A Security Analysis of Harmony's 1wallet

SHASHANK AGRAWAL

1wallet is also known as one-wallet. The analysis here is based on the following:

- 1wallet wiki (draft v0.1.5) [1]
- 1wallet client security (last edited June 29) [2]
- 1wallet code v0.15.1 [3]
- Issues 5, 59, 63 [4–6]

Notation. Let us set up some notation first. We will use C to denote a client, S to denote a smart contract, and A to denote an air-gapped authenticator (which can be set up on a mobile phone for instance). OTP stands for one-time passwords.

Note that we make a distinction between users and clients. We will use the former to mean real human users and the latter to mean a machine from where a user issues transactions (this machine may also store some public or private user data). We cannot expect any kind of security for a user who herself/himself is compromised, but we could certainly expect some level of security for users whose clients are compromised.

Cryptographic tools. Hash functions are one of the most important building blocks of cryptocurrencies. A good hash function behaves like a random function and has important properties like one-wayness, collision-resistance, avalanche effect, etc. [7]. Some prominent examples are SHA-2, SHA-3, BLAKE2, BLAKE3, etc.

A Merkle tree is a way to hash data in such a way that short proofs can be provided for the data in the tree. The tree-like structure of Merkle trees relies on a hash function like SHA-2 to "populate" the nodes of the tree. See [8] for a good overview.

OTPs are widely used these days as the primary or secondary means of user authentication. As the name suggests, OTPs are for one-time use only. They are typically valid for just a minute or so. HMAC-based OTP (HOTP) [9] and time-based OTP (TOTP) [10] are the two most popular types of OTPs.

We do not discuss digital signatures here because 1wallet's design doesn't rely on it.

1 DESIGN FEATURES

A detailed description of 1wallet's design can be found in the wiki [1]. We discuss some unique aspects of the design here.

No hardware wallet. 1wallet's design does not assume that people carry a hardware wallet, which seems reasonable since most people don't have it anyway.

Smart contract based. 1wallet is designed to be managed by a smart contract. The rules regarding spending, authentication, recovery, etc. are all coded into the contract. Most people would start with some standard smart contract wallet template and modify it based on their individual needs.

Google Authenticator. The primary method of authentication in 1wallet is a 6-digit time-based OTP generated through Google Authenticator. The secret seed for OTP is generated on the client. The client uses the seed to generate a large number of OTPs. These OTPs are

hashed into a Merkle tree form. The client discards the seed and the OTPs immediately afterwards (it only keeps the Merkle tree around). The root of the Merkle tree is stored in the smart contract. (For more details on this, please see the wiki [1] and/or the code [3]).

Commit and Reveal. Authentication in 1wallet is a two-step process. Suppose a user wants to do a certain operation on the wallet. First, she will commit an OTP (or two), details of the operation, and some other data (with the help of a hash function). Later, she will *open* the commitment to reveal everything inside of it. (The exact details of what is committed, how it is committed, what is revealed, etc. can be found in the code [3]. We discuss some of the details in Section 4.) The operation is executed at the time of reveal. Note that the commitment is a cryptographic commitment: while it binds the user to the value committed, it hides the value at the same time.

Authentication is carried out in two-steps to prevent *front-running* attacks (see below). If authentication was just a single step process, then an attacker could generate a valid transaction based on the victim's transaction: copy the authenticating information (like the OTP) and modify some parameters as desired. The attacker could then *front-run* the victim and cause a significant damage. For example, if a correct OTP allows a user to transfer up to X amount of crypto-currency, then the attacker could transfer X to any account of its choosing.

Observe that even if the user's transaction contains a one-way function of the OTP (like a hash), the attack still applies. The main point here is that if the contract has a way to validate the user's transaction, it will also validate the attacker's. Also observe that even though the user can digitally sign the contents of a transaction (with an existentially unforgeable scheme like ECDSA), the smart contract wallet is not associated with any public keys to verify the signature. In other words, the contract is not tied to any specific account/address so it has to treat transactions from all types of accounts in the same way.

1wallet's two-step authentication approach cryptographically ties the authenticating information with the actual intent of the user (say, transferring some crypto to another account) in the first transaction, without revealing anything in the process. An attacker could still steal information from the transaction but it won't be able to modify it meaningfully. Certain attacks are still possible if one is not careful; we will see how 1wallet prevents them.

