VIBRANIUM AUDITS

Security Assessment

shark roll

Vibranium Audits Verified on Oct 30th, 2024

Summary

Finding

DISCLAIMER

Vibranium audits verified on Nov 16th 2024

shark roll

The security assessment was prepared by Vibranium audits , the leader in Web3 security

Executive Summary

TYPES	ECOSYSTEM	METHODS
DEFI	Binance smart chain / EVM	Formal Verification, Manual review

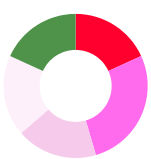
LANGUAGE

Solidity

Codebase

https://bscscan.com/address/0xC4aF96932c2A9c5BB63dEF288aE147265d46e703#code
updated version : https://sepolia.etherscan.io/address/0xC4aF96932c2A9c5BB63dEF288aE147265d46e703#code

Vulnerability Summary



4	0	2	2	0	0
Total Findings	Critical	High	Medium	Minor	Informational

High	Resolved	Storage Inflation in pendingWithdrawals: msg.value is added without validating the actual balance, allowing inflation of storage and potential gas exhaustion, locking funds or blocking withdrawals.
High	Resolved	Logic Lockup in withdrawPendingNative: Inflated pendingWithdrawals prevents successful withdrawal execution, leaving legitimate balances locked in the contract.
Medium	Resolved	Unbounded Loops in createSubWallets: Unchecked loops in createSubWallets and withdrawERC20 may cause gas exhaustion for large iterations, leading to transaction failure.
Medium	Resolved	Ownership Overreach: onlyOwner modifier is widely used. If the owner's private key is compromised, it grants full control over funds and critical operations.

Codebase | **shark roll**

Address

https://bscscan.com/address/0xC4aF96932c2A9c5BB63dEF288aE147265d46e703#code
updated version : https://sepolia.etherscan.io/address/0xC4aF96932c2A9c5BB63dEF288aE147265d46e703#code

Approach & Methods | shark roll

This report has been prepared for shark roll to identify potential issues and vulnerabilities in the source code of the project, including smart contracts and the token minting mechanism. A comprehensive examination has been conducted, utilizing Static Analysis and Manual Review techniques to ensure compliance with best security practices.

The auditing process pays special attention to the following considerations:

Testing the smart contracts against common and uncommon attack vectors such as reentrancy attacks, overflow vulnerabilities, and unauthorized access.

Assessing the codebase for compliance with industry best practices, ensuring code security, scalability, and maintainability.

Ensuring contract logic meets the specifications and requirements of the shark roll project, specifically for token minting, access control, and transaction handling.

Cross-referencing contract structure and functionality against similar projects in the blockchain ecosystem to ensure adherence to standard practices.

Performing a line-by-line manual review of the entire codebase to identify any hidden or obscure vulnerabilities that automated tools may miss.

The security assessment identified vulnerabilities across a range of severity levels, from Critical to Informational. We recommend addressing these findings promptly to enhance the overall security of the shark roll project. Key recommendations include:

Testing smart contracts against both common and uncommon attack vectors, ensuring robustness against real-world threats.

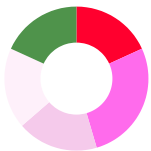
Implementing secure coding practices to enhance the quality and security of the codebase.

Expanding unit tests to cover all possible use cases and edge cases, especially for the token minting functions.

Improving code documentation and comments for better readability, particularly for critical functions involving token minting and transfers.

Ensuring transparency in privileged activities, such as the minting process, and implementing additional safeguards for authorization.

shark roll | Finding



4	0	2	2	0	0
Total Findings	Critical	High	Medium	Minor	Informational

This report provides a detailed audit of the shark roll project's codebase, focusing on the token contract . The objective of this audit is to uncover potential security vulnerabilities, optimize code efficiency, and identify areas for improvement to strengthen the project's security and reliability.
In total, four vulnerabilities were identified, categorized by severity as follows: 2 High and 2 Medium .

ID	Title	Category	Severity	Status
SH-01	Storage Inflation in pendingWithdrawals	Data Validation & Storage	High	Resolved
SH-02	Logic Lockup in withdrawPendingNative	Withdrawal Execution Logic	High	Resolved
SH-03	Unbounded Loops in createSubWallets	Gas Efficiency & Scalability	Medium	Resolved
SH-04	Ownership Overreach	Access Control & Authorization	Medium	Resolved

SH-01 | Storage Inflation in pendingWithdrawals

Category	Severity	Location	Status
Data Validation & Storage	High	Contract SubWallet / function receive	Resolved

Description

The receive() function in the SubWallet contract adds msg.value directly to the pendingWithdrawals mapping without validating the actual balance of the contract. An attacker can exploit this by sending artificially large values in a transaction, causing the pendingWithdrawals value to grow uncontrollably. This bloats storage and leads to excessive gas costs or even failure in executing subsequent withdrawals.

