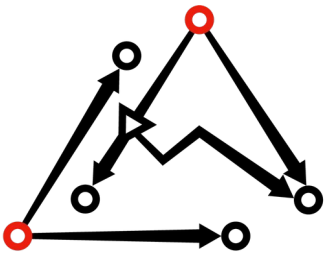
MetaMask Mobile Application

Final Security Audit Report

MetaMask

Report Version: 9 April 2019



Least Authority

PRIVACY MATTERS

Table of Contents

Overview

MetaMask has requested that Least Authority perform a security audit of their mobile application, a wallet and developer tool for 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 mobile application is built in React Native within a single codebase for both iOS and Android platforms. MetaMask previously built and released a web extension providing the same functionality, which is included within the mobile application.

The audit was performed from February 18 - March 7, 2019, by Lily Anne Hall and Dominic Tarr, with dedicated project management support by Hind Abu-Amr. The initial report was issued on March 8, 2019. A final report has been issued following the discussion and verification phase on April 9, 2019.

Coverage

Target Code and Revision

For this audit, we performed research, investigation, and review of the MetaMask followed by issue reporting, along with mitigation and remediation instructions outlined in this report. The following code repositories are in scope:

Specifically, we examined the Git revisions:

`gaba@92cf95476c0732a13c5e30cadfbf9296cdd7b1cf`

`metamask@b57476b142cedbddf725f8787b668ca64642b4c2`

`eth-keyring-controller@9e180e5b10c0ceeb437f6d44360b525b3083c723`

`browser-passworder@089893779ce366a9f0ee038b9c71708649fc0e1d`

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

Areas of Concern

Our investigation focused on the following areas:

- Any attack that impacts funds, such as draining or manipulating of funds;
- Exploitation of the webview to gain control of the wallet;
- Areas where insufficient validation allows for hostile input;
- Application of cryptography to protect secrets;
- Storing private keys and assets securely;
- Exposure of any critical information during user interactions with the blockchain and external libraries;
- General use of external libraries;
- Secure usage of the feature synchronizing the wallet from the web extension to the phone using a QR code;
- Potential points of failure resulting from the use of native code for encryption which calls back to different libraries on iOS and Android;
- Use of Gaba; and

- Anything else as identified during the initial analysis phase.

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 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.

Findings

Code Quality

Overall, the packages we reviewed were structured in a manner that was simple and intuitive to follow. However, we found that there is liberal use of external dependencies, both third party and otherwise. In some instances, this made it somewhat difficult to trace some of the more sensitive code paths. Due to the scope of the audit being limited to Gaba and MetaMask, we were unable to fully evaluate a number of packages that are authored by the MetaMask team and used in Gaba and MetaMask, such as the group of "eth-*" generic packages. While this type of modularity can be useful for making specific functionality available to generic uses, it has a negative impact on security evaluations. In the *Issues* section we have addressed many of these challenges as specific issues.

While we found many areas of improvement, we did not find any critical security vulnerabilities that pose an immediate and clear threat to value stored in MetaMask. Upon verification of the issue remediation, we were pleased to find that all of our concerns were addressed and easy to verify due to the development practices utilized by the team, including good code organization, pull request compartmentalization, commit messages, and more.

As a whole, we found the code base to be of exceptional quality and that it adheres to best development practices.

Issues

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: Dependencies Should Be Pinned to Exact Versions	*Resolved*
Issue B: Polling Timers Can Exhaust File Descriptors	*Resolved*
Issue C: The safelyExecute Function Swallows Exceptions	*Resolved*
Issue D: Transaction Validation is Insufficient	*Resolved*
Issue E: Password Strength Calculation is Insufficient	*Resolved*
Issue F: The isSmartContractAddress Function is Unreliable Over Time	*Resolved*
Suggestion 1: Add Validation to the AddressBookController	*Resolved*