2 RISKS

Public nature of smart contracts. By design, smart contracts are public in nature. The code of the contract, its internal state (which changes over time), and all the transactions sent to it are fully visible to everyone. This puts some constraints on the design of smart contract based wallets. For instance, one cannot store any data that is supposed to be secret on a smart contract.

Front-running attacks. Since anyone can observe the transactions being routed through a blockchain network, an attacker can *front-run* a transaction on the network. Specifically, when the attacker spots a transaction tx on the network, it can copy some parts of it into a new transaction $\mathsf{tx}^\star$, and try to get $\mathsf{tx}^\star$ accepted into the blockchain before tx is considered (say by paying higher fee). Sometimes the consequences of a front-running attack can

be devastating: the attacker may even be able to drain all the funds out of the contract account.

Network delays. Depending on the blockchain design and the state of the network, the state of the blockchain may change significantly between the time a transaction is created and the time it is confirmed. In particular, a user may generate a transaction for a smart contract based on its current view of the contract's state (which may not even be the latest state) but by the time the transaction is processed, the state may have changed dramatically (even when there is no targeted interference).

Single points of failure. A important security benefit of blockchain systems is their decentralized design, i.e. there are no single points of failure. Ideally, we would like wallets to have the same property too. However, a wallet has several different components and avoiding single points of failure is easier said than done.

3 THREAT MODEL & SECURITY GOALS

While achieving the highest level of security is important, it's also important to keep the system usable. In other words, the wallet design must strike a good balance between security and usability. The wiki describes several security and usability goals for 1wallet at a high level. We will focus on the ones for which we can present a formal argument.

We will use the notation Adv to denote an adversary/attacker. For every situation discussed below, we assume that Adv:

(1) can read/observe all messages sent by users on the blockchain network (can generate new messages of its own based on this);
(2) has full information about all the blocks ever created;
(3) can sync with the latest state of the blockchain (even when the victim is a few blocks behind); and,
(4) can delay transactions to some extent (eventually, all transactions will be processed though).

Passive & active attackers. In cryptography literature, passive vs active is a very common classification for attack type. Both passive and active attackers can compromise one or nodes on the network but passive attackers don't modify the behavior of compromised nodes while active attackers do. Usually, communication channels between nodes are assumed to be private and secure (if they are not, a PKI set-up could be used to make them). Here, we will use the same terms passive and active but in a different sense, primarily because the blockchain setting is a very *open* one (nodes are not identified, they communicate over open channels, etc.).

A *passive* attacker is one who doesn't have access to and doesn't do anything beyond the things listed above. Specifically, Adv has access to public data (blocks on the chain, mempool, etc.) and controls the network to some extent (e.g., delays transactions). It doesn't, in particular, has the ability or opportunity to compromise nodes on the network.

With the knowledge available and some control on the network, a passive adversary can try to impersonate a user, attack her wallet (drain funds out, block operations, etc.), etc.

An *active* attacker has all the abilities of a passive attacker and, in addition, is able to compromise one or more clients, authenticators, etc.

Client compromise. Recall that client stores a hash tree of a large number of OTPs. The OTPs themselves and the OTP seed are discarded right after the tree has been generated from them (and before the wallet set up). If a client is compromised, then the whole hash tree could be revealed to Adv. In the least, this would enable Adv to brute-force the OTPs. We require that this is essentially the best any attacker should be able to do. In other words, no attacker should be able to launch an attack better than the obvious brute-force one. Furthermore, we require that the task of brute-forcing ℓ OTPs should be no easy than ℓ times the amount of work required to brute-force one (for small values of ℓ).

Authenticator compromise. Typically, the authenticator would be on a mobile device like user phone. If the mobile device is compromised, then Adv may be able to extract the OTP seed, and generate any number of OTPs on its own. This attack is certainly quite severe and there is little one can expect to salvage when it happens.

One vs many. We will also consider attackers who target more than one user by compromising their clients or authenticators. We will expect that no matter what strategy an attacker takes, the strategy doesn't scale sub-linearly. In other words, a strategy that needs X amount of resources to attack one user should need $O(nX)$ resources for n users.

4 AUTHENTICATION SCHEME

In this section, we will go into some of the details of the authentication mechanism of 1wallet. The two most important files for us in the code-base are `code/lib/onewallet.js` (client-side code) and `code/contracts/ONEwallet.sol` (smart contract code). The code-base uses standard hash functions like SHA and Argon2 for hashing purposes.

