SMART CONTRACT AUDIT REPORT **30-05-2022**

By: REWARD CYCLE

Introduction

This audit report highlights the overall security of the DEfi King and their token. With this report, I have tried to ensure the reliability of the smart contract by completing the assessment of their system's architecture and smart contract codebase.

Auditing approach and Methodologies applied

In this audit, I consider the following crucial features of the code.

- Whether the implementation of protocol standards.
- Whether the code is secure.
- Whether the code meets the best coding practices.
- Whether the code meets the SWC Registry issue.

The audit has been performed according to the following procedure:

- Manual audit

    - Inspecting the code line by line and revert the initial algorithms of the protocol and then compare them with the specification
    - Manually analyzing the code for security vulnerabilities.
    - Assessing the overall project structure, complexity & quality.
    - Unit testing by writing custom unit testing for each function.
    - Checking whether all the libraries used in the code of the latest version.
    - Analysis of security on-chain data.
    - Analysis of the failure preparations to check how the smart contract performs in case of bugs and vulnerability.

- Automated analysis
- Manually verifying (reject or confirm) all the issues found by tools.
- Performing Unit testing.
- Manual Security Testing (SWC-Registry, Overflow)
- Running the tests and checking their coverage.

Report: All the gathered information is described in this report.

Audit details

Project Name: Defi Kings

Token symbol: DFK

Contract Address:

0x8956692426786F16CF96922181553ef2d308de5C

Token Supply: 1,000,000,000,000 **DFK**

Language: Solidity

Platform and tools: Remix, VScode, securify and other tools mentioned in the automated analysis section.

Audit Goals

The focus of this audit was to verify whether the smart contract is secure, resilient, and working properly according to the specs. The audit activity can be grouped in three categories.

Security: Identifying the security-related issue within each contract and system of contracts.

Sound architecture: Evaluating the architect of a system through the lens of established smart contract best practice and general software practice.

Code correctness and quality: A full review of contract source code. The primary area of focus includes.

- Correctness.
- Section of code with high complexity.
- Readability.
- Quantity and quality of test coverage.

Security

Every issue in this report was assigned a severity level from the following:

High severity issues

Issues mentioned here are critical to smart contract performance and functionality and should be fixed before moving to main net.

Medium severity issues

This could potentially bring the problem in the future and should be fixed.

Low severity issues

These are minor details and warnings that can remain unfixed but would be better if it got fixed in the future.

No. of issue per severity

Severity	High	Medium	Low
Open	0	0	6

Manual audit

Following are the reports from our manual analysis.

High severity issues

No High Severity Issue found.

Medium severity issues

No Medium Severity Issue found.

Low Severity Issues :

There were 6 low severity issues found in protocol and token contract.

- Unsafe Assumptions About Average Time Between Blocks

The current implementation of the protocol uses *blocks* rather than *seconds* to measure the time between interest accruals. This makes the implementation highly sensitive to changes in the average time between Ethereum blocks.

For example, the average time between blocks may increase by significant factors due to the difficulty bomb or decrease by significant factors during the transition to Serenity. The difference between the actual time between blocks and the assumed time between blocks causes proportional differences between the intended interest rates and the actual interest rates.

While the admin can combat this by adjusting the interest rate model when the average time between blocks changes, such adjustments are manual and happen only after-the-fact. Errors in blocktime assumptions are cumulative, and fixing the model after-the-fact does not make users whole – it only prevents incorrect interest calculations moving forward (until the next change in blocktime).

Consider refactoring the implementation to use *seconds* rather than *blocks* to measure the time between accruals. While block.

Timestamp can be manipulated by miners within a narrow window, these errors are small and, importantly, are not cumulative. This would decouple the interest rate model from Ethereum's average block time.

- Require Statement Without Error Message

Consider adding a message to inform users in case of a revert. This msg will be displayed during failed operation.

- 0 Address for Mints & Burns

Although not technically part of the EIP20 specification, it is common practice to use the zero address as the source for all Transfer events after minting, and as the destination for Transfers upon burning tokens

- Use assert

This is confirming a property that should never fail for any user input. In such a situation, consider using an **assert** statement instead as part of the right software practice.

In Token DefiKing.sol

- Costly loop [line 209]

Binance is a very resource-constrained environment. Prices per computational step are orders of magnitude higher than with centralized providers. Moreover, Binance miners impose a limit on the total number of gas consumed in a block. If **array.length** is large enough, the function exceeds the block gas limit, and transactions calling it will never be confirmed:

```
for (uint256 i = 0; i < array.length ; i++) { cosltyFunc();
}
```

This becomes a security issue, if an external actor influences array.length. E.g., if the array enumerates all registered

addresses, an adversary can register many addresses, causing the problem described above.

