Ember Protocol

Security Assessment

September 9th, 2025 — Prepared by OtterSec

Michał Bochnak embe221ed@osec.io

Thiago Tavares thitav@osec.io

Table of Contents

01 — Executive Summary

Overview

Bluewater Labs and Upshift engaged OtterSec to assess the `ember-vaults` program. This assessment was conducted between August 6th and August 15th, 2025. For more information on our auditing methodology, refer to Appendix B.

Key Findings

We produced 13 findings throughout this audit engagement.

In particular, we identified a vulnerability where the vault burns withdrawal shares before checking for blacklisted users, allowing the operator to blacklist after a request, resulting in irreversible loss of funds (OS-BUV-ADV-02). Additionally, the vault's share-to-asset and asset-to-share conversions utilize integer math without guarding against zero-rounding, allowing scenarios where shares are burned or assets taken without compensation, or where shares may be minted for free, resulting in direct fund loss or asset inflation (OS-BUV-ADV-06). Furthermore, when charging platform fee, the function may calculate a fee amount of zero due to integer truncation, which still triggers the 24-hour cooldown and prevents the operator from charging again until the next day (OS-BUV-ADV-05).

We also made recommendations regarding modifications to the codebase for better functionality and to mitigate potential security issues (OS-BUV-SUG-03), and suggested the need to ensure adherence to coding best practices (OS-BUV-SUG-05). Moreover, we advised incorporating additional checks within the codebase for improved robustness and security (OS-BUV-SUG-02), and certain optimizations to reduce overall redundancy and improve clarity (OS-BUV-SUG-04).

02 — Scope

The source code was delivered to us in a Git repository at https://github.com/fireflyprotocol/ember-vaults. This audit was performed against 93a46fb.

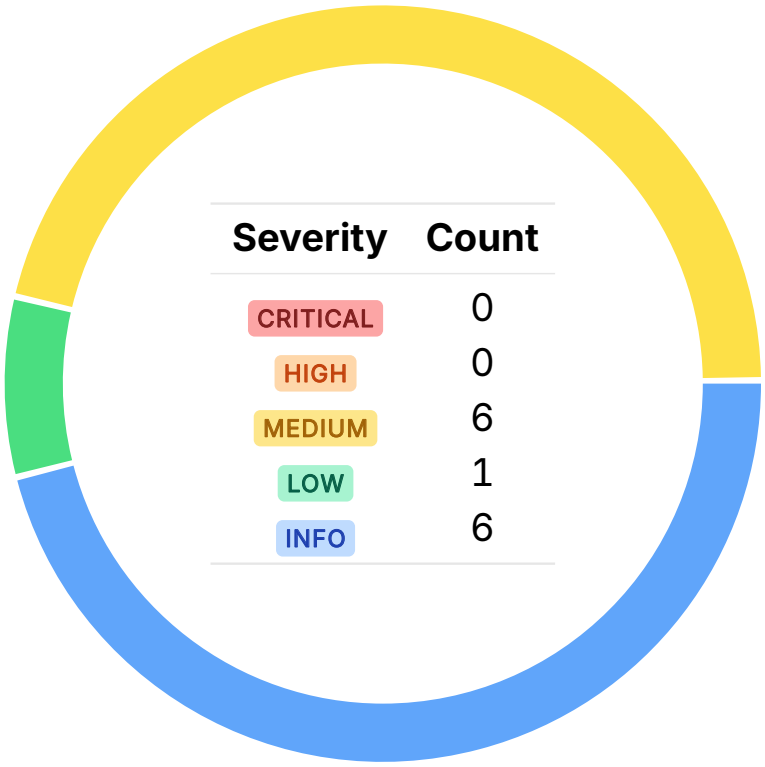
A brief description of the program is as follows:

Name	Description
ember-vaults	A single-asset DeFi vault system on Sui that lets users deposit assets, earn yield at configurable rates, and withdraw through a managed FIFO queue. It features role-based access control, automated fee accrual, and safety tools like pausing, blacklisting, and rate limits to ensure secure and efficient vault operations.

