Neuron Options Security Analysis

by Pessimistic

This report is public

June 7, 2022

Abstract 2
Disclaimer 2
Summary 2
General recommendations 2
Project overview 3
Project description 3
Codebase update 3
Procedure 4
Manual analysis 5
Critical issues 5
Medium severity issues 6
M01. Incorrect require condition (fixed) 6
Low severity issues 7
L01. Unused code (fixed) 7
L02. Functions visibility (fixed) 7
L03. Possible locked tokens (fixed) 7
L04. Gas consumption (fixed) 7
L05. Unused error codes (fixed) 8
L06. Unchecked array lengths (addressed) 8
L07. Possibility of data loss (fixed) 8
Notes 9
N01. NatSpec issue (fixed) 9
N02. Trustful design 9
N03. Price manipulation risk 9
N04. Overpowered role 9
N05. Ambiguous array hashing 10

Abstract

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

The initial audit identified one issue of medium severity: Incorrect require condition. Also, several low-severity issues were found. After the initial audit, the codebase was updated. The developers fixed or acknowledged all of the issues from the initial audit.

General recommendations

We recommend adding detailed documentation to the project.

Project overview

Project description

For the audit, we were provided with Neuron Options project on a public GitHub repository, commit e9d7dfafe56fa245829553ec35ec0840f302397b.

The project has a README.md file with a short description of the project. The code has detailed NatSpec comments. However, there is no detailed documentation available.

All 98 unit tests pass. However, the code coverage is not measured. Also, all 50 integration tests fail.

The total LOC of audited sources is 2957.

Codebase update

After the initial audit, the codebase was updated. For the recheck, we were provided with commit 51972d1d9caa551360d947ae3a9e9aebf4e16a16.

The developers addressed all the issues and refactored the codebase. No new issues were found.

The developers also added a new test to the project and fixed the old tests.

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.

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.

The audit showed no critical issues.

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. Incorrect require condition (fixed)

In the **Controller** contract, the `_withdrawLong` function requires that `onTokenAddress == address(0)` at line 625. Therefore, the function attempts to send `IERC20(address(0))` tokens at line 634 from the pool. As a result, the transaction performs successfully, but the user does not receive any tokens. Consider modifying the check to `onTokenAddress != address(0)`.

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. Unused code (fixed)

The following code in the project is unused:

1. The `sync` function in the **Controller** contract at line 419.
2. Unused `call()` in the **Controller** contract at line 891.
3. The `_isNotEmpty` function in the **ArrayAddressUtils** contract at line 21.
4. The `uint256ArraysAdd` function in the **MarginCalculator** contract at line 264.
5. The `_isCalleeWhitelisted` function in the **Controller** contract at line 912.
6. The `BASE` constant declared at line 87 of **Controller** contract.

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

L02. Functions visibility (fixed)

In the **MarginVault** library, consider declaring functions as `external` instead of `public` at lines 74, 100, 171, 190, 210, 224, and 251 to improve code readability and optimize gas consumption.

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

L03. Possible locked tokens (fixed)

In the **Controller** contract, anyone can call the `donate` function. However, this function does not verify that the provided `_asset` can be used as collateral. Thus, tokens might be locked. Consider adding a proper check of the `_asset` address or limiting access to the function.

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

L04. Gas consumption (fixed)

Consider saving `vaults[_args.owner][_args.vaultId]` to a local variable to optimize gas consumption and improve code readability in the **Controller** contract at lines 722, 729, 746, 748, and 751.

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

L05. Unused error codes (fixed)

Error codes documented, but not implemented: `C14,18,21,22,25,30,32-34,36-41` in the **Controller** contract.

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

L06. Unchecked array lengths (addressed)

There are functions in the project that process multiple arrays simultaneously, considering these arrays have the same lengths. However, their lengths are not checked to match.

1. In the **MarginCalculator** contract, the `uint256ArraysAdd` function does not verify that the arrays `_array` and `_array2` have the same lengths. Consider adding a proper check.

   > *Comment from the developers: Function `uint256ArraysAdd` was removed from the contract and therefore arrays length check for it is not required anymore.*

2. In the **Controller** contract, the `_redeem` function iterates through `collaterals` and `payout` arrays using the same index. Consider adding an `assert` check to verify that lengths match before the `for`-loop at line 826.

   > *Comment from the developers: Checks ensuring payout and collaterls lengths match is not neccesarry since payout array is created the same length as oToken 's collaterls array. Look at `getExpiredPayoutRate` function in MarginCalculatorMarginCalculator, line 127*

L07. Possibility of data loss (fixed)

In the **MarginVault** library, if anyone calls `addShort` function with `address(0)` value of the `_shortONtoken` argument, the next call of the function with non-zero `_shortONtoken` argument will discard the current value of `_vault.shortAmount` at line 82.

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

Notes

N01. NatSpec issue (fixed)

A separate NatSpec `@return` should be specified for each return value in the **MarginCalculator** contract at line 829.

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

N02. Trustful design

The `setExpiryPrice` function of the **Oracle** contract heavily relies on the Pricer to be called at the correct time. It does not restrict `block.timestamp` from above, so the price values can differ from the actual price in a volatile market.

> *Comment from the developers: Restricting `setExpiryPrice` from above by `block.timestamp` can lead to a never settled option if `setExpiryPrice` transaction will fail to execute due to some issue.*

N03. Price manipulation risk

The `Controller` contract requests the actual collateral price from the oracle to calculate the amount of `ONtoken` to mint. Users can manipulate the collateral price using flash loans to get more ONtokens for smaller collateral. We recommend using TWAP wherever possible.

> *Comment from the developers: It's not possible because prices are taken from chainlink.*

N04. Overpowered role

The owner of the **Oracle** contract can set the disputer role, who can change the expiry price of the option asset. Also, the owner can set the dispute period. If the owner's keys compromise, the expiry price may be set to an arbitrary value, and the dispute period may be set to zero to abuse the incorrect asset price instantly.

The owner of the **AddressBook** contract can change the address of any contract in the system to any value. This can result in the loss of assets.

We recommend designing contracts in a trustless manner or implementing proper key management, e.g., setting up a multisig.

> *Comment from the developers: We will set the owner as multisig during deploy.*

N05. Ambiguous array hashing

The **Whitelist** contract calculates a hash from an array of addresses at lines 90, 101, 138, 164, 177, and 188. If the order of addresses in the array changes, the hash value will also change. We recommend performing on-chain verification that the array is sorted before calculating the hash.

> *Comment from the developers: Sorting array onchain can lead to increased gas used. Arrays of collaterals are whitelisted by the owner role of the contract and it's presumed that we will not set the same arrays in different order twice.*

This analysis was performed by Pessimistic:

Evgeny Marchenko, Senior Security Engineer
Vladimir Tarasov, Security Engineer
Vladimir Pomogalov, Security Engineer
Boris Nikashin, Analyst
Irina Vikhareva, Project Manager

June 7, 2022