Ember Vaults EVM OFT

Security Assessment

May 5th, 2026 — Prepared by OtterSec

Nicholas R. Putra nicholas@osec.io

Reyn Shaw reyn@osec.io

Table of Contents

Executive Summary **2**

Overview 2

Key Findings 2

Scope **3**

Findings **4**

Vulnerabilities **5**

OS-EFT-ADV-00 | Unrestricted Deposit Timestamp Manipulation 7

OS-EFT-ADV-01 | Failure to Accrue Platform Fees Before Mutations 8

OS-EFT-ADV-02 | quoteOFT Overstates Bridgeable Supply Cap 10

OS-EFT-ADV-03 | Withdrawal Queue DoS via Reverting ETH Receiver 11

OS-EFT-ADV-04 | Missing Combined Withdrawal Fee Cap 12

OS-EFT-ADV-05 | Withdrawal Penalty Bypass via Transferable Shares 13

OS-EFT-ADV-06 | Platform Fee Compounds on Itself 14

General Findings **15**

OS-EFT-SUG-00 | Absence of Zero Address and Same Value Checks 16

OS-EFT-SUG-01 | Inconsistent Vault Configuration Boundary Checks 18

OS-EFT-SUG-02 | Unbounded Per-Account Pending Withdrawal Array 20

OS-EFT-SUG-03 | Code Refactoring 21

Appendices

Vulnerability Rating Scale **23**

Procedure **24**

01 — Executive Summary

Overview

Ember Protocol engaged OtterSec to assess the `ember-vaults-evm-oapp-oft` program. This assessment was conducted between April 27th and May 5th, 2026. For more information on our auditing methodology, refer to Appendix B.

Key Findings

We produced 11 findings throughout this audit engagement.

In particular, we identified a vulnerability where there is a lack of access control in the deposit recording logic, allowing anyone to overwrite another user's last deposit timestamp and set a fresh timestamp, resulting in withdrawal fees calculation incorrectly applying time-based withdrawal fees (OS-EFT-ADV-00). Also, platform fee accrual is not consistently synchronized before fee collection or state changes that affect fee calculations, resulting in stale, over-counted, or under-counted fees when collecting, changing fee percentages, or bridging shares (OS-EFT-ADV-01).

Additionally, permanent and time-based withdrawal fee percentages are each capped below 100% individually, but their combined fee may exceed 100%, and since withdrawals subtract the summed fees from the withdraw amount, this may underflow and revert, blocking withdrawals (OS-EFT-ADV-04).

Furthermore, the time-based withdrawal fee is tied to the withdrawing address's last deposit timestamp, not to the shares themselves, enabling users to transfer or bridge shares to a fresh address with no recent deposit record and redeem there to avoid the penalty (OS-EFT-ADV-05).

We also suggested including zero-address and duplicate-value checks to prevent unnecessary and redundant operations and event emissions (OS-EFT-SUG-00), and recommended refactoring the code to improve functionality and overall robustness (OS-EFT-SUG-03). We further identified inconsistencies in Ember vault configuration checks between initialization and update paths, and advised ensuring consistent validation across both implementations (OS-EFT-SUG-01).

02 — Scope

The source code was delivered to us in a Git repository at https://github.com/fireflyprotocol/ember-vaults-evm-smart-contracts. This audit was performed against commit cbe90fd.

