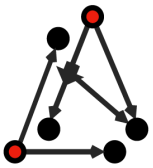

Least Authority
Freedom Matters

Security Audit Report

Melonport

Report version: 23.12.2017

Overview

Melonport has requested Least Authority perform a security audit of their Melon protocol implementation in anticipation of their move to the Ethereum Mainnet. Melon protocol is a blockchain protocol for digital asset management built on the Ethereum platform. For explanation of the protocol, see https://github.com/melonproject/greenpaper/blob/master/melonprotocol.pdf.

Coverage

Target Code and Revision

For this audit, we will look at the Melon protocol (https://github.com/melonproject/protocol) implementation, including the Melonport core (set of smart contracts) and the modules (also smart contracts) created by Melonport.

Specifically, we examined the Git revisions:

```
197116671b1144912667f19e361b38916c20645b
```

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

Manual Code Review

In manually reviewing all of the contract code, we looked for any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also kept an eye out for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus was on the contract code, we examined dependency code and behavior when it was relevant to a particular line of investigation.

Our investigation focused on the following areas:

- Any attack that impacts funds, such as draining or manipulating of funds and/or results in unbalanced trading
- Other ways to exploit contracts
- Interactions with 3rd party contracts
- Anything else as identified during the initial analysis phase

The files we manually reviewed included:

- src/Fund.sol
- src/assets/Asset.sol
- src/assets/Shares.sol

- src/compliance/Compliance.sol
- src/dependencies/DBC.sol
- src/dependencies/Owned.sol
- src/exchange/adapter/simpleAdapter.sol
- src/exchange/thirdparty/SimpleMarket.sol
- src/modules/ModuleRegistrar.sol
- src/modules/SimpleCertifier.sol
- src/pricefeeds/PriceFeed.sol
- src/riskmgmt/RiskMgmt.sol
- src/system/Governance.sol
- src/version/Version.sol

Findings

Code Quality

The code is well organized and not too long, following ethereum best practices and avoids known bugs such as re-entrancy, it does not depend particularly on economic assumptions about what is or is not in the rational self interest of traders. The main source of concern is that it interfaces with many 3rd party contracts and needs safety measures if they do not behave as expected.

Issues

We list the issues we found in the code in the order we reported them.

Issue / Suggestion	**Status**
Issue A: Funds are vulnerable to owning malicious assets	Reported: 20/12/2017
Issue B: Funds are exposed to vulnerabilities in modules	Reported: 20/12/2017
Issue C: Funds are vulnerable to re-entrance from modules	Reported: 20/12/2017
Issue D: Use of confusing module names	Reported: 18/12/2017
Issue E: Reputation of module authors	Reported: 14/12/2017

Issue A: Funds are vulnerable to owning malicious assets

Reported: 20/12/2017

Synopsis: A fund is exposed to risk that any owned asset is malicious. Any third-party contract, but especially assets, should be regarded as untrusted code.

Impact: It can become impossible to successfully call any method that iterates over held Assets: `calcGav`, `redeemOwnedAssets`, which affects `performCalculations`, `calcSharePrice`, `executeRequest` (to buy or sell shares). This effectively freezes the fund, although shares can still be transferred, as they are valid ERC223 tokens.

Since the risk of this vulnerability is that *any* asset is malicious, if a large amount of assets are held the risk increases significantly. Fund managers are likely to estimate their risk in proportion to the amount held, but this does not work like that. Holding *any* amount of a malicious asset means the entire fund is at risk of being frozen.

Preconditions: An attacker creates a ERC20/223 token with a backdoor that allows them to disable correct behaviour on any asset method, for example by making `transfer` or `balanceOf throw`, then they socially engineer fund managers into holding some of that asset. It's also possible to exploit this vulnerability by hacking an asset that the fund already holds. Although, it would be unlikely that an honestly written but hacked contract really fails in the worst way to trigger every precondition for this vulnerability. It would also be far easier to construct a backdoored asset, identify fund managers pursuing high-risk strategies and invite them to hold your malicious asset.

Feasibility: Since the main obstacle is just to socially engineering fund managers, and given that Melonport significantly lowers the barriers to becoming the manager of an investment fund, this attack seems highly feasible.

Technical Details: Melonport funds can hold up to 90 assets, each of these assets is a third party contract that must execute its own code as part of the normal fund operation. If a fund holds an asset that later stops behaving correctly, various fund functionality is disrupted. The malicious creator of an malicious asset would then force the fund manager to pay a ransom to regain access to their fund.

If `EvilAsset#transfer` returns false, then `redeemOwnedAssets` will call `revert` instead of succeeding, meaning investors cannot redeem fund assets. If `EvilAsset#balanceOf` throws an error then `calcGav` (gross asset value) will fail, which means the value of the fund cannot be calculated. If `EvilAsset#approve` fails, it's not possible to trade that asset, but this does not affect the ability to trade other assets.

