BlockApex

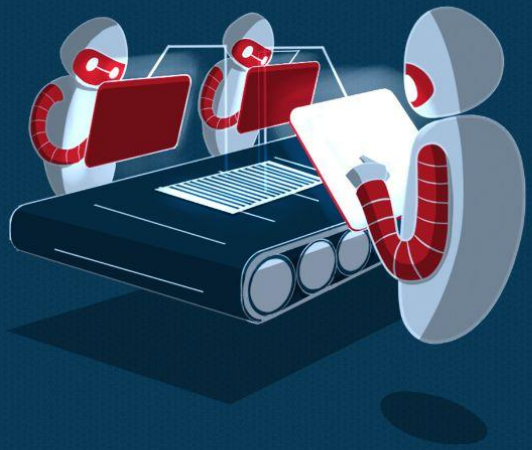
SMART CONTRACT SECURITY ANALYSIS REPORT

```
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";
    }
}
```



Powered by XORD

PREFACE

Objectives

The purpose of this document is 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 intellectual property of the client. It also contains information about potential risks and the processes involved in mitigating/exploiting the risks mentioned below. The usage of information provided in this report is limited, internally, to the client. However, this report can be disclosed publicly with the intention to aid our growing blockchain community; under the discretion of the client.

Key understandings

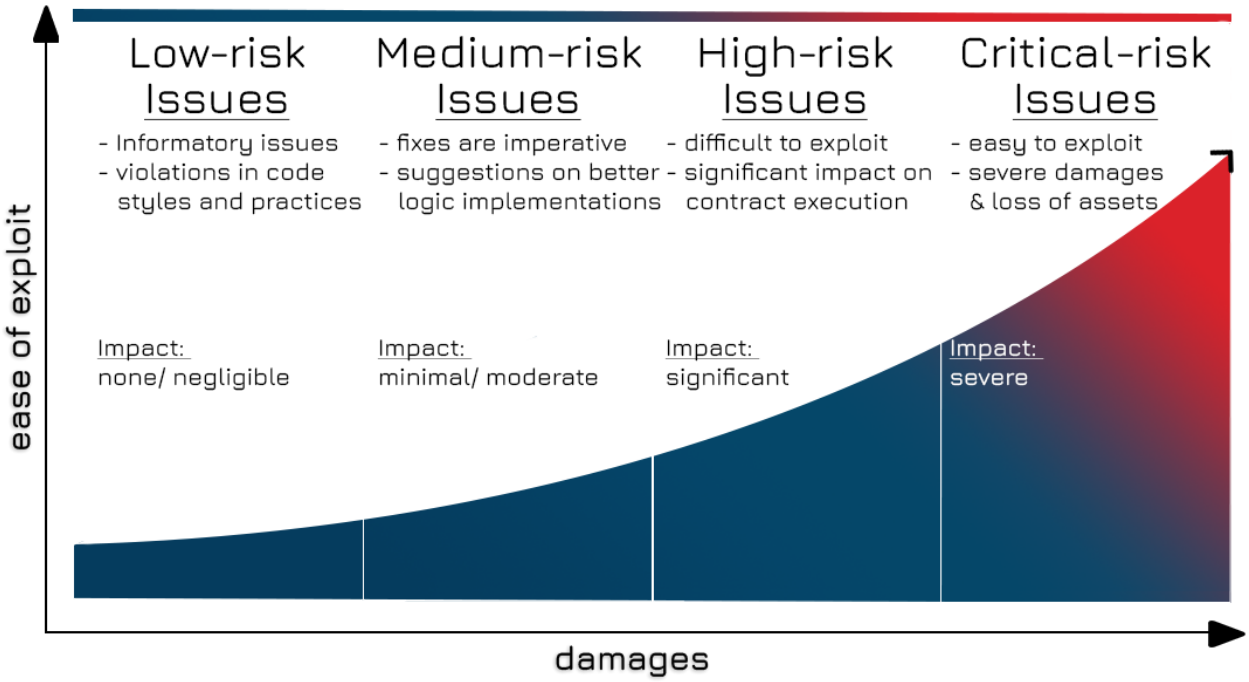


TABLE OF CONTENTS

INTRODUCTION

BlockApex (Auditor) was contracted by DeFiGeek Community (DFGC) (Client) for the purpose of conducting a Smart Contract Audit/Code Review. This document presents the findings of our analysis which started on June 20th, 2022.

