src/+/master/docs/design/sandbox.md
- IPC Security: https://www.chromium.org/developers/design-documents/inter-process-communication/
- Spectre Mitigations: https://www.chromium.org/Home/chromium-security/ssca/
- Process Isolation Best Practices: https://www.w3.org/TR/security-privacy-questionnaire/#isolation

6.7 System Resource Access Security Assessments

This section covers assessment procedures for requirements SYS-REQ-1 through SYS-REQ-32, addressing sandbox enforcement, Hardware Abstraction Layer (HAL) security, PWA permissions, filesystem access, device API security, and system resource isolation.

Assessment: SYS-REQ-1 (Process sandbox enforcement)

Reference: SYS-REQ-1 - Browser shall enforce process-level sandboxing to isolate web content from the host operating system

Given: A conformant browser with SYS-1 or higher capability

Task: Renderer processes handle untrusted web content and are the primary attack surface for browser exploitation. Without OS-level sandboxing, successful renderer exploits grant attackers full system access to read files, execute processes, exfiltrate data, and install malware. Process-level sandboxing using platform-specific mechanisms (seccomp-bpf, sandbox-exec, Job Objects) with syscall filtering and privilege reduction contains renderer compromises, preventing sandbox escapes and limiting attack impact to the isolated process.

Verification:

1. Launch the browser with sandbox debugging enabled (–enable-sandbox-logging or equivalent)
2. Open the browser's internal process viewer (chrome://sandbox, about:support, or Task Manager)
3. Navigate to a test web page and verify it runs in a sandboxed renderer process
4. Attempt to execute system calls from JavaScript that should be blocked (file system access, process creation)
5. Use platform-specific tools to inspect process sandbox status (Process Explorer on Windows, ps with security flags on Linux/macOS)
6. Verify renderer processes run with reduced privileges using tools like icacls, getfacl, or sandbox-exec -p
7. Test that sandboxed processes cannot access parent process memory
8. Attempt to escape sandbox through known attack vectors and verify containment
9. Monitor system calls using strace (Linux), dtruss (macOS), or Process Monitor (Windows) to verify syscall filtering
10. Verify that sandbox violations are logged and processes are terminated on policy violations
11. All renderer processes execute within OS-level sandbox (seccomp-bpf on Linux, sandbox-exec on macOS, Job Objects on Windows)
12. Sandboxed processes cannot access filesystem outside designated cache directories
13. System call filtering is active and blocks dangerous syscalls (execve, fork, ptrace)
14. Process privileges are reduced (no root, limited capabilities, restricted tokens)
15. Memory isolation prevents cross-process memory access
16. Network access is mediated through broker process
17. Sandbox escape attempts are blocked and logged
18. Process termination occurs on sandbox policy violations

Pass Criteria: All renderer processes execute in OS-level sandbox AND dangerous system calls are filtered AND privilege reduction is verified AND sandbox escapes are prevented

Fail Criteria: Any renderer process runs without sandbox OR system calls are not filtered OR privileges are not reduced OR sandbox escape succeeds

Evidence: Process sandbox status screenshots, syscall trace logs showing filtering, privilege analysis outputs (icacls, capabilities), sandbox violation logs, security tool reports (Process Explorer, sandbox-exec output)

References:

- Chromium Sandbox Design: https://chromium.googlesource.com/chromium/src/+/master/docs/design/sandbox.md
- Linux seccomp-bpf: https://www.kernel.org/doc/html/latest/userspace-api/seccomp_filter.html
- macOS Sandbox: https://developer.apple.com/documentation/security/app_sandbox
- Windows Sandbox: https://docs.microsoft.com/en-us/windows/security/threat-protection/windows-sandbox/windows-sandbox-overview

Assessment: SYS-REQ-2 (Renderer process isolation)

Reference: SYS-REQ-2 - Browser shall isolate renderer processes from each other and from browser core processes

Given: A conformant browser with SYS-1 or higher capability

Task: Renderer process isolation is fundamental to browser security architecture, preventing compromised renderers from accessing data belonging to other origins. Without process-per-origin isolation, a successful exploit in one tab could steal credentials, session tokens, and sensitive data from all other open tabs, violating Same-Origin Policy at the process level. Site Isolation with distinct processes, mediated IPC, and no shared memory prevents cross-origin data theft, Spectre attacks, and cascading process crashes.

Verification:

1. Open multiple tabs with different origins in the browser
2. Use the browser's process viewer to verify each origin runs in a separate renderer process
3. Open developer tools and use performance profiling to identify process boundaries
4. Test Site Isolation by navigating to cross-origin iframes and verifying separate processes
5. Attempt to access memory or data from one renderer process in another using side-channel attacks
6. Verify that process IDs are distinct for different origins using OS tools (ps, Task Manager)
7. Test that renderer crashes in one tab do not affect other tabs or the browser process
8. Monitor inter-process communication to verify it goes through secure IPC channels
9. Use memory analysis tools to verify no shared memory regions between renderers
10. Test process-per-site-instance isolation for enhanced security
11. Each origin or site instance runs in a dedicated renderer process
12. Process IDs are distinct and verifiable through OS tools
13. Renderer process crashes are isolated and do not cascade
14. No shared memory regions exist between different renderer processes
15. Inter-process communication uses secure, mediated IPC channels
16. Browser core process (broker) is isolated from all renderers
17. GPU process isolation is separate from renderer isolation
18. Side-channel attacks cannot leak data between renderer processes

Pass Criteria: Different origins run in separate processes AND processes have distinct PIDs AND crashes are isolated AND no memory sharing exists

Fail Criteria: Same process handles multiple origins OR process crash cascades OR shared memory exists OR IPC is not secured

Evidence: Process viewer screenshots showing multiple renderer processes, PID listings from OS tools, crash isolation test results, memory map analysis, IPC traffic logs, Site Isolation verification reports

References:

- Chromium Site Isolation: https://www.chromium.org/Home/chromium-security/site-isolation/
- Firefox Fission: https://wiki.mozilla.org/Project_Fission

Assessment: SYS-REQ-3 (GPU process isolation)

Reference: SYS-REQ-3 - Browser shall isolate GPU rendering operations in a separate sandboxed process

Given: A conformant browser with SYS-1 or higher capability

Task: GPU processes execute untrusted shader code and interact with complex graphics drivers that have historically been sources of vulnerabilities. Without GPU process isolation, exploits targeting graphics drivers or shader compilers could escape to access the filesystem, network, or other process memory, bypassing renderer sandbox protections. Isolated GPU processes with command buffer validation and sandboxing contain GPU-related exploits while enabling graceful degradation through software rendering fallbacks.

Verification:

1. Launch browser and navigate to chrome://gpu or about:support to verify GPU process information
2. Open a WebGL-intensive page (e.g., https://webglsamples.org/) and verify GPU process activation
3. Use OS process viewer to identify the GPU process and verify it's distinct from renderers
4. Check GPU process sandbox status using platform-specific security tools
5. Verify GPU process has limited capabilities and cannot access filesystem directly
6. Test that GPU process crashes do not terminate the browser or renderer processes
7. Monitor GPU command buffer submissions to verify they're sanitized and validated
8. Attempt to exploit GPU driver vulnerabilities and verify sandbox containment
9. Use graphics debugging tools (apitrace, RenderDoc) to analyze GPU process isolation
10. Verify that software rendering fallback maintains process isolation
11. GPU process runs as separate, distinct process with unique PID
12. GPU process executes within OS-level sandbox with reduced privileges
13. GPU command buffers are validated before submission to driver
14. GPU process cannot directly access filesystem or network
15. Crashes in GPU process trigger graceful degradation (software rendering)
16. Graphics driver access is mediated and monitored
17. Shader compilation occurs in isolated context
18. GPU memory is isolated from CPU-accessible memory

Pass Criteria: GPU process is isolated with distinct PID AND sandbox is enforced AND command validation occurs AND crashes are contained

Fail Criteria: No GPU process isolation OR sandbox not enforced OR commands not validated OR crashes cascade

Evidence: GPU process information screenshots, PID verification, sandbox status reports, crash test results, GPU command trace logs, shader compilation logs, graphics debugging tool outputs

References:

- Chromium GPU Process Architecture: https://www.chromium.org/developers/design-documents/gpu-accelerated-compositing-in-chrome/
- GPU Sandbox: https://chromium.googlesource.com/chromium/src/+/master/docs/design/sandbox.md#gpu-process
- WebGL Security: https://www.khronos.org/registry/webgl/specs/latest/1.0/#security
- Angle Project Security: https://chromium.googlesource.com/angle/angle
- GPU Denylist and Security: https://chromium.googlesource.com/chromium/src/+/master/gpu/config/software_renderin

Assessment: SYS-REQ-4 (Network service isolation)

Reference: SYS-REQ-4 - Browser shall isolate network operations in a separate sandboxed process or service

Given: A conformant browser with SYS-1 or higher capability

Task: Network operations in renderer processes create attack vectors for certificate validation bypass, CORS violations, and direct socket access that could enable data exfiltration or network-based attacks. Without

network service isolation, compromised renderers could directly manipulate network connections, bypass security policies, or exploit network stack vulnerabilities. Isolating network operations in a separate service with mediated access ensures certificate validation, CORS enforcement, and CSP compliance occur in a privileged, monitored context outside attacker control.

Verification:

1. Open browser internal pages to view process architecture (chrome://process-internals)
2. Verify network service runs as separate process or is isolated within browser process
3. Use network analysis tools (Wireshark, tcpdump) to monitor network requests from different processes
4. Test that renderer processes cannot directly create network sockets
5. Verify all network requests are mediated through network service/process
6. Attempt to bypass network service from renderer process and verify blocking
7. Monitor network service sandbox status using platform security tools
8. Test certificate validation occurs in network service, not renderer
9. Verify CORS and CSP enforcement happens in network service layer
10. Test that network service crashes trigger appropriate error handling
11. Network operations execute in isolated network service/process
12. Renderer processes cannot directly access network APIs
13. All network requests are mediated through network service
14. Certificate validation occurs in privileged context
15. CORS and content security policies are enforced at network layer
16. Network service runs with minimal necessary privileges
17. Socket creation is controlled and monitored
18. Network service crashes are handled gracefully

Pass Criteria: Network service is isolated AND renderers use IPC for network access AND certificate validation is isolated AND CORS/CSP enforced at network layer

Fail Criteria: Renderers have direct network access OR no network service isolation OR certificate validation in renderer OR enforcement bypassed

Evidence: Process architecture diagrams, network traffic captures, IPC logs showing network requests, sandbox status for network service, certificate validation traces, CORS enforcement logs

References:

- Chromium Network Service: https://www.chromium.org/developers/design-documents/network-stack/
- Network Sandbox: https://chromium.googlesource.com/chromium/src/+/master/services/network/README.md
- CORS and Fetch Standard: https://fetch.spec.whatwg.org/
- Certificate Transparency: https://www.certificate-transparency.org/
- Mozilla Network Security: https://wiki.mozilla.org/Security/Server_Side_TLS

Assessment: SYS-REQ-5 (Filesystem access control)

Reference: SYS-REQ-5 - Browser shall enforce strict access controls on filesystem operations, limiting access to user-approved locations

Given: A conformant browser with SYS-1 or higher capability

Task: Unrestricted filesystem access from web content enables attackers to read sensitive files, access system directories, exfiltrate browser credentials from profile directories, or write malicious files to startup locations. Without strict access controls, file:// URLs could read arbitrary local files, and File APIs could access system directories without user awareness. User-mediated filesystem access with sandboxed namespaces, IPC-brokered operations, and revocable per-origin permissions prevents unauthorized file access while enabling legitimate file operations.

Verification:

1. Attempt to read local files using file:// URLs and verify restrictions

2. Test File API access from web content and verify it requires user gesture
3. Use File System Access API to request directory access and verify user prompt appears
4. Monitor filesystem access from renderer process using system tools (auditd, OpenBSM, Process Monitor)
5. Verify browser cache and profile directories are protected from direct renderer access
6. Test that sandboxed filesystem namespace limits visible paths
7. Attempt to access system directories (/etc, C:\Windows) from web content and verify blocking
8. Verify file uploads use secure IPC to broker process for filesystem access
9. Test that downloaded files are stored in user-designated locations only
10. Check that filesystem access permissions are revoked when tab closes
11. file:// URL access is restricted or requires user opt-in
12. File API requires user gesture (click/tap) for access
13. File System Access API shows permission prompts
14. Renderer processes cannot directly access filesystem
15. System directories are not accessible from web content
16. Browser profile and cache protected from renderer access
17. File operations use IPC to privileged broker process
18. Filesystem permissions are per-origin and revocable
19. Sandboxed filesystem namespace limits path visibility

Pass Criteria: Filesystem access requires user permission AND system directories are blocked AND renderer uses IPC for file operations AND permissions are revocable

Fail Criteria: Direct filesystem access from renderer OR no user permission required OR system directories accessible OR permissions not revocable

Evidence: File access audit logs, permission prompt screenshots, filesystem monitoring traces, sandbox policy dumps, IPC logs for file operations, directory access test results

References:

- File System Access API: https://wicg.github.io/file-system-access/
- File API Specification: https://www.w3.org/TR/FileAPI/
- OWASP File Upload Security: https://cheatsheetseries.owasp.org/cheatsheets/File_Upload_Cheat_Sheet.html
- Same-Origin Policy for file: URLs: https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy#file_origins

Assessment: SYS-REQ-6 (Device API permissions)

Reference: SYS-REQ-6 - Browser shall implement permission controls for all device hardware access APIs

Given: A conformant browser with SYS-1 or higher capability

Task: Device hardware APIs provide access to sensitive capabilities like cameras, microphones, sensors, and location data that can be abused for surveillance, data theft, or privacy violations. Without permission controls, malicious websites could silently activate cameras for spying, record audio, or track user location. Per-origin permission prompts with explicit user consent, revocability, and cross-origin isolation prevent unauthorized device access while enabling legitimate functionality for trusted origins.

Verification:

1. Navigate to test page that requests camera access using navigator.mediaDevices.getUserMedia()
2. Verify permission prompt appears and requires explicit user action
3. Test microphone access and verify separate permission prompt
4. Check permission settings in browser UI (chrome://settings/content, about:preferences#privacy)
5. Revoke camera permission and verify future access is blocked
6. Test permission persistence across browser restarts
7. Verify permissions are per-origin and not shared across origins
8. Test permission inheritance in cross-origin iframes (should be blocked)
9. Attempt to access device without permission and verify SecurityError thrown

10. Verify permissions can be permanently denied by user
11. All device API access triggers permission prompts
12. User shall explicitly grant permission (no auto-grant)
13. Permissions are origin-scoped and isolated
14. Cross-origin iframe access is blocked by default
15. Permission state is persistent and survives restarts
16. Users can revoke permissions at any time
17. Denied permissions throw appropriate errors
18. Permission prompts include clear device/API information
19. Temporary permission grants expire appropriately

Pass Criteria: Device access requires explicit permission AND prompts are clear AND permissions are per-origin AND revocation works

Fail Criteria: Device access without permission OR auto-grant occurs OR permissions not per-origin OR revocation doesn't work

Evidence: Permission prompt screenshots, settings UI showing permissions, console logs of SecurityErrors, cross-origin test results, permission persistence tests, revocation verification

References:

- Permissions API: https://www.w3.org/TR/permissions/
- Media Capture and Streams: https://www.w3.org/TR/mediacapture-streams/
- Permission Delegation: https://www.w3.org/TR/permissions-policy-1/
- MDN Permissions: https://developer.mozilla.org/en-US/docs/Web/API/Permissions_API

Assessment: SYS-REQ-7 (PWA permission management)

Reference: SYS-REQ-7 - Browser shall enforce equivalent permission controls for Progressive Web Apps as for regular web content

Given: A conformant browser with PWA-1 and SYS-1 or higher capability

Task: Progressive Web Apps installed as standalone applications may appear more trustworthy to users, creating opportunities for permission abuse if PWAs receive elevated privileges compared to web contexts. Auto-granting permissions during PWA installation would bypass informed consent, while allowing service workers to circumvent permission checks enables background surveillance. Enforcing equivalent permission controls for PWAs as web content prevents privilege escalation through installation while ensuring permission revocation upon uninstallation.

Verification:

1. Install a test PWA with manifest requesting various permissions
2. Verify that PWA installation does not auto-grant permissions
3. Launch PWA and trigger permission requests (camera, location, notifications)
4. Verify permission prompts appear identical to browser context
5. Check that PWA permissions are isolated per origin in browser settings
6. Test that uninstalling PWA revokes all granted permissions
7. Verify PWA cannot request more permissions than web context
8. Test permission state is synchronized between PWA and browser views of same origin
9. Attempt to bypass permission via service worker and verify blocking
10. Verify PWA display mode (standalone, fullscreen) does not affect permission requirements
11. PWA installation does not auto-grant permissions
12. Permission prompts appear for all sensitive APIs
13. Permissions are per-origin, shared with web context
14. Uninstalling PWA revokes granted permissions
15. Service workers cannot bypass permission checks
16. Display mode does not affect permission requirements

17. PWA permissions visible in browser settings
18. Permission state synchronized across contexts

Pass Criteria: PWA permissions equal to web permissions AND no auto-grant on install AND uninstall revokes permissions AND service workers respect permissions

Fail Criteria: PWA gets extra permissions OR auto-grant on install OR uninstall doesn't revoke OR service worker bypass

Evidence: PWA installation flow screenshots, permission prompt comparisons, settings showing PWA permissions, uninstall verification tests, service worker permission logs, display mode test results

References:

- Web App Manifest: https://www.w3.org/TR/appmanifest/
- PWA Permissions: https://web.dev/articles/install-criteria
- Service Worker Security: https://www.w3.org/TR/service-workers/#security-considerations
- Permissions Policy in PWAs: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Permissions-Policy

Assessment: SYS-REQ-8 (Geolocation permission enforcement)

Reference: SYS-REQ-8 - Browser shall enforce user permission requirements for geolocation API access

Given: A conformant browser with SYS-1 or higher capability

Task: Geolocation APIs expose precise user location data that enables physical tracking, stalking, burglary planning, and profiling of user movements and routines. Without HTTPS requirements and permission controls, attackers on insecure connections could intercept location data, while malicious sites could track users without consent. HTTPS enforcement, per-origin permission prompts, immediate revocation, and cross-origin isolation prevent unauthorized location tracking while enabling legitimate location-based services.

Verification:

1. Navigate to test page that calls navigator.geolocation.getCurrentPosition()
2. Verify permission prompt appears before any location data is returned
3. Test that HTTPS context is required for geolocation (HTTP should fail)
4. Grant permission and verify location data is returned
5. Revoke permission and verify subsequent calls are denied
6. Test high-accuracy mode requires explicit permission
7. Verify watchPosition() respects same permission model
8. Test that cross-origin iframes require permission policy delegation
9. Attempt geolocation access without user gesture and verify it still requires permission
10. Verify location permission can be set to "ask every time"
11. Geolocation API requires HTTPS context (except localhost)
12. Permission prompt appears before any location data access
13. Permission is per-origin and persistent
14. High-accuracy mode requires explicit permission
15. Cross-origin access blocked without delegation
16. Revoked permissions immediately block access
17. Users can set "ask every time" preference
18. Permission state is accessible via Permissions API

Pass Criteria: HTTPS required AND permission prompt appears AND per-origin isolation AND revocation works AND cross-origin blocked

Fail Criteria: HTTP allows access OR no permission prompt OR not per-origin OR revocation doesn't work OR cross-origin allowed

Evidence: Permission prompt screenshots, HTTPS requirement test results, permission settings showing geolocation, cross-origin test logs, revocation verification, console errors for denied access

References:

- Geolocation API: https://www.w3.org/TR/geolocation-API/
- Geolocation Security: https://w3c.github.io/geolocation-api/#security
- Secure Contexts: https://www.w3.org/TR/secure-contexts/
- MDN Geolocation: https://developer.mozilla.org/en-US/docs/Web/API/Geolocation_API

Assessment: SYS-REQ-9 (Camera/microphone access control)

Reference: SYS-REQ-9 - Browser shall enforce strict permission controls for camera and microphone access with user-visible indicators

Given: A conformant browser with SYS-1 or higher capability

Task: Camera and microphone access enables covert surveillance, recording private conversations, capturing sensitive visual information, and violating user privacy. Without visible indicators, malicious sites could secretly record audio/video for blackmail, espionage, or data theft. Permission prompts with device selection, persistent active-use indicators, immediate mid-stream revocation, and cross-origin blocking prevent unauthorized surveillance while providing user transparency and control over their devices.

Verification:

1. Navigate to test page that requests camera access via getUserMedia({video: true})
2. Verify permission prompt appears with device selection options
3. Grant permission and verify camera indicator appears in browser UI (red dot, icon)
4. Test microphone access separately and verify distinct permission prompt
5. Request both camera and microphone and verify single combined prompt
6. Verify active use indicators remain visible while devices are active
7. Test that stopping media stream removes indicators
8. Verify users can revoke permission mid-stream and devices immediately stop
9. Test that cross-origin iframes cannot inherit camera/microphone permissions
10. Verify permission prompts identify requesting origin clearly
11. Separate permission prompts for camera and microphone
12. Device selection available in permission prompt
13. Visual indicators appear when camera/microphone active
14. Indicators remain visible for entire use duration
15. Stopping stream immediately removes indicators
16. Mid-stream revocation immediately stops device access
17. Cross-origin iframe access blocked without delegation
18. Permission prompts clearly show requesting origin
19. Users can select specific device or deny access

Pass Criteria: Permission prompts appear AND active-use indicators visible AND mid-stream revocation works AND cross-origin blocked

Fail Criteria: No permission prompt OR no indicators OR revocation doesn't stop devices OR cross-origin allowed

Evidence: Permission prompt screenshots, active camera/microphone indicator screenshots, device selection UI, cross-origin test results, mid-stream revocation tests, origin display verification

References:

- Media Capture and Streams: https://www.w3.org/TR/mediacapture-streams/
- getUserMedia Security: https://w3c.github.io/mediacapture-main/#security-and-permissions
- Firefox Camera Privacy: https://support.mozilla.org/en-US/kb/how-manage-your-camera-and-microphone-permissions

Assessment: SYS-REQ-10 (Clipboard access restrictions)

Reference: SYS-REQ-10 - Browser shall restrict clipboard access to require user interaction or explicit permission

Given: A conformant browser with SYS-1 or higher capability

Task: Clipboard access enables theft of sensitive data like passwords, credit card numbers, authentication tokens, and private communications that users copy. Unrestricted clipboard reading allows malicious sites to silently exfiltrate clipboard contents, while background clipboard access enables persistent monitoring. User gesture requirements for writing, permission prompts for reading, background access blocking, and cross-origin restrictions prevent clipboard-based data theft while enabling legitimate copy/paste functionality.

