Common Prefix

1st Audit Report for ONEWallet

Dimitris Karakostas, Andrianna Polydouri, Dionysis Zindros

July 31th, 2021

Executive Summary

The current implementation of ONE wallet proposes an OTP-based wallet which uses a smart contract implementation to maintain user funds. The main goal of the wallet is to offer reasonable security with great usability.

At this stage of implementation, the security of ONE wallet relies entirely on the security of the wallet's client, in the sense that if the client is compromised then the wallet is exploitable with a rather small amount of computational resources. For an enhanced, two level security construction several solutions are considered by Harmony and are planned to be launched in early August 2021. The wallet proposal also includes several additional mechanisms to ensure multi-factor security, such as guardians and social factors. These mechanisms are planned to be implemented in a future version of the wallet.

For this audit report, we assume that the wallet's client remains secure and consider attackers that try to exploit any weak points of the protocol and its implementation along with the inherent weaknesses of the blockchain construction (e.g. front-running attacks). We focus on possible vulnerabilities under this threat model but also provide suggestions that may help make the smart contract's functionality more efficient or help for a clearer and more easily maintainable codebase.

Smart Contract Implementation

We inspected Harmony's ONE wallet implementation at commit hash 485ca526d347f30a7a539713b10642ad0c4aac67. We mainly focused on security vulnerabilities under the current assumptions of the wallet's threat model described in the Executive Summary section.

Critical severity issues *(Resolved)*

1. One of the three reveal functions in ONEWallet.sol does not check the correctness of the Merkle proof provided. Specifically,
   `function revealSetLastResortAddress(bytes32[] calldata neighbors, uint32 indexWithNonce, bytes32 eotp, address payable lastResortAddress_) external`
   is missing the `isCorrectProof` modifier, resulting in a critical security issue.

We consider this issue to be of critical severity as an attacker could set an arbitrary last-resort address and drain the wallet upon the end of the wallet's lifespan.

Alleviation
This issue was immediately reported to ONE wallet's developer, Aaron Li, and was fixed right away in https://github.com/polymorpher/one-wallet/commit/23072934322033e3f702dbf53d953eb74d2d4bb4.

High severity issues *(Resolved)*

1. Front-running

A reveal transaction is accepted as long as it is posted within 60 seconds after the corresponding commit transaction (because of the requirement for successful execution of `ONEwallet::isRevealTimely()` upon revealing).
As a result, a malicious party can perform a front-running attack, as follows.

Assume that the wallet's owner A wants to perform a transfer $\tau = (P^t, Q^t, tr_{amount}, tr_{dest})$. First, A's client commits to it, by creating a transaction t_1 that calls the smart contract's *commit* function with the hash of τ. Following, as soon as A's client receives confirmation that the commitment transaction is finalized, it creates a transaction t_2 that calls the `revealTransfer` of the smart contract and contains τ.

The attacker behaves as follows:

1. **Malicious relayer:** Upon receiving t_2, the attacker (who controls the relayer) does not publish it on the ledger.

2. **Front-running attacker**: Upon observing t_2, the attacker publishes the two front-running transactions with much higher gas than t_2 (such that miners prioritize them).

In both cases, the attacker creates the following two transactions:

1. A transaction that commits to (P^t, Q^t, tr'_{amount}, tr'_{dest}), where tr'_{dest} is an address controlled by the attacker.
2. A transaction that calls `revealTransfer` with arguments (P^t, Q^t, tr'_{amount}, tr'_{dest}).

Observe that the attacker's transactions are successfully executed if:

1. They are executed before t_2 (which is ensured as above).
2. The OTP Q^t is valid upon commitment (which is true, since the client does not wait for enough time before broadcasting t_2).

As a result, the attacker steals tr'_{amount} from the wallet (an amount possibly up to the daily limit).

Vulnerable operations:

The OTPs are used for three operations:

1. Transfer of funds (`revealTransfer`).
2. Setting the recovery address (`revealSetLastResortAddress`).
3. Draining the wallet, by transferring its balance to the recovery address (`revealRecovery`).

Therefore, the above front-running attack can be used to: i) set the contract's recovery address to an adversarial one; ii) forcibly drain the contract (possibly to an adversarial address, if two front-running attacks are performed).

To prevent such attacks we suggest that an extra check be added in each reveal-related function, requiring that the interval period (30 seconds) since the commit transaction has passed.

