Least Authority
PRIVACY MATTERS

MetaMask Snaps

Security Audit Report

Consensys Software Inc.

Final Audit Report: 8 September 2023

Table of Contents

Overview

Background

Consensys Software Inc. has requested that Least Authority perform a security audit of their MetaMask Snaps.

Project Dates

- **April 4, 2023 - June 8, 2023:** Initial Code Review *(Completed)*
- **June 12, 2023:** Delivery of Initial Audit Report *(Completed)*
- **September 6, 2023:** Verification Review *(Completed)*
- **September 8, 2023:** Delivery of Final Audit Report *(Completed)*

Review Team

- Jehad Baeth, Security Researcher and Engineer
- Alejandro Flores, Security Researcher and Engineer
- Ann-Christine Kycler, Security Researcher and Engineer
- JR, Security Researcher and Engineer

Coverage

Target Code and Revision

For this audit, we performed research, investigation, and review of the MetaMask Snaps 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:

- MetaMask Snaps:
  https://github.com/metamask/snaps-monorepo

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

- `47258ef13ea928aa1bb1e1c6227073e57577d272`

For the verification, we examined the Git revision:

- `4b6bcfa933b2f09a12baebbb026b8ef160217857`

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

- MetaMask Snaps:
  https://github.com/LeastAuthority/MetaMask_Snaps

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:

- MetaMask Introduction:
  https://docs.metamask.io/guide/snaps.html

- MetaMask Docs:
  https://docs.metamask.io/guide/snaps.html
- Snaps Platform Audit Scope 2022-11.pdf *(shared with Least Authority via email on 9 November 2022)*
- MetaMask Snaps Diagram.pdf *(shared with Least Authority via email on 9 November 9 2022)*
- Snaps Vector Attack Tree Example.pdf *(shared with Least Authority via email on 8 December 2022)*

In addition, this audit report references the following documents:

- BIP32:
  https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki#user-content-Extended_keys

Areas of Concern

Our investigation focused on the following areas:

- Correctness of the implementation;
- Adversarial actions and other attacks on the wallet;
- Attacks that impact funds, such as the draining or manipulation of funds;
- Mismanagement of funds via transactions;
- Malicious attacks and security exploits that would impact the wallet;
- Vulnerabilities in the wallet code and whether the interaction between the related and network components is secure;
- Exposure of any critical or sensitive information during user interactions with the wallet and use of external libraries and dependencies;
- Proper management of encryption and storage of private keys;
- Inappropriate permissions and excess authority;
- Data privacy, data leaking, and information integrity; and
- Anything else as identified during the initial analysis phase.

Additionally, the MetaMask team requests that the following be addressed in the Snaps Platform Audit:

Possible Critical Issues:

- Escaping Secure ECMAScript (SES) confinement; and
  - Accessing globals not injected during SES sandbox creation
  - Spoofing MetaMask-extension <-> execution environment communication
  - Escaping execution environment's iframe altogether
- Exfiltrating MetaMask's internal state.

Possible Major Issues:

- Corrupting MetaMask's internal state (either in-memory or persistent);
- Getting access to endowments not specified in Manifest / not approved by user;
  - Getting access to other coin types than those requested (especially important for `snap_getBip44Entropy_*`)
- Executing Denial of Service (DoS) attacks on the main MetaMask extension; and
- Exfiltrating state of other Snaps (either in-memory or persisted through `snap_manageState` endowment).

Possible Minor Issues:

- Corrupting another Snap's internal state (either in-memory or persisted through `snap_manageState` endowment);
- Allowing a Snap to execute code outside of expected time boundaries;

    - Escaping request timeout and execution during idle timeout
    - Escaping idle timeout altogether and avoiding termination
    - Executing code after endowments have been torn down
    - See known issues below:
        - Corrupting execution environment iframe’s state;
        - Allowing the DApp to connect to a Snap without previous user approval; and
        - Executing DoS attacks on other Snaps.

Other Potential Issues To Be Verified:

- Snaps can get access to root global scope or communicate with other Snaps by traversing / modifying globals injected in SES.

Findings

General Comments

MetaMask Snaps (Snaps) is an interface intended to enable the interaction between the MetaMask Extension Wallet users and dApps. The Snaps system is composed of a server component that is controlled by the MetaMask team, and a client component that is implemented by the developers of dApps. Snaps provides a mechanism for a user to grant dApps permissions to execute actions in the wallet.

