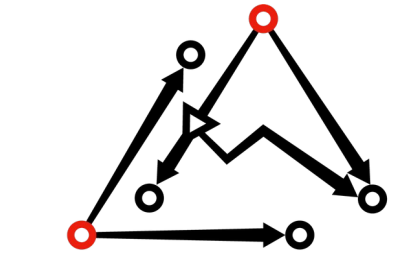

Least Authority

PRIVACY MATTERS

MetaMask Permissions System + CapNode

Final Security Audit Report

ConsenSys AG

Report Version: 27 November 2019

Table of Contents

Overview

Background

ConsenSys AG has requested that Least Authority perform a security audit of MetaMask, a browser extension that enables interaction with applications built on Ethereum. MetaMask allows users to browse the web and interact with Ethereum applications, sign messages and transactions, and securely manage and store their private keys and assets.

The following components are in scope:

1. Login Permissions System (OCAP)
    a. npm module
    b. MetaMask branch utilizing npm module
2. Plugin System
    a. CapNode

Project Dates

- **August 28 - September 18** : Code review completed (*complete)*
- **September 21** : Delivery of Initial Audit Report (*complete)*
- **November 25 - November 26:** Verification completed (*complete)*
- **November 27:** Delivery of Final Audit Report (*complete)*

Review Team

- Lily Anne Hall, Security Researcher and Engineer
- Dominic Tarr, Security Researcher and Engineer
- Alexander Leitner, Security Researcher and Engineer

Coverage

Target Code and Revision

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

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

- Login Permissions System (OCAP):
    - npm Module: https://github.com/MetaMask/json-rpc-capabilities-middleware
    - MetaMask branch using npm module: https://github.com/MetaMask/metamask-extension/tree/LoginPerSite
- Plugin System
    - CapNode (allows plugins to provide an API to sites):
        - https://www.npmjs.com/package/capnode
        - https://github.com/danfinlay/capnode

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

`metamask-extension@15b78d32e6284e85018de12845cc7563b3aa7f81`

`json-rpc-capabilities-middleware@e097c14ab307f16743724e8a1cb592769398624`

5

`capnode@243c0bca0dc664f2626f73719f865d110d99a402`

For the verification, we examined the Git revision:

`json-rpc-capabilities-middleware@aa9820d61ebd4b81b1c84ae5653c761f6abc059b`

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

Supporting Documentation

The following documentation was available to the review team:

- Login Permissions System: https://hackmd.io/7huV-aIpTOKhoZlEMhjNvw?view

Areas of Concern

Our investigation focused on the following areas:

- Correctness of the implementation;
- Vulnerabilities within each component as well as secure interaction between the network components;
- Data privacy, data leaking, and information integrity;
- Key management implementation: secure private key storage and proper management of encryption and signing keys;
- Storing assets securely;
- Any attack that impacts funds, such as draining or manipulating of funds;
- Mismanagement of funds via transactions;
- Exposure of any critical information during user interactions with the blockchain and external libraries;
- General use of external libraries;
- Inappropriate permissions and excess authority; and
- Anything else as identified during the initial analysis phase.

Findings

General Comments

The MetaMask permissions system showed a high quality in structure and legibility that we have come to expect from the MetaMask team. Our review was facilitated by the logical architecture and well-defined structure of the system components. The code is overall comprehensible; the various pieces that connect do so in a manner that was easy to understand and reason about.

This allowed us to discover some critical vulnerabilities that could potentially be exploited to undermine the entire permission system as well as several bugs that could lead to unexpected or undefined behavior. Some of these issues enable silent circumvention of user choice in allowing or denying applications access to certain permissions as well as potentially allowing manipulation and disruption of the operation of the wallet.

We acknowledge that some of these issues are resolvable through Lava Moat, which is currently in development. However, given the unfinished state of that effort, we strongly encourage that these problems are addressed individually as early as possible.

Specific Issues & Suggestions

We list the issues we found in the code 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: Permission Requests Can Be Circumvented	Resolved
Issue B: Subset Check In Reverse Order	Invalid
Issue C: Method IDs Have Half Expected Entropy	Resolved
Issue D: Lack of Method Registry Access Control	Partially Resolved
Issue E: Method Registry Uses Reverse Index of Functions by their…	Invalid
Suggestion 1: Redundant Check if Variable is Undefined	Resolved
Suggestion 2: Redundant Check if IOriginMetadata.id is Empty When Always Set	Resolved

Issue A: Permission Requests Can Be Circumvented

Location

https://github.com/LeastAuthority/metamask-json-rpc-capabilities-middleware/blob/master/index.ts#L558-L585

Synopsis

The capabilities middleware grants permission based on the resolution of a promise returned from a user-defined function. This leaves the security of the permission system up to the Promise implementation.

