syntropy

Syntropy Security Analysis

by Pessimistic

This report is public.

Published: September 20, 2021

Abstract

In this report, we consider the security of smart contracts of Syntropy 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. One 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, security audit is not an investment advice.

Summary

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

The initial audit showed a few issues of medium severity, including Overpowered roles and Withdrawal blockage issues. Also, a few issues of low severity were found.

The project has a documentation; however, we consider it incomplete.

Some tests do not pass.

After the initial audit, the code base was updated. In this update, two contracts were added to the repository. All issues of low severity mentioned in previous report were fixed. However, a few issues were found in new contracts.

After the recheck, update #2 of the code base was performed. Also, additional documentation was provided for the project with this update.

General recommendations

We recommend fixing the mentioned issues, providing more detailed NatSpecs for the code, and decreasing the dependency of the project on the back-end. We also recommend improving code coverage, adding CI to run tests, calculate code coverage, and analyze code with linters and security tools.

Project overview

Project description

For the audit, we were provided with Syntropy project on a private GitHub repository, commit 28a28e4513b1151e4b34a6c067efc50ea42eabb9.

The documentation for the project was provided as a separate file:
Syntropy_ERC-20_Smart_Contracts_summary.pdf,
sha1sum is 7de58a620f9a5a224afe76561b757f10daeb8b13.

However, we consider this documentation insufficient.

The project compiles without any issues.

Two tests out of 47 do not pass, the code coverage is 88.89%.

The total LOC of audited sources is 165.

Code base update 1

After the initial audit, the code base was updated. For the recheck #1, we were provided with commit c4e4132ef4d20ff70cb9c5b272dfc331b49f2b4e.

In this update, **Nominator** and **NominatorOwner** contracts were added to the repository. Also, tests coverage was improved up to 100%.

Code base update 2

After the recheck #1, the code base was updated once again. For the recheck #2, we were provided with commit e677a8ae45d14656c61d5b12e7883d6c06577858.

In this update, some issues were fixed, and additional tests were added. All tests pass.

Also, additional private documentation was provided for the project.

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 project's code base with automated tools: Slither and SmartCheck.
  - We manually verify (reject or confirm) all the issues found by tools.
- Manual audit
  - We manually analyze code base for security vulnerabilities.
  - We assess 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 smart contracts security. We highly recommend fixing them.

The audit showed no critical issues.

Medium severity issues

Medium issues can influence project operation in current implementation. We highly recommend addressing them.

Overpowered roles

In **Custodian** contract, any authorized user can call `withdrawTo()` function to withdraw all the rewards to any address. The owner of the contract can authorize or deauthorize users.

Also, the owner of **NominatorOwner** contract can delay users' withdrawals by increasing `bondingPeriod` value with `setBondingPeriod()` function.

In the current implementation, the system depends heavily on the owner of the contract. Thus, there are scenarios that may lead to undesirable consequences for investors, e.g., if the owner's private keys become compromised.

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

Withdrawal blockage

When a reward is accrued to a user, the system calculates a hash for this reward, recalculates Merkle tree for the block, and updates its root hash off-chain. To store this hash root on-chain, the owner must call `updateRoot()` function of **ValidatorOwner** and **NominatorOwner** contracts. However, if an owner provides invalid hash value, users will be unable to withdraw their rewards since `isValidProof()` function will always return `false`.

Tests issues (fixed)

Some tests do not pass.

Testing is crucial for code security and audit does not replace tests in any way.

We recommend adding CI to run tests.

<u>Tests coverage was improved. All tests pass in the latest version of the code.</u>

Low severity issues

Low severity issues can influence project operation in future versions of code. We recommend taking them into account.

Code quality

- Consider declaring `public` functions as `external` where possible.

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

- If you plan to filter events, consider using `indexed` parameters for them.

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

- In **Custodian** contract, consider replacing `_authorized[addr] == false` condition with `!_authorized[addr]`.

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

- Avoid combining ternary operators into complex expressions as it significantly decreases code readability. Consider using standard `if-else` expression at line 53 of **Validator** contract.

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

- **Nominator** and **Validator** contracts have intersecting functionality. **NominatorOwner** and **ValidatorOwner** contracts have the same issue. Consider using inheritance to avoid code duplication.

Gas consumption (fixed)

In `withdrawTo()` function of **Custodian** contract, consider removing the check at line 22 since it increases gas consumption for users who perform valid transfers.

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

Incomplete documentation (fixed)

The provided documentation is incomplete and confusing. The purpose of Nominators and Validators is unclear. The interaction with back-end is poorly described. Also, the code is lacking NatSpecs.

As a result, it is sometimes unclear what the intention of the code is, and whether its behavior is correct, and the architecture of the project is appropriate.

We recommend adding NatSpecs to the code and providing a complete description of the project architecture and details of the implementation.

Additional documentation was provided for the project.

This analysis was performed by Pessimistic:

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

September 20, 2021