CashShuffle Security Audit

Final Report, 2019-03-25

FOR PUBLIC RELEASE

VISIONATI

Contents

1 Summary

CashShuffle is a plugin for the ElectronCash Bitcoin wallet software. CashShuffle implements a superset of the CoinShuffle protocol, whose aim is to anonymize cryptocurrency ownership by pooling a number of users together and performing a randomized shuffle of their transactions to new addresses.

Visionati hired Kudelski Security to perform a security assessment of the CoinShuffle component of the ElectronCash wallet. We focused on the cryptographic functionalities of the code and implementation of security good practices.

The repository concerned is: https://github.com/clifordsymack/Electron-Cash, we mainly focused on the code part contained at
https://github.com/clifordsymack/Electron-Cash/tree/master/plugins/shuffle

More specifically, we audited commit `71c0d3b`.

This document reports the security issues identified and our mitigation recommendations, as well as some observations regarding the code base. A "Status" section reports the feedback from Visionati's developers, and includes a reference to the patches related to the reported issues.

We report:

- 2 security issues of medium severity
- 1 security issue of low severity
- 4 observations related to general code safety

The audit was performed jointly by Dr. Tommaso Gagliardoni, Cryptography Expert, and Yolan Romailler, Senior Cryptography Engineer, with support of Dr. Jean-Philippe Aumasson, VP of Technology, and involved 7 person-days of work.

2 Methodology

In this code audit, we performed four main tasks:

1. informal security analysis of the original protocol;
2. actual code review with code safety issues in mind;
3. assessment of the cryptographic primitives used;
4. compliance of the code with the CoinShuffle paper.

This was done in a static way and no dynamic analysis has been performed on the codebase. We discuss more in detail our methodology in the following sections.

2.1 Protocol Security

We analyzed the CashShuffle protocol in view of the claimed goals and use cases, and we inspected the original CoinShuffle protocol description, looking for possible attack scenarios. We focused on the following aspects:

- possible threat scenarios;
- necessary trust assumptions between involved parties;
- necessary trust assumptions between parties and server;
- resistance to deanonimization attacks;
- resilience to double-spending attacks;
- resilience to funds stealing;
- resistance to DoS attacks;
- interaction between the protocol and the network layer;

- interaction between the protocol and the blockchain;
- blame phase and cheater unmasking;
- edge cases and resistance to protocol misuse.

2.2 Code Safety

We analyzed the provided code, in particular the codebase of the shuffle plugin. We checked the Python code for things such as:

- general code safety and susceptibility to known vulnerabilities;
- bad coding practices and unsafe behavior;
- leakage of secrets of other sensitive data through memory mismanagement;
- susceptibility to misuse and system errors;
- error management and logging;
- safety against malformed or malicious input from other network participants.

2.3 Cryptography

We analyzed the cryptographic primitives and subprotocols used in CashShuffle, with particular emphasis on randomness and hash generation, signatures, key management, and encryption. We checked in particular:

- matching of the proper cryptographic primitives to the desired cryptographic functionality needed;
- security level of cryptographic primitives and of their respective parameters (key lengths, etc.);
- safety of the randomness generation in the general case and in case of failure;
- safety of key management;
- assessment of proper security definitions and compliance to the use cases;
- checking for known vulnerabilities in the primitives used.

2.4 Protocol Specification Matching

We analyzed the original CoinShuffle paper, and checked that the CashShuffle plugin matches the specification. We checked for things such as:

- proper implementation of the protocol phases;
- proper error handling;
- correct implementation of the blame phase;
- correct interaction with the blockchain network;
- adherence to the protocol logical description.

3 Findings

This section reports security issues found during the audit.

The "Status" section includes feedback from the developers received after delivering our draft report.

KS-CSSH-F-001: Modulo bias in `generate_random_sk()`

Severity: Medium

Description

In `client.py`, in `generate_random_sk()`, a random value is drawn between 0 and $2^{256} - 1$ and then reduced modulo the curve order:

```
230 def generate_random_sk():
231      G = generator_secp256k1
232      _r  = G.order()
233      pvk = ecdsa.util.randrange( pow(2,256) )
234      eck = EC_KEY(number_to_string(pvk,_r)) %_r
235     return eck
```

This introduces a so-called modulo bias.

