CertiK Audit Report for Streamr

CertiK Eng Team
Mar 3, 2020
Version 2.0.2

Executive Summary

We have performed a detailed review of the code provided by the client, including, but not limited to, the following vectors: overflow & underflow, incorrect control flow, incorrect arithmetics, reentrancy attacks, replay attacks, and others, and have not found any major or minor vulnerabilities in the system.

Scope of Audit

The scope of the audit was the contract Monoplasma with its full Solidity inheritance chain.

The file that was audited, and the file that all line numbers refer to, is:

- https://github.com/streamr-dev/monoplasma/blob/8f3cecec5c9dcc3ddd340a7173eabf3089a8e53c/contracts/Monoplasma.sol

Analysis

Roles

The contract has two roles:

- owner (appoints operator and sets their commission)
- operator (commits blocks and collects commission. Note that this is not enforced on a smart contract level)

We have not found any vulnerabilities regarding the authoritative roles of the contract.

Commits

The contract uses BalanceVerifier's `commit` function and a custom `onCommit` hook.

The client should note blockNumber's can be thought of in full generality as a set - blockNumber is a unique identifier rather than an order imposer. It depends on the intentions of the client whether this is desired.

We have not found any vulnerabilities regarding commits.

Proofs

The contract uses BalanceVerifier's `prove` function and a custom `onVerifySuccess` hook.

In general Merkle tree's are susceptible to "second preimage attacks" unless hashing of the leaf elements is performed during the proof. That is the case here, so we don't think it is suceptible to this form of attack. We have not found any other attack vectors.

Withdrawals

The contract offers rich functionality in terms of withdrawing.

All functions end up calling `_withdraw` which executes the withdrawal. We have not found any vulnerabilities in this area.

Holistic analysis

Let us use:

- *max provable* = the maximum balance across all commits for an account
- *total provable* = the sum of max provable for all accounts
- *total withdrawable* = total provable amount - totalWithdrawn,
- *net balance* = contract balance - total withdrawable,
- *safe situation* to denote the case when the net balance is ≥ 0.

Note that the net balance can only increase between commits or remain the same (i.e. cannot decrease), simpliy because the contract doesn't have a way to transfer / approve tokens without increasing totalWithdrawn.

Let us assume we are in a safe situation. When the operator publishes a block, one of the following must be true:

1. net balance remained the same since the last commit
2. net balance increased since the last commit

Case 1

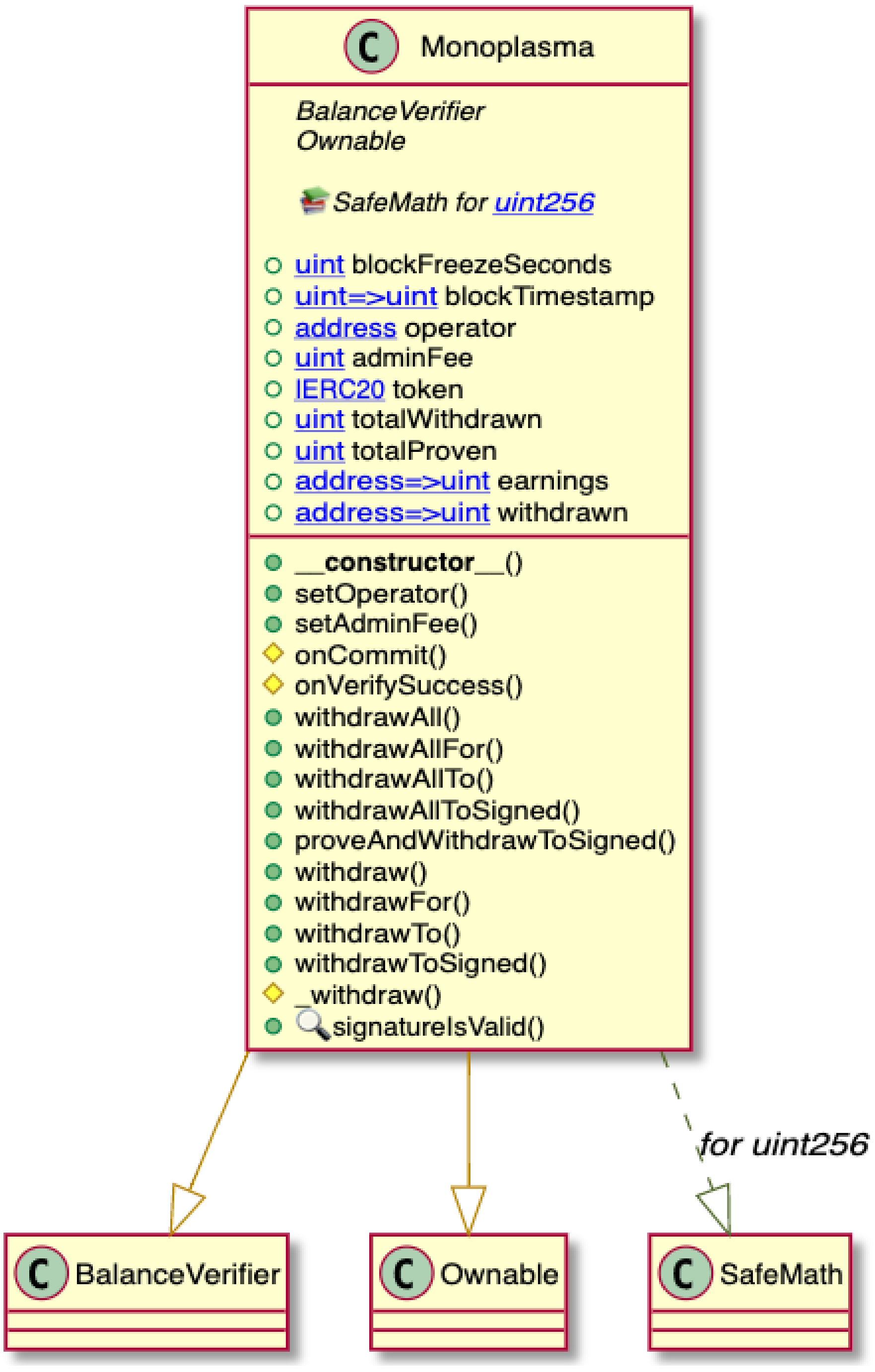
Let's assume the commit leads to an unsafe situation. Regardless of data availability, users can reference the previous safe commit and safely withdraw.

Case 2

If the operator publishes a tree that users do not agree with (e.g. by increasing the max provable of an account controlled by the operator), then they can withdraw similarly, leaving only those that agree with the new tree. The unsafe situation case is similar to the one above.

The only threat vector we see here is if the time interval (freeze period) is not sufficient for everyone. However, given there is a trade-off for the topic of usability, we haven't found a way to unilaterally solve it.

Appendix A: Overview graph



Monoplasma
BalanceVerifier
Ownable
SafeMath for uint256
uint blockFreezeSeconds
uint=>uint blockTimestamp
address operator
uint adminFee
IERC20 token
uint totalWithdrawn
uint totalProven
address=>uint earnings
address=>uint withdrawn
__constructor__()
setOperator()
setAdminFee()
onCommit()
onVerifySuccess()
withdrawAll()
withdrawAllFor()
withdrawAllTo()
withdrawAllToSigned()
proveAndWithdrawToSigned()
withdraw()
withdrawFor()
withdrawTo()
withdrawToSigned()
_withdraw()
signatureIsValid()
for uint256
BalanceVerifier
Ownable
SafeMath