ZEALYNX SECURITY

Web3 Security & Smart Contract Development

RIBBON

Security Assessment

June 18th, 2024 – Prepared by Zealynx Security

ZealynxSecurity@protonmail.com

Secoalba	@Seecoalba
Bloqarl	@TheBlockChainer

Contents

1. About Zealynx

Zealynx Security, founded in January 2024 by Bloqarl and Sergio, specializes in smart contract audits, development, and security testing using Solidity and Rust. Our services include comprehensive smart contract audits, smart contract fuzzing, and formal verification test suites to enhance security reviews and ensure robust contract functionality. We are trusted by clients such as Shieldify, AuditOne, Bastion Wallet, Side.xyz, Possum Labs, and Aurora (NEAR Protocol-based).

Our team believes in transparency and actively contributes to the community by creating educational content. This includes challenges for Foundry and Rust, security tips for Solidity developers, and fuzzing materials like Foundry and Echidna/Medusa. We also publish articles on topics such as preparing for an audit and battle testing smart contracts on our blog, Bloqarl's blog, and Sergio's blog.

Zealynx has achieved public recognition, including a Top 5 position in the Beanstalk Audit public contest, rewarded with $8k. Our ongoing commitment is to provide value through our expertise and innovative security solutions for smart contracts.

2. Disclaimer

A security review of a smart contract can’t guarantee there are no vulnerabilities. It’s a limited effort in terms of time, resources, and expertise to find as many vulnerabilities as possible. We can not assure 100% security post-review or guarantee the discovery of any issues with your smart contracts.

3. Overview

3.1 Project Summary

A time-boxed independent security assessment of the Ribbon protocol was done by Zealynx Security, with a focus on the security aspects of the application's implementation.
We performed the security assessment based on the agreed scope, following our approach and methodology. Based on our scope and our performed activities, our security assessment revealed 3 Medium, and 4 Low severity security issues.

3.2 About Ribbon

Website : https://ribbonprotocol.org/
Docs: Governance, DeFi

Ribbon Protocol is the operating system for Web3 HealthFi: a global health financing & health information system platform for tokenized primary healthcare and universal health coverage, that leverages high value health information NFTs as a secure collateral digital asset class to back public health development finance.

3.3 Audit Scope

The code under review is composed of multiple smart contracts written in Solidity language and includes 202 nLOC- normalized source lines of code (only source-code lines).
The core contracts on the scope are the following:

- points
- ribbonVault

4. Audit Methodology:

Approach

During our security assessments, we uphold a rigorous approach to maintain high-quality standards. Our methodology encompasses thorough unit and fuzz testing and meticulous manual security reviews.

Throughout the smart contract audit process, we prioritize the following aspects to uphold excellence:

1. **Code Quality:** We diligently evaluate the overall quality of the code, aiming to identify any potential vulnerabilities or weaknesses.
2. **Best Practices:** Our assessments emphasize adherence to established best practices, ensuring that the smart contract follows industry-accepted guidelines and standards.
3. **Documentation and Comments:** We meticulously review code documentation and comments to ensure they accurately reflect the underlying logic and expected behavior of the contract.

5. Severity Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

6. Executive Summary

Summary of Findings

Vulnerability	Severity
[M-1] Implement Safe Transfer Methods for ERC20 Tokens	Medium
[M-2] Incorrect handling of token decimals	Medium
[M-3] Incorrect Handling of Fee-On-Transfer Tokens in swapToPaymentCoinAdmin Function	Medium
[L-1] Single-Step Ownership Transfer Vulnerability in RibbonVault Constructor	Low
[L-2] Potential Replay Attack Prevention Issue in Signature Verification Logic	Low
[L-3] Lack of Zero address checks	Low
[L-4] Lack of input parameter validation for amount parameters	Low

Security Tests Implemented

The tests are documented within this report.