Suggestion 2: Add Test Coverage For Gaba's KeyringController	*Resolved*
Suggestion 3: PreferencesController Should Match Against Non-Public IPs	*Resolved*
Suggestion 4: Force TLS in RPC URL in AppSettings	*Resolved*
Suggestion 5: Allow dApps to Access MetaMask by Whitelist Only	*Resolved*
Suggestion 6: Require TLS for Opening dApp Deeplinks	*Resolved*
Suggestion 7: Remove Links to Questionable dApps	*Resolved*
Suggestion 8: Improve the isDecimal Number Utility	*Resolved*
Suggestion 9: Do Not Send Regular Logs to Crashlytics	*Void*

Issue A: Dependencies Should Be Pinned to Exact Versions

Synopsis

Both Gaba and MetaMask each contain over 1000 dependencies, most of which are not pinned to an exact version but set to compatible version (^x.x.x). This can potentially enable dependency attacks as observed with the event-stream package with the Copay Bitcoin Wallet.

Impact

Critical. Could lead to complete loss of funds.

Preconditions

The author, maintainer, or attacker with publish access to any of the 1000+ dependencies that MetaMask and Gaba has publishes a new compatible version that sneaks in malicious code to steal MetaMask user private keys or otherwise subvert the security of the application.

Feasibility

Easy. MetaMask is a very high value target and, while the copay/event-stream attack did not actually succeed in stealing any funds, attackers may consider trying again.

Mitigation

Pinning dependencies to an exact version (=x.x.x) can reduce the possibility of inadvertently introducing a malicious version of a dependency in the future.

Remediation

As acknowledged by the MetaMask team as part of a roadmap briefing provided, the future use of SES containers for module sandboxing can eliminate this class of attacks.

Status

The MetaMask dependencies have been pinned to exact versions and the Gaba dependencies have been locked using npm-shrinkwrap.

Verification

Resolved.

Issue B: Polling Timers Can Exhaust File Descriptors

Synopsis

Gaba contains a number of classes that initiate a polling mechanism for querying remote services for various information. Due to the way in which these timers are written, certain network conditions could lead to “stacking” connections, thus causing an exhaustion of sockets/file descriptors.

Impact

Running out of file descriptors could lead to MetaMask and other applications being unable to operate correctly.

Preconditions

Various networks conditions that might lead to a connection opening but hanging indefinitely, such as the use of an unresponsive proxy, some networks with captive portals, or simply high latency.

Feasibility

Low. This is an edge case that is probably not very likely to impact most users.

Technical Details

All of the polling timers are constructed in the same way. The async action is triggered, immediately followed by a timeout to call it again. This means that the timeout is started without concern for whether or not the action is completed. As a result, if the network conditions are right, these polling timers can consume more and more file descriptors over time.

The following modules are affected: AccountTrackerController, AssetsDetectionController, CurrencyRateController, NetworkStatusController, PhishingController, ShapeShiftController, TokenBalancesController, TokenRatesController, TransactionController.

Remediation

Rework the polling/interval logic used to wait for the completion of the current async action before triggering the next timeout. This would prohibit the application from continuing to stack retries on top of one another If these network conditions are problematic.

Status

The interval based timers have been updated to timeout-base polling as recommended.

Verification

Resolved.

Issue C: The safelyExecute Function Swallows Exceptions

Synopsis

Gaba often makes use of a utility function called `safelyExecute`. The purpose of this function is to call a supplied function without handling any exceptions or errors that may occur.

Impact

Unknown. Swallowing errors is never a good idea. If a bug were introduced in any of the code passed to `safelyExecute`, it can make it extremely difficult to trace. Furthermore, it could potentially hide issues that could be related to security vulnerabilities.

Preconditions

A bug is introduced into code that gets passed to `safelyExecute`.

Feasibility

Moderate. Given that there is very good test coverage, the probability of this happening is low. However, test suites are not always adequate for detecting all edge cases.

Technical Details

The `safelyExecute` function accepts a function that returns a Promise and then calls it inside of a `try...catch` block. The catch block does not handle exceptions, provide a way for the caller to handle them, or even log the error. Because of this failures and critical issues can go unnoticed.

Mitigation

Adding an error logger can at least provide feedback for understanding failures.

Remediation

Do not swallow errors, instead handle all errors.

Status

The recommended mitigation strategy was implemented and errors are now logged instead of completely swallowed.

Verification

Resolved.