Recommendation

This behavior should be avoided, by either using rejection sampling (i.e. generating fresh random numbers of necessary bitlength until one small enough is found falling within the desired range, which is the default behaviour of Python's ECDSA `randrange` function), or by using directly the right bounds in a secure random function.

We usually recommend to use Python 3.6's Secrets module, which provides directly the

function you need such as `randbelow` or, if using older Python versions `os.urandom()` along with rejection sampling.

This could also be fixed by using directly the order in ECDSA's `randrange` function.

Status

This was fixed in commit `7e5e5c7`.

KS-CSSH-F-002: Modulo bias in `generate_key_pair()`

Severity: Medium

Description

In `crypto.py`, in `generate_random_sk()`, a random value is drawn between 0 and $2^{256} - 1$ and then reduced modulo the curve order:

```
15 def generate_key_pair(self):
16         "generate encryption/decryption pair"
17         self.private_key = ecdsa.util.randrange(pow(2, 256))
18         self.eck = EC_KEY(number_to_string(self.private_key, self._r)) %_r
19         self.public_key = point_to_ser(self.private_key*self.G, True)
```

This introduces a so-called modulo bias.

Recommendation

This behavior should be avoided, as described in KS-CSSH-F-001.

Status

This was fixed in commit `480710b`.

KS-CSSH-F-003: Secret data not zeroized after use

Severity: Low

Description

Variables containing sensitive data (e.g., secret keys, curve points used in intermediate computation, etc) are not overwritten after use. This might potentially leave sensitive data in some areas of memory.

As an example, in `client.py`, the `generate_random_sk()` function is implemented as follows:

```
230 def generate_random_sk():
231      G = generator_secp256k1
232      _r  = G.order()
233      pvk = ecdsa.util.randrange( pow(2,256) )
234      eck = EC_KEY(number_to_string(pvk,_r)) %_r
235     return eck
```

This leaves unzeroized the value `pvk`, which might allow to recover the secret key from a malicious memory access.

Recommendation

Always overwrite with zeroes any variable containing potentially sensitive data after use. Although memory zeroization is not reliable in languages like Python, and the presence of leftover copies of data in memory can never be ruled out, it is still desirable to perform the zeroization step at least on a logical level, in order to reduce the possible attack surface of memory leaks.

Status

The CashShuffle team acknowledges the issues but considers it to be an intrinsic limitation of the Python language. Specifically: it is impossible to write to immutable types in Python, and even if you could, Python provides no guarantees of correctly overwriting every copy of the variables in memory. In any case, the practical exploitability risk of such limitation is to be considered extremely remote.

4 Observations

This section reports various observations that are not security issues to be fixed, such as improvement or defense-in-depth suggestions.

KS-CSSH-O-001: Typo in filename `test_announecment.py`

The filename of `plugins/shuffle/tests/test_announecment.py` contains a typo.

Status

Fixed in commit `7b3b08f2`.

KS-CSSH-O-002: Modulo biases in tests

The file `test.py` contains two modulo biases: the first one at line 84:

```
81      def __init__(self):
82          G = generator_secp256k1
83          _r  = G.order()
84          pvk = ecdsa.util.randrange( pow(2,256) ) %_r
85          eck = EC_KEY.__init__(self, number_to_string(pvk,_r))
```

and the second one at line 106:

```
105 def generate_fake_key_pair(self):
106         self.fake_private_key = ecdsa.util.randrange( pow(2,256) ) %self._r
107         self.fake_eck = EC_KEY(number_to_string(self.fake_private_key, self._r))
108         self.fake_public_key = point_to_ser(self.fake_private_key*self.G,True)
```

Status

Fixed in commit `7b3b08f2`.

KS-CSSH-O-003: Using `assert` in production is not recommended

Throughout the codebase, `assert` statements are used to verify certain conditions, such as the fact that the amount is bigger than 0 and other such checks. However, according to the Python documentation, "assert statements are a convenient way to insert debugging assertions into a program", and they are not run when optimization is requested.

