// Security Assessment

10.20.2025 - 10.23.2025

Concrete Predeposit Vault

Blueprint Finance

HALBORN

Concrete Predeposit Vault - Blueprint Finance

Prepared by: HALBORN

Last Updated 11/10/2025

Date of Engagement: October 20th, 2025 - October 23rd, 2025

Summary

100% ⓘ OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
4	0	1	0	1	2

TABLE OF CONTENTS

1. Introduction

Blueprint Finance engaged Halborn to perform a security assessment of their smart contracts beginning on October 20th, 2025 and ending on October 23rd, 2025. The assessment scope was limited to the smart contracts provided to Halborn. Commit hashes and additional details are available in the Scope section of this report.

The reviewed contracts implement a cross-chain share claiming mechanism built on LayerZero that enables users to deposit assets on Layer 1 (L1), burn their shares, and receive equivalent shares on Layer 2 (L2).

2. Assessment Summary

Halborn assigned 1 full-time security engineer to conduct a comprehensive review of the smart contracts within scope. The engineer is an expert in blockchain and smart contract security, with advanced skills in penetration testing and smart contract exploitation, as well as extensive knowledge of multiple blockchain protocols.

The objectives of this assessment were to:

- Identify potential security vulnerabilities within the smart contracts.
- Verify that the smart contract functionality operates as intended.

In summary, Halborn identified several areas for improvement to reduce the likelihood and impact of security risks, which were addressed by the Blueprint Finance team. The main ones were:

- Add address validation and event emission in emergency withdrawals to prevent accidental fund loss.
- Implement explicit replay prevention on the origin chain to align with the destination chain.
- Compact batch arrays before sending cross-chain messages to avoid oversized payloads.
- Add array length checks in batch processing to avoid mismatched input errors.

3. Test Approach And Methodology

`Halborn` performed a combination of a manual review of the source code and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of the program assessment. While manual testing is recommended to uncover flaws in business logic, processes, and implementation; automated testing techniques help enhance coverage of programs and can quickly identify items that do not follow security best practices.

The following phases and associated tools were used throughout the term of the assessment:

- Research into the architecture and purpose of the smart contracts.
- Manual code review and walkthrough of the smart contracts.
- Manual assessment of critical Solidity variables and functions to identify potential vulnerability classes.
- Manual testing using custom scripts.
- Static security analysis of the scoped contracts and imported functions.
- Local deployment and testing with Foundry & Hardhat.

4. RISK METHODOLOGY

Every vulnerability and issue observed by Halborn is ranked based on **two sets** of **Metrics** and a **Severity Coefficient**. This system is inspired by the industry standard Common Vulnerability Scoring System.

The two **Metric sets** are: **Exploitability** and **Impact**. **Exploitability** captures the ease and technical means by which vulnerabilities can be exploited and **Impact** describes the consequences of a successful exploit.

The **Severity Coefficients** is designed to further refine the accuracy of the ranking with two factors: **Reversibility** and **Scope**. These capture the impact of the vulnerability on the environment as well as the number of users and smart contracts affected.

The final score is a value between 0-10 rounded up to 1 decimal place and 10 corresponding to the highest security risk. This provides an objective and accurate rating of the severity of security vulnerabilities in smart contracts.

The system is designed to assist in identifying and prioritizing vulnerabilities based on their level of risk to address the most critical issues in a timely manner.

4.1 EXPLOITABILITY

ATTACK ORIGIN (AO):

Captures whether the attack requires compromising a specific account.

ATTACK COST (AC):

Captures the cost of exploiting the vulnerability incurred by the attacker relative to sending a single transaction on the relevant blockchain. Includes but is not limited to financial and computational cost.

ATTACK COMPLEXITY (AX):

Describes the conditions beyond the attacker's control that must exist in order to exploit the vulnerability. Includes but is not limited to macro situation, available third-party liquidity and regulatory challenges.

METRICS:

EXPLOITABILITY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Origin (AO)	Arbitrary (AO:A) Specific (AO:S)	1 0.2
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33

EXPLOITABILITY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Complexity (AX)	Low (AX:L) Medium (AX:M) High (AX:H)	1 0.67 0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

4.2 IMPACT

CONFIDENTIALITY (C):

Measures the impact to the confidentiality of the information resources managed by the contract due to a successfully exploited vulnerability. Confidentiality refers to limiting access to authorized users only.