Impact

Critical. An attacker could silently subvert the permissions system and gain full access to users' wallets.

Preconditions

Attacker is able to overwrite the Promise implementation, by compromising a dependency.

Feasibility

Unknown. The attacker's ability to compromise a dependency depends on how frequently and thoroughly the dependencies are reviewed and audited, the security practices of maintainers of the dependencies, the security of the package registry, build pipelines, and many other areas of concern. Due to the surface area of dependency poisoning attacks, we are not able to say definitively what the feasibility of such an attack is, given that much of this area is outside of the scope of this audit.

Technical Details

The JSON RPC capabilities middleware package provides access to a particular set of methods by requesting permission from the end user. In the context of this package, that permission request is an

unspecified, user-designated, promise-returning function. The package that uses this (the extension itself) supplies this function, which displays a dialog in the user interface and, based on the end user's input, either resolves or rejects the Promise.

In the middleware, it is decided whether or not to provide access to these methods based on the Promise's ending status. If resolved with the appropriate permission data, then the user is given access, otherwise access is denied. In our attack, we act as a rogue dependency to overwrite the global Promise implementation used such that all Promises always resolve with the permission data for which we desire access. In doing so, regardless of how the user interacts with the permission dialog, the middleware will always act as though the permissions were granted.

This is particularly problematic because this may be completely invisible to the end user. The user will deny the permission request and believe that the denial was successful, while in truth, the attacker may then make use of whichever wallet functions were exposed. This attack was validated by modifying a Promise polyfill to suit our attack and using it to overwrite the global Promise, then authoring a test within the existing test suite.

```
diff --git a/test/requestPermissions.js b/test/requestPermissions.js
index cab1a02..39e9db7 100644
--- a/test/requestPermissions.js
+++ b/test/requestPermissions.js
@@ -6,6 +6,7 @@ const rpcErrors = require('eth-json-rpc-errors')
 const USER_REJECTION_CODE =
require('../dist/src/errors').USER_REJECTED_ERROR.code
 const INVALID_REQUEST_CODE = rpcErrors.ERROR_CODES.jsonRpc.invalidRequest

+
 test('requestPermissions with user rejection creates no permissions', async
(t) => {
   const expected = []

@@ -182,3 +183,51 @@ async function sendRpcMethodWithResponse(ctrl, domain,
req) {
    }
   })
 }
+
+test('modifying the promise prototype can grant self permissions',
function(t) {
+  OldPromise = Promise;
+  Promise = require('./bad-promise')
+
+  const expected = {
+    parentCapability: 'restricted',
+    invoker: 'all.your.base'
+  };
+
+  const ctrl = new CapabilitiesController({
+    requestUserApproval: () => {
+      return new Promise(function(resolve, reject) {
+        reject(new Error('User rejected permissions'));
+      });
+    },
+    restrictedMethods: {
+      restricted: (req, res, next, end) => {
+        res.result = 'Wahoo!';
+        end();
```

```
+     }
+   }
+ })
+
+ const domain = { origin: 'all.your.base' }
+ let req = {
+   method: 'requestPermissions',
+   params: [
+     {restricted: {}}
+   ]
+ }
+ let res = {}
+
+ ctrl.providerMiddlewareFunction(domain, req, res, next, end)
+
+ function next() {
+   t.ok(false, 'next should not be called')
+   t.end()
+ }
+
+ function end() {
+   const perms = ctrl.getPermissionsForDomain(domain.origin)[0];
+   t.ok(equal(perms.parentCapability, expected.parentCapability), 'has
correct parentCapability');
+   t.ok(equal(perms.invoker, expected.invoker), 'has correct invoker');
+   Promise = OldPromise;
+   t.end()
+ }
+});
```

The Promise polyfill used can be reviewed at:
https://gist.github.com/emeryrose/c0195b091d7910d14cf0762073674b16#file-bad-promise-js-L439-L441

Mitigation

While Promises are used extensively throughout the code and a complete evaluation of how this type of attack might impact all of those areas was not feasible within the timeframe, we do know that the impact on the requestUserPermissions function is that an attacker can completely circumvent the permission dialog. An effective mitigation for this specific area of concern would be to simply trade the use of a Promise returning function for a standard node-style callback.

