BLOCKAPEX

SMART CONTRACT SECURITY

V 1.0

DATE: 8th AUG 2024

PREPARED FOR: YAMATO PROTOCOL

SECURITY REPORT
BLOCKAPEX VERIFIED

About BlockApex

Founded in early 2021, is a security-first blockchain consulting firm. We offer services in a wide range of areas including Audits for Smart Contracts, Blockchain Protocols, Tokenomics along with Invariant development (i.e., test-suite) and Decentralized Application Penetration Testing. With a dedicated team of over 40+ experts dispersed globally, BlockApex has contributed to enhancing the security of essential software components utilized by many users worldwide, including vital systems and technologies.

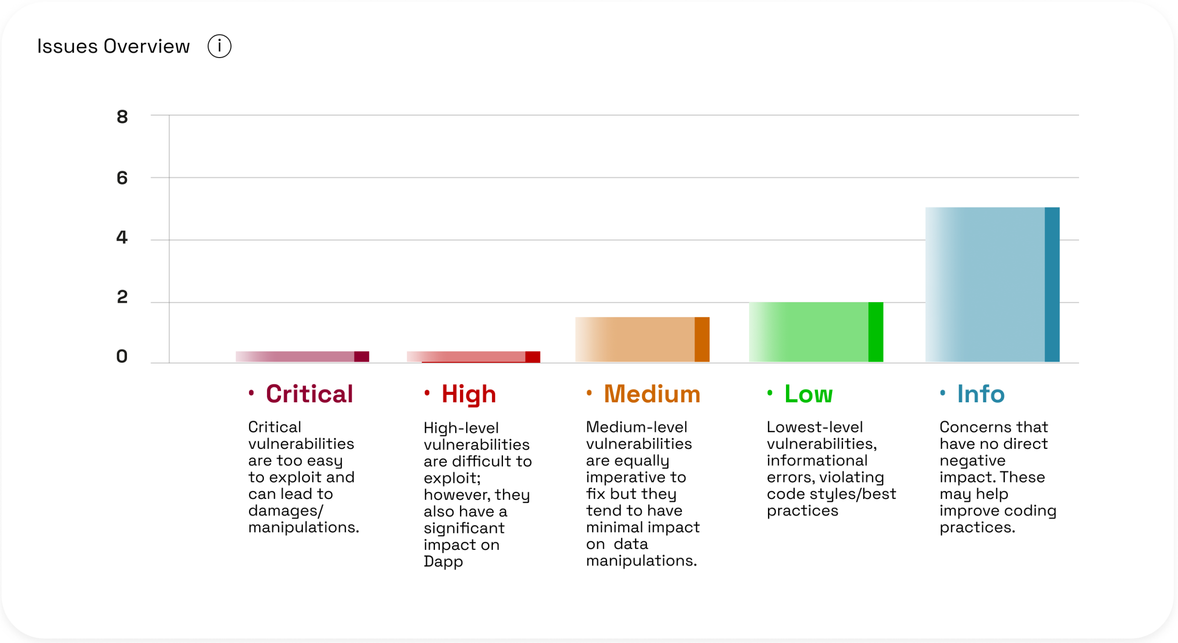
BlockApex has a focus on blockchain security, maintaining an expertise hub to navigate this dynamic field. We actively contribute to security research and openly share our findings with the community. Our work is available for review at our public repository, showcasing audit reports and insights into our innovative practices.

To stay informed about BlockApex's latest developments, breakthroughs, and services, we invite you to follow us on Twitter and explore our GitHub. For direct inquiries, partnership opportunities, or to learn more about how BlockApex can assist your organization in achieving its security objectives, please visit our Contact page at our website , or reach out to us via email at hello@blockapex.io.

Contents

1 Executive Summary

Our team performed a technique called Filtered Audit, where two individuals separately audited the Yamato Smart Contracts. After a thorough and rigorous manual testing process involving line by line code review for bugs. All the raised flags were reviewed and re-tested to identify any false positives.