INTEGRITY (I):

Measures the impact to integrity of a successfully exploited vulnerability. Integrity refers to the trustworthiness and veracity of data stored and/or processed on-chain. Integrity impact directly affecting Deposit or Yield records is excluded.

AVAILABILITY (A):

Measures the impact to the availability of the impacted component resulting from a successfully exploited vulnerability. This metric refers to smart contract features and functionality, not state. Availability impact directly affecting Deposit or Yield is excluded.

DEPOSIT (D):

Measures the impact to the deposits made to the contract by either users or owners.

YIELD (Y):

Measures the impact to the yield generated by the contract for either users or owners.

METRICS:

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Confidentiality (C)	None (C:N) Low (C:L) Medium (C:M) High (C:H) Critical (C:C)	0 0.25 0.5 0.75 1

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Integrity (I)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Availability (A)	None (A:N) Low (A:L) Medium (A:M) High (A:H) Critical (A:C)	0 0.25 0.5 0.75 1
Deposit (D)	None (D:N) Low (D:L) Medium (D:M) High (D:H) Critical (D:C)	0 0.25 0.5 0.75 1
Yield (Y)	None (Y:N) Low (Y:L) Medium (Y:M) High (Y:H) Critical (Y:C)	0 0.25 0.5 0.75 1

Impact I is calculated using the following formula:

$$I = max(m_I) + \frac{\sum m_I - max(m_I)}{4}$$

4.3 SEVERITY COEFFICIENT

REVERSIBILITY (R):

Describes the share of the exploited vulnerability effects that can be reversed. For upgradeable contracts, assume the contract private key is available.

SCOPE (S):

Captures whether a vulnerability in one vulnerable contract impacts resources in other contracts.

METRICS:

SEVERITY COEFFICIENT (C)	COEFFICIENT VALUE	NUMERICAL VALUE
Reversibility (r)	None (R:N) Partial (R:P) Full (R:F)	1 0.5 0.25
Scope (s)	Changed (S:C) Unchanged (S:U)	1.25 1

Severity Coefficient C is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score S is obtained by:

$$S = min(10, EIC * 10)$$

The score is rounded up to 1 decimal places.

SEVERITY	SCORE VALUE RANGE
Critical	9 - 10
High	7 - 8.9
Medium	4.5 - 6.9
Low	2 - 4.4
Informational	0 - 1.9

5. SCOPE

REPOSITORY

(a) Repository: earn-v2-core

(b) Assessed Commit ID: a0adc68

(c) Items in scope:

- src/periphery/predeposit/PredepostVaultOApp.sol
- src/periphery/predeposit/ShareDistributor.sol
- src/periphery/lib/PredepostVaultOAppStorageLib.sol
- src/implementation/ConcretePredepositVaultImpl.sol
- src/interface/IConcretePredepositVaultImpl.sol
- src/lib/storage/ConcretePredepositVaultImplStorageLib.sol
- periphery/auxiliary/TwoWayFeeSplitter.sol

Out-of-Scope: LayerZero protocol internals, existing vault and strategy logic.

REMEDIATION COMMIT ID:

- 354fe6b

Out-of-Scope: New features/implementations after the remediation commit IDs.

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW
0	1	0	1

INFORMATIONAL
2

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
INCONSISTENT REPLAY PREVENTION BETWEEN ORIGIN AND DESTINATION CHAINS CAN LEAD TO LOSS OF FUNDS	HIGH	SOLVED - 10/27/2025
UNCOMPACTED BATCH ARRAYS CAN EXCEED MESSAGE SIZE LIMITS POTENTIALLY INTRODUCING A DENIAL OF SERVICE	LOW	SOLVED - 10/27/2025
MISSING ARRAY LENGTH VALIDATION WHEN HANDLING BATCH CLAIMS	INFORMATIONAL	SOLVED - 10/27/2025
MISSING RECIPIENT ADDRESS VALIDATION DURING EMERGENCY WITHDRAWAL	INFORMATIONAL	SOLVED - 10/27/2025

7. FINDINGS & TECH DETAILS

7.1 INCONSISTENT REPLAY PREVENTION BETWEEN ORIGIN AND DESTINATION CHAINS CAN LEAD TO LOSS OF FUNDS

// HIGH

Description

The protocol's cross-chain predeposit and claim mechanism relies on coordinated logic between an origin chain (L1) and a destination chain (L2). When users move shares between chains, both sides must enforce consistent validation and replay protection rules to prevent multiple claims for the same balance.