Name
Yamato Stablecoin Lending
Audited by
BlockApex
Platform
Ethereum | Solidity
Type of review
Manual Code Review | Automated Tools Analysis
Methods
Architecture Review | Functional Testing | Property Testing | Computer-Aided Verification
Git repository/ Commit Hash
https://github.com/DeFiGeek-Community/yamato
White paper/ Documentation
https://defigeek.xyz/blog/yamato-protocol-v1-overview/
Document log
Initial Audit Completed: July 13th, 2022
Final Audit Completed: Sep 5th, 2022

Scope

The git-repository shared was checked for common code violations along with 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	ERC20 API violation	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 that develops DeFi DApps and middlewares that contribute 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 Japanese Yen stablecoin "CJPY". It is being developed by DeFiGeek Community Japan, a decentralized autonomous organization.

Yamato Protocol is a crypto-asset overcollateralized stable coin 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%)

System Architecture

The idea for the Yamato Protocol is based on Maker and Liquity 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.

Methodology & Scope

The codebase was audited using a filtered audit technique. A pair of auditors scanned the codebase in an iterative process spanning over a span of 2.5 weeks.

Starting with the recon phase, a basic understanding was developed and the auditors worked on establishing presumptions for the codebase and the relevant documentation/ whitepaper provided or found publicly.

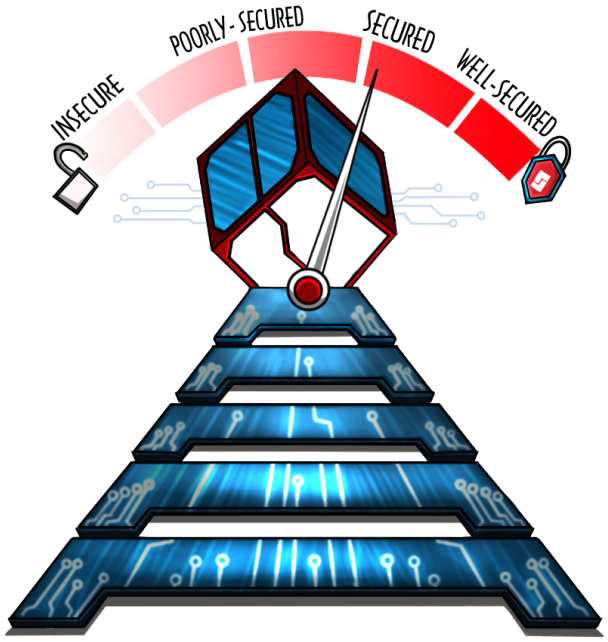
Furthermore, the audit moved on with the manual code reviews with the motive to find logical flaws in the codebase complemented with code optimizations, software and security design patterns, code styles, best practices and identifying false positives that were detected by automated analysis tools.

AUDIT REPORT

Executive Summary

The analysis indicates that the contracts under scope of audit are **working properly**.

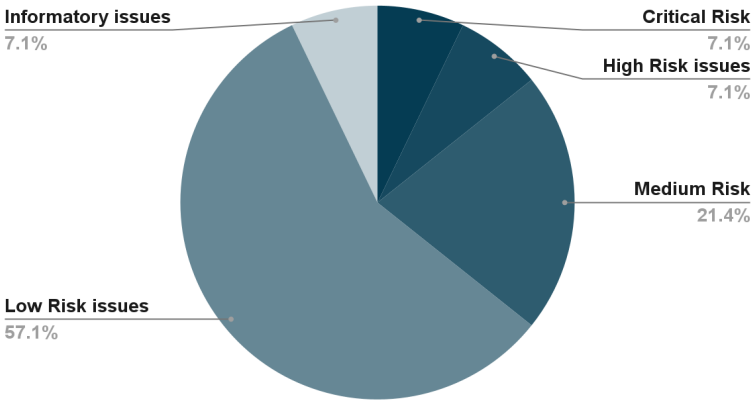
Our team performed a technique called “Filtered Audit”, where the contract was separately audited by two individuals. After a thorough and rigorous process of manual testing, an automated review was carried out using tools like slither for an extensive static analysis and foundry for property testing the invariants of the system. All the flags raised were manually reviewed in collaboration and re-tested to identify the false positives.