03 — Findings

Overall, we reported 13 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	0
MEDIUM	6
LOW	1
INFO	6

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-BUV-ADV-00	MEDIUM	RESOLVED	`create_vault` does not ensure `treasury_cap.total_supply == 0`, allowing receipt tokens to be minted before vault creation.
OS-BUV-ADV-01	MEDIUM	RESOLVED	If `min_withdrawal_shares` is set to zero, users may create zero-share withdrawal requests. These requests are removed immediately, leaving no account state, which results in `process_request` to abort.
OS-BUV-ADV-02	MEDIUM	RESOLVED	The vault burns withdrawal shares before checking for blacklisted users, allowing the operator to blacklist after a request, resulting in irreversible loss of funds.
OS-BUV-ADV-03	MEDIUM	RESOLVED	`process_request` may compute a zero `withdraw_amount` due to vault rate, preventing the operator from progressing through the withdrawal queue, creating a denial-of-service risk.
OS-BUV-ADV-04	MEDIUM	RESOLVED	The fee accrual mechanism may be exploited by withdrawing before accrual, reducing fee liability.

OS-BUV-ADV-05	MEDIUM	RESOLVED	`charge_platform_fee` may calculate a `fee_amount` of zero due to integer truncation, which still triggers the 24-hour cooldown and prevents the operator from charging again until the next day.
OS-BUV-ADV-06	LOW	RESOLVED	The vault's share-to-asset and asset-to-share conversions utilize integer math without guarding against zero-rounding, allowing scenarios where shares are burned or assets taken without compensation, or where shares may be minted for free, resulting in direct fund loss or asset inflation.

Risk of Uninitialized Treasury Cap MEDIUM OS-BUV-ADV-00

Description

`vault::create_vault` does not validate that `treasury_cap.total_supply == 0` when initializing a new vault. Without this check, shares may be minted before the vault is officially created, allowing tokens to exist without backing assets and breaking the invariant that shares must always correspond to deposited assets.

Remediation

Assert that the `treasury_cap.total_supply` is zero during vault creation.

Patch

Resolved in PR#16.

Zero-Share Withdrawal Requests MEDIUM OS-BUV-ADV-01

Description

`vault::create_vault` allows `min_withdrawal_shares = 0`. In this case, a user may create a redeem request with `shares_to_redeem = 0`, which passes validation in `redeem_shares`. This results in the creation of an `account_state` entry that is immediately removed in `update_account_state` since its total pending shares remain zero. When the operator later processes the withdrawal queue, `process_request` tries to borrow the user's account state, but it no longer exists, aborting the function.

Remediation

Validate that `min_withdrawal_shares > 0` in `create_vault`.

Patch

Resolved in PR#16.

Potential for Fund Loss Due to Post-Burn Blacklist Check MEDIUM OS-BUV-ADV-02

Description

In `vault::process_request`, withdrawal shares are burnt before checking if the request's owner or receiver is blacklisted. If a user submits a withdrawal request and is later blacklisted before the request is processed, the vault will burn their shares but skip transferring the underlying assets. This implies the user permanently loses the value of those shares with no compensation, effectively allowing the vault operator to confiscate funds by blacklisting after the fact. Since the blacklist check occurs after the burn, there is no safeguard against this loss.

>_ sources/vault.move RUST

```
fun process_request<T,R>(vault: &mut Vault<T,R>, request: &WithdrawalRequest, current_time: u64,
    ↪  ctx: &mut TxContext): (bool, u64, u64){
    [...]
    // if the owner or receiver is blacklisted, the request is invalid and the request is
        ↪  skipped
    // TODO: should the shares be returned to the owner? or should they be burnt? currently they
        ↪  are burnt
    // burn the shares
    burn_shares(vault, request.shares, ctx);
    update_account_pending_shares_to_withdraw(vault, request.owner, request.shares, false);
    if(is_blacklisted(vault, request.owner) || is_blacklisted(vault, request.receiver)){
        skipped = true;
        withdraw_amount = 0;
    }
    [...]
}
```

Remediation

Perform the blacklist check before burning, skip such requests entirely, or return the shares instead of burning them so the user may reclaim them if they are ever removed from the blacklist.

Patch

Resolved in e8187e9.

Denial of Service in Withdrawal Request Processing MEDIUM OS-BUV-ADV-03

Description

In `vault::process_request`, if the calculated `withdraw_amount` is zero (due to integer truncation when shares are too small compared to the vault rate), the function aborts. While this prevents burning user shares without transferring assets, it introduces a denial-of-service risk because withdrawal requests are processed strictly in queue order. A single zero-withdrawal request will block the operator from processing any subsequent valid requests.

Remediation

Ensure the requests are gracefully skipped by returning the shares to the user and continuing with the next request.

Patch

Resolved in 570c515.

