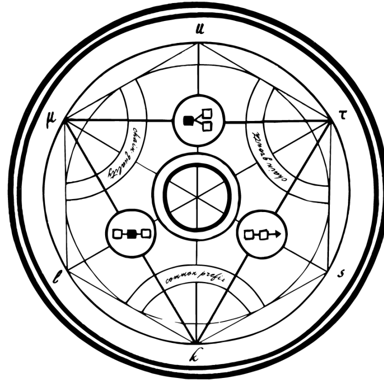

Common Prefix

Preliminary Security Analysis of ONEWallet

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 wallet proposal includes several additional mechanisms to ensure multi-factor security, such as guardians and social factors. The main goal of the wallet is to offer reasonable security with great usability.

During the audit of the wallet, the Common Prefix team identified some critical issues with it. Our main concern regarding the current version of ONE wallet is that *it does not offer any more usability or security compared to plain old private-key–based wallets*. The design and deployment of this wallet could replace the OTP-based functionality with a plain private-key–based construction without loss of functionality or a gain in security or usability.
Given the current implemented version of ONE wallet, some security issues identified at an earlier stage of this report were confirmed and resolved or currently dismissed.

For an enhanced, two level security construction several solutions are considered by Harmony. These solutions are described in the Client Security document and are planned to be implemented for a future version of the wallet. We provide an analysis of these proposals in the second part of this report.

Material

This report is based on the following material:

- https://github.com/polymorpher/one-wallet/wiki/Home/3f658a4dfe5ae36b267d3ceb59282c1182b1f6c7 (wiki)
- https://github.com/polymorpher/one-wallet/wiki/Security-Goals/c74c63a4d6fcc595d15a44b8306fbe866f7c6df5 (security goals)
- https://raw.githubusercontent.com/polymorpher/one-wallet/f6456805a43a64cd5b71cfc727d93d76da162a76/wiki/protocol.pdf (protocol)
- https://github.com/polymorpher/one-wallet/tree/414c0dbe0a3b4c6b7cfa382934e7d2d73702451b/code (code)

Analysis

Protocol

1. OTP protocol compared to private-key wallet

The current design is a design based on the SmartOTP paper. However, contrary to the SmartOTP paper, we see a few critical design differences:

1. The OTPs have few bits of security (20 bits)
2. The hashes of the OTPs must remain secret in order for security to be maintained, even for amounts as low as $0.01
3. The client must maintain the secret hashes in order to be able to spend

Consider the current wallet design W, which contains an OTP module, as illustrated in Figure 1. This wallet design can be replaced by a different wallet design W', which contains a private-key–based module in place of the OTP module. The wallet W' is better in everything compared to W.

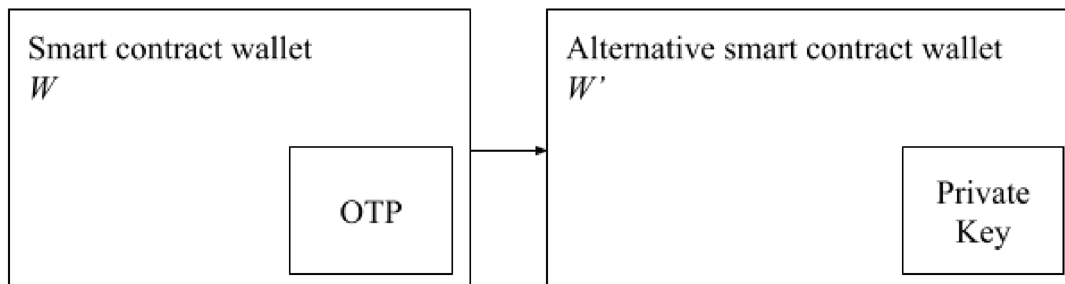

Figure 1: The current wallet design W containing an OTP module can be replaced by a different wallet design W' containing a private-key–based module in its place.

The *usability* gain, from a *user experience* point of view, in the wallet W' stems from the following observation:

1. In the OTP-based wallet, the user must both press a button to pay and enter an OTP code.
2. In the password-based wallet, the user must only press a button to pay, and does not need to enter an OTP code.

The *usability* gain, from a *performance* point of view, in the wallet W' stems from the following observation:

1. In the case of an OTP-based wallet, two transactions must be posted on the blockchain, and a sufficient time must be allowed between the two to ensure no adversary can reorg the chain. This imposes a time delay for each transaction performed. In particular, the payment requires waiting for two liveness and two safety periods[1] before it becomes stable.
2. In the case of a private-key–based wallet, only one transaction must be posted on the blockchain, imposing no additional delay between the issuance of a payment and its posting on the chain. In particular, the payment requires one liveness and one safety period before it becomes stable. As such, W' is twice as performant as W.