4.1 Client-side Code

Many different types of hashed values are computed in the code. We will focus on the following here: `hseed`, `eotp`, Merkle hash, `commitHash`, `paramsHash`, and `verificationHash`.

hseed. As of now, 1wallet supports OTPs in three different ways: legacy mode, single OTP mode, and double OTP mode. To compute the Merkle tree, one can provide a single seed (`otpSeed`) or two different seeds (`otpSeed` and `otpSeed2`). Both legacy and single OTP modes rely on a single seed but the former doesn't use any *randomness* while the latter does (more on this later). Double OTP mode relies on both the seeds and some randomness. `hseed` is defined to be a hash of one or two seeds, depending on the mode.

EOTP. `eotp` is a hash of `hseed`, `nonce`, `otp`, `otp2`, and four bytes of randomness (depending on the mode). Irrespective of the mode, the same amount of data goes into computing `eotp` by adjusting the length of `hseed` used. We will not consider the nonce value here because it is defined to be (`i % maxOperationsPerInterval`) and the latter variable is being set to 1 right now (so all nonce values are just zero).[1] The EOTPs are hashed individually to obtain leaves of a tree; the leaves are then hashed to build a Merkle tree. All of this is accomplished inside the function `computeMerkleTree` in `onewallet.js`. After

[1]There seems to be some *over-counting* of `maxOperationsPerInterval` in the code: `n` is defined to be $2^{\texttt{height}-1}$ and `height` takes into account `maxOperationsPerInterval`, but then `maxOperationsPerInterval` shows up again in the number of iterations for the loop where `input` is defined. However, if `maxOperationsPerInterval` is just 1, then it doesn't matter.

computing the tree, OTP seeds (`otpSeed` and `otpSeed2`), any and all randomness, and EOTPs are immediately discarded. The client just holds on to the Merkle tree (all the leaves, the intermediate nodes, and the root), `hseed`, and a few other things.

commitHash, paramsHash, and verificationHash. `commitHash` of an EOTP is a hash of `eotp`, its neighbor in the tree, and its index in the tree[2], computed in the function `computeCommitHash`. `paramsHash` depends on the operation a user wants to carry out through the wallet. For instance, it's a hash of destination and amount for a transfer operation (see `computeTransferHash`). Lastly, `verificationHash` is a hash of `paramsHash` and `eotp` (see `computeVerificationHash`).

Computing EOTPs later. An important thing to note is that for single and double OTP modes, even the legitimate user has to do some work to compute EOTPs from OTPs after the Merkle tree has been generated because all the randomness is discarded. Specifically, she has to try all possible values for the random string to find the one that helps generate the stored leaf (see `recoverRandomness`). Of course an attacker would have to deal with a similar slowdown too.

Even though four bytes are reserved for the randomness, the total number of distinct random values is 2^{20} if the hash function SHA256 is used and slightly lower if Argon2 is used. One can make the range of values even bigger to make the attacker's life harder, but the legitimate user would suffer too. Therefore, the size of the range needs to be carefully chosen.

4.2 Smart Contract Code

Now we turn our attention to the smart contract code in `ONEwallet.sol`. Recall that authentication in 1wallet consists of two phases, commit and reveal. The function `commit` takes three parameters: `commitHash`, `paramsHash`, and `verificationHash` (here `commitHash` is just called `hash`). It stores `commitHash` and adds `paramsHash`, `verificationHash`, and block timestamp to a list indexed by `commitHash`. (See the file `CommitManager.sol`.) The function `reveal` is a lot more complex and is defined with the help of several other functions.

Starting version 0.15, there is a new structure called *core* to keep track of all the authenticators a wallet can be bound to. 1wallets created since 0.15 generate 7 OTP Merkle trees instead of just 1. The 6 new OTP Merkle trees are for recovery purposes. Each OTP Merkle tree corresponds to a core. See the release notes for more details [11].

Through the different cores, `reveal` handles several important operations like recovery, forwarding, displacing, etc., but we will restrict ourselves to the usual operations here. The function `reveal` calls `authenticate` which in turn calls `authenticateCores`. See the contract file `Reveal.sol`.

