Choise Security Analysis

by Pessimistic

This report is public

July 4, 2022

Abstract

In this report, we consider the security of smart contracts of Choise project. Our task is to find and describe security issues in the smart contracts of the platform.

Disclaimer

The audit does not give any warranties on the security of the code. A single audit cannot be considered enough. We always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contracts. Besides, a security audit is not investment advice.

Summary

In this report, we considered the security of Choise smart contracts. We performed our audit according to the procedure described below.

The initial audit showed one critical issue: Arbitrary memory manipulation. The audit also revealed several issues of medium severity: Unfinished Code, Missing security check, Overpowered owner. Moreover, several low-severity issues were found.

After the initial audit, the codebase was updated. The developers fixed the attack enabled by Arbitrary memory manipulation and several other issues. However, the project architecture retained its flaws. Also, one new low-severity issue was found.

Later the developers provided codebase update with the fixes to all found issues.

After the fixed, the system still has some limitations, that we described in the Notes section. However, the code quality has greatly improved and the code meets business requirements.

General recommendations

We recommend redesigning the smart contracts in a more conventional way.

Project overview

Project description

For the audit, we were provided with Choise project on a private GitHub repository, commit 6a09bd0657eae2dc1fa4d5c173f8a7c1f6022303.

The scope of the audit includes everything.

We were provided with the internal link to the documentation.

All 18 tests pass, test coverage is 90.63%.

The total LOC of audited sources is 142.

Codebase update #1

After the initial audit, the codebase was updated. For the recheck, we were provided with commit 416f92a6a95fea9f658a87bf735ad06b2d54ea79. The developers addressed several issues. However, a new low-severity issue was discovered.

All 21 tests pass, test coverage is 81.19%.

Codebase update #2

After the first recheck, the developers updated the codebase. For the second recheck, we were provided with commit 16e23a26b95f43b2b1a396ef2b49864af21a5f28. The update contains fixes to all issues in the report.

All 21 tests pass, test coverage is 73.68%.

Procedure

In our audit, we consider the following crucial features of the code:

1. Whether the code is secure.
2. Whether the code corresponds to the documentation (including whitepaper).
3. Whether the code meets best practices.

We perform our audit according to the following procedure:

- Automated analysis
  - We scan the project's codebase with the automated tool Slither.
  - We manually verify (reject or confirm) all the issues found by the tool.
- Manual audit
  - We manually analyze the codebase for security vulnerabilities.
  - We assess the overall project structure and quality.
- Report
  - We reflect all the gathered information in the report.

Inter alia, we:

- Compared the project with analogues
- Verified that no standard Solidity issues are present in the codebase.
- Checked the correctness of the assembly code (see the following issue).

Manual analysis

The contracts were completely manually analyzed, their logic was checked. Besides, the results of the automated analysis were manually verified. All the confirmed issues are described below.

Critical issues

Critical issues seriously endanger project security. They can lead to loss of funds or other catastrophic consequences. The contracts should not be deployed before these issues are fixed.

C01. Arbitrary memory manipulation (fixed)

In the code, there is an issue with memory manipulation that allows an attacker to steal users' funds.

Pipeline contract includes a `mstore()` assembly command at line 45. It updates a `data + pos` memory slot with a `value` derived from earlier computations. However, the attacker can manipulate both `pos` and `value`, which allows them to corrupt the memory. Since internal functions don't have separate memory space, the attacker can modify any parameter stored in memory, including `targetData`. They can update `targetData.target` value with `pipelineProxy` address, effectively bypassing security checks at lines 116-118. Without these checks, the attacker can transfer any tokens approved to the **PipelineProxy** contract.

The issue has been fixed and is not present in the latest version of the code.

Medium severity issues

Medium issues can influence project operation in the current implementation. Bugs, loss of potential income, and other non-critical failures fall into this category, as well as potential problems related to incorrect system management. We highly recommend addressing them.

M01. Unfinished code (fixed)

In the current implementation, `callFunctionWithFullSignature` has `public` visibility for testing purposes. One can use it to steal any tokens approved to **PipelineProxy** contract. `callFunctionWithFullSignature` function visibility should be changed to `internal` before deployment, as it is stated in the comment.

The issue has been fixed and is not present in the latest version of the code.