Nº	Test Name
1	test_ReplayAttackWithChangedV
2	test_ReplayAttackWithChangedR
3	test_ReplayAttackWithChangedS
4	test_ReplayAttackWithDifferentComponents
5	test_SignatureWithDifferentPrivateKey
6	test_InvalidSignature
7	testFuzz_createVault
8	testFuzz_createVaultTransfer
9	test_createVault_insufficientBalance
10	testFuzz_permitSwapToPaymentCoin_notEnoughPoints
11	testFuzz_permitSwapToPaymentCoin_freezePermit
12	testFuzz_permitSwapToPaymentCoin_lessThanMinPoints
13	testFuzz_permitSwapToPaymentCoin_invalidSignature
14	testFuzz_permitSdwapToPaymentCoin_freezePermit
15	testFuzz_RepermitRiopSwapToPaymentCoin_notEnoughPoints
16	testFuzz_withdrawfees_withNewDeposit
17	testFuzz_claimPointsAdmin_notApproved
18	testFuzz_xclaimPointsAdmin
19	testFuzz_xswapToPaymentCoinAdmin
20	testFuzz_withdrawfees_withoutNewDeposit
21	testFuzz_claimPointsAdmin_approvedAdmin

7. Findings

7.1. Medium Findings

[M-1] Implement Safe Transfer Methods for ERC20 Tokens

Summary

The error pertains to an Unchecked Transfer vulnerability in the code. This error arises when the return value of an external transfer call is not checked.

Vulnerability Description

The return value of an external **transfer** call is not checked. Several tokens do not revert in case of failure and instead return false. Without checking the return value, the contract may assume a successful transfer even if it fails, leading to potential exploits.
The protocol aims to support all ERC20 tokens but uses the original transfer functions. Some tokens, like USDT, do not implement the EIP20 standard correctly and their transfer/transferFrom function return void instead of a success boolean. Calling these functions with the correct EIP20 function signatures will revert.

Impact

Not checking the return value of transfer calls can result in incorrect balances and allow attackers to perform operations that should fail. For example, an attacker could call deposit without transferring tokens, gaining an incorrect balance. Tokens that do not correctly implement EIP20, like USDT, will revert transactions, making them unusable in the protocol.

Mitigation

Ensure return values of **transfer** calls are checked. Use libraries like **SafeERC20** or manually check the return values in the code.

- **vault.withdrawfees(address)** (src/ribbonVault.sol#145)
- **vault.claimPointsAdmin(address,uint256)** (src/ribbonVault.sol#152)
- **vault.swapToPaymentCoinAdmin(address,uint256)** (src/ribbonVault.sol#166, 167)
- **vault.permitClaimPoints(address,uint256,uint256,uint8,bytes32,bytes32)** (src/ribbonVault.sol#177)

- **vault.permitSwapToPaymentCoin(address,uint256,uint256,uint8,bytes32,bytes32)** (src/ribbonVault.sol#193, 194)
- **vault.emergencyWithdraw(address,address,uint256)** (src/ribbonVault.sol#201)

Recommendations

- Use the **SafeERC20** library to handle token transfers safely and ensure all return values are checked.
- Manually check the return values of **transfer** calls.
- Update the protocol to handle non-standard-compliant tokens using OpenZeppelin's **SafeERC20** versions with **safeTransfer** and **safeTransferFrom** functions.

[M-2] Incorrect handling of token decimals

Summary

The contract incorrectly handles ERC20 tokens with different decimal places, assuming all tokens have 18 decimals. This oversight can lead to incorrect token amounts being transferred, burned, or swapped, causing significant issues in the contract's functionality.

Vulnerability Description

The contract assumes that all ERC20 tokens have 18 decimals, which is not guaranteed. This incorrect assumption is hardcoded into various functions that handle token transfers, calculations, and conversions. This can result in incorrect token amounts being processed.

