Switcheo Ethereum Contracts Security Audit

September 28th, 2019

@lucash-dev

https://hackerone.com/lucash-dev

https://github.com/lucash-dev

Disclaimer

Security audits are inherently limited -- there is no possibility of strict proofs that a given system doesn't contain further vulnerabilities. The best that can be offered is a good faith effort to find as many vulnerabilities as possible within the resource constraints of the project.

Given the above, this audit report is provided "AS IS", without any guarantee of correctness or completeness. The author takes no responsibility for any loss or damage caused by the use, misuse or failure to use the information contained in this document, as well as for any information that might be missing from it (e.g. missed vulnerabilities).

By using the information contained in this report, you agree to do so at your own risk, and not to hold the authors liable for any consequences of such use.

Summary

This document contains the results of the audit conducted on the Switcheo Ethereum smart contracts (V2). The scope and methodology of the audit are briefly described, then all issues found during the process are listed, categorized as follows:

- **Exploitable Vulnerabilities**: This section contains vulnerabilities found in the contracts for which attack vectors were identified during the audit.

  *Deployment of the contracts without fixes for these issues **will** expose users to the described attacks.*

- **Potential Vulnerabilities**: This sections contains issues found in the smart contracts for which no concrete attack vector was found during the audit. These can be deviations from best security practices or other design flaws that limit the reviewer's ability to identify security issues.

*Deploying the contracts without fixing these issues **can** increase the probability of exposing users to unknown attacks.*

- **Design Issues**: This section contains issues regarding code readability, maintainability, and other design issues. While not directly related to security, they render scanning the code for vulnerabilities harder and code reviews less reliable.

  *Fixing these issues before deployment **may** lead to uncovering existing vulnerabilities or preventing the introduction of new ones, thus allowing for increased security in the final product.*

The author strongly recommends that at least the issues in the first category be all fixed before deployment.

Scope

The scope of this audit are the contracts `BrokerV2.sol` and `Utils.sol` as found at `https://github.com/Switcheo/switcheo-eth` at the commit `c562da947986c2800833a0aad48854f79729b38d`.

Methodology

The audit was conducted manually by an experienced security researcher, by inspecting the code in two different ways:

- Systematic review of every function in the contracts, spotting missing best practices and logical flaws.
- Free exploration of interaction points and execution flows, searching for concrete attack vectors and invalid assumptions.

While the first approach mimics a more traditional code review, the second one tries to reproduce the kind of coverage you would obtain from a "bug bounty" program.

It's the author's belief that combining these approaches offer much higher likelihood of catching high-severity issues than using either individually.

Issues Found

Exploitable Vulnerabilities

1. Unrestricted pre-image length in Atomic Swaps

Contract BrokerV2 introduced the atomic swap feature, which allows for trustless trading of assets between different ledgers/blockchains. The mechanism used is having

balances/outputs in both chains that are unlocked by revealing the same pre-image to a hash.

The flow of the operation can be simplified as:

1. Users A and B are matched for a trade.
2. User A creates a secret pre-image.
3. User A signs and sends a request to create a swap to the Broker operator.
4. Broker operator uses method `createSwap` to initiate the atomic swap (in the blockchain C1)
5. User B identifies the hash of the pre-image, and uses it to perform on the blockchain C2 operations analogous to (3) and (4). Creating a balance in C2 that can be transfered to A by revealing the pre-image.
6. User A transfers the balance from (5) by revealing the secret pre-image, and committing a transaction in chain C2.
7. User B obtains the balance from (2) by using the revealed pre-image, committing a transaction to the blockchain C1.

If the transactions in (7) can't be committed by any reason, then user B loses their part of the deal, and the swap becomes essentially a theft.

Unfortunately, that scenario can be obtained by user A -- if the the maximum valid transaction size in blockchain C2 is bigger than the maximum transaction size in C1.