M02. Missing security check (fixed)

Most contracts that transfer tokens to and from users include a minimal expected return check. It protects users from attacks and bugs and is especially important if the contract calls arbitrary contracts. E.g., it helps against slippage, sandwich attacks, and off-chain code malfunction.

We highly recommend adding minimal return parameter to `run` and `runWithPool` functions and providing reasonable values when preparing transaction data for the users.

The issue has been fixed and is not present in the latest version of the code.

M03. Overpowered owner (fixed)

The project heavily relies on the `owner` role. It can change the trusted contract in **PipelineProxy**, i.e. the contract that is allowed to transfer user tokens to anyone. Moreover, the owner is able to change proxy address in **Pipeline** contract.

There are scenarios that can lead to undesirable consequences for the project and its users. E.g., if the private keys of the owner become compromised. We recommend designing contracts in a trustless manner or implementing proper key management, e.g., setting up a multisig.

The issue has been fixed and is not present in the latest version of the code.

Low severity issues

Low severity issues do not directly affect project operation. However, they might lead to various problems in future versions of the code. We recommend fixing them or explaining why the team has chosen a particular option.

L01. Gas Optimization (fixed)

Since **Pipeline** contract does not store any tokens, it is safe to give infinite approvals from **Pipeline** contract to swapper contracts. This optimization will reduce gas costs when interacting with popular contracts.

The issue has been fixed and is not present in the latest version of the code.

L02. Gas Optimization (fixed)

Consider marking `ETH_ADDRESS` variable of **Pipeline** contract as constant to reduce gas consumption.

The issue has been fixed and is not present in the latest version of the code.

L03. Low-level code (fixed)

`callFunctionWithFullSignature` directly modifies memory with assembly to prepare data for the next external call. This approach violates important safety conventions and might lead to hard-to-detect issues, including the one described above. One can achieve similar results with the help of built-in encoding functions.

The issue has been fixed and is not present in the latest version of the code.

L04. Incorrect ETH support (fixed)

`distToken` variable indicates the resulting currency that will be sent to the user. According to lines 127-142 and 232-245 in **Pipeline.sol**, `distToken` can be either ERC20 token or ether. However, lines 144 and 248 inquires ERC20 token balance. These lines will not work if `distToken` is ether and stores `ETH_ADDRESS` value.

The issue has been fixed and is not present in the latest version of the code.

Notes

N01. Limitations of the project

The project aims to integrate various DeFi protocols under one interface and create a single-transaction deposit. However, the project has some limitations on its use cases. Currently, the project only supports protocols requiring a single token for staking. Among other things, it also does not cover protocols that do not have tokenized positions and do not have `depositFor` function. In addition, the `Pipeline` contract does not support token unstaking: users have to interact with protocol contracts directly. Moreover, the project does not support "money-lego" pools.

> *Comment from the developers:*
>
> *Due to our business requirements, we need to support only projects that have single token staking. It's ok for us, because many projects allow to enter multitokens pool through single token.*
>
> *Also we don't support nontokenized positions, because if we can't deposit tokens for user, or deposit for us and then transfer tokens to user, then only option for user is to deposit by himself, without intermediate contract. Such case will be resolved with front-end.*
>
> *Token unstaking is already implemented with Pipeline contract. The process is the same as staking: user allows staking contract to PipelineProxy and then call Pipeline, that will unstake token and swap it to desired token.*
>
> *"Money-lego" pools also is not supported due to their rareness.*

N02. Protocol tokens handling

The code is designed to work with multiple protocols: both tokenized and non-tokenized. However, in the current implementation, `balanceOf()` function of the expected LP token is always called at lines 92 and 156. We recommend adding a special value for `distToken` variable that indicates that `depositFor()` function was used and that the protocol has no LP token

N03. Reliance on the backend

Pipeline contract operates with complex transactions with custom data. The project relies on off-chain code to prepare transaction data. Since any error might lead to a loss of funds for the user, we recommend paying extra attention to the quality and security of the off-chain code.

This analysis was performed by Pessimistic:

Evgeny Marchenko, Senior Security Engineer
Pavel Kondratenkov, Security Engineer
Nikita Kirillov, Junior Security Engineer
Irina Vikhareva, Project Manager
Alexander Seleznev, Founder

July 4, 2022