Mitigation

Validate Ether Balance Before Updating Storage:

- Ensure the amount added to pendingWithdrawals aligns with the actual balance received by the contract.
- Example implementation:

```
uint256 currentBalance = address(this).balance;
require(currentBalance >= msg.value, "Invalid Ether value");
```

Set Limits on Pending Amounts:

- Add a maximum cap to the pendingWithdrawals value to avoid indefinite growth.

SH-02 | Logic Lockup in withdrawPendingNative

Category	Severity	Location	Status
Withdrawal Execution Logic	High	Contract SubWallet / function receive	Resolved

Description

The vulnerability stems from the receive() function, which adds the msg.value to the pendingWithdrawals[parentContract] mapping without validating whether the contract's balance can actually cover this amount. An attacker can send an artificially large msg.value to inflate the pendingWithdrawals.
When the admin attempts to withdraw using the withdrawPendingNative function:

1. The function tries to transfer the inflated amount to the parentContract.
2. The transfer fails because the contract's actual Ether balance is insufficient to cover the amount.
3. The failed transfer adds the same pendingWithdrawals amount back to the mapping.
4. This creates a loop where the pendingWithdrawals amount grows with every failed attempt, making the function unusable and locking legitimate funds in the contract indefinitely.

Mitigation

Validate msg.value in receive() and in withdrawPendingNative:

- Ensure that the amount being added to pendingWithdrawals corresponds to the actual balance received by the contract. Reject transactions where the Ether balance does not match the expected msg.value.

SH-03 | Unbounded Loops in createSubWallets

Category	Severity	Location	Status
Gas Efficiency & Scalability	Medium	Main contract / createSubWallets function	Resolved

Description

The createSubWallets function contains an unbounded loop that iterates count times to create new sub-wallets. If the count parameter is large, the gas required for execution can exceed the block gas limit, causing the transaction to fail. This makes the function unusable for large batch operations and can potentially block the contract owner from scaling sub-wallet creation as intended.

Mitigation

Limit the Number of Sub-Wallets Per Transaction:

- Enforce a maximum value for count to ensure the function operates within safe gas limits.

Implement Batched Execution:

- If creating more sub-wallets is required, split the operation into multiple transactions.

SH-04 | Ownership Overreach

Category	Severity	Location	Status
Access Control & Authorization	Medium	Main contract	Resolved

Description

The contract relies heavily on the onlyOwner modifier for critical operations such as:

- Creating sub-wallets (createSubWallets).
- Withdrawing Ether and ERC20 tokens.
- Performing token swaps (swapTokens).
- Managing sub-wallet interactions.

If the owner's private key is compromised, the attacker gains full control over all operations, including the ability to drain funds, perform unauthorized swaps, or disrupt the functionality of the system. This introduces a single point of failure, jeopardizing the entire contract's security.

Mitigation

Introduce Role-Based Access Control (RBAC):

- Use libraries like OpenZeppelin’s AccessControl to define roles for different operations. For instance:
  - ADMIN_ROLE: Can manage sub-wallets.
  - FINANCE_ROLE: Can withdraw funds or perform token swaps.

DISCLAIMER | VIBRANIUM AUDITS

This security assessment has been prepared by Vibranium Audits for the shark roll project to identify potential vulnerabilities in the project's codebase. Vibranium Audits applied its expertise in Web3 and smart contract security to perform a detailed analysis of the project, utilizing a combination of manual code review, automated tools, and industry best practices.

This report represents Vibranium Audits' assessment as of the verification date (November 16th, 2024). The findings and recommendations provided here in reflect the state of the project at the time of the audit. As blockchain technology and associated security risks continuously evolve, we strongly recommend ongoing monitoring, upgrades, and reassessments to maintain a robust security posture.

Vibranium Audits assumes no responsibility for any losses, damages, or consequences, whether financial or otherwise, resulting from the use, misuse, or reliance on this report. This document is intended solely for informational purposes and should not be construed as financial, legal, or investment advice, nor as an endorsement or guarantee of the project.

Vibranium Audits Securing the Web3 World

Vibranium Audits is a blockchain security company that was founded in 2021 by professors from the University of Greenwich and cyber-security engineers from ITI Capital. As pioneers in the field, Vibranium Audits utilizes best-in-class Formal Verification and AI technology to secure and monitor blockchains, smart contracts, and Web3 apps.