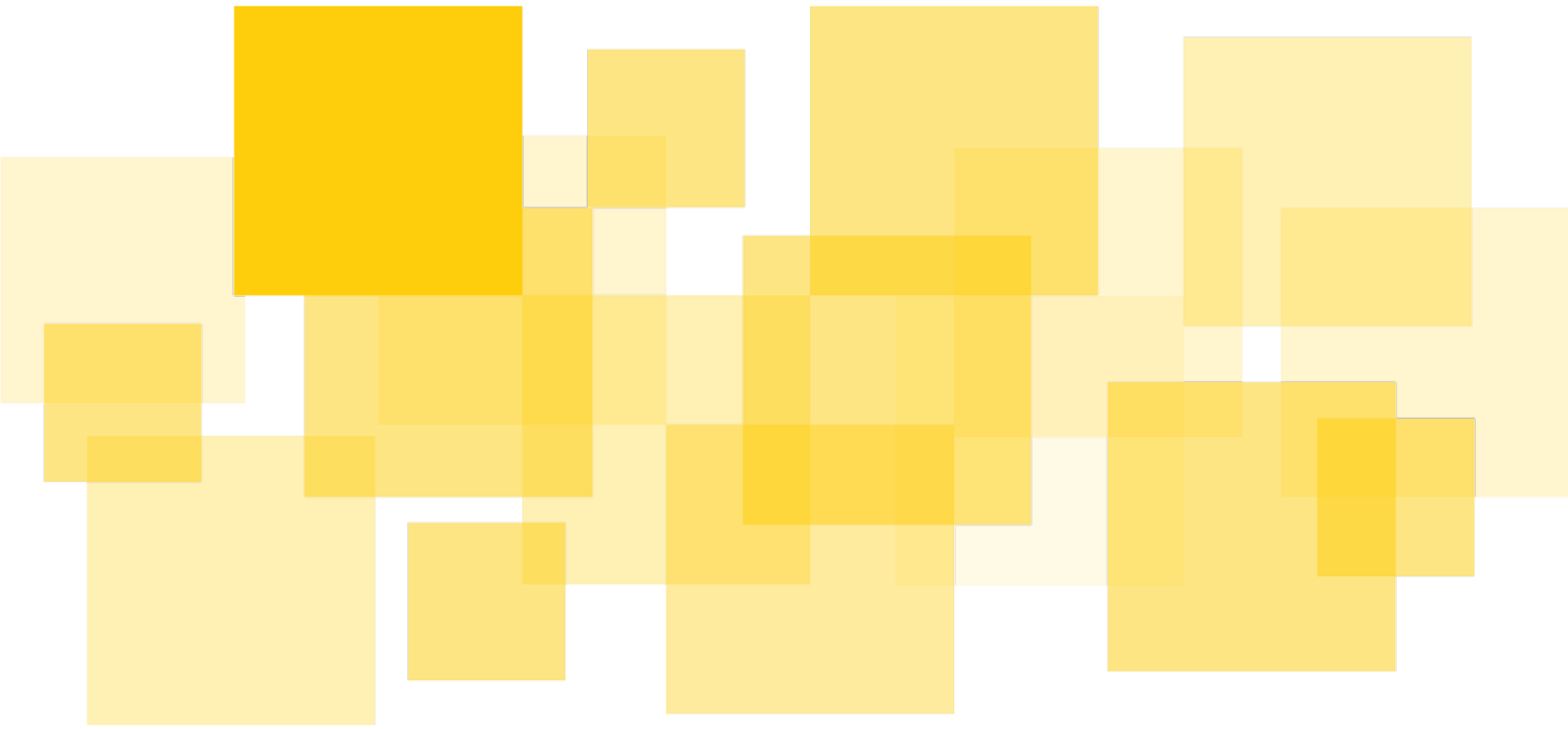
Security Audit Report

Yieldly Finance Multi-token Staking Pool

2 December, 2021



Prepared for Yieldly Finance by

runtime verification

Contents

Summary

Yieldly Finance engaged Runtime Verification Inc to conduct a flash security audit of the smart contract implementing their multi-token staking pool.

The objective was to review the contract's business logic and implementation in TEAL and identify any issues that could potentially cause the system to malfunction or be exploited.

Timeline

The flash audit has been conducted over a period of 2 weeks.

Scope

The audit was conducted by Georgy Lukyanov on the following artefact provided by Yieldly Finance:

- Multi-token Staking Pool Smart Contract staking.teal, as of commit 99b5f89babbfb5bf4709523f480c94112e08ad8c.