Additionally, in terms of *usability*, there is a difference in gas fees costs:

1. In the case of an OTP-based wallet, two on-chain transactions are required. These transactions are non-trivial and require additional gas.
2. In the case of a private-key–based wallet, only a single and simple on-chain transaction is required. This transaction spends strictly less gas than *each* of the respective OTP transactions.

While gas is not currently an issue with Harmony, it might become an issue as the blockchain becomes more popular.

In terms of *security* due to the Client being compromised, the security of the wallet W' is equal to the security of the wallet W as follows:

1. If the Client is compromised in the case of the OTP, the OTPs are trivially compromised.
2. If the Client is compromised in the case of the private key, the wallet is, again, trivially compromised.
3. If the Client is not compromised in the case of the OTP, the OTPs are secure.
4. If the Client is not compromised in the case of a private-key–based wallet, the private-key–based wallet is secure.

In terms of *security* due to a Common-Prefix–breaking adversary, the security of the wallet W' is superior to the security of the wallet W by the following argument:

1. An honest uncompromised Client's money in the case of W can be stolen in case of a Common Prefix violation. The chain can be reverted between the Commit and the Reveal phase.
2. An honest uncompromised Client's money in the case of W' cannot be stolen due to Common Prefix violations (although such violations can cause other harm in the chain).

---

[1] Garay, J., Kiayias, A., & Leonardos, N. (2015, April). The bitcoin backbone protocol: Analysis and applications. In *Annual international conference on the theory and applications of cryptographic techniques* (pp. 281-310). Springer, Berlin, Heidelberg.

In terms of security due to attack surface:

1. The W wallet contains many moving parts and a complex smart contract implementation. Even if there are no obvious bugs in the contract, there are many more locations an adversary can look for opportunities.
2. The W' wallet is a standard wallet which contains a strict subset of the moving parts of the W wallet. Hence, if the attack surface of W' contains no opportunities for compromise, then so does an implementation of W, as W would also require a handling of public key cryptography. On the contrary, a compromise of W' does not entail a compromise of W.

In terms of *functionality*, it is proposed that the OTP wallet can be used to spawn temporary wallets that have only a particular lifetime. However, the same functionality can be achieved in a private-key–based wallet, too. The mechanism to do that is to replace the OTP-based Merkle Tree with a Merkle Tree of public keys. If Harmony wishes to explore this avenue in more detail, we can provide a complete design allowing these abilities.

Lastly, we remark that, while OTP might not be the only component of the W wallet in the future, the OTP component itself can be replaced within the wallet, while leaving all other functionality intact. Therefore, if the OTP-based wallet is extended with social features, guardians, or daily limits, these can likewise be implemented in a private-key–based wallet, yielding a more secure, more usable, and just as functional wallet.

2. Client is single point of trust and failure

The major issue of the current version of the wallet is that it relies completely on the client's security.

Specifically, we consider two cases of client corruption: A) during setup, B) at any point during the execution:

A. If the Client is compromised during setup, then the seed *k* is leaked. As a result, the adversary can compute all OTPs, past and future.
B. If the Client is compromised during the execution, then the hash of the seed k_h and all Merkle Tree leaves $\{ L_i^{-j} : j = 0 \ldots d-1, i = 1 \ldots 2^{(d-j)} \}$ are leaked.

Since the OTP search space is only 10^6, an attacker with access to k_h can reverse (via brute-force) the OTP of any period and compare it to the respective leaf (this is also detailed in the "client security" and "protocol" documents). This conflicts with the properties "resilient", "sufficient", "composable", as defined in the "security goals" document.

In summary, until the "brute-force" bug is fixed, the OTP wallet seems of no added value compared to a common private key based wallet. Specifically, it does not increase security (as it

completely relies on the client's security) and adds extra complexity and cost (for performing the on-chain transactions), whereas it could be simply run on the client module.

Alleviation

Note that Harmony is aware of the security implications coming with a compromised Client and is working on solutions to strengthen the overall security. The proposed solutions are described at https://github.com/polymorpher/one-wallet/wiki/Client-Security/369d114fe00a61a02788914928aa5ac5a96aa8b3 and a combination of them is planned to be implemented in a next version of the wallet. For an analysis of these proposals refer to the second part of this report Client Security Document Analysis.

3. Front-running attack

The wiki states that: *"In practice, the Client should refrain from revealing the details (the second stage) until the OTP expires for its 30-second time window."*

However, this is:

I. not defined in the protocol;
II. not required in respect to the implementation of ONE Wallet - a reveal transaction is accepted regardless of the elapsed time since the corresponding commit transaction.

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).

Recovery address reset

The OTPs are used for three operations:

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

Therefore, the above front-running attack can also be used to: i) reset 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).

Alleviation

The protocol has been updated by Harmony's team and such front-running attacks are no longer possible. For details, refer to the next section of this document Smart Contract Implementation > High Severity Issues.