1.1 Scope

1.1.1 In Scope

The audit comprehensively covered the Yamato Protocol, focusing on critical components essential for the platform's functionality and security. Below are the details of the modules that were in scope:

Deposit & Withdrawal (ETH)

- **ETH Deposit** The ETH deposit functionality allows users to pledge ETH as collateral with a minimum deposit requirement of 0.1 ETH. Users can add more ETH to meet the minimum requirement if their collateral falls below this threshold due to redemptions.
- **ETH Withdrawal** Users must maintain a minimum of 0.1 ETH as collateral if they have outstanding debt. If there is no outstanding debt, users can fully withdraw their ETH collateral. The withdrawal process ensures the user's collateral and debt ratios remain within acceptable ranges to protect the protocol's integrity.

Health Rate & Fee Rate

- **One-Time Borrowing Fee** A one-time fee is charged at the time of borrowing, calculated as the borrowing amount multiplied by the applicable fee rate. The fee rate varies based on the health rate after borrowing, which ensures that users with higher collateral ratios pay lower fees.
- **Fee Calculation and Repayment** Fees are deducted upfront from the borrowed amount, meaning users must purchase CJPY to fully repay their debt. Users can strategically minimize fees by borrowing at a high health rate and then withdrawing collateral.
- **Fee Allocation** Collected fees are allocated to the redemption pool and the repayment pool in an 8:2 ratio, ensuring that the system remains robust and can handle redemptions efficiently while also supporting the repayment pool.

Borrowing & Repayment

- **Borrowing Mechanism** Users can borrow against their deposited collateral, with the amount they can borrow determined by their collateral ratio and the corresponding fee rate. The fee is calculated as a percentage of the borrowed amount and is deducted upfront, ensuring that users need to purchase CJPY to fully repay their debt.
- **Repayment Mechanism** Users can repay their debt to reclaim their collateral. The repayment amount is checked against the user's balance and debt to ensure validity.

Redemption Mechanism The redemption mechanism in the Yamato Protocol is designed to improve the health ratio of Pledges by enabling the purchase of collateral at market value and the repayment of CJPY. Redemptions are prioritized based on the lowest collateral ratio, ensuring that the most at-risk pledges are addressed first. There are two primary methods of redemption: by user and by protocol.

Redemption Eligibility

- Only pledges with a health rate of less than 130% are eligible for redemption.
- Redemptions proceed from the pledge with the lowest collateral ratio.

Redemption by User

- Users can redeem using their CJPY holdings.
- This redemption acts as an exchange of CJPY to ETH at a rate of 1 CJPY = 1 JPY.
- Users burn their CJPY to retrieve ETH collateral from pledges.

Redemption by Protocol

- The protocol itself can trigger redemptions using accumulated CJPY.
- Anyone can initiate a protocol redemption, and the initiator receives 1% of the collateral obtained.
- This method also operates at a rate of 1 CJPY = 1 JPY.

Sweep Module: Yamato Subrogation Repayment The sweep module in the Yamato Protocol addresses pledges with a health rate below 100%, ensuring the repayment of CJPY obligations through Yamato subrogation, using accumulated CJPY fees within the protocol. When the health rate is less than 100%, even after redemption, these pledges are repaid using Yamato's CJPY reserve. The trigger executor receives 1% of the repayment amount in CJPY as an incentive. This operation is carried out by an incentivized DAO member who scans all pledges to identify those with debt greater than 0 and collateral equal to 0. The process continues until the targeted sweeping amount is reached or all eligible pledges are swept.

Score Registry

The Score Registry is designed to manage user scores, which are based on their activity and financial status within the protocol. It tracks key metrics like collateral ratios and total debt to determine user eligibility for rewards.

Priority Registry

The Priority Registry is intended to manage the priority of user pledges, ensuring that the most critical pledges (those with the lowest collateral ratios) are handled first during operations like redemption and sweeping.

Weight Controller The Weight Controller oversees the assignment and adjustment of weights to user scores, directly affecting how rewards and incentives are distributed within the protocol.