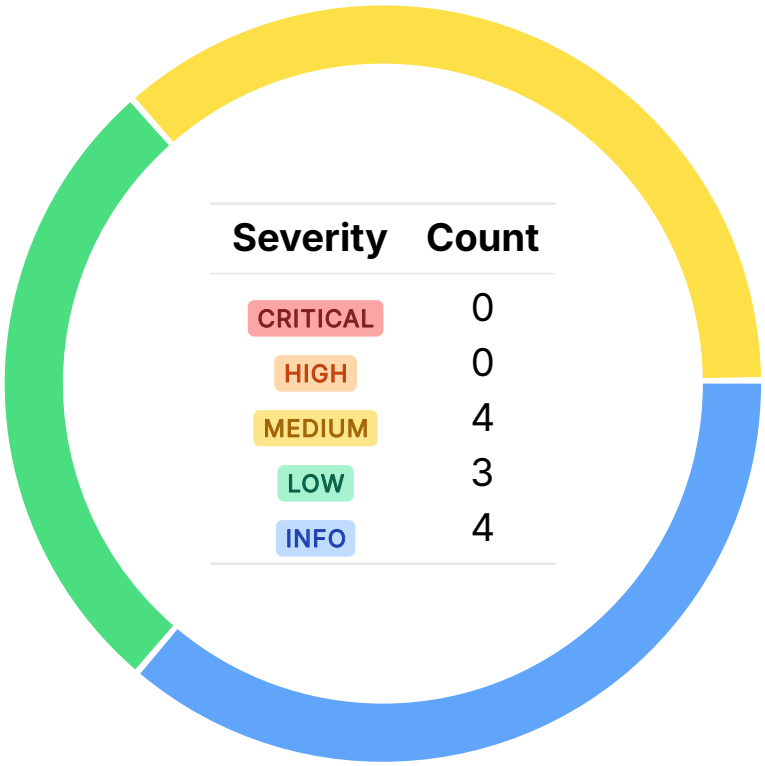
A brief description of the program is as follows:

Name	Description
ember-vaults-evm-oapp-oft	Adds support for OApp/OFTs in EVM Ember vaults using LayerZero tooling and cross-chain messaging infrastructure.

03 — Findings

Overall, we reported 11 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	4
LOW	3
INFO	4

04 — Vulnerabilities

Here, we present a technical analysis of the vulnerabilities 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-EFT-ADV-00	MEDIUM	RESOLVED	`recordDeposit()` lacks access control, allowing anyone to overwrite another user's last deposit timestamp and set a fresh timestamp, resulting in `calculateWithdrawalFees()` to incorrectly apply time-based withdrawal fees.
OS-EFT-ADV-01	MEDIUM	RESOLVED	Platform fee accrual is not consistently synchronized before fee collection or state changes that affect fee calculations, resulting in stale, overcounted, or undercounted fees when collecting, changing fee percentages, or bridging shares.
OS-EFT-ADV-02	MEDIUM	RESOLVED	`quoteOFT()` may overstate the maximum bridgeable amount because it only utilizes `maxBridgeAmount()` and ignores the source-chain `totalSupply()`.
OS-EFT-ADV-03	MEDIUM	RESOLVED	`processWithdrawalRequests()` may be stalled indefinitely by enqueuing a request whose `receiver` reverts on receiving native `ETH`, blocking every later request in the FIFO queue.
OS-EFT-ADV-04	LOW	RESOLVED	Permanent and time-based withdrawal fee percentages are each capped below 100% individually, but their combined fee may exceed 100%. Since withdrawals subtract the summed fees from `withdrawAmount`, this may underflow and revert, blocking withdrawals.

OS-EFT-ADV-05	LOW	RESOLVED	The time-based withdrawal fee is tied to the withdrawing address's last deposit timestamp, not to the shares themselves. Users may transfer or bridge shares to a fresh address with no recent deposit record and redeem them there to avoid the penalty.
OS-EFT-ADV-06	LOW	RESOLVED	`EmberETHVault` computes the next fee accrual against the full `totalAssets()` without first subtracting `platformFee.accrued`, so previously accrued (but uncollected) fees themselves accrue additional fees.

Unrestricted Deposit Timestamp Manipulation MEDIUM OS-EFT-ADV-00

Description

`EmberVaultValidator::recordDeposit()` is externally callable without restricting `msg.sender`, so anyone may overwrite `_lastDepositTimestamp[vault][victim]` with a recent timestamp. Since calculateWithdrawalFees() applies the time-based withdrawal fee when lastDeposit + threshold > currentTime, an attacker may refresh a victim's deposit timestamp right before withdrawal processing and force them to pay a fee they should not owe.

