Paladin Security Analysis

by Pessimistic

This report is public

1 October, 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 initial audit showed a Bug and many issues of low severity.

The project has a documentation. The code base has detailed NatSpec comments and is covered with tests. The admin role has excessive powers.

After the initial audit, the code base was updated. Most of the issues were fixed in this update. Also, developers added new functionality to the project and improved tests coverage.

General recommendations

We recommend 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 85.95%.

The total LOC of audited sources is 2210.

Code base update

After the initial audit, the code base was updated. For the recheck, we were provided with commit 58834abee524794e5ac22d30c3f43ac0087526d4.

This update includes new implementation of loan contracts and fixes for most issues. Also, developers added new tests. The overall code coverage was improved and became 87.04%.

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.

Bug (fixed)

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

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's operations. However, they might lead to various problems in the future versions of the code. We recommend taking them into account.

Gas consumption

- 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.

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

- 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.

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

- If a variable is not intended to change its value, consider declaring it as `immutable`:
  - **InterestCalculator** contract, lines 24–30.
  - **PalPoolStkAave** contract, lines 29 and 31.

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

- Consider replacing `preventReentry` modifier at lines 37–43 of **PalPool** contract with`nonReentrant` from **ReentrancyGuard** contract.

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

- 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.

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

- Consider using minimal proxy instead of **PalLoan** contract`s functions.

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

- `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.

  > *Comment from developers: We want, for future use, that any id (currently active or burned) to be there. The Burned version allows us to track old Loans (that are still listed in the PalPool also), and know how was the last owner in case it was killed.*

- In **PalPool** contract, consider removing duplicating checks at lines 129, 165, and 616.

  > *Comment from developers: Even if SafeMath allows to prevent for the same issue than those requires, we want to keep them to return Error codes similar to the rest of the contract.*

- In **PalPool** contract, the amount check at line 764 is redundant, since the second includes this check in itself.

  > *Comment from developers: This check here was set to make sure that both the storage value tracking the Reserve, but also that the balance of the contract match the current state.*

Code quality

- In **SnapshotDelegator** contract, consider declaring an `immutable` or `constant` variable and using it at lines 50, 75, 98, and 122.

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

- Consider declaring **DelegateRegistry** contract as `interface`.

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

- 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.

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

- Consider using `SafeMath` library when increasing allowance at line 121 of **PalToken** contract.

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

- CEI (checks-effects-interactions) pattern is violated in multiple functions. We highly recommend following CEI pattern since it improves usage predictability.

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

- 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.

> *Comment from developers: We want those requires to get Error codes similar to the rest of the system.*

- Consider declaring functions as `external` instead of `public` when possible.

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.

> *Comment from developers: The way the fee logic is separated is set to be upgradable. Delegators have no fee calculation really, they only have calculation to match what was already calculated by the Pool, and transfer the right amounts of tokens to the right addresses. And the rest of the Fee logic is separated between the Pools and the Interest Calculator so all the Pools have access to the same logic for rates calculation, that can be upgraded if needed (the same way we went from the basic InterestCalculator to InterestCalculatorV2 during the testnet).*

Project management

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

Notes

Fees logic

There are complex interactions between liquidity providers and borrowers.
Consider a scenario, in which a user borrows tokens for the minimum period of seven days, while they only need these tokens for three days. If the utilization rate increases, they won't be able to return tokens early to partially recover fees due to early-closing penalties. Therefore, there is no incentive for the user to return loan early.

Rational actor would wait for someone else to deposit the tokens or return loans, which will decrease utilization rate. Others might start closing their loans, so the utilization rate will continue decreasing. In this case the best strategy for the said actor might still be to wait even logner to further reduce closing fee.

That scenario illustrates a positive feedback loop for borrowers, which might destabilize the system. We recommend performing additional research to discover any potential tokenomics issues.

> *Comment from developers: Of course, borrowers have a positive feedback when someone else loan is closed/killed, or when tokens are deposited. But this logic is simple demand/offer logic, and also brings the same negative feedback when other users want to borrow (since we expect the borrows to be around Governance votes, meaning most borrowers might borrow around the same period), or when tokens are withdrawn, forcing them either to add more fees into their loan, or to give up the voting power before the expected date.*

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.

> *Comment from developers: Multisig is used as admin.*

This analysis was performed by Pessimistic:

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

1 October, 2021