BlockÂpex

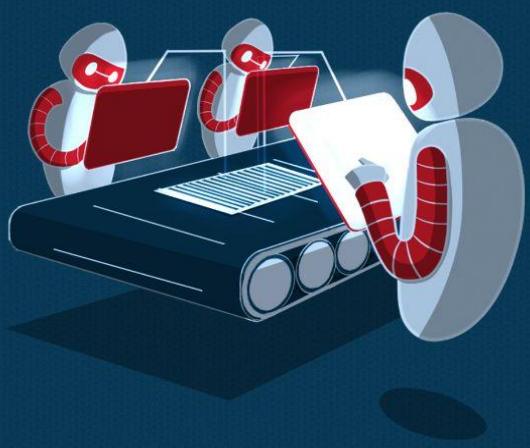
SMART CONTRACT SECURITY ANALYSIS REPORT

```
pragma solidity 0.7.0;
contract Contract {

    function hello() public returns (string) {
        return "Hello World!";
    }

    function findVulnerability() public returns (string) {
        return "Finding Vulnerability";
    }

    function solveVulnerability() public returns (string) {
        return "Solve Vulnerability";
    }
}
```



PREFACE

Objectives

This document aims to highlight any identified bugs/issues in the provided codebase. This audit has been conducted in a closed and secure environment, free from influence or bias of any sort. This document may contain confidential information about IT systems/architecture and the client's intellectual property. It also includes information on potential risks and the processes involved in mitigating/exploiting the risks mentioned below. The usage of the information provided in this report is limited, internally, to the client. However, this report can be disclosed publicly to aid our growing blockchain community; at the client's discretion.

Key understandings

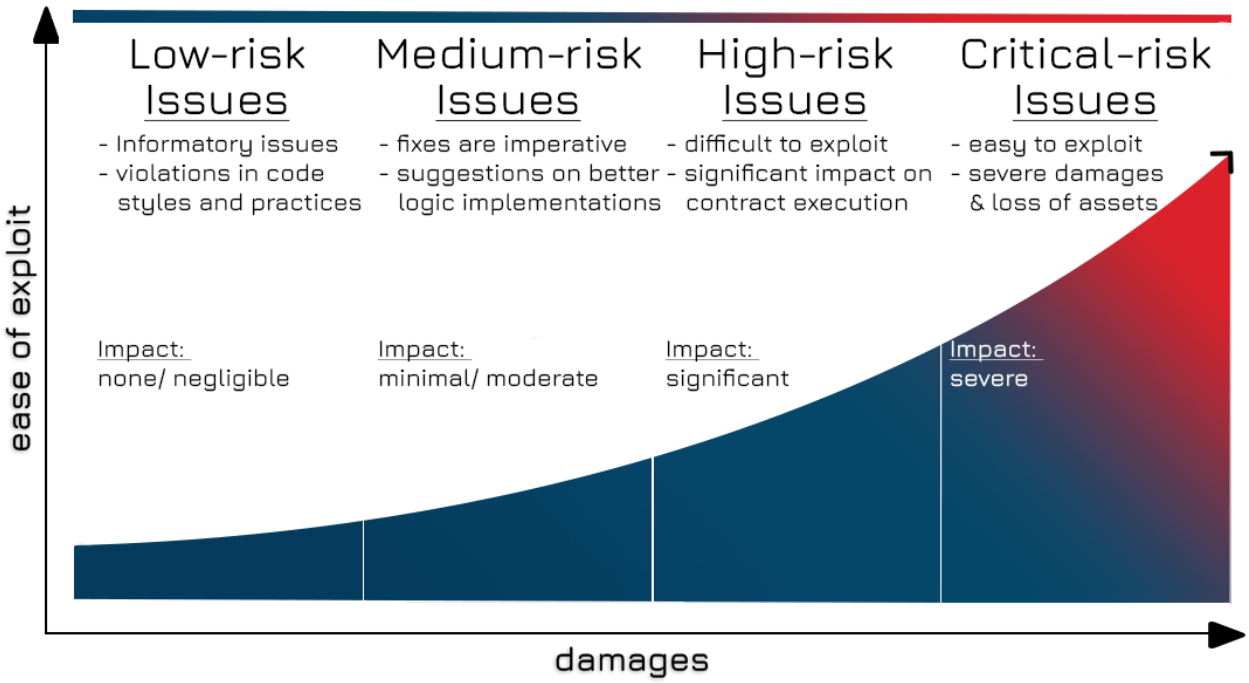


TABLE OF CONTENTS

INTRODUCTION

BlockApex (Auditor) was contracted by DeFiGeek Community (Client) for the purpose of conducting a Smart Contract Audit/ Code Review. This document presents the findings of our analysis, which started on 11th July '2023