YMT Token The YMT (Yamato) token is the primary token for the Yamato Protocol, designed to support dividends and inflation. The token's supply can increase over time, but the rate of increase is designed to decrease gradually.

YMT Minter The YMT Minter is responsible for minting YMT tokens based on user scores, ensuring that tokens are distributed fairly according to user contributions and activities within the protocol.

veYMT The veYMT (voting-escrowed YMT) token is a locked version of the YMT token, designed to provide time-weighted voting power to users who commit their tokens for a longer period. This mechanism incentivizes long-term commitment to the protocol.

Fee Pool The Fee Pool module is designed to manage and distribute protocol fees to veYMT token holders. It ensures that users who lock their YMT tokens and participate in the protocol governance are rewarded with fees.

CurrencyOSV3 The CurrencyOSV3 module manages the minting and burning of the protocol's stablecoin. It interacts with various dependencies, including price feeds and fee pools, to ensure stable and secure operations.

Contracts in Scope:

- contracts/CurrencyOSV3.sol
- contracts/FeePoolV2.sol
- contracts/ScoreRegistry.sol
- contracts/ScoreWeightController.sol
- contracts/YMT.sol
- contracts/YamatoBorrowerV2.sol
- contracts/YamatoDepositorV3.sol
- contracts/YamatoRedeemerV5.sol
- contracts/YamatoRepayerV3.sol
- contracts/YamatoSweeperV3.sol
- contracts/YamatoV4.sol
- contracts/YamatoWithdrawerV3.sol
- contracts/YmtMinter.sol
- contracts/veYMT.sol

Initial Commit Hash: 6f595cd351f8b140cab0fab1d0d6fb806cc8b390

Final Commit Hash: 957037eb96b9fa34a8dbbe29fad108b82751f438

1.1.2 Out of Scope

All features or functionalities not delineated within the "In Scope" section of this document shall be deemed outside the purview of this audit. Additionally, the audit of the scope was limited to the Yamato Protocol v1.5. Any further updates beyond this version are not within the scope of this audit.

1.2 Methodology

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 2.5 weeks. 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 and best practices

1.3 Questions for Security Assessment

Deposit & Withdrawal (ETH)

1. How does the system validate and enforce the minimum deposit requirement of 0.1 ETH to ensure collateral adequacy?
2. What mechanisms are in place to prevent unauthorized or invalid withdrawals, especially in cases where users have outstanding debt?
3. How does the system maintain the integrity of collateral and debt ratios during deposit and withdrawal operations?

Health Rate & Fee Rate

4. How is the one-time borrowing fee calculated, and what safeguards are in place to ensure the fee rate is correctly applied based on the health rate?
5. What measures are implemented to prevent manipulation of the health rate to lower borrowing fees?
6. How does the system ensure the accurate allocation of collected fees to the redemption pool and repayment pool?

Borrowing & Repayment

7. What mechanisms are in place to determine the maximum borrowing amount based on collateral ratios and fee rates?
8. How does the system ensure the validity and integrity of repayment transactions to prevent overpayment or underpayment of debt?
9. What safeguards are implemented to protect against unauthorized borrowing or repayment actions?

Redemption Mechanism

10. How does the system prioritize redemptions based on the lowest collateral ratio, and what measures prevent abuse or manipulation of this process?
11. What validations are performed to ensure only eligible pledges (health rate less than 130%) are subject to redemption?
12. How does the protocol ensure the accurate exchange rate of 1 CJPY = 1 JPY during user and protocol redemptions?

Sweep Module: Yamato Subrogation Repayment

13. How does the sweep module identify and target pledges with a health rate below 100% for subrogation repayment?
14. What mechanisms ensure the accurate calculation and distribution of the 1% incentive to the trigger executor?
15. How does the system prevent unauthorized or malicious scanning and sweeping of pledges?

Score Registry

16. What measures are in place to accurately track and update user scores based on collateral ratios and total debt?
17. How does the system ensure the integrity of score-related data to prevent manipulation or falsification?
18. How are score limits calculated and enforced to maintain the stability of the protocol?