Our team found:

# of issues	Severity of the risk
1	Critical Risk issue(s)
1	High Risk issue(s)
3	Medium Risk issue(s)
8	Low Risk issue(s)
1	Informatory issue(s)

Proportion of Vulnerabilities



Key Findings

#	Findings	Risk	Status
1.	Potential classic frontrunning attack	Critical	Acknowledged
2.	Ineffectual Upgradeability	High	Fixed
3.	Tractable balance accounting	Medium	Fixed
4.	Inconsistent non-reentrant pattern	Medium	Fixed
5.	Incorrect version calling	Medium	Fixed
6.	Ineffectual check for ICRPertenk	Low	Fixed
7.	Ineffectual check for equality	Low	Fixed
8.	Misplaced logic statement	Low	Fixed
9.	Unused variable	Low	Fixed
10.	Inexistent Zero Address check	Low	Fixed
11.	Unused and Unnecessary Functions	Low	Fixed
12.	Inconsistent safemath	Low	Fixed
13.	Inconsistent arguments type	Low	Fixed
14.	Inexplicable balances variable	Low	Fixed
15.	Additional Informatory Issues	Informatory	Acknowledged

Detailed Overview

Critical-risk issues

1. Potential classic frontrunning attack

File: contracts/Currency.sol

Description:
The race condition issue for the ERC20 approve() function is an exploitable code of the CJPY ERC20 token. The implementation does not follow the SWC-114 requirement properly and thus falls trap to the sandwich attack, commonly known as MEV.

```
function approve(address spender, uint256 amount)
    public
    override
    returns (bool)
{
    require(
        _msgSender() != spender,
        "sender and spender shouldn't be the same."
    );
    require(amount > 0, "Amount is zero.");

    _approve(_msgSender(), spender, amount);
    return true;
}
```

Remedy:
The issue is described in detail in the official SWC registry considered as the industry standard here. Alternatively, you could search on the internet for issue number SWC-114 which stands for Smart Contract Weakness Classification and Test Cases and is an industry-wide, accepted and maintained registry.

Status:
Acknowledged

High-risk issues

2. Ineffectual Upgradeability

File: contracts/Dependencies/UUPSBase.sol

Description:
In the light of our assumptions with the protocol and the understanding of the UUPS Proxy Pattern, the contract UUPSBase should not implement the `initializer` modifier on the constructor.
The audit team deems this modifier as an anti-pattern practice, according to the industry standards and best practices for the implementation of the Universal Upgradeable Proxy Standard.

```
/// @custom:oz-upgrades-unsafe-allow constructor
constructor() initializer {}
```

Remedy:
In case of such scenarios that follow any misassumed and authorized usage, the protocol may end up to an unreachable state and the funds locked up.

Note: For a to-the-point understanding and confirmation of the assumed protocol deployment, the audit team requires a walkthrough of the deployment procedure from the devs PoV which is not explicitly described in specs/tests or supported documentations.

Else, the priority remedy is to remove the `initializer` modifier along with the UUPSBase constructor.

```
/// @custom:oz-upgrades-unsafe-allow constructor
constructor() {}
```

Status:
Fixed

Medium-risk issues

3. Tractable balance accounting

File: contracts/PoolV2.sol

Description:
Although the `receive` function expects a calling from the `onlyYamato` authorization, the funds will still be accepted by the contract through `self destruct`.
This can bring the contract to an unaccounted state for its ether balances, upon which the `sendETH` and `refreshColl` functions, that manages the user's collateral and releasing mechanism, can lose the accounting of the total balance and always assume a wrong balance.

```
receive() external payable onlyYamato {
    emit ETHLocked(msg.sender, msg.value, address(this).balance);
}
```

Remedy:
It is recommended that a minimal data structure be laid out in the Pool contract to manage the user's balances and collaterals through proper mappings and always let the user pull their right amount of value from the protocol.

