GYSR Security Analysis

by Pessimistic

This report is public.

Published: October 16, 2020

Abstract

In this report, we consider the security of smart contracts of the GYSR project. Our task is to find and describe security issues in 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 smart contracts of GYSR project. We performed our audit according to the procedure described below.

In the initial audit, we found a vulnerability that allowed an attacker to prevent users from receiving reward on their stake. We also found discrepancies with the documentation, a few issues that might result in vulnerabilities in further versions of the code, and several issues of low severity regarding code style and gas optimization.

After initial audit, both the whitepaper and the code were updated to the latest version. All the issues were fixed except 3 low severity issues regarding gas optimization: for those, the developers provided detailed comments.

General recommendations

We do not have any further recommendations.

Procedure

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

1. Whether code logic corresponds to the specification.
2. Whether the code is secure.
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.
  - We manually verify (reject or confirm) all the issues found by tools.
- Manual audit
  - We inspect the specification and check whether the logic of smart contracts is consistent with it.
  - 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.

Project overview

Project description

In our analysis we consider smart contracts of GYSR project on private GitHub repository, commit 0e6ecccf0ff38eb07a76e9545c059d1d2a2fc96b, GYSR whitepaper draft (gysr_wp_draft_20200929.pdf, sha1sum 81fea478925ee73415b3bf89bb3ada7ac531d70f), and audit notes (notes.pdf, sha1sum aa4c76e5e97adbc3e11122ce1177c9dbb7c45b77).

The total LOC of audited sources is 693.

Latest version of the code

After initial audit, the whitepaper and the code were updated. The code base was moved to a public GitHub repository. For the recheck, we were provided with a commit bef83833845d39a79f4a76914890c9e6885f810a and whitepaper (gysr_whitepaper_20201009.pdf, sha1sum d1236c1d07cacd260621f4df5e768dc5ffd18635).

Automated analysis

We used several publicly available automated Solidity analysis tools. All the issues found by tools were manually checked (rejected or confirmed). Cases when these issues lead to actual bugs or vulnerabilities are described in the next section.

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.

Discrepancies with documentation (fixed)

The following parts show inconsistencies between code and whitepaper:

- Section `7.3 Multiplier function` states that the function achieves `absolute limit of 9x given 10M total $GYSR supply`. But according to the given formula, which is implemented in the code correctly, the limit is 10x.
- Section `8.2 Geyser Management` states that all $GYSR tokens should be withdrawn at once. However, `withdraw()` function in **Geyser.sol** at lines 276–286 receives amount to withdraw as a parameter.

The issues have been fixed and are not present in the latest version of the code.

Code logic (fixed)

Function `stakeFor()` in **Geyser.sol** calls `update()` function, which performs bookkeeping for `msg.sender`, while `stakeFor()` function should update beneficiary's info. An attacker might call `stakeFor()` with minimal amounts for other users that will renew `user.lastUpdated` without any changes to `user.shareSeconds`, effectively "disabling" staking for those users.

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

Underflow (fixed)

There is an underflow in a for-loop in `preview()` function of **Geyser.sol** at line 710. It does not result in the vulnerability in the current implementation due to other checks inside the function. However, this approach is dangerous and can easily result in a vulnerability in further versions of the code.

The issues have been fixed and are not present in the latest version of the code.

Low severity issues

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

Code style (fixed)

- Solidity compiler version is not declared explicitly anywhere in the project. Moreover, `pragma solidity ^0.6.0` is used. Consider specifying certain Solidity version and optimization settings for the compiler.
- **IGeyserFactory** contract should be `interface`.
- `fund()` function in **Geyser** contract at lines 207–271 is excessively complicated. Consider implementing separate method for expired funding detection.
- Composing struct from pieces (**Geyser.sol**, lines 255–261) is error prone. Consider initializing it in-place (e.g. `Funding({field : value, ...})`), this improves code readability and optimizes gas consumption.
- Return value of `_unlockTokens()` function in **Geyser.sol** is never used.
- `_unstakeFirstInLastOut()` function in **Geyser.sol** at line 473 does not update `user.lastUpdated` explicitly. In the current version of the code it is safe as `_unstakeFirstInLastOut()` can only be called from `_unstake()` function, which, in time, calls `_update()` and thus, sets an appropriate value for `user.lastUpdated` field. However, this approach is error prone and might result in serious issues in further versions of the code.

The issues have been fixed and are not present in the latest version of the code.

Gas optimization

- Consider using `list.length` instead of `count` variable at line 57 of **GeyserFactory.sol**.

  The issue has been fixed and is not present in the latest version of the code.
- Consider using `immutable` for variables that are set via constructor and never changed.

  The issue has been fixed and is not present in the latest version of the code.
- Consider using smaller `uint`s in **Geyser.sol** at lines 35–56 to reduce storage footprint of structs.

  > *Comment from developers: This was tried and benchmarked. Unfortunately the increased gas cost of converting between uint64 and uint256 actually negated the benefit of reduced size. The gas cost was actually slightly worse. This was left as is.*

- Consider checking `allowance` in the beginning of the `_stake()` function of **Geyser.sol**.

> *<u>Comment from developers</u>: We wanted to avoid a redundant check with the ERC20 contract. Helps with saving on contract size. This is guided from the webapp to help prevent users from losing gas fees on failed transactions. This was left as is.*

- `User.shareSeconds` value is properly maintained but never used. Consider recalculating its value on-the-go in `view` functions if it is kept for informational purposes.

> *<u>Comment from developers</u>: It is true this value is not used aside from providing information. However due to the complexity that arises with many different stakes, it can become intractable to compute on-the-fly. This was left as is.*

This analysis was performed by Pessimistic:

Evgeny Marchenko, Senior Security Engineer

Boris Nikashin, Analyst

Alexander Seleznev, Founder

October 16, 2020