Verification:

1. Test document.execCommand('copy') and verify it requires user gesture
2. Attempt clipboard write without user gesture and verify it's blocked
3. Test Async Clipboard API (navigator.clipboard.writeText()) and verify permission model
4. Attempt clipboard read using navigator.clipboard.readText() and verify permission prompt
5. Test clipboard access in background tab and verify it's blocked
6. Verify cross-origin iframe clipboard access requires permission policy
7. Test that clipboard events (copy, cut, paste) are only triggered by user actions
8. Verify sensitive data types (images, rich text) require explicit permission
9. Test that clipboard access from service workers is restricted
10. Verify clipboard history is not accessible without permission
11. Legacy clipboard API requires user gesture
12. Async Clipboard API requires permission for reading
13. Background clipboard access is blocked
14. Cross-origin access requires permission policy delegation
15. Clipboard events only fire from user-initiated actions
16. Sensitive data types require explicit permission
17. Service worker clipboard access is restricted
18. No access to clipboard history without permission
19. Permission prompts are clear about clipboard access

Pass Criteria: User gesture required for write AND permission required for read AND background access blocked AND cross-origin requires delegation

Fail Criteria: Write without gesture OR read without permission OR background access allowed OR cross-origin not restricted

Evidence: Clipboard permission prompt screenshots, console logs showing blocked access, user gesture test results, cross-origin test logs, background tab test results, service worker restriction verification

References:

- Clipboard API: https://www.w3.org/TR/clipboard-apis/
- Async Clipboard API: https://w3c.github.io/clipboard-apis/
- Clipboard Security Model: https://w3c.github.io/clipboard-apis/#security
- MDN Clipboard API: https://developer.mozilla.org/en-US/docs/Web/API/Clipboard_API

Assessment: SYS-REQ-11 (Notification permission management)

Reference: SYS-REQ-11 - Browser shall enforce permission controls for web notifications with user-visible prompts

Given: A conformant browser with SYS-1 or higher capability

Task: Web notifications enable persistent user engagement but create vectors for notification spam, phishing through fake system alerts, social engineering attacks via deceptive messages, and user annoyance leading

to permission fatigue. Without permission controls, malicious sites could bombard users with unwanted notifications or craft convincing fake alerts mimicking system messages. User gesture requirements, permission prompts, per-origin isolation, and service worker permission enforcement prevent notification abuse while enabling legitimate push messaging.

Verification:

1. Test Notification.requestPermission() and verify user prompt appears
2. Verify notification requests require user gesture (click/tap)
3. Grant permission and test notification display using new Notification()
4. Verify notifications from different origins are isolated
5. Test notification permission revocation and verify no more notifications appear
6. Verify service worker notifications respect same permission model
7. Test that cross-origin iframes cannot inherit notification permission
8. Verify permission state is accessible via Notification.permission
9. Test notification action buttons and verify they maintain security context
10. Verify silent notifications (without sound/vibration) still require permission
11. Notification permission requires explicit user grant
12. Permission prompt appears before any notification shown
13. User gesture required to trigger permission prompt
14. Permissions are per-origin and isolated
15. Service worker notifications use same permission
16. Cross-origin iframe access blocked without delegation
17. Permission revocation immediately prevents notifications
18. Notification.permission accurately reflects state
19. Action buttons maintain security context
20. All notification types require permission

Pass Criteria: Permission prompt required AND user gesture needed AND per-origin isolation AND service workers respect permissions

Fail Criteria: No permission prompt OR no user gesture required OR not per-origin OR service worker bypass

Evidence: Permission prompt screenshots, user gesture requirement tests, service worker notification tests, cross-origin test results, revocation verification, notification display examples

References:

- Notifications API: https://notifications.spec.whatwg.org/
- Notification Security: https://notifications.spec.whatwg.org/#security-and-privacy
- Push API: https://www.w3.org/TR/push-api/
- Service Worker Notifications: https://web.dev/articles/push-notifications-overview
- Chrome Notifications: https://developer.chrome.com/docs/extensions/reference/notifications/

Assessment: SYS-REQ-12 (USB device access security)

Reference: SYS-REQ-12 - Browser shall enforce strict permission and security controls for WebUSB device access

Given: A conformant browser with SYS-1 or higher capability and WebUSB support

Task: WebUSB provides direct hardware access to USB devices, creating risks of firmware attacks, data exfiltration through storage devices, keystroke logging via HID devices, and unauthorized control of sensitive peripherals. Without restrictions, malicious sites could access mass storage to read private files, reprogram device firmware, or communicate with security keys to bypass authentication. HTTPS requirements, device picker prompts, dangerous class filtering, and per-device permissions prevent USB-based attacks while enabling legitimate device interaction.

Verification:

1. Navigate to test page that calls navigator.usb.requestDevice()
2. Verify permission prompt appears with device picker showing available USB devices
3. Test that HTTPS context is required for WebUSB (HTTP should fail)
4. Grant access to specific USB device and verify connection succeeds
5. Verify that only explicitly granted device is accessible
6. Test device access from cross-origin iframe and verify it's blocked
7. Attempt to access USB device without user gesture and verify it's blocked
8. Revoke USB permission and verify device access is immediately blocked
9. Test that dangerous device classes (HID, mass storage) are filtered from device picker
10. Verify device disconnect/reconnect requires re-authorization if permission was revoked
11. WebUSB requires HTTPS context (except localhost)
12. Permission prompt shows device picker with clear device identification
13. Only explicitly selected devices are accessible
14. User gesture required to trigger device selection
15. Cross-origin access blocked without permission policy
16. Dangerous device classes (HID, storage) are not available
17. Permission revocation immediately blocks device access
18. Device access is per-origin and isolated
19. Device picker shows only appropriate devices
20. Reconnected devices respect permission state

Pass Criteria: HTTPS required AND device picker shown AND only selected devices accessible AND dangerous classes blocked

Fail Criteria: HTTP allows access OR no device picker OR all devices accessible OR dangerous classes available

Evidence: WebUSB permission prompt screenshots, device picker UI, HTTPS requirement tests, dangerous device class filtering tests, cross-origin test results, revocation verification

References:

- WebUSB API: https://wicg.github.io/webusb/
- WebUSB Security: https://wicg.github.io/webusb/#security-and-privacy
- USB Device Class Codes: https://www.usb.org/defined-class-codes
- Chrome WebUSB: https://developer.chrome.com/articles/build-for-webusb/

Assessment: SYS-REQ-13 (Bluetooth permission enforcement)

Reference: SYS-REQ-13 - Browser shall enforce permission controls and security restrictions for Web Bluetooth API

Given: A conformant browser with SYS-1 or higher capability and Web Bluetooth support

Task: Web Bluetooth enables wireless communication with Bluetooth devices, creating risks of unauthorized pairing with sensitive peripherals, GATT service exploitation to extract data or modify device settings, and attacks on Bluetooth-enabled security devices or medical equipment. Without controls, malicious sites could pair with fitness trackers to steal health data, connect to Bluetooth keyboards to log keystrokes, or interact with dangerous device types. HTTPS requirements, device picker prompts, service UUID filtering, and blocklist enforcement prevent Bluetooth-based attacks.

Verification:

1. Navigate to test page that calls navigator.bluetooth.requestDevice()
2. Verify permission prompt appears with Bluetooth device picker
3. Test that HTTPS context is required for Web Bluetooth (HTTP should fail)
4. Grant access to specific Bluetooth device and verify GATT connection

5. Verify only explicitly granted device is accessible
6. Test service UUID filtering in device picker
7. Attempt Bluetooth access without user gesture and verify blocking
8. Test cross-origin iframe access and verify it's blocked
9. Revoke Bluetooth permission and verify device access is blocked
10. Verify Bluetooth blocklist prevents access to dangerous device types
11. Web Bluetooth requires HTTPS context (except localhost)
12. Permission prompt shows Bluetooth device picker
13. Only explicitly selected devices are accessible
14. Service UUID filtering works correctly
15. User gesture required to trigger device selection
16. Cross-origin access blocked without delegation
17. Dangerous device types blocked by blocklist
18. Permission revocation immediately blocks access
19. Device access is per-origin and isolated
20. GATT operations respect permission boundaries

Pass Criteria: HTTPS required AND device picker shown AND only selected devices accessible AND blocklist enforced

Fail Criteria: HTTP allows access OR no device picker OR all devices accessible OR blocklist not enforced

Evidence: Bluetooth permission prompt screenshots, device picker UI, service UUID filtering tests, HTTPS requirement verification, blocklist enforcement tests, cross-origin test results

References:

- Web Bluetooth API: https://webbluetoothcg.github.io/web-bluetooth/
- Web Bluetooth Security: https://webbluetoothcg.github.io/web-bluetooth/#security-and-privacy
- Bluetooth GATT Services: https://www.bluetooth.com/specifications/gatt/
- Chrome Web Bluetooth: https://developer.chrome.com/articles/bluetooth/
- Web Bluetooth Blocklist: https://github.com/WebBluetoothCG/registries/blob/master/gatt_blocklist.txt

Assessment: SYS-REQ-14 (File System Access API security)

Reference: SYS-REQ-14 - Browser shall enforce strict security controls for File System Access API including user permission and path restrictions

Given: A conformant browser with SYS-1 or higher capability and File System Access API support

Task: File System Access API provides powerful capabilities to read and write local files and directories, creating risks of unauthorized data exfiltration, ransomware-style file encryption, malicious file modification, and access to sensitive system directories. Without strict controls, malicious sites could silently read user documents, modify critical files, or encrypt files for ransom. OS-native file pickers, separate write confirmation, system directory filtering, and per-access authorization prevent filesystem abuse while enabling legitimate file editing applications.

Verification:

1. Test window.showOpenFilePicker() and verify file picker dialog appears
2. Verify user should explicitly select files through OS file picker
3. Test window.showDirectoryPicker() and verify directory picker dialog
4. Grant directory access and verify files within are accessible
5. Test write access requires separate user confirmation
6. Attempt to access system directories and verify blocking/filtering
7. Test that file handles persist and verify permission prompt on reuse
8. Verify cross-origin iframes cannot access file handles
9. Test permission revocation clears all file handles
10. Verify HTTPS context required for persistent permissions

11. OS file/directory picker appears for all access requests
12. User should explicitly select files/directories
13. Write access requires separate confirmation
14. System directories are blocked or filtered from picker
15. File handles require permission on reuse after restart
16. Cross-origin access to file handles is blocked
17. HTTPS required for persistent file handle permissions
18. Permission revocation clears all granted handles
19. Each file/directory access is separately authorized
20. No programmatic file system enumeration possible

Pass Criteria: OS picker required AND write needs confirmation AND system directories blocked AND handles require reauthorization

Fail Criteria: No picker shown OR write without confirmation OR system directories accessible OR handles work without reauth

Evidence: File picker screenshots, directory picker UI, write confirmation prompts, system directory blocking tests, file handle persistence tests, cross-origin blocking verification

References:

- File System Access API: https://wicg.github.io/file-system-access/
- File System Access Security: https://wicg.github.io/file-system-access/#privacy-considerations
- Chrome File System Access: https://developer.chrome.com/articles/file-system-access/
- WHATWG File System: https://fs.spec.whatwg.org/
- OWASP File Security: https://cheatsheetseries.owasp.org/cheatsheets/File_Upload_Cheat_Sheet.html

Assessment: SYS-REQ-15 (WebUSB security controls)

Reference: SYS-REQ-15 - Browser shall implement comprehensive security controls for WebUSB including device filtering, permission management, and secure contexts

Given: A conformant browser with SYS-1 or higher capability and WebUSB support

Task: WebUSB's comprehensive device access requires layered security controls beyond basic permission prompts to prevent exploitation of protected device classes, dangerous control transfers, and vendor-sensitive devices. Without interface class filtering, attackers could claim HID interfaces to log keystrokes or access mass storage to exfiltrate files. Control transfer validation, protected class filtering, vendor opt-out respect, and secure context requirements create defense-in-depth protection for USB device interactions.

Verification:

1. Test navigator.usb.getDevices() and verify only previously authorized devices returned
2. Verify protected USB classes are filtered (HID keyboards/mice, mass storage, video, audio)
3. Test USB device access requires user activation (transient user gesture)
4. Verify vendors can opt out devices via USB device descriptor
5. Test that WebUSB requires secure context (HTTPS or localhost)
6. Attempt interface claiming on protected interface classes and verify blocking
7. Test USB device connection events fire only for authorized devices
8. Verify control transfers are validated and potentially dangerous ones blocked
9. Test that permissions-policy can restrict WebUSB in iframes
10. Verify USB device access is auditable through DevTools protocol
11. Protected USB device classes are never shown in picker
12. Only secure contexts can access WebUSB API
13. User activation required for device requests
14. Previously authorized devices require getDevices() call
15. Protected interface classes cannot be claimed
16. Device connection events only for authorized devices

17. Control transfers are validated for safety
18. Permissions Policy successfully restricts WebUSB
19. DevTools shows USB activity for debugging
20. Vendor opt-out mechanism is respected

Pass Criteria: Protected classes filtered AND secure context required AND user activation needed AND control transfers validated

Fail Criteria: Protected classes available OR insecure context works OR no user activation required OR dangerous transfers allowed

Evidence: Device picker showing filtered devices, secure context requirement tests, protected interface class blocking logs, control transfer validation tests, Permissions Policy test results

References:

- WebUSB Specification: https://wicg.github.io/webusb/
- WebUSB Protected Interface Classes: https://wicg.github.io/webusb/#protected-interface-classes
- USB Implementers Forum: https://www.usb.org/
- Chrome WebUSB Security: https://chromium.googlesource.com/chromium/src/+/refs/heads/main/docs/security/permis for-powerful-web-platform-features.md

Assessment: SYS-REQ-16 (WebBluetooth security)

Reference: SYS-REQ-16 - Browser shall implement security controls for Web Bluetooth including GATT blocklist, device filtering, and permission management

Given: A conformant browser with SYS-1 or higher capability and Web Bluetooth support

Task: Web Bluetooth GATT services provide deep access to device functionality, creating risks of HID service exploitation for keystroke injection, firmware update service abuse for device bricking or malware installation, and fingerprinting through device name enumeration. Without a comprehensive GATT blocklist, malicious sites could exploit dangerous services to compromise connected devices or user privacy. GATT blocklist enforcement, service UUID filtering, device name sanitization, and secure context requirements prevent Bluetooth-based attacks.

Verification:

1. Test navigator.bluetooth.getDevices() returns only previously authorized devices
2. Verify GATT blocklist prevents access to dangerous services (HID, firmware update)
3. Test that Web Bluetooth requires secure context (HTTPS or localhost)
4. Verify user activation required for requestDevice() calls
5. Test service UUID filters work correctly in device selection
6. Attempt to access blocklisted GATT characteristics and verify blocking
7. Test that optional services still require user awareness
8. Verify device name filtering prevents fingerprinting
9. Test permissions-policy restricts Web Bluetooth in cross-origin iframes
10. Verify Bluetooth scanning requires explicit user permission
11. Secure context (HTTPS/localhost) required for all Web Bluetooth APIs
12. User activation required for device requests
13. GATT blocklist prevents access to dangerous services/characteristics
14. Service UUID filtering correctly limits accessible services
15. Blocklisted characteristics return errors when accessed
16. Optional services declared in requestDevice()
17. Device names sanitized to prevent fingerprinting
18. Permissions Policy successfully restricts Web Bluetooth
19. Bluetooth scanning requires separate permission
20. Only previously granted devices in getDevices()

Pass Criteria: Secure context required AND GATT blocklist enforced AND user activation needed AND fingerprinting prevented

Fail Criteria: Insecure context works OR blocklist bypassed OR no user activation required OR fingerprinting possible

Evidence: Secure context requirement tests, GATT blocklist enforcement logs, service UUID filtering results, fingerprinting prevention tests, Permissions Policy test results

References:

- Web Bluetooth Specification: https://webbluetoothcg.github.io/web-bluetooth/
- Web Bluetooth GATT Blocklist: https://github.com/WebBluetoothCG/registries/blob/master/gatt_blocklist.txt
- Bluetooth GATT Specifications: https://www.bluetooth.com/specifications/specs/
- Web Bluetooth Security Model: https://webbluetoothcg.github.io/web-bluetooth/#security-and-privacy-considerations
- Chrome Web Bluetooth Security: https://sites.google.com/a/chromium.org/dev/developers/design-documents/bluetooth-design-docs

Assessment: SYS-REQ-17 (WebNFC permission management)

Reference: SYS-REQ-17 - Browser shall enforce permission controls for Web NFC API with user prompts and secure context requirements

Given: A conformant browser with SYS-1 or higher capability and Web NFC support

Task: Web NFC enables reading and writing NFC tags, creating risks of malicious tag writing to deploy phishing attacks, NFC relay attacks, unauthorized data collection from contactless payment cards, and privacy violations through persistent NFC scanning. Background NFC access could enable covert tag reading while users are unaware. Secure context requirements, permission prompts with user gestures, background operation blocking, and dangerous tag filtering prevent NFC-based attacks while enabling legitimate tag interactions.

Verification:

1. Test NDEFReader.scan() and verify permission prompt appears
2. Verify Web NFC requires secure context (HTTPS or localhost)
3. Test that NFC access requires user gesture for permission prompt
4. Grant NFC permission and verify scan operations work
5. Test NDEFReader.write() and verify it respects same permission
6. Verify cross-origin iframe NFC access is blocked without permission policy
7. Test permission revocation immediately stops NFC scanning
8. Verify NFC operations blocked when page in background
9. Test that dangerous NFC tag types are filtered or sandboxed
10. Verify NFC access is per-origin and isolated
11. Secure context (HTTPS/localhost) required for Web NFC
12. Permission prompt appears before NFC access granted
13. User gesture required to trigger permission prompt
14. Both scan and write operations respect same permission
15. Cross-origin iframe access blocked without delegation
16. Permission revocation stops active scans
17. Background pages cannot perform NFC operations
18. Dangerous tag operations are restricted
19. Permissions are per-origin and isolated
20. NFC indicators show when NFC is active

Pass Criteria: Secure context required AND permission prompt shown AND user gesture needed AND background access blocked

Fail Criteria: Insecure context works OR no permission prompt OR no user gesture required OR background access allowed

Evidence: NFC permission prompt screenshots, secure context requirement tests, user gesture verification, background access blocking tests, cross-origin test results, dangerous tag filtering verification

References:

- Web NFC API: https://w3c.github.io/web-nfc/
- Web NFC Security: https://w3c.github.io/web-nfc/#security-and-privacy
- NFC Forum Specifications: https://nfc-forum.org/our-work/specification-releases/
- Web NFC Explainer: https://github.com/w3c/web-nfc/blob/gh-pages/EXPLAINER.md

Assessment: SYS-REQ-18 (Sensor API permissions)

Reference: SYS-REQ-18 - Browser shall enforce permission controls for Generic Sensor APIs including accelerometer, gyroscope, and magnetometer

Given: A conformant browser with SYS-1 or higher capability and Sensor API support

Task: Generic Sensor APIs expose motion and environmental data that enable fingerprinting, keylogging through motion analysis, PIN theft via accelerometer side channels, and location tracking through magnetometer readings. High-frequency sensor access amplifies these attacks by providing precise timing data for cryptographic attacks. Secure context requirements, permission controls, background operation suspension, frequency limits, and Permissions Policy enforcement prevent sensor-based attacks while enabling legitimate motion and orientation detection.

Verification:

1. Test Accelerometer creation and verify permission prompt or policy enforcement
2. Verify secure context required for sensor APIs
3. Test Gyroscope access and verify same permission model
4. Create Magnetometer sensor and verify permissions
5. Test that high-frequency sensor access may require additional permissions
6. Verify sensors stop when page moves to background
7. Test cross-origin iframe sensor access requires permission policy delegation
8. Verify sensor permissions are per-origin
9. Test that ambient light sensor respects privacy considerations
10. Verify sensor access can be blocked via Permissions Policy
11. Secure context required for all Sensor APIs
12. Permission prompts or policies apply before sensor access
13. High-frequency access may require explicit permission
14. Sensors automatically pause in background
15. Cross-origin access requires Permissions Policy delegation
16. Permissions are per-origin and isolated
17. Privacy-sensitive sensors have additional restrictions
18. Permissions Policy can block sensor access
19. Sensor frequency is limited to prevent fingerprinting
20. Clear user controls for sensor permissions

Pass Criteria: Secure context required AND permissions enforced AND background pausing works AND Permissions Policy respected

Fail Criteria: Insecure context works OR no permissions enforced OR background access allowed OR policy ignored

Evidence: Sensor permission prompt screenshots, secure context requirement tests, background pausing verification, Permissions Policy test results, frequency limiting tests, cross-origin blocking verification

References:

- Generic Sensor API: https://www.w3.org/TR/generic-sensor/
- Sensor Security Model: https://www.w3.org/TR/generic-sensor/#security-and-privacy
- Accelerometer API: https://www.w3.org/TR/accelerometer/
- Gyroscope API: https://www.w3.org/TR/gyroscope/
- Permissions Policy: https://www.w3.org/TR/permissions-policy-1/

Assessment: SYS-REQ-19 (Battery Status API restrictions)

Reference: SYS-REQ-19 - Browser shall implement privacy restrictions for Battery Status API to prevent fingerprinting

Given: A conformant browser with SYS-1 or higher capability

Task: Battery Status API historically enabled precise device fingerprinting through battery level, charging time, and discharge rate patterns that uniquely identify devices across browsing sessions and origins. High-precision battery data combined with timing information creates a persistent tracking identifier resistant to cookie deletion. Rounding battery levels, quantizing timing data, throttling updates, and rate limiting prevent battery-based fingerprinting while providing sufficient information for legitimate power management features.