Issue D: Transaction Validation is Insufficient

Synopsis

The `validateTransaction` utility function in Gaba is not sufficient in preventing hostile input.

Impact

Moderate. If an attacker were able to insert a fake transaction or if a math bug elsewhere in the code were introduced, it could cause things like balances to be unreadable and potentially create issues with sending transactions based on that balance.

Preconditions

Remote API changes response format, a math bug is introduced elsewhere in the code, or an attacker manages to manipulate input.

Feasibility

Unknown.

Technical Details

The value is converted to a string and then checked for "-" and "." to determine if it is a negative or floating point number respectively. There are no other checks.

The current incarnation of this function allows the numerical values `Infinity` and `NaN`as well as unsafe numbers like `10000000000000000` and even strings like "one million dollar$" . This can potentially lead to a class of bugs like `9007199254740992 === 9007199254740993; // true`

Remediation

Instead of type casting and checking for substrings, the best approach here is to check `Number.isFinite()` , `!Number.isNaN()` , and `Number.isSafeInteger()` . This will provide the validation needed and perform better.

Status

Transaction validation was improved in a manner consistent with the recommended remediation.

Verification

Resolved.

Issue E: Password Strength Calculation is Insufficient

Synopsis

The ChoosePassword view has a password validation routine that attempts to enforce a certain degree of password strength, yet currently "abc123!" and "passw0rd!" will receive high strength scores. Since the password is used with browser-passworder to protect wallet keys, it's likely that it's worth brute forcing.

Impact

Critical. Could lead to complete loss of funds.

Preconditions

Attacker obtains an encrypted MetaMask backup or wallet. This is as simple as stealing the user's phone or computer.

Feasibility

Easy. A moderately determined thief should have no problem stealing a device.

Technical Details

Encryption at rest is only good with a good password. Users are likely to choose common passwords, using common substitutions. If an attacker possessed an encrypted MetaMask backup, a dictionary attack using common passwords could likely be all that is needed. Password hashing with a salt does make brute forcing a password more expensive, but it's still easy to check thousands of passwords, which will cover a lot of users.

Remediation

Exclude use of passwords included in the top 10,000 known passwords. There are modules for estimating password strength such as `zxcvbn` , however, the common practice to estimate strength in time tends to give an overestimated sense of security. It is recommended to estimate password strength in money (i.e.

one year to break a password sounds considerably strong, however, that's about $170 on ec2. Of course, it can be parallelized so it also wouldn't require a full year).

Status

Password strength is now checked using zxcvbn per the recommended remediation strategy.

Verification

Resolved .

Issue F: The isSmartContractAddress Function is Unreliable Over Time

Synopsis

The utility function for determining if an address belongs to a smart contract is only reliable when called. The result, if cached, could be different at any time in the future.

Impact

Unknown. Depends on how the result is used.

Preconditions

Contract is either deployed after the function is called or self destructed after the function is called.

Feasibility

Easy. Contract owner has complete control over the contract.

Technical Details

This method of checking if an address has a contract (getcode) is only reliable when it is called, as the address may not have always been a contract and may not be in the future.

Remediation

Care should be taken now and in the future to ensure that this method should be called every time the respective result is needed - not cached.

Status

The number of areas where the result of this call is cached has been greatly reduced and the areas where it is still cached are deemed safe.

Verification

Resolved.

Suggestions

Suggestion 1: Add Validation to the AddressBookController

Synopsis

The AddressBookController in Gaba does not perform any validation on the input and allows invalid addresses.

Mitigation

Validate addresses added to the address book.

Status

Address validation has been added to the controller.

Verification

Resolved .

Suggestion 2: Add Test Coverage For Gaba's KeyringController

Synopsis

Test coverage is incorrectly reported at 100% due to the lack of importing and testing of the KeyringController . The "heavy lifting" of this controller is handled in another package, but it's still important to test this controller to ensure proper usage.

Mitigation

Author a test suite for the KeyringController .

Status

Tests were added for the KeyringController.

Verification

Resolved.

Suggestion 3: PreferencesController Should Match Against Non-Public IPs

Synopsis

There is a condition in the PreferencesController that checks if the RPC URL is http://localhost:8545 , presumably to check if the node is a local testing node. However, a user may conceivably run their node on a different port, which would fail this check.

