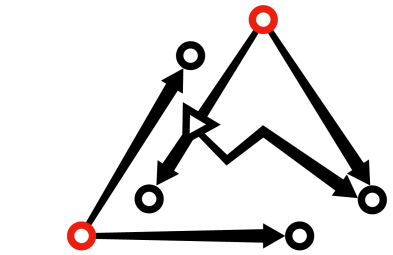

Least Authority

PRIVACY MATTERS

XMTP

Security Audit Report

MetaMask Snap

Final Audit Report: 18 August 2023

Table of Contents

Overview

Background

XMTP has requested that Least Authority perform a security audit of their Metamask Snap.

Project Dates

- **July 20, 2023 - July 24, 2023**: Initial Code Review *(Completed)*
- **July 26, 2023**: Delivery of Initial Audit Report *(Completed)*
- **August 17, 2023:** Verification Review *(Completed)*
- **August 18, 2023:** Delivery of Final Audit Report *(Completed)*

Review Team

- Jehad Baeth, Security Researcher and Engineer
- Nikos Iliakis, Security Researcher and Engineer

Coverage

Target Code and Revision

For this audit, we performed research, investigation, and review of the MetaMask Snap followed by issue reporting, along with mitigation and remediation instructions as outlined in this report.

The following code repositories are considered in scope for the review:

- XMTP Snap:
  https://github.com/xmtp/snap

Specifically, we examined the Git revision for our initial review:

- `89c7a7e6c80f36ebc671c1b2c237e1c241102f1b`

For the review, this repository was cloned for use during the audit and for reference in this report:

- XMTP Snap:
  https://github.com/LeastAuthority/xmtp_snap/tree/initial-release

For the verification, we examined the Git revision:

- `20185c87f3d79fbd66717a88e77f78fd79e4479c`

All file references in this document use Unix-style paths relative to the project's root directory.

In addition, any dependency and third-party code, unless specifically mentioned as in scope, were considered out of scope for this review.

Supporting Documentation

The following documentation was available to the review team:

- XMTP:
  https://github.com/xmtp
- Website:
  https://xmtp.org

In addition, this audit report references the following documents:

- EIP-1102: https://eips.ethereum.org/EIPS/eip-1102
- EIP-1193: https://eips.ethereum.org/EIPS/eip-1193#user-account-exposure-and-account-changes
- JavaScript Date object: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Date
- MetaMask Documentation: https://docs.metamask.io/snaps/concepts/design-guidelines

Areas of Concern

Our investigation focused on the following areas:

- Correctness of the Snap implementation;
- Potential misuse and gaming of the Snap;
- Adversarial actions and other attacks on the network;
- Denial of Service (DoS) and other security exploits that would impact the intended use of the Snap or disrupt the execution of the Snap capabilities;
- Vulnerabilities in the Snap code;
- Protection against malicious attacks and other ways to exploit Snap code;
- Inappropriate permissions and excess authority;
- Data privacy, data leaking, and information integrity; and
- Anything else as identified during the initial analysis phase.

Findings

General Comments

Our team performed a security review of the Extensible Message Transport Protocol (XMTP) MetaMask Snap. XMTP is an open protocol, network, and standards for security and privacy oriented Web3 messaging. The Snap is an implementation of the Keystore API, a defined interface for XMTP clients to interact with a Keystore holding XMTP key material.

Our team investigated the storage and handling of secrets in the implementation, in addition to XMTP's utilization of the MetaMask security framework and the use of permissions. We also examined the handling of data for leaks of sensitive information. Overall, we found that XMTP Snap adheres to MetaMask's Snaps design guidelines.

System Design

Our team found that security has been taken into consideration in the design of the XMTP Snap. During our review, we found that there is reliance on the operating system clock to determine token expiry. Furthermore, XMTP adheres to MetaMask's framework and utilizes its security measures correctly as demonstrated by limiting needed permissions, utilizing MetaMask's encryption scheme for storing secrets, compartmentalizing running processes, and minimizing the number of used dependencies. However, we found that the authorization is taking place in the client operations, and this can be easily manipulated by the user (Issue B).

Code Quality

Our team found the Snap implementation to be well-organized, and in adherence to the guidelines set out by MetaMask. There are sufficient comments explaining the codebase.

Tests

Our team found that there are tests covering the basic functionality of the system.

Documentation

While XMTP has comprehensive project documentation, the MetaMask Snap package currently lacks sufficient documentation, likely due to its early development stage. We recommend the Snap documentation be improved (Suggestion 3).

Code Comments

We found that code comments sufficiently describe the intended behavior of security-critical components and functions.

Scope

The scope of this review was generally sufficient and included all security-critical components. Nonetheless, given that the Snap package operates at a higher level and utilizes `@xmtp/xmtp-js` internally, we recommend conducting a focused audit for this critical package.