Further patches and developments following that commit were not formally audited. However, in this report we acknowledge small patches related to some of the findings.

Disclaimer

This report does not constitute legal or investment advice. The preparers of this report present it as an informational exercise documenting the due diligence involved in the secure development of the target contract only, and make no material claims or guarantees concerning the contract's operation post-deployment. The preparers of this report assume no liability for any and all potential consequences of the deployment or use of the contract.

Smart contracts are still a nascent software arena, and their deployment and public offering carries substantial risk. This report makes no claims that its analysis is fully comprehensive, and recommends always seeking multiple opinions and audits.

This report is also not comprehensive in scope, excluding a number of components critical to the correct operation of this system.

The possibility of human error in the manual review process is very real, and we recommend seeking multiple independent opinions on any claims which impact a large quantity of funds.

Goals

The goals of the audit are:

- Review the architecture of the multi-token staking pool smart contract based on the provided documentation;
- Review the TEAL implementation of the contract to identify any programming errors;
- Cross check the TEAL code of the contracts with the documented high-level design.

The audit focuses on trying to identify issues in the system's logic and its implementation that could potentially render the system vulnerable to attacks or cause it to malfunction.

Scope

We audit the multi-stacking pool smart contract source code, staking.teal, in Yieldly Finance private GitHub repository.

Additionally, we use the description of atomic transaction groups and flowcharts provided by Yieldly Finance as documentation.

The audit has been performed on the frozen commit 99b5f89babbfb5bf4709523f480c94112e08ad8c. Further patches and developments following that commit were not formally audited. However, in this report we acknowledge small patches related to some of the findings.

Methodology

The timeline was tight, therefore we have only carried out a best-effort flash audit. That means that we were not able to perform all the checks we deem important, but we still strove to perform the best possible analysis.

We carried out the flash audit in the following roughly-defined stages:

1. We reviewed the documentation provided by Yieldly Finance to get acquainted with the system and build an understanding of its functions. To clarify any ambiguities, we communicated with Yieldly Finance, who were very responsive and quickly provided answers to our questions about the system.
2. We performed a manual code review of the contract's implementation in TEAL, which led us to identifying several potential problematic scenarios.
3. To test the potentially problematic scenarios identified, we have set up a private Algorand network, and deployed the smart contract onto it, along with the required accounts and assets configurations. We modified the provided test-suite to include the problematic scenarios and tested our hypotheses. The confirmed problematic scenarios can be found on the findings section.
4. In the time remaining, we analysed the rewards calculation logic to identify any potential problems introduces by rounding errors. We built a Python model to simulate the rewards unlocking logic and identified several potential improvements. See findings for more information.
5. Finally, we allocated time at the end of the schedule to do a final sweep through the code and prepare this report.

Findings

A1. Unchecked transaction in `stacking` allows shadow staking

Description

The `check_bail` subroutine call on line 568 is supplied with argument `int 3`, whereas it should be `int 2`. With the current argument, the `check_bail` function checks the transactions starting from the **forth** one in the group, not the third one. Therefore, the third transaction remains unchecked by the logic in `check_bail`.

The problem is somewhat mitigated by the `bail` subroutine which processes the transaction when it becomes the current one. The `bail` subroutine only approves `NoOp` application calls with the first item in `ApplicationArgs` set to the string `"bail"`. This measure blocks any other application call transactions being placed in that slot.

However, a user can still place an asset transfer transaction in this slot, thus effectively allowing them to shadow stake any amount or type of tokens, including zero, bypassing the rewards calculation logic. Even though this scenario does not seem an immediate candidate for a potential attack, it still should not be allowed.

Recommendation

Change the integer constant on line 568 with the appropriate one, which would reflect the index of the first `"bail"` transaction in the group. In this case, `2`.

Status

Addressed. The constant has been updated in one of the further commits to the repository.

A2. Unchecked transaction in `withdraw` allows shadow staking

Description

The `withdraw` subroutine has a off-by-one error in the call to `check_bail` subroutine on line 657. The implications are similar to the finding A1.

Recommendation

Change the integer constant on line 657 with the appropriate one, which would reflect the index of the first `"bail"` transaction in the group. In this case, `1`.

Status

Addressed. The constant has been updated in one of the further commits to the repository.