Remediation

The longer term remediation for this entire class of issues is a complete and verified implementation of SESify/LavaMoat. We acknowledge that this project is currently in development and that this particular issue is part of an entire class of issues for which the MetaMask team is already actively pursuing a solution.

Status

The global Promise implementation is frozen upon boot in the extension background and interface, before any other dependencies are imported.

https://github.com/MetaMask/metamask-extension/pull/7309.

Verification

Resolved.

Issue B: Subset Check In Reverse Order

Location

https://github.com/MetaMask/json-rpc-capabilities-middleware/blob/master/src/caveats.ts#L24

Synopsis

The design of permission caveats are intended to restrict options to a fixed list given in the caveat, however, the arguments to isSubset were the wrong way around. As a result, as long as the options given in the caveat are provided, other unlisted options may also be used.

Impact

The impact depends on what the specific permissioned API does and what options it uses. If option A was left out of the caveat that grants option B, a call enabling options A and B would pass.

Preconditions

An API that has some features that are restricted but are enabled by an option, possibly to reveal private data, or excede a spend limit.

Feasibility

Unknown. Depends upon the implementation. See *Preconditions* .

Technical Details

The arguments to isSubset were in reverse order, meaning it was checking if the caveat options are a superset not a subset. This was not caught by the test cases because they only tested with equal inputs (and an equal set is also a subset).

The error is that the isSubset module had arguments in an unexpected order. Instead of reading left to right, isSubset(A,B) meaning “is A subset of B”, it was in reverse order. While the metamask code had the correct intentions, it was assumed that the arguments applied from left to right.

Remediation

Correct the order of the options to isSubset . Test cases that checked against inputs that were actually subsets or supersets would have also caught this problem.

Status

The MetaMask team clarified that the subset check was in the correct order, but the naming/documentation of the caveats was confusing. The subset check remains unchanged, however the documentation has been clarified.

Verification

Invalid.

Issue C: Method IDs Have Half Expected Entropy

Location

https://github.com/LeastAuthority/metamask-capnode/blob/243c0bca0dc664f2626f73719f865d110d99a402/index.ts#L22

Synopsis

The constant K_BYTES_ENTROPY = 20 suggests that message IDs with enough entropy to be unguessable were selected. This is then passed to a module that interprets this as a character length, returning a 20 character long string that has only 10 bytes of entropy.

Impact

The entropy is reduced from a level that would be cryptographically unguessable to one that would potentially be guessable. It would require a significant amount of time to do so but is no longer in the bounds that would make the entropy unguessable.

Preconditions

To brute force a message ID, it would be necessary to have an oracle of some form (a part of the system that behaves in a way that "yes" and "no" answers can be required of it). As it stands, sends messages with random IDs would suffice, usually returning an error, but if you get one right it would return a valid response.

Feasibility

Low. This attack would be pretty difficult to pull off in practice and would use a lot of local computing resources, which the user might notice.

Technical Details

Send protocol messages directly with random values for the method ID.

Remediation

Use the amount of entropy originally intended, which is 40 characters in hex or 27 characters in base64.

Status

The entropy parameter has been doubled.

Verification

Resolved.

Issue D: Lack of Method Registry Access Control

Location

https://github.com/LeastAuthority/metamask-capnode/blob/master/src/method-registry/index.ts

Synopsis

An attacker knowing the method ID can call a registered method they should not be able to access.

Impact

Critical. Other extensions and web applications may be able to move user funds without consent.

Preconditions

The attacker must know the method ID. Since the entropy of the method ID is low (*Issue C*) it's conceivable that it could be guessed. The method ID could also be obtained by exploiting a vulnerability in another remote, or the remote could leak it intentionally.

Feasibility

The feasibility depends greatly upon the implementation. We do know that even a modestly sized botnet could brute force the 10-byte key used within a reasonable amount of time, especially if it is evident that user funds could be compromised.

Technical Details

An incoming message is passed to `processMessage` which then checks the message type and forwards the message to the appropriate handler, along with a reference to an `emitMessage` function that allows the handler to respond to the caller. For an invocation (calling a remote function), the arguments are deserialized (which includes regenerating functions from any method IDs given in the serialized form) and these are passed to the function. The return value is then sent back to the calling remote (if it was a Promise, it would wait until it is resolved). If a function was at some point passed to a call, its method ID gets registered, along with a reference to a function that send messages to the remote, which passed that function. If the `messageid` chosen by the first remote becomes known to another remote, they may now call it, and the CapNode instance will act as a proxy between the two remotes.