Mitigation

Simply match against 127.0.0.1, localhost, or any non-public IP address.

Status

The referenced code was removed entirely.

Verification

Resolved.

Suggestion 4: Force TLS in RPC URL in AppSettings

Synopsis

The AppSettings UI allows the user to input a custom RPC URL, however, there doesn't appear to be any enforcement on the protocol to use HTTPS.

Mitigation

Either enforce that users use HTTPS or display a warning to inform the user that communication with the RPC server could be monitored or manipulated if HTTP.

Status

Non-TLS connection are now only allowed if the host is a loopback interface or on the local network.

Verification

Resolved.

Suggestion 5: Allow dApps to Access MetaMask by Whitelist Only

Synopsis

The current whitelist system is in place for the purpose of bypassing the phishing warning for reported sites. This is a reactive approach to whitelisting that means someone or some group of users will have had to have been phished and reported it in order to prevent the offending dApp from being automatically granted access.

Mitigation

Using the whitelist as a means for allowing **any** dApp to use the MetaMask API would be a proactive approach and prevent unknown or unwanted dApps from accessing MetaMask.

Status

The MetaMask team has enabled “Privacy Mode” for all mobile users, a feature allowing sites to access user accounts only with explicit consent from the user (unless they have intentionally disabled Privacy Mode). In the near future, they plan on further mitigating the risk for extension users by building a UI allowing the user to initiate a force address exposure.

Verification

Resolved .

Suggestion 6: Require TLS for Opening dApp Deeplinks

Synopsis

Deeplinks for dApps can be opened via cleartext HTTP, which is easily susceptible to interception and manipulation. For dApps that interact with a user’s funds, this is very undesirable and dangerous.

Mitigation

Either enforce that deeplink use HTTPS or display a warning to inform the user that communication with the dApp could be monitored or manipulated if HTTP.

Status

Deeplinks are now forced to use HTTPS.

Verification

Resolved .

Suggestion 7: Remove Links to Questionable dApps

Synopsis

Some of the URLs in the dApp URL list are insecure (no HTTPS) and several are for projects of questionable value (such as seemingly self-aware Ponzi schemes).

Mitigation

Have a more strict curated list of dApps to include in the application. Appearing to promote questionable projects can confuse users into accepting possible scams as valid when compared to things like "PonzICO" and / or similar.

Status

DApp list was rewritten to include well-known dApps instead of the large unverified list.

Verification

Resolved.

Suggestion 8: Improve the isDecimal Number Utility

Synopsis

Using a regular expression for number validation may be confusing and could lead to issues during type coercion.

Mitigation

Avoid type coercion and use JavaScript's built in number validation:
`Number.isFinite(parseFloat(value)) && !Number.isNaN(parseFloat(value))`

Status

Switched from regex to using JS number validation.

Verification

Resolved.

Suggestion 9: Do Not Send Regular Logs to Crashlytics

Synopsis

There is code in place in the logger that suggests there are plans to push all application logs to the Crashlytics API. This is a privacy concern.

Mitigation

Only submit errors and crash reports to Crashlytics and only do so as an opt-in preference for the user.

Status

Following further discussion of the suggestion with the MetaMask team, it is understood that the public beta version of the app will include an onboarding screen that requests a user's consent to send error logs to Crashlytics, and only if the user opts-in will the error logs be sent to Crashlytics.

Verification

Void .

Recommendations

We recommend that any unresolved or partially resolved *Issues* and *Suggestions* stated above are addressed as soon as possible.

Additionally, the use of packages should be continuously reviewed for further mitigation of the risks introduced. As noted in the *Code Quality* section, with so many external dependencies, there is much opportunity for both unintentional vulnerabilities and direct attacks to users funds. Periodically running `npm audit fix` to take care of reported vulnerabilities automatically is a good practice and carefully evaluating new dependencies before introducing them is vital.

We found that the overall design demonstrated that MetaMask is being developed with security in mind. We commend this practice and recommend that future development releases of the application continue to apply security best practices and that additional security audits be conducted to address any potential issues and vulnerabilities.