Verification:

1. Test navigator.getBattery() and observe battery information returned
2. Verify battery level is rounded to prevent high-precision fingerprinting
3. Test battery timing information is quantized to prevent tracking
4. Verify battery status updates are throttled
5. Test that battery information is not available in insecure contexts
6. Verify battery status in cross-origin iframes requires permission policy
7. Test that frequent battery queries are rate-limited
8. Verify battery API can be disabled via browser settings or policy
9. Test that battery charging state changes are debounced
10. Verify no access to detailed battery analytics or history
11. Battery level rounded to coarse granularity (e.g., 1% or 5%)
12. Timing information quantized to prevent precise measurements
13. Update frequency throttled to prevent tracking
14. Secure context recommended for battery API access
15. Cross-origin access requires Permissions Policy
16. Rate limiting prevents rapid polling
17. Battery API can be disabled by user/policy
18. Charging state changes debounced
19. No historical battery data exposed
20. API surface minimized for privacy

Pass Criteria: Battery data quantized AND updates throttled AND rate limiting enforced AND no detailed analytics exposed

Fail Criteria: Precise battery data OR no throttling OR no rate limiting OR historical data exposed

Evidence: Battery level precision tests, timing quantization measurements, update frequency analysis, rate limiting verification, cross-origin test results, privacy analysis reports

References:

- Battery Status API: https://www.w3.org/TR/battery-status/
- Battery API Privacy Concerns: https://www.w3.org/TR/battery-status/#privacy-considerations
- Web API Privacy: https://www.w3.org/TR/fingerprinting-guidance/
- Chrome Battery Status: https://chromestatus.com/feature/4537134732017664

Assessment: SYS-REQ-20 (Hardware resource limits)

Reference: SYS-REQ-20 - Browser shall enforce resource limits to prevent excessive consumption of CPU, memory, and system resources

Given: A conformant browser with SYS-1 or higher capability

Task: Unrestricted hardware resource consumption enables denial-of-service attacks that freeze browsers, crash systems, drain battery, or make devices unusable through memory exhaustion, CPU monopolization, or GPU resource depletion. Malicious scripts with infinite loops or excessive allocations can render browsers unresponsive. Per-origin resource limits, background throttling, script timeouts, and memory quotas prevent resource-based DoS attacks while maintaining browser and system responsiveness.

Verification:

1. Create test page that attempts to allocate excessive memory and verify limits
2. Test CPU-intensive operations and verify throttling or limits applied
3. Monitor browser resource usage with intensive JavaScript loops
4. Test WebWorker resource limits and verify isolation
5. Verify background tab resource throttling is active
6. Test WebAssembly memory limits and verify enforcement
7. Monitor GPU memory usage and verify limits on WebGL contexts
8. Test that runaway scripts trigger timeout warnings or termination
9. Verify resource limits apply per-origin or per-process
10. Test that browser remains responsive under resource pressure
11. Memory allocation limits prevent excessive consumption
12. CPU-intensive operations are throttled
13. Background tabs have reduced resource quotas
14. WebWorkers have separate resource limits
15. WebAssembly memory is bounded and enforced
16. GPU memory limits prevent resource exhaustion
17. Script timeouts prevent infinite loops
18. Resource limits are per-origin or per-process
19. Browser UI remains responsive under load
20. User can terminate runaway processes/tabs

Pass Criteria: Memory limits enforced AND CPU throttling active AND background throttling works AND script timeouts prevent hangs

Fail Criteria: No memory limits OR no CPU throttling OR background tabs not throttled OR scripts can hang indefinitely

Evidence: Memory allocation test results, CPU usage graphs showing throttling, background tab resource measurements, WebWorker limit tests, script timeout logs, browser responsiveness tests

References:

- WebAssembly Memory: https://webassembly.github.io/spec/core/syntax/modules.html#memories
- Script Execution Limits: https://html.spec.whatwg.org/multipage/webappapis.html#long-tasks
- Firefox Process Limits: https://wiki.mozilla.org/Project_Fission

Assessment: SYS-REQ-21 (Memory isolation enforcement)

Reference: SYS-REQ-21 - Browser shall enforce memory isolation between processes to prevent cross-process memory access

Given: A conformant browser with SYS-1 or higher capability

Task: Memory isolation failures enable cross-process data theft, Spectre-class attacks to read arbitrary memory, privilege escalation through memory corruption, and exploitation of use-after-free vulnerabilities.

Without ASLR, SharedArrayBuffer restrictions, and Spectre mitigations, attackers can reliably exploit memory vulnerabilities to breach process boundaries. Memory isolation with ASLR, cross-origin isolation requirements, memory zeroing, and hardware protections prevents cross-process memory attacks.

Verification:

1. Launch browser with multiple processes and identify their memory spaces
2. Use memory analysis tools to verify process memory isolation (valgrind, windbg, lldb)
3. Test that renderer processes cannot access browser process memory
4. Verify SharedArrayBuffer requires cross-origin isolation
5. Test that different origins cannot share memory without explicit mechanisms
6. Verify ASLR (Address Space Layout Randomization) is enabled
7. Test that Spectre mitigations prevent speculative execution memory leaks
8. Verify memory is zeroed when deallocated and reused
9. Test that memory dumps do not leak data between processes
10. Verify hardware memory protection (NX, DEP) is enabled
11. Each process has isolated virtual memory space
12. Renderer processes cannot read browser process memory
13. SharedArrayBuffer requires explicit CORS headers
14. ASLR randomizes memory addresses per process
15. Spectre/Meltdown mitigations prevent side-channel leaks
16. Memory is zeroed on deallocation
17. No cross-process memory leaks detectable
18. Hardware NX/DEP prevents code execution in data pages
19. Process crashes don't leak memory to other processes
20. Memory forensics shows proper isolation

Pass Criteria: Process memory isolated AND SharedArrayBuffer restricted AND ASLR enabled AND Spectre mitigations active

Fail Criteria: Cross-process memory access possible OR SharedArrayBuffer unrestricted OR no ASLR OR Spectre vulnerable

Evidence: Memory map analysis, process memory dumps, SharedArrayBuffer test results, ASLR verification, Spectre/Meltdown mitigation tests, memory leak detection reports

References:

- SharedArrayBuffer Security: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Sha
- Spectre Mitigations: https://www.chromium.org/Home/chromium-security/ssca/
- ASLR: https://en.wikipedia.org/wiki/Address_space_layout_randomization
- Site Isolation: https://www.chromium.org/Home/chromium-security/site-isolation/

Assessment: SYS-REQ-22 (CPU resource quotas)

Reference: SYS-REQ-22 - Browser shall enforce CPU resource quotas to prevent any single origin from monopolizing processor resources

Given: A conformant browser with SYS-1 or higher capability

Task: CPU resource monopolization by malicious scripts enables cryptojacking attacks that steal processing power for cryptocurrency mining, denial-of-service through CPU exhaustion, battery drain on mobile devices, and system slowdowns affecting user productivity. Background tabs consuming CPU enable covert mining operations. CPU quotas with background throttling, timer throttling, requestAnimationFrame suspension, and per-origin limits prevent CPU monopolization while maintaining responsiveness for foreground content.

Verification:

1. Create test page with intensive CPU computation (tight loop or crypto operations)
2. Monitor CPU usage per process using OS tools (top, Task Manager, Activity Monitor)

3. Test that background tabs have reduced CPU quotas
4. Verify WebWorker CPU usage is counted toward origin quota
5. Test CPU throttling for invisible pages (display:none, zero opacity)
6. Verify timer throttling for background processes (setTimeout, setInterval)
7. Test that requestAnimationFrame is paused for background tabs
8. Monitor overall system CPU and verify browser doesn't exceed limits
9. Test that users can identify and terminate high-CPU tabs
10. Verify CPU priority scheduling favors foreground/visible content
11. Background tabs throttled to <1% CPU typically
12. Foreground tabs have priority CPU access
13. WebWorker CPU usage tracked per origin
14. Invisible elements have reduced CPU quotas
15. Timers throttled in background (1Hz or lower)
16. requestAnimationFrame paused when not visible
17. Browser task manager shows per-tab CPU usage
18. Users can terminate high-CPU tabs easily
19. System CPU usage remains reasonable under load
20. CPU scheduling prioritizes user-visible content

Pass Criteria: Background throttling active AND timers throttled AND rAF paused AND user can terminate high-CPU tabs

Fail Criteria: No background throttling OR timers not throttled OR rAF active in background OR no termination controls

Evidence: CPU usage graphs per tab, background throttling measurements, timer frequency tests, rAF execution logs, task manager screenshots, system CPU monitoring

References:

- Page Lifecycle API: https://developer.chrome.com/blog/page-lifecycle-api/
- Timer Throttling: https://developer.mozilla.org/en-US/docs/Web/API/setTimeout#throttling
- requestAnimationFrame: https://developer.mozilla.org/en-US/docs/Web/API/window/requestAnimationFrame
- Firefox Process Priority: https://wiki.mozilla.org/Project_Fission#Process_prioritization

Assessment: SYS-REQ-23 (Network bandwidth limits)

Reference: SYS-REQ-23 - Browser shall enforce network bandwidth limits to prevent excessive network resource consumption

Given: A conformant browser with SYS-1 or higher capability

Task: Uncontrolled network resource consumption enables bandwidth exhaustion attacks, connection pool depletion preventing legitimate requests, network-based denial-of-service, and excessive data usage on metered connections. Malicious origins opening unlimited connections can monopolize network resources. Per-origin connection limits, HTTP/2 stream limits, WebSocket quotas, and background deprioritization prevent network resource abuse while enabling efficient multiplexing and user control.

Verification:

1. Create test page that attempts high-volume network requests
2. Monitor network usage per origin using browser DevTools Network panel
3. Test connection limits per origin (typically 6-10 connections)
4. Verify HTTP/2 and HTTP/3 multiplexing limits
5. Test bandwidth throttling for background tabs
6. Verify WebSocket connection limits per origin
7. Test that large downloads can be paused or cancelled
8. Monitor overall browser network usage and verify limits
9. Test that fetch() requests respect connection pooling limits

10. Verify users can identify and control network-heavy tabs
11. Connection limits enforced per origin (6-10 connections)
12. HTTP/2 stream limits enforced (max concurrent streams)
13. Background tabs have reduced network priority
14. WebSocket connections limited per origin
15. Large transfers pausable and cancellable
16. Connection pooling prevents excessive sockets
17. Browser task manager shows network usage per tab
18. Network throttling options available in DevTools
19. Overall browser network reasonable under load
20. Users can control network-heavy operations

Pass Criteria: Connection limits enforced AND background deprioritization AND WebSocket limits AND user controls available

Fail Criteria: Unlimited connections OR no background priority OR unlimited WebSockets OR no user controls

Evidence: Network connection graphs, DevTools Network panel screenshots, connection limit test results, WebSocket limit verification, background priority measurements, task manager network stats

References:

- HTTP Connection Management: https://developer.mozilla.org/en-US/docs/Web/HTTP/Connection_management_in_
- HTTP/2 Specification: https://httpwg.org/specs/rfc7540.html
- Chrome Network Throttling: https://developer.chrome.com/docs/devtools/network/reference/#throttling
- WebSocket Limits: https://datatracker.ietf.org/doc/html/rfc6455
- Firefox Network Priority: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Priority

Assessment: SYS-REQ-24 (Storage quota enforcement)

Reference: SYS-REQ-24 - Browser shall enforce storage quotas for origin-scoped storage mechanisms including IndexedDB, Cache API, and local storage

Given: A conformant browser with SYS-1 or higher capability

Task: Unrestricted storage enables disk space exhaustion attacks, storage-based denial-of-service, tracking through persistent data storage, and system instability from full disks. Malicious origins filling storage prevent legitimate applications from functioning. Per-origin storage quotas, persistent storage permissions, QuotaExceededError enforcement, and separate incognito limits prevent storage abuse while enabling adequate space for web applications and maintaining system stability.

Verification:

1. Test StorageManager.estimate() to query available storage quota
2. Attempt to exceed storage quota using IndexedDB and verify QuotaExceededError
3. Test localStorage quota limits (typically 5-10MB)
4. Verify Cache API respects storage quota
5. Test that storage quota is per-origin and isolated
6. Verify persistent storage requires explicit user permission
7. Test that storage usage is accurately reported in browser settings
8. Verify storage can be cleared per-origin by user
9. Test that incognito/private mode has separate, limited storage
10. Verify storage quota warnings or prompts for large allocations
11. Storage quotas enforced per-origin (typically 50-100MB default)
12. StorageManager API accurately reports usage and quota
13. QuotaExceededError thrown when limits reached
14. localStorage limited to 5-10MB per origin
15. Cache API storage counted toward quota

16. Persistent storage requires user permission
17. Browser settings show per-origin storage usage
18. Users can clear storage per-origin
19. Incognito mode has separate, limited quota
20. Large storage requests trigger user prompts

Pass Criteria: Quotas enforced per-origin AND QuotaExceededError thrown AND persistent storage requires permission AND user controls available

Fail Criteria: No quota enforcement OR errors not thrown OR persistent storage auto-granted OR no user controls

Evidence: StorageManager.estimate() results, QuotaExceededError screenshots, localStorage limit tests, Cache API quota tests, browser settings storage UI, incognito mode quota verification

References:

- Storage API: https://storage.spec.whatwg.org/
- StorageManager: https://developer.mozilla.org/en-US/docs/Web/API/StorageManager
- IndexedDB Quota: https://www.w3.org/TR/IndexedDB/#dom-idbdatabase-transaction
- Chrome Storage Quotas: https://web.dev/articles/storage-for-the-web
- Firefox Storage Limits: https://developer.mozilla.org/en-US/docs/Web/API/Storage_API/Storage_quotas_and_evictio

Assessment: SYS-REQ-25 (Process priority management)

Reference: SYS-REQ-25 - Browser shall implement process priority management to ensure responsive user experience and fair resource allocation

Given: A conformant browser with SYS-1 or higher capability

Task: Without process priority management, background tabs can monopolize CPU resources causing foreground tab lag, audio playback in background can be interrupted unfairly, and browser responsiveness suffers from equal treatment of all processes. Priority inversion where background processes starve foreground operations degrades user experience. Dynamic process priority management ensuring foreground preference, audio continuity, and fair scheduling prevents resource contention while maintaining responsive user interaction.

Verification:

1. Launch browser with multiple tabs and identify process priorities using OS tools
2. Verify foreground tab processes have higher priority than background
3. Test that audio-playing tabs maintain elevated priority
4. Verify visible tabs have higher priority than hidden tabs
5. Test process priority changes when switching tabs
6. Monitor CPU scheduling and verify foreground processes get preference
7. Test that extension processes have appropriate priority
8. Verify utility processes (network, GPU) have appropriate priority
9. Test that frozen background tabs have lowest priority
10. Verify process priorities don't allow starvation
11. Foreground tabs run at higher OS process priority
12. Background tabs run at reduced priority (low or below normal)
13. Audio-playing tabs maintain adequate priority
14. Priority changes dynamically when switching tabs
15. CPU scheduler gives preference to high-priority processes
16. Extension processes have appropriate priority
17. Utility processes prioritized by criticality
18. Frozen tabs have lowest priority
19. No process starvation occurs
20. User-facing processes most responsive

Pass Criteria: Foreground tabs high priority AND background tabs low priority AND dynamic priority adjustment AND no starvation

Fail Criteria: All same priority OR no background reduction OR static priorities OR starvation occurs

Evidence: Process priority listings (nice values, Windows priority classes), CPU scheduling analysis, tab switching priority changes, audio playback priority verification, process responsiveness measurements

References:

- Chrome Process Model: https://www.chromium.org/developers/design-documents/multi-process-architecture/
- Linux Process Priority: https://man7.org/linux/man-pages/man2/nice.2.html
- Windows Process Priority: https://docs.microsoft.com/en-us/windows/win32/procthread/scheduling-priorities
- Page Lifecycle: https://developer.chrome.com/blog/page-lifecycle-api/

Assessment: SYS-REQ-26 (Sandbox escape prevention)

Reference: SYS-REQ-26 - Browser shall implement multiple layers of defense to prevent sandbox escape attacks

Given: A conformant browser with SYS-1 or higher capability

Task: Sandbox escapes are high-severity vulnerabilities that allow compromised renderers to break containment and gain full system access, enabling malware installation, data exfiltration, and system compromise. Single-layer sandboxes are vulnerable to bypass through exploitation of implementation bugs or kernel vulnerabilities. Defense-in-depth with multiple sandbox layers, syscall filtering, IPC validation, kernel exploit mitigations, and continuous fuzzing prevents sandbox escapes even when individual defenses are bypassed.

Verification:

1. Review browser's sandbox architecture documentation and layers of defense
2. Test syscall filtering using strace/dtruss to verify blocked calls
3. Attempt known sandbox escape techniques and verify they're mitigated
4. Verify IPC validation prevents malformed messages from escaping sandbox
5. Test that renderer processes cannot ptrace or debug other processes
6. Verify memory corruption in renderer doesn't allow privilege escalation
7. Test that GPU process sandbox is separate from renderer sandbox
8. Verify kernel exploit mitigations (SMEP, SMAP, KPTI) are active
9. Test fuzzing results for sandbox bypass vulnerabilities
10. Verify security updates address discovered sandbox escapes
11. Multiple sandbox layers (syscall filter, namespace isolation, capability dropping)
12. Dangerous syscalls blocked (execve, ptrace, mount, etc.)
13. IPC messages validated in broker process
14. Process debugging APIs blocked
15. Memory corruption contained within sandbox
16. GPU sandbox independent from renderer
17. Kernel exploit mitigations active (SMEP, SMAP, KPTI)
18. Regular fuzzing for sandbox bypasses
19. Security updates address known escapes
20. Defense-in-depth architecture

Pass Criteria: Multiple sandbox layers AND syscall filtering AND IPC validation AND kernel mitigations active

Fail Criteria: Single layer only OR syscalls not filtered OR IPC not validated OR kernel mitigations missing

Evidence: Sandbox architecture documentation, syscall trace logs, sandbox escape test results, IPC validation logs, kernel mitigation verification, fuzzing reports, security update changelogs

References:

- Chromium Sandbox: https://chromium.googlesource.com/chromium/src/+/master/docs/design/sandbox.md
- Linux Namespace Isolation: https://man7.org/linux/man-pages/man7/namespaces.7.html
- Seccomp BPF: https://www.kernel.org/doc/html/latest/userspace-api/seccomp_filter.html
- SMEP and SMAP: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance.html
- Project Zero Sandbox Escapes: https://googleprojectzero.blogspot.com/

Assessment: SYS-REQ-27 (Speculative execution mitigations)

Reference: SYS-REQ-27 - Browser shall implement mitigations for speculative execution vulnerabilities including Spectre and Meltdown

Given: A conformant browser with SYS-1 or higher capability

Task: Spectre and Meltdown enable attackers to exploit speculative execution in modern CPUs to read arbitrary memory across security boundaries, including passwords, encryption keys, and cross-origin data. Without mitigations, malicious JavaScript can use timing side channels to leak sensitive data from other processes or origins. Site Isolation, timer quantization, SharedArrayBuffer restrictions, CORB, and COOP/COEP headers prevent speculative execution attacks by ensuring cross-origin data never shares address space.

Verification:

1. Verify Site Isolation is enabled using chrome://process-internals or equivalent
2. Test that high-resolution timers are reduced precision (performance.now() granularity)
3. Verify SharedArrayBuffer requires cross-origin isolation headers
4. Test that cross-origin data is not accessible in same process
5. Verify CORB (Cross-Origin Read Blocking) prevents sensitive data leaks
6. Test that speculative execution cannot leak cross-origin data
7. Verify process-per-site-instance isolation for sensitive origins
8. Test that COOP/COEP headers enable process isolation
9. Verify kernel page-table isolation (KPTI) is active on system
10. Test that Spectre PoC exploits are mitigated
11. Site Isolation active for all sites or at least sensitive origins
12. Timer precision reduced to 100 microseconds or coarser
13. SharedArrayBuffer requires COOP+COEP headers
14. Cross-origin data in separate processes
15. CORB blocks cross-origin reads of sensitive MIME types
16. Spectre PoC exploits fail to leak data
17. Process-per-site-instance for banks, login pages
18. COOP/COEP enable strict process isolation
19. KPTI active on vulnerable systems
20. Regular updates address new speculative execution vulnerabilities

Pass Criteria: Site Isolation enabled AND timer precision reduced AND SharedArrayBuffer restricted AND CORB active

Fail Criteria: No Site Isolation OR high-precision timers OR SharedArrayBuffer unrestricted OR CORB disabled

Evidence: Site Isolation status screenshots, timer precision measurements, SharedArrayBuffer tests with/without headers, CORB blocking logs, Spectre PoC test results, KPTI verification

References:

- Spectre Mitigations: https://www.chromium.org/Home/chromium-security/ssca/
- Site Isolation: https://www.chromium.org/Home/chromium-security/site-isolation/
- SharedArrayBuffer: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/SharedArray

- CORB: https://chromium.googlesource.com/chromium/src/+/master/services/network/cross_origin_read_blocking_ex
- Spectre Attack: https://spectreattack.com/

Assessment: SYS-REQ-28 (Side-channel attack mitigations)

Reference: SYS-REQ-28 - Browser shall implement mitigations for side-channel attacks including timing attacks, cache attacks, and fingerprinting

Given: A conformant browser with SYS-1 or higher capability

Task: Side-channel attacks exploit timing differences, cache behavior, rendering performance, and other observable effects to infer sensitive information like user history, cross-origin data, or cryptographic keys. High-precision timers enable microarchitectural attacks, CSS :visited timing leaks browsing history, and cache timing reveals cross-origin content. Timer quantization, process isolation for cache partitioning, cross-origin timing sanitization, and fingerprinting surface minimization prevent side-channel information leakage.

