Paladin Dullahan Security Analysis

by Pessimistic

This report is public

April 10, 2023

Abstract

In this report, we consider the security of smart contracts of Paladin Dullahan 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 Paladin Dullahan smart contracts. We described the audit process in the section below.

Some of the project's tests on the Goerli network may fail because of the misconfiguration of Aave contracts on the Goerli testnet, which does not affect the security of the Dullahan protocol.

The initial audit showed several issues of medium severity: DoS prevents pod liquidations, Incorrect fee index initialization, Incorrect fee scaling, Inflation attack. Also, several low-severity issues were found.

After the initial audit, the codebase was updated. The developers fixed the issues of medium severity: DoS prevents pod liquidations, Incorrect fee index initialization, Incorrect fee scaling. They also provided comment for the Inflation attack issue of medium severity and fixed most of the low severity issues.

General recommendations

We recommend fixing the remaining issues. We also recommend implementing CI to run tests, and analyze code with security tools.

Project overview

Project description

For the audit, we were provided with Paladin Dullahan project on a private GitHub repository, commit ca106d90f30c5ca8c5bcb8a742049e26893228f9.

The scope of the audit included all contracts from the repository.

The documentation for the project included README and docs directory from the repository.

The project includes unit tests and Goerli network tests: 469 out of 477 pass successfully. The code coverage is 93.87%.

According to the documentation, the test suite for Goerli encounters some revert issues, coming from the Aave contracts, for 2 reasons:

- $stkAAVE on Goerli does not distribute rewards (which makes the part of the test for claiming and re-staking to fail).
- The Aave V3 market sometimes reverts when repaying the full debt / withdrawing the full balance.

The total LOC of audited sources is 2213.

Codebase update

After the initial audit, the codebase was updated. For the recheck, we were provided with commit fa1d08c91888388ec65539e00d222915a6ffc5e2.

This update included fixes or comments for most of the issues.

Audit process

We started the audit on March 16 and finished on March 31, 2023.

We inspected the materials provided for the audit. Then, we contacted the developers for an introduction to the project. After a discussion, we performed preliminary research and specified those parts of the code and logic that require additional attention during an audit:

- Aave V3 integration.
- Reward mechanism and its possible manipulation.
- Access control is implemented correctly.
- Gas consumption is optimized.

During the work, we stayed in touch with the developers and discussed confusing or suspicious parts of the code.

We manually analyzed all the contracts within the scope of the audit and checked their logic. Among other, we verified the following properties of the contracts:

- Liquidating a pod's GHO borrow position in Aave cannot disrupt the Dullahan protocol.
- Vault correctly implements the ERC4626 standard.
- The amount of rented $stkAave is limited to reserve a buffer for user withdrawals.
- Pod owners cannot repay their borrow position directly to Aave, bypassing Dullahan.
- One cannot use a GHO flashmint to exploit the protocol.
- Main functionality is restricted while contracts are not initialized.
- Contracts that hold $stkAave should be able to delegate their voting and proposal power to a Governance module.

We scanned the project with the static analyzer Slither and our private plugin with an extended set of rules and manually verified the output.

We ran tests and calculated the code coverage.

We combined in a private report all the verified issues we found during the manual audit or discovered by automated tools.

After the initial audit, we made the recheck on April 4-10. We checked fixes and comments for the issues of medium and low severities.

Finally, we updated 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 severity 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. DoS prevents pod liquidations (fixed)

The `liquidatePod()` function of the **DullahanPodManager** contract allows anyone to pay GHO in exchange for the collateral of a specific pod. During this process, the liquidator repays all fees accrued for $stkAave renting, and all $stkAave are returned from the pod back to the vault. When the current $stkAave balance of the pod is subtracted from `rentedAmount` on line 428:
`pods[pod].rentedAmount -= currentStkAaveBalance`, the call will revert if `currentStkAaveBalance > pods[pod].rentedAmount`. An attacker could send just 1 $stkAave to the pod once it is deployed, making future liquidations impossible.

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

M02. Incorrect fee index initialization (fixed)

In the **DullahanPodManager** contract `lastUpdatedIndex` defines a fee rate accrued since the last index update and should be equal to zero after the deployment of the contract. However, it is currently initialized to `block.timestamp` in the constructor, which results in incorrect fee calculation - pod owners will be charged less than they should.

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

M03. Incorrect fee scaling (fixed)

The `getCurrentFeePerSecond()` function of the **DullahanFeeModule** contract returns a fee rate, which increases if the utilization rate goes above a certain threshold. If `utilRate >= THRESHOLD`, the fee is calculated as `(currentFee * multiplier) / UNIT`, where the `multiplier` is scaled by 10^36 and `UNIT` equals 10^18. So the resulting fee is scaled by an extra 10^18, compared to the base `feePerStkAavePerSecond`, which is returned if utilization rate does not hit the threshold. This discrepancy will result in unreasonably high fees (x10^18) for pod owners if util rate exceeds the threshold.

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

M04. Inflation attack (resolved)

The `totalAssets()` of the **DullahanVault** contract relies on the current $stkAave balance of the vault. This is why the amount of minted shares is inversely related to the balance of the vault. An attacker could front-run a deposit from a user with a direct $stkAave transfer to the vault, deflating the index and decreasing the amount of shares minted for the user. The index is easy to manipulate if the initial deposit is shallow.