This also affects other interactions around method IDs. For example, peers can dealloc methods they do not own but know the method ID (except for index methods).

Mitigation

Resolving *Issue C* would prevent the possibility of guessing the method ID. Tracking which remote owns a function and checking that the calling remote is acceptable, would probably be a small change to the current code.

Remediation

To fully prevent the problem, have separate registries for each remote and for each set of local methods exposed to a particular remote. That way, it's always clear what context a registered method is to be used in, then it doesn't matter if two remotes share the same method id. This means they can even be incrementing integers.

Status

The usage of the method registry code has been updated to ensure that instances of the registry are on a per-connection basis. The MetaMask team has also indicated that this will be the first of a series of improvements that should prevent shared instance usage between connections.

Verification

Partially Resolved.

Issue E: Method Registry Uses Reverse Index of Functions by their Source String

Location

https://github.com/danfinlay/capnode/blob/243c0bca0dc664f2626f73719f865d110d99a402/src/method-registry/index.ts#L24

Synopsis

Javascript Maps index functions by their source. However, due to closure references, functions can have the same source but display different behaviour. This will result in bugs.

Impact

This is an ordinary correctness bug. Since it does work as intended if only one or two functions are registered, it could be considered a security bug, but it is also likely to break on many non malicious applications.

Preconditions

More than one function that happens to have the same string value is registered. Due to closure scope, if using functional programming techniques (functions that return functions), it is likely that functions with the same source have different behavior. If two of these are registered, the second one will steal the first one's method ID.

Feasibility

Likely to happen accidentally in a medium to large application.

Technical Details

Javascript Maps store a key:value array that accept Javascript objects or primitives as keys. However, if a function is used as the key, the source string of that function is used. That means if you have two functions that are not equal, `fn1 !== fn2`, but have the same source, `fn1.toString() === fn2.toString()`, the second function will steal the method ID of the first function when it is registered.

Remediation

Remove the reverse function map, allowing the same function to be registered twice and giving it more than one method ID.

Alternatively, instead of maintaining a reverse index of functions, iterate over the registry and check each one with `===` when checking if a function already exists. This will check functions by reference, which will behave as expected.

Status

This issue was reported based on the incorrect assumption that JavaScript maps were indexed by their source string, however this is not the case - they are indexed by their ID. This was an oversight in our understanding of how JavaScript maps are implemented.

Verification

Invalid.

Suggestion 1: Redundant Check if Variable is Undefined

Location

https://github.com/LeastAuthority/metamask-json-rpc-capabilities-middleware/blob/master/index.ts#L457

Synopsis

When domain settings do not already exist, a new key value pair is created and then a check is done to verify domain is not undefined.

Mitigation

Delete the redundant check.

Status

The check has been replaced with a more useful validation.

Verification

Resolved.

Suggestion 2: Redundant Check if IOriginMetadata.id is Empty When Always Set

Location

https://github.com/LeastAuthority/metamask-json-rpc-capabilities-middleware/blob/master/index.ts#L567

Synopsis

There is a check if `metadata.id` is empty. If it is not set, the `metadata.id` is created with a new `uuid()`. The updated metadata is set on the `permissionsRequest`. As a result, there is a redundant check if `permissionsRequest.metadata.id` is empty.

Mitigation

Delete the redundant check.

Status

The extraneous check has been removed.

Verification

Resolved.

Recommendations

We recommend that any partially resolved *Issues* and *Suggestions* stated above are addressed as soon as possible and followed up with verification by the auditing team.

Least Authority also recommends that continuous audits be conducted on future development releases to ensure that any potential issues and vulnerabilities are identified, addressed, and verified.

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 C, C++, Python, Haskell, Rust, Node.js, Solidity, Go, and JavaScript for common security vulnerabilities and specific attack vectors. The team has reviewed implementations of cryptographic protocols and distributed system architecture, including in cryptocurrency, blockchains, payments, and smart contracts. 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. Although we are a small team, we believe that 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 included manual code analysis, user interface interaction, and whitebox penetration testing. We look at the project's web site 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. While 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, creating Issue entries, and for each we follow the following Issue Investigation and Remediation process.

Documenting Results

We follow a conservative, 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 though we have not yet 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 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 report, and before the details are made public.

Responsible Disclosure

Before our report or any details about our findings and suggested solutions are made public, 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 resolution that balances the impact on the users and the needs of your project team. We take this agreed timeline into account before publishing any reports to avoid the necessity for full disclosure.