>_ *contracts/EmberVaultValidator.sol* SOLIDITY

```solidity
/// @notice Records the last deposit timestamp for the receiver
/// @dev Called by the vault after a successful deposit
function recordDeposit(address vault, address receiver, uint256 timestamp) external {
  _lastDepositTimestamp[vault][receiver] = timestamp;
}
```

Remediation

Validate `msg.sender == vault`, so only successful vault deposits may update the timestamp.

Patch

Resolved in 50c234b.

Failure to Accrue Platform Fees Before Mutations MEDIUM OS-EFT-ADV-01

Description

Currently, fee accounting may become stale or mispriced because `collectPlatformFee()`, `setFeePercentage()`, and bridge mint/burn paths do not consistently call `_chargeAccruedPlatformFees()` before collecting fees or changing the fee base as detailed below:

1. `EmberVault::collectPlatformFee()` transfers only the already stored `platformFee.accrued` amount, but it does not first call `_chargeAccruedPlatformFees()` to accrue fees up to the current timestamp. As a result, the operator may collect a stale fee amount that excludes fees accumulated since the last fee-charging action. This is inconsistent with the `collectPlatformFee()` implementation in `EmberETHVault`, which explicitly refreshes accrued fees before collection.

   >_ *contracts/EmberVault.sol* SOLIDITY

   ```
   function collectPlatformFee() external nonReentrant onlyOperator returns (uint256 amount)
       ↪ {
     IEmberProtocolConfig configProxy = protocolConfig;
     // Check protocol is not paused
     if (configProxy.getProtocolPauseStatus()) revert ProtocolPaused();
     // Check vault privileged operations are not paused
     if (pauseStatus.privilegedOperations) revert OperationPaused();
     // Check that there are accrued fees
     if (platformFee.accrued == 0) revert ZeroAmount();
     amount = platformFee.accrued;
     [...]
   }
   ```

2. `setFeePercentage()` updates `platformFee.platformFeePercentage` directly without first calling `_chargeAccruedPlatformFees()`, so any fees accrued since the last charge are effectively calculated later with the new fee rate instead of the old one. This may overcharge or undercharge the vault, depending on whether the fee percentage is increased or decreased. The function also does not independently enforce `EmberProtocolConfig::getMaxAllowedFeePercentage()`.
3. Bridge minting and burning changes `totalSupply()`, which directly changes Ember's rate-based `totalAssets()` because `totalAssets()` is calculated as `convertToAssets(totalSupply())`. If `bridgeMint()` or `bridgeBurn()` occurs without first calling `_chargeAccruedPlatformFees()`, fee accrual may skip or misprice the period before the supply change.

Remediation

1. Call `_chargeAccruedPlatformFees()` after pause checks and before the `platformFee.accrued == 0` check, so collection always reflects fees due as of the collection transaction.
2. Accrue fees at the existing rate before updating the fee percentage, and reject `newFeePercentage` if it exceeds the protocol-defined maximum.
3. Ensure the bridge functions accrue platform fees before mutating the bridged share supply, so fees are calculated against the pre-bridge TVL.

Patch

Resolved in 50c234b.

quoteOFT Overstates Bridgeable Supply Cap MEDIUM OS-EFT-ADV-02

Description

`EmberVaultMintBurnOFTAdapter::quoteOFT()` currently reports `maxAmountLD` from `bridgeableToken.maxBridgeAmount()`, treating zero as unlimited, but it does not cap that value by the token's available `totalSupply()`. In a mint/burn OFT model, a source-chain send burns existing receipt tokens, so the maximum amount that may be bridged from that chain cannot exceed the receipt token supply on that chain. If `maxBridgeAmount()` is unset or larger than `totalSupply`, the quote may overstate the transferable amount

>_ *contracts/EmberVaultMintBurnOFTAdapter.sol* SOLIDITY

```solidity
function quoteOFT(
  SendParam calldata _sendParam
)
  external
  view
  virtual
  returns (
    OFTLimit memory oftLimit,
    OFTFeeDetail[] memory oftFeeDetails,
    OFTReceipt memory oftReceipt
  )
{
  // Query vault's bridge limits for accurate quote
  uint256 minAmountLD = bridgeableToken.minBridgeAmount();
  uint256 vaultMaxAmount = bridgeableToken.maxBridgeAmount();
  // If vault has no max set (0), use type(uint256).max
  uint256 maxAmountLD = vaultMaxAmount > 0 ? vaultMaxAmount : type(uint256).max;
  [...]
}
```

Remediation

Return `min(configuredMaxAmount, bridgeableToken.totalSupply())`, with `totalSupply()` also acting as the effective cap when the configured maximum is unlimited.