Priority Registry

19. How does the Priority Registry determine the priority of user pledges, and what safeguards are in place to prevent manipulation of priority rankings?
20. What mechanisms ensure the correct handling of pledges during operations like redemption and sweeping?

Weight Controller

21. How does the Weight Controller assign and adjust weights to user scores, and what measures prevent unauthorized changes?
22. What safeguards are in place to ensure the fair distribution of rewards and incentives based on user scores?

YMT Token

23. How is the inflation rate of YMT tokens managed, and what mechanisms ensure the rate decreases over time as designed?
24. What measures are in place to prevent unauthorized minting or burning of YMT tokens?
25. How does the system ensure the secure transfer and storage of YMT tokens?

YMT Minter

26. How does the YMT Minter verify user scores before minting tokens, and what safeguards are in place to prevent manipulation of scores?
27. What mechanisms ensure the fair distribution of minted YMT tokens based on user contributions and activities?

28. How does the system handle edge cases or errors during the minting process to prevent inconsistencies?

veYMT

29. How does the veYMT mechanism ensure the accurate calculation of time-weighted voting power based on the length of token lock-up?
30. What safeguards are in place to prevent unauthorized changes to veYMT token locks or balances?
31. How does the system incentivize long-term commitment to the protocol through veYMT tokens?

Fee Pool

32. How does the Fee Pool manage and distribute protocol fees to veYMT token holders, and what measures ensure fairness in distribution?
33. What mechanisms prevent unauthorized access to or manipulation of the fee pool?
34. How does the system ensure the accurate calculation and distribution of fees to eligible participants?

CurrencyOSV3

35. How does the CurrencyOSV3 module manage the minting and burning of the protocol's stablecoin, and what safeguards are in place to prevent unauthorized actions?
36. What measures ensure the stability and security of the stablecoin operations, including interactions with price feeds and fee pools?
37. How does the system handle potential discrepancies or errors in price feeds to maintain stablecoin value?

General Security Concerns

38. Are there any potential vulnerabilities in the smart contracts that could lead to reentrancy attacks, and how are they mitigated?
39. How does the system ensure the integrity and consistency of state across complex transactions involving multiple modules?
40. What access control mechanisms are implemented to safeguard against unauthorized actions by owners or privileged users?
41. Are safe mathematical operations used throughout the smart contracts to prevent overflows and underflows, especially in financial computations?
42. How does the protocol ensure the security and integrity of data during contract upgrades or migrations, if applicable?
43. Are there any design flaws or vulnerabilities that could lead to direct financial loss for users or the protocol, and how are they addressed?

1.4 Status Descriptions

Acknowledged: The issue has been recognized and is under review. It indicates that the relevant team is aware of the problem and is actively considering the next steps or solutions.

Fixed: The issue has been addressed and resolved. Necessary actions or corrections have been implemented to eliminate the vulnerability or problem.

Closed: This status signifies that the issue has been thoroughly evaluated and acknowledged by the development team. While no immediate action is being taken.

1.5 Summary of Findings Identified

S.No	Severity	Findings	Status
#1	MEDIUM	Potential Front-Running and Denial of Service in veYMT depositFor Function	FIXED
#2	LOW	Lack of Input Validation in setAdmin Function Allows Setting Invalid Admin Address	ACKNOWLEDGED
#3	LOW	Precision Loss Due to Division Before Multiplication in Calculations	ACKNOWLEDGED
#4	INFO	Unsafe Typecasting During Deposit in veYMT	FIXED
#5	INFO	Missing Checks-Effects-Interactions (CEI) Pattern in _mintFor Function	FIXED
#6	INFO	Inefficient Use of Revert Strings Instead of Custom Errors	ACKNOWLEDGED
#7	INFO	Gas Efficiency: Prefer Pre-increment (++i) over Post-increment (i++)	FIXED
#8	INFO	Redundant Initialization of Variables with Default Values	FIXED

2 Findings and Risk Analysis

2.1 Potential Front-Running and Denial of Service in veYMT `depositFor` Function

Severity: Medium

Status: Fixed

Location :

1. `contracts/veYMT.sol`

Description In the `veYMT` contract, the `depositFor` function allows users to deposit tokens on behalf of other users, provided they have the necessary *approval*. However, this functionality introduces a potential security issue. If a user has already created a lock and sets an allowance, a malicious attacker could front-run the transaction by calling `depositFor` with a small amount. This action reduces the user's allowance, potentially causing the legitimate `increaseAmount` call to fail. This issue can lead to a denial of service (DoS) attack, preventing the user from increasing their locked amount, and allowing the attacker to gain more voting power within the protocol.