Verification:

1. Test timer precision reduction (performance.now(), Date.now()) to verify quantization
2. Verify SharedArrayBuffer is disabled or requires isolation headers
3. Test that cache timing attacks are mitigated through process isolation
4. Verify rendering timing cannot leak cross-origin information
5. Test that event loop timing is not exploitable for side channels
6. Verify CSS timing attacks are mitigated (e.g., :visited link timing)
7. Test that network timing doesn't leak cross-origin data
8. Verify SVG filter timing attacks are prevented
9. Test that WebGL cannot be used for timing side channels
10. Verify fingerprinting surfaces are minimized or permissions-gated
11. Timer precision reduced (100 microseconds or coarser)
12. SharedArrayBuffer requires COOP+COEP isolation
13. Process isolation prevents cache-based side channels
14. Rendering timing doesn't leak cross-origin data
15. Event loop timing not exploitable
16. :visited link styling restricted to prevent timing attacks
17. Network timing (Resource Timing API) sanitized for cross-origin
18. SVG filter timing attacks mitigated
19. WebGL timing attacks prevented through various restrictions
20. Fingerprinting APIs require permission or are restricted

Pass Criteria: Timer quantization active AND SharedArrayBuffer isolated AND process isolation prevents cache attacks AND cross-origin timing sanitized

Fail Criteria: High-precision timers OR SharedArrayBuffer unrestricted OR no cache isolation OR timing leaks exist

Evidence: Timer precision tests, SharedArrayBuffer isolation verification, cache timing attack tests, cross-origin timing measurements, fingerprinting surface analysis, side-channel PoC test results

References:

- Web Platform Security: https://www.w3.org/TR/fingerprinting-guidance/
- Timing Attacks: https://www.w3.org/TR/hr-time-2/#clock-resolution
- CSS :visited Privacy: https://developer.mozilla.org/en-US/docs/Web/CSS/Privacy_and_the_:visited_selector
- Resource Timing: https://www.w3.org/TR/resource-timing-2/
- Fingerprinting Resistance: https://brave.com/privacy-updates/3-fingerprint-randomization/

Assessment: SYS-REQ-29 (Hardware token security)

Reference: SYS-REQ-29 - Browser shall implement secure access controls for hardware security tokens and authenticators (WebAuthn)

Given: A conformant browser with SYS-1 or higher capability and WebAuthn support

Task: Hardware security tokens provide strong authentication but require secure browser integration to prevent phishing, credential theft, and authenticator cloning attacks. Without HTTPS requirements and origin binding, attackers could steal credentials or use fake authenticators. User verification and presence requirements prevent remote attacks. HTTPS enforcement, origin-bound credentials, attestation validation, user presence requirements, and extension isolation ensure WebAuthn security while enabling passwordless authentication.

Verification:

1. Test WebAuthn registration with hardware security key (e.g., YubiKey)
2. Verify registration requires HTTPS context (except localhost)
3. Test that user verification (PIN/biometric) is enforced when required
4. Verify authenticator attestation is validated correctly
5. Test that credentials are origin-bound and not accessible cross-origin
6. Verify user presence is required (physical touch/button press)
7. Test that CTAP2 protocol is properly implemented for FIDO2 devices
8. Verify browser extensions cannot intercept WebAuthn operations
9. Test that authenticator selection respects user choice
10. Verify platform authenticators (Touch ID, Windows Hello) are isolated
11. WebAuthn requires secure context (HTTPS or localhost)
12. User verification enforced when requested by relying party
13. Attestation validation prevents cloned/fake authenticators
14. Credentials strictly origin-bound (site isolation)
15. User presence required (physical interaction)
16. CTAP2 protocol correctly implemented
17. Extensions cannot intercept or modify WebAuthn
18. Users choose authenticator from picker
19. Platform authenticators use OS-level secure enclaves
20. Private keys never accessible to JavaScript

Pass Criteria: HTTPS required AND user verification enforced AND origin-bound credentials AND user presence required

Fail Criteria: HTTP allowed OR no user verification OR cross-origin access OR no user presence

Evidence: WebAuthn registration screenshots, attestation validation logs, cross-origin test results, user presence verification, CTAP2 protocol traces, platform authenticator tests

References:

- WebAuthn Specification: https://www.w3.org/TR/webauthn-2/
- FIDO2 CTAP: https://fidoalliance.org/specs/fido-v2.0-ps-20190130/fido-client-to-authenticator-protocol-v2.0-ps-20190130.html
- WebAuthn Security: https://www.w3.org/TR/webauthn-2/#sctn-security-considerations
- FIDO Security Reference: https://fidoalliance.org/specifications/

Assessment: SYS-REQ-30 (Accessibility API security)

Reference: SYS-REQ-30 - Browser shall implement security controls for accessibility APIs to prevent information leakage and unauthorized access

Given: A conformant browser with SYS-1 or higher capability

Task: Accessibility APIs expose detailed page structure, content, and user interactions to assistive technologies, creating vectors for information leakage, fingerprinting, unauthorized content scraping, and privacy violations if not properly controlled. Malicious software masquerading as assistive technology could harvest passwords, track user actions, or extract sensitive content. OS-level permission requirements, sensitive data masking, cross-origin restrictions, and remote access authentication prevent accessibility API abuse while serving users with disabilities.

Verification:

1. Enable OS accessibility features (screen reader, voice control)
2. Verify browser exposes accessibility tree to authorized assistive technology only
3. Test that accessibility API requires user opt-in or OS-level permission
4. Verify accessibility tree doesn't leak sensitive data (passwords, hidden content)
5. Test that cross-origin content has restricted accessibility exposure
6. Verify accessibility events don't leak timing or user interaction information
7. Test that remote accessibility access (enterprise tools) requires authentication
8. Verify accessibility API cannot be abused for fingerprinting
9. Test that ARIA attributes don't leak security-sensitive information
10. Verify screen reader access is logged for security auditing
11. Accessibility API access requires OS-level permission
12. Only authorized assistive technology can access accessibility tree
13. Sensitive content (passwords) properly masked in accessibility tree
14. Cross-origin content has limited accessibility exposure
15. Accessibility events sanitized to prevent leakage
16. Remote accessibility requires authentication
17. Accessibility API cannot be used for fingerprinting
18. ARIA attributes don't leak sensitive information
19. Accessibility access auditable
20. Browser communicates securely with assistive technology

Pass Criteria: OS permission required AND sensitive data masked AND cross-origin limited AND remote access authenticated

Fail Criteria: No permission required OR passwords exposed OR cross-origin fully accessible OR remote access unauthenticated

Evidence: Accessibility permission flows, accessibility tree dumps showing masking, cross-origin accessibility tests, remote access authentication verification, fingerprinting tests, audit logs

References:

- WAI-ARIA: https://www.w3.org/TR/wai-aria-1.2/
- Accessibility Object Model: https://wicg.github.io/aom/spec/
- Chrome Accessibility: https://www.chromium.org/developers/design-documents/accessibility/
- macOS Accessibility: https://developer.apple.com/documentation/accessibility
- Windows Accessibility: https://docs.microsoft.com/en-us/windows/win32/winauto/entry-uiauto-win32

Assessment: SYS-REQ-31 (Native messaging restrictions)

Reference: SYS-REQ-31 - Browser shall enforce security restrictions on native messaging between extensions and native applications

Given: A conformant browser with SYS-1 and EXT-1 or higher capability

Task: Native messaging bridges the browser extension sandbox and native applications, creating risks of sandbox escape, privilege escalation, arbitrary code execution, and malware installation if not properly controlled. Malicious extensions could exploit native messaging to execute system commands, access sensitive files, or install malware. Manifest permissions, host registration, extension ID validation, message validation, and privilege limitation prevent native messaging abuse while enabling legitimate browser-native integration.

Verification:

1. Create test extension with nativeMessaging permission
2. Verify extension requires explicit permission declaration in manifest
3. Test that native messaging host is registered with browser
4. Verify only explicitly allowed extensions can communicate with native host
5. Test that native messaging requires user installation of native component
6. Verify message validation occurs on browser side
7. Test that native host process runs with limited privileges
8. Verify native messaging cannot bypass extension sandbox
9. Test that uninstalling extension terminates native messaging connections
10. Verify enterprise policies can restrict or acceptlist native messaging
11. nativeMessaging permission required in manifest
12. Native host is registered in browser's native messaging directory
13. Only permitted extensions (by ID) can connect to host
14. User should manually install native host application
15. Messages validated by browser before passing to/from host
16. Native host runs with limited OS privileges
17. Extension sandbox not bypassed via native messaging
18. Extension uninstall terminates native connections
19. Enterprise policies can control native messaging
20. Native host path validation prevents arbitrary execution

Pass Criteria: Manifest permission required AND host registration required AND extension ID validation AND user installs host

Fail Criteria: No permission required OR no host registration OR any extension can connect OR automatic host installation

Evidence: Extension manifest examples, native host registration files, permission validation tests, message validation logs, privilege analysis of native host, enterprise policy tests

References:

- Chrome Native Messaging: https://developer.chrome.com/docs/extensions/develop/concepts/native-messaging
- Firefox Native Messaging: https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/Native_messagin

Assessment: SYS-REQ-32 (Host OS integration security)

Reference: SYS-REQ-32 - Browser shall securely integrate with host operating system features while maintaining sandbox boundaries and security isolation

Given: A conformant browser with SYS-1 or higher capability

Task: Host OS integration provides essential functionality but creates attack surfaces for credential theft, certificate validation bypass, policy evasion, and privilege escalation if not properly secured. Insecure credential access exposes passwords, certificate store bypass enables MITM attacks, and policy violations undermine enterprise security. Secure IPC for credentials, OS certificate store usage, file association validation, policy compliance, and update signature verification maintain security boundaries while enabling OS integration.

Verification:

1. Test OS credential integration (Windows Credential Manager, macOS Keychain) and verify secure access
2. Verify browser uses OS certificate store with proper validation
3. Test OS notification integration respects permission model
4. Verify file type association handlers are validated and sandboxed
5. Test protocol handler registration requires user confirmation

6. Verify OS-level print dialog prevents renderer access to print system
7. Test that OS share APIs require user gesture and confirmation
8. Verify browser respects OS security policies (AppLocker, Gatekeeper, SELinux)
9. Test that browser updates use OS-level signature verification
10. Verify browser profile directories use OS access controls
11. OS credential storage accessed securely (encrypted IPC)
12. OS certificate store used with proper validation
13. OS notifications require permission and are per-origin
14. File associations validated before handler invocation
15. Protocol handlers require user confirmation
16. Print system accessed via broker, not renderer
17. OS share APIs require user gesture
18. Browser respects OS security policies
19. Updates verified with OS-level signatures
20. Profile directories protected with OS ACLs

Pass Criteria: Secure credential access AND certificate store used AND handlers validated AND security policies respected

Fail Criteria: Insecure credential access OR certificate store bypassed OR handlers not validated OR policies ignored

Evidence: Credential access flow analysis, certificate store usage verification, file association tests, protocol handler registration flows, OS policy compliance tests, update signature verification, ACL analysis

References:

- Windows Credential Manager: https://docs.microsoft.com/en-us/windows/security/identity-protection/credential-guard/
- macOS Keychain: https://developer.apple.com/documentation/security/keychain_services
- Code Signing: https://developer.apple.com/documentation/xcode/notarizing_macos_software_before_distribution
- Windows AppLocker: https://docs.microsoft.com/en-us/windows/security/threat-protection/windows-defender-application-control/applocker/applocker-overview
- SELinux: https://www.redhat.com/en/topics/linux/what-is-selinux

6.8 Embedded Browser Security Assessments

This section covers assessment procedures for requirements EMB-REQ-1 through EMB-REQ-32, addressing the unique security challenges of embedded browsers (WebView components, browser engines integrated into native applications). These assessments focus on JavaScript bridge security, native API exposure control, content source trust management, and host application boundary protection.

Assessment: EMB-REQ-1 (JavaScript bridge API allowlists)

Reference: EMB-REQ-1 - JavaScript bridges shall implement explicit allowlists of exposed native APIs with per-API access controls

Given: A conformant embedded browser with JavaScript bridge capability (EMB-1 or higher)

Task: JavaScript bridges expose native device APIs to web content, creating critical attack surfaces where unrestricted API exposure enables privilege escalation, data exfiltration, malware installation, and complete device compromise. Without explicit allowlists, malicious web content can invoke any native API through reflection or dynamic invocation. Explicit API allowlisting with per-API access controls, runtime immutability, reflection blocking, and comprehensive logging prevents bridge-based attacks while enabling controlled native functionality.

Verification:

1. Review the embedded browser's JavaScript bridge configuration and implementation code

2. Identify all native APIs exposed to web content through the JavaScript bridge
3. Verify that an explicit allowlist mechanism exists (configuration file, API declaration, or programmatic registration)
4. Attempt to call native APIs that are not in the allowlist from web content
5. Verify that per-API access controls exist (e.g., origin-based restrictions, permission requirements)
6. Test that the allowlist cannot be modified by web content at runtime
7. Attempt to use reflection or dynamic invocation to bypass the allowlist
8. Verify that the bridge logs all API access attempts including denied attempts
9. JavaScript bridge exposes only explicitly allowlisted native APIs
10. Configuration files or code clearly declare which APIs are accessible
11. Per-API access controls restrict which origins or contexts can call each API
12. Attempts to call non-allowlisted APIs result in errors or exceptions
13. Web content cannot modify the API allowlist at runtime
14. Reflection or dynamic method invocation does not bypass allowlist enforcement
15. All API access attempts are logged for security auditing

Pass Criteria: Only explicitly allowlisted APIs are callable from web content AND per-API access controls are enforced AND allowlist cannot be modified by web content

Fail Criteria: Any non-allowlisted API is callable OR access controls are bypassable OR web content can modify the allowlist

Evidence: JavaScript bridge configuration files, implementation code review, test results showing blocked API calls, security logs showing denied access attempts, penetration test reports

References:

- OWASP Mobile Top 10 - M1: Improper Platform Usage: https://owasp.org/www-project-mobile-top-10/
- Android WebView addJavascriptInterface Security: https://developer.android.com/reference/android/webkit/WebView#
- iOS WKWebView Configuration: https://developer.apple.com/documentation/webkit/wkwebviewconfiguration
- Electron Context Bridge: https://www.electronjs.org/docs/latest/api/context-bridge
- CWE-749: Exposed Dangerous Method or Function: https://cwe.mitre.org/data/definitions/749.html

Assessment: EMB-REQ-2 (JavaScript bridge input validation)

Reference: EMB-REQ-2 - All data crossing the JavaScript bridge shall be validated, sanitized, and type-checked on the native side

Given: A conformant embedded browser with JavaScript bridge capability (EMB-1 or higher)

Task: JavaScript bridge parameters originate from untrusted web content and can contain injection payloads, type confusion attacks, buffer overflows, path traversal sequences, or prototype pollution that exploit native code vulnerabilities. Without comprehensive validation, attackers can achieve SQL injection into native databases, command injection for arbitrary code execution, or path traversal to access sensitive files. Native-side validation with type checking, sanitization, range checking, and path canonicalization prevents injection attacks across the trust boundary.

Verification:

1. Identify all native APIs exposed via JavaScript bridge that accept parameters
2. For each API, review the native-side validation and sanitization logic
3. Create test cases with malicious inputs: SQL injection strings, path traversal sequences, command injection payloads, excessively long strings, null bytes, Unicode exploits
4. Invoke bridge APIs with malformed data: wrong types, missing parameters, extra parameters, null values, undefined values
5. Attempt to pass JavaScript objects with prototype pollution characteristics
6. Test that numeric parameters are range-checked and validated
7. Verify that file paths are canonicalized and validated before use
8. Confirm that validation failures trigger errors and logging rather than silent failures

9. All bridge API parameters undergo validation on the native side before processing
10. Type checking rejects parameters of incorrect types
11. String sanitization removes or escapes dangerous characters
12. Numeric parameters are range-checked
13. File paths are validated and canonicalized
14. Malicious inputs trigger validation errors without causing security issues
15. Validation failures are logged with sufficient detail for security monitoring
16. No SQL injection, command injection, or path traversal vulnerabilities exist

Pass Criteria: All bridge parameters are validated, sanitized, and type-checked AND malicious inputs are rejected safely AND validation failures are logged

Fail Criteria: Any injection vulnerability exists OR type checking is missing OR validation can be bypassed

Evidence: Code review showing validation logic, penetration test results, fuzzing test results, validation error logs, security assessment reports

References:

- OWASP Input Validation Cheat Sheet: https://cheatsheetseries.owasp.org/cheatsheets/Input_Validation_Cheat_Sheet.h
- CWE-20: Improper Input Validation: https://cwe.mitre.org/data/definitions/20.html
- CWE-89: SQL Injection: https://cwe.mitre.org/data/definitions/89.html
- CWE-78: OS Command Injection: https://cwe.mitre.org/data/definitions/78.html
- Android WebView Security Best Practices: https://developer.android.com/develop/ui/views/layout/webapps/webview#s

Assessment: EMB-REQ-3 (JavaScript bridge logging)

Reference: EMB-REQ-3 - JavaScript bridge communications shall be logged with sufficient detail for security auditing

Given: A conformant embedded browser with JavaScript bridge capability (EMB-1 or higher)

Task: JavaScript bridge logging is critical for detecting and investigating attacks where malicious web content attempts to exploit native APIs through the bridge. Without comprehensive audit logs, security teams cannot detect unauthorized API access, track attack patterns, or investigate security incidents. Proper logging of all bridge communications enables security monitoring systems to identify suspicious patterns such as repeated failed API calls, attempts to access restricted functionality, or unusual parameter values that may indicate injection attacks or reconnaissance activities.

Verification:

1. Configure the embedded browser to enable JavaScript bridge logging
2. Invoke various bridge APIs from web content
3. Review the generated logs and verify they contain: API name, timestamp, calling origin, parameters (with sensitive data redacted), return values, success/failure status
4. Attempt to call blocked or restricted APIs and verify denials are logged
5. Generate high-volume bridge traffic and verify logging continues without loss
6. Test that sensitive data (passwords, tokens) is redacted from logs
7. Verify logs are stored securely with appropriate access controls
8. Confirm log retention policies are documented and enforced
9. All JavaScript bridge API invocations are logged
10. Logs include sufficient context for security analysis (API, origin, timestamp)
11. Both successful and failed API calls are logged
12. Sensitive parameters are redacted or hashed in logs
13. Logs are tamper-resistant with integrity protection
14. Log retention policies balance security needs and privacy requirements
15. Logs are accessible to security monitoring systems

Pass Criteria: All bridge API calls are logged with sufficient detail AND sensitive data is redacted AND logs have integrity protection

Fail Criteria: API calls are not logged OR insufficient detail is recorded OR sensitive data appears in plaintext

Evidence: Log samples showing bridge API calls, log configuration files, security monitoring dashboards, log retention policy documentation

References:

- OWASP Logging Cheat Sheet: https://cheatsheetseries.owasp.org/cheatsheets/Logging_Cheat_Sheet.html
- NIST SP 800-92: Guide to Computer Security Log Management: https://csrc.nist.gov/publications/detail/sp/800-92/final
- CWE-778: Insufficient Logging: https://cwe.mitre.org/data/definitions/778.html

Assessment: EMB-REQ-4 (Context isolation)

Reference: EMB-REQ-4 - Embedded browsers shall implement context isolation to prevent web content from accessing host application objects

Given: A conformant embedded browser with context isolation capability (EMB-1 or higher)

Task: Context isolation is essential to prevent malicious web content from breaking out of its sandbox and directly accessing host application objects, which could lead to privilege escalation, data theft, or code execution in the host application context. Without proper isolation, attackers can enumerate the JavaScript global namespace to discover and exploit unintended host APIs, access sensitive application data structures, or manipulate host application behavior through prototype pollution. This boundary violation represents one of the most critical security threats in embedded browser architectures.

Verification:

1. Load web content in the embedded browser
2. From web content JavaScript, attempt to access global objects that might belong to the host: `window.hostApp`, `window.native`, `window.android`, `window.webkit`, etc.
3. Attempt to enumerate window properties to discover host-exposed objects
4. Test if web content can access Node.js globals (if applicable): `process`, `require`, `Buffer`, `global`
5. Verify that the JavaScript bridge API is the only official communication channel
6. Test that prototype pollution cannot affect host application objects
7. Confirm that context isolation is enforced even for trusted content origins
8. Verify that iframes inherit context isolation from the parent
9. Web content cannot access host application objects directly
10. Global namespace pollution from host application is prevented
11. Node.js globals (if applicable) are not accessible to web content
12. Only explicitly exposed JavaScript bridge APIs are accessible
13. Window property enumeration does not reveal internal objects
14. Prototype pollution attacks do not affect host application
15. Context isolation applies to all embedded content regardless of origin

Pass Criteria: Web content has no direct access to host objects AND only official bridge APIs are accessible

Fail Criteria: Any host application object is directly accessible from web content OR context isolation can be bypassed

Evidence: Penetration test results showing blocked access attempts, code review of context isolation implementation, test scripts attempting to access host objects, browser console showing security errors

References:

- Electron Context Isolation: https://www.electronjs.org/docs/latest/tutorial/context-isolation

- Chromium Isolated Worlds: https://chromium.googlesource.com/chromium/src/+/master/third_party/blink/renderer/b
  World
- iOS WKWebView JavaScript Isolation: https://developer.apple.com/documentation/webkit/wkwebview
- Android WebView Isolation: https://developer.android.com/reference/android/webkit/WebView

Assessment: EMB-REQ-5 (User consent for sensitive APIs)

Reference: EMB-REQ-5 - Native APIs exposed via JavaScript bridges shall require explicit user consent for sensitive operations

Given: A conformant embedded browser exposing sensitive APIs via JavaScript bridge (EMB-2 or higher)

Task: User consent for sensitive APIs is critical to prevent malicious web content from silently accessing privacy-sensitive functionality like camera, microphone, location, files, or contacts without the user's knowledge. Without explicit consent requirements, attackers can exploit JavaScript bridge APIs to exfiltrate sensitive data, track user location, or access private information. Proper consent mechanisms protect users from invisible attacks where compromised or malicious web content attempts unauthorized access to device capabilities that could violate user privacy or security expectations.

Verification:

1. Identify all sensitive operations exposed via the JavaScript bridge (e.g., file access, location, camera, contacts, payments)
2. Attempt to invoke sensitive APIs from web content without prior user consent
3. Verify that a clear, non-dismissible consent dialog appears for each sensitive operation
4. Test that consent dialogs clearly identify the requesting origin and the specific permission being requested
5. Verify that consent can be granted permanently, temporarily (session-only), or denied
6. Test that denied permissions are enforced consistently
7. Verify that users can revoke previously granted permissions
8. Confirm that consent state persists across browser sessions according to user choice
9. Sensitive API calls trigger user consent prompts before execution
10. Consent dialogs clearly identify the origin and permission type
11. Users can grant, deny, or revoke permissions
12. Denied permissions prevent API execution
13. Consent state is properly persisted and enforced
14. No sensitive operations occur without explicit user consent
15. Consent UI cannot be spoofed or bypassed by web content

Pass Criteria: All sensitive API calls require explicit user consent AND consent UI is clear and non-spoofable AND permission state is properly enforced

Fail Criteria: Any sensitive operation occurs without user consent OR consent UI is bypassable OR permissions are not enforced

Evidence: Screenshots of consent dialogs, test results showing blocked operations without consent, user consent audit logs, permission state persistence verification

References:

- Android Runtime Permissions: https://developer.android.com/training/permissions/requesting
- iOS Permission Prompts: https://developer.apple.com/design/human-interface-guidelines/patterns/accessing-private-data
- W3C Permissions API: https://www.w3.org/TR/permissions/

Assessment: EMB-REQ-6 (No system-level API exposure without controls)

Reference: EMB-REQ-6 - Host applications shall not expose system-level APIs (file system, process execution, native module loading) through JavaScript bridges without additional security controls

Given: A conformant embedded browser with JavaScript bridge capability (EMB-1 or higher)

Task: Exposing system-level APIs through JavaScript bridges without strict controls creates critical vulnerabilities that can lead to arbitrary code execution, data exfiltration, or complete system compromise. Malicious web content could exploit unrestricted file system access to read sensitive files, execute arbitrary commands to take control of the host system, or load native modules to escalate privileges. Path traversal attacks could bypass directory restrictions, and unbounded network socket access could enable attackers to pivot to internal systems, making proper security controls on system-level APIs absolutely essential for embedded browser security.