Patch

Resolved in 50c234b.

Withdrawal Queue DoS via Reverting ETH Receiver MEDIUM OS-EFT-ADV-03

Description

`EmberETHVault::_executeWithdrawalWithETH()` forwards unwrapped `ETH` to `request.receiver` via a low-level `call` and reverts with `ETHTransferFailed` on failure.

>_ *contracts/EmberETHVault.sol* SOLIDITY

```
function _executeWithdrawalWithETH(...) internal returns (...) {
  [...]
  // Unwrap WETH to ETH
  IWETH(asset()).withdraw(withdrawAmount);
  (bool success, ) = request.receiver.call{ value: withdrawAmount }("");
  if (!success) revert ETHTransferFailed();
  [...]
}
```

The loop in `processWithdrawalRequests()` has no per-request `try/catch`, so a single failing transfer aborts the whole batch and leaves `withdrawalQueueStartIndex` unchanged. An attacker may deploy a contract that rejects `ETH`, submit a `redeemShares` request naming it as `receiver`, and permanently stall every queued request behind it.

Remediation

Fall back to delivering `WETH` to the receiver instead of reverting, on failed `ETH` transfer.

Patch

The issue was acknowledged by the development team.

Missing Combined Withdrawal Fee Cap LOW OS-EFT-ADV-04

Description

In `EmberProtocolConfig`, `updateVaultPermanentFeePercentage()` and `updateVaultTimeBasedFeePercentage()` each only enforce `newPercentage < 1e18` independently, so both fees may be configured close to 100% at the same time. During withdrawal processing, the validator returns both fee amounts and the vault sums them before subtracting from `withdrawAmount`. If the combined fee exceeds the withdrawal amount, `withdrawAmount -= totalFee` will underflow and revert, resulting in the affected withdrawals to fail.

>_ *contracts/EmberProtocolConfig.sol* SOLIDITY

```
function updateVaultPermanentFeePercentage(
  address vault,
  uint256 newPercentage
) external nonReentrant {
  if (newPercentage >= 1e18) revert InvalidValue();
  IEmberVaultValidator validator = IEmberVault(vault).vaultValidator();
  if (newPercentage == validator.withdrawalFee(vault).permanentFeePercentage) revert
      ↪  SameValue();
  validator.setPermanentFeePercentage(msg.sender, vault, newPercentage);
}

function updateVaultTimeBasedFeePercentage(
  address vault,
  uint256 newPercentage
) external nonReentrant {
  if (newPercentage >= 1e18) revert InvalidValue();
  IEmberVaultValidator validator = IEmberVault(vault).vaultValidator();
  if (newPercentage == validator.withdrawalFee(vault).timeBasedFeePercentage) revert
      ↪  SameValue();
  validator.setTimeBasedFeePercentage(msg.sender, vault, newPercentage);
}
```

Remediation

Enforce a joint cap, such as `permanentFeePercentage + timeBasedFeePercentage <= 1e18`, or the withdrawal path should explicitly validate `totalFee <= withdrawAmount` before subtraction.

Patch

Resolved in 50c234b.

Withdrawal Penalty Bypass via Transferable Shares LOW OS-EFT-ADV-05

Description

The time-based withdrawal fee is keyed to `lastDepositTimestamp[vault][owner]`, but receipt shares are transferable and bridgeable. A user may deposit, transfer their shares to a fresh address, or bridge them to another chain, and then redeem from that address, where no recent deposit timestamp is recorded. Since `calculateWithdrawalFees()` checks the withdrawing `owner_` rather than the provenance of the shares, the time-deposit penalty may be bypassed.