Smart Contract Implementation

We inspected Harmony's one-wallet implementation at commit hash 485ca526d347f30a7a539713b10642ad0c4aac67. We mainly focused on security vulnerabilities other than any inherent weaknesses of the protocol design.

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 easily exploit it to execute arbitrary transactions, even draining the wallet.

Alleviation

This issue was immediately reported to Harmony's engineering team and fixed right away in

https://github.com/polymorpher/one-wallet/commit/23072934322033e3f702dbf53d953eb74d2d4bb4.

High severity issues *(Resolved)*

1. Front-running

Front-running attacks are currently possible, as described above in *Protocol/Front-running attack.*

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 Harmony's team, for efficiency reasons. Instead, the team 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's team 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

Harmony decided to dismiss these issues as such attacks are considered rather unlikely in harmony's 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 - the contract was enriched with detailed comments, however not following the NatSpec format)*
   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

Differences between smart contract and protocol spec

- The smart contract performs a cleanup of old (i.e., older than 60 seconds) commits and nonces. This operation is not part of the protocol's specification.
- The smart contract allows the user to reset the recovery address, via the *revealSetLastResortAddress* function. This is not part of the protocol's specification; instead, the spec implies that the recovery address can only be set during the contract's deployment (*"if last resort address is not set when the wallet was created, the user may choose a new address to transfer the funds"*).

- The smart contract enforces an upper bound on the operations per interval (*maxOperationsPerInterval_*). This is not part of the current protocol's specification.
- The smart contract enforces a 60-second upper bound delay (*REVEAL_MAX_DELAY*), between committing and revealing an operation. Instead, the protocol set this bound to 30 seconds (Section 2.5.2 bullet (a).ii).
- In the protocol's specification, the commitment contains all internal nodes of the Merkle proof's path. In the smart contract, the commitment contains only the first neighbor of the leaf (*neighbors[0]*).

Miscellaneous

- In Section 2.2, bullet 7 of the protocol, the Client waits for 2 seconds, after a commit transaction is published on the ledger and before it reveals its commitment. This seems a rather small amount of time to wait, in order to protect against chain reorganizations. Still, it depends on the underlying ledger's properties, so it should either be dynamic or additional analysis should be made to find if it suffices.
- During recovery, the protocol states (Section 3.1.2) that the Client receives only the seed k from the user and then regenerates all information. In reality, it should also obtain t_0, T so the user should either i) supply these values directly or ii) supply the smart contract's address. In case of (ii), the variables t_0 and T are internal (L15-16 of code/contracts/ONEWallet.sol), so the client should manually parse the contract's deploying transaction to obtain them.
- The client relies entirely on Binance for price discovery (L90 of code/lib/api/index.js), a noteworthy trust assumption.

Client Security Document Analysis

Material:

- https://github.com/polymorpher/one-wallet/wiki/Client-Security/369d114fe00a61a02788914928aa5ac5a96aa8b3

Summary

In the client security document, a number of possible solutions are briefly described and analyzed, aiming to increase the security of ONE Wallet. More specifically, the goal is to prevent the client from being a single point of failure, in the sense that any attack leveraging a compromised client should fail, unless additional information from the authenticator is leaked.

Solution 1: Controlled Randomness

Overview

The core idea of this solution is to increase the search space of the brute-forcing adversary, while requiring some brute-forcing from the client (albeit over a much smaller search space).

Particularly, for each leaf, the proposed solution:

1. increases the search space for the adversary to 10^{12} and
2. requires that the client brute-forces a value from a search space of 10^{6}

The document references GPU benchmarks (https://github.com/siseci/hashcat-benchmark-comparison/blob/45a27b32a2f24d317cc29741d64fc739f3a30cb5/1x%20Gtx%201080%20TI%20with%20Overclock%20Hashcat%20Benchmark 2018) to estimate that the time needed for the adversary to break the wallet is 10 minutes.

Following, the document proposes to double the OTP size (by requiring 2 OTPs from the user), thus increasing the attacker's search space to 10^{18}, which is then estimated to require 19 years using a high-end NVIDIA GPU and ~3 hours using an AntMiner (~100TH/s).

Analysis

1. The report suggests that increasing the search space for the adversary to 10^{12} presumably delays attacks (but not stop them) and the attacker could speed up the attack to only a few minutes (or even seconds) by using modern GPUs and/or ASICs, so it seems like a half-measure only.
2. The report suggests that increasing the search space for the adversary to 10^{18} delays attacks, although again does not stop them if the adversary uses custom hardware.

3. It is unclear how requiring two OTPs from the user would work in practice. This proposal implies changes in the smart contract and the client's logic, which is unclear how much effort would require to be implemented and how they would work.