This is not directly an issue, but it might be possible for example that a group of people wrongly request optimization by using a command line provided by a third party, thus disabling many required safety checks. We did not characterize the actual impact of running the codebase in optimized mode.

Status

The CashShuffle team acknowledges the issue, but this behavior cannot be changed, as part of the code was inherited directly from the ElectronCash wallet - which in some cases specifically relies on the assert behavior.

KS-CSSH-O-004: Choice of SHA-224 as a general purpose hash

While the hash extension attack is not a threat per se for the protocol, using SHA-224 as a common usage hash function is an uncommon choice that is not documented and seems arbitrary. We would recommend switching to a more recent hash function that has no known attack vector, or to a construction that prevents the hash extension problems, such as SHA3.

This would be part of the so-called defense-in-depth approach that we encourage: while there are currently no parts of the codebase that might suffer from a hash length extension attack, other implementations or future changes might inadvertently introduce vulnerabilities because of the use of the same hash function.

Status

This was changed in commit `c618bc9` and SHA-224 was replaced by a double round of SHA-256, *à la* bitcoin, which is not vulnerable to known attacks such as the extension length one.

5 Assessment

The following is a general assessment of the way the CashShuffle codebase is implementing a superset of the CoinShuffle protocol, whose aim is to anonymize cryptocurrency ownership by pooling a number of users together and performing a randomized shuffle of their transactions to new addresses.

It is important to notice that the CoinShuffle paper does not consider the problem of bootstrapping the CoinShuffle protocol, most notably, it does not consider how the users that wish to participate in the mixing are finding each other.

In the CashShuffle setup, the bootstrapping is handled using a mixing server that will help participants create a pool to do the mixing, but that also handles the banned players by preventing them to enter a pool. The pool size N is determined by the server, however the codebase is currently checking in `client.py` in the function `gather_the_keys()` that the number of participants in a shuffle is at least 3.

5.1 Caveats

In order for participants to take part to a shuffle, they need to join a server that would handle the bootstrapping process. This notably means that some servers might decide to ask a fee to let a participant join a pool, just like a mixnet server. And in the same way, a CashShuffle server can be free, but any mixnet server could decide to do the mixing for free, if so wished. The free nature of CashShuffle servers is therefore not an intrinsic property offered by the protocol.

Furthermore, the server has to be trusted: a malicious server might match a given client in a "fake pool" with freshly created fake identities in order to be able to later deanonymize that client. This means that the need to trust a central authority remains. However, thanks to the CoinShuffle protocol, it is true that the server is not able to steal funds, which is an improvement over mixnet servers.

It should be noted that we did not review the server code.

An intrinsic limitation not just of the CoinShuffle protocol, but of mixing networks in general, is that players cannot anonymize transactions with arbitrary amount of funds. The reason is that blockchain protocols such as Bitcoin do not hide the amount of coin transferred, so users can be tracked by just looking at the amount of the transaction. CashShuffle avoids this by setting fixed amounts of coins for each player, and providing separate pools according to the transaction threshold chosen by the user.

Finally, notice that this code audit and specification assessment was done on commit `71c0d3b` of the CashShuffle codebase, and that this codebase has been rapidly evolving, even during the audit.

5.2 Deviations from the protocol

We have noticed very little deviations from the CoinShuffle protocol in the codebase we reviewed.

We noticed the following differences in `coin_shuffle.py`:

- The original protocol is assuming the phase to be passed around in the messages as a number, whereas strings are used in the codebase. This might be for instance replaced with an `Enum` instead, as the codebase is using Python>3.4. But this is not required and strings are fine for that purpose as well.
- On lines 248–249 the `computed_hash` required by phase 4 is computed as being $h_i = \mathrm{H}(T_{\text{out}}, (ek_1, \cdots, ek_N))$ instead of $h_i = \mathrm{H}((ek_2, \cdots, ek_N), T_{\text{out}})$. There are two slight deviations from the CoinShuffle paper here:
  - wrong order, first should come the output addresses, then the keys ;
  - the first player's encryption key is also encoded in the hash, even though it is never used and is skipped completely in the CoinShuffle paper.