We consider this issue to be of high severity because of its possible impact (loss of user's funds) but also due to the increasing frequency of front-running attacks nowadays.

Alleviation

A forced 30-second delay between the two stages for a transaction's execution was considered unacceptable by ONE wallet's developer, for efficiency reasons. Instead, it was decided to slightly alter the commitment scheme in order to mitigate such front-running attacks. The corresponding changes can be found at https://github.com/polymorpher/one-wallet/pull/56.

However, Commit-Reveal team found that this new version allowed for another front-running related vulnerability. In this attack the user's funds are not in danger, but the attacker can delay the user's transactions for an arbitrary period of time, i.e. causing a DoS. The detailed discussion can be found at https://github.com/polymorpher/one-wallet/issues/59.

Harmony immediately responded to this issue by applying a patch, which can be found at https://github.com/polymorpher/one-wallet/pull/60.

Low severity issues *(Dismissed)*

A number of timing hazards are possible in the current implementation.
The smart contract relies on block timestamps to:

1. Verify whether a reveal is timely (*_isRevealTimely*)
2. Cleanup commits (*_cleanupCommits*) and nonces (*_cleanupNonces*)
3. Drain the wallet after its lifespan expires (*retire*)
4. Calculate the total daily transferred amount (*revealTransfer*)

However, miners can manipulate the block's timestamps and possibly result in unexpected behaviour. For example, a miner may set a large timestamp to force a reveal to fail or retire the wallet ahead of time.
We consider these hazards to be of low severity because miners are not generally expected to be easily motivated to tamper with a block's timestamp.

Alleviation

ONE wallet's developer decided to dismiss these issues as such attacks are considered rather unlikely in Harmony's blockchain ecosystem, where miners are strongly disincentivized to tamper with timestamps.

Suggestions

We provide some suggestions that we believe would make the code more readable and efficient.

1. NatSpec Format for comments *(Partially Resolved, Partially Dismissed - the contract was enriched with @notice comments. The full functionality of NatSpec format for special purpose comment annotation is considered unnecessary at this point)*
   Consider providing comments in the Ethereum Natural Language Specification (NatSpec) for more readable and easily maintainable code. Solc compiler can parse comments in this format and produce documentation in JSON files.
   https://docs.soliditylang.org/en/v0.8.6/natspec-format.html

2. Immutable variables *(Resolved)*
   Many of the contract's state variables are constants in the sense that they are assigned during construction and never change their value. Specifically, these variables are the following:

```
bytes32 root
uint8 height
uint8 interval
uint32 t0
```

```
uint32 lifespan
uint8 maxOperationsPerInterval
```

We suggest that these variables are declared `immutable`. Reading immutable state variables is significantly cheaper than reading from regular state variables, since immutables are not stored in storage, but their values are directly inserted into the runtime code.

3. Gas-efficient (safe) arithmetic operations *(Resolved)*
   Since Solidity v0.8 the compiler automatically checks arithmetic operations for overflow or underflow. While this feature enhances the security of a smart contract, it comes with higher gas costs for each arithmetic operation. This is aimless in cases where the calculations are never going to result in overflow/underflow. For example, in ONEWallet::_cleanupCommits:

```
for (uint32 i = 0; i < commits.length; i++) {
      Commit storage c = commits[i];
      if (c.timestamp >= bt - REVEAL_MAX_DELAY) {
      commitIndex = i;
      break;
      }
}
```

Subtraction `bt - REVEAL_MAX_DELAY` is never going to underflow.

Another repeated and safe arithmetic operation (`i - commitIndex`) lies in ONEWallet::_cleanupCommits:

```
for (uint32 i = commitIndex; i < len; i++) {
      commits[i - commitIndex] = commits[i];
}
```

but also `numValidIndices++` in ONEWallet::_cleanupNonces:

```
for (uint8 i = 0; i < nonceTracker.length; i++) {
      uint32 index = nonceTracker[i];
      if (index < indexMin) {
            delete nonces[index];
      } else {
            nonZeroNonces[numValidIndices] = index;
            numValidIndices++;

      }
}
```

We suggest that the safe arithmetic operations inside for-loops are wrapped in `unchecked{}` for gas-efficiency.

Reference:
https://docs.soliditylang.org/en/v0.8.0/control-structures.html#checked-or-unchecked-arithmetic