Impact This vulnerability can be exploited to perform a denial of service (DoS) attack on users, preventing them from increasing their locked amount in the veYMT contract. This can skew the voting power dynamics within the protocol, allowing malicious actors to gain an unfair advantage.

Proof of Concept :

1. User sets an allowance of 1 ETH to increase their locked amount.
2. Malicious actor front-runs the transaction and calls depositFor with a minimal amount (e.g., 0.01 ETH).
3. User’s transaction fails due to insufficient allowance.
4. Malicious actor repeats the process, effectively preventing the user from increasing their locked amount.

POC

````
function testDepositFor() public {
    address callerBoii = address(0x1238);
    deal(callerBoii, 10 ether);
    vm.startPrank(callerBoii);

    (bool s, ) = address(yamatoProxy).call{value: 5 ether}(
        abi.encodeWithSignature(`deposit()`, ``)
    );
    require(s, `Deposit failed`);

    vm.stopPrank();
    vm.roll(block.number + 4);
    skip(block.timestamp + 4 seconds);
    vm.prank(myBoi);

    _ethPriceAggregatorInUSDAddress.setLastPrice(1);
    vm.prank(myBoi);
    _jpyPriceAggregatorInUSDAddress.setLastPrice(1);

    vm.startPrank(callerBoii);
    (s, ) = address(yamatoProxy).call(
        abi.encodeWithSignature(`borrow(uint256)``, 1 ether)
    );
    require(s, `Borrow failed`);
    skip(block.timestamp + 1 seconds);
    //call YMT() from ymtMinterProxy to check if its not zero address
    address(ymtMinterProxy).call(abi.encodeWithSignature(`YMT()``, ``));
    (s, ) = address(ymtMinterProxy).call(
        abi.encodeWithSignature(`mint(address)`, address(scoreRegistryProxy))
    );
    require(s, `Mint failed`);

    uint256 ymtBalanceCallerBoi = ymt.balanceOf(callerBoi);
    console.log(`ymtBalanceCallerBoi: %d`, ymtBalanceCallerBoi);

    vm.stopPrank();
    vm.startPrank(callerBoi);
    // approve ymt from callerboi to veymt contract
    (s, ) = address(ymt).call(
        abi.encodeWithSignature(`approve(address,uint256)`, address(veYMTContract), 2 ether)
    );
    // call createlock from veYMTContract
    (s, ) = address(veYMTContract).call(
        abi.encodeWithSignature(`createLock(uint256,uint256)`, 1 ether, block.timestamp + 1 weeks )
    );
    require(s, `CreateLock failed`);

    uint256 ymtBalanceCallerBoii = ymt.balanceOf(callerBoi);
    console.log(`ymtBalanceCallerBoiAfter: %d`, ymtBalanceCallerBoii);

    vm.stopPrank();

    // Log the lock end time
    uint256 lockEnd = veYMTContract.lockedEnd(callerBoi);
    console.log(`Lock end time: %s`, lockEnd);

    // Log the current block timestamp
    console.log(`Current block.timestamp: %s`, block.timestamp);

    // create another user account and call depositfor function from the veymt contract
````

```
        and on behalf of the new user account to the callerboi
    address newCallerBoi = address(0x1239);
    vm.startPrank(newCallerBoi);
    (s, ) = address(veYMTContract).call(
        abi.encodeWithSignature(`depositFor(address,uint256)``, callerBoi, 1 ether)
    );
    require(s, `DepositFor failed`);

    vm.stopPrank();
    // call increase lock from callerboi to increase the lock amount
    vm.startPrank(callerBoi);
    (s, ) = address(veYMTContract).call(
        abi.encodeWithSignature(`increaseAmount(uint256)`, 1 ether)
    );
    require(s, `IncreaseAmount failed`);
    vm.stopPrank();
}
```