Impact

The incorrect handling of token decimals can lead to multiple vulnerabilities:

1. **Incorrect Token Amounts :** Tokens with fewer or more than 18 decimals can result in incorrect amounts being transferred or swapped, leading to financial discrepancies.
2. **Loss of Funds :** Recipients may receive fewer tokens than intended, or senders may lose more tokens than expected due to incorrect decimal handling.
3. **Contract Reversion :** Transactions may revert if they exceed the sender’s balance or the token’s total supply due to incorrect decimal calculations.
4. **Economic Exploits :** Malicious users might exploit these discrepancies to their advantage, draining tokens from the contract or manipulating balances.

5. **Inconsistent State :** The contract's internal state might become inconsistent if token balances do not align with the expected decimal precision, leading to further logical errors.

Mitigation

To address the issue of incorrect token decimal handling, ensure that the contract dynamically handles tokens according to their specific decimal places.

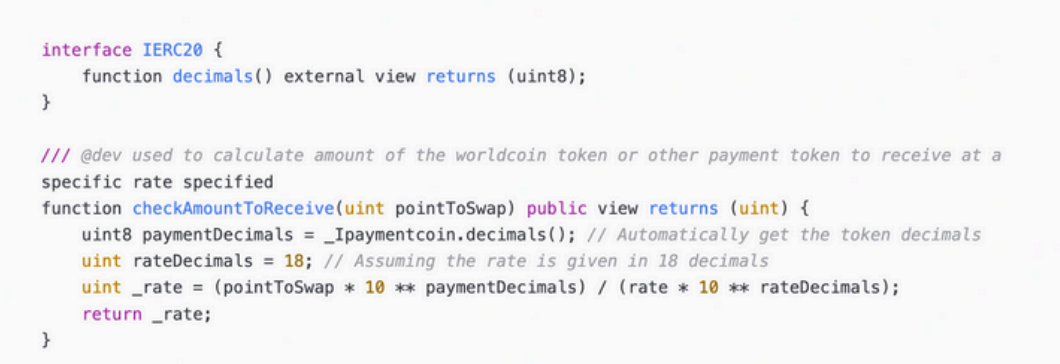
Implementation: Modify functions to automatically detect the token's decimal count and adjust calculations accordingly. This approach ensures accurate handling of different tokens.

Recommendations

- Automatically detect and handle token **decimals** using the decimals function provided by the ERC20 standard. This approach minimizes potential errors and ensures accurate and reliable operations.

Example Code Adjustment

To handle different token decimals, modify the `**checkAmountToReceive**` function to automatically get the token's decimals:



```
interface IERC20 {
    function decimals() external view returns (uint8);
}

/// @dev used to calculate amount of the worldcoin token or other payment token to receive at a
specific rate specified
function checkAmountToReceive(uint pointToSwap) public view returns (uint) {
    uint8 paymentDecimals = _Ipaymentcoin.decimals(); // Automatically get the token decimals
    uint rateDecimals = 18; // Assuming the rate is given in 18 decimals
    uint _rate = (pointToSwap * 10 ** paymentDecimals) / (rate * 10 ** rateDecimals);
    return _rate;
}
```

[M-3] Incorrect Handling of Fee-On-Transfer Tokens in swapToPaymentCoinAdmin Function

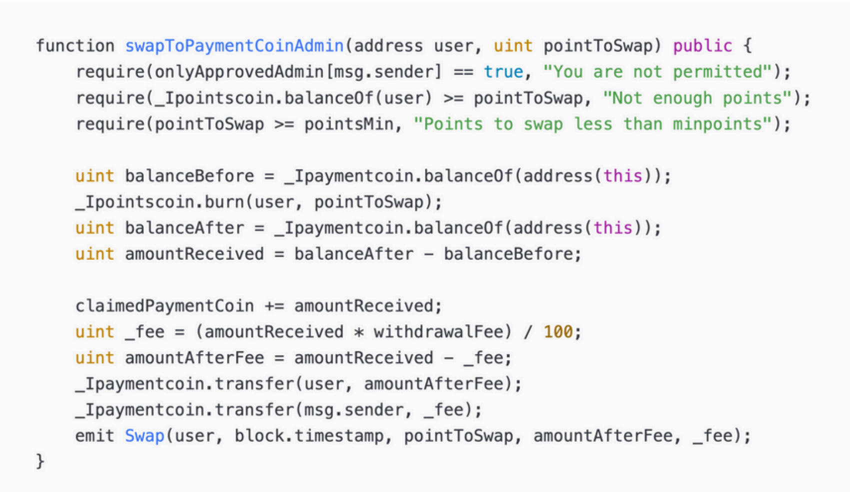
Description

The protocol intends to support all ERC20 tokens but does not currently support fee-on-transfer tokens. These tokens charge a fee on each transfer, meaning the amount received by the recipient is less than the amount sent by the sender. The current implementation of the **swapToPaymentCoinAdmin** function and similar functions in the **Vault** contract assumes that the transferred amount is received in full, which may lead to incorrect calculations and potential vulnerabilities when handling fee-on-transfer tokens.

Impact

1. **Incorrect Token Amounts** : If a token charges a transfer fee, the contract may end up with fewer tokens than expected, leading to incorrect calculations and potential financial discrepancies.
2. **Potential Exploits** : Malicious users could exploit this discrepancy to manipulate token balances in their favor, causing financial loss to the contract or other users.
3. **Operational Failures** : Functions that rely on the assumption of exact amounts being transferred may fail or behave unpredictably, affecting the contract's functionality and reliability.

Some tokens take a transfer fee (e.g., STA, PAXG), while others do not currently charge a fee but may do so in the future (e.g., USDT, USDC). (Read here about fee-on-transfer tokens)



```
function swapToPaymentCoinAdmin(address user, uint pointToSwap) public {
    require(onlyApprovedAdmin[msg.sender] == true, "You are not permitted");
    require(_Ipointscoin.balanceOf(user) >= pointToSwap, "Not enough points");
    require(pointToSwap >= pointsMin, "Points to swap less than minpoints");

    uint balanceBefore = _Ipaymentcoin.balanceOf(address(this));
    _Ipointscoin.burn(user, pointToSwap);
    uint balanceAfter = _Ipaymentcoin.balanceOf(address(this));
    uint amountReceived = balanceAfter - balanceBefore;

    claimedPaymentCoin += amountReceived;
    uint _fee = (amountReceived * withdrawalFee) / 100;
    uint amountAfterFee = amountReceived - _fee;
    _Ipaymentcoin.transfer(user, amountAfterFee);
    _Ipaymentcoin.transfer(msg.sender, _fee);
    emit Swap(user, block.timestamp, pointToSwap, amountAfterFee, _fee);
}
```

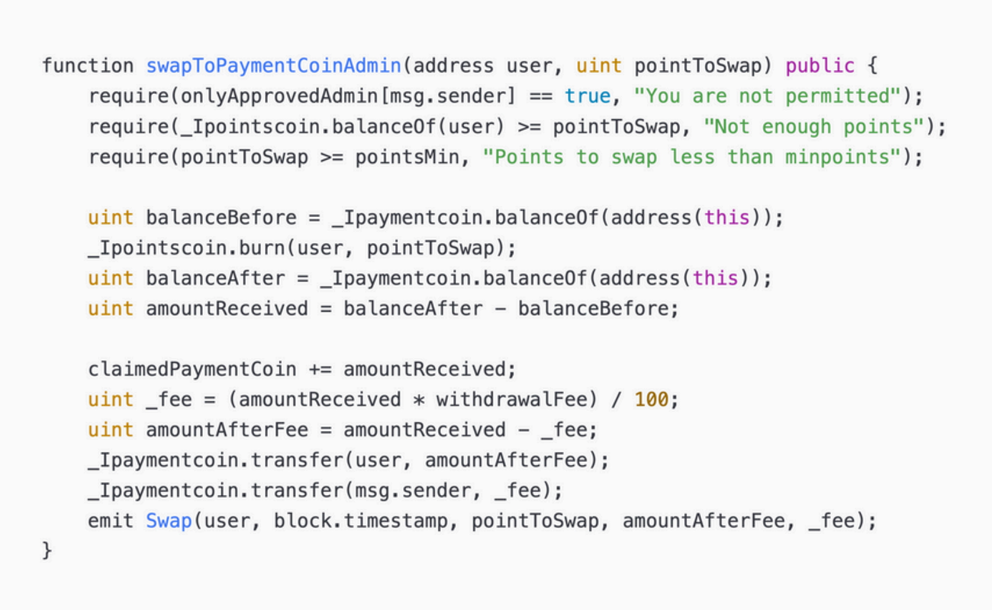
Potential Issue

If the **_Ipaymentcoin** charges a fee on transfer, **amountReceived** will be less than **pointToSwap**, causing the function to behave incorrectly. This issue arises because the implementation verifies that the transfer was successful by checking that the balance of the recipient is greater than or equal to the initial balance plus the amount transferred. This check will fail for fee-on-transfer tokens because the actual received amount will be less than the input amount.

Mitigation

To mitigate this issue, modify the function to check the contract's balance before and after the transfer to calculate the actual amount received.

Updated Function



```
function swapToPaymentCoinAdmin(address user, uint pointToSwap) public {
    require(onlyApprovedAdmin[msg.sender] == true, "You are not permitted");
    require(_Ipointscoin.balanceOf(user) >= pointToSwap, "Not enough points");
    require(pointToSwap >= pointsMin, "Points to swap less than minpoints");

    uint balanceBefore = _Ipaymentcoin.balanceOf(address(this));
    _Ipointscoin.burn(user, pointToSwap);
    uint balanceAfter = _Ipaymentcoin.balanceOf(address(this));
    uint amountReceived = balanceAfter - balanceBefore;

    claimedPaymentCoin += amountReceived;
    uint _fee = (amountReceived * withdrawalFee) / 100;
    uint amountAfterFee = amountReceived - _fee;
    _Ipaymentcoin.transfer(user, amountAfterFee);
    _Ipaymentcoin.transfer(msg.sender, _fee);
    emit Swap(user, block.timestamp, pointToSwap, amountAfterFee, _fee);
}
```

Explanation

1. **Check Balance Before Transfer** : Capture the contract's balance of the payment token before the transfer.
2. **Perform the Transfer** : Burn the points token from the user and transfer the payment token to the contract.
3. **Check Balance After Transfer** : Capture the contract's balance of the payment token after the transfer.
4. **Calculate Amount Received** : Calculate the actual amount received by the contract.
5. **Proceed with Fee Calculation and Transfer** : Calculate and transfer the fee, then transfer the remaining amount to the user.

Recommendations

1. **Calculate Actual Amount Received** : Ensure the function calculates the actual amount received by checking the balance before and after the transfer.
2. **Handle Transfer Fees** : Update the function to account for transfer fees and adjust the transferred amounts accordingly.
3. **Documentation** : Clearly document the handling of fee-on-transfer tokens to inform users and developers of the expected behavior.

Mitigation

To mitigate this issue, modify the function to check the contract's balance before and after the transfer to calculate the actual amount received.

7.2. Low Findings

[L-1] Single-Step Ownership Transfer Vulnerability in RibbonVault Constructor

Summary

Constructor Vulnerability in **RibbonVault** Contract

Vulnerability Description

The constructor in the **RibbonVault** contract inherits from OpenZeppelin's Ownable contract, which utilizes a single-step ownership transfer pattern. This can lead to a situation where, if an incorrect address is provided for the new owner, none of the onlyOwner marked methods will be callable again. This single-step transfer pattern poses a risk of permanently locking out the admin from executing essential functions if an error is made during the ownership transfer.

Impact

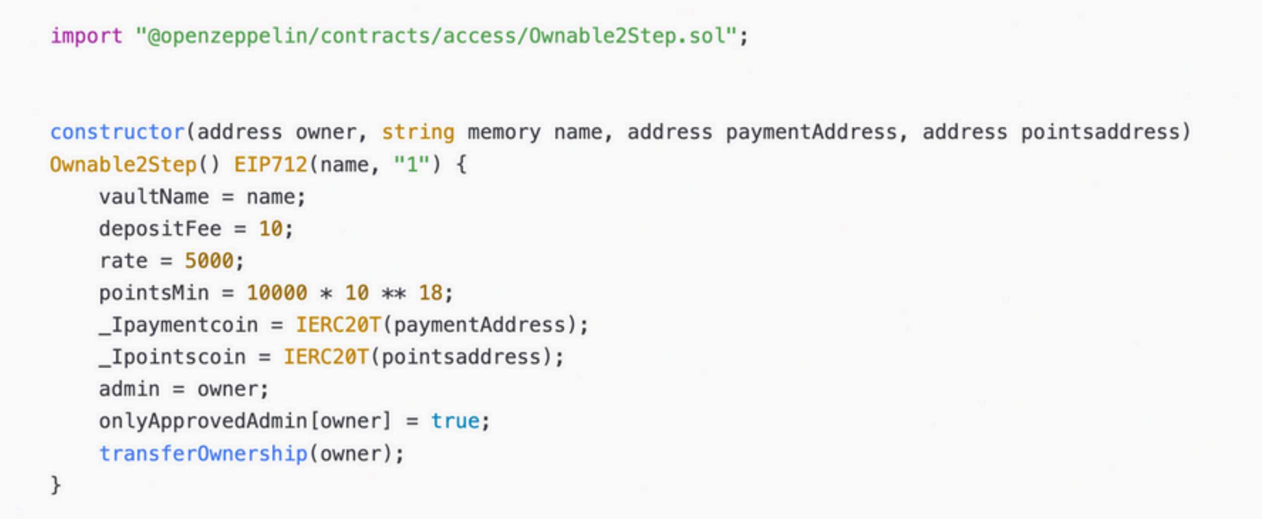
High, because it bricks core protocol functionality. If an incorrect owner address is set, the admin will lose access to all functions protected by the **onlyOwner** modifier, preventing essential administrative tasks and potentially causing significant disruption to the contract's operations.

Mitigation

Use OpenZeppelin's **Ownable2Step** contract instead of **Ownable**. The **Ownable2Step** contract implements a two-step ownership transfer process, which requires the new owner to explicitly accept the ownership transfer. This approach significantly reduces the risk of errors during the transfer process and ensures that the new owner can take over the contract management without issues.

Recommendations

Replace the inheritance of **Ownable** with **Ownable2Step** in the **RibbonVault** contract. The constructor should be updated to initiate the two-step ownership transfer process, as demonstrated below:



```
import "@openzeppelin/contracts/access/Ownable2Step.sol";


constructor(address owner, string memory name, address paymentAddress, address pointsaddress)
Ownable2Step() EIP712(name, "1") {
    vaultName = name;
    depositFee = 10;
    rate = 5000;
    pointsMin = 10000 * 10 ** 18;
    _Ipaymentcoin = IERC20T(paymentAddress);
    _Ipointscoin = IERC20T(pointsaddress);
    admin = owner;
    onlyApprovedAdmin[owner] = true;
    transferOwnership(owner);
}
```

By using **Ownable2Step**, the ownership transfer process becomes more secure, reducing the likelihood of administrative errors and ensuring the continued operability of the contract's core functionalities.

[L-2] Potential Replay Attack Prevention Issue in Signature Verification Logic

Summary

Potential Replay Attack Prevention Issue in Signature Verification Logic

Vulnerability Description

The **_permit** function in the **RibbonVault** contract utilizes the **require(sig_v[v]==false || sig_r[r] == false || sig_s[s]==false,"sig used");** condition to prevent signature replay attacks. This logic checks if any one of the **v**, **r**, or **s** parameters of the signature has been used before. While this condition appears to prevent signature reuse, it is not the most robust approach as it only requires one of the conditions to be false for the signature to be accepted, which can potentially lead to unforeseen issues in certain edge cases.

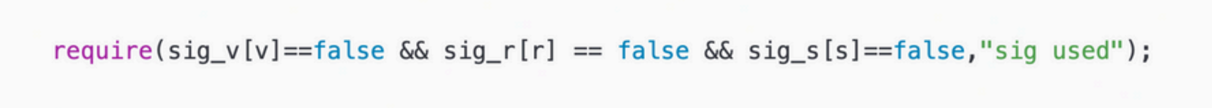
Impact

Low Impact:

- The current implementation passes all standard tests and does not exhibit immediate vulnerabilities. However, the use of **||** instead of **&&** in the replay prevention logic is not a best practice and might lead to potential risks in more complex scenarios or under specific conditions.
- Potential future issues could arise if this logic is exploited in unforeseen ways, potentially allowing partial signature reuse under very specific circumstances.

Mitigation

To enhance the robustness of the signature replay prevention mechanism, it is recommended to use a stricter condition that ensures all parts of the signature (**v**, **r**, **s**) are unique before accepting the transaction. This can be achieved by modifying the condition to use **&&** instead of **||**:



```
require(sig_v[v]==false && sig_r[r] == false && sig_s[s]==false,"sig used");
```

[L-3] Lack of Zero address checks

Description

The contract **Points** lacks zero address checks for several key parameters and function inputs. Specifically, functions like **constructor**, **setVaultAdmin**, **setApproveToBurn**, **mint**, **transferToVault**, and **createVault** do not validate that the provided addresses are non-zero. Allowing zero addresses can lead to unintended behavior or potential vulnerabilities in the contract's operation.

Impact

The absence of zero address checks can lead to multiple issues, including:

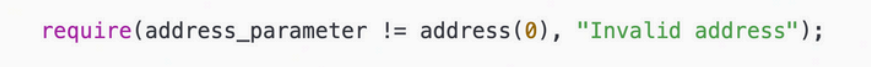
1. **Loss of Tokens**: Functions like **mint** and **transferToVault** could potentially mint or transfer tokens to the zero address, resulting in the permanent loss of those tokens.

2.**Privilege Escalation**: Setting administrative roles or approval addresses to the zero address might disrupt the intended access control, allowing unauthorized entities to perform restricted actions.

3. **Operational Disruptions:** Using zero addresses in critical mappings (like **approveToBurn**) can lead to logical errors and unexpected behavior in contract operations.

Mitigation

To mitigate the risk associated with zero address inputs, it is essential to implement checks that ensure all provided addresses are non-zero. This can be achieved by adding a simple require statement in the relevant functions. Below are the modifications required:



```
require(address_parameter != address(0), "Invalid address");
```

[L-4] Lack of input parameter validation for amount parameters

Description

The **Points** and **Vault** contracts lack input validation for several parameters, specifically ensuring that amount-related parameters (such as token amounts, fees, and rates) are within valid ranges. This oversight can lead to minor operational inefficiencies, unnecessary gas costs, and potential logical inconsistencies.

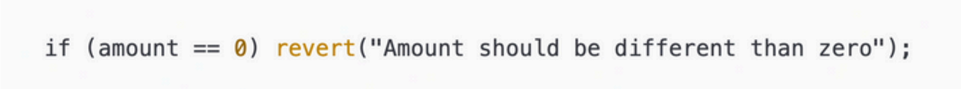
Impact

While the immediate impact of this lack of input validation is relatively low, it can lead to several issues:

1. Operational Inefficiencies: Unchecked zero amounts can result in unnecessary transactions, consuming extra gas without performing meaningful operations.
2. Minor Logical Inconsistencies: Functions performing actions with zero amounts might lead to minor inconsistencies or state changes that do not align with the intended logic.
3. Usability Concerns: Incorrect fees or rates, if not validated, can make the contract less user-friendly or cause unexpected behaviors that may confuse users.

Mitigation

Implement input validation for these parameters to ensure they are different than zero.



```
if (amount == 0) revert("Amount should be different than zero");
```