Allowing third-party developers to execute code inside a user's wallet carries inherent risks. Snaps uses Agoric’s Secure ECMAScript (SES) to build a sandbox (compartment) where untrusted code can be executed safely. In a previous audit published on March 4, 2020, our team audited SES and did not identify any breaches of SES compartments. During our review, our team assumed that SES works as intended and did not attempt to break SES itself. Rather, our focus was on whether the Snaps system used SES in a secure manner. We examined the JavaScript APIs available in the SES compartment and did not find any way to escape the SES compartment, nor to get unattenuated JavaScript objects inside the compartment. Additionally, we did not find a way to exfiltrate or corrupt MetaMask’s internal state from within the SES compartment.

Our team closely investigated the relationship between endowments and permissions. While the permissions system can be used to request access to restricted methods and endowments, we found that the permission system only allows granting access to endowments that are defined in the Snaps system.

We investigated whether untrustworthy actors could use malicious Scalable Vector Graphics (SVG) files using the Snaps’ logo but did identify any vulnerabilities. Additionally, we did not find a way to inject code into the templating system via the Snap Manifest file.

We investigated the Remote Procedure Call (RPC) communication system between dApps and Snaps, as well as between Snaps. We found that Snaps could not install Snaps, nor invoke them.

Once dynamic permissions were disabled, we did not find a way for Snaps to request access to what they were not initially given in the Manifest. This includes accessing derivation paths for tokens that were not specified in the Manifest.

Our team found that if `__proto__` items were inserted inside the Snap Manifest, these could be added without changing the `Shasum` of the Snap. Furthermore, we found that injected `__proto__` items inside `initialPermissions` were loaded into the extension and present during the installation and review of the `initialPermissions`. We did not find a way for this to be exploited into a full prototype pollution vulnerability (Issue A).

We also did not find a way to manipulate the state of other Snaps, nor to exceed the storage limitations of a Snap (100mb), and are continuing to investigate the possibility of a Denial of Service (DoS) related to storage overflows, where a Snaps storage is filled and causes a failure in the MetaMask extension.

We inspected whether Snaps could access derivation paths for BIP32 and BIP44 beyond what is specified in the Manifest, and were not able to bypass the validation checks. However, we recommend that additional validation be performed (Suggestion 2).

We found that Snaps will timeout when a Snap is continually sending requests to make the execution time of Snaps unbounded. We observed one Snap, which will continually make requests, without the long-running endowment run, for at least an hour (Issue B).

It is clear that the system was designed with security in mind. The SES compartments and endowments are thoughtfully considered so that only authorized and hardened APIs are available to Snaps. Additionally, all communication is performed over restricted RPC channels that further isolate the Snaps execution environment from the larger JavaScript environment.

Code Quality

In our review of the Snaps Monorepo code, our team found that the implementation is well-organized and in adherence with best practices.

Tests

We found that sufficient test coverage of Snaps is implemented to check for implementation errors and unexpected behavior that could lead to security vulnerabilities.

Documentation

The MetaMask ecosystem is generally well-documented. Our team noted the Snap Improvement Proposal (SIP) system, which provides a uniform methodology for proposing, discussing, and documenting design and implementation changes.

Code Comments

The Snaps implementation is well-commented, with functions and their parameters well-described.

Scope

This review has been designed with the current first phase covering the Snaps design and implementation. The second phase of this review will cover the extension and allow our team to make a comprehensive assessment of the system.

During the course of the audit, the MetaMask team discovered and addressed Issues exploitable through the RPC method. Patches for these Issues were cherry-picked into a new commit for the audit. One of these patches disabled the dynamic requesting and allocations of permissions. This solved an Issue identified by our team whereby permissions granted to a Snap for one domain were also granted to all domains for that Snap. An additional Issue resolved by these patches was that the permissions infrastructure was shared by both Snaps and dApps. As a result, dApps could, in theory, request permissions that were Snap specific. These Issues were found to be resolved in the latest commit. With dynamic permissions disabled, we found no way for a dApp to get access to restricted Snap methods or permissions.

Subsequently, the scope of our review was updated with commits that resolve the Issues identified by the MetaMask team. Our team validated these Issues, and reviewed the fixes in the commits.

While the Permission Controller was out of scope for this audit, the functionality where Snaps interacts closely with the Permission Controller code was investigated for any potential security issues.

Dependencies

We found that two of the example packages within the Snaps Monorepo have been classified as malware. Additionally, there were other high-risk vulnerabilities found in the dependencies, which we recommend be upgraded or replaced (Suggestion 3).