Inaccurate Fee Accrual Basis MEDIUM OS-BUV-ADV-04

Description

Fee accrual currently utilizes `vault::get_vault_total_shares_in_circulation`, which excludes shares pending withdrawal. This allows users to request withdrawals right before fee accrual (as it is manually triggered by the vault operator), lowering the circulating supply and reducing the fee accrual.

>_ *sources/vault.move* RUST

```
public fun get_vault_total_shares_in_circulation<T,R>(vault: &Vault<T,R>): u64 {
    coin::total_supply<R>(&vault.receipt_token_treasury_cap) -
        ↪  balance::value(&vault.pending_shares_to_burn)
}
```

Remediation

Utilize `vault::get_vault_total_shares`, which is suitable for the current model of manual fee accrual.

Patch

Resolved in 570c515.

Risk of Lockout With Zero Fee Charged MEDIUM OS-BUV-ADV-05

Description

In `vault::charge_platform_fee`, the `fee_amount` is calculated as `tvl * vault.fee_percentage` but there is no check to ensure it is greater than zero. If the vault has a very low `tvl` or the `fee_percentage` is small enough, this multiplication may yield zero. When this occurs, no fee is accrued, but the function still updates `last_charged_at` to the current time. Since the function enforces a 24-hour interval between fee charges, the operator becomes locked out from trying again until this interval elapses. This effectively wastes the operator's charging window, delaying fee accrual for the platform.

>_ sources/vault.move RUST

```
public fun charge_platform_fee<T,R>(clock: &Clock, vault: &mut Vault<T,R>, config:
    ↪  &ProtocolConfig, ctx: &mut TxContext){
    [...]
    let shares = get_vault_total_shares_in_circulation(vault);
    let tvl = math::mul(shares, vault.rate);
    let fee_amount = math::mul(tvl, vault.fee_percentage);
    [...]
}
```

Remediation

Assert that `fee_amount > 0` before proceeding with updates, ensuring that each charge attempt is meaningful.

Patch

Resolved in 0960421.

Asset Inflation and Fund Loss via Improper Rounding LOW OS-BUV-ADV-06

Description

In `vault::process_request`, the withdrawal amount is computed as `math::div(request.shares, vault.rate)`, which performs integer division and truncates fractional results. Thus, if `request.shares` is less than `vault.rate`, the result becomes zero. However, the function still proceeds to burn the user's shares, but since `withdraw_amount` is zero, no assets are transferred back, resulting in users permanently losing funds. To mitigate this, add a check to ensure `withdraw_amount` is greater than zero.

>_ *sources/vault.move* RUST

```
fun process_request<T,R>(vault: &mut Vault<T,R>, request: &WithdrawalRequest, current_time: u64,
    ↪  ctx: &mut TxContext): (bool, u64, u64){
    [...]
    let mut withdraw_amount = math::div(request.shares, vault.rate);
    burn_shares(vault, request.shares, ctx);
    [...]
}
```

Similarly, `vault::mint_shares` utilizes `math::div(shares, vault.rate)` to compute the `total_amount` from `shares` and `vault.rate`. Since integer division truncates toward zero, if `shares` is very small relative to `vault.rate`, the result may round down to zero. This effectively allows a user to mint shares without requiring them to contribute any actual assets. This may be exploited repeatedly to inflate share balances at no cost, diluting the value of legitimate holders and draining the vault. Enforce `total_amount > 0` to ensure no shares are minted without corresponding asset deposits.

>_ *sources/vault.move* RUST

```
public fun mint_shares<T,R>(vault: &mut Vault<T,R>, config: &ProtocolConfig, balance: &mut
    ↪  Balance<T>, shares: u64, ctx: &mut TxContext): Coin<R> {
    [...]
    // Calculate the deposit amount needed for the requested shares
    // shares = deposit_amount * vault.rate  => deposit_amount = shares / vault.rate
    // But since vault.rate is a fixed-point (1e9), we use integer division
    let total_amount = math::div(shares, vault.rate);
    [...]
}
```

Furthermore, in `vault::deposit_asset`, the number of shares to mint (`shares_minted`) is calculated as `math::mul(total_amount, vault.rate)`. Thus, if `total_amount` is very small or `vault.rate` is low, `shares_minted` may be rounded to zero. Consequently, the user's assets will be added to the vault without receiving any receipt tokens in return, effectively resulting in a loss of funds for the depositor. Assert `shares_minted > 0` before joining the balance (`balance::join`), ensuring deposits always yield receipt tokens.