A3. Unchecked transaction in `close`

Description

The `close` subroutine has a off-by-one error in the call to `check_bail` subroutine on line 929. The implications are similar to the finding A1 and A2

Recommendation

Change the integer constant on line 929 with the appropriate one, which would reflect the index of the first `"bail"` transaction in the group. In this case, `1`.

Status

Addressed. The constant has been updated in one of the further commits to the repository.

A4. Accidentally deleting the application can leave assets locked indefinitely

Description

The code handling the `DeleteApplication` transaction approves the request for deletion of an application signed by an authorised administrator account. However, the code does not transfer the application assets upon deletion and does not handle the local state of accounts who opted in. Upon deletion, the Algorand network will not have the `ApplicationID` which used to be assigned to the contract, but will still have the contract account holding all the assets which are now inaccessible.

Status

Acknowledged.

The development team at Yieldly Finance plans to block the application deletion functionality in the final version of the pool smart contract. The version to be deployed on Algorand mainnet will not be prone to this problem.

A5. Inconsistent reward amounts unlocked upon changing parameters of live pools

Description

The contracts provides endpoints for prolonging a pool's lifespan after its creation and providing additional rewards. The rewards unlocking logic can behave inconsistently, i.e. unlocking large or very small portions of rewards if these endpoints are used carelessly.

Recommendation

We recommend making sure that the proportions of additional rewards supplied and new pool end date are in sync with the original configuration of the pool

Status

Acknowledged.

The development team at Yieldly Finance plans to block remove this functionality in the final version of the pool smart contract. The version to be deployed on Algorand mainnet will not be prone to this problem.

A6. Emergency withdrawal admin subroutines pose a significant centralised risk

Description

The contract contains a subroutine called `emerg_withdraw` and another one called `emerg_withdraw_algo`. These subroutines are only available for the smart contract's administrator to execute.

On an approved `NoOp` application call transaction calling these subroutines, the smart contract issues an inner transaction transferring the requested amount of tokens (either ASA or Algos) to the administrator.

If an attacker was to gain control of the administrator account, they would have a straightforward way to withdraw all funds from the pool.

Recommendation

Such functionality is a double-edged sword. If it is to remain in the contract, we suggest making the starkers aware of its implications. Additionally, we suggest making sure the administrator account is a multi signature one.

Status

Acknowledged, intentional design decision.

Additional Findings

B1. Unnecessary re-computation of pool lifespan

Description

The `update_rewards` subroutine, computes the pool lifespan (1) on every call.

$$T = t_{end} - t_{start}$$

Figure 1: Pool lifespan

Recommendation

The pool lifespan, i.e. the difference `end_timestamp - start_timestamp`, can be precomputed. This simplification will save some computation costs at every interaction with the pool.

Status

Acknowledged.

B2. Possible delay in reward unlocking close to pool's creation time

Description

Fixed-point rounding errors in the logic could cause delay rewards unlocking at close to pool's creation time.

If the difference between the current timestamp $t_{current}$ and the pool creation time t_{start} is small, and the value of the rewards locked RL is small too, the fraction

$$\frac{(t_{current} - t_{start})RL}{t_{end} - t_{start}}$$

can go to zero because of truncation in the integer division. Therefore zero rewards will be unlocked at that step.

Recommendation

To avoiding a possible rounding error in the calculation of rewards unlocked, we suggest moving the scaling factor 10^{10} before the division by the pool lifespan value.

Status

Acknowledged.

B3. `txn/gtxn` opcodes used to access array fields

Description

Some instances of `txn` and `gtxn` opcodes are used to access the array fields. Interestingly, in other places the proper opcodes, i.e. `txna` and `gtxna` are used to do the same.

Recommendation

We recommend to bring consistency and do not depend on the undocumented behaviours of the TEAL assembler.

Unsolicited uses of `txn`: 117, 121, 125, 130, 1187, 1312, 1316, 1360, 1392, 1397, 1423, 1512.

Status

Acknowledged.

B4. Unnecessary update of user's stake on close out

Description

The `close` subroutine sets the variable `"User_Stake"` in the sender's local state allocated to the application to zero.

Recommendation

On `CloseOut`, the sender will have their local state deleted, therefore it is safe to remove this code.

Status

Acknowledged.

B5. Unreachable code in `staking.teal`

Description

The very end of the file contains subroutines `failed` and `finished` which are not called anywhere in the code.

Recommendation

These subroutines can be safely removed to reduce the code size.

Status

Acknowledged. The subroutines have been remove in a subsequent patch.