Ferum

Security Assessment

March 20th, 2024 — Prepared by OtterSec

Matteo Oliva matt@osec.io

Table of Contents

01 — Executive Summary

Overview

Ferum Labs engaged OtterSec to assess the `blackwing-EVM-contracts` programs. This assessment was conducted between March 11th and March 15th, 2024. For more information on our auditing methodology, refer to Appendix B.

Key Findings

We produced 4 findings throughout this audit engagement.

In particular, We identified a high-risk vulnerability related to the possibility of front-running in the deposit functionality. This arises due to the formula used to calculate shares for the first depositor, allowing an attacker to steal a portion of the first deposit by inflating the share value (OS-FRM-ADV-00). Additionally, we highlighted another issue where users may bypass withdrawal time locks by depositing assets, transferring their shares, and immediately withdrawing, thus circumventing the intended delay (OS-FRM-ADV-01). We also pointed out the ability to bypass the proxy interface and directly initialize the implementation contracts (OS-FRM-ADV-02).

We also made recommendations for modifying the code base to enhance its security and efficiency (OS-FRM-SUG-00).

02 — Scope

The source code was delivered to us in a Git repository at https://github.com/ferumlabs/evm-contracts. This audit was performed against commit 1ff330b.

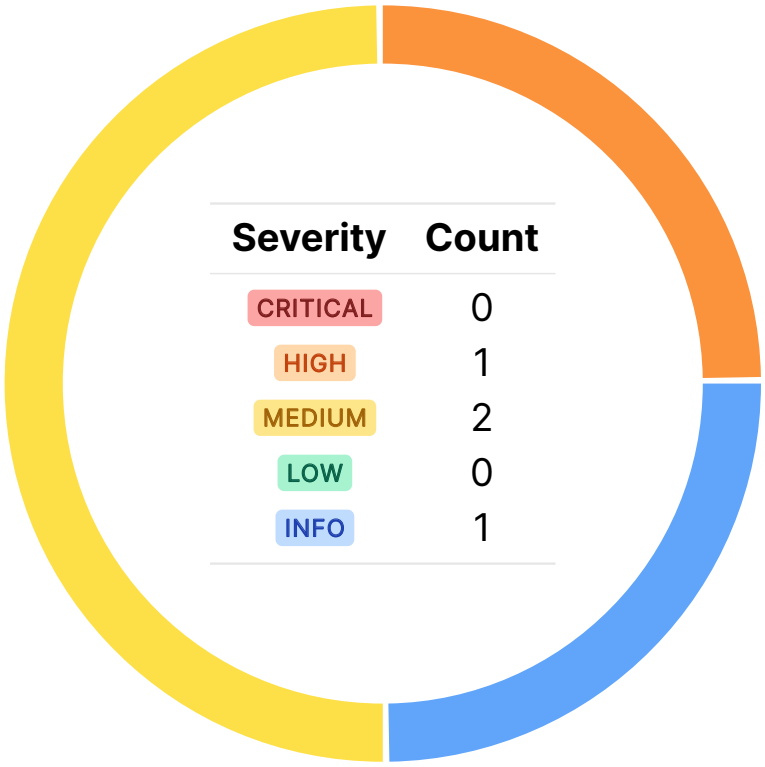
A brief description of the programs is as follows:

Name	Description
blackwing-EVM-ontracts	Allows users to deposit ERC20 tokens into a vault and receive vault tokens representing their share of the assets. It facilitates access control mechanisms and utility functions to facilitate deposits, withdrawals, and asset management.

03 — Findings

Overall, we reported 4 findings.

We split the findings into **vulnerabilities** and **general findings**. Vulnerabilities have an immediate impact and should be remediated as soon as possible. General findings do not have an immediate impact but will aid in mitigating future vulnerabilities.