This is possible because the method `executeSwap` allows for an arbitrary-length array of bytes as the pre-image, which allows user A to create a pre-image long enough that it can't be committed as part of a transaction in C1, but can still be used to obtain the funds in C2.

In particular, if C1 is a chain other than Ethereum, then user A can use a very long pre-image that can be used to unlock the funds in the Ethereum Broker, but might not be possible to use in a transaction in C1.

Conversely, if C1 is Ethereum, then the same attack can be performed unless the Broker implementation in C2 imposes a maximum pre-image size.

Suggested remedy: for atomic swaps to be safe, all implementations should enforce a maximum pre-image size that is known to be safe to use in all targeted blockchains.

2. Reentrancy in `deposit` during call to `networkTrade`

The `networkTrade` method allows for whitelisted Dapps to perform trades against the Broker, matching orders in the Broker against the Dapp's own. That operation requires that the Dapp to transfer any outstanding ETH/token balance to Broker, which is obtained by the Broker contract calling the `trade` method from within the Broker's `networkTrade` method. The behavior of the Dapp is validated by comparing the Broker contract's balances before and after the call to `trade` and checking that it increased by the appropriate amount.

Since `networkTrade` calls an external method in another contract, it is possibly vulnerable to reentrancy attacks. The method is annotated with the `nonReentrant` modifier, which guarantees the external method can't call back `networkTrade` or any other method in the Broker contract that is also annotated as `nonReentrant`.

Unfortunately, the `nonReentrant` modifier *doesn't* prevent the external method from calling back other methods in Broker *not* marked as `nonReentrant`. In particular, the method `deposit` -- used by users for depositing ETH -- isn't marked as a `nonReentrant` and can be called from within the Dapp's `trade`.

That means a malicious (or vulnerable) Dapp can complete a network trade by depositing ETH in a *specific contract's* balance in Broker, rather than just tranfering ETH to the Broker. That way, the network trade method can complete successfully, as the balances before and after `trade` will match the expectations, but the attacker can then withdraw the ETH back from the Broker, effectively stealing ETH from the contract.

Suggested Remedy: make the `deposit` method `nonReentrant`.

3. DoS due to permissive method `markNonce`

The BrokerV2 contract uses nonces to prevent messages signed by users from being replayed -- that is, prevent an attacker from reusing the same message and signature to repeat the operation without consent from the signing user.

The previous version of the contract -- let's call it V1 -- also used nonces for the same purpose, but a key change in implementation was introduced in V2. While in V1 the replay prevention code stored in the blockchain state the hash of the signed message (including nonce), so that an identical message can't be used twice (but a nonce can be reused), the V2 contract stores only the nonce, thus preventing other messages from using the same *nonce*.

However, the method `markNonce` can be freely executed on any nonce by any whitelisted contract (a "spender" contract). This allows a malicious or vulnerable extension to effectively DoS a user, by front-running their messages, and marking their nonces as used before they are processed by the Broker.

That issue is particularly severe since the order in which nonces are used must be reasonably predictable in order to take advantage of the gas cost reductions mentioned above, which means front-running is quite easy.

While it's understood that "spender" contracts need to be white-listed by the Broker administrator, these extensions are also supposed to be white-listed by each specific user who wants to have access to the extended functionality. The attack described above, however, breaks that barrier and allow a "spender" contract to mark nonces currently assigned to users that haven't explicitly authorized them.

That means that, while the attack might not be practical given thorough vetting of extension contracts by the Broker administrator -- it still breaks the trust model of extensions.

Suggested remedy: create separate used nonce maps for each "spender" contract. See issue (5) for another suggestion.

Potential Vulnerabilities

4. Stateful methods do not guard against reentrancy

As explained in issue (2), the modifier `nonReentrant` is used in the contracts as a form of protection against reentrancy attacks. The modifier, however, isn't used in all external methods that can cause state changes, rather being added only to the methods that actively call other contracts.