Dependencies

Our team found that vulnerable dependencies are used in the implementation, which could lead to security vulnerabilities (Issue A). We also found instances of unused dependencies (Suggestion 2), as well as a deprecated dependency (Suggestion 4).

Specific Issues & Suggestions

We list the issues and suggestions found during the review, in the order we reported them. In most cases, remediation of an issue is preferable, but mitigation is suggested as another option for cases where a trade-off could be required.

ISSUE / SUGGESTION	STATUS
Issue A: Usage of Vulnerable Dependencies	Resolved
Issue B: XMTP Snap Uses Client's Operating System Time To Determine Expiry of Keys	Unresolved
Suggestion 1: Update Code Comments for Current Implementation	Resolved
Suggestion 2: Remove Unused Dependencies	Partially Resolved
Suggestion 3: Improve Documentation	Unresolved
Suggestion 4: Replace Deprecated ethereum.enable Dependency	Resolved

Issue A: Usage of Vulnerable Dependencies

Location

`package.json`

Synopsis

Analyzing `package.json` for dependency versions using npm audit shows 20 vulnerable dependencies (18 Moderate, 2 High).

Impact

We cannot assess the exact impact of using vulnerable dependencies unless we evaluate the reported advisories. However, due to the security measures imposed by MetaMask's utilization of SES and LavaMoat, the impact is minimized.

Remediation

We recommend following a process that emphasizes secure dependency usage to avoid introducing vulnerabilities to the XMTP wallet. By removing known vulnerabilities, new potential issues will be easier to identify and address.

We suggest the following mitigation strategies:

- Manually assessing and regularly monitoring and maintaining security-critical dependencies;
- Using commit hashes instead of release number tags to point to the latest releases, as needed;
- Updating dependencies when security issues and bugs are detected and/or fixed; and
- Pinning updated dependency versions to releases compatible with the XMTP Snap in order to avoid breaking the codebase upon automatic dependency upgrades.

Status

The XMTP team fixed most of the vulnerable dependencies, but a few are part of the `site` package and are transitive dependencies from Gatsby that seem to have no patched version (all moderate severity). Additionally, the team noted that the `site` package is not part of the build process for the actual Snap and is purely a demonstration application.

Verification

Resolved.

Issue B: XMTP Snap Uses Client's Operating System Time To Determine Expiry of Keys

Location

`packages/snap/src/authorizer.ts`

Synopsis

The XMTP Snap uses JavaScript's `Date` object to determine the expiry of imported keys. The `Date` object uses the client's operating system (OS) as a source, which can be easily manipulated by simply changing the OS time.

Impact

Client-provided information may be unreliable, potentially allowing users to deceive the authorization mechanism.

Remediation

We recommend refraining from using the `Date.now` function to create timestamps that are used for authorization. Instead, we suggest using an external time server as a source for time information.

Status

The XMTP team acknowledged the finding but stated that they will not implement the remediation for the following reasons:

1. This vulnerability cannot be exploited to give new origins access to the XMTP Snap. It can only be used to extend access for a previously authorized origin. It also cannot be exploited to extract the XMTP secrets from the Snap without some other exploit. This greatly limits the usefulness of an attack;
2. In order to exploit this vulnerability, an attacker would have to obtain enough permissions to modify the system time. Hence, the pre-conditions are non-trivial since broad access to the user's machine is required; and
3. The suggested remediation would require that the XMTP team request networking permission in their Snap (which they do not currently need/request) to get the current time from a trusted time server.

Verification

Unresolved.

Suggestions

Suggestion 1: Update Code Comments for Current Implementation

Location

[packages/snap/src/utils.ts]

Synopsis

There are instances in the code where comments describe a different (older) implementation.

For example, as noted in the source code:

```
// returns the first and last four characters of the address, separated by ellipses:
export function prettyWalletAddress(address: string) {
return `${address.slice(0, 6)}...${address.slice(-4)}`;
```

Impact

The code's complexity makes it challenging to discern the desired functionality, as it is unclear whether to reference the actual code or the comments.

Remediation

We recommend updating the comments in the source code.

Status

The XMTP team has implemented the remediation as recommended.

Verification

Resolved.

Suggestion 2: Remove Unused Dependencies

Location

`package.json`

Synopsis

Unused dependencies were identified during the review process, which can cause confusion and disorganization in the codebase. As a result, reviewers and contributors may experience increased difficulty in understanding the system's intended behavior.

Unused `devDependencies`:

- `@metamask/auto-changelog`
- `Prettier-plugin-packagejson`
- `semantic-release`

Remediation

We recommend removing any unused dependencies.

Status

The XMTP team has only removed the `@metamask/auto-changelog package`.