`authenticateCores` first calls `isCorrectProof` on the revealed `neighbors` (a path in the Merkle tree), `indexWithNonce` (ignore nonce like before), and `eotp` (all part of `AuthParams`) to check if the Merkle proof is correct. Secondly, it calls `getRevealHash`

[2]In `code/contracts/ONEwallet.sol`, `commitHash` is supposed to be committing to `indexWithNonce` instead of just index. Again, if `maxOperationsPerInterval` is just 1, when `index` and `nonce` are computed in `verifyReveal`, `index` comes out to be the same as `indexWithNonce` and `nonce` comes out to be just zero.

on the immediate neighbor, `indexWithNonce`, `eotp`, `operationType`, and operation related things (first three part of `AuthParams`, last two part of `OperationParams`) to obtain `commitHash` and `paramsHash`. Third, it calls `verifyReveal` with `commitHash`, `paramsHash`, `eotp`, and a few other things. `verifyReveal` goes through the list stored against `commitHash` to find a position where all the following conditions are satisfied: the stored verification hash matches with the computed verification hash, the stored params hash matches with the supplied params hash, this position is not marked complete, and not too much time has passed between commit and reveal. Lastly, it calls `completeReveal` to mark the position complete.

Now going back to the main contract file `ONEWallet.sol`, the operation requested by the user is executed through the function `_execute`.

5 ANALYSIS

We will make the following assumptions in the analysis: Client is not compromised in the initial wallet set-up phase when it uses the seed to generate a large number of OTPs. Recall that the seed is discarded right after all the OTPs have been generated, meaning that the client holds the seed for a very short time. Thus our assumption is meaningful.

5.1 Security against passive attacks

Suppose an attacker Adv targets a user U who has a smart contract based 1wallet. Adv does not have any visibility into U's client C or her authenticator A, but has access to or is capable of all the four points listed at the beginning of Section 3.

Adv can observe two types of transactions on the network, commit transactions and reveal transactions. A commit transaction basically consists of three hashed values which are `commitHash`, `paramsHash`, and `verificationHash`. Just because all these values are outputs of a hash function, we cannot say that they leak no information. For instance, if U wants to transfer some crypto, `paramsHash` will be a hash of destination and amount. One can study U's past transactions (and other publicly available information) to determine a small set of candidates for the destination and amount, and then check against the `paramsHash` to find out which one is right. This simple strategy would work with a good probability in most cases. However, 1wallet doesn't strive to hide users' intent. The security goal is that no one should be able to carry out operations that U does *not* intend.

For simplicity, assume that Adv knows the pre-image for all the hashes in all the commit transactions, since most commits would be revealed anyway. A reveal transaction reveals a bit more information though: for the EOTP, it also reveals all the nodes in the path leading up to the root, which acts as a *proof*. Given all of this information from several commit and reveal transactions, can Adv execute an operation that U doesn't intend to? First observe that to carry out an operation, a valid EOTP is needed. The smart contract checks the validity of an EOTP through the function `isCorrectProof` by hashing it to obtain a leaf, using the path provided to derive a root, and then comparing it with the stored root. Due to the collision-resistance property of hash functions, only one of the EOTPs generated in the wallet set up phase would be accepted by `isCorrectProof`.

How does the attacker get hold of one of these EOTPs? There are two options: somehow derive an EOTP on its own or capture an EOTP revealed by U. First let's check the feasibility of the first option by modeling the hash function used to derive EOTPs as a random oracle.

EOTPs are of length 256 bits so simple guessing is clearly infeasible. Further, EOTPs are computed as a hash of `hseed`, `nonce`, `otp`, `otp2`, and some randomness (depending on the mode). Between the different modes, `hseed` is at least 18 bytes (144 bits) long and it's derived as a hash of one or two seeds, which are themselves random 256-bit long numbers. As a result, guessing `hseed` is infeasible too (the same `hseed` goes into all the EOTPs). To sum up, it's infeasible for any attacker to find a valid EOTP on its own.

However, Adv could still use the EOTPs revealed by U to carry out its own operations or disrupt U's operations. Let's look into this more closely with the help of a bit of history.

5.1.1 Issue #59 in major version 6. Up until version 6, there were two hashes involved in the commit-reveal mechanism, `commitHash` and `paramsHash` (instead of the three now). `commitHash` committed to the proof related values and `paramsHash` committed to the operation related values. The contract stored only one `paramsHash` against a `commitHash`, the idea being that if an attacker tried to *reuse* the same `commitHash` (which is much harder to compute) to commit to a different operation (thus a different `paramsHash`), its commit would simply be rejected by the contract.