Recommendation :

1. **Whitelisting Mechanism**: If possible, add a whitelisting mechanism so that users can whitelist specific addresses that are allowed to call depositFor on their behalf. This would reduce the risk of unauthorized or malicious actors performing front-running attacks.
2. **User Notification**: Update the documentation to inform users about the risks associated with setting allowances

2.2 Lack of Input Validation in setAdmin Function Allows Setting Invalid Admin Address

Severity: Low

Status: Acknowledged

Location :

1. `contracts/YMT.sol`

Description The `setAdmin` function in `ymt.sol` lacks input validation for the _admin parameter. This oversight allows the possibility of setting the admin address to an invalid address, such as the zero address (`0x0000000000000000000000000000000000000000`), which can disrupt the contract's administrative functionality and governance.

Affected Code

```

    function setAdmin(address _admin) external onlyAdmin {
        admin = _admin;
        emit SetAdmin(_admin);
    }
```

Recommendation Add a validation check to ensure the `_admin` parameter is not the zero address or any other invalid address.

2.3 Precision Loss Due to Division Before Multiplication in Calculations

Severity: Low

Status: Acknowledged

Location :

1. `contracts/veYMT.sol`
2. `contracts/FeePoolV2.sol`
3. `contracts/ScoreRegistry.sol`

Description Several instances in the Yamato Protocol code perform division before multiplication in their calculations, which can lead to precision loss due to integer division truncation. In Solidity, integer division truncates the decimal portion, which can result in inaccurate calculations when performed before multiplication

Affected Code

```
function increaseUnlockTime(uint256 unlockTime_) external nonReentrant {
      LockedBalance memory _locked = locked[msg.sender];
      uint256 _unlockTimeRounded;
      unchecked {
          _unlockTimeRounded = (unlockTime_ / WEEK) * WEEK;
      }
```

Recommendation Perform multiplication before division to maintain precision.

2.4 Unsafe Typecasting During Deposit in veYMT

Severity: Info

Status: Fixed

Location :

1. `contracts/veYMT.sol`

Description The `depositFor` function in the veYMT contract has an issue due to unsafe typecasting. Specifically, the line *locked.amount += int128(uint128(value*)); casts a uint256 value to uint128 and then to int128. This can cause an overflow if `value_` exceeds the maximum limit of `int128 (2^127 - 1)`. This could lead to incorrect calculations for `_locked.amount`, potentially affecting the overall lock amount and voting power calculations.

Affected Code:

```
// Adding to existing lock, or if a lock is expired - creating a new one
        _locked.amount += int128(uint128(value_));
        if (unlockTime_ != 0) {
            _locked.end = unlockTime_;
        }
        locked[addr_] = _locked;
```

Recommendation :

1. Safe Typecasting: Ensure that the typecasting from uint256 to int128 is safe by checking that value_ does not exceed the maximum limit of int128.
2. Use safecast libraries

2.5 Missing Checks-Effects-Interactions (CEI) Pattern in _mintFor Function

Severity: Info

Status: Fixed

Location :

1. `contracts/YmtMinter.sol`

Description The `_mintFor` function in `ymtMinter.sol` does not follow the Checks-Effects-Interactions (CEI) pattern, which is a best practice to prevent reentrancy attacks. The current implementation first interacts with external contracts (IYMT(YMT()).mint), and only after that does it update the internal state (`minted[for_][scoreAddr_] = totalMint`). This order of operations can leave the contract vulnerable to reentrancy attacks

Affected Code:

```
function _mintFor(
    address scoreAddr_,
    address for_
) internal returns (uint256) {
    require(
        IScoreWeightController(scoreWeightController()).scores(scoreAddr_) &gt;
            0,
        `dev: score is not added`
    );

    IScoreRegistry(scoreAddr_).userCheckpoint(for_);
    uint256 totalMint = IScoreRegistry(scoreAddr_).integrateFraction(for_);
    uint256 _toMint = totalMint - minted[for_][scoreAddr_];

    if (_toMint != 0) {
        IYMT(YMT()).mint(for_, _toMint);
        minted[for_][scoreAddr_] = totalMint;

        emit Minted(for_, scoreAddr_, totalMint);
    }
    return _toMint;
}
```

Recommendation Reorder the operations in the function to follow the Checks-Effects-Interactions pattern.