Verification:

1. Review all APIs exposed via the JavaScript bridge
2. Identify any system-level capabilities: file system access, process/command execution, native module loading, network socket access, registry access
3. For each system-level API, verify additional security controls exist: capability-based permissions, path allowlisting, command allowlisting, sandboxing, user consent
4. Attempt to use exposed file system APIs to access sensitive paths (/etc/passwd, C:\Windows\System32, application data directories)
5. Attempt to execute arbitrary commands or load arbitrary native modules
6. Test that path traversal sequences (../, ..\) are blocked or neutralized
7. Verify that system-level operations are logged extensively
8. Confirm that least-privilege principles are enforced (minimal necessary access)
9. System-level APIs either not exposed or have strict security controls
10. File system access is restricted to specific allowlisted directories
11. Command execution is restricted to specific allowlisted commands or completely prohibited
12. Native module loading is restricted or prohibited
13. Path traversal attacks are prevented
14. System-level operations require elevated permissions or user consent
15. All system-level operations are logged with full details
16. Security controls cannot be bypassed

Pass Criteria: No unrestricted system-level APIs are exposed AND all system APIs have additional security controls AND controls are enforced effectively

Fail Criteria: Any system-level API is exposed without restrictions OR security controls can be bypassed OR arbitrary file/command access is possible

Evidence: API documentation, code review of exposed APIs, penetration test results, logs showing blocked system access attempts, security architecture documentation

References:

- CWE-78: OS Command Injection: https://cwe.mitre.org/data/definitions/78.html
- CWE-22: Path Traversal: https://cwe.mitre.org/data/definitions/22.html
- OWASP Command Injection: https://owasp.org/www-community/attacks/Command_Injection
- Electron Security Best Practices - Shell Execution: https://www.electronjs.org/docs/latest/tutorial/security#6-do-not-use-shellexecuteshellopen-or-related-methods-with-user-content
- Android File System Security: https://developer.android.com/training/data-storage

Assessment: EMB-REQ-7 (Immutable bridge configuration)

Reference: EMB-REQ-7 - JavaScript bridge configurations shall be immutable after WebView initialization to prevent runtime tampering

Given: A conformant embedded browser with JavaScript bridge capability (EMB-1 or higher)

Task: Immutable bridge configuration prevents runtime tampering attacks where malicious code attempts to dynamically modify the JavaScript bridge to add new attack surfaces, remove security controls, or replace legitimate APIs with malicious implementations. If configuration can be changed after initialization, attackers could exploit race conditions, code injection vulnerabilities, or confused deputy scenarios to inject malicious

APIs, escalate privileges, or disable security restrictions. Configuration immutability ensures that the security boundary established at initialization time cannot be undermined during runtime.

Verification:

1. Initialize the embedded browser with a specific JavaScript bridge configuration
2. After initialization, attempt to modify the bridge configuration from host application code
3. From web content, attempt to add new APIs to the bridge dynamically
4. Test if existing bridge APIs can be removed or replaced at runtime
5. Verify that configuration changes after initialization either fail or require browser restart
6. Test that web content cannot influence bridge configuration through any means
7. Review code to confirm configuration is set during initialization phase only
8. Verify that attempting to modify configuration triggers security logging
9. Bridge configuration is set during initialization and cannot be changed afterward
10. Host application cannot add or remove bridge APIs after initialization
11. Web content cannot influence bridge configuration
12. Attempts to modify configuration fail with appropriate errors
13. Configuration immutability is enforced at the code level
14. Any attempted configuration changes are logged as security events
15. Browser restart is required for configuration changes (if supported at all)

Pass Criteria: Bridge configuration is immutable after initialization AND modification attempts fail AND attempts are logged

Fail Criteria: Configuration can be modified after initialization OR web content can influence configuration

Evidence: Code review showing configuration immutability enforcement, test results showing failed modification attempts, security logs showing configuration change attempts, API documentation

References:

- Android WebView Configuration Best Practices: https://developer.android.com/reference/android/webkit/WebSettings
- iOS WKWebView Configuration Immutability: https://developer.apple.com/documentation/webkit/wkwebviewconfigura
- Electron Security - Immutable Configuration: https://www.electronjs.org/docs/latest/tutorial/security
- CWE-732: Incorrect Permission Assignment: https://cwe.mitre.org/data/definitions/732.html

Assessment: EMB-REQ-8 (Host credential protection)

Reference: EMB-REQ-8 - Embedded browsers shall prevent web content from accessing host application credentials, tokens, or cryptographic keys

Given: A conformant embedded browser with context isolation (EMB-1 or higher)

Task: Protecting host credentials from web content access is essential to prevent credential theft attacks where malicious web content attempts to steal authentication tokens, API keys, or cryptographic keys stored by the host application. If web content can access host credentials through JavaScript bridge APIs, shared storage, or side channels, attackers can impersonate the host application, access protected resources, or decrypt sensitive data. This credential isolation requirement prevents scenarios where a compromised web page loaded in the embedded browser could steal the host's authentication credentials and use them for unauthorized access.

Verification:

1. Store test credentials/tokens in the host application's secure storage (keychain, keystore, credential manager)
2. Load web content in the embedded browser
3. Attempt to access host credentials through various methods: JavaScript bridge APIs, localStorage inspection, cookie inspection, memory inspection
4. Verify that credentials stored by the host are isolated from web content storage
5. Test that shared storage mechanisms (if any) do not leak credentials

6. Attempt to extract credentials by triggering host application features that use them
7. Verify that authentication tokens are not exposed in error messages or logs accessible to web content
8. Test that cryptographic operations using host keys cannot be triggered arbitrarily by web content
9. Host application credentials are completely isolated from web content
10. No JavaScript bridge API exposes credentials directly or indirectly
11. Web content storage (localStorage, cookies, IndexedDB) is separate from host credential storage
12. Shared storage mechanisms do not leak credentials
13. Authentication tokens are not exposed in error messages or debug output
14. Host cryptographic keys cannot be accessed or used arbitrarily by web content
15. Credential isolation is maintained even for trusted origins

Pass Criteria: Web content has no access to host credentials AND credential storage is completely isolated AND cryptographic keys are protected

Fail Criteria: Any credential or token is accessible to web content OR cryptographic keys can be misused

Evidence: Penetration test results, code review of credential storage isolation, test attempts to access credentials, memory dumps showing isolation, security audit reports

References:

- Android Keystore System: https://developer.android.com/training/articles/keystore
- iOS Keychain Services: https://developer.apple.com/documentation/security/keychain_services
- CWE-522: Insufficiently Protected Credentials: https://cwe.mitre.org/data/definitions/522.html

Assessment: EMB-REQ-9 (JavaScript bridge security review)

Reference: EMB-REQ-9 - All JavaScript bridge implementations shall be reviewed for injection vulnerabilities before production deployment

Given: A conformant embedded browser with JavaScript bridge capability (EMB-1 or higher)

Task: Security reviews for JavaScript bridge implementations are critical because bridges represent the most dangerous attack surface in embedded browsers, where improper input validation can lead to injection attacks, privilege escalation, or arbitrary code execution. Without formal security reviews, subtle vulnerabilities like SQL injection through bridge parameters, command injection in system-level APIs, or XSS through improperly sanitized return values can go undetected. Comprehensive security assessments by qualified professionals ensure that all bridge APIs are properly validated, sanitized, and protected against common attack vectors before they can be exploited in production.

Verification:

1. Obtain documentation of the security review process for JavaScript bridge implementations
2. Verify that a formal security code review was conducted before production deployment
3. Review security assessment reports for the JavaScript bridge implementation
4. Verify that common injection vulnerabilities were tested: SQL injection, command injection, path traversal, XSS, code injection
5. Confirm that automated security scanning tools were used (static analysis, dynamic analysis)
6. Verify that findings from security reviews were remediated before deployment
7. Confirm that security reviews are repeated after significant bridge changes
8. Verify that third-party security assessments were conducted (if applicable)
9. Documented security review process exists for JavaScript bridge code
10. Security code reviews were conducted by qualified security professionals
11. Common injection vulnerability types were specifically tested
12. Automated security scanning tools were applied
13. Security findings were tracked and remediated
14. Re-reviews occur after significant code changes
15. Security review reports are available for audit

Pass Criteria: Formal security review was conducted AND injection vulnerabilities were specifically tested AND findings were remediated

Fail Criteria: No security review was conducted OR injection vulnerabilities were not tested OR findings were not remediated

Evidence: Security review reports, code review checklists, penetration test reports, static analysis tool outputs, vulnerability remediation tracking records, security sign-off documentation

References:

- OWASP Code Review Guide: https://owasp.org/www-project-code-review-guide/
- CWE Top 25 Most Dangerous Software Weaknesses: https://cwe.mitre.org/top25/archive/2023/2023_top25_list.html
- NIST SP 800-53 - Security Assessment: https://csrc.nist.gov/publications/detail/sp/800-53/rev-5/final
- OWASP Mobile Application Security Verification Standard: https://github.com/OWASP/owasp-masvs

Assessment: EMB-REQ-10 (Bridge API rate limiting)

Reference: EMB-REQ-10 - Host applications shall implement rate limiting on JavaScript bridge API calls to prevent abuse

Given: A conformant embedded browser with JavaScript bridge capability (EMB-2 or higher)

Task: Rate limiting on JavaScript bridge APIs prevents abuse scenarios where malicious web content floods the bridge with excessive API calls to perform denial-of-service attacks, exhaust system resources, or brute-force security controls. Without rate limits, attackers can rapidly probe for vulnerabilities, overwhelm logging systems to hide their tracks, or cause performance degradation that impacts legitimate users. Proper rate limiting also prevents reconnaissance attacks where malicious scripts systematically enumerate all available APIs or attempt to bypass security controls through repeated high-frequency attempts.

Verification:

1. Identify rate limits for each JavaScript bridge API (requests per second, burst limits)
2. Create test scripts that invoke bridge APIs at high frequency
3. Verify that rate limits are enforced: excessive calls are rejected or throttled
4. Test that rate limit violations trigger logging and security alerts
5. Verify that rate limits are per-origin (different origins have independent quotas)
6. Test that rate limits reset appropriately (per second, per minute, per session)
7. Attempt to bypass rate limits by using multiple execution contexts (iframes, workers)
8. Verify that legitimate use cases are not blocked by rate limits
9. Rate limits are defined and documented for all bridge APIs
10. Excessive API calls are rejected or throttled
11. Rate limit violations trigger security logging and alerts
12. Rate limits are enforced per-origin or per-context
13. Rate limits reset on appropriate time boundaries
14. Multiple contexts cannot bypass rate limits
15. Legitimate high-frequency use cases are accommodated
16. Rate limit parameters are configurable by host application

Pass Criteria: Rate limits are enforced on all bridge APIs AND violations are logged AND limits cannot be bypassed

Fail Criteria: No rate limiting exists OR rate limits are bypassable OR legitimate use is blocked

Evidence: Rate limit configuration documentation, test results showing throttling behavior, security logs showing rate limit violations, performance test results

References:

- OWASP API Security - Rate Limiting: https://owasp.org/API-Security/editions/2023/en/0xa4-unrestricted-resource-consumption/

- CWE-770: Allocation of Resources Without Limits: https://cwe.mitre.org/data/definitions/770.html
- NIST SP 800-95: Guide to Secure Web Services: https://csrc.nist.gov/publications/detail/sp/800-95/final
- Rate Limiting Best Practices: https://cloud.google.com/architecture/rate-limiting-strategies-techniques

Assessment: EMB-REQ-11 (Granular capability-based permissions)

Reference: EMB-REQ-11 - JavaScript bridges shall support granular capability-based permissions rather than all-or-nothing access

Given: A conformant embedded browser with advanced bridge capabilities (EMB-2 or higher)

Task: Granular capability-based permissions prevent over-privileged access scenarios where granting access to one API inadvertently exposes all bridge functionality to potentially malicious web content. All-or-nothing permission models force developers to choose between denying all access or granting excessive privileges, creating security risks. Fine-grained permissions enable the principle of least privilege, allowing each origin to access only the specific APIs it needs while preventing lateral movement attacks where compromise of one feature leads to unauthorized access to unrelated sensitive functionality.

Verification:

1. Review the permission model for the JavaScript bridge
2. Verify that permissions can be granted at individual API level rather than all-or-nothing
3. Test different permission configurations: grant some APIs but deny others to the same origin
4. Verify that capability tokens or permissions can be scoped by origin, time, or usage count
5. Test that temporary permissions expire correctly
6. Verify that permission grants are logged with full context
7. Test that users or administrators can configure granular permissions
8. Verify that the principle of least privilege is enforced by default
9. Permission system supports individual API-level grants
10. Different origins can have different permission sets
11. Permissions can be scoped by time, usage count, or other constraints
12. Temporary permissions expire as configured
13. Permission grants and revocations are logged
14. Users/administrators have control over granular permissions
15. Default configuration follows least-privilege principle
16. Permission model is clearly documented

Pass Criteria: Granular per-API permissions are supported AND permissions are properly scoped AND least-privilege is enforced by default

Fail Criteria: Only all-or-nothing permissions exist OR no permission scoping is available OR overly permissive defaults

Evidence: Permission configuration files, permission model documentation, test results showing granular control, user/admin permission management interface screenshots, audit logs

References:

- Capability-Based Security: https://en.wikipedia.org/wiki/Capability-based_security
- Principle of Least Privilege (NIST): https://csrc.nist.gov/glossary/term/least_privilege
- W3C Permissions API: https://www.w3.org/TR/permissions/
- Android Permission Model: https://developer.android.com/guide/topics/permissions/overview

Assessment: EMB-REQ-12 (Storage isolation from host)

Reference: EMB-REQ-12 - Embedded browsers shall isolate storage (cookies, localStorage, IndexedDB) from the host application's native storage

Given: A conformant embedded browser (EMB-0 or higher)

Task: Storage isolation between embedded browsers and host applications prevents data leakage attacks where malicious web content attempts to access sensitive application data stored in native storage mechanisms. Without proper isolation, compromised web content could read authentication tokens from host storage, modify application configuration data, or exfiltrate user information. Storage namespace collisions could also enable web content to poison host application data, leading to privilege escalation or application malfunction. Proper isolation ensures that the web security model and native application security model remain separate and cannot be used to undermine each other.

Verification:

1. Store data in the host application's native storage (SharedPreferences, UserDefaults, native database)
2. Load web content and attempt to access host storage through web APIs (localStorage, IndexedDB, cookies)
3. Verify that web content cannot read native host storage
4. Store data in web content storage and attempt to access it from host application
5. Verify that host application code cannot directly access web storage without explicit bridge APIs
6. Test that different embedded browser instances have isolated storage
7. Verify that clearing host application data does not affect web storage (unless explicitly configured)
8. Test that malicious web content cannot pollute host storage namespace
9. Web content storage (localStorage, IndexedDB, cookies) is isolated from host native storage
10. Host native storage is not accessible from web APIs
11. Web storage is not directly accessible from host code (without explicit bridge)
12. Multiple embedded browser instances have independent storage
13. Storage isolation is maintained even for same-origin content in different instances
14. Storage clearing operations are properly scoped
15. No namespace collisions between web and native storage

Pass Criteria: Complete storage isolation between web content and host application AND no cross-contamination is possible

Fail Criteria: Web content can access host storage OR host can access web storage without proper APIs OR storage namespaces collide

Evidence: Code review of storage isolation implementation, test results showing blocked access attempts, storage dump comparisons, penetration test results

References:

- Android WebView Data Storage: https://developer.android.com/reference/android/webkit/WebView#getDataDir()
- Chromium Embedded Framework Storage Isolation: https://bitbucket.org/chromiumembedded/cef/wiki/GeneralUsage
- Web Storage API: https://html.spec.whatwg.org/multipage/webstorage.html

Assessment: EMB-REQ-13 (CSP enforcement for embedded content)

Reference: EMB-REQ-13 - Host applications shall implement Content Security Policy (CSP) enforcement for all embedded web content

Given: A conformant embedded browser (EMB-1 or higher)

Task: Content Security Policy enforcement for embedded web content provides defense-in-depth protection against XSS attacks, malicious script injection, and unauthorized resource loading that could be used to exfiltrate data or establish command-and-control channels. Without CSP, compromised or malicious web content can inject inline scripts, load external malicious resources, or use eval() to execute attacker-controlled code. Host-configured CSP ensures that even if web content is compromised, the attack surface is constrained by preventing dangerous JavaScript patterns and restricting where content can be loaded from.

Verification:

1. Configure the embedded browser with a restrictive CSP policy
2. Load web content and verify CSP is enforced: inline scripts blocked, external resource loading restricted

3. Test that CSP violations trigger browser console errors and violation reports
4. Verify that CSP cannot be bypassed by web content
5. Test that meta tag CSPs are respected but cannot weaken configured policy
6. Verify that CSP applies to all content including iframes
7. Test CSP report-uri or report-to functionality
8. Verify that host application can configure CSP policy programmatically
9. CSP policy is configurable by host application
10. CSP is enforced for all embedded web content
11. Inline scripts, eval, and other dangerous features are blocked per policy
12. CSP violations generate console errors and reports
13. Web content cannot bypass or weaken CSP policy
14. CSP applies to all nested contexts (iframes)
15. Violation reports are sent to configured endpoints
16. Default CSP is restrictive following security best practices

Pass Criteria: Host-configured CSP is enforced for all content AND violations are reported AND policy cannot be bypassed

Fail Criteria: CSP is not enforced OR can be bypassed OR violations are not reported

Evidence: CSP configuration code, browser console showing CSP violations, violation report logs, test results demonstrating CSP enforcement, penetration test results

References:

- Content Security Policy Level 3: https://www.w3.org/TR/CSP3/
- CSP Best Practices (OWASP): https://cheatsheetseries.owasp.org/cheatsheets/Content_Security_Policy_Cheat_Sheet.h
- Android WebView CSP: https://developer.android.com/reference/android/webkit/WebSettings#setMixedContentMode(i
- iOS WKWebView CSP Support: https://webkit.org/blog/7929/content-security-policy-for-webkit/

Assessment: EMB-REQ-14 (Encrypted cross-process bridge)

Reference: EMB-REQ-14 - JavaScript bridge communications shall be encrypted when crossing process boundaries

Given: A conformant embedded browser with multi-process architecture (EMB-2 or higher)

Task: Encrypted cross-process bridge communications protect against local privilege escalation attacks where a malicious process on the same system attempts to intercept or tamper with JavaScript bridge messages passing between the browser process and host application process. Without encryption, attackers with local access could use debugging tools, IPC monitoring utilities, or memory inspection to capture sensitive bridge data including authentication tokens, API parameters, or user data. Message integrity protection prevents tampering attacks, while anti-replay mechanisms prevent attackers from capturing and reusing legitimate bridge messages to perform unauthorized operations.

Verification:

1. Identify whether the embedded browser uses a multi-process architecture
2. Verify that JavaScript bridge messages cross process boundaries
3. Analyze inter-process communication (IPC) mechanisms used for bridge messages
4. Verify that IPC channels are encrypted or use secure transport
5. Attempt to intercept bridge messages using debugging tools or IPC monitoring
6. Verify that message integrity is protected (authentication, MACs)
7. Test that replay attacks are prevented (nonces, sequence numbers)
8. Verify that encryption keys are securely managed and rotated
9. Multi-process architecture uses encrypted IPC for bridge messages
10. Bridge messages are not observable in plaintext during IPC
11. Message integrity is cryptographically protected
12. Replay attacks are prevented through anti-replay mechanisms

13. Encryption keys are properly generated and managed
14. IPC interception does not reveal sensitive bridge data
15. Encryption meets current cryptographic standards (AES-256, authenticated encryption)

Pass Criteria: Bridge IPC is encrypted AND integrity-protected AND replay-protected when crossing process boundaries

Fail Criteria: Plaintext bridge messages observable in IPC OR no integrity protection OR replay attacks possible

Evidence: Architecture documentation showing multi-process design, IPC traces showing encrypted messages, code review of encryption implementation, penetration test results, cryptographic audit reports

References:

- Chromium Mojo IPC Security: https://chromium.googlesource.com/chromium/src/+/master/mojo/README.md
- Electron IPC Security: https://www.electronjs.org/docs/latest/tutorial/ipc
- NIST SP 800-52: TLS Guidelines: https://csrc.nist.gov/publications/detail/sp/800-52/rev-2/final
- CWE-319: Cleartext Transmission of Sensitive Information: https://cwe.mitre.org/data/definitions/319.html

Assessment: EMB-REQ-15 (Native UI overlay prevention)

Reference: EMB-REQ-15 - Embedded browsers shall prevent web content from triggering native UI overlays or permission prompts that could mislead users

Given: A conformant embedded browser (EMB-1 or higher)

Task: Native UI overlay prevention protects users from social engineering attacks where malicious web content attempts to display fake permission prompts, system dialogs, or security warnings that appear to originate from the operating system or host application. Without proper controls, attackers can create convincing spoofed UI elements to trick users into granting permissions, entering credentials, or taking dangerous actions. Preventing web content from triggering misleading native UI and ensuring clear origin identification helps users make informed trust decisions and prevents clickjacking attacks that overlay malicious content on top of legitimate security prompts.

