Paladin Security Analysis

by Pessimistic

This report is private

22 September, 2021

Abstract

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

The audit showed two issues of medium severity, including a Bug and an Overpowered role. Also, many issues of low severity were found.

The project has a documentation. The code base has comments and detailed NatSpecs and is covered with tests.

General recommendations

We recommend fixing the mentioned issues, clarifying the logic of the code, and 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 Paladin project on a private GitHub repository, commit 65597284f8491648c1bb94470b04506abe8084c7.

The project has README.md file and public documentation, the code has useful comments and detailed NatSpecs.

All tests pass, the code coverage is 94.02%.

The total LOC of audited sources is 2210.

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 the security of the project. They can lead to loss of funds or other catastrophic results. We do not recommend deploying the contracts until these issues are fixed.

The audit showed no critical issues.

Medium severity issues

Issues of medium severity can influence the project operation in the current implementation. This category of issues includes bugs, potential loss of income, incorrect system management, and other non-critical issues. We highly recommend addressing them.

Overpowered role

In **PaladinController** contract, an admin can:

- Withdraw all tokens from the reserve using `removeReserveFromPool()` function.
- Block pool transactions by providing an incorrect controller address in `setPoolsNewController()` function or by removing the pool with `removePool()` function call.

In **AddressRegistry** contract, an admin can font-run users' transactions and change the address of the pool using `_setPool()` function.

In **PalPool** contract, an admin can modify `interestModule`, `minBorrowLength`, and `killerRation` variables that affect the operation of the whole system.

In the current implementation, the system depends heavily on the admin role. Therefore, there are scenarios that can lead to undesirable consequences for the project and its users, e.g., if admin's private keys become compromised.

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

Bug

`removePool()` function of **PaladinController** contract emits an incorrect event at line 122. It should be `RemovePalPool` event instead of `NewPalPool`.

Low severity issues

Low severity issues do not directly affect project's operations. However, they might lead to various problems in the future versions of the code. We recommend taking them into account.

Gas consumption

- `burn()` function of **PalLoanToken** contract stores the history of token burning using **BurnedPalLoanToken** contract. Consider reconstructing the history off-chain using `BurnLoanToken` events. This will allow to optimize gas consumption by removing `burnedToken.mint` call at line 506.
- Consider saving `ownedTokens[owner]` value from storage to a local variable at lines 265, 291, 317, and 318 of **PalLoanToken** contract, since reading from storage on each for-loop iteration is expensive.
- In `allLoansOfForPool()` function of **PalLoanToken** contract, `burned[j.sub(tokenCount)]` variable can be read only once on each iteration of the loop at line 325.
- If a variable is not intended to change its value, consider declaring it as `immutable`:
  - In **PalLoan** contract, lines L27-35,39
  - **PalPoolStorage** contract, lines 54, 57, and 60.
  - **InterestCalculator** contract, lines 24–30.
  - **PalPoolStkAave** contract, lines 29 and 31.
- Consider replacing `preventReentry` modifier at lines 37–43 of **PalPool** contract with`nonReentrant` from **ReentrancyGuard** contract.
- In **PalPool** contract, consider removing duplicating checks at lines 129, 165, and 616.
- In **PalPool** contract, the amount check at line 764 is redundant, since the second includes this check in itself
- Avoid reading properties of storage variables on each iteration of `for`-loops since it requires a lot of gas. Instead, consider saving it to a local variable. It applies to:
  - **PaladinController** contract, lines 39, 109, and 279.
  - **GovernorMultiplier** contract, lines 63 and 78.
  - **AaveMultiplier** contract, lines 73 and 88.
  - **Doomsday** contract, lines 38, 102, and 266.
- Consider using minimal proxy instead of **PalLoan** contract`s functions.

Code quality

- In **SnapshotDelegator** contract, consider declaring an `immutable` variable and using it at lines 50, 75, 98, and 122.
- Delegators should implement one interface.
- Consider declaring **DelegateRegistry** contract as `interface`.
- In **PalPool** contract, the calculation of `_newBorrowIndex` variable at line 670 performs excessive operations. Instead of multiplying the expression by `1e18` and then dividing it by `1e36`, consider only dividing it by `1e18` without multiplying.
- Consider using `SafeMath` library when increasing allowance at line 121 of **PalToken** contract.
- In **PalToken** contract, checks that the value is not less than `amount` at lines 91 and 103 are redundant since they duplicate the check of `sub` function in **SafeMath** library.
- Consider declaring functions as `external` instead of `public` when possible.
- CEI (checks-effects-interactions) pattern is violated in multiple functions. We highly recommend following CEI pattern since it improves usage predictability.

Project design

The logic of fee calculation is split between interest module, delegators, and pools. It creates unnecessary coupling and complicates the code. As a result, current fee logic is error-prone and difficult to verify.

Project management

Standard code from `OpenZeppelin` library is copied into the project. Consider managing it as a dependency instead.

Notes

Fees logic

- In **PalPool** contract, when the borrowing fee is minimal and is spent enough to kill the borrow, a user can kill the loan before the end of the minimal non-penalized period.
- There are complex interactions between liquidity providers and borrowers. Consider a user borrowing tokens for three days. This three-days period is shorter than the minimum penalty-free period of 7 days. In this case, the utilization rate increases, so the penalty for the user also increases. Thus, there is no incentive for the user to return loan early, unless someone else deposits the tokens or returns their loan. This will slightly decrease utilization rate, and others might start closing their loans, so the utilization rate will continue decreasing. That means, there is a positive feedback loop for borrowers, which might destabilize the system. We recommend performing additional research to discover any potential tokenomics issues.

This analysis was performed by Pessimistic:

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

22 September, 2021