Severity	Count
CRITICAL	0
HIGH	1
MEDIUM	2
LOW	0
INFO	1

04 — Vulnerabilities

Here, we present a technical analysis of the vulnerabilities we identified during our audit. These vulnerabilities have *immediate* security implications, and we recommend remediation as soon as possible.

Rating criteria can be found in Appendix A.

ID	Severity	Status	Description
OS-FRM-ADV-00	HIGH	RESOLVED	`deposit` is vulnerable to front-running due to the formula used to calculate shares for the first depositor, potentially enabling an attacker to steal a portion of the first deposit by inflating the share value.
OS-FRM-ADV-01	MEDIUM	TODO	Users may bypass withdrawal time locks in `withdraw` by depositing assets, transferring their shares, and immediately withdrawing, thus circumventing the intended delay.
OS-FRM-ADV-02	MEDIUM	RESOLVED	The implementation contracts lack `_disableInitializers()`, enabling the bypass of the proxy layer to directly interact with the implementation contracts.

Front-Running In Deposit Functionality HIGH OS-FRM-ADV-00

Description

The vulnerability lies in the initial calculation of vault tokens to mint for the first depositor in `deposit` within `BlackwingVault`. It allows the first depositor to potentially exploit the system by front-running with a very small deposit amount and inflating their share of the vault. Since the total supply of vault tokens (`totalVaultTokenSupply`) is initially zero, the formula calculates the vault tokens to mint (`vaultTokensToMint`) by multiplying the small amount deposited by `INITIAL_LIQUIDITY_MULTIPLIER` , artificially inflating the attacker's share of the vault.

>_ *evm/launch_vault/vault.sol* rust

```rust
function deposit(IERC20 asset, uint amount) public {
    requireAssetRegistered(asset);

    PoolInfo memory pool = pools[asset];
    uint totalVaultTokenSupply = pool.vaultToken.totalSupply();
    uint vaultTokensToMint = 0;

    if (totalVaultTokenSupply == 0) {
      vaultTokensToMint = amount * INITIAL_LIQUIDITY_MULTIPLIER;
    } [...]

    emit BalanceChange(true, address(asset), msg.sender, amount);
}
```

Legitimate users depositing larger amounts after the attacker may receive a smaller share of the vault than expected, leading to a loss of funds as the attacker effectively steals a portion of the vault's assets. However, due to the time lock mechanism, legitimate users may not immediately withdraw their funds after depositing, reducing the effectiveness of the attack since the attacker cannot immediately withdraw the inflated share.

Remediation

Initialize the vault with funds in the initial transaction. This approach prevents the artificial inflation of the initial depositor's share by pre-loading the vault with assets.

Patch

Fixed by adding a deposit on pool initialization.

LP Transfer Timelock Circumvention MEDIUM OS-FRM-ADV-01

Description

In `withdraw`, circumventing withdrawal time locks may occur when liquidity provider transfers are enabled. `withdraw` verifies that the user has waited for the required minimum number of blocks since their last deposit before allowing withdrawal. This is intended to prevent users from front-running withdrawals. However, this check relies solely on the last deposit block recorded for the user (`user.lastDepositBlock`).

>_ *evm/launch_vault/vault.sol* rust

```
function withdraw(IERC20 asset, uint vaultTokensToBurn) public {
    requireAssetRegistered(asset);
    require(withdrawsEnabled, WITHDRAWS_DISABLED_ERR);

    UserInfo memory user = users[msg.sender];
    require(block.number > user.lastDepositBlock + minBlocksSinceLastDeposit,
        ↪  MIN_BLOCKS_SINCE_LAST_DEPOSIT_ERR);
    [...]
}
```

Therefore, when liquidity provider transfers are enabled, users may deposit assets into the vault, transfer their liquidity provider shares to a new address, and immediately withdraw from the new address. This allows users to bypass any withdrawal time locks in place since the withdrawal is not made from the original address that deposited the assets.

Remediation

Check whether a user already exists in the system by verifying if `user.isValue == true`.