That approach doesn't fully protect against reentrancy attacks, since a contract called from a method marked as `nonReentrant` can still call back the Broker contract's methods that are not annotated with that modifier.

While the attack described in (2) is the only concrete example of reentrancy vulnerability found, it's possible that other methods can be used to the same effect.

Suggested remedy: add the `nonReentrant` modifier to all external and public methods that aren't views.

5. Lack of isolation of nonce uses

As explained in issue (3), the Broker contract prevents nonce reuse in order to protect against replay attacks. However, since nonces are global, they can't be reused between different users or contracts.

This design decision makes it possible in general to prevent one user's message from being processed, by processing first a message that uses the same nonce.

As afar as the author was able to determine, all methods that require a nonce must be called by an administrator -- with exception of the method `markNonce` as explained in issue (3). Thus it seems to be theoretically possible to construct a coordinator system in such a way such that no nonce collisions can happen, assuming issue (3) is fixed.

Nevertheless, the author would recommend isolating nonce use in a more fundamental level in the contract itself, thus defending against unforeseen scenarios arising from issues in the contract or flaws in the coordinator.

While the present solution might allow for better gas savings, we think that the added security of better isolation between users is worth the extra cost.

Suggester remedy: isolate nonce use by signer, i. e. have a separate used nonces map for each public key. This would still allow for gas savings when the same user sends multiple messages. At a minimum consider carefully whether the increased gas savings are truly worth the weaker isolation.

Design Issues

6. Replicated validation logic

Throughout the code, there are numerous examples of the same validation repeated in different methods, evidencing code reuse by copying. See, for example, the logic for ensuring round values are used both in `validateMatches` and `validateNetworkMatches`.

While we understand that function calls can be expensive in terms of gas, having multiple copies of the same logic is universally considered a bad software engineering practice, as it makes maintenance harder. Regarding security particularly, it makes it likely that fixes are only locally applied and issues linger in other part of the code base.

Suggested remedy: consider extracting all common logic into private methods. In each instance, the risk of introducing issues should be carefully weighed against the benefits of minor gas savings.

7. Replicated parameter decoding logic

Most methods in the Broker contract use a pattern for encoding parameters in a "compacted" form, that keeps arrays of values rather than named parameters, and even encode different parameters in different bits of these values.

While that strategy help keep gas costs low, and reduce transaction sizes, it also makes the code less readable. Moreover, the logic for converting the compacted values into actual, meaningfully named parameters is spread throughout the code. Often the logic for the extraction of the exact same bits from the parameters is replicated at different points in the same method, and between methods, and the only indication of the meaning of the values being extracted lies in comments.

That pattern makes the code significantly harder to read and follow, and means any updates to the input format needs to be carefully applied to multiple points in the code.

Suggested remedy: parameter encoding and decoding logic used in more than one point should be extracted to private methods (or local variables) whenever possible. Even when the logic is used just once, the values should still be extracted into local variables before use, making the code more readable.

8. Widespread use of magic numbers

This issue is related to (7) as it occurs most often in the parameter decoding logic. In particular, number literals that indicate indices of values and bit position of values in method arguments are almost never extracted into constants and very often repeated throughout the code base.

Again, the issue makes the code very hard to read and make maintenance more costly.

Suggested remedy: Good software engineering practice would recommend these

values be extracted in constants so that their meaning becomes obvious, and we don't need to rely on multiple similar comments to explain logic.

9. Unnecessary use of nonce in `depositToken`

The `depositToken` method uses the ERC-20 authorization functionality for limiting the amount (if any) of tokens the Broker contract can transfer from the user's balance. That means adding any allowance to the Broker contract implicitly authorizes it to grab the tokens.

The method, however requires a `nonce` parameter that is marked as used. The use of the nonce doesn't add any security, increases the gas cost, and is just confusing. It might also lead a reviewer to misunderstand the security model of the method, leading to not catching newly introduced issues.

Suggested remedy: the `nonce` parameter and corresponding validation should be removed from `depositToken`.