However, the current implementation introduces an asymmetric replay prevention model. On the origin chain, users can trigger `claimOnTargetChain()` multiple times because there is no explicit check marking them as having already claimed. The function burns the user's shares and sends a cross-chain message to initiate distribution on the target chain. Meanwhile, on the destination chain, the `ShareDistributor` contract maintains a mapping that prevents multiple claims.

Additionally, this issue also exists within the batch claim logic. If a single user in a batch has already claimed their shares on L2, the `_handleBatchClaim()` function reverts the entire transaction when it encounters the `AlreadyClaimed` condition. Because this happens after all L1 changes are finalized, all users in that batch could lose their shares. The L1 state shows burned tokens, while the L2 distribution never occurs due to the revert, breaking the invariant that "L1 burned amount equals L2 distributed amount".

Code Location

The `ConcretePredepositVaultImpl::claimOnTargetChain` function and its batch claim variant allow multiple claims on the origin chain:

```
73  function claimOnTargetChain(bytes calldata options) external payable nonReentrant withYieldAcc
74      PDVLib.ConcretePredepositVaultImplStorage storage $ = PDVLib.fetch();
75
76      // Ensure self claims are enabled
77      require($.selfClaimsEnabled, SelfClaimsDisabled());
78
79      _validateClaimConditions($);
80
81      // Get user's current share balance
82      uint256 userShares = balanceOf(msg.sender);
83      require(userShares != 0, NoSharesToClaim());
84
85      // decrease cached totalAssets proportionally to the user's shares to maintain the share p
86      uint256 assets = userShares.calcConvertToAssets(totalSupply(), cachedTotalAssets(), Math.R
87      CachedVaultStateLib.fetch().cachedTotalAssets = cachedTotalAssets() - assets;
88
89      _burn(msg.sender, userShares);
90
91      // Store locked shares
92      $.lockedShares[msg.sender] += userShares;
93
94      bytes memory payload = abi.encode(MSG_TYPE_CLAIM, msg.sender, userShares);
95
96      // Send the message via the OApp (quote and fee validation done internally)
97      IPredepostVaultOApp($.oapp).send{value: msg.value}(payload, options, msg.sender);
98
99
```

```
    emit SharesClaimedOnTargetChain(msg.sender, userShares);
}
```

The `ShareDistributor::_handleSingleClaim` function and its batch claim variant restricts multiple claims on the destination chain:

```
function _handleSingleClaim(bytes calldata message, bytes32 guid) internal {
    // Decode single claim: msgType, user address, shares amount
    (, address user, uint256 shares) = abi.decode(message, (uint16, address, uint256));

    // Check if user has already claimed
    uint256 alreadyClaimed = claimedShares[user];
    if (alreadyClaimed != 0) {
        revert AlreadyClaimed(user, alreadyClaimed);
    }

    // Check if distributor has enough shares
    uint256 availableShares = IERC20(targetVault).balanceOf(address(this));
    if (availableShares < shares) {
        revert InsufficientShares(shares, availableShares);
    }

    // Record claimed amount before transfer
    claimedShares[user] = shares;

    // Transfer shares from distributor to user
    IERC20(targetVault).transfer(user, shares);

    // Emit event for tracking
    emit SharesDistributed(user, shares, guid);
}
```