Remediation

Track deposit timestamps/penalty state per share lot.

Patch

Partially resolved in 50c234b. The team has acknowledged a residual bypass via pre-positioned addresses with stale `lastDepositTimestamp` and accepted the trade-off.

Platform Fee Compounds on Itself LOW OS-EFT-ADV-06

Description

`EmberETHVault::_chargeAccruedPlatformFees()` utilizes `currentTotalAssets` directly as the multiplier when computing the new fee for the elapsed period, without removing the unclaimed `platformFee.accrued` amount that is already part of `totalAssets()`. Because the accrued fees are protocol-owned and not LP-owned, they should not themselves be charged the platform fee rate. Each time `_chargeAccruedPlatformFees` runs while a non-zero accrued balance is uncollected, the prior accrual is treated as if it were LP-owned TVL and re-charged at the platform fee rate, causing the platform fee to compound on itself. The overcharge relative to the intended fee grows with both the uncollected period and the fee percentage.

>_ *contracts/EmberETHVault.sol* SOLIDITY

```
function _chargeAccruedPlatformFees() internal {
  [...]
  uint256 currentTotalAssets = totalAssets();
  uint256 feePercentage = platformFee.platformFeePercentage;

  uint256 accruedFee = (currentTotalAssets * feePercentage * timeSinceLastCharge) /
    FEE_DENOMINATOR;
  [...]
}
```

Remediation

Exclude the accrual fee from the calculated TVL.

Patch

Resolved in fba897f.

05 — General Findings

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

ID	Description
OS-EFT-SUG-00	There are multiple instances where zero-address and duplicate-value checks are missing, resulting in unnecessary and redundant operations and event emissions.
OS-EFT-SUG-01	The Ember vault configuration checks are inconsistent across initialization and update paths, as some boundary values are rejected at deployment but allowed later or vice versa.
OS-EFT-SUG-02	`redeemShares()` pushes onto the array that stores the sequence numbers for pending withdrawal requests with no per-account cap, allowing a spammer to grow the array until linear scans over it consume excessive gas.
OS-EFT-SUG-03	Suggestions to refactor the code to improve functionality and overall robustness.

Absence of Zero Address and Same Value Checks OS-EFT-SUG-00

Description

1. `EmberETHVault::setPausedStatus()` updates a pause flag even when the requested paused value is already the current value, resulting in redundant state updates, unnecessary `sequenceNumber` increments, and misleading `VaultPauseStatusUpdated` events that imply a real status change occurred. Add a `SameValue()` check before assignment to ensure the function is consistent with other setters and prevent no-op pause updates.

>_ *contracts/EmberETHVault.sol* SOLIDITY

```
function setPausedStatus(
  address caller,
  string calldata operation,
  bool paused
) external nonReentrant onlyProtocolConfig onlyAdmin(caller) {
  bytes32 operationHash = keccak256(bytes(operation));
  if (operationHash == DEPOSITS_HASH) {
    pauseStatus.deposits = paused;
  } else if (operationHash == WITHDRAWALS_HASH) {
    pauseStatus.withdrawals = paused;
  } else if (operationHash == PRIVILEGED_OPS_HASH) {
    pauseStatus.privilegedOperations = paused;
  }
  [...]
}
```

2. `EmberETHVault::collectPlatformFee()` retrieves `feeRecipient` from `EmberProtocolConfig::getPlatformFeeRecipient()` and transfers accrued `WETH` to it without checking whether the returned address is `address(0)`. This is inconsistent with the behavior of `EmberVault`. Check that `feeRecipient` is not `address(0)` before calling `safeTransfer`

>_ *contracts/EmberETHVault.sol* SOLIDITY

```
function collectPlatformFee() external nonReentrant onlyOperator returns (uint256 amount)
    ↪ {
  [...]
  address feeRecipient = configProxy.getPlatformFeeRecipient();
  // Transfer WETH to platform fee recipient (NOT unwrapped)
  IERC20(asset()).safeTransfer(feeRecipient, amount);
  [...]
}
```

3. `EmberVaultMintBurnOFTAdapter::_credit()` mints directly to `_to` without handling the zero-address case. If an incompatible cross-chain message decodes `sendTo()` as `address(0)`, the call to `bridgeMint(address(0), _amountLD)` may revert. The `MintBurnOFTAdapter` pattern redirects zero-address recipients to `address(0xdead)` to avoid failed delivery. Check if `_to == 0x0` and change it to `0xdead` in that case.