Mitigation: As the code stands, fund managers must be very careful to audit every asset they hold. To decrease the risk of holding malicious assets it's advisable to hold fewer assets, as a chain is only as strong as its weakest link, so a shorter chain is probably stronger than a longer chain. This vulnerability does not disrupt the ability to transfer shares of the fund (since they are ERC223 tokens). If suspicious assets are held into by a high risk fund, and then shares in that fund are owned by the main fund, and the high risk fund is frozen by a malicious asset, then the main fund will still operate correctly. However this fund structure must be put in place before the malicious asset turns malicious.

Remediation: This vulnerability is difficult to remediate without increasing the complexity of the fund or giving the fund manager arbitrary power. The key is that it needs to be possible to operate the fund without calling any methods on the malicious asset. The simplest way to achieve this would be to give the fund manager the ability to disable or burn an asset- to just dump it out of the holdings, temporarily or permanently.

Permanently burning the asset by adding the asset to a list of assets that can now never be held again by the fund would mean the fund manager could not abuse this to manipulate the value of the fund, because the action being one-way would disincentive a rational fund manager from dumping a non-malicious asset.

Another remediation would be to have the ability to wrap any asset in a holding contract. This would be like a fund that only held one asset. It would enforce correct ERC223 behaviour, and ensure that it's always possible to call `transfer` or `balanceOf`. If you called `transfer` on the holding contract, that would attempt to call `transfer` on the held asset. However, iif that failed, it would instead transfer a share of the holding contract, which could be redeemed if or when the held asset starts working again. This could also be useful for assets which possibly limit transfers in some way, such as, only after a given time.

Note: It's also possible that the other modules are malicious, but given that modules are selected only at the creation of the contract it's somewhat a smaller risk.

Status: *Partial remediation.* Melonport chose a remediation whereby investors can call `redeemOwnedAssets` with a list of assets, where they forfeit assets not in the list. Although, this is a satisfactory remediation, there must be no iteration over the held assets (malicious assets must be fully excluded) otherwise the call will not succeed.

Verification: *Still vulnerable.* First attempt at remediation was a helpful but did not eliminate iteration over held assets.

Issue B: Funds are exposed to vulnerabilities in modules

Reported: 20/12/2017

Synopsis: Functionality that is critical to Melonport funds are supplied by third-party modules selected by the fund manager. Any vulnerability in these modules can affect

Melonport funds using them. Therefore, it is critical that fund owners have the tools they need to make good security decisions.

Impact: Vulnerabilities in modules can effect fund functionality in various ways. Problems with risk management or compliance modules is unlikely to have catastrophic effects. Vulnerabilities in pricefeed or exchange modules could cause unprofitable trades to be made, or to manipulate the calculated value of the fund. Popular modules are likely to be high value targets.

Preconditions: A vulnerability exists in some module(s) used in one or more funds and someone exploits them. This includes potentially desirable features, such as upgradability and ownership of the module code.

Feasibility: Exploiting modules is likely to be difficult, but they are also high value. There have already been some improbably spectacular hacks of Ethereum contracts, so we must assume that it is indeed feasible.

Technical Details: There are many places vulnerabilities could be inserted and hidden in Ethereum contracts. For example, Solidity is a compiled language that "should" be deterministic, but there are various versions of the compiler which may apply different optimizations to save gas, creating different EVM contracts. If compilation is non-deterministic, it's hard to detect trojans in the compiler, which is unlikely but would be devastating. It's also hard to detect a malicious EVM inserted after compilation, which would be easy to detect if compilation is deterministic and is checked. The Solidity code still needs to be audited and automated tools such as Oyente help, but they cannot detect if the contract is just malicious.

Mitigation: Each module requires careful auditing and this is time consuming and expensive. However, auditing tools can be created to make this easier, and there is information that can make this easier on auditors. In particular, we advise storing the Git commit of the module in the registry so that it's possible for auditors to check deterministic compilation. This allows auditors to audit the Solidity code, and not the EVM.

It is probably a good idea to consider escape hatches to redeem a fund even if the pricefeed and other essential modules have failed. Some notable high profile attacks, such as the Parity attack by devops199, only resulted locked contracts. If an attacker caused a module pricefeed or exchange module to self destruct, the fund could not operate as normal. For example, if it `redeemOwnedAssets` didn't call `allocateUnclaimedReward` then it would not depend on any modules, and this would mean investors could get their shares out. They would have to exchange their share of the fund's assets themselves, but they don't lose their money.

If an attacker gains total control over the exchange it may be possible for them to steal funds in some way. However, being able to `redeemOwnedAssets` will probably allow investors to save some of their money.

Remediation: It's not truly possible to fully remedy this problem. Although tooling can be created to assist evaluating the security of modules, fund managers need to understand the seriousness of selecting modules carefully and the benefit of having them be well audited.

Status: Storing git commit is implemented.

Verification: Mitigation of storing Git commit hash is verified.