The **DullahanRewardsStaking** contract is vulnerable to the same kind of attack. We recommend using the internal accounting instead of the ERC20 balance in `totalAssets()` function in both contracts.

> *Comment from the developers:*
>
> *Dullahan Vault: In case of the Vault, the shares are rebasing tokens, meaning any depositor will receive the same exact amount of dstkAAVE that the stkAAVE deposited. In case of a direct transfer to the Vault be an attacker before a user deposit, it will not impact the amount of dstkAAVE minted for the depositor. Instead, it will increase all previous depositors dstkAAVE balances (the same way it is increased when the Vault claims AAVE rewards and stake them for stkAAVE)*
>
> *Dullahan Rewards Staking: In the case of the Staking contract, this type of inflation attack would work only if the depositor was the 1st to deposit, and the contract was empty. This is why an initial deposit is done during the initialization, to prevent from such attacks. After, if an attacker sends tokens directly to the contract right before an user deposit, the funds will be absorbed in the index (increasing all previous depositors position), but for the user depositing, the amount of shares minted will end up representing their share of the total funds in the contract correctly.*

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. Redundant if-case (fixed)

In the **DullahanPod** contract the parameter `amount` of the `liquidateCollateral()` function is checked to be equal to `type(uint256).max`. As this function can only be called from the **DullahanPodManager** contract, which does not pass a `type(uint256).max` value, this if-case is redundant.

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

L02. Unused constants (fixed)

The following constants are declared but never used:

- `UNIT` constant at line 35 in the **DullahanPod** contract;
- `MAX_BPS` constant in the **DullahanFeeModule**.

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

L03. Misleading comments (fixed)

The descriptions of `lastUpdatedIndex` и `lastIndexUpdate` variables are mixed up at lines 101, 103 in the **DullahanPodManager** contract.

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

L05. Dependency management

`OpenZeppelin` contracts are copied into the project, but should be managed as a dependency.

L06. Immutable (fixed)

`protocolFeeChest` variable **DullahanPodManager** contract is set during contract deployment and never change later. Consider declaring it as immutable to reduce gas consumption.

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

L07. The rules of OpenZeppelin hooks (fixed)

`_beforeTokenTransfer` should be virtual and call super inside due to OpenZeppelin hooks documentation at line 660 in **DullahanVault** contract.

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

L08. Gas consumption

In the cases below there are multiple readings from the storage. To reduce gas consumption, consider reading storage variable to the local variable once and then using this variable.

- **DullahanPodManager**:
  - (fixed) `_pod.collateral` variable at lines 286, 288, 300, 441, 443, 455, 470;
  - `extraLiquidationRatio` variable at lines 290, 301, 445, 456;
- `podManagers[manager].totalRented` at line 524, 527 in the **DullahanVault** contract;
- (fixed) `collateral` at lines 233, 240, 242 in the **DullahanPod** contract.

The issues have been partially fixed.

L09. Checks-Effects-Interactions

The CEI (checks-effects-interactions) pattern is violated in the contracts below since storage variables are updated after external calls. We highly recommend following CEI pattern to increase the predictability of the code execution and protect from some types of re-entrancy attacks.

- **DullahanVault**:
  - `reserveAmount` variable at lines 842, 863 is changed after `_getStkAaveRewards()` function;
  - storage update at line 581 in `_mint` function after `_getStkAaveRewards` and `safeTransferFrom`;
  - `podManagers[manager].totalRented` and `totalRentedAmount` variables update after `_getStkAaveRewards()` function at lines 536-537;
  - `reserveAmount` at line 692;
- `userScaledBalances[receiver]`, `totalScaledAmount` variables at lines 319, 320 in the **DullahanRewardStaking** contract;
- **DullahanPodManager**:
  - (fixed) `_pod.accruedFees` and `reserveAmount` at lines 464-465;
  - `pods[pod].rentedAmount` at line 395.

The issues have been partially fixed.

Notes

N01. Unclear buffer amount calculation (commented)

According to the documentation, the vault's buffer amount is $stkAave, reserved for user withdrawals. It is calculated as a certain percentage (`bufferRatio`) of `totalAssets()`, which includes the actual balance of the vault and `totalRentedAmount` - the total amount of $stkAave, rented by pods. `totalRentedAmount` does not add to the solvency of the protocol and is controlled by the GHO debt of the pods, so it should not be used to predict the amount reserved for user withdrawals. Also `totalAssets()` does not include the `reserveAmount`, which reflects the amount of $stkAave, deposited by the admin or accrued from staking $AAVE to the Safety Module.

> *Comment from the developers: The bufferRatio is applied on user funds held by the Vault, and not protocol owned part of the Vault (represented by reserveAmount), which is why the reserveAmount is not included in the bugger calculations. The totalRentedAmount is included, even if it is not current protocol solvency (as it is rented), it is users owned assets, which need to be part of the buffer calculations.*

This analysis was performed by Pessimistic:

Daria Korepanova, Security Engineer
Vladimir Pomogalov, Security Engineer
Ivan Gladkikh, Security Engineer
Irina Vikhareva, Project Manager

April 10, 2023