Krystal Security Analysis

by Pessimistic

This report is public.

Published: March 11, 2021

Abstract....2
Disclaimer....2
Summary....2
General recommendations....2
Procedure....3
Project overview....4
Project description....4
Latest version of the code....4
Manual analysis....5
Critical issues....5
Medium severity issues....6
Bugs (fixed)....6
Unchecked response (fixed)....7
Incorrect protocol usage (fixed)....7
Incorrect debt calculation (new)....7
Insufficient tests coverage....7
Discrepancy with documentation (fixed)....8
Low severity issues....9
Code quality....9
Code logic (fixed)....10
Gas consumption....10

Abstract

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

The initial audit showed several issues of medium severity, including four Bugs, Unchecked response, Incorrect protocol usage, Discrepancy with the documentation, and others; several issues of low severity.

The project has the documentation and tests. However, tests coverage is insufficient.

After the initial audit, the code base was updated to the latest version. Bugs, Unchecked response, Incorrect protocol usage, Discrepancy with the documentation, and several other issues were fixed. The rest of the issues were commented. However, two new issues appeared in the code, including Incorrect debt calculation. We discussed them with developers.

General recommendations

We recommend fixing the mentioned issues and improving tests coverage.

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.

Project overview

Project description

For the audit, we were provided with Krystal project on GitHub repository, commit 7e2ad3326fd5a2ee6e78a89b01c5f09720f6611c.

The project has a documentation on Google Docs, sha1sum of downloaded txt file is 6f4940929dc6926bc997936c5ac3582fd487e269.

The total LOC of audited sources is 2032.

Latest version of the code

After the initial audit, the code base was updated. For the recheck, we were provided with commit 2f2f7cff3d5bbd9fae891c52d3ccde3589b44b16.

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.

Bugs (fixed)

- In **SmartWalletLending** contract, `repayBorrowTo()` function approves allowance for `aaveLendingPool.lendingPoolV1` at line 299.

  However, according to Aave documentation, allowance should be approved for `aaveLendingPool.lendingPoolCoreV1`.

- In **BurnGasHelper** contract, `getAmountGasTokensToBurn()` function calculates `gasSpent` value incorrectly. It should be calculated this way:

  ```
  uint256 gasSpent = 21000 + 16 * msg.data.length + gasConsumption;
  ```

  Consider reading this post for more information.

- In **SmartWalletSwapImplementation** contract, `claimPlatformFees()` function does not reset `platformWalletFees` value after `transferToken()` function is called. Therefore, anyone can call `claimPlatformFees()` function multiple times. This can affect the distribution of fees between `plaftformWallets` addresses.

  Consider resetting `platformWalletFees` value after `transferToken()` call.

- In **FetchAaveDataWrapper** contract, `getSingleUserReserveDataV1()` function calculates `data.poolShareInPrecision` value incorrectly at line 213:

  ```
  data.poolShareInPrecision = aToken.balanceOf(_user) / totalSupply;
  ```

  In this case, the calculated value of `data.poolShareInPrecision` will always equal to `0` since `aToken.balanceOf(_user) < aToken.totalSupply()`.

  The same situation occurs at line 244 in `getSingleUserReserveDataV2()` function.

- During the audit, the developers discovered a bug in **SmartWalletLending** contract that caused revert in most cases of `repayBorrowTo()` call due to usage of `amount` value instead of `payAmount`.

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

Unchecked response (fixed)

In **SmartWalletLending** contract, `repayBorrowTo()` function does not check the returned value of `ICompErc20(cToken).repayBorrowBehalf(...)` call at line 323. However, Compound does not fail in case of an error, but returns an error code. Thus, the returned value must be checked.

The returned value of `poolV2.repay()` call at lines 310 and 313 should also be checked.

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

Incorrect protocol usage (fixed)

`ProtocolDataProvider.getReserveData()` function returns `availableLiquidity` as its first return value. However, in **FetchAaveDataWrapper** contract it is used as `totalLiquidity` at line 101.

As a result, `reservesData[i].utilizationRate` value at line 115 will always be equal to `0`.

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

Incorrect debt calculation (new)

In **SmartWalletLending** contract, in order to obtain user's debt, `getUserDebt()` function calls `cToken.borrowBalanceStored()`. However, `borrowBalanceStored()` reads the value from storage and returns the debt size at the moment of last interaction. Therefore, `swapKyberAndRepay()` and `swapUniswapAndRepay()` functions do not repay user's debt at Compound completely.

To get the actual debt value, use `cToken.borrowBalanceCurrent()` method that accrues interest internally.

The developers are aware of this issue and will address it soon.