In summary, increasing the search space of the brute-force to 10^{12} or 10^{18} may appear as an improvement, but does not really prevent attacks. Particularly, since renting hash power is very cheap (see below), it is not very realistic to assume that the attacker could not control a few PH/s of mining power for a few hours. Another concern is that an altered protocol that requires two OTP inputs for each transaction is not clearly described so far and may come with other security implications.

Time-Cost evaluation

According to https://www.crypto51.app for SHA-256, renting 1 PH/hr costs $14.5, so:

- Renting 38 TH/s (which completely breaks the 10^{12} proposal in less than a second) costs a few cents (specifically $1/hr)
- To break the 10^{18} proposal in an hour costs $2100

It is not easy to conclude whether these costs are high enough to consider the wallet satisfyingly secure or not. A profound criterion would be the possible relative profit of an attacker, so in the case of $2100 cost for a transaction one could argue that a rational player would attack only if the gains exceed that cost. However, it is not trivial to reason about the incentives an attacker could have, which may be irrelevant to the immediate profits themselves. For example, consider an attacker that has huge indirect profits from disputing ONE Wallet's security.

Solution 2: Complex Hash Function

Overview

The core idea of this solution is to disrupt a brute-force attack by using a memory-hard hash function. The document proposes using argon2, for which it is argued that no custom ASIC exists and that it performs well for a small number of operations (eg. 10^6 needed by the client following solution 1).

Analysis

As with solution 1, employing a memory-hard hash function (and possibly increasing the brute force search space to 10^{12} or 10^{18}) may prohibit adversaries with limited capabilities, but does not protect against adversaries that can rent hashing power. As such, the time and cost required to successfully attack a wallet is not much and, with hardware improving and more mining rental options being provided, we expect it not to be impossible to bypass this solution.

Importantly, the document proposes to combine this solution with solution 1, but does not offer any estimations on the time/effort needed for the client to perform the 10^6 computations that this would take. To understand the performance needed and confirm whether this solution is realistic, the (currently somewhat vague) description should include concrete numbers regarding commercial hardware.

Time-Cost evaluation

According to https://www.miningrigrentals.com/rigs/argon2dchukwa for argon2, renting 1MH/day costs roughly 0.0015 BTC (i.e., $45 when 1BTC = $30,000) and a machine typically gives 3.6MH/s, so:

- To break the standard wallet (i.e., compute 10^6 hashes) costs 45$ and can be done in a day (using a single machine) or less (by renting multiple machines)
- To break the 10^{12} proposal costs $6,165,000 and can be done in 77 hours or less
- It is not possible (under the current available for-rental equipment) to break the 10^{18} proposal

Solution 3: Scrambled Memory Layout

Overview

The core idea of this solution is to store each leaf of the Merkle tree in specific locations of the client's memory, which are easy to find only if one has access to the OTP.

Analysis

As the document acknowledges, "this technique is the least straightforward". Indeed, it is also the least well-defined, so it is unclear what assumptions it has and what guarantees it provides.

The first glaring omission is the adversarial assumptions. Particularly, it is unclear what kind of access the attacker is assumed to have. For instance, the proposal does not offer any extra security, if:

- the attacker can access the input devices (thus obtain the OTP) and the memory
- the attacker can control the client's software; in that case, the attacker can simply obtain the OTP at the same time as the software

Without a clear description of the adversarial assumptions, it is not possible to evaluate the given estimations.

The second omission is regarding the implementation requirements. Specifically, it is unclear whether this proposal can be implemented on commercial hardware or specialized modules and how computationally-intensive it is.

Finally, the document acknowledges that solution 2 cannot be combined with this one (*“we cannot use the techniques in Section II by replacing the hash function to a complex one, since the hash function must be supported on blockchain as well and must be economical and fast enough to compute on blockchain”*), so the final sentence (*“this technique can be composed with the techniques in Part I or II or both”*) is incorrect.

Conclusion

There are three core issues with the proposals of this document:

1. A lack of assumptions regarding the adversary’s control of the client and/or computational (or other) power. The document assumes only a few ad-hoc adversaries; instead, an assumption-based analysis would ensure that the security guarantees are as described and/or needed.
2. A lack of computational requirements from the part of the client. All 3 proposals require the client to either perform multiple (10^6) or complex computations (specific memory retrieval). It is unclear how much time and computational effort these solutions require and whether they can be run on low-performance and/or commercial hardware.
3. A specific protocol description for the case of requiring 2 OTPs from the user. This solution increases the search space for the adversary the most ($\sim 10^{18}$) but, given the limitations of Google Authenticator, it is possible that new security implications come with this solution.

From our understanding at this stage, it appears that this proposal offers some interesting insights towards making ONE wallet secure even with a compromised client. However, a more detailed description should be provided defining the adversarial assumptions and implementation specifics wherever needed. Further analysis should be performed before adopting one or more of these solutions for the next version of the wallet.