The `ConcretePredepositVaultImpl::batchClaimOnTargetChain` function burns the shares on the origin chain before relaying the message call:

```
function batchClaimOnTargetChain(bytes calldata addressesData, bytes calldata options)
    external
    payable
    nonReentrant
    withYieldAccrual
    onlyRole(RolesLib.VAULT_MANAGER)
{
    PDVLib.ConcretePredepositVaultImplStorage storage $ = PDVLib.fetch();

    _validateClaimConditions($);

    // Decode addresses array
    address[] memory addresses = abi.decode(addressesData, (address[]));
    require(addresses.length > 0, EmptyAddressesArray());

    uint256[] memory sharesArray = new uint256[](addresses.length);
    uint256 totalShares = 0;

    for (uint256 i = 0; i < addresses.length; i++) {
        address user = addresses[i];
        require(user != address(0), InvalidUserAddress());

        uint256 userShares = balanceOf(user);
        if (userShares == 0) continue; // Skip users with no shares, already claimed, duplicat

        // decrease cached totalAssets proportionally to the user's shares to maintain the sha
        uint256 assets = userShares.calcConvertToAssets(totalSupply(), cachedTotalAssets(), Ma
        CachedVaultStateLib.fetch().cachedTotalAssets = cachedTotalAssets() - assets;

        _burn(user, userShares);

        // Store locked shares
        $.lockedShares[user] += userShares;

        // Store in batch arrays
        sharesArray[i] = userShares;
```

```
        totalShares += userShares;

        emit SharesClaimedOnTargetChain(user, userShares);
    }

    require(totalShares > 0, NoSharesInBatch());

    bytes memory payload = abi.encode(MSG_TYPE_BATCH_CLAIM, addresses, sharesArray);

    // Send the message via the OApp (quote and fee validation done internally)
    IPredepostVaultOApp($.oapp).send{value: msg.value}(payload, options, msg.sender);
}
```

The `ShareDistributor::_handleBatchClaim` function will continually revert if any of the users in the array has already claimed:

```
function _handleBatchClaim(bytes calldata message, bytes32 guid) internal {
    // Decode batch claim: msgType, addresses array, shares array
    (, address[] memory users, uint256[] memory sharesArray) = abi.decode(message, (uint16, ad

    // Process each user in the batch
    for (uint256 i = 0; i < users.length; i++) {
        if (sharesArray[i] == 0) continue; // Skip if no shares

        // Check if user has already claimed
        uint256 alreadyClaimed = claimedShares[users[i]];
        if (alreadyClaimed != 0) {
            revert AlreadyClaimed(users[i], alreadyClaimed);
        }

        // Check if distributor has enough shares
        uint256 availableShares = IERC20(targetVault).balanceOf(address(this));
        if (availableShares < sharesArray[i]) {
            revert InsufficientShares(sharesArray[i], availableShares);
        }

        // Record claimed amount before transfer
        claimedShares[users[i]] = sharesArray[i];

        // Transfer shares from distributor to user
        IERC20(targetVault).transfer(users[i], sharesArray[i]);
    }

    // Emit batch event for tracking
    emit BatchSharesDistributed(users, sharesArray, guid);
}
```