Insufficient tests coverage

The project has tests. However, the coverage is insufficient. Also, the coverage itself is not calculated in the project.

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

We highly recommend checking and improving test coverage regularly as the project grows.

> *Comment from developers: More tests have been added since the last commit but will continue to improve coverage.*

Discrepancy with documentation (fixed)

The documentation states:

```
Users will only pay platform fees for swap transactions.
```

However, in **SmartWalletSwapImplementation** contract, `safeForwardTokenAndCollectFee()` function charges fee depending on `platformFeeBps` parameter of the transaction, though this function can only be called for non-swap interactions.

We recommend updating the documentation or modifying this behavior of the code to match the documentation.

The issue has been fixed and is 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 quality

- In **SmartWalletLending** contract, `depositTo()` and `repayBorrowTo()` functions support ether deposits. However, these functions are not payable and are never used to deposit ether.

  > *Comment from developers: ETH or tokens are already transferred to the **SmartWalletLending** contract via the swap that happens beforehand, and where the destination of that swapped amounts is the **SmartWalletLending** contract.*

- Consider splitting **SmartWalletLending** contract into three (`V1`, `V2`, and `Compound`) that implement **ISmartWalletLending** interface.

  > *Comment from developers: We looked into splitting the contracts, however we will use the current structure for now.*

- Consider splitting **FetchAaveDataWrapper** contract into two (`V1` and `V2`) that implement **IFetchAaveDataWrapper** interface.

  > *Comment from developers: We looked into splitting the contracts, however we will use the current structure for now.*

- Starting from Solidity `0.6.8`, `type(typeName).max` syntax can be used to get the maximum value of type `typeName`. Consider using it in **SmartWalletSwapStorage** contract at line 16 instead of `uint256(-1)`.

  > *Comment from developers: As our SmartWallet contracts are dependent on contracts using Solidity 0.6.6 (such as our @kyber.network/utils-sc contract package), the current implementation is acceptable.*

- **SmartWalletSwapStorage** contract keeps state variables for both swap implementation and proxy logic. Consider implementing eip-1967 pattern or using OZ UpgradeableProxy to avoid this coupling.

  To separate proxy and implementation logic, we recommend storing `admin` and `implementation` values in special slots in **Proxy** and use `Logic.initialize()` to setup it. Only the **Logic** contract should inherit from **Storage**. This helps to prevent storage corruption and to avoid issues during updates.

  Also, consider replacing hexadecimal string with the following expression:

  ```
  bytes32(uint256(keccak256("SmartWalletSwapImplementation")) - 1)
  ```

  when initializing `IMPLEMENTATION` constant. This will improve code readability.

- In **SmartWalletSwapImplementation** contract, events `UpdateKyberProxy` and `UpdateUniswapRouters` declared at lines 18–22 are never used.

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

- The NatSpec for `withdrawFromLendingPlatform()` function is missing description for `minReturn` parameter in **SmartWalletSwapImplementation** contract.

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

Code logic (fixed)

- In **SmartWalletLending** contract, `safeApproveAllowance()` function should not be called with `depositTo()` and `repayBorrowTo()` functions as max `uint256` value can be used as infinity for allowance approve. Consider moving the call of `safeApproveAllowance()` function to `updateAaveLendingPoolData()` and `updateCompoundData()` functions. This will also optimize gas consumption.
- In **SmartWalletSwapImplementation** contract, events parameters at lines 15–16, 24, 26–27, and 31–32 should not be `indexed`.

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

Gas consumption

- In **SmartWalletLending** contract, `ReentrancyGuard` functionality is not used.

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

- Consider passing `msg.data.length` instead of `msg.data` to `BurnGasHelper` in `burnGasTokensAfter()` function of **SmartWalletSwapImplementation** contract. Alternatively, a precalculated cost of the transaction, execution, and `msg.data` can be passed to `burnGasHelper.getAmountGasTokensToBurn()`.

  *The method was updated. However, new implementation uses `msg.data.length` of inter-contract call to `BurnGasHelper.getAmountGasTokensToBurn()` which is always equal to `4 + 32 = 36` and does not depend on `msg.data` of the initial **SmartWalletSwap** call.*

  > *Comment from developers: this only affects up to one gas token (lower than optimal one, better than burn more than needed).*

- `swapKyberAndRepay()` function of **SmartWalletSwapImplementation** contract calculates `payAmount` value twice.

This analysis was performed by Pessimistic:

Evgeny Marchenko, Senior Security Engineer
Vladimir Tarasov, Junior Security Engineer
Boris Nikashin, Analyst
Alexander Seleznev, Founder

March 11, 2021