Bitwarden Marketing Website Security Report

ISSUE SUMMARIES, IMPACT ANALYSIS, AND RESOLUTION

BITWARDEN, INC

Table of Contents

Summary

In December 2024, Bitwarden engaged with cybersecurity firm Paragon Initiative Enterprises (PIE) to perform a dedicated audit of the Bitwarden marketing website. A team of testers from PIE were tasked with preparing and executing the audit over three business days to reach total coverage of the system under review.

Six issues were discovered during the audit. Four issues were resolved post-assessment. Two issues were accepted.

This report was prepared by the Bitwarden team to cover the scope and impact of the issues found during the assessment and their resolution steps. For completeness and transparency, a copy of the Findings section within the report delivered by PIE has also been attached to this report.

Issues

Cryptography Implementation Issues with Public Demo Script (Low)

Status: Issue was fixed post-assessment.

The public demo script showcasing cryptography features was out of date, no longer used, and removed entirely from the system.

Use Sub-Resource Integrity for Externally Hosted Assets (Info)

Status: Issue was fixed post-assessment.

A resource missing Sub-Resource Integrity was removed and replaced with a new service that can provide the security capability.

Purify Strings Before Overwriting InnerHTML (Info)

Status: Issue was fixed post-assessment.

Client and server-side content sanitization was applied using the SafeHTML library.

Purify HTML Sourced from Contentful (Info)

Status: Accepted.

Sanitization is already being applied in the rendering layer and additional work would be unnecessary / duplicative.

Prevent the Installation of Insecure Composer Packages (Info)

Status: Accepted.

Bitwarden utilizes multiple layers of review and governance over the addition of new libraries and packages, and alerting tools such as Dependabot and Renovate are in place to notify about any risks in current source code.

Ensure Appropriate Use of Security Headers (Info)

Status: Issue was fixed post-assessment.

All appropriate security headers were added or confirmed working in responses.

II. Security Issues

1. Cryptography Implementation Issues with Public Demo Script

Severity: **Low**

The web page `public/crypto.html` includes JavaScript code that demonstrates symmetric-key encryption. This code is fine as a proof-of-concept but includes some design decisions that could undermine security if implemented in the Bitwarden password manager.

Bitwarden confirmed via Slack that their implementation has evolved over time from what was published on this web page. For completeness, here is a break-down of the issues identified.

MAC Tag Stripping

The biggest concern is the mixture of unauthenticated encryption and authenticated encryption:

```
const encTypes = {
  AesCbc256_B64: 0,
  AesCbc128_HmacSha256_B64: 1,
  AesCbc256_HmacSha256_B64: 2,
  Rsa2048_OaepSha256_B64: 3,
  Rsa2048_OaepSha1_B64: 4,
  Rsa2048_OaepSha256_HmacSha256_B64: 5,
  Rsa2048_OaepSha1_HmacSha256_B64: 6,
}
```

An attacker can take a ciphertext encrypted under an encType of **2**, change the type to **0**, and strip off the MAC.

Without any integrity guarantees, AES-CBC is vulnerable to Vaudenay's padding oracle attack, in which an attacker can replay modified ciphertexts and learn the plaintext from whether a padding error occurs on the decryption of a candidate message.

Concerns With Double HMAC

The way HMAC tags are compared is similar to the Double HMAC technique, but with the same key that message authentication relies on.

Double HMAC with a random key is secure against side-channel leakage because the operation is blinded by a random value.

Using the same key that calculated the original HMAC, to re-hash the authentication tag, doesn't have the same blinding property as an ephemeral key. It is therefore possible that some timing signal could still be leaking to an attacker, albeit over many thousands of samples.

Additionally, the loop to compare the outer HMAC bytes fails fast on a mismatched byte, which would be the prefect precondition for a timing leak if the static key permitted some signal to leak.

Finally, this last comparison could be replaced with `window.crypto.subtle.verify()`, which uses the built-in constant-time comparison functions.

```
  const importedMacKey = await window.crypto.subtle.importKey('raw', key, alg,
false, ['sign'])
  const mac1 = await window.crypto.subtle.sign(alg, importedMacKey, mac1Data)
- const mac2 = await window.crypto.subtle.sign(alg, importedMacKey, mac2Data)

- if (mac1.byteLength !== mac2.byteLength) {
-   return false
- }
-   const arr1 = new Uint8Array(mac1)
-   const arr2 = new Uint8Array(mac2)
-
-   for (let i = 0; i < arr2.length; i++) {
-     if (arr1[i] !== arr2[i]) {
-       return false
-     }
-   }
-
- return true
+ return await window.crypto.subtle.verify(alg, importedMacKey, mac1, mac2Data);
```

We disclosed this finding on the private bitwarden.com GitHub repository as issue 199.

III. Additional Recommendations

1. Use Sub-Resource Integrity for Externally Hosted Assets

When ever an external resource (JavaScript, CSS) is loaded from a CDN, it's highly recommended to populate the `integrity` attribute with a SHA2 hash (i.e., SHA-384) of the expected contents of this file.

This is not a widespread issue in the Bitwarden.com codebase, but was identified in the public/crypto.html demo script.

Without this integrity check, if the CDN is compromised by an attacker, the scripts or stylesheets could be replaced. If the asset in scope is a JavaScript file, it achieves the impact as an unmitigated cross-site scripting exploit. If it it's only a CSS file, they could still serve malicious CSS that defaces the script and hurts your brand reputation.

2. Purify Strings Before Overwriting InnerHTML

There are two patterns throughout the codebase with similar consequences, but should be largely mitigated by the Content-Security-Policy headers.

1. Setting the `innerHTML` property directly. For example: [1]
2. Using React's `dangerouslySetInnerHTML` attribute. For example: [2]

Although we could not find any immediate means to introduce a cross-site scripting exploit through either usage, we recommend building a robust mechanism for purifying strings before including them in the inner HTML of a parent element.

Consider adopting the DOMPurify library, which sanitizes HTML strings to prevent XSS without destroying the rich content.

3. Purify HTML Sourced from Contentful

Similar to Recommendation #2, except the best library for preventing XSS while allowing HTML in PHP is HTMLPurifier. This is not considered a significant security risk, since the sync process from Contentful includes an HMAC tag of the contents, which is verified at runtime.

The current caching mechanisms should already mitigate any performance hit with HTMLPurifier.

4. Prevent the Installation of Insecure Composer Packages

The Composer package, Roave/SecurityAdvisories, is a developer dependency that can be added to your package to prevent any known-vulnerable components from being installed.

This is achieved by introducing a version conflict with any known-vulnerable versions of a Composer package, sourced from various security advisory databases.

5. Ensure Appropriate Use of Security Headers

While the production environment was not in the scope of our investigation, we would be remiss to not recommend using Security Headers to ensure you're enabling and correctly configuring the appropriate browser security protecting your visitors and your brand.

A few headers to look out for:

- Strict-Transport-Security: You always want this header.
- X-Frame-Options: Prevent clickjacking.
- X-Content-Type-Options: Disables MIME-sniffing, which is a feature that causes some browsers to accidentally introduce client-side security vulnerabilities (i.e., XSS).
- Permissions Policy: Specifies which browser features (e.g., notifications) are requested by the current web page.

Normally this list includes Content-Security-Policy, but they're already implemented in the application code.