Name
Yamato Protocol
Auditors
Moazzam Arif | Muhammad Jarir Uddin
Platform
Ethereum and EVM Compatible Chains | Solidity
Type of review
Manual Code Review | Automated Tools Analysis
Methods
Architecture Review | Functional Testing | Computer-Aided Verification | Manual Review
Git repository/ Commit Hash
https://github.com/yamatoprotocol/core | 8dec52f981856c7e1c478b609bf39e7b818cfc53
White paper/ Documentation
Protocol Overview
Document log
Initial Audit Completed: July 17th '2023
Final Audit (Fixed): July 24th '2023 - Commit Hash

Scope

The shared git-repository was checked for common code violations and vulnerability-specific probing to detect major issues/vulnerabilities. Some specific checks are as follows:

Code review		Functional review
Reentrancy	Unchecked external call	Business Logics Review
Ownership Takeover	Fungible token violations	Functionality Checks
Timestamp Dependence	Unchecked math	Access Control & Authorization
Gas Limit and Loops	Unsafe type inference	Escrow manipulation
DoS with (Unexpected) Throw	Implicit visibility level	Token Supply manipulation
DoS with Block Gas Limit	Deployment Consistency	Asset's integrity
Transaction-Ordering Dependence	Repository Consistency	User Balances manipulation
Style guide violation	Data Consistency	Kill-Switch Mechanism
Costly Loop		Operation Trails & Event Generation

Project Overview

DeFiGeek Community (DFGC) is an open community developing DeFi DApps and middlewares contributing to the Web3 era. DFGC develops and envisions multiple DApps. The first is the Yamato Protocol, a crypto-secured stablecoin generator DApp pegged to JPY. Yamato Protocol is a lending decentralized financial application (DeFi) that can generate the Japanese Yen stablecoin "CJPY". DeFiGeek Community Japan, a decentralized autonomous organization, is developing it.

Yamato Protocol is a crypto-asset overcollateralized stablecoin issuance protocol. V1 allows the issuance of CJPY (Japanese Yen equivalent coin) using ETH as collateral. It has the following features:

- No immediate forced liquidation
- No obligation to repay debt
- No interest rates (lump-sum fees)
- Low secured rates (minimum 130%)

NOTE: The current audit scope covers a smart contract consisting of Yamato PriceFeedV3, which is a feature update from the previously audited version of the Yamato protocol.

Yamato PriceFeedV3

Yamato PriceFeedV3 is a version update from its predecessor PriceFeedV2 where the main update is the removal of a secondary (backup) oracle to fetch the prices. The latest PriceFeed smart contract now relies on Chainlink as its sole provider of the latest prices to determine the flow of the DeFi protocol.

System Architecture

The idea for DeFiGeek Community's Yamato Protocol is based on Maker and Liquidity protocols—crypto over-collateralized stablecoin render. Token Economics implements the Voting Escrow model with reference to Curve's ecosystem of CRV and veCRV (This will be implemented in a later version).

The initial model will be launched on Ethereum, and by depositing ETH, you can borrow a Japanese Yen stablecoin called CJPY (ERC-20).

As a P2C (Peer to Contract) Lender, it is an epoch-making design that eliminates the need for forced clearing. It is a basic application of web3 as a decentralized application that can borrow Japanese Yen equivalent coins (Convertible JPY (ticker: CJPY)) with ETH as collateral. In V2, DFGC will implement CUSD and CEUR and further develop as a global stablecoin render.

The smart contract feature update for PriceFeedV3 is dependent on a few interfaces that define the architecture of the smart contract. The main contract PriceFeedV3 offers Chainlink as its sole price oracle to keep track of the latest prices within the Blockchain ecosystem.

Yamato PriceFeedV3:

The contract removes a set of functionalities along with the attached dependencies to ensure that the new architecture is based only on Chainlink. The contract PriceFeedV3 imports the following interfaces and manages the relevant data structures that support the feature update in the smart contract;

- ./Interfaces/IPriceFeedV3.sol
- ./Interfaces/IPriceFeedFlexV3.sol

Along with the above relevant interfaces, the feature update also covers a couple of other interfaces, namely IPriorityRegistryFlexV6.sol and IPriorityRegistryV6.sol that introduce a data structure to support the previous functionalities.

Methodology & Scope

The codebase was audited using a filtered audit technique. A band of two (2) auditors scanned the codebase in an iterative process for a time spanning six (6) days.

Starting with the recon phase, a basic understanding was developed, and the auditors worked on developing presumptions for the developed codebase and the relevant documentation/whitepaper. Furthermore, the audit moved on with the manual code reviews to find logical flaws in the codebase complemented with code optimizations, software, and security design patterns, code styles, best practices, and identifying false positives detected by automated analysis tools.