Verification:

1. Attempt to trigger native UI elements from web content: system dialogs, permission prompts, notifications
2. Verify that all UI elements clearly identify their source (host app vs. web content)
3. Test that web content cannot create UI elements that appear to be from the OS or host app
4. Attempt clickjacking attacks using iframes and native UI
5. Verify that permission prompts clearly show the requesting origin
6. Test that web content cannot programmatically trigger permission prompts in rapid succession
7. Verify that native UI elements cannot be overlaid or obscured by web content
8. Test that fullscreen mode does not hide native UI security indicators
9. Web content cannot trigger misleading native UI elements
10. All UI clearly indicates whether it originates from web content or host/OS
11. Permission prompts show the requesting origin clearly
12. Clickjacking attempts are prevented by UI design
13. Rapid permission prompt abuse is prevented
14. Native security indicators cannot be hidden or spoofed
15. Fullscreen mode maintains critical security UI
16. Users can distinguish web content UI from native UI

Pass Criteria: Web content cannot trigger misleading native UI AND all prompts clearly show origin AND security indicators are protected

Fail Criteria: Misleading UI can be triggered OR origin is not clear OR security indicators can be hidden

Evidence: Screenshots of UI elements showing origin indicators, test results of UI spoofing attempts, clickjacking test results, user study results on UI clarity, security audit reports

References:

- OWASP Clickjacking Defense: https://cheatsheetseries.owasp.org/cheatsheets/Clickjacking_Defense_Cheat_Sheet.html
- Android UI Security: https://developer.android.com/topic/security/risks/tapjacking
- CWE-1021: Improper Restriction of Rendered UI Layers: https://cwe.mitre.org/data/definitions/1021.html

Assessment: EMB-REQ-16 (API surface allowlisting over denylisting)

Reference: EMB-REQ-16 - Host applications shall implement allowlists for JavaScript bridge API surface rather than denylists

Given: A conformant embedded browser with JavaScript bridge (EMB-1 or higher)

Task: Allowlist-based API surface control is critical for preventing unauthorized access to internal APIs that were never intended for web content exposure. Denylist approaches are inherently insecure because they require developers to anticipate and block every dangerous API, while inevitably missing newly added functionality or overlooking indirect access paths. With allowlisting, only explicitly registered APIs are accessible, ensuring that refactoring, new feature development, or internal API additions cannot accidentally expand the attack surface. This prevents attackers from discovering and exploiting unintended API endpoints through reflection, code analysis, or brute-force enumeration.

Verification:

1. Review the JavaScript bridge implementation to determine if it uses allowlist or denylist approach
2. Verify that only explicitly registered APIs are accessible (allowlist approach)
3. Attempt to discover and call APIs that are not in the allowlist
4. Test that new APIs require explicit registration before becoming accessible
5. Verify that the absence of an API from a denylist does not make it accessible
6. Review code to confirm allowlist-based access control logic
7. Test that reflection or introspection cannot enumerate unlisted APIs
8. Verify that the allowlist is the sole source of truth for API accessibility
9. JavaScript bridge uses allowlist-based access control
10. Only explicitly allowlisted APIs are accessible from web content
11. APIs not in allowlist are not accessible regardless of denylist status
12. New APIs require explicit allowlist registration
13. Denylists (if present) are used only as an additional layer, not primary control
14. Reflection/introspection cannot discover non-allowlisted APIs
15. Code review confirms allowlist-first architecture
16. Security documentation clearly describes allowlist approach

Pass Criteria: Allowlist-based access control is implemented AND only allowlisted APIs are accessible AND denylisting is not the primary control

Fail Criteria: Denylist-based access control is used as primary mechanism OR non-allowlisted APIs are accessible

Evidence: Code review showing allowlist implementation, security architecture documentation, test results showing blocked non-allowlisted API access, API registration procedures

References:

- OWASP Access Control: https://cheatsheetseries.owasp.org/cheatsheets/Access_Control_Cheat_Sheet.html
- Allowlist vs Denylist Best Practices: https://owasp.org/www-community/vulnerabilities/Improper_Data_Validation
- CWE-183: Permissive List of Allowed Inputs: https://cwe.mitre.org/data/definitions/183.html
- Electron Context Bridge Allowlisting: https://www.electronjs.org/docs/latest/api/context-bridge

Assessment: EMB-REQ-17 (Certificate validation for remote content)

Reference: EMB-REQ-17 - Embedded browsers shall validate SSL/TLS certificates for all remote content sources with certificate pinning for critical origins

Given: A conformant embedded browser loading remote content (EMB-1 or higher)

Task: Certificate validation and pinning for remote content protects against man-in-the-middle attacks where attackers intercept network traffic to inject malicious content into the embedded browser. Without proper certificate validation, attackers on compromised networks can present fraudulent certificates to serve malicious JavaScript that exploits bridge APIs or steals sensitive data. Certificate pinning for critical origins provides additional protection against certificate authority compromises and ensures that only the expected server certificates are trusted, preventing sophisticated attacks using rogue or compromised CAs to issue valid but malicious certificates for trusted domains.

Verification:

1. Configure the embedded browser to load content from HTTPS origins
2. Attempt to load content from a server with an invalid certificate (expired, self-signed, wrong hostname)
3. Verify that certificate validation errors block content loading
4. Test that user/host cannot bypass certificate errors for critical origins
5. Configure certificate pinning for critical origins used by the application
6. Attempt to use a valid but non-pinned certificate for a pinned origin
7. Verify that pinning violations block content loading and trigger alerts
8. Test that certificate transparency information is validated
9. Verify that pin backups exist to prevent lockout
10. All HTTPS content undergoes full certificate validation
11. Invalid certificates block content loading
12. Certificate errors generate clear error messages and logs
13. Certificate pinning is enforced for configured critical origins
14. Pinning violations prevent content loading and trigger security events
15. Backup pins exist to allow pin rotation without lockout
16. Certificate Transparency validation is performed
17. No user/host bypass options for certificate errors on critical origins

Pass Criteria: Full certificate validation is enforced AND certificate pinning works for critical origins AND no bypasses exist for pinned origins

Fail Criteria: Certificate validation can be bypassed OR pinning is not enforced OR no backup pins exist

Evidence: Certificate validation test results, pinning configuration, test results with invalid certificates, security logs showing validation failures, Certificate Transparency logs

References:

- RFC 6797: HTTP Strict Transport Security (HSTS): https://datatracker.ietf.org/doc/html/rfc6797
- OWASP Certificate Pinning: https://owasp.org/www-community/controls/Certificate_and_Public_Key_Pinning
- Android Network Security Config: https://developer.android.com/training/articles/security-config
- iOS App Transport Security: https://developer.apple.com/documentation/security/preventing_insecure_network_conne
- RFC 6962: Certificate Transparency: https://datatracker.ietf.org/doc/html/rfc6962

Assessment: EMB-REQ-18 (Trusted origin allowlisting)

Reference: EMB-REQ-18 - Host applications shall implement allowlists of trusted content origins and reject content from unapproved sources

Given: A conformant embedded browser (EMB-1 or higher)

Task: Trusted origin allowlisting prevents the embedded browser from loading malicious or untrusted content that could exploit JavaScript bridge APIs or compromise user data. Without origin restrictions, the browser

could be tricked into loading attacker-controlled content through phishing links, open redirects, or DNS rebinding attacks. Enforcing a strict allowlist ensures that only vetted, trusted origins can execute in the privileged embedded browser context, preventing attackers from injecting malicious web pages that would have access to sensitive bridge functionality or user information intended only for legitimate application content.

Verification:

1. Review the host application's origin allowlist configuration
2. Attempt to load content from allowlisted origins and verify it loads successfully
3. Attempt to load content from non-allowlisted origins and verify it is blocked
4. Test redirect chains that exit allowlisted domains and verify they are blocked
5. Verify that allowlist enforcement applies to main frame, iframes, and XHR/fetch requests
6. Test that URL parameters or fragments cannot be used to bypass allowlist
7. Verify that allowlist configuration is immutable by web content
8. Test that allowlist violations trigger logging and security alerts
9. Origin allowlist is configured and documented
10. Only allowlisted origins can load in the embedded browser
11. Non-allowlisted origins are blocked with appropriate error messages
12. Redirect chains exiting allowlisted domains are blocked
13. Allowlist applies to all resource types and contexts
14. URL manipulation cannot bypass allowlist enforcement
15. Allowlist violations generate security events and logs
16. Users/administrators can configure the allowlist

Pass Criteria: Origin allowlist is enforced for all content AND violations are blocked and logged AND allowlist cannot be bypassed

Fail Criteria: Non-allowlisted origins can load OR allowlist can be bypassed OR violations not logged

Evidence: Origin allowlist configuration, test results with blocked non-allowlisted origins, security logs showing violations, redirect chain test results, penetration test reports

References:

- OWASP Content Security Policy: https://owasp.org/www-community/controls/Content_Security_Policy
- Android WebView URL Filtering: https://developer.android.com/reference/android/webkit/WebViewClient#shouldOver
- iOS WKNavigationDelegate: https://developer.apple.com/documentation/webkit/wknavigationdelegate
- CSP frame-ancestors directive: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy/frame-ancestors

Assessment: EMB-REQ-19 (Subresource Integrity for external scripts)

Reference: EMB-REQ-19 - Embedded browsers shall enforce Subresource Integrity (SRI) for all external scripts loaded by trusted content

Given: A conformant embedded browser loading trusted content with external dependencies (EMB-2 or higher)

Task: Subresource Integrity enforcement protects against supply chain attacks where compromised CDNs or third-party script providers serve malicious code to embedded browsers. Without SRI, attackers who compromise external script sources can inject malicious JavaScript that would execute with full access to JavaScript bridge APIs and user data. SRI ensures that external scripts match their expected cryptographic hashes, preventing execution of modified or substituted content even if the hosting server is compromised, providing critical defense-in-depth protection for embedded browsers that load resources from external origins.

Verification:

1. Configure trusted content pages to load external scripts from CDNs

2. Add SRI integrity attributes to script tags: `<script src="..." integrity="sha384-..." crossorigin="anonymous">`
3. Verify that scripts with correct integrity hashes load successfully
4. Modify the external script content and verify that integrity validation fails and blocks execution
5. Attempt to load scripts without integrity attributes from external origins and verify policy enforcement
6. Test that integrity validation applies to CSS, scripts, and other subresources
7. Verify that SRI failures trigger console errors and CSP violation reports
8. Test that the host application can enforce SRI policy via CSP require-sri-for directive
9. SRI integrity checking is enforced for external scripts
10. Scripts with correct integrity hashes load and execute
11. Scripts with incorrect or missing hashes are blocked per policy
12. SRI violations generate console errors and logs
13. SRI applies to scripts, stylesheets, and other subresources
14. Host application can mandate SRI via CSP policy
15. Cryptographic hash algorithms used are secure (SHA-384, SHA-512)

Pass Criteria: SRI is enforced for external scripts AND integrity violations block resource loading AND policy can be mandated by host

Fail Criteria: SRI is not enforced OR can be bypassed OR weak hash algorithms are accepted

Evidence: HTML source showing SRI attributes, browser console showing integrity check results, test results with modified scripts, CSP headers mandating SRI, security logs

References:

- Subresource Integrity specification: https://www.w3.org/TR/SRI/
- SRI Best Practices (MDN): https://developer.mozilla.org/en-US/docs/Web/Security/Subresource_Integrity
- CSP require-sri-for: https://www.w3.org/TR/CSP3/#directive-require-sri-for
- OWASP Secure Headers Project: https://owasp.org/www-project-secure-headers/

Assessment: EMB-REQ-20 (Certificate pinning with backup pins)

Reference: EMB-REQ-20 - Certificate pinning configurations shall include backup pins and pin rotation mechanisms to prevent lockout

Given: A conformant embedded browser with certificate pinning (EMB-2 or higher)

Task: Backup pins and pin rotation mechanisms are essential to prevent application lockout scenarios where certificate rotation or CA changes leave deployed applications unable to connect to pinned origins. Without backup pins, a single certificate expiration or key compromise could render all deployed applications non-functional until users install updates. Proper pin management with backup pins and documented rotation procedures ensures that certificate pinning provides strong security against MITM attacks while maintaining operational resilience, preventing scenarios where security controls inadvertently cause widespread application failures that force emergency unpinning and reduce overall security posture.

Verification:

1. Review certificate pinning configuration for all pinned origins
2. Verify that at least one backup pin exists for each pinned origin
3. Test that the primary pin is enforced correctly
4. Simulate primary certificate rotation and verify backup pin is accepted
5. Verify that pin rotation procedures are documented
6. Test that pin expiration dates are configured appropriately (not too distant future)
7. Verify that pin updates can be delivered to deployed applications
8. Test fallback behavior when all pins fail (should block, not fall back to unpinned)
9. Each pinned origin has at least one backup pin configured
10. Primary and backup pins are both enforced correctly
11. Certificate rotation can occur using backup pins without lockout

12. Pin rotation procedures are documented and tested
13. Pin expiration/rotation schedules are reasonable
14. Pin updates can be delivered via app updates or dynamic configuration
15. Total pin failure blocks connections rather than falling back to no pinning
16. Pins include Subject Public Key Info (SPKI) hashes, not just certificates

Pass Criteria: Backup pins exist for all pinned origins AND pin rotation is supported AND lockout scenarios are prevented

Fail Criteria: No backup pins exist OR pin rotation is not supported OR lockout is possible OR insecure fallback behavior

Evidence: Certificate pinning configuration files, pin backup documentation, pin rotation test results, pin expiration monitoring, security architecture documentation

References:

- OWASP Certificate Pinning Cheat Sheet: https://cheatsheetseries.owasp.org/cheatsheets/Pinning_Cheat_Sheet.html
- RFC 7469: Public Key Pinning Extension for HTTP: https://datatracker.ietf.org/doc/html/rfc7469
- Android Network Security Config - Pin Backup: https://developer.android.com/training/articles/security-config#CertificatePinning
- iOS Certificate Pinning Best Practices: https://developer.apple.com/news/?id=g9ejcf8y

Assessment: EMB-REQ-21 (Mixed content prevention)

Reference: EMB-REQ-21 - Embedded browsers shall prevent mixed content (HTTPS pages loading HTTP resources) in all configurations

Given: A conformant embedded browser (EMB-1 or higher)

Task: Mixed content prevention protects against downgrade attacks where attackers intercept insecure HTTP resources loaded by HTTPS pages to inject malicious code or steal sensitive data. Even a single HTTP script loaded by an HTTPS page undermines the entire security model, allowing network attackers to execute arbitrary JavaScript with full access to the page's privileges and JavaScript bridge APIs. Blocking mixed content ensures that the security guarantees of HTTPS connections cannot be bypassed through insecure subresources, preventing attackers from using unencrypted channels to compromise otherwise secure embedded browser sessions.

Verification:

1. Load an HTTPS page in the embedded browser
2. Attempt to load HTTP subresources from the HTTPS page: images, scripts, stylesheets, XHR, fetch
3. Verify that passive mixed content (images, media) is blocked or triggers warnings
4. Verify that active mixed content (scripts, stylesheets, iframes) is always blocked
5. Test that mixed content blocking cannot be disabled by web content
6. Verify that host application cannot weaken mixed content protection
7. Test that browser console shows clear mixed content errors
8. Verify that CSP upgrade-insecure-requests directive is supported
9. Active mixed content is always blocked
10. Passive mixed content is blocked or triggers prominent warnings
11. Mixed content errors appear in browser console with details
12. Web content cannot disable mixed content protection
13. Host application cannot weaken mixed content blocking
14. upgrade-insecure-requests CSP directive automatically upgrades HTTP to HTTPS
15. No configuration option exists to allow mixed content on HTTPS pages

Pass Criteria: Active mixed content is always blocked AND passive mixed content protection is enforced AND cannot be disabled

Fail Criteria: Mixed content loads successfully OR protection can be disabled OR insufficient warnings

Evidence: Browser console showing mixed content errors, test results with blocked HTTP resources, CSP header analysis, security configuration review, network traffic captures

References:

- W3C Mixed Content specification: https://www.w3.org/TR/mixed-content/
- Mixed Content (MDN): https://developer.mozilla.org/en-US/docs/Web/Security/Mixed_content
- CSP upgrade-insecure-requests: https://www.w3.org/TR/upgrade-insecure-requests/
- Chromium Mixed Content Policy: https://www.chromium.org/Home/chromium-security/deprecating-powerful-features-on-insecure-origins

Assessment: EMB-REQ-22 (Trust decision logging)

Reference: EMB-REQ-22 - Trust decisions shall be logged with sufficient detail to detect trust boundary violations

Given: A conformant embedded browser with trust management (EMB-1 or higher)

Task: Trust decision logging is critical for detecting and investigating attacks that attempt to cross trust boundaries in embedded browsers, such as untrusted origins trying to access privileged JavaScript bridge APIs or certificate validation bypasses. Without comprehensive logging of trust decisions, security teams cannot identify patterns of attack attempts, investigate security incidents, or detect compromised trusted origins that begin exhibiting suspicious behavior. Detailed trust logs enable security monitoring systems to flag anomalies like repeated origin allowlist violations, certificate pinning failures, or unusual patterns of bridge API access that may indicate reconnaissance or active exploitation attempts.

Verification:

1. Configure logging for trust-related decisions in the embedded browser
2. Load content from trusted and untrusted origins
3. Review logs and verify they contain: origin, timestamp, trust decision outcome, reason for decision
4. Attempt trust boundary violations (untrusted origin accessing privileged APIs)
5. Verify that violations are logged with full context
6. Test that certificate validation decisions are logged
7. Verify that origin allowlist enforcement is logged
8. Test that logs are stored securely and cannot be tampered with by web content
9. Verify log retention policies for security-relevant events
10. All trust decisions are logged with sufficient detail
11. Logs include origin, timestamp, decision, and rationale
12. Trust boundary violations are logged with full context
13. Certificate validation results are logged
14. Origin allowlist enforcement is logged
15. Logs are protected from tampering
16. Log retention balances security needs and privacy
17. Logs are accessible for security monitoring and incident response

Pass Criteria: All trust decisions are logged with sufficient detail AND violations are logged AND logs are tamper-protected

Fail Criteria: Trust decisions are not logged OR insufficient detail OR logs can be tampered with

Evidence: Log samples showing trust decisions, log configuration documentation, test results showing violation logging, log integrity verification, log retention policy documentation

References:

- OWASP Logging Cheat Sheet: https://cheatsheetseries.owasp.org/cheatsheets/Logging_Cheat_Sheet.html
- NIST SP 800-92: Guide to Computer Security Log Management: https://csrc.nist.gov/publications/detail/sp/800-92/final
- CWE-778: Insufficient Logging: https://cwe.mitre.org/data/definitions/778.html

Assessment: EMB-REQ-23 (Cryptographic signature verification for local content)

Reference: EMB-REQ-23 - Embedded browsers loading local/bundled content shall verify cryptographic signatures before execution

Given: A conformant embedded browser loading bundled/local content (EMB-2 or higher)

Task: Cryptographic signature verification for local content protects against tampering attacks where attackers with file system access attempt to modify bundled HTML, JavaScript, or CSS files to inject malicious code. Without signature verification, local privilege escalation or compromised backup/restore processes could allow attackers to silently modify embedded browser content that executes with full JavaScript bridge API access. Digital signatures ensure content integrity from build time through deployment, detecting any unauthorized modifications and preventing execution of tampered local resources that could otherwise compromise the entire host application through bridge exploitation.

Verification:

1. Identify local/bundled content loaded by the embedded browser (HTML, JS, CSS files)
2. Verify that all bundled content is cryptographically signed
3. Review signature verification implementation in the browser initialization code
4. Attempt to modify bundled content and verify that signature verification fails
5. Test that signature verification failures prevent content loading
6. Verify that signature verification uses secure algorithms (RSA-2048+, ECDSA P-256+)
7. Test that signature verification cannot be bypassed or disabled
8. Verify that signing keys are properly protected in the host application
9. All local/bundled content has cryptographic signatures
10. Signature verification occurs before content execution
11. Modified content fails signature verification
12. Signature verification failures block content loading
13. Secure signature algorithms are used
14. Signature verification cannot be bypassed
15. Signing keys are protected from extraction
16. Verification process is logged

Pass Criteria: All local content is signed AND signatures are verified before execution AND verification cannot be bypassed

Fail Criteria: Unsigned content can be loaded OR signature verification is bypassable OR weak signature algorithms

Evidence: Signed content bundles, signature verification code review, test results with modified content, cryptographic algorithm analysis, key protection documentation

References:

- Code Signing Best Practices (NIST): https://csrc.nist.gov/publications/detail/sp/800-183/final
- Android APK Signature Scheme: https://source.android.com/docs/security/features/apksigning
- Authenticode for Windows: https://docs.microsoft.com/en-us/windows-hardware/drivers/install/authenticode

Assessment: EMB-REQ-24 (Redirect chain trust enforcement)

Reference: EMB-REQ-24 - Host applications shall implement mechanisms to detect and prevent redirect chains that exit trusted domains

Given: A conformant embedded browser with origin allowlist (EMB-1 or higher)

Task: Redirect chain trust enforcement prevents open redirect attacks where trusted origins inadvertently redirect to attacker-controlled domains, potentially exposing JavaScript bridge APIs to malicious content. Without redirect validation, attackers can exploit legitimate redirects on trusted sites to bypass origin allowlists, tricking the embedded browser into loading malicious content that appears to come from a

trusted source. Multi-hop redirect chains are particularly dangerous because each hop in the chain could be manipulated to eventually land on an untrusted domain, requiring comprehensive validation of the entire redirect path to prevent trust boundary violations.

