Avely Scilla Contracts Security Audit

February 27th, 2023 (updated March, 9th)

@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 Avely smart contracts, containing:

- **Methodology**: a general description of the methodology used to find possible issues.
- **Potential Vulnerabilities**: a list of actually issues found that might lead to vulnerabilities depending on factors internal or external to the smart contracts being audited, depending on assumptions about configurations, fees, other contracts interacted with or other external pieces of technology.
- **Attack Scenarios**: a brief description of further attack scenarios considered during this audit, even though no means of performing these were found.

Scope

The scope of this audit are the contracts `stzill.scilla`, `buffer.scilla`, `holder.scilla` and `aswap.scilla`, as found at https://github.com/avely-finance/avely-contracts, at the commit `f70cfc5a9c427809ed22af732bf9c6b60075b454`.

Update: fixes to the issues have been reviewed as of commit `b3586bd32d4f9c999b6ca1782e536f8f1106f02c`

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 higher severity issues than using either individually.

Potential Vulnerabilities

This section describes issues that might be exploitable, depending on a combination of factors internal (such as configuration, fees, etc) or external (such as specifics of other contracts holding stZIL) to the contracts.

In this audit the risk that the issues listed below be used for concrete attacks in the deployed contracts was evaluated as low, given the expected use cases of contracts.

However, it is important the development team evaluate these risks themselves, and weigh the risks of changing the code to fix potential issues with the likelihood and severity of attacks, as information outside of the code base and documentation might be available -- and lead to a different evaluation.

1. [Low] Aswap: Rounding error can lead to stealing tokens under specific conditions

The transition `AddLiquidity` in the Aswap contract (`aswap.scilla`) calculates the amount of tokens to be transferred from the caller based on the amount of the native token (ZIL) sent in the transition message. The code that calculates that amount is as follows:

```
        (* dY = dX * Y / X *)
        (* dX is always the QA transferred *)
        result_u256 = fraction amount_u256 zil_reserve_u256 token_reserve_u256;
        result_u128 = fall_u256 result_u256;
```

where

```
   (* computes the amount of the fraction x / d that is in y *)
   let fraction: Uint256 -> Uint256 -> Uint256 -> Uint256 =
     fun (d: Uint256) =>
     fun (x: Uint256) =>
     fun (y: Uint256) =>
       let d_times_y = builtin mul d y in
```

```
builtin div d_times_y x
```

Please note that division *rounds down* by default. That means the amount of tokens collected from the sender is *almost always* down by one minimal unit, which means the amount of `contribution` obtained is slightly in excess of what it should.

While in most cases any excess amounts obtained by an attacker have minimal value, in certain extreme corner cases the issue might allow the stealing of funds by an attacker.

Hypothetically, let's consider an example where all fees are zero, and the minimum amount of ZIL contributions is zero or very small:

1 - Prior to the attack, let's assume the pool's reserves are 32zil / 32stzil. Let's also assume other users have 32 "contribution" shares.

2 - An attacker buys almost all stZIL from the pool, leaving reserves at 1024 / 1 (attacker spends 992 ZIL, and buys 31 stZIL).

3 - Attacker *adds liquidity* to the pool with 1023 ZIL and 0 stZIL (due to rounding down).

4 - Reserves are now 2047/1, the attacker spent 2015 ZIL, and obtained 31 stZIL.

5 - Now the attacker sells back his 31 stZIL, obtaining back 1983 stZIL.

6 - Reserves are now 64/32.

7 - Now attacker *removes his 31 liquidity shares*, obtaining 31 ZIL, and 15 stZIL.

8 - Reserves are now 33/17. The attacker spent a total of 1 ZIL, and obtained a total of 15 stZIL, with a net profit of 14 ZIL.

Please note that the above scenario is very unlikely in an stZIL/ZIL pool, given minimum ZIL contributions and fees.

If liquidity shares are ever tokenized and loaned, that might make the attack more realistic though. If pools are created for tokens that have a small number of decimals and an extremely low price compared to ZIL, the attack might become feasible as well. It is also possible other scenarios exist that have not been identified, which would make exploiting the rounding issue more realistic.

Suggested remedy: amount of token *provided* by the sender should be *rounded up*. All amounts *obtained* by the sender should be still *rounded down*. As a general rule, this means that any rounding errors benefits the contract, making exploiting them to steal funds impossible.

Update: the development team has acknowledged the issue and implemented a fix as of the latest commit reviewed.

2. [Low] Aswap: "share reset" attacks

The `RemoveLiquidity` transition in `aswap.scilla` allows for removing all liquidity in the pool, which leads to all pool data being removed.

A side-effect of this is that the value of the "contribution share" is reset to 1 ZIL = 1 contribution share.

If the contribution shares are ever tokenized and loaned (by external contracts that don't necessarily are vetted by Avely), it allows for a class of attack where:

1 - Attacker loans *all* liquidity shares (at a value higher than 1/1).

2 - Attacker removes *all* liquidity from pool.

3 - Attacker adds back liquidity, thus resetting the share value. Note that fewer ZIL/tokens would be needed here.

4 - Attacker repays the liquidity loan, that is now worth less in terms of ZIL/tokens.

While this would technically be an attack on the loan contract, rather than the pool, it might be interesting to consider future-proofing the pools against it -- in particular if there are any plans to allow users to tokenize the liquidity shares in a contract that then loans it.

That scenario might seem too far-fetched but this report's author has seen (and reported) that very scenario in a mainnet contract holding millions in funds.

Suggested Remedy: follow the best practice in UniswapV2 pairs, and keep a tiny share of liquidity that can never be removed. It's important to weigh the risks of changing the code at that stage with the low likelihood of the attack in the near future.