Remediation

Add the missing value checks.

Patch

Resolved in 50c234b.

Inconsistent Vault Configuration Boundary Checks OS-EFT-SUG-01

Description

`EmberVault::initialize()` and `EmberETHVault::initialize()` reject `params.rateUpdateInterval == configProxy.getMinRateInterval()`, effectively requiring the interval to be strictly greater than the minimum. However, `updateVaultRateUpdateInterval()` in `EmberProtocolConfig`, allows `newInterval == protocolConfig.minRateInterval`, since it only rejects values below the minimum or above the maximum. This creates inconsistent configuration semantics, where a vault cannot be initialized with the protocol minimum, but may later be updated to that same value.

>_ *contracts/EmberProtocolConfig.sol* SOLIDITY

```
function updateVaultRateUpdateInterval(address vault, uint256 newInterval) external nonReentrant
    ↪ {
  if (
    newInterval < protocolConfig.minRateInterval || newInterval > protocolConfig.maxRateInterval
  ) revert InvalidInterval();
  if (newInterval == IEmberVault(vault).rate().rateUpdateInterval) revert SameValue();
  IEmberVault(vault).setRateUpdateInterval(msg.sender, newInterval);
}
```

Similarly, `EmberVault::initialize()` and `EmberETHVault::initialize()` reject `params.feePercentage >= configProxy.getMaxAllowedFeePercentage()`, implying the initial platform fee must be strictly less than the protocol maximum. However, `EmberProtocolConfig::updateVaultFeePercentage()` only rejects `newFeePercentage > protocolConfig.maxFeePercentage`, so it allows the fee to be updated to exactly the maximum. This creates inconsistent fee-boundary semantics as a vault cannot be initialized at the max allowed fee, but may be changed to that value later.

>_ *contracts/EmberProtocolConfig.sol* SOLIDITY

```
function updateVaultFeePercentage(address vault, uint256 newFeePercentage) external nonReentrant
    ↪ {
  if (newFeePercentage > protocolConfig.maxFeePercentage) {
    revert InvalidFeePercentage();
  }
  if (newFeePercentage == IEmberVault(vault).platformFee().platformFeePercentage) {
    revert SameValue();
  }
  IEmberVault(vault).setFeePercentage(msg.sender, newFeePercentage);
}
```

Furthermore, `EmberVault::initialize()` and `EmberETHVault::initialize()` store `params.maxRateChangePerUpdate` without rejecting zero, so a vault may be deployed with `maxRateChangePerUpdate == 0`. However, `EmberProtocolConfig::updateVaultMaxRateChangePerUpdate()` explicitly rejects zero with `InvalidRate()`, implying the same value is invalid during updates but valid during initialization.

>_ *contracts/EmberProtocolConfig.sol* SOLIDITY

```
function updateVaultMaxRateChangePerUpdate(
  address vault,
  uint256 newMaxRateChangePerUpdate
) external nonReentrant {
  if (newMaxRateChangePerUpdate == 0) revert InvalidRate();
  if (newMaxRateChangePerUpdate == IEmberVault(vault).rate().maxRateChangePerUpdate)
    revert SameValue();
  IEmberVault(vault).setMaxRateChangePerUpdate(msg.sender, newMaxRateChangePerUpdate);
}
```

Lastly, `EmberProtocolConfig` initializes `minRateInterval` to `MIN_RATE_INTERVAL = 60 * 60 * 1000` (1 hour), but `updateMinRateInterval()` later allows the owner to reduce it as low as `60 * 1_000` (1 minute). This renders the deployed default and post-deployment lower bound inconsistent. If the intended invariant is a 1-hour minimum, `updateMinRateInterval()` should enforce `minRateInterval_ >= MIN_RATE_INTERVAL`.