Issue C: Funds are vulnerable to re-entrance from modules

Reported: 20/12/2017

Synopsis: Melonport code largely designs to prevent re-entrance from calling contracts. Calls to modules, however, permit a malicious module to re-enter.

Impact: Module developers may be able to piggyback on users' transactions and re-enter the contract. This could allow price feeds to front-run users, and/or manipulate fees.

Preconditions: A module developer must design a module to re-enter the fund contract. A fund manager must choose this malicious module when setting up the fund.

Feasibility: As with issues D and E, feasibility depends greatly on the user interface. If there is not a reliable way to identify the author of a module or audit the integrity of module code over time I would consider this attack feasible.

Technical Details: `Fund.sol` calls outside modules without validating results or taking steps to prevent re-entry from module code. This means that modules may re-enter the fund class, and update its state in advance of the results of the user's call. For instance, the pricefeed module, when called in `redeemOwnedAssets` may hold assets in the fund, and may make a sale or purchase in advance of the user's redemption of assets. The sale made by the price-feed would change the GAV of the fund, which would influence the user's redemption and the manager's fee.

Remediation: User interface must reflect that modules are potentially dangerous code. Fund managers must be aware that using modules made by third-party developers risks user funds. Modules listed on a public market should be made to undergo a third-party review process.

Status: Not addressed.

Verification: Not done, yet.

Issue D: Use of confusing module names

Reported: 18/12/2017

Synopsis: Modules can be published with the same or similar names as a phishing attack, which may induce a fund manager to select a vulnerable or malicious module. Confusing names can also be used for injection attacks on pages that describe or let managers select modules.

Impact: Fund managers may be fooled into selecting insecure modules.

Preconditions: An attacker publishes a module that looks very similar to a popular module but is malicious, and hopes that a fund manager selects it for their fund.

Feasibility: The feasibility greatly depends on how the user interface presents modules. If there are inconsistencies in information about the modules throughout the module selection process, it could easily go unnoticed.

Technical Details: Module names can be an arbitrary binary value, assuming utf8 this can include values that look the same, but differ in ways not visually obvious, such as the zero width joiner character. Also, in other module systems vulnerabilities have been shown to arise because users forget punctuation in the module name, such as hyphens or underscore, or the exact capitalization is not used.

Mitigation: Module names should probably be restricted to safe ASCII characters, without punctuation. It's acceptable to show capitals and punctuation in the user interface but it should not be allowed to publish both an uppercase and lowercase version of the same name, or a hyphenated or non-hyphenated name. This mitigation could be applied in the user interface, which would just filter out and not display modules that have potentially confusing names, as long as it detects cases where the two versions of the same name are published and takes the displays the first published. It would be better if this was enforced on publish via the contract, from the perspective of being confidently audited, but this has the downside of having less chance to fix bugs.

A good search system would also help, because it can suggest the best modules, and detect typos where they probably meant something else. This could give the user a second chance to select the module they intended to select.

Remediation: It is very difficult to fully remedy this vulnerability, but with enough mitigation it would become a unprofitable attack vector.

Status: Not verified.

Verification: Not done, yet.

Issue E: Reputation of module authors

Reported: 14/12/2017

Synopsis: A different address is used to publish each module, making it difficult to acquire reputation as a module developer. The lack of a robust identity for module authors lowers the bar for impersonation attacks.

Impact: It is harder for fund managers to know whether they should trust a given module, which may lead them to using a vulnerable or malicious module.

Preconditions: Overwhelmed with choice, a fund manager chooses a module from a developer they recognise, but actually an attacker has made a malicious module that appears to be by someone else.

Feasibility: As with Issue C, feasibility depends greatly on the user interface. If there is not a reliable way to identify the author of a module, this attack is more feasible.

Technical Details: It is actually possible to publish multiple modules as the same address, but a map type is used to point from the publish address to the last module published, which implies that the module developer has only one module. This was done to avoid loops and module authors should just use a separate identity per module. This property is not actually used in another other Solidity code in the protocol repo, so it must be assumed that it's used by the user interface.

In the mitigation of Issue B storing the Git commit hash and GitHub repo was implemented. This is a good move, but if there isn't a more robust way to identify the author of a module, a link back to this repo will probably end up in the UI and managers shopping for modules will click it. A link from a reputable GitHub repo should not be considered as positive proof of the reputation of the commit, since only the head of the master branch is really expected to be correct. An attacker could get a malicious commit onto a good repo by making a pull request that fixed a legitimate problem (or maybe just typo) but across several commits. In the malicious commit they would add a backdoor, but then in a good commit they'd remove it. When the maintainer evaluates the pull request they would not look at every individual commit, but at the difference between the master branch and all commits combined.

Remediation: Allow and encourage module authors to publish under from the same address consistently, or some way they can cryptographically sign their modules. Instead of tracking the last module published by an address, just have a list of modules which is iterated over when the UI reads from the contract state. Although iteration is slow, it's not actually used inside the contracts (just the UI) so gas cost is not a concern.

Status: Not addressed.

Verification: Not done, yet.