Update: the development team acknowledges the issue, and will prevent it by manually adding to each pool a minimum liquidity that will not be removed.

3. [Informative] Aswap: Deflationary tokens lead to incorrect reserve calculations

While this limitation is likely to be known to the development team, it is still worth noting it for completeness.

The Aswap contract only accounts for increases and decreases in reserve amounts as results of its own transitions. That is, the balance of the contract isn't checked.

That means that any token that allows for changes in balances by ways other than direct transfers of amounts (e.g. deflationary tokens that burn a fee with each transfer) will lead the Aswap contract to behave incorrectly and cause loss of funds.

Suggested Remedy: as this is likely an explicit design decision by the development team, the only suggestion is that the limitation that the contract doesn't support deflationary tokens be made clear in all documentation as well as comments in the code.

Update: the development team acknowledged the issue, and implemented a fix in the form of a whitelist of tokens, as of the latest commit mentioned above.

4. [Low] stZIL: DoS withdrawals by staking and withdrawing large amounts in the same cycle.

There's a single stZIL balance for each user that stakes their ZIL in the stZIL contract, and the staking operation makes the stZIL available immediately after the deposit.

However ZIL staked is transferred at first to one of the `buffer` contracts (after which it is delegated to an SSN), while any withdrawal is removed from the `holder` contract.

That means it is possible for a user to stake a large amount of ZIL, and immediately initiate the withdrawal of the same -- emptying the holder contract of funds (delegated to SSNs) for initiating withdrawals.

This process could be used to DoS other user's withdrawals.

It must be noted that there are a few *mitigating factors* that limit the impact of this issue:

- The ability to swap ZIL for stZIL in an "Aswap" pool (though that incurs in fees and slippage).
- The cost of the attack in terms of fees and loss of liquidity (until the attacker's withdrawal is completed).

Given the above this can be understood as a *grieving attack* as costs for the attacker would likely exceed damage caused to legitimate users.

Suggested Remedy: consider adding a cool down period between staking and withdrawals. Users that want to immediately access stZIL can still obtain the tokens by swapping in the ZIL/stZIL pool. The risk of introducing this change should be weighed against the low impact of the issue.

Update: the development team acknowledges the issue, and evaluates the risk and impact are too low to warrant a major, risky change in the code.

5. [Informative] Aswap/stZIL MEV by swaping ZIL/stZIL before and after a `PerformAutoRestake`

Claimed rewards are incorporated into the stZIL contract's reserve balance after a successful call to `PerformAutoRestake`, increasing the value of each stZIL.

This means an attacker who can place a transaction to obtain stZIL (swap or staking) right before the re-staking, and a transaction right after the re-staking to return the stZIL for ZIL (swap of withdraw) would be able to extract value from the contract (aswap or stZIL). This is particularly relevant if the attacker uses an amount of ZIL comparable to the stZIL reserves.

Suggested Remedy: this possible attack can be made impractical by claiming rewards and re-staking often, while keeping swap fees and withdrawal fees high enough to compensate for any rewards obtained in a possible value-extraction attack. No change in the contract code should be needed.

Update: the development team acknowledges the issue, and that no code change is needed to address it.

Further Attack Scenarios

This section describes the main (non-exhaustive list) attack scenarios considered during the audit of the code. *No vulnerability* was found that could enable an attacker to perform the attacks listed below.

- **Unauthorized user can perform privileged transitions on contracts.**

  A malicious user, without proper authorization, is able to perform administrator or other privileged transitions in any of the contracts.

- **Unauthorized movement of user balances.**

  A malicious user, without proper authorization, is able to perform any movement of funds that belong to another user, such as: transfer, staking, withdrawing, removing liquidity.

- **Unauthorized movement of contract funds.**

  A malicious user, without proper authorization, is able to perform any movement of funds owned by the contract, such as: transfer, staking, withdrawing, removing liquidity.

- **Double-spending.**

  A malicious user is able to withdraw twice the same stZIL balance, or removing the same share of liquidity twice from "Aswap".

- **DoS by adding too many withdrawals in the same block.**

  A malicious user could initiate too many withdrawals in the same block, leading to calling `ClaimWithdrawal` for that block too costly.

- **DoS of Buffer contract rotation.**

  A malicious user is able to manipulate the stZIL contract in such a way that it can't perform the round-robin rotation of active buffers -- thus preventing new staking or delegation.

- **Reentrancy Attacks.**

  While the Scilla language, and the Zilliqa execution model prevent classical, direct

reentrancy attacks, it is important to note that whenever state changes or validations are performed in a callback transition, after a call to an externally controlled contract, there *is* the possibility of an attack equivalent to a reentrancy.

That happens as breaking the logic in two parts, with the execution of an external contract between the two is the exact equivalent of a direct call. In programming language theoretical terms, Solidity (and the EVM) uses *direct style calls* while Scilla (and Zilliqa) use *continuation passing style calls*. Since both are equivalent, they are also vulnerable to the same sort of attacks.

In the same way as the checks-effects-interactions pattern can prevent reentrancy attacks in Solidity, Scilla contracts can prevent such "pseudo-reentrancy attacks" by never placing logic in a callback called by an untrusted contract.

Conclusion

This document described the methodology for conducting the audit of the code, and some of the potential attacks considered.

All issues found are described, and the risk of a practical exploit is evaluated. While the risk is considered low for all reported issue, possible further mitigations and improvements were discussed.

Update: the issues reported have been acknowledged. Two issues were fixed in the code base, while the others will either be mitigated without code changes, or are considered of too low risk/severity require any action.