Verification

Partially Resolved.

Suggestion 3: Improve Documentation

Synopsis

The Snap component is insufficiently documented, which inhibits maintenance, security review, and safe use of the system by users.

Mitigation

We recommend creating user documentation to help users make informed security decisions when using the Snap. The documentation should detail the permissions utilized by the Snap as well as the reason justifying the use of these permissions. We also recommend detailing all sensitive user data that is handled or collected by the Snap.

Status

At the time of verification, our team found no evidence that the project documentation has been improved.

Verification

Unresolved.

Suggestion 4: Replace Deprecated ethereum.enable Dependency

Location

`packages/site/src/utils/metamask.ts`

Synopsis

The dependency `window.ethereum.enable` is deprecated and should be replaced.

Remediation

We recommend replacing the `enable` function with a request to `eth_requestAccounts` , as explained in EIP-1102 and EIP-1193.

For example:

```
window.ethereum

.request({ method: 'eth_requestAccounts' })

.then((accounts) => console.log(accounts))

.catch((error) => console.error(error));
```

Status

The XMTP team has implemented the remediation as recommended.

Verification

Resolved.

About Least Authority

We believe that people have a fundamental right to privacy and that the use of secure solutions enables people to more freely use the Internet and other connected technologies. We provide security consulting services to help others make their solutions more resistant to unauthorized access to data and unintended manipulation of the system. We support teams from the design phase through the production launch and after.

The Least Authority team has skills for reviewing code in multiple Languages, such as C, C++, Python, Haskell, Rust, Node.js, Solidity, Go, JavaScript, ZoKrates, and circom, for common security vulnerabilities and specific attack vectors. The team has reviewed implementations of cryptographic protocols and distributed system architecture in cryptocurrency, blockchains, payments, smart contracts, zero-knowledge protocols, and consensus protocols. Additionally, the team can utilize various tools to scan code and networks and build custom tools as necessary.

Least Authority was formed in 2011 to create and further empower freedom-compatible technologies. We moved the company to Berlin in 2016 and continue to expand our efforts. We are an international team that believes we can have a significant impact on the world by being transparent and open about the work we do.

For more information about our security consulting, please visit https://leastauthority.com/security-consulting/.

Our Methodology

We like to work with a transparent process and make our reviews a collaborative effort. The goals of our security audits are to improve the quality of systems we review and aim for sufficient remediation to help protect users. The following is the methodology we use in our security audit process.

Manual Code Review

In manually reviewing all of the code, we look for any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus is on the in-scope code, we examine dependency code and behavior when it is relevant to a particular line of investigation.

Vulnerability Analysis

Our audit techniques include manual code analysis, user interface interaction, and whitebox penetration testing. We look at the project's website to get a high level understanding of what functionality the software under review provides. We then meet with the developers to gain an appreciation of their vision of the software. We install and use the relevant software, exploring the user interactions and roles. As we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results, search for similar projects, examine source code dependencies, skim open issue tickets, and generally investigate details other than the implementation. We hypothesize what vulnerabilities may be present and possibly resulting in Issue entries, then for each, we follow the following Issue Investigation and Remediation process.

Documenting Results

We follow a conservative and transparent process for analyzing potential security vulnerabilities and seeing them through successful remediation. Whenever a potential issue is discovered, we immediately create an Issue entry for it in this document, even before having verified the feasibility and impact of the issue. This process is conservative because we document our suspicions early even if they are later shown to not represent exploitable vulnerabilities. We generally follow a process of first documenting the suspicion with unresolved questions, then confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating our confirmation. After this, we analyze the feasibility of an attack in a live system.

Suggested Solutions

We search for immediate and comprehensive mitigations that live deployments can take, and finally, we suggest the requirements for remediation engineering for future releases. The mitigation and remediation recommendations should be scrutinized by the developers and deployment engineers, and successful mitigation and remediation is an ongoing collaborative process after we deliver our Initial Audit Report, and before we perform a verification review.

Before our report, including any details about our findings and the solutions are shared, we like to work with your team to find reasonable outcomes that can be addressed as soon as possible without an overly negative impact on pre-existing plans. Although the handling of issues must be done on a case-by-case basis, we always like to agree on a timeline for a resolution that balances the impact on the users and the needs of your project team.

Resolutions & Publishing

Once the findings are comprehensively addressed, we complete a verification review to assess that the issues and suggestions are sufficiently addressed. When this analysis is completed, we update the report and provide a Final Audit Report that can be published in whole. If there are critical unaddressed issues, we suggest the report not be published and the users and other stakeholders be alerted of the impact. We encourage that all findings be dealt with and the Final Audit Report be shared publicly for the transparency of efforts and the advancement of security learnings within the industry.