In Solidity, functions that do not read from the state or modify it can be declared as pure.

Recommendation**:**

Do not declare functions that read from or modify the state as pure. The following statements are considered modifying the state:

- Writing to state variables
- Emitting events;
- Creating other contracts;
- Using selfdestruct;
- Sending Ether via calls;
- Calling any function not marked view or pure;
- Using low-level calls;
- Using inline assembly that contains certain opcodes.

The following statements are considered reading from the state:

- Reading from state variables;
- Accessing this.balance or <adress>.balance;
- Accessing any of the members of block, tx, msg (with the exception of msg.sig and msg.data);
- Calling any function not marked pure;
- Using inline assembly that contains certain opcodes

Automated test :

We have used multiple automated testing frameworks. This makes code more secure common attacks. The results are below.

Smart Check:

SmartCheck automatically checks Smart Contracts for vulnerabilities and bad practices. Automated tests have been conducted and got the following report. F

- Hardcoded Address:

The contract contains unknown address. This address might be used for some malicious activity. Please check hardcoded address and it's usage.

Recommendation:

It is required to check the address. Also, it is required to check the code of the called contract for vulnerabilities.

The length of the dynamic array is changed directly. In this case, the appearance of gigantic arrays is possible and it can lead to a storage overlap attack (collisions with other data in storage).

Recommendation

If possible, avoid changing the length of the dynamic array directly.

  - Use uint[] storage arrayName = new uint[](7) to create a dynamic array of the desired length.
  - Use delete arrayName to clear a dynamic array.
  - Use .push() (instead of .length++) to write to the end of the dynamic array.
  - Starting with version 0.5.0 of the Solidity compiler, use .pop() (instead of

.length--) to delete the last element of the dynamic array.

- Multiplication after division

Solidity operates only with integers. Thus, if the division is done before the multiplication, the rounding errors can increase dramatically.

Recommendation:

Multiplication before division may increase the rounding precision.

Recommendation:

Only use the approve function of the ERC-20 standard to change the allowed amount to 0 or from 0 (wait till transaction is mined and approved).

- Costly Loop

Ethereum is a very resource-constrained environment. Prices per computational step are orders of magnitude higher than with centralized providers. Moreover, Ethereum miners impose a limit on the total number of gas consumed in a block. If array.length is large enough, the function exceeds the block gas limit, and transactions calling it will never be confirmed:

for (uint256 i = 0; i < array.length ; i++) { cosltyFunc();

}

This becomes a security issue, if an external actor influences array.length.

Recommendation:

Avoid loops with a big or unknown number of steps.

- Locked money
- contracts/Unitroller.sol [Line 10-148]

Contracts programmed to receive ether should implement a way to withdraw it, i.e., call transfer (recommended), send, or call.value at least once.

Recommendation

Implement a withdraw function or reject payments (contracts without a fallback function do it automatically).

- msg.value == 0 check

The msg.value == 0 condition check is meaningless in most cases.

Recommendation:

Avoid meaningless checks.

- Overpowered role

This function is callable only from one address. Therefore, the system depends heavily on this address. In this case, there are scenarios that may lead to undesirable consequences for investors, e.g., if the private key of this address becomes compromised.

Recommendation

We recommend designing contracts in a trustless manner. For instance, this functionality can be implemented in the contract's constructor. Another option is to use MultiSig wallet at this address.

It is recommended to follow the latter example, as future compiler versions may handle certain language constructions in a way the developer did not foresee.

Recommendation:

Specify the exact compiler version (pragma solidity x.y.z;).

The detailed report of all above and other errors/notes can be found at this link.

Note

The repo given for audit contains both protocol and token contract.

The token contract was written in a standard format. There was no specific function to test recommended in the test case code written on the repo.

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 the code. Besides, a security audit,

please don’t consider this report as investment advice.

Summary

The use of smart contracts is simple and the code is relatively small. Altogether the code is written and demonstrates effective use of abstraction, separation of concern, and modularity. But there are a few issues/vulnerabilities to be tackled at various security levels, it is recommended to fix them before deploying the contract on the main network. Given the subjective nature of some assessments, it will be up to the Amplify-protocol team to decide whether any changes should be made.