>_ *contracts/EmberProtocolConfig.sol* SOLIDITY

```
function updateMinRateInterval(uint256 minRateInterval_) external nonReentrant onlyOwner {
  if (minRateInterval_ < 60 * 1_000 || minRateInterval_ > protocolConfig.maxRateInterval)
    revert InvalidInterval();
  if (minRateInterval_ == protocolConfig.minRateInterval) revert SameValue();
  uint256 previous = protocolConfig.minRateInterval;
  protocolConfig.minRateInterval = minRateInterval_;
  emit MinRateIntervalUpdated(previous, minRateInterval_);
}
```

Remediation

Ensure consistency between the initilization and update paths.

Patch

Resolved in 50c234b.

Unbounded Per-Account Pending Withdrawal Array OS-EFT-SUG-02

Description

Every `redeemShares()` call appends to `accountState.pendingWithdrawalRequestSequenceNumbers` via `_updateAccountState`, and that array is scanned linearly by `cancelPendingWithdrawalRequest()` and by the removal branch of `_updateAccountState` during `processWithdrawalRequests()`. An attacker spamming many minimum-size redeems from a single account can inflate the array until those scans approach the block gas limit, slowing the operator's batch when processing the attacker's requests and eventually bricking the attacker's own cancel/process paths. Triggering this is gas-expensive for the attacker, but the asymmetric cost-to-effect warrants mitigation.

>_ *contracts/EmberETHVault.sol* SOLIDITY

```
function _updateAccountState(WithdrawalRequest memory request, bool add, uint256 index) internal
    ↪ {
  [...]
  if (add) {
    [...]
    accountState.pendingWithdrawalRequestSequenceNumbers.push(request.sequenceNumber);
  } else {
    [...]
  }
}
```

Remediation

Enforce a per-account cap on outstanding pending withdrawal requests in `redeemShares()`.

Patch

The issue was acknowledged by the development team.

Code Refactoring OS-EFT-SUG-03

Description

1. `EmberETHVault::_mint()` calculates the `WETH` required for an exact share mint via `convertToAssets(shares)` , which applies floor rounding by default. If the share-to-asset conversion has a remainder, the user may pay slightly less `WETH` than the required amount while still receiving the full requested shares. For exact-share minting, Utilize ceil rounding to favor rounding assets up so the vault/protocol is not underpaid.

>_ contracts/EmberETHVault.sol SOLIDITY

```
function _mint(
  uint256 shares,
  address receiver,
  address depositor
) internal returns (uint256 assets) {
  [IEmberProtocolConfig configProxy = protocolConfig;
  [...]
  // Calculate assets needed using rate-based conversion
  assets = convertToAssets(shares);
  if (assets == 0) revert ZeroAmount();
  // Transfer WETH from depositor to vault (if depositor is not vault itself)
  if (depositor != address(this)) {
    IERC20(asset()).safeTransferFrom(depositor, address(this), assets);
  }]
  // Mint shares to receiver
  _mint(receiver, shares);
  [...]
}
```

2. `bridgeMint()` and `bridgeBurn()` are guarded only by `onlyBridgeAdapter` and do not check `EmberProtocolConfig::getProtocolPauseStatus()` or the vault's pause flags. As a result, even when the vault or protocol is paused to halt operations, the bridge adapter may still mint or burn receipt shares and change vault supply/accounting.
3. `bridgeMint()` and `bridgeBurn()` do not check whether `to` or `from` is blacklisted in `protocolConfig`, allowing blacklisted accounts to potentially move receipt tokens cross-chain, or receive bridged shares on a destination vault, even though normal deposits and withdrawals enforce blacklist checks. Ensure to validate the bridged sender/recipient against `EmberProtocolConfig::isAccountBlacklisted()`.

Remediation

Incorporate the above refactors.

Patch

Resolved in 50c234b.

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.