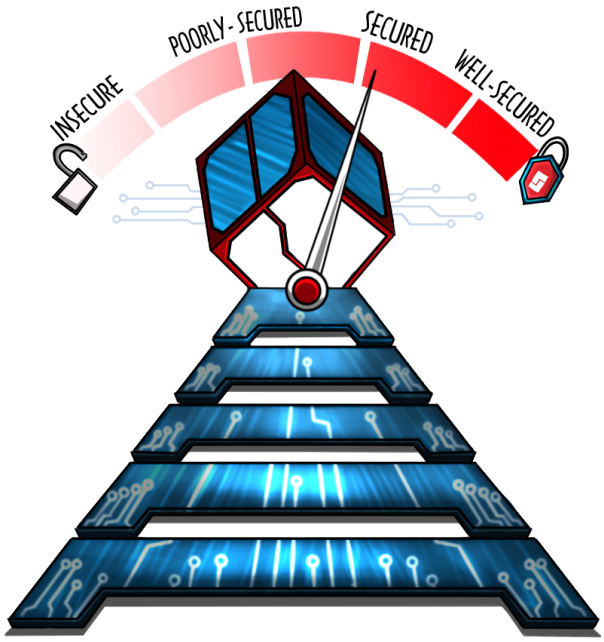
AUDIT REPORT

Executive Summary

Our team performed a technique called *Filtered Audit*, where two individuals separately audited the Yamato Protocol (Feature Update).

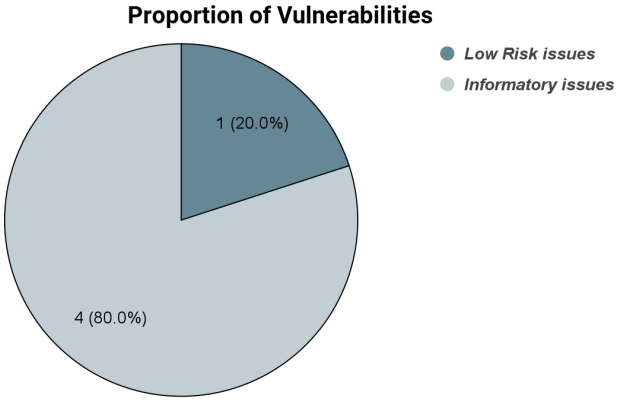
After a thorough and rigorous manual testing process involving line-by-line code review for bugs, an automated tool-based review was carried out using Slither for static analysis and Foundry for fuzzing invariants.

All the flags raised were manually reviewed and re-tested to identify the false positives.



Our team found:

#Issues	Severity Level
0	High-Risk issue(s)
0	Medium-Risk issue(s)
1	Low-Risk issue(s)
4	Informatory issue(s)



Key Findings

#	Findings	Risk	Status
1	Existence of Unused Function	Low	Fixed
2	Absence of Dev Comments in Interfaces	Info	Fixed
3	Unnecessary Wrapper Function	Info	Fixed
4	Commented Code in fetchPrice Function	Info	Fixed
5	Presence of Commented State Variable	Info	Fixed

Detailed Overview

Low-risk issues

ID	1
Title	Existence of Unused Function
Path	contracts/PriceFeedV3.sol
Severity	Low
Status	Fixed
Function Name	_changeStatus

Description:

The function _changeStatus was identified in the codebase. This function, which was previously utilized in the fetchPrice function, has been replaced by a newly introduced function, _storePrice. The existence of unused functions can lead to unnecessary code complexity and potential security risks.

Code-Affected:

```
function _changeStatus(Status _status) internal {
  if (lastSeen == block.number) return;
  if (status == _status) return;
  status = _status;
  emit PriceFeedStatusChanged(_status);
}
```

Impact:

Unused functions increase the complexity of the codebase, making it harder to read and maintain. They can also pose potential security risks if they contain overlooked vulnerabilities because the function is not in use.

Recommendation:

To ensure optimal code efficiency and security, removing the unused function _changeStatus from the codebase is advised.

Informatory Issues and Optimizations

ID	2
Title	Absence of Dev Comments in Interfaces
Path	contracts/Interfaces/IPriceFeedV3.sol
Severity	Info
Status	Fixed
Function Name	-

Description:

The interfaces in the codebase lack developer comments. Proper NatSpec comments are considered best practice, providing valuable context and understanding for developers and auditors reviewing the code.

Code-Affected:

```
pragma solidity 0.8.4;
interface IPriceFeedV3 {
  function fetchPrice() external returns (uint256);
  function getPrice() external view returns (uint256);
  function lastGoodPrice() external view returns (uint256);
}
```

Impact:

The absence of developer comments in interfaces can make the code harder to understand for other developers or auditors, increasing the likelihood of misunderstandings and errors. It can also make the codebase harder to maintain and extend in the future.