>_ sources/vault.move RUST

```rust
public fun deposit_asset<T,R>(vault: &mut Vault<T,R>, config: &ProtocolConfig, balance:
    ↪  Balance<T>, ctx: &mut TxContext): Coin<R> {
    [...]
    let shares_minted = math::mul(total_amount, vault.rate);
    let receipt_coin = coin::mint(&mut vault.receipt_token_treasury_cap, shares_minted, ctx);
    [...]
}
```

Remediation

Ensure `withdraw_amount > 0`, `total_amount > 0`, and `shares_minted > 0` before proceeding with the burning or minting of shares in `process_request`, `mint_shares`, and `deposit_asset` respectively.

Patch

Resolved in 0960421, b2dabe8, and 3e1afa0.

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-BUV-SUG-00	`vault::mint_shares` suffers from a rounding-down issue, allowing users to mint more shares than their deposit covers, resulting in free shares.
OS-BUV-SUG-01	The `process_request` logic unnecessarily sets an index for skipped requests, resulting in redundant handling in `update_account_state`. Only canceled requests should modify the cancellation vector.
OS-BUV-SUG-02	There are several instances where the validation logic may be improved to increase the robustness of the code.
OS-BUV-SUG-03	Recommendation for refactoring the codebase for better functionality and mitigating potential security issues.
OS-BUV-SUG-04	The code may be optimized to reduce overall redundancy and improve clarity.
OS-BUV-SUG-05	Suggestions to ensure adherence to best coding practices.

Improper Rounding During Share Minting OS-BUV-SUG-00

Description

The vulnerability in `vault::mint_shares` comes from integer division rounding down when converting shares to required assets. Since the calculation truncates the decimal part, users may deposit fewer assets than needed but still receive the full number of shares. For example, with a vault rate of 500% (1 asset = 5 shares), a user requesting 9 shares only needs to deposit 1 asset instead of 1.8, effectively getting 4 shares for free.

Remediation

Round up the calculation to benefit the protocol.

Patch

This issue was acknowledged.

Inadequate Cancel Withdrawal Requests Logic OS-BUV-SUG-01

Description

The current logic utilized for the cancellation of withdrawal requests is inadequate. `vault::process_request` sets an `option_index` for skipped withdrawal requests, where the index will be `none`, even though these requests are not actually canceled. Consequently, `update_account_state` utilizes this index to potentially remove entries from the `cancel_withdraw_request` vector, which is unnecessary as only cancel requests will have the index option filled, and if the index is invalid for some reason, the function will revert anyway.

Remediation

Remove the logic for setting an `option_index` for skipped withdrawal requests in `process_request`, and also remove the nested `if` condition in `update_account_state`.

Patch

Resolved in PR#16.

Missing Validation Checks

OS-BUV-SUG-02

Description

1. Both `vault::process_withdrawal_requests` and `vault::process_withdrawal_requests_up_to_timestamp` may process an unbounded number of requests in a single transaction. Add a maximum request limit per transaction to ensure predictable execution and mitigate abuse or excessive gas utilization.
2. There is a potential governance and operational risk in the vault's rate update mechanism due to a lack of restriction on how frequently the vault rate may be updated, rendering `vault.max_rate_change_per_update` ineffective, as users may experience rapid, unexpected fluctuations in the vault's effective rate, undermining predictability and creating user distrust. Implement a minimum and maximum time interval between consecutive rate updates to ensure that changes are not unexpected and to prevent sudden fluctuations.
3. In `vault::update_vault_fee_percentage`, `vault::change_vault_rate_update_interval`, and `vault::update_vault_rate`, update `fee_percentage`, `rate_update_interval`, and `rate.last_updated_at` respectively, only if the previous values differ from the new values. This is especially necessary when updating `vault.rate.last_updated_at`, since there is a timeout for updating the rate.
4. In the admin update functions, include a check to ensure the new admin value differs from the current one, executing the logic only when a change occurs. This prevents unnecessary event emission when no state change takes place.
5. `vault::create_vault` lacks essential validations, allowing vaults to be created with invalid `admin` / `operator` accounts (with `@0x0` address), unreasonable fee percentages, or zero minimum withdrawal shares. Verify that `admin` and `operator` accounts are valid (`!= @0x0`), and that `fee_percentage` is within the protocol limits. Adding these checks ensures vault integrity and prevents misconfiguration.

Remediation

Add the missing validations mentioned above.

Patch

1. Issue #1 was acknowledged.
2. Issue #2 was resolved in 883add5.
3. Issue #3 was resolved in e8187e9.
4. Issues #4 and #5 were resolved in PR#16.