Status:
Fixed

4. Inconsistent non-reentrant pattern

File: contracts/YamatoDepositorV2.sol

Description:
The `runDeposit()` function implements an incorrect *checks-effects-interactions* pattern in the commented steps 1 and 2 of the codebase. Refer to the following code snippet for detail and the suggested remedy.

```
/*
    1. Compose a pledge in memory
*/
IYamato.Pledge memory pledge = IYamato(yamato()).getPledge(_sender);
(uint256 totalColl, , , , , ) = IYamato(yamato()).getStates();

pledge.coll += _ethAmount;
if (!pledge.isCreated) {
    // new pledge
    pledge.isCreated = true;
    pledge.owner = _sender;
}

/*
    2. Validate lock
*/
require(
    !IYamato(yamato()).checkFlashLock(_sender),
    "Those can't be called in the same block."
);
require(
    pledge.coll >= IYamatoV3(yamato()).collFloor(),
    "Deposit or Withdraw can't make pledge less than floor size."
);
```

The collateral amount is added right after the pledge is created which leads to the violation of the infamous checks-effects-interactions pattern. The checks are placed further below in step 2 which leads to an inconsistent pattern for the codebase.

Remedy:
The remedy states that the checks be implemented before creating any effects in the storage so that lesser gas cost along with an optimized codebase is achieved.

```
require(
    !IYamato(yamato()).checkFlashLock(_sender),
    "Those can't be called in the same block."
);

IYamato.Pledge memory pledge = IYamato(yamato()).getPledge(_sender);
(uint256 totalColl, , , , , ) = IYamato(yamato()).getStates();

pledge.coll += _ethAmount;

require(
    pledge.coll >= IYamatoV3(yamato()).collFloor(),
    "Deposit or Withdraw can't make pledge less than floor size."
);

if (!pledge.isCreated) {
    // new pledge
    pledge.isCreated = true;
    pledge.owner = _sender;
}
```

Status:
Fixed

5. Incorrect version calling

File: contracts/YamatoDepositorV2.sol

Description:
The `runDeposit()` function calls `fetchPrice()` from the V1 interface of PriceFeed whereas the code for the same in V2 has many modifications and changes. This leads to an inconsistent external calling of functions and ultimately the wrongly assumed outcome.

```
function runDeposit(address _sender) public payable override onlyYamato {
    IPriceFeed(feed()).fetchPrice();
```

Remedy:
It is recommended to change the function such that it inherits and implements the correct IPriceFeed interface or in case this is the correct versioning then comment down the presumptions for the `fetchPrice()` function.

Status:
Fixed

Low-risk issues

6. Ineffectual check for ICRPertenk

File: contracts/Dependencies/PledgeLib.sol

Description:
The function `FR()` implements an unnecessary check in the if block `[13000 <= _ICRpertenk]` for the value of `ICRPertenk` variable which is already handled in the `require` statement at the start of the function.

```
function FR(uint256 _ICRpertenk) public view returns (uint256 _FRpertenk) {
    require(_ICRpertenk >= 13000, "ICR too low to get fee data.");
    // if (11000 <= _ICRpertenk && _ICRpertenk < 13000) {
    //     _FRpertenk = 2000 - ((_ICRpertenk - 11000) * 80) / 100;
    // } else
    if (13000 <= _ICRpertenk && _ICRpertenk < 15000) {
        _FRpertenk = 400 - ((_ICRpertenk - 13000) * 10) / 100;
```

Remedy:
We recommend that this check be removed to save gas and optimize the function.

Status:
Fixed

7. Ineffectual check for equality

File: contracts/YamatoWithdrawerV2.sol

Description:
The function `runWithdraw()` implements a check using >= (greater equals) whereas the statement still fails even if the value of both sides is equal.
For instance, a scenario with the value for `pledge.getICR(feed())` and `uint256(IYamato(yamato()).MCR()) * 100` equal to `13000` will fail.

```
require(
    pledge.getICR(feed()) >= uint256(IYamato(yamato()).MCR()) * 100,
    "Withdrawal failure: ICR is not more than MCR."
);
```

Remedy:
We recommend that checking equality be removed to save gas and optimize the function for edge cases.