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: SHA Checksum Bypass	Resolved
Issue B: Snap Execution Exceeds Timeout	Unresolved
Suggestion 1: Implement Proper Validation of the BIP32 Derivation Path	Resolved
Suggestion 2: Remove __proto__ objects From initialPermissions	Resolved
Suggestion 3: Upgrade Insecure Dependencies	Resolved

Issue A: SHA Checksum Bypass

Location

`packages/snaps-utils/src/snaps.ts#L201`

Synopsis

The SHA `checksum` of a Snap can be bypassed by inserting `__proto__` objects into the Manifest. This will change the Manifest without changing the `Shasum` Hash.

Impact

Items in `__proto__` can end up inside the application, even when the Snap is not resigned. While we were able to identify malicious `__proto__` objects inside the `initialPermissions` object during Snap install, we found no way to exploit this.

Remediation

We recommend that `__proto__` objects be included as part of the hashing pre-image, or ensuring that all `__proto__` objects are removed from the Manifest before being processed (See Suggestion 2).

Status

The MetaMask team addressed this Issue by adding a sanitizing function in the out-of-scope library `@metamask/utils` and using it whenever JSON has to be parsed for generating the Manifest file. This strips any `__proto__` objects from the JSON.

Verification

Resolved.

Issue B: Snap Execution Exceeds Timeout

Synopsis

Repeated frivolous RPCs using `await` promises in a Snap can manipulate the `requestQueue` so that the Snap execution timeout is overridden, and the Snap executes indefinitely.

Impact

This Issue could result in the unintended behavior of the Snap.

Technical Details

Consider the following proof of concept that illustrates an attack:

```
let now = Date.now();

while(true) {

        let state = await snap.request({

          method: 'snap_manageState',

          params: { operation: 'get'}

      })

      console.log(state);

      console.log('Doing other stuff here');

      let duration = Date.now();

      console.log(`${duration - now} time passed`);

}
```

The aforementioned code requires the Snap to have both RPC and `manageState` permissions, but does not require the long-running endowment.

With the above code, we observed execution times from 15 to 45 minutes. However, this only occurs when the `await` keyword is used. When the code is changed so that the `snap.request` call resolves into a promise using `.then`, the Snap execution times out after approximately 60 seconds.

Remediation

Our team did not find a viable solution during the time of this audit to mitigate this Issue sufficiently. We recommend that this be subject to further investigation in the future.

Status

The MetaMask team stated that they were unable to prevent a Snap from executing an arbitrary function indefinitely. Hence, after evaluation, the team decided to defer the Issue, as each Snap is sufficiently sandboxed, such that this does not pose a major risk to the platform.

Verification
Unresolved.

Suggestions

Suggestion 1: Implement Proper Validation of the BIP32 Derivation Path

Location.
[src/manifest/validation.ts#L42]

Synopsis
According to the Bitcoin Improvement Proposal BIP32, numbers have to be in a specific range in the derivation path. However, the Snap permits the setting of a derivation path outside of the specified range $[0, 2^{31} - 1]$. This is not properly checked during Manifest validation. Consequently, if the `getBIP32Entropy` call is executed with the incorrect validation path, the Snap will allow the RPC to be performed, but it will then fail in the `MetaMask/key-tree` dependency. This will result in an error in the extension, which will then be relayed back to the DApp and displayed as an alert error message.

The validation `RegEx` currently checks if the input is a slash number followed by potentially another slash and another number. However, it does not validate that the input is in that specific range $[0, 2^{31} - 1]$.

Mitigation
Although this suggestion does not result in an exploit, we recommend performing proper validation of the BIP32 derivation path, according to the standard explained in the BIP32 Proposal.

Status
The MetaMask team has added a range check to the out-of-scope function `isValidBIP32PathSegment` from the `@metamask/key-tree` package and is using it to validate BIP32 validation paths.

Verification
Resolved.

Suggestion 2: Remove __proto__ objects From initialPermissions

Synopsis
`__proto__` objects in the `initialPermissions` of the Snap Manifest are present in the `initialPermissions` object during Snap installation. While this was not exploitable as a prototype pollution issue, the presence of arbitrary objects in `__proto__` is unnecessary.

Mitigation
We recommend that the `initialPermissions` object be sanitized by running the object through `JSON.stringify` and then `JSON.parse`.

Status
The MetaMask team has resolved this suggestion by implementing the `getSafeJSON` function from the `@metamask/utils` package that strips any `__proto__` objects from the JSON.

Verification

Resolved.

Suggestion 3: Upgrade Insecure Dependencies

Synopsis

Running `npm audit` on the codebase reveals that several dependencies are outdated and have vulnerabilities. While it is not clear to what extent these would be exploitable, it is good practice to keep dependencies up-to-date in order to avoid importing vulnerable code.

Mitigation

We recommend updating or replacing the reported dependencies.

Status

The MetaMask team has upgraded the dependencies 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.