This has been acknowledged as an accepted risk.

Implementation Contract Takeover MEDIUM OS-FRM-ADV-02

Description

The implementation contracts lack the call to `_disableInitializers()` within the constructor. This omission allows unauthorized initializations of the implementation contract, bypassing the proxy interface. Unauthorized direct interactions with the implementation contract may compromise system integrity and upgradeability, potentially resulting in discrepancies between the proxy and its implementations in terms of state and behavior.

Remediation

Include a constructor with a call to `_disableInitializers()` to prevent initializations at the implementation level.

Patch

Fixed in 0447743.

05 — General Findings

Here, we present a discussion of general findings during our audit. While these findings do not present an immediate security impact, they represent anti-patterns and may result in security issues in the future.

ID	Description
OS-FRM-SUG-00	Recommendations for modifying the code base to enhance its security and efficiency.

Code Refactoring OS-FRM-SUG-00

Description

1. Improve error identification within the system by assigning a unique number to each error and categorizing them based on the module they belong to. This will simplify troubleshooting and debugging processes by providing clear and structured error messages that allow easy identification of the source and nature of errors
2. When creating a new `VaultToken`, include a definition for the number of decimals. While this omission does not directly pose a security threat because calculations typically do not involve decimals, it is crucial for consistency, especially for liquidity providers dealing with non-standard tokens like `USDC`, which may have different decimal representations. Including a definition for decimals ensures uniformity and clarity in token representations, particularly in scenarios involving liquidity providers who need to interact with tokens consistently across the system.

Remediation

Implement the above-mentioned suggestions.

A — Vulnerability Rating Scale

We rated our findings according to the following scale. Vulnerabilities have immediate security implications. Informational findings may be found in the General Findings.

CRITICAL

Vulnerabilities that immediately result in a loss of user funds with minimal preconditions.

Examples:

- Misconfigured authority or access control validation.
- Improperly designed economic incentives leading to loss of funds.

HIGH

Vulnerabilities that may result in a loss of user funds but are potentially difficult to exploit.

Examples:

- Loss of funds requiring specific victim interactions.
- Exploitation involving high capital requirement with respect to payout.

MEDIUM

Vulnerabilities that may result in denial of service scenarios or degraded usability.

Examples:

- Computational limit exhaustion through malicious input.
- Forced exceptions in the normal user flow.

LOW

Low probability vulnerabilities, which are still exploitable but require extenuating circumstances or undue risk.

Examples:

- Oracle manipulation with large capital requirements and multiple transactions.

INFO

Best practices to mitigate future security risks. These are classified as general findings.

Examples:

- Explicit assertion of critical internal invariants.
- Improved input validation.

B — Procedure

As part of our standard auditing procedure, we split our analysis into two main sections: design and implementation.

When auditing the design of a program, we aim to ensure that the overall economic architecture is sound in the context of an on-chain program. In other words, there is no way to steal funds or deny service, ignoring any chain-specific quirks. This usually requires a deep understanding of the program's internal interactions, potential game theory implications, and general on-chain execution primitives.

One example of a design vulnerability would be an on-chain oracle that could be manipulated by flash loans or large deposits. Such a design would generally be unsound regardless of which chain the oracle is deployed on.

On the other hand, auditing the program's implementation requires a deep understanding of the chain's execution model. While this varies from chain to chain, some common implementation vulnerabilities include reentrancy, account ownership issues, arithmetic overflows, and rounding bugs.

As a general rule of thumb, implementation vulnerabilities tend to be more "checklist" style. In contrast, design vulnerabilities require a strong understanding of the underlying system and the various interactions: both with the user and cross-program.

As we approach any new target, we strive to comprehensively understand the program first. In our audits, we always approach targets with a team of auditors. This allows us to share thoughts and collaborate, picking up on details that the other missed.

While sometimes the line between design and implementation can be blurry, we hope this gives some insight into our auditing procedure and thought process.