Status:
Fixed

8. Misplaced logic statement

File: contracts/YamatoRedeemerV4.sol

Description:
The function `runRedeem()` stores the argument's value at a point which is illogical in light of the `require` and conditional statements.

```
        _args.wantToRedeemCurrencyAmount = IPool(pool())
            .redemptionReserve();
```

Remedy:
It is recommended that the value is stored outside the if block as follows.

```
    vars.ethPriceInCurrency = IPriceFeed(feed()).fetchPrice();
    _args.wantToRedeemCurrencyAmount = IPool(pool()).redemptionReserve();
    if (_args.isCoreRedemption) {
        require(
            _args.wantToRedeemCurrencyAmount > 0,
            "The redemption reserve is empty."
        );
    } else {
        require(
            _cjpy.balanceOf(_args.sender) >=
                _args.wantToRedeemCurrencyAmount,
            "Insufficient currency balance to redeem."
        );
    }
```

Status:
Fixed

9. Unused variable

File: contracts/YamatoRedeemerV4.sol

Description:
The memory variable `reminder` in the `runRedeem()` function is redundant as it stores the value which is never used throughout the local scope of the function.

```
    vars._reminder = _args.wantToRedeemCurrencyAmount;
```

Remedy:
We recommend that the unnecessary variable be removed from the data structure and in the function for execution gas and storage optimization.

Status:
Fixed

10. Inexistent Zero Address check

File: contracts/Currency.sol

Description:
The `setCurrencyOS(address _currencyOSAddr)` function does not validate the input for the param `_currencyOSAddr` whether a user provides a non-zero address.

```
function setCurrencyOS(address _currencyOSAddr) public onlyGovernance {
    currencyOS = _currencyOSAddr;
}
```

Remedy:
It is recommended that a `require` statement with correct input validations be included in the codebase.

Status:
Fixed

11. Unused and Unnecessary Functions

File: contracts/Dependencies/ [Ownable.sol && YamatoStore.sol]

Description:
The `_renounceOwnership()` and `__YamatoStore_init_unchained()` functions are never called and contain empty code blocks respectively, hence, not performing any valuable activity.

```
function _renounceOwnership() internal {
    emit OwnershipTransferred(_owner, address(0));
    _owner = address(0);
}
```

```
function __YamatoStore_init_unchained() internal initializer {}
```

Remedy:
It is recommended to remove the functions from the codebase in favor of execution risks and storage optimizations.

Status:
Fixed

12. Inconsistent safemath

File: contracts/YamatoWithdrawerV2.sol

Description:
The `pledge.coll` attribute can be updated using the consistent pattern as in the Depositor and Borrower contracts.

```
        /*
            4. Update pledge
        */
        // Note: SafeMath unintentionally checks full withdrawal
        pledge.coll = pledge.coll - _ethAmount;
```

Remedy:
It is recommended to utilize a consistent coding practice throughout the Yamato Actions smart contracts.

```
        pledge.coll -= _ethAmount;
```

Status:
Fixed

Additional informatory issues and optimizations

Note:
Increase test cases and code coverage for a satisfactory upgradeable contracts scenario. The current state of test cases do not engage any kind of upgradeability tests which poses a good deal of threat for a wide range of attack vectors.

This list contains additional coding style guides and best practices for a consistent codebase of the Yamato Protocol.

- **General Codebase Recommendations:**
  - Include thorough NatSpec documentation in the codebase.
  - Remove the usage of hardhat/console.sol as it increases the deployment and execution gas cost.
  - Convert all public functions to external if the contract itself or the inherited contracts do not use them.
  - Avoid using SafeMath for redundant operations as the codebase is already using solidity versions ^0.8.0.
- **Mark Variable Visibility Explicitly:**
  - In YamatoBase.sol: L20
  - In YamatoV3.sol: L69, L86
  - In PriorityRegistryV6.sol: L26, L30, L34, L35 (tight packing)
  - In PriceFeedV2.sol: L61, L62 (L94-L98: tight packing & wrap in struct)
- **Mark Function Visibility Explicitly:**
  - In YamatoV3.sol:
    - `setDeps()` as external
    - `toggle()` as external
    - `setPriorityRegistry()` as external
  - In PriorityRegistryV6.sol:
    - `bulkUpsert()` as internal
  - In CurrencyOSV2.sol:
    - `setDeps()` as external
    - `setPriceFeed()` as external
  - In UUPSBase.sol:
    - `setGovernance()` as external