Code Refactoring OS-BUV-SUG-03

Description

1. `vault::process_withdrawal_requests` currently aborts if `num_requests` exceeds the number of pending withdrawals in the queue, because `queue::dequeue` fails on an empty queue. This results in all previously processed requests in the same transaction to be reverted, wasting gas and leaving users' withdrawals unprocessed. Either process only the available requests via `min(num_requests, vault.pending_withdrawals.len())` or check that `num_requests` does not exceed the queue length before starting.

>_ sources/vault.move RUST

```
public fun process_withdrawal_requests<T,R>(clock: &Clock, vault: &mut Vault<T,R>, config:
    ↪  &ProtocolConfig, num_requests: u64, ctx: &mut TxContext){
    [...]
    let mut i =0;
    while(i < num_requests){
        let request = queue::dequeue(&mut vault.pending_withdrawals);
        [...]
    };
    [...]
}
```

2. `queue::Queue` utilizes `u64` counters (`head` and `tail`) to track positions, and `Queue.tail` keeps incrementing every time an element is added. If the queue is utilized heavily over a long period, `Queue.tail` may theoretically reach `u64::max_value`, which will result in an overflow and break the queue logic, creating a denial of service. Ensure to reset `Queue.tail` and `Queue.head` back to zero when the queue gets empty.

>_ sources/queue.move RUST

```
/// FIFO Queue
///
/// Parameters:
/// - id: The unique identifier for the queue.
/// - table: The table to store the queue elements.
/// - head: The index of the first element in the queue.
/// - tail: The index of the last element in the queue.
public struct Queue<phantom T: store> has key, store {
    id: UID,
    table: Table<u64, T>,
    head: u64,
    tail: u64,
}
```

3. Align the current design with `ERC-4626` and `ERC-7540` to standardize deposit/withdrawal logic, share accounting, and cross-protocol interoperability. This will reduce integration complexity, improve compatibility with aggregators, and make the vaults easier for other protocols and tools to support.
4. `vault::update_accounts` should fail when attempting to remove an account that does not exist in the list. Without this check, functions such as `vault::set_sub_account` and `vault::set_blacklisted_account` may succeed even when removing non-existent accounts.
5. The `rate != vault.rate.value` check in `vault::update_vault_rate` is currently performed late in the function, resulting in wasted validation if it fails. Move it to the beginning to allow for an early exit, saving gas and improving efficiency.

Remediation

Incorporate the above refactors.

Patch

1. Issue #1 was resolved in 585bcfa.
2. Issue #2 was resolved in 7b046bc.
3. Issue #3 was acknowledged.
4. Issue #4 was resolved in e8187e9.
5. Issue #5 was acknowledged.

Code Clarity

OS-BUV-SUG-04

Description

1. There are multiple modifications that may be included to ensure improved clarity and better efficiency, including utilizing defined functions instead of repeating code and dividing functions for increased composability.
2. Utilize vector macros instead of manually indexing and borrowing vector elements, which is verbose and error-prone as currently done in `vault::cancel_pending_withdrawal_request`.
3. Ensure to follow the Sui and Move best practices guides.
4. In `math::safely_cast_to_u64`, the maximum `u64` value (`18446744073709551615`) is hard-coded, which reduces readability and maintainability. Utilize `std::u64::max_value!()` instead for clearer code.

>_ sources/math.move RUST

```rust
fun safely_cast_to_u64(result: u128): u64 {
    assert!(result <= (18446744073709551615 as u128), EOverflow); // u64::MAX overflow
        ↪  check
    result as u64
}
```

Remediation

Ensure to update the codebase with the above changes.

Patch

1. Issue #1, #2, and #3 resolved in dfbbfc6 and e8187e9.
2. Issue #4 was resolved in dfbbfc6.

Code Maturity OS-BUV-SUG-05

Description

There are multiple instances where the comments are inaccurate or misleading, such as mentioning an admin capability in `vault::change_vault_operator` that does not exist, and omitting the clock parameter in `process_withdrawal_requests`. Additionally, in `admin::ProtocolConfig` `max_rate` should be 50% instead of 500%. Correct any inaccurate comments, ensuring a clear distinction between the `vault_admin` and the `protocol_admin` to prevent potential confusion. Also, the comments describing `vault::update_account_state` and `admin::pause_non_admin_operations` should be corrected to reflect the code implementations.

Remediation

Implement the above suggestion.

Patch

Resolved in 5035eff and e8187e9.

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 others may have 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.