GOLOM Protocol Ethereum Contracts Security Audit

June 11th, 2022 (updated July, 1st 2022)

@lucash-dev

https://hackerone.com/lucash-dev

https://github.com/lucash-dev

lucash.dev@gmail.com

Disclaimer

Security audits are inherently limited -- there is no possibility of strict proofs that a given system doesn't contain further vulnerabilities. The best that can be offered is a good faith effort to find as many vulnerabilities as possible within the resource constraints of the project.

Given the above, this audit report is provided "AS IS", without any guarantee of correctness or completeness. The author takes no responsibility for any loss or damage caused by the use, misuse or failure to use the information contained in this document, as well as for any information that might be missing from it (e.g. missed vulnerabilities).

By using the information contained in this report, you agree to do so at your own risk, and not to hold the authors liable for any consequences of such use.

This audit does not mean endorsement of any product, service, or investment.

Summary

This document contains the results of the audit conducted on the DSLA Protocol Ethereum smart contracts.

This document is divided in the following sections:

- **Scope**: brief description of the scope of the audit.
- **Previous work**: reference of relevant previous work by the reporter.
- **Methodology**: a general description of the methodology used to find possible issues.
- **Attack Scenarios**: a non-exhaustive list of potential attack scenarios/impacts considered during the audit.
- **Exploitable Vulnerabilities:** issues found that can be exploited using clearly identified attack vectors.
- **Improvement Suggestions**: a list of issues that, if corrected, might lead to increased security, though there is no evidence they currently present a security risk

as they were found.

Scope

The scope of this audit are the smart contracts found at the GitHub repo `https://github.com/golom-protocol/contracts` as of commit `17c77d13d386b33d6012a8ae9e1bc4788f8a3ed5`.

The audit is focused on the *implementation* and *logic* of the smart contracts code, in particular in finding *viable attack vectors* that can be used for a practical exploit. While some improvement suggestions regarding code quality and other factors not directly related

The following items are a non-exhaustive list of elements that are outside the scope of the audit:

- Vulnerabilities on other systems interacting with the smart contract (such as web apps).
- Vulnerabilities in the EVM implementation or the Ethereum platform in general.
- Weaknesses in the underlying cryptographic primitives (hash functions, digital signature schemes).
- Economic soundness and financial feasibility of the protocol.
- Ownership structure of governance tokens or privileged cryptographic keys (if applicable).

Previous Work

To illustrate the experience of the author, here are a few links to previous audits and responsible disclosures:

- Stacktical Protocol Audit (2021): https://storage.googleapis.com/stacktical-public/audits/audit1v2.pdf
- Unbound Finance Audit (2021, mentioned, text not publicly available): https://unboundfinance.medium.com/unbound-finance-roadmap-6acaf40f878e
- MCDEX responsible disclosure (2021): https://medium.com/immunefi/mcdex-insufficient-validation-bug-fix-postmortem-182fc6cab899
- ZAP Finance responsible disclosure (2021): https://medium.com/immunefi/zapper-arbitrary-call-data-bug-fix-postmortem-d75a4a076ae9
- Switcheo Audit (2020): https://github.com/Switcheo/carbon-polynetwork-evm/blob/ca27ab529263368fb52e6532b6f6adbfbacb31f1/audits/audit_2020_12_17.pdf
- Switcheo Audit (2019): https://github.com/Switcheo/switcheo-eth/blob/master/audits/v2/LucashDev-v2_audit.pdf
- MakerDAO responsible disclosure (2019): https://hackerone.com/reports/684092

The list above is far from a complete collection of research, as most of audits and responsible disclosures aren't publicly available.

Possibly relevant, here is a list of older contributions to Bitcoin Core (2018-2019) by the author:

- https://github.com/bitcoin/bitcoin/pull/14696
- https://github.com/bitcoin/bitcoin/pull/14906
- https://github.com/bitcoin/bitcoin/pull/13443
- https://github.com/bitcoin/bitcoin/pull/13419
- https://github.com/bitcoin/bitcoin/pull/13404

Methodology

The audit was conducted manually by an experienced security researcher, by inspecting the code in two different ways:

- Systematic review of every function in the contracts, spotting missing best practices and logical flaws.
- Free exploration of interaction points and execution flows, searching for concrete attack vectors and invalid assumptions.

While the first approach mimics a more traditional code review, the second one tries to reproduce the kind of coverage you would obtain from a "bug bounty" program.

It's the author's belief that combining these approaches offer much higher likelihood of catching high-severity issues than using either individually.

Attack Scenarios

To give an idea of the possible attacks considered during the audit, we present a non-exhaustive list of attack scenarios that were evaluated.

No vulnerabilities were found corresponding to the *attack scenarios listed* in this section.

Please refer to the next sections for issues that were *found*.

Trader

- Unauthorized transfer from user.
- Incorrect validation of cryptographic signature.
- Second pre-image attack on signed orders (maleabillity).
- Order replay.
- Second pre-image attack on Merkle tree implementation.
- Reentrancy on order processing leading to loss of funds.
- Insufficient validation of amounts provided by caller.

Rewards/Airdrop

- Second pre-image attack on Merkle tree implementation.
- Balance manipulation of VE contract leading to minting of excess tokens.
- Unauthorized claiming of rewards or airdrop tokens.