However, it was discovered that such an implementation would still allow attackers to deny service to legitimate users by front-running them, because if an attacker's `paramsHash` is recorded first (no matter what it is), then the user's commitment would be declined [5]. To fix this, three changes were proposed: First, the contract would record multiple entries against the same `commitHash`. Second, an extra hash value called the `verificationHash` will be used, which will be a hash of `paramsHash` and `eotp`. Third, the contract would only execute the first entry that has a valid `verficationHash` (if it hasn't already been executed; see `verifyReveal` in `Reveal.sol`).

With these changes, an attacker has two options. It can try to front-run the user's commit with a different `paramsHash` but, to generate the corresponding valid `verficationHash`, it needs to guess the hash itself or the EOTP, both of which are infeasible. The second option is to wait for the user to reveal her EOTP and use it to derive the valid `verficationHash`. However, in this case attacker's commit will be recorded *after* the user's under the same `commitHash`, so the attacker's operation will never be executed.

It's important to note here that the first valid `verficationHash` recorded against `commitHash` is the only one that can be executed, even though other valid `verficationHash`(es) may be committed later.

5.2 Security against active attack

Any kind of active attack is very severe and could provide a significant advantage to the adversary.

Client compromise. If a user's client is compromised, then the entire Merkle tree is revealed to the attacker (along with `hseed` and a few other things). On the flip side, OTP seeds, randomness (if any), and EOTPs were all discarded in the very beginning, so they are not exposed.

Though the attacker now has neighbors (actually, the whole paths) for all the EOTPs, to gain any real advantage from this attack, it still has to guess one or more EOTPs. Recall that EOTPs are computed as a hash of `hseed`, `nonce`, `otp`, `otp2`, and some randomness (depending on the mode), out of which the latter three are still not available.

Here the legacy mode would provide the lowest level of security because the attacker needs to guess just one six-digit OTP. Single and double OTP modes would provide more security. In particular, cracking EOTPs for the double OTP mode would be quite difficult due to the use of two six-digit OTPs and four bytes of randomness. In the worst case, this could translate to brute-forcing $10^{12} \times 2^{32}$ values (actually, lower than this because the range of random values is smaller than 2^{32}; see the discussion under 'Computing EOTPs later').

Authenticator compromise. Authenticator compromise is quite disastrous because it gives the ability to generate any number of OTPs. Once OTPs are available, EOTPs can be generated without much trouble, similar to how a client would do it. With non-legacy modes, there is some randomness involved, but the attacker can go through all possible values just as a client would.

5.3 One vs many

Here the argument is simple: attacks do not scale because OTP seeds are generated randomly every time a wallet is set up.

6 CONCLUSION & FUTURE WORK

In this write-up, we analyzed the security of the basic operations of 1wallet against various types of attacks. We could see from the analysis that 1wallet strikes a good balance between security and usability.

There are several other important aspects of 1wallet that we did not analyze here like recovery (which involves the use of six OTP Merkle trees), forwarding, doubling spending limit, etc. We leave them as interesting targets for future work.

REFERENCES

[1] Aaron Li. 1wallet Wiki. https://github.com/polymorpher/one-wallet/wiki.
[2] Aaron Li. Client Security. https://github.com/polymorpher/one-wallet/wiki/Client-Security.
[3] 1wallet GitHub. https://github.com/polymorpher/one-wallet/tree/v0.15.1.
[4] TODO before launching to a larger set of users (10k-100k people). https://github.com/polymorpher/one-wallet/issues/5.
[5] DoS vulnerability in secure commit-reveal mechanism (major version 6). https://github.com/polymorpher/one-wallet/issues/59.
[6] Client Security - Revision of Scrambled Memory Layout method. https://github.com/polymorpher/one-wallet/issues/63.
[7] Cryptographic hash function. https://en.wikipedia.org/wiki/Cryptographic_hash_function.
[8] What is a Merkle Tree? https://decentralizedthoughts.github.io/2020-12-22-what-is-a-merkle-tree/.
[9] Mountain View, David M'Raihi, Frank Hoornaert, David Naccache, Mihir Bellare, and Ohad Ranen. HOTP: An HMAC-Based One-Time Password Algorithm. RFC 4226, December 2005.
[10] Mountain View, Johan Rydell, Mingliang Pei, and Salah Machani. TOTP: Time-Based One-Time Password Algorithm. RFC 6238, May 2011.
[11] 1Wallet v15 release notes. https://github.com/polymorpher/one-wallet/releases/tag/v0.15.1.