Verification:

1. Configure trusted origin allowlist in the embedded browser
2. Load a page from a trusted origin that redirects to another trusted origin
3. Verify that trusted-to-trusted redirects are allowed
4. Load a page from a trusted origin that redirects to an untrusted origin
5. Verify that trusted-to-untrusted redirects are blocked
6. Test multi-hop redirect chains: trusted → trusted → untrusted
7. Verify that all redirects in the chain are validated against allowlist
8. Test that redirect blocking triggers security logging
9. Verify that JavaScript-based redirects are also subject to trust enforcement
10. Redirect chain validation is enforced
11. Trusted-to-trusted redirects are allowed
12. Any redirect to untrusted origin is blocked
13. Multi-hop redirect chains are fully validated
14. Both HTTP and JavaScript redirects are subject to enforcement
15. Blocked redirects trigger error pages and security logs
16. Users are notified when redirects are blocked
17. No bypass mechanism exists for redirect enforcement

Pass Criteria: Redirect chains are fully validated AND untrusted redirects are blocked AND enforcement covers all redirect types

Fail Criteria: Redirects to untrusted origins succeed OR JavaScript redirects bypass enforcement OR multi-hop validation is incomplete

Evidence: Redirect test results, browser console showing blocked redirects, security logs, network traffic captures showing redirect chains, error page screenshots

References:

- OWASP Open Redirect: https://cheatsheetseries.owasp.org/cheatsheets/Unvalidated_Redirects_and_Forwards_Cheat_
- CWE-601: URL Redirection to Untrusted Site: https://cwe.mitre.org/data/definitions/601.html
- Android WebView shouldOverrideUrlLoading: https://developer.android.com/reference/android/webkit/WebViewClient
- iOS WKNavigationDelegate decidePolicyFor: https://developer.apple.com/documentation/webkit/wknavigationdelegate/webview

Assessment: EMB-REQ-25 (HSTS enforcement for trusted origins)

Reference: EMB-REQ-25 - Embedded browsers shall enforce strict transport security (HSTS) for all trusted content origins

Given: A conformant embedded browser (EMB-1 or higher)

Task: HSTS enforcement for trusted origins protects against SSL stripping attacks where network attackers intercept initial HTTP requests and prevent upgrade to HTTPS, allowing interception of all subsequent traffic. Without HSTS, embedded browsers are vulnerable during the brief window between application launch and the first HTTPS connection, when attackers can downgrade connections to HTTP and conduct man-in-the-middle attacks. HSTS ensures that all connections to trusted origins use HTTPS, with non-bypassable certificate validation, preventing network-level attackers from intercepting or modifying content loaded by the embedded browser even on compromised networks.

Verification:

1. Configure trusted content origins with HSTS headers: `Strict-Transport-Security: max-age=31536000; includeSubDomains; preload`

2. Make initial HTTPS request to trusted origin and verify HSTS header is received and cached
3. Attempt to make HTTP request to same origin and verify automatic upgrade to HTTPS
4. Test that HSTS enforcement continues after browser restart (persistence)
5. Verify that HSTS includeSubDomains flag is respected for subdomains
6. Test that HSTS preload list is supported and enforced
7. Attempt to bypass HSTS through direct IP access or hosts file manipulation
8. Verify that certificate errors on HSTS hosts cannot be bypassed
9. HSTS headers are parsed and cached correctly
10. HTTP requests to HSTS hosts are automatically upgraded to HTTPS
11. HSTS state persists across browser sessions
12. includeSubDomains flag applies to all subdomains
13. HSTS preload list is maintained and enforced
14. HSTS cannot be bypassed through IP access or other means
15. Certificate errors on HSTS hosts are non-bypassable
16. HSTS enforcement is logged

Pass Criteria: HSTS is enforced for all trusted origins AND HTTP is automatically upgraded AND certificate errors are non-bypassable

Fail Criteria: HSTS can be bypassed OR HTTP connections succeed to HSTS hosts OR certificate errors are bypassable

Evidence: Network traffic captures showing HTTPS upgrades, HSTS cache inspection, test results with HTTP requests, certificate error bypass attempts, HSTS preload list verification

References:

- RFC 6797: HTTP Strict Transport Security (HSTS): https://datatracker.ietf.org/doc/html/rfc6797
- HSTS Preload List: https://hstspreload.org/
- OWASP HSTS: https://cheatsheetseries.owasp.org/cheatsheets/HTTP_Strict_Transport_Security_Cheat_Sheet.html
- Chromium HSTS Implementation: https://www.chromium.org/hsts

Assessment: EMB-REQ-26 (Certificate validation failure notification)

Reference: EMB-REQ-26 - Certificate validation failures for trusted content sources shall trigger immediate user notification and content blocking

Given: A conformant embedded browser loading trusted remote content (EMB-1 or higher)

Task: Immediate notification and blocking on certificate validation failures for trusted content prevents silent man-in-the-middle attacks where users unknowingly interact with compromised content. Without clear error notifications and strict blocking, users might not realize that certificate validation has failed, allowing attackers to serve malicious content that appears to come from trusted origins. Non-bypassable certificate error handling for trusted origins ensures that even sophisticated attacks using fraudulent certificates cannot trick the embedded browser into loading untrusted content, protecting the integrity of the JavaScript bridge API access model.

Verification:

1. Configure trusted content origins with valid certificates
2. Simulate certificate validation failures: expired certificate, hostname mismatch, untrusted CA, revoked certificate
3. Verify that each failure type triggers immediate error notification
4. Verify that content loading is completely blocked (no partial loading)
5. Test that error notifications clearly explain the problem to users
6. Verify that no “continue anyway” option exists for trusted origin failures
7. Test that certificate failures are logged to security event log
8. Verify that host application is notified of certificate failures via callback
9. Certificate validation failures are detected for all failure types

10. User receives immediate, clear error notification
11. Content loading is completely blocked on validation failure
12. Error messages explain the specific certificate problem
13. No bypass mechanism exists for certificate errors on trusted origins
14. Certificate failures are logged with full details
15. Host application can register for certificate failure callbacks
16. Failure handling differs for trusted vs. untrusted origins

Pass Criteria: All certificate failures block content AND trigger immediate user notification AND are non-bypassable for trusted origins

Fail Criteria: Any certificate failure allows content loading OR notifications are missing/unclear OR bypass options exist

Evidence: Error page screenshots, certificate failure logs, test results with various failure types, user notification interface documentation, host callback implementation

References:

- RFC 5280: Certificate Validation: https://datatracker.ietf.org/doc/html/rfc5280
- CA/Browser Forum Baseline Requirements: https://cabforum.org/baseline-requirements-documents/
- Android WebView onReceivedSslError: https://developer.android.com/reference/android/webkit/WebViewClient#onRec
- iOS WKNavigationDelegate didReceive challenge: https://developer.apple.com/documentation/webkit/wknavigationdeleg webview

Assessment: EMB-REQ-27 (Network security configuration)

Reference: EMB-REQ-27 - Host applications shall implement network security configurations that prevent cleartext traffic to trusted domains

Given: A conformant embedded browser (EMB-1 or higher)

Task:

1. Review network security configuration for the embedded browser (Android Network Security Config, iOS App Transport Security, or equivalent)
2. Verify that cleartext HTTP traffic is disabled for trusted domains
3. Attempt to make HTTP requests to trusted domains and verify they are blocked
4. Test that HTTPS-only enforcement cannot be disabled at runtime
5. Verify that network security configuration applies to all network APIs (WebView, XHR, fetch, WebSocket)
6. Test that certificate pinning configuration is integrated with network security config
7. Verify that configuration is declarative and cannot be modified by web content
8. Test that cleartext traffic blocking is logged

Verification:

- Network security configuration enforces HTTPS-only for trusted domains
- Cleartext HTTP traffic is blocked at the network layer
- HTTPS enforcement applies to all network APIs uniformly
- Configuration is declarative and immutable at runtime
- Certificate pinning is integrated into network security config
- Blocked cleartext traffic is logged
- Configuration is documented and auditable
- Platform-specific security features are properly utilized (NSConfig, ATS)

Pass Criteria: Cleartext traffic to trusted domains is blocked at network layer AND configuration is immutable AND applies to all network APIs

Fail Criteria: Cleartext traffic succeeds to trusted domains OR configuration can be modified OR inconsistent enforcement across APIs

Evidence: Network security configuration files, network traffic captures showing blocked HTTP, test results with various network APIs, configuration documentation, platform security feature usage audit

References:

- Android Network Security Configuration: https://developer.android.com/training/articles/security-config
- iOS App Transport Security: https://developer.apple.com/documentation/security/preventing_insecure_network_conne
- CWE-319: Cleartext Transmission of Sensitive Information: https://cwe.mitre.org/data/definitions/319.html

Assessment: EMB-REQ-28 (CSP enforcement for third-party content)

Reference: EMB-REQ-28 - Embedded browsers shall prevent trusted content from loading untrusted third-party content without explicit CSP declarations

Given: A conformant embedded browser with CSP support (EMB-1 or higher)

Task:

1. Configure trusted content page with restrictive CSP that allows only specific third-party domains
2. Attempt to load third-party content from CSP-allowed domains and verify success
3. Attempt to load third-party content from non-allowed domains and verify blocking
4. Test that inline scripts from third-party content are blocked unless allowed by CSP
5. Verify that CSP applies to all third-party resource types (scripts, styles, images, frames)
6. Test that CSP violations are reported via report-uri or report-to
7. Verify that web content cannot weaken or disable CSP policy
8. Test that host-configured CSP takes precedence over page-specified CSP

Verification:

- CSP restricts third-party content loading to explicitly allowed domains
- Non-allowed third-party content is blocked
- CSP applies consistently across all resource types
- CSP violations generate reports to configured endpoint
- Web content cannot bypass or weaken CSP restrictions
- Host-configured CSP cannot be overridden by page CSP
- CSP enforcement is logged
- Default CSP policy is restrictive (deny by default)

Pass Criteria: Third-party content loading is restricted by CSP AND violations are blocked and reported AND policy cannot be weakened by web content

Fail Criteria: Third-party content bypasses CSP OR policy can be weakened OR violations not reported

Evidence: CSP policy headers, browser console showing CSP violations, violation reports, test results with blocked third-party content, network traffic analysis

References:

- Content Security Policy Level 3: https://www.w3.org/TR/CSP3/
- CSP script-src directive: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy/script-src
- CSP Reporting: https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP#reporting
- OWASP CSP Cheat Sheet: https://cheatsheetseries.owasp.org/cheatsheets/Content_Security_Policy_Cheat_Sheet.html

Assessment: EMB-REQ-29 (Per-instance trust policies)

Reference: EMB-REQ-29 - Trust policies shall be configurable per embedded browser instance to support multi-tenant applications

Given: A conformant embedded browser supporting multiple instances (EMB-2 or higher)

Task:

1. Create multiple embedded browser instances within the same host application
2. Configure different trust policies for each instance: different origin allowlists, different certificate pins, different CSP policies
3. Load content in each instance and verify that instance-specific policies are enforced
4. Verify that instances cannot access each other's storage or state
5. Test that trust policy configuration is isolated per instance
6. Attempt to modify one instance's trust policy from another instance
7. Verify that instance destruction cleans up trust policy state
8. Test that host application can dynamically create instances with custom policies

Verification:

- Trust policies can be configured independently per instance
- Each instance enforces its own trust policy correctly
- Instances are isolated from each other (no cross-instance access)
- Trust policy configuration is scoped to individual instances
- Instance creation accepts custom policy configuration
- Instance destruction cleans up policy state
- No interference between instances' trust enforcement
- Multi-tenant scenarios are properly supported

Pass Criteria: Per-instance trust policies are supported AND enforced correctly AND instances are isolated from each other

Fail Criteria: Policies cannot be configured per-instance OR instances interfere with each other OR policy isolation is broken

Evidence: Code showing per-instance policy configuration, test results with multiple instances, storage isolation verification, policy enforcement logs per instance, multi-tenant test scenarios

References:

- Android WebView Multiple Instances: https://developer.android.com/reference/android/webkit/WebView
- iOS WKWebView Configuration: https://developer.apple.com/documentation/webkit/wkwebviewconfiguration
- Electron BrowserView Isolation: https://www.electronjs.org/docs/latest/api/browser-view
- Multi-tenancy Security Best Practices: https://owasp.org/www-project-application-security-verification-standard/

Assessment: EMB-REQ-30 (Certificate Transparency verification)

Reference: EMB-REQ-30 - Embedded browsers shall implement certificate transparency verification for trusted content origins

Given: A conformant embedded browser with Certificate Transparency support (EMB-2 or higher)

Task:

1. Configure trusted content origins that require Certificate Transparency
2. Load content from origins with valid SCTs (Signed Certificate Timestamps) in certificates or via TLS extension
3. Verify that content loads successfully when valid SCTs are present
4. Attempt to load content with certificates lacking SCTs from CT-required origins

5. Verify that missing or invalid SCTs block content loading for CT-required origins
6. Test that SCTs are validated against known CT logs
7. Verify that CT enforcement can be configured per-origin or globally
8. Test that CT validation failures are logged

Verification:

- Certificate Transparency verification is supported
- Valid SCTs allow content loading from CT-required origins
- Missing or invalid SCTs block content loading
- SCTs are validated against known CT logs
- CT enforcement policy is configurable (per-origin or global)
- CT validation failures generate clear error messages and logs
- CT log list is kept up-to-date
- CT enforcement integrates with certificate validation flow

Pass Criteria: CT verification is implemented AND enforced for configured origins AND invalid SCTs block content loading

Fail Criteria: CT verification not implemented OR can be bypassed OR invalid SCTs are accepted

Evidence: Network traffic captures showing SCTs, CT validation logs, test results with missing SCTs, CT log list version verification, CT enforcement configuration

References:

- RFC 6962: Certificate Transparency: https://datatracker.ietf.org/doc/html/rfc6962
- Chromium CT Policy: https://www.chromium.org/Home/chromium-security/certificate-transparency/
- Certificate Transparency Logs: https://www.certificate-transparency.org/
- Google CT Policy: https://github.com/GoogleChrome/CertificateTransparency/blob/master/ct_policy.md

Assessment: EMB-REQ-31 (DNS rebinding attack prevention)

Reference: EMB-REQ-31 - Host applications shall detect and prevent DNS rebinding attacks targeting trusted local/internal origins

Given: A conformant embedded browser with local/internal origin access (EMB-2 or higher)

Task:

1. Identify local/internal origins accessed by the embedded browser (localhost, 127.0.0.1, private IP ranges, .local domains)
2. Simulate DNS rebinding attack: external domain initially resolves to attacker IP, then rebinds to local/internal IP
3. Verify that DNS rebinding is detected and blocked
4. Test DNS cache behavior and TTL enforcement
5. Verify that private IP address access requires explicit allowlisting
6. Test that DNS resolution results are validated against expected address ranges
7. Verify that Host header validation prevents rebinding exploitation
8. Test that DNS rebinding attempts trigger security logging

Verification:

- DNS rebinding attacks are detected
- Unexpected DNS resolution changes trigger blocking
- Private IP address access requires explicit configuration
- DNS cache prevents rapid rebinding
- Host header validation is enforced
- Resolution results are validated against address allowlists
- Rebinding attempts are logged as security events

- Local/internal origins have additional protection

Pass Criteria: DNS rebinding attacks are detected and blocked AND private IP access is restricted AND rebinding attempts are logged

Fail Criteria: DNS rebinding succeeds OR no private IP restrictions OR rebinding not detected

Evidence: DNS rebinding test results, DNS cache configuration, Host header validation tests, security logs showing rebinding attempts, private IP access policy documentation

References:

- DNS Rebinding Explained: https://en.wikipedia.org/wiki/DNS_rebinding
- CWE-346: Origin Validation Error: https://cwe.mitre.org/data/definitions/346.html
- Private Network Access specification: https://wicg.github.io/private-network-access/

Assessment: EMB-REQ-32 (Trust boundary violation security events)

Reference: EMB-REQ-32 - Trust boundary violations shall trigger security events with sufficient context for incident response

Given: A conformant embedded browser with comprehensive logging (EMB-1 or higher)

Task:

1. Configure security event logging for the embedded browser
2. Trigger various trust boundary violations: accessing blocked origins, calling restricted APIs, certificate validation failures, CSP violations
3. Review generated security events and verify they contain: timestamp, event type, origin/source, detailed description, severity level, stack trace/context
4. Verify that security events are distinct from general application logs
5. Test that security events can be forwarded to SIEM or security monitoring systems
6. Verify that high-severity violations trigger immediate alerts
7. Test that security events include sufficient context for incident investigation
8. Verify that event generation does not disrupt application functionality

Verification:

- Trust boundary violations generate security events
- Events include comprehensive context: timestamp, type, origin, description, severity
- Events are distinguishable from general logs (separate channel or marked)
- Events can be exported or forwarded to security monitoring systems
- High-severity events trigger immediate alerts
- Events include actionable information for incident response
- Event logging is reliable and does not impact app stability
- Events are tamper-resistant

Pass Criteria: All boundary violations generate security events AND events include sufficient context AND can be forwarded to monitoring systems

Fail Criteria: Violations don't generate events OR insufficient context OR cannot be monitored externally

Evidence: Security event log samples, event schema documentation, SIEM integration configuration, alert configuration, incident response playbooks referencing events, event log integrity verification

References:

- NIST SP 800-61: Computer Security Incident Handling Guide: https://csrc.nist.gov/publications/detail/sp/800-61/rev-2/final
- OWASP Logging Cheat Sheet: https://cheatsheetseries.owasp.org/cheatsheets/Logging_Cheat_Sheet.html
- CEF (Common Event Format): https://www.microfocus.com/documentation/arcsight/arcsight-smartconnectors/pdfdoc/common-event-format-v25/common-event-format-v25.pdf

- SIEM Integration Best Practices: https://www.sans.org/white-papers/36427/

Annex A (informative): Mapping between the present document and CRA requirements

Table mapping technical security requirements from Section 5 of the present document to essential cybersecurity requirements in Annex I of the CRA. The purpose of this is to help identify missing technical security requirements.

CRA requirement	Technical security requirements(s)
No known exploitable vulnerabilities	UPD-0-REQ-1 through UPD-0-REQ-24 (Forced automatic updates), UPD-1-REQ-1 through UPD-1-REQ-25 (Automatic updates with postponement), LOG-REQ-14 (Incident detection), LOG-REQ-15 (Audit trail completeness), EMB-REQ-9 (JavaScript bridge security review), EXT-REQ-4 (Manifest validation), EXT-REQ-17 (Extension signature validation)
Secure design, development, production	EMB-REQ-9 (JavaScript bridge security review), EXT-REQ-4 (Manifest validation), EXT-REQ-17 (Extension signature validation), UPD-REQ-2 (Update signature verification), UPD-REQ-23 (Binary reproducibility), ENC-REQ-12 (Secure random number generation), SYS-REQ-26 (Sandbox escape prevention), SYS-REQ-27 (Spectre/Meltdown mitigations), SYS-REQ-28 (Side-channel mitigations)
Secure by default configuration	DOM-0-REQ-1 through DOM-0-REQ-6 (Full isolation by default), ENC-0-REQ-1 through ENC-0-REQ-23 (Full encryption by default), DOM-REQ-5 (SameSite=Lax default), DOM-REQ-12 (document.domain restricted by default), ENC-REQ-16 (HTTPS-first mode), UPD-0-REQ-1 (Automatic updates enabled by default), LOG-REQ-9 (User consent for telemetry), SYS-0-REQ-1 through SYS-0-REQ-13 (Sandboxed by default), EMB-0-REQ-1 through EMB-0-REQ-7 (Isolated by default)

CRA requirement	Technical security requirements(s)
Secure updates	UPD-REQ-1 (Automatic update mechanism), UPD-REQ-2 (Update signature verification), UPD-REQ-3 (HTTPS-only delivery), UPD-REQ-4 (Manifest integrity), UPD-REQ-5 (Rollback protection), UPD-REQ-6 (Channel isolation), UPD-REQ-7 (Component updates), UPD-REQ-8 (Emergency updates), UPD-REQ-9 (Verification before installation), UPD-REQ-10 (Failure recovery), UPD-REQ-11 (Transparency logging), UPD-REQ-12 (Delta update security), UPD-REQ-13 (Server authentication), UPD-REQ-14 (Timing jitter), UPD-REQ-15 (Background enforcement), UPD-REQ-16 (Notification UI), UPD-REQ-17 (Forced critical updates), UPD-REQ-18 (Verification chain), UPD-REQ-19 (Source pinning), UPD-REQ-20 (Integrity verification), UPD-REQ-21 (Staged rollout), UPD-REQ-22 (Domain validation), UPD-REQ-23 (Binary reproducibility), EXT-REQ-10 (Extension update verification)
Authentication and access control mechanisms	DOM-REQ-1 (Process-per-site isolation), DOM-REQ-3 (Cross-origin DOM access prevention), DOM-REQ-4 (CORS preflight), DOM-REQ-5 (SameSite cookies), DOM-REQ-6 (Storage isolation), EXT-REQ-1 (Extension permission model), EXT-REQ-3 (Extension API access control), EXT-REQ-7 (Host permissions validation), SYS-REQ-6 (Device API permissions), SYS-REQ-7 (PWA permission management), SYS-REQ-8 through SYS-REQ-19 (Device-specific permissions), EMB-REQ-1 (JavaScript bridge API allowlists), EMB-REQ-5 (User consent for sensitive operations), EMB-REQ-11 (Granular capability-based permissions), PRO-REQ-2 (User consent for custom protocols), PRO-REQ-3 (Protocol allowlist enforcement)

CRA requirement	Technical security requirements(s)
Confidentiality protection	ENC-REQ-1 (TLS 1.3+ support), ENC-REQ-2 (Certificate validation), ENC-REQ-3 (Certificate pinning), ENC-REQ-4 (HSTS enforcement), ENC-REQ-5 (Mixed content blocking), ENC-REQ-6 (Certificate Transparency), ENC-REQ-11 (Web Crypto API), ENC-REQ-13 (Subresource Integrity), ENC-REQ-14 (Encrypted SNI/ECH), ENC-REQ-16 (HTTPS-first mode), ENC-REQ-20 (Cryptographic key isolation), ENC-REQ-21 (Certificate store security), DOM-REQ-2 (CORB), DOM-REQ-6 (Storage isolation), EMB-REQ-4 (Context isolation), EMB-REQ-8 (Host credential protection), EMB-REQ-12 (Storage isolation from host), EMB-REQ-14 (Encrypted cross-process bridge), EMB-REQ-17 (Certificate validation for embedded content), EMB-REQ-21 (Mixed content prevention), EMB-REQ-27 (Network security configuration)
Integrity protection for data and configuration	ENC-REQ-2 (Certificate validation), ENC-REQ-13 (Subresource Integrity), UPD-REQ-2 (Update signature verification), UPD-REQ-4 (Update manifest integrity), UPD-REQ-5 (Rollback protection), UPD-REQ-20 (Update integrity verification), LOG-REQ-11 (Log integrity protection), EMB-REQ-2 (JavaScript bridge input validation), EMB-REQ-7 (Immutable bridge configuration), EMB-REQ-19 (SRI for embedded content), EMB-REQ-23 (Cryptographic signature verification for local content), EXT-REQ-4 (Manifest validation), EXT-REQ-17 (Extension signature validation), DOM-REQ-9 (CORP), DOM-REQ-11 (COEP)
Data minimization	LOG-REQ-7 (Log data minimization), LOG-REQ-8 (Log anonymization), LOG-REQ-12 (Log retention policies), LOG-REQ-18 (Privacy-preserving analytics), EXT-REQ-16 (Extension telemetry privacy), DOM-REQ-6 (Storage isolation limits data sharing), EMB-REQ-12 (Storage isolation from host), PRO-REQ-5 (Protocol parameter sanitization to prevent data leakage)