    - `acceptGovernance()` as external
    - `revokeGovernance()` as external
- **In PledgeLib.sol:**
    - NatSpec documentation not found for multiple functions in the library file as follows;
      `toMem()`, `clone()`, `addDebt()`, `sync()`, `assign()`, `nil()`, `toBeRedeemed()`, `cappedRedemptionAmount()`
    - Though the contracts use solidity version 0.8.4 that comes prebuilt with SafeMath, the library implementation uses SafeMath and functions that check for overflow/underflow which is redundant and unnecessary.
- **In YamatoStore.sol:**
    - The contract contains an empty function with the `initializer` modifier without any actual value or processing inside, but just as part of the UUPS pattern. Suggested to remove the modifier for redundant calling.
- **In YamatoV3.sol:**
    - The `setFlashLock()` function can be optimized to replace the redundant check `(lock.blockHeight < block.number)` and the redundant setting `lock.blockHeight = block.number;` to be implemented in an optimized way only once at the first if block instead of each conditional check.
    - The code block in `setPledge()` function can be optimized and made consistent by moving it to the top or below the remaining singular checks as follows;

```
Pledge storage p = pledges[_owner];
if (_p.debt == 0 && _p.coll == 0) {
    _p.owner = address(0);
    _p.isCreated = false;
    _p.priority = 0;
}
if (p.coll != _p.coll) {
    p.coll = _p.coll;
}
if (p.debt != _p.debt) {
    p.debt = _p.debt;
}
if (p.owner != _p.owner) {
    p.owner = _p.owner;
}
```

```
        if (p.isCreated != _p.isCreated) {
            p.isCreated = _p.isCreated;
        }
        if (p.priority != _p.priority) {
            p.priority = _p.priority;
        }
```

  - Move the `onlyYamato` modifier to the top of the contract to follow the code style guides as in the Solidity docs.
  - Correct the typo on line 349; *`/// @dev Nood reentrancy guard.`*
  - Correct the spelling for the method `getIndivisualStates()` to `getIndividualStates()`
- **In YamatorRepayerV2.sol:**
  - The return variable `_wantToRepayCurrencyAmount` naming is misleading and can be corrected to `_repayAmountInCurrency` for consistency.
  - On L85 & L86, the counting for steps can be corrected to 6-1 and 6-2.
  - On L86, the comment should be corrected as the *`redeemer`* should be *`repayer`*
- **In CurrencyOSV2.sol:**
  - The getter method for `feed()` and the following instances of price feed should be replaced with `priceFeed()` for self explanatory use cases throughout the codebase.
  - Always emit an event when `addYamato()` is called.
  - Remove redundant import for PriceFeed.sol
  - Change the access modifier to *view* for the `_permitMe()` function.
  - Inherit the IYamatoV3.sol interface instead of IYamato.sol
- **In Currency.sol:**
  - The modifier `onlyGovernance` should move to the top of the contract to follow the Solidity docs code style.
  - Emit events for `setCurrencyOS()` and `revokeGovernance()` functions.
- **In PriceFeedV2.sol:**
  - The struct `ChainlinkResponse` and `TellorResponse` can be moved to the interface file.
  - The function `_simulatePrice` being internal can be moved to the bottom of the contract.
  - Use return variable `_price` in function signature declaration as
    - `function getPrice() external view override returns (uint256 _price) {`

        - `function _scaleTellorPriceByDigits(uint256 _price) internal pure returns (uint256 _price) {`
    - Remove this redundant check: `else if (_answerDigits < TARGET_DIGITS)`
    - Move all public functions to the top of the contract file to follow the solidity docs guideline.
- **In UUPSBase.sol:**
    - The modifier `onlyTester` is a redundant authorized role which should be removed from the production codebase.

DISCLAIMER

The smart contracts provided by the client for audit purposes have been thoroughly analyzed in compliance with the global best practices till 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 results of the 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 other 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 it 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 its 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 as 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 security of smart contracts.