Proof of Concept

This POC demonstrates that users can lose funds permanently when the origin chain burns their shares but the destination chain rejects the claim due to replay protection:

```
pragma solidity ^0.8.24;
import {ConcretePredepositVaultImplBaseSetup} from "../../common/ConcretePredepositVaultImplBaseSetup
import {OptionsBuilder} from "@layerzerolabs/oapp-evm/contracts/oapp/libs/OptionsBuilder.sol";
import {IERC20} from "@openzeppelin-contracts/token/ERC20/IERC20.sol";
import {Origin} from "@layerzerolabs/oapp-evm/contracts/oapp/OApp.sol";
contract FIND001InconsistentReplayPreventionTest is ConcretePredepositVaultImplBaseSetup {
    using OptionsBuilder for bytes;
    address public alice;
    uint256 public constant INITIAL_DEPOSIT = 1000e18;
    error AlreadyClaimed(address user, uint256 previouslyClaimed);
    function setUp() public override {
        super.setUp();
        alice = makeAddr("alice");
        asset.mint(alice, INITIAL_DEPOSIT * 2);
    }
    function test_L1_allows_multiple_claims_but_L2_prevents_them() public {
        vm.startPrank(alice);
```

```
        asset.approve(address(predepositVault), INITIAL_DEPOSIT * 2);
        predepositVault.deposit(INITIAL_DEPOSIT, alice);
        vm.stopPrank();
        bytes memory firstMessage = abi.encode(uint16(1), alice, INITIAL_DEPOSIT);
        bytes32 firstGuid = keccak256("first-claim");
        Origin memory firstOrigin = Origin({srcEid: aEid, sender: addressToBytes32(address(predeposit
        vm.prank(address(endpoints[bEid]));
        distributor.lzReceive(firstOrigin, firstGuid, firstMessage, address(0), "");
        assertEq(IERC20(address(destinationVault)).balanceOf(alice), INITIAL_DEPOSIT);
        bytes memory secondMessage = abi.encode(uint16(1), alice, INITIAL_DEPOSIT);
        bytes32 secondGuid = keccak256("second-claim");
        Origin memory secondOrigin = Origin({srcEid: aEid, sender: addressToBytes32(address(predeposi
        vm.prank(address(endpoints[bEid]));
        vm.expectRevert(abi.encodeWithSelector(AlreadyClaimed.selector, alice, INITIAL_DEPOSIT));
        distributor.lzReceive(secondOrigin, secondGuid, secondMessage, address(0), "");
        assertEq(IERC20(address(destinationVault)).balanceOf(alice), INITIAL_DEPOSIT);
    }
}
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:M/D:H/Y:N (8.8)

Recommendation

Introduce explicit replay protection on the origin chain to align with the destination chain.

Remediation Comment

SOLVED: The **Blueprint Finance team** solved this issue by removing replay protection on L2 and allowing claim accumulation on both origin and destination chains.

Remediation Hash

https://github.com/Blueprint-Finance/earn-v2-core/commit/354fe6b5c3c53465e46f1fc706d79d8602343869

7.2 UNCOMPACTED BATCH ARRAYS CAN EXCEED MESSAGE SIZE LIMITS POTENTIALLY INTRODUCING A DENIAL OF SERVICE

// LOW

Description

The `batchClaimOnTargetChain()` function allows the contract to process multiple user claims in one transaction and send them as a single message through LayerZero's cross-chain messaging framework. The function currently constructs arrays (`addresses` and `sharesArray`) using the full input length even when many of those users hold zero shares or are duplicates.

Instead of compacting the data to include only valid claim entries, the logic encodes all entries (including those with zero shares) into the final payload. This can lead to excessively large payloads, since each address adds 20 bytes and each share value adds 32 bytes. LayerZero imposes a maximum payload size (10,000 bytes) for safety and efficiency. If this limit is exceeded, the message transmission reverts, blocking the entire batch claim from being processed.

Furthermore, sending unnecessarily large payloads wastes gas and increases cross-chain messaging costs.

Code Location

The `ConcretePredepositVaultImpl::batchClaimOnTargetChain` function only skips user addresses with zero share amount but does not prune the batch:

```
103 function batchClaimOnTargetChain(bytes calldata addressesData, bytes calldata options)
104     external
105     payable
106     nonReentrant
107     withYieldAccrual
108     onlyRole(RolesLib.VAULT_MANAGER)
109 {
110     PDVLib.ConcretePredepositVaultImplStorage storage $ = PDVLib.fetch();
111
112     _validateClaimConditions($);
113
114     // Decode addresses array
115     address[] memory addresses = abi.decode(addressesData, (address[]));
116     require(addresses.length > 0, EmptyAddressesArray());
117
118     uint256[] memory sharesArray = new uint256[](addresses.length);
119     uint256 totalShares = 0;
120
121     for (uint256 i = 0; i < addresses.length; i++) {
122         address user = addresses[i];
123         require(user != address(0), InvalidUserAddress());
124
125         uint256 userShares = balanceOf(user);
126         if (userShares == 0) continue; // Skip users with no shares, already claimed, duplicat
127
128         // decrease cached totalAssets proportionally to the user's shares to maintain the sha
129         uint256 assets = userShares.calcConvertToAssets(totalSupply(), cachedTotalAssets(), Ma
130         CachedVaultStateLib.fetch().cachedTotalAssets = cachedTotalAssets() - assets;
131
132         _burn(user, userShares);
133
134         // Store locked shares
135         $.lockedShares[user] += userShares;
136
```