Recommendation:

To enhance code readability and maintainability, it is recommended to add appropriate NatSpec comments to the interfaces in the codebase.

ID	3
Title	Unnecessary Wrapper Function
Path	contracts/PriceFeedV3.sol
Severity	Info
Status	Fixed
Function Name	_chainlinkIsBroken

Description:

There is a function called `_chainlinkIsBroken`, which is a wrapper function. The visibility of this function is internal; this function calls another internal function `_badChainlinkResponse`, which seems unnecessary. Since, the feature is now updated to handle only one price oracle, i.e. Chainlink, the internal functions are not called in differing cases throughout the smart contract, and therefore, leaving the function hierarchy to more depth is not recommended.

Code-Affected:

```
function _chainlinkIsBroken(
  ChainlinkResponse memory _currentResponse
) internal view returns (bool) {
  return _badChainlinkResponse(_currentResponse);
}
```

Impact:

Redundant wrapper functions increase the complexity of the codebase and can lead to confusion about the code's intended behavior. They can also increase the gas cost of transactions if they are called frequently.

Recommendation:

To streamline the code and reduce complexity, removing the `_chainlinkIsBroken` function and renaming `_badChainlinkResponse` to `_chainlinkIsBroken` is suggested.

ID	4
Title	Commented Code in fetchPrice Function
Path	contracts/PriceFeedV3.sol
Severity	Info
Status	Fixed
Function Name	fetchPrice

Description:

The fetchPrice function has been modified, and a new function, _storePrice, has been introduced. However, remnants of the previous code remain in the form of commented lines. This can lead to confusion and potential errors in future code modifications.

Code-Affected:

```
function fetchPrice() external override returns (uint256) {
  uint256 _price = _simulatePrice();
  // _changeStatus(_status);
  _storePrice(_price);
  // if (_isAdjusted) {
  // lastAdjusted = block.number;
  // }
  return _price;
} //remove commented code
```

Impact:

Commented-out code in a function can lead to confusion about the function's intended behavior and the evolution of its implementation. This can make future modifications more

error-prone and can lead to bugs if the commented code is accidentally uncommented or not properly updated during refactoring.

Recommendation:

Removing the commented code from the fetchPrice function is recommended to maintain code clarity and prevent potential misunderstandings.

ID	5
Title	Presence of Commented State Variable
Path	contracts/PriceFeedV3.sol
Severity	Info
Status	Fixed
Function Name	-

Description:

Upon careful examination of the code, it has been observed that there exists a state variable that has been commented out. This is considered a code hygiene issue and can lead to confusion for future developers or auditors who may review this code.

Code-Affected:

```
// uint256 public lastAdjusted; //remove this declaration
```

Impact:

The presence of commented-out code can lead to confusion and misinterpretation of the code's functionality. It can also make the codebase harder to maintain and increase the likelihood of errors during future development or refactoring.

Recommendation:

To maintain the highest standards of code cleanliness and readability, it is recommended that this commented state variable be removed from the codebase.

DISCLAIMER

The smart contracts provided by the client for audit purposes have been thoroughly analyzed in compliance with the global best practices to date w.r.t cybersecurity vulnerabilities and issues in smart contract code, the details of which are enclosed in this report.

This report is not an endorsement or indictment of the project or team, and they do not in any way guarantee the security of the particular object in context. This report is not considered and should not be interpreted as an influence on the potential economics of the token, its sale, or any other aspect of the project.

Crypto assets/tokens are the results of emerging blockchain technology in the domain of decentralized finance, and they carry with them high levels of technical risk and uncertainty. No report provides any warranty or representation to any third-Party in any respect, including regarding the bug-free nature of code, the business model or proprietors of any such business model, and the legal compliance of any such business. No third party should rely on the reports in any way, including for the purpose of making any decisions to buy or sell any token, product, service, or another asset. Specifically, for the avoidance of doubt, this report does not constitute investment advice, is not intended to be relied upon as investment advice, is not an endorsement of this project or team, and is not a guarantee as to the absolute security of the project.

Smart contracts are deployed and executed on a blockchain. The platform, its programming language, and other software related to the smart contract can have vulnerabilities that can lead to hacks. The scope of our review is limited to a review of the Solidity code and only the Solidity code we note as being within the scope of our review within this report. The Solidity language itself remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer or any other areas beyond Solidity that could present security risks.

This audit cannot be considered a sufficient assessment regarding the utility and safety of the code, bug-free status, or any other statements of the contract. While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only - we recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contracts.