  Both of these modifications are not lowering the security of the protocol and neither are a concern. The second one is actually probably lowering the code complexity and might be a direct consequence of the fact that the first player is not known in advance at the time of the key generation, as this is decided by the lexicographic ordering of the players in the player's `dict`.
- In phase 5, a participant that has already spent her coin should be blamed, according to the paper, but this is currently not enforced in the code. Notice

that this would not be a problem as the shuffle would simply fail, but the guilty player would not be blamed, which might be something desirable.

Overall these deviations are not aberrant and some are to be expected in any practical instantiation. We recommend possibly fixing the latest one, but do not consider the other deviations to lower the security level of the implementation.

5.3 Specific Remarks

At page 11, section "Public-Key Encryption", the CoinShuffle paper claims that the following properties are necessary for the public-key encryption used:

1. *IND-CCA*: it is not explained why this is necessary instead of, e.g., a more standard (but weaker) IND-CPA property. The paper does not mention what we believe is the main reason, i.e., that IND-CCA guarantees *non-malleability of ciphertexts* unlike IND-CPA. This is usually a desirable property for protocols where forging new ciphertexts from valid known ones has to be avoided, although it is unclear whether this is imperative in CashShuffle.
2. *Length-regularity*: this is usually not a standard requirements for many applications of public-key encryption, and the need for it is not documented. We believe that it has to do with the fact that in CashShuffle, the public-key encryption is used in a multi-layer fashion to encrypt fresh addresses that must not be traceable.

The correctness of the randomness generation for the shuffling is not implemented in a verifiable way. However, this is not a concern, because *every* player takes part in the shuffle with locally generated randomness. This in particular means that as long as two of the players perform the shuffle honestly, all the transactions are guaranteed to be untraceable.

Moreover, we notice that the position of the players in the queue is decided by the lexicographic ordering of the verification keys given in the initial request, as per the CoinShuffle paper example. This might allow players to position themselves, e.g., as first or last player in the pool with high probability, by brute-forcing the generation of their addresses to produce low- or high-value hashes. Although we do not believe that a preferential position of a player in the pool can lead to meaningful attacks, it could

have been preferable to generate the player ordering in a different way, possibly by including in the hash determining their position *all* the involved verification keys, not just on a player-by-player basis.

The fact that under certain conditions a player can cause a shuffle to fail and not be blamed for it has to be taken into account, as it might be possible to perform DoS attacks against the shuffle process. However, there are probably many ways for a shuffle to fail and for the guilty player to go unblamed. For instance, timeouts at the network level and communication issues are not directly handled using the method from Dissent as outlined in the CoinShuffle paper, and a participant that would repeatedly disturb shuffles by going offline is not necessarily "punished" for it. This could lead to some sort of DoS, if many participants are actively trying to disrupt the protocol, but can also easily be mitigated at a later stage by implementing more blame conditions.

5.4 Conclusion

Notice that the specification of both CoinShuffle and CashShuffle is not formal and lots of the implementation details are not being explicitly covered by any specification.

Overall, we believe that the analysis from the CoinShuffle paper is correct, but that in practice such a protocol has some limitations. For example, shuffles can easily be detected and are a known red flag for AML analysis tools. Also, the paper does not extensively cover the problem of malicious behavior, nor does it discuss the assumptions on the network (guaranteed delivery, delays, etc.).

However, we believe that the CoinShuffle protocol and the CashShuffle implementation provide a practical solution to the problem of mixing transactions without the risk of funds being stolen in the process.

We further believe that the CashShuffle codebase provided by Visionati, that we reviewed, is implementing the protocol as correctly as they can and we did not find any evidence of malicious intent, flawed logic or potential backdoor in the codebase.

6 About

Kudelski Security is an innovative, independent Swiss provider of tailored cyber and media security solutions to enterprises and public sector institutions. Our team of security experts delivers end-to-end consulting, technology, managed services, and threat intelligence to help organizations build and run successful security programs. Our global reach and cyber solutions focus is reinforced by key international partnerships.

Kudelski Security is a division of Kudelski Group. For more information, please visit https://www.kudelskisecurity.com.

Kudelski Security
Route de Genève, 22-24
1033 Cheseaux-sur-Lausanne
Switzerland