```
137             // Store in batch arrays
138             sharesArray[i] = userShares;
139             totalShares += userShares;
140
141             emit SharesClaimedOnTargetChain(user, userShares);
142         }
143
144         require(totalShares > 0, NoSharesInBatch());
145
146         bytes memory payload = abi.encode(MSG_TYPE_BATCH_CLAIM, addresses, sharesArray);
147
148         // Send the message via the OApp (quote and fee validation done internally)
149         IPredepostVaultOApp($.oapp).send{value: msg.value}(payload, options, msg.sender);
150     }
```

BVSS

AO:A/AC:L/AX:M/R:N/S:U/C:N/A:M/I:N/D:N/Y:N (3.4)

Recommendation

Consider refactoring the function to compact arrays before encoding, including only users who have a positive share balance.

Remediation Comment

SOLVED: The **Blueprint Finance team** solved this issue by capping batch size at 150 users, keeping payloads under LayerZero's 10KB limit.

Remediation Hash

https://github.com/Blueprint-Finance/earn-v2-core/commit/354fe6b5c3c53465e46f1fc706d79d8602343869

7.3 MISSING ARRAY LENGTH VALIDATION WHEN HANDLING BATCH CLAIMS

// INFORMATIONAL

Description

The `_handleBatchClaim()` function on the destination chain decodes two arrays from the incoming LayerZero message: a list of users and a corresponding list of share amounts. The function assumes both arrays have identical lengths but never explicitly validates this assumption.

While the current implementation on the origin chain does encode equal-length arrays, this dependency makes the destination logic fragile. If future code changes cause the arrays to differ in length, this could lead to to inconsistent behavior and potential discrepancies in user distribution.

Code Location

Missing array length match validation in `ShareDistributor::_handleBatchClaim` function :

```
137 function _handleBatchClaim(bytes calldata message, bytes32 guid) internal {
138     // Decode batch claim: msgType, addresses array, shares array
139     (, address[] memory users, uint256[] memory sharesArray) = abi.decode(message, (uint16, ad
140 
141     // Process each user in the batch
142     for (uint256 i = 0; i < users.length; i++) {
143         if (sharesArray[i] == 0) continue; // Skip if no shares
144 
145         // Check if user has already claimed
146         uint256 alreadyClaimed = claimedShares[users[i]];
147         if (alreadyClaimed != 0) {
148             revert AlreadyClaimed(users[i], alreadyClaimed);
149         }
150 
151         // Check if distributor has enough shares
152         uint256 availableShares = IERC20(targetVault).balanceOf(address(this));
153         if (availableShares < sharesArray[i]) {
154             revert InsufficientShares(sharesArray[i], availableShares);
155         }
156 
157         // Record claimed amount before transfer
158         claimedShares[users[i]] = sharesArray[i];
159 
160         // Transfer shares from distributor to user
161         IERC20(targetVault).transfer(users[i], sharesArray[i]);
162     }
163 
164     // Emit batch event for tracking
165     emit BatchSharesDistributed(users, sharesArray, guid);
166 }
```

BVSS

AO:A/AC:L/AX:M/R:N/S:U/C:N/A:L/I:N/D:N/Y:N (1.7)

Recommendation

A check must be added at the start of `_handleBatchClaim` to ensure `users.length == sharesArray.length` , and the transaction must be reverted with a clear custom error (for example,

`ArrayLengthMismatch`) if a mismatch is detected. Iteration must be performed using the validated length, and explicit bounds-safe access must be enforced before reading `sharesArray[i]`.

Remediation Comment

SOLVED: The **Blueprint Finance team** solved this issue by adding a length check ensuring `users` and `sharesArray` arrays match before processing.

Remediation Hash

https://github.com/Blueprint-Finance/earn-v2-core/commit/354fe6b5c3c53465e46f1fc706d79d8602343869

7.4 MISSING RECIPIENT ADDRESS VALIDATION DURING EMERGENCY WITHDRAWAL

// INFORMATIONAL

Description

The `emergencyWithdraw()` function in the `ShareDistributor` contract allows the contract owner to withdraw tokens to any specified address but fails to validate the recipient parameter. This omission can lead to irreversible fund loss if the function is called with the zero address `(address(0))` or an incorrect recipient. Similar functions in the contract validate address parameters, but this one does not, creating an inconsistency and potential for human error. Additionally, no event is emitted to track emergency withdrawals.

Code Location

Absence of data validation on the recipient parameter of the `ShareDistributor::emergencyWithdraw` function:

```
181 function emergencyWithdraw(uint256 amount, address recipient) external onlyOwner {
182     IERC20(targetVault).transfer(recipient, amount);
183 }
```

BVSS

AO:S/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:H/Y:N (1.5)

Recommendation

Add checks to ensure the recipient is not the zero address and that the amount is greater than zero. Additionally, add the appropriate event emission.

Remediation Comment

SOLVED: The **Blueprint Finance team** solved this issue by hardcoding emergency withdrawals to `msg.sender` instead of accepting user input.

Remediation Hash

https://github.com/Blueprint-Finance/earn-v2-core/commit/354fe6b5c3c53465e46f1fc706d79d8602343869