2.6 Inefficient Use of Revert Strings Instead of Custom Errors

Severity: Info

Status: Acknowledged

Location The use of revert strings is widespread across several modules, including but not limited to:

1. `contracts/YamatoDepositorV3.sol`
2. `contracts/YamatoRedeemerV5.sol`
3. `contracts/YamatoBorrowerV2.sol`

Description Throughout the Yamato Protocol codebase, there are multiple instances where revert strings are used in require and revert statements instead of custom errors. Revert strings can lead to higher gas costs because the entire string is stored in the transaction's logs. Solidity version 0.8.4 introduced custom errors, which offer a more gas-efficient way to handle errors. By using custom errors, the Yamato Protocol can reduce both deployment and runtime costs, improving overall efficiency.

Affected Code:

```
require(
    !IYamato(yamato()).checkFlashLock(_sender),
    `Those can``t be called in the same block.`
);
require(pledge.isCreated, `This pledge is not created yet.`);
require(
    _ICRAfter &gt;= uint256(IYamato(yamato()).MCR()) * 100,
    `This minting is invalid because of too large borrowing.`
);
require(fee &gt; 0, `fee must be more than zero.`);
require(
    returnableCurrency &gt; 0,
    `(borrow - fee) must be more than zero.`
);
```

Recommendation Define custom errors for different failure scenarios at the beginning of the contract.

2.7 Gas Efficiency: Prefer Pre-increment (++i) over Post-increment (i++)

Severity: Info

Status: Fixed

Location :

1. `contracts/CurrencyOSV3.sol`
2. `contracts/FeePoolV2.sol`
3. `contracts/YamatoV4.sol`
4. `contracts/PriorityResgistryV6.sol`
5. `contract/ymt.sol`

Description When incrementing an unsigned integer within Solidity, using pre-increment (++i) is more gas-efficient than post-increment (i++) or the addition assignment (i += 1). This efficiency gain stems from the inherent workings of these operations. With pre-increment, the value is incremented first and then returned, eliminating the need for an intermediary variable. On the other hand, post-increment increments the value but returns its initial state, thus necessitating a temporary variable when utilized.

Proof of Concept Example Illustration:

```
Using post-increment:
uint256 i = 1;
i++; // Result is 1, but `i`gets updated to 2 subsequently
Using pre-increment:
uint256 i = 1;
++i; // Directly results in 2, and `i`is also updated to 2 without any intermediary steps.
```

Recommendation To optimize gas usage in loops or recurring operations, consider adopting pre-increment over its alternatives. Ensuring these best practices can lead to tangible gas savings over numerous iterations, especially in contracts with high transaction frequency.

2.8 Redundant Initialization of Variables with Default Values

Severity: Info

Status: Fixed

Location :

1. `contracts/CurrencyOSV3.sol`
2. `contracts/PriorityRegistryv6.sol`
3. `contracts/ScoreRegistry.sol`
4. `contracts/veYMT.sol`
5. `contracts/YamatoV4.sol`

Description In Solidity, variables are implicitly initialized with default values, such as 0 for uint, false for bool, and address(0) for address types. It is unnecessary and inefficient to explicitly set these variables to their default values. Doing so results in superfluous gas consumption and can be regarded as a programming anti-pattern.

Recommendation We advise a thorough review of the contract to identify and remove any explicit initializations with default values. Simplifying the code in this manner will optimize gas consumption and align with best practices in Solidity development.

Disclaimer: