Ribbon Protocol
Security Assessment Report

Date: July, 2024
Prepared by: Gecko Security
Version 1.0

Contents

About Gecko Security

Gecko Security is a vulnerability research firm specializing in blockchain security, including EVM, Solana, and Stacks. We assess L1s and L2s, cross-chain protocols, wallets, applied cryptography, zero-knowledge circuits, and web applications. Our team consists of security re- searchers and software engineers with backgrounds ranging from intelligence agencies to companies like Binance.

Gecko aims to treat clients on a case-by-case basis and to consider their individual, unique business needs. Our services include comprehensive smart contract audits, smart contract fuzzing and formal verification. Our goal is to see the long-term success of our partners rather than simply provide a list of present security issues. We believe in transparency and actively contribute to the community creating open-source security tools and educational content.

Disclaimer

This assessment does not guarantee the discovery of all possible issues within its scope, nor does it ensure the absence of future issues. Additionally, Gecko Security Inc. cannot guarantee the security of any code added to the project after our assessment. Since a single assessment is never comprehensive, we recommend conducting multiple independent assessments and implementing a bug bounty program.

For each finding, Gecko Security Inc. provides a recommended solution. These recommendations illustrate potential fixes but may not be tested or functional. These recommendations are not exhaustive and should be considered a starting point for further discussion. We are available to provide additional guidance and advice as needed.

Overview

Project Summary

About Ribbon

A time-boxed independent security assessment of the Ribbon protocol was done by Gecko 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.

Website: https://ribbonprotocol.org/

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.

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

Audit Methodology

Approach

During a security assessment, Gecko works through standard phases of security auditing, including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope. Gecko Security was provided with the source code and documentation that outlines the expected behavior. Our auditors spent 1 week auditing the source code provided, which includes understanding the context of use, analyzing the boundaries of the expected behavior of each contract and function, understanding the implementation by the development team (including dependencies beyond the scope to be audited) and identifying possible situations in which the code allows the caller to reach a state that exposes some vulnerability.

Audit 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.

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.

Affected Functions

- `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

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

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

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

- 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.
- Potential Exploits: Malicious users could exploit this discrepancy to manipulate token balances in their favor, causing financial loss to the contract or other users.
- 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.

Potential Issue

If the `_paymentcoin` charges a fee on transfer, `amountReceived` will be less than `pointsToSwap`, 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

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

Recommendations

- Calculate Actual Amount Received: Ensure the function calculates the actual amount received by checking the balance before and after the transfer.

- Handle Transfer Fees: Update the function to account for transfer fees and adjust the transferred amounts accordingly.
- Documentation: Clearly document the handling of fee-on-transfer tokens to inform users and developers of the expected behavior.

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:

[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 `||`.

[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:

- 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.
- 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.
- 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:

[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:

- Operational Inefficiencies: Unchecked zero amounts can result in unnecessary transactions, consuming extra gas without performing meaningful operations.
- 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.
- 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.