CRA requirement	Technical security requirements(s)
Availability protection	SYS-REQ-20 (Hardware resource limits), SYS-REQ-21 (Memory isolation), SYS-REQ-22 (CPU quotas), SYS-REQ-23 (Network bandwidth limits), SYS-REQ-24 (Storage quotas), SYS-REQ-25 (Process priority management), UPD-REQ-10 (Update failure recovery), UPD-REQ-21 (Staged rollout), EMB-REQ-10 (Bridge API rate limiting), EXT-REQ-5 (Extension sandboxing to prevent interference)
Minimize impact on other devices or services	DOM-REQ-1 (Process-per-site isolation), SYS-REQ-1 (Process sandbox enforcement), SYS-REQ-2 (Renderer process isolation), SYS-REQ-3 (GPU process isolation), SYS-REQ-4 (Network service isolation), SYS-REQ-20 (Resource limits), SYS-REQ-26 (Sandbox escape prevention), EXT-REQ-5 (Extension sandboxing), EXT-REQ-6 (Cross-extension isolation), EMB-REQ-4 (Context isolation), PRO-REQ-13 (Handler capability restrictions)
Limit attack surface	EXT-0-REQ-1 through EXT-0-REQ-3 (No extension support), SYS-0-REQ-1 through SYS-0-REQ-13 (Fully sandboxed), EMB-0-REQ-1 through EMB-0-REQ-7 (No JavaScript bridge), PRO-0-REQ-1 through PRO-0-REQ-5 (HTTP/HTTPS only), DOM-REQ-7 (iframe sandboxing), DOM-REQ-8 (Opaque origin handling), EMB-REQ-6 (No system-level API exposure), EMB-REQ-16 (Allowlists over denylists), EXT-REQ-12 (Background script restrictions), ENC-REQ-19 (Legacy crypto deprecation)
Exploit mitigation by limiting incident impact	DOM-REQ-1 (Process-per-site isolation limits cross-site impact), SYS-REQ-1 (Sandbox enforcement), SYS-REQ-2 (Process isolation), SYS-REQ-21 (Memory isolation), SYS-REQ-26 (Sandbox escape prevention), SYS-REQ-27 (Spectre/Meltdown mitigations), SYS-REQ-28 (Side-channel mitigations), DOM-REQ-2 (CORB), DOM-REQ-9 (CORP), DOM-REQ-10 (COOP), DOM-REQ-11 (COEP), EXT-REQ-2 (Content script isolation), EXT-REQ-5 (Extension sandboxing), EXT-REQ-6 (Cross-extension isolation), EMB-REQ-4 (Context isolation), UPD-REQ-5 (Rollback protection), UPD-REQ-21 (Staged rollout limits blast radius)

CRA requirement	Technical security requirements(s)
Logging and monitoring mechanisms	LOG-REQ-1 (Security event logging), LOG-REQ-2 (Certificate error logging), LOG-REQ-3 (Extension security events), LOG-REQ-4 (CSP violation reporting), LOG-REQ-5 (Network Error Logging), LOG-REQ-6 (Crash reporting), LOG-REQ-10 (Secure log transmission), LOG-REQ-11 (Log integrity protection), LOG-REQ-13 (Security dashboard), LOG-REQ-14 (Incident detection), LOG-REQ-15 (Audit trail completeness), LOG-REQ-16 (Real-time security alerts), LOG-REQ-17 (Forensic log export), LOG-REQ-19 (Compliance logging), LOG-REQ-20 (Log access controls), EMB-REQ-3 (JavaScript bridge logging), EMB-REQ-22 (Trust decision logging), EMB-REQ-32 (Trust boundary violation events), UPD-REQ-11 (Update transparency logging), PRO-REQ-9 (Protocol handler logging)
Secure deletion and data transfer	DOM-REQ-6 (Storage isolation enables secure per-origin deletion), ENC-REQ-1 (TLS 1.3+ for secure transfer), ENC-REQ-3 (Certificate pinning for critical transfers), ENC-REQ-5 (Mixed content blocking), EMB-REQ-17 (Certificate validation for embedded content transfers), EMB-REQ-21 (Mixed content prevention), EMB-REQ-27 (Network security configuration), SYS-REQ-24 (Storage quotas with cleanup mechanisms), EXT-REQ-11 (Extension storage isolation enables clean uninstall)

Annex B (informative): Mapping of Use Cases to Capabilities and Requirements

This annex provides a comprehensive mapping of each use case defined in Section 4.4 to the relevant browser capabilities and their associated requirement sets. This mapping helps manufacturers and assessors identify which requirements apply to specific deployment contexts.

B.1 Use Case Mapping Methodology

For each use case, the mapping identifies:

1. **Primary Capabilities**: Core security capabilities that are essential for the use case
2. **Recommended Condition Levels**: Specific condition levels (e.g., DOM-1, EXT-2) appropriate for the use case's risk profile
3. **Critical Requirements**: Specific requirement sets that are satisfied for the use case
4. **Optional Enhancements**: Additional requirements that may be appropriate based on deployment specifics

B.2 Use Case to Capability Mappings

UC-B1: General Purpose Web Browsing (Risk Level: Standard)

Primary Capabilities and Recommended Conditions:

- **DOM (Domain/Origin Isolation)**: DOM-1 (Controlled isolation)
- **EXT (Extension System)**: EXT-1 or EXT-2
- **ENC (Encryption)**: ENC-1
- **LOG (Logging/Monitoring)**: LOG-1
- **UPD (Updates)**: UPD-1
- **PRO (Protocol Handlers)**: PRO-1
- **SYS (System Resources)**: SYS-1

Critical Requirements: DOM-1-REQ-1 through DOM-1-REQ-9, ENC-1-REQ-1 through ENC-1-REQ-19, UPD-1-REQ-1 through UPD-1-REQ-25, EXT-1-REQ-1 through EXT-1-REQ-14, LOG-1-REQ-1 through LOG-1-REQ-18, SYS-1-REQ-1 through SYS-1-REQ-22

Assessment References: All DOM, ENC, UPD, EXT, LOG, PRO, SYS assessments apply

---

UC-B2: Development and Testing Environments (Risk Level: High)

Primary Capabilities and Recommended Conditions:

- **DOM**: DOM-2
- **EXT**: EXT-2
- **ENC**: ENC-1
- **LOG**: LOG-2
- **UPD**: UPD-1 or UPD-2
- **PRO**: PRO-2
- **SYS**: SYS-2

Critical Requirements: DOM-2-REQ-1 through DOM-2-REQ-12, EXT-2-REQ-1 through EXT-2-REQ-10, LOG-2-REQ-1 through LOG-2-REQ-20, PRO-2-REQ-1 through PRO-2-REQ-12, SYS-2-REQ-1 through SYS-2-REQ-15

Assessment References: All capability assessments, emphasis on EXT-REQ-9, DOM-REQ-9-11, SYS-REQ-14-17

UC-B3: Kiosks and Shared Terminals (Risk Level: High)

Primary Capabilities and Recommended Conditions:

- **DOM**: DOM-0 or DOM-1
- **EXT**: EXT-0
- **ENC**: ENC-0 or ENC-1
- **LOG**: LOG-3
- **UPD**: UPD-0
- **PRO**: PRO-0
- **SYS**: SYS-0
- **EMB** (if embedded): EMB-1 or EMB-2

Critical Requirements: DOM-0-REQ-1 through DOM-0-REQ-6, EXT-0-REQ-1 through EXT-0-REQ-3, ENC-0-REQ-1 through ENC-0-REQ-23, LOG-3-REQ-1 through LOG-3-REQ-20, UPD-0-REQ-1 through UPD-0-REQ-24, PRO-0-REQ-1 through PRO-0-REQ-5, SYS-0-REQ-1 through SYS-0-REQ-13

If Embedded: EMB-1-REQ-1 through EMB-1-REQ-17, EMB-REQ-8, EMB-REQ-3, EMB-REQ-22

Additional: Domain allowlisting, session auto-termination, no credential storage, remote monitoring

Assessment References: Strictest criteria; DOM-REQ-1-8, ENC-REQ-1-6, UPD-REQ-1-11, SYS-REQ-1-4, LOG-REQ-10-11

UC-B4: Financial Services Access (Risk Level: High)

Primary Capabilities and Recommended Conditions:

- **DOM**: DOM-1
- **EXT**: EXT-1
- **ENC**: ENC-0 or ENC-1
- **LOG**: LOG-1
- **UPD**: UPD-0 or UPD-1
- **PRO**: PRO-1
- **SYS**: SYS-1
- **EMB** (if embedded): EMB-1 or EMB-2

Critical Requirements: ENC-0-REQ-1 through ENC-0-REQ-23 OR ENC-1-REQ-1 through ENC-1-REQ-19, DOM-1-REQ-1 through DOM-1-REQ-9, EXT-1-REQ-1 through EXT-1-REQ-14, LOG-REQ-2, LOG-REQ-14

If Embedded: EMB-1-REQ-1 through EMB-1-REQ-17, EMB-REQ-17, EMB-REQ-20, EMB-REQ-2, EMB-REQ-8

Assessment References: ENC-REQ-1-7, ENC-REQ-17, DOM-REQ-5, LOG-REQ-2, EMB-REQ-1-10 (if embedded)

UC-B5: Healthcare and Medical Systems (Risk Level: High)

Primary Capabilities and Recommended Conditions:

- **DOM**: DOM-1 or DOM-2
- **EXT**: EXT-1
- **ENC**: ENC-0
- **LOG**: LOG-3
- **UPD**: UPD-0 or UPD-1
- **PRO**: PRO-1
- **SYS**: SYS-1
- **EMB** (if embedded): EMB-1 or EMB-2

Critical Requirements: ENC-0-REQ-1 through ENC-0-REQ-23, LOG-3-REQ-1 through LOG-3-REQ-20, LOG-REQ-7, LOG-REQ-8, LOG-REQ-19, DOM-1-REQ-1 through DOM-1-REQ-9, UPD-0-REQ-17, EXT-1-REQ-1 through EXT-1-REQ-14

If Embedded: EMB-1-REQ-1 through EMB-1-REQ-17, EMB-REQ-8, EMB-REQ-3, EMB-REQ-22

Compliance: GDPR data protection, session re-auth, auto-timeout, audit trails

Assessment References: ENC-REQ-1-7, LOG-REQ-7-9, LOG-REQ-19, LOG-REQ-11, EMB-REQ-3, EMB-REQ-22 (if embedded)

UC-B6: E-Government Services Access (Risk Level: High)

Primary Capabilities and Recommended Conditions:

- **DOM**: DOM-1
- **EXT**: EXT-1
- **ENC**: ENC-0
- **LOG**: LOG-2 or LOG-3
- **UPD**: UPD-0 or UPD-1
- **PRO**: PRO-1
- **SYS**: SYS-1 or SYS-2

Critical Requirements: ENC-0-REQ-1 through ENC-0-REQ-23, ENC-REQ-2, ENC-REQ-3, SYS-REQ-29, DOM-1-REQ-1 through DOM-1-REQ-9, LOG-REQ-15, UPD-0-REQ-1 through UPD-0-REQ-24

Special: Digital signatures, smart card integration, eIDAS compliance, legal non-repudiation

Assessment References: ENC-REQ-2-3, ENC-REQ-6, SYS-REQ-29, LOG-REQ-15

UC-B7: Enterprise Applications (Risk Level: High)

Primary Capabilities and Recommended Conditions:

- **DOM**: DOM-2
- **EXT**: EXT-1 or EXT-2
- **ENC**: ENC-1
- **LOG**: LOG-3
- **UPD**: UPD-0 or UPD-1
- **PRO**: PRO-2
- **SYS**: SYS-2
- **EMB** (if Electron/CEF/Tauri): EMB-2 or EMB-3

Critical Requirements: DOM-2-REQ-1 through DOM-2-REQ-12, LOG-3-REQ-1 through LOG-3-REQ-20, LOG-REQ-13, LOG-REQ-14, LOG-REQ-16, EXT-2-REQ-10, UPD-0-REQ-1 through UPD-0-REQ-24, PRO-2-REQ-11, SYS-2-REQ-11 through SYS-2-REQ-15

Enterprise Features: SSO, DLP, extension allowlisting, profile separation, BYOD containerization

Assessment References: DOM-REQ-9-11, LOG-REQ-13-16, LOG-REQ-19, EXT-REQ-3, PRO-REQ-3, SYS-REQ-7

UC-B8: Critical Infrastructure (Risk Level: CRITICAL)

Primary Capabilities and Recommended Conditions:

- **DOM**: DOM-0 or DOM-1
- **EXT**: EXT-0
- **ENC**: ENC-0
- **LOG**: LOG-3
- **UPD**: UPD-0
- **PRO**: PRO-0 or PRO-1
- **SYS**: SYS-0 or SYS-1
- **EMB** (if SCADA/ICS): EMB-0 or EMB-1

Critical Requirements: DOM-0-REQ-1 through DOM-0-REQ-6, EXT-0-REQ-1 through EXT-0-REQ-3, ENC-0-REQ-1 through ENC-0-REQ-23, LOG-3-REQ-1 through LOG-3-REQ-20, LOG-REQ-11, UPD-0-REQ-1 through UPD-0-REQ-24, UPD-REQ-5, UPD-REQ-11, PRO-0-REQ-1 through PRO-0-REQ-5, SYS-0-REQ-1 through SYS-0-REQ-13

If Embedded: EMB-0-REQ-1 through EMB-0-REQ-7, EMB-REQ-17, EMB-REQ-20, EMB-REQ-27, EMB-REQ-31

Additional: Zero trust, mTLS, RBAC, air-gapped deployment, supply chain controls, physical security

Assessment References: ALL assessments at strictest criteria; ENC-REQ-1-11, UPD-REQ-1-11, LOG-REQ-10-11, SYS-REQ-26-28, EMB-REQ-17-31 (if embedded)

UC-B9: Security Research (Risk Level: CRITICAL)

Primary Capabilities and Recommended Conditions:

- **DOM**: DOM-2 or DOM-3
- **EXT**: EXT-3
- **ENC**: ENC-1
- **LOG**: LOG-3
- **UPD**: UPD-2 or UPD-3
- **PRO**: PRO-3
- **SYS**: SYS-3

Critical Requirements: LOG-3-REQ-1 through LOG-3-REQ-20, LOG-REQ-6, LOG-REQ-17, DOM-3-REQ-9, EXT-3-REQ-9 through EXT-3-REQ-12, SYS-3-REQ-15

Environment Isolation: Disposable VMs, network capture, air-gapped zones, snapshot/rollback, behavioral logging

Important: Deploy ONLY in isolated research environments; NOT for production

Assessment References: LOG-REQ-1-20, LOG-REQ-17, EXT-REQ-4, all assessments in adversarial conditions

---

UC-B10: Adapted Browser with Modified Features (Risk Level: Standard to High)

Primary Capabilities and Recommended Conditions:

- **All capabilities**: Inherit from upstream browser
- **LOG**: May vary (LOG-1, LOG-2, or LOG-3)
- **UPD**: UPD-0 or UPD-1 (manufacturer-controlled)
- **EMB** (if native integration added): EMB-2 or EMB-3

Critical Requirements: All upstream requirements PLUS UPD-REQ-2, UPD-REQ-11, EMB-REQ-9, LOG-REQ-9, LOG-REQ-7, LOG-REQ-8

If Native Integration Added: EMB-2-REQ-1 through EMB-2-REQ-10 OR EMB-3-REQ-1 through EMB-3-REQ-12, EMB-REQ-1, EMB-REQ-2, EMB-REQ-3, EMB-REQ-9

For Bundled Extensions: EXT-REQ-4, EXT-REQ-17, supply chain security

Manufacturer Obligations: Timely upstream patches, security review, transparency, maintaining security controls, supply chain security

Risk Level: Standard (minimal modifications) to High (extensive modifications, sensitive data, high-risk deployment)

Assessment References: All upstream assessments PLUS EMB-REQ-9, UPD-REQ-2, LOG-REQ-9, EMB assessments if native integration

---

B.3 Capability Condition Level Selection Guide

Use Case Risk	DOM	EXT	ENC	LOG	UPD	PRO	SYS	EMB
Standard	DOM-1	EXT-1/2	ENC-1	LOG-1	UPD-1	PRO-1	SYS-1	EMB-1
High	DOM-1/2	EXT-0/1	ENC-0/1	LOG-2/3	UPD-0/1	PRO-0/1	SYS-0/1/2	EMB-0/1/2
Critical	DOM-0/1	EXT-0	ENC-0	LOG-3	UPD-0	PRO-0	SYS-0/1	EMB-0/1

Note: Specific deployments shall conduct detailed risk assessments per Annex D to determine appropriate condition levels.

B.4 Cross-Reference to Assessments

All assessments in Chapter 6 map to requirements referenced in this annex:

- **Section 6.1**: DOM-REQ-1 through DOM-REQ-12
- **Section 6.2**: EXT-REQ-1 through EXT-REQ-18
- **Section 6.3**: ENC-REQ-1 through ENC-REQ-21
- **Section 6.4**: LOG-REQ-1 through LOG-REQ-20
- **Section 6.5**: UPD-REQ-1 through UPD-REQ-23
- **Section 6.6**: PRO-REQ-1 through PRO-REQ-23
- **Section 6.7**: SYS-REQ-1 through SYS-REQ-32
- **Section 6.8**: EMB-REQ-1 through EMB-REQ-32

Annex C (informative): Relationship between the present document and any related ETSI standards (if any)

List any related ETSI standards and how they interact with the present document.

Annex D (informative): Risk identification and assessment methodology

C.1 Assets

C.1.1 Data

What data is stored on the product?

C.1.2 Product functions

See the functions in Section 4.4.

C.2 Threats

Based on the assets, what are the threats during:

- *Use for intended purpose or reasonably foreseeable use*
- *When integrated into another product*

Example threats can be found in the same documents suggested in the section on security requirements.

C.3 Assumptions

List assumptions that are relevant to the risk analysis for these threats. Everything is hackable if you try hard enough. What kinds of threats are in and out of scope? What are you assuming is the sophistication of attack? Relate to use cases. Some examples might include:

- *Not being attacked by a state actor*
- *Not using sophisticated or expensive hardware snooping techniques*
- *No secret hardware backdoors in other components*

C.4 Risk assessments of threats

For each threat identified above, use likelihood and magnitude of the threat to assess its risk in the context of use cases. The results should be consistent with the mapping of use cases to security levels.

Guidance from latest PT1 draft:

> *An analysis in terms of likelihood and magnitude of a product's threats is required to be able to determine the product's risks. NOTE 1 This document does not require a specific methodology for a cybersecurity risk analysis as long as the cybersecurity risk estimation is based on the likelihood of occurrence and magnitude of loss or disruption of cybersecurity risks. Thus, different approaches and models such as the fishbone model, event tree analysis or fault tree models can be used within the analysis of cybersecurity risks. NOTE 2 A qualitative estimation of the cybersecurity risks can be performed using risk matrices that map qualitative categories of the likelihood of occurrence and qualitative categories of magnitude of loss or disruption to cybersecurity risk categories. NOTE 3 A quantitative estimation of the cybersecurity risks can be performed using scoring systems that map qualitative categories of the likelihood of occurrence and qualitative categories of magnitude of loss or disruption to certain values.*

Annex E (informative): Risk evaluation guidance

E.1 Mapping of risks to requirements

Table mapping the identified risks to requirements

E.2 Risks not treated by the requirements

If any risks are not treated by the normative requirements, describe non-normative suggestions to mitigate them.

E.3 Risk acceptance criteria

Describe how to decide if residual risks are tolerable.

E.4 Residual risks

Describe how to treat any residual risks, for example by documenting them or informing the user.

Annex J

- potential vulnerability
- discovered vulnerability
- known vulnerability
- publicly known vulnerability

- actively exploited vulnerability
- exploited vulnerability
- exploitable vulnerability
- known exploitable vulnerability
- known newly emerged vulnerability
- notified vulnerability
- AI specific vulnerability
- fixed vulnerability
- https://wpt.fyi/results/cors?label=stable&label=master&aligned
- https://html5test.co/
- https://caniuse.com/
- https://chromium.googlesource.com/chromium/src/+/HEAD/docs/security/faq.md
- https://chromium.googlesource.com/chromium/src/+/master/docs/security/compromised-renderers.md

Annex K

Crypto todo

https://certification.enisa.europa.eu/publications/eucc-guidelines-cryptography_en

Annex L (informative): Relationship between the present document and the requirements of EU Regulation 2024/2847

DRAFT ANNEX L - DO NOT CONSIDER THE CONTENT

The present document has been prepared under the Commission's standardisation request C(2025) 618 final to provide one voluntary means of conforming to the requirements of Regulation (EU) No 2024/2847 of the European Parliament and of the Council of 23 October 2024 on horizontal cybersecurity requirements for products with digital elements and amending Regulations (EU) No 168/2013 and (EU) No 2019/1020 and Directive (EU) 2020/1828 (Cyber Resilience Act). Once the present document is cited in the Official Journal of the European Union under that Regulation, compliance with the normative clauses of the present document given in table A.1 confers, within the limits of the scope of the present document, a presumption of conformity with the corresponding requirements of that Regulation and associated EFTA regulations. > NOTE: The above paragraphs have to be repeated in the Foreword.

The annex shall have a table for a clear indication of correspondence between normative clauses of the standard and the legal requirements aimed to be covered.

It should be evaluated - on the basis of the legal requirements supported and other information given in a harmonised standard - how detailed correspondence can be indicated between the normative elements of the harmonised standard and the legal requirements aimed to be covered. However, where this correspondence is expressed in too general terms, it could lead to a situation where the Commission cannot assess whether the Harmonised Standard satisfies the requirements, which it aims to cover, and subsequently publication of its references in the OJEU according to Article 10(6) of the Regulation is significantly delayed or is not possible at all.

Annex : Change history

Date	Version	Information about changes
<Month year>	<#>	<Changes made are listed in this cell>

History

Version	Date	Milestone
	<#>	