Voting Escrow

- Voting by unauthorized delegatees.
- Unauthorized transfer of voting NFTs.
- Unauthorized merge of voting NFTs.
- Numerical flaws leading to incorrect vote amounts.

Governance/Timelock

- Double-voting.
- Replay of proposals.
- Execution of proposal without quorum.
- Execution of proposal before mandatory delay.
- Partial execution of proposal.
-

Exploitable Vulnerabilities

Governance Proposal DoS due to Transaction collision

The Timelock (`Timlock.sol`) contract uniquely identifies a queued transaction by the hash of its parameters `target, value, signature, data, eta`. This hash is used for keeping track of transactions being queued, executed, or cancelled. On the other hand, the Governance contract (`GovernerBravo.sol`) allows any user with enough voting power to create new proposals (function `propose`) containing arbitrary transaction data (`target, value, signature, data`)-- that isn't validated.

Since a proposer can cancel their own proposals by removing their voting power before it's executed, it is possible for one proposer to de-queue the transactions of another, if there are transactions present in both proposals.

This attack feasilibility is limited by the need for a quorum, as well the `eta` of both sets of transactions being the same -- which can be achieved by queueing in the same block, or by other manipulations of the TimeLock delay. There's also the possibility that legitimate users might create similar proposals (with some identical transactions) at the same time, and then enqueuing them at the same time as soon as voting ends.

In these scenarios, the Guardian would be at a loss to determine which proposal is legitimate and wich one isn't and should be cancelled before enqueing. If a policy of canceling both proposals is used, then it becomes even easier for a malicious proposer to DoS other proposals.

If the ability for a proposal to *veto* a different proposal is intentional, it should be explicitly implemented in a separate function and properly documented.

Suggestion: an easy fix for this issue would be to record used eta's in the Governor and prevent eta collision between proposals. That would still leave the possibility of temporarily DoS-ing proposals being enqueued by front-running and enqueing other proposals with same eta -- however, that would at most delay the legitimate proposal by a few blocks.

Ideally, the Timelock should modified to include a proposal id in the transaction hash.

Update: the development team fixed the issue by preventing more than one proposal from being enqueued in the same block.

Potential Vulnerabilities and Improvement Suggestions

This section contains suggestions for improvement of the contracts and issues that don't represent an immediate security risk at this moment.

1. EIP712 Implementation

The EIP712 implementation in the Trader contract doesn't follow perfectly the standard. The deviation is that the chain Id used is hard-coded. While it is unlikely this will cause security issues, there is a small likelihood deployments to different chains (e.g. mainnet and testnet) might end up using the same contract address, which would allow replay attacks across the chains.

Suggestion: modify the code to read the chain Id from the EVM, rather than hardcode it.

Update: this issue was fixed by the development team, by following the suggestion above.

2. Unused Order parameter in Airdrop

The `order` paramater in the `claim` function in the Airdrop contract is used exclusively for validating it's signature against `msg.sender`. The only practical effect of having this parameter is enforcing only EOAs can execute `claim` -- as any user can generate a random "order" then sign it.

If checking the EOA status was the intended use case, there are other ways of enforcing the EOA status of `msg.sender` such as checking `tx.origin` or a signature over a specific message (or even arbitrary bytes).

Suggestion: remove the `order` parameter from `claim`. If a test for EOA is needed, it should be explicitly implemented as either a check on `tx.origin` or a signature over a specific message.

Update: the issue was fixed by the development team by removing the `order` parameter from the `claim` function.

3. Repeated Logic

There are several copied and pasted pieces of code -- in particular Order validation logic -- repeated multiple times across the code base.

While it doesn't seem the case right now, that repetition increases the likelihood a future change will make these multiple implementations different, possibly introducing

vulnerabilities.

Suggestion: remove repeated code, rather using a base contract or calling external views to reuse the Order validation logic.

Update: the issue was acknowledged by the development team. However, refactoring code was considered too risky as it might introduce novel issues.

4. Reentrancy in Trader:

The only reentrancy protection in the trader "fill" functions is the check that the total isn't yet filled. While the code *does* follow the checks-effects-interactions pattern, which should make all interactions idempotent, it is still possible to change the state of orders from within filling the order.

While no way to directly exploit this was found -- it might lead to confusing scenarios, in particular, confusing reordering of logs (for example, an order seems to be filled after it was cancelled), which might lead to hard to predict scenarios in downstream systems (e.g. UI, bots).

Suggestion: Adding an explicit reentrancy check would prevent these issues (and also any possible violation of the checks-effects-interactions pattern not caught in the audit) with minimal cost.

Update: the issue was fixed by adding reentrancy guards as suggested above.

5. Merkle Tree Hardening

While the Merkle Tree implementations in the code base don't seem to be vulnerable to second-preimage attacks, future changes to formats of leafs might introduce this issue.

To prevent introducing issues in future versions of the code, we recommend introducing *explicit* domain separation between leafs and nodes (concatenating a prefix or tag to the value before hashing). **Important** we recommend these changes be made **only** if further modifications to the MerkleTree code are necessary.

Update: the issue was acknowledged by the development team. As no further change to the code was needed, they took note of the potential for introducing that issue as something to review in future iterations of the code.

Conclusion

One exploitable vulnerability was found during this audit and described in the report. Some of the many potential attack scenarios evaluated were listed, and some potential issues and possible improvements were discussed.

The issues pointed out were either fixed or acknowledged by the development team, as noted in updates above.