// Security Assessment

11.19.2025 - 11.20.2025

Earn v2- Whitelisting Hook

Blueprint Finance

HALBORN

Earn v2- Whitelisting Hook - Blueprint Finance

Prepared by: HALBORN

Last Updated 12/18/2025

Date of Engagement: November 19th, 2025 - November 20th, 2025

Summary

100% ⓘ OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
5	0	0	0	0	5

TABLE OF CONTENTS

1. INTRODUCTION

Blueprint Finance engaged Halborn to conduct a security assessment of their Earn V2 Core smart contracts , beginning on November 19th, 2025, and concluding on November 20th, 2025. The assessment focused on the updated contracts provided in the earn-v2-core GitHub repository, which was supplied to the Halborn team for a diff audit. The specific commit hash containing the changes under review, along with additional scope details, is outlined in the Scope section of this report.

The reviewed changes implement a new mechanism restricting Earn V2 participation exclusively to whitelisted addresses. This update ensures that only approved participants can interact with the Earn V2 system, while the rest of the protocol remains unchanged. The assessment focused solely on verifying the correctness, security, and integration of this whitelist feature with the existing Earn V2 Core architecture.

2. ASSESSMENT SUMMARY

Halborn was provided with 2 days for this engagement and assigned a full-time security engineer to assess the security of the smart contracts in scope. The assigned engineer possess deep expertise in blockchain and smart contract security, including hands-on experience with multiple blockchain protocols.

The objective of this assessment is to:

- Identify potential security issues within the Blueprint Finance protocol smart contracts.
- Ensure that smart contract of Blueprint Finance protocol functions operate as intended.

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

- Enforced whitelist restrictions on share transfers to prevent non-whitelisted users from acquiring shares indirectly.
- Applyed whitelist validation to the receiver parameter in deposit() and mint() to ensure only whitelisted addresses can receive shares.

3. TEST APPROACH AND METHODOLOGY

`Halborn` performed a combination of manual review of the code and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of the smart contract assessment. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of smart contracts 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 `Blueprint Finance` protocol.
- Manual code review and walkthrough of the `Blueprint Finance` in-scope contracts.
- Manual assessment of critical `Solidity` variables and functions to identify potential vulnerability classes.
- Manual testing using custom scripts.
- Static Analysis of security for scoped contract, and imported functions. (`Slither`).
- Local deployment and testing with (`Foundry` , `Remix IDE`).

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: 162409c

(c) Items in scope:

- src/periphery/hooks/WhitelistUserDepositHook.sol
- src/periphery/hooks/UserDepositCapHook.sol

Out-of-Scope: Third party dependencies and economic attacks.

REMEDIATION COMMIT ID:

- 354f1fb
- b8df5c0

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

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW
0	0	0	0

INFORMATIONAL
5

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
BYPASS OF WHITELIST RESTRICTIONS VIA SHARE TRANSFER	INFORMATIONAL	ACKNOWLEDGED - 11/25/2025

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
WHITELIST CHECK APPLIES TO MSG.SENDER INSTEAD OF RECEIVER, ALLOWING WHITELIST BYPASS	INFORMATIONAL	ACKNOWLEDGED - 11/25/2025
POTENTIAL OUT-OF-GAS RISK IN BATCH WHITELISTING AND UNWHITELISTING	INFORMATIONAL	ACKNOWLEDGED - 11/25/2025
MISSING ZERO-ADDRESS VALIDATION IN WHITELISTING	INFORMATIONAL	SOLVED - 11/24/2025
MISSING CHECK FOR ALREADY WHITELISTED USERS	INFORMATIONAL	SOLVED - 11/24/2025

7. FINDINGS & TECH DETAILS

7.1 BYPASS OF WHITELIST RESTRICTIONS VIA SHARE TRANSFER

// INFORMATIONAL

Description

In the `WhitelistUserDepositHook` contract, the whitelist check is applied only during `preDeposit()` and `preMint()`, where the caller (`sender`) must be listed as whitelisted. Transfers of ERC20 shares are not restricted by the protocol. After a whitelisted address deposits and receives shares, those shares can be transferred to a non-whitelisted address without restriction. Consequently, non-whitelisted parties can obtain shares indirectly, bypassing the intended whitelist controls.

This behavior violates the core requirement that only approved participants may interact with the Earn V2 system.

Code Location

In `WhitelistUserDepositHook` contract's `preDeposit` and `preMint` there are no share-transfer restrictions implemented anywhere in the system.

```
73  function preDeposit(address sender, uint256 assets, address receiver, uint256 totalAssets)
74      public
75      view
76      override
77  {
78      if (!whitelist[sender]) revert NotWhitelisted(sender);
79      super.preDeposit(sender, assets, receiver, totalAssets);
80  }
```

```
90  function preMint(address sender, uint256 shares, address receiver, uint256 totalAssets)
91      public
92      view
93      override
94  {
95      if (!whitelist[sender]) revert NotWhitelisted(sender);
96      super.preMint(sender, shares, receiver, totalAssets);
97  }
```

Proof of Concept

The following proof of concept demonstrates that the Earn V2 vault allows a non-whitelisted user to obtain shares and withdraw assets by receiving them through an ERC20 transfer, bypassing the whitelist enforced in `preDeposit()` and `preMint()`.

The steps of the proof of concept are as follows:

- Whitelist only Alice. Bob remains non-whitelisted.
- Alice deposits 500 assets, passes `preDeposit()` successfully, and receives vault shares.

- Bob attempts to deposit, which correctly reverts because he is not whitelisted.
- Alice transfers her shares to Bob, bypassing all whitelist checks because share transfers are unrestricted.
- Bob redeems his received shares, withdrawing assets from the vault even though he is not whitelisted.

This demonstrates that the whitelist validation on deposit/mint is insufficient, because unrestricted ERC20 share transfers allow non-whitelisted users to indirectly participate.

```
function test_POC_whitelistBypassViaTransfer() public {
        // Step 1: Whitelist Alice only (Bob is NOT whitelisted)
        address[] memory users = new address[](1);
        users[0] = alice;
        vm.prank(hookOwner);
        hook.whitelistUsers(users);

        // Verify Bob is NOT whitelisted
        assertFalse(hook.isWhitelisted(bob));
        assertTrue(hook.isWhitelisted(alice));

        // Step 2: Alice deposits assets and receives shares
        vm.startPrank(alice);
        asset.approve(address(concreteStandardVault), 500 ether);
        uint256 sharesReceived = concreteStandardVault.deposit(500 ether, alice);
        vm.stopPrank();

        // Verify Alice has shares
        assertEq(concreteStandardVault.balanceOf(alice), sharesReceived);
        assertGt(sharesReceived, 0);

        // Step 3: Bob tries to deposit directly - should fail (not whitelisted)
        vm.startPrank(bob);
        asset.approve(address(concreteStandardVault), 100 ether);
        vm.expectRevert(abi.encodeWithSelector(WhitelistUserDepositHook.NotWhitelisted.selecto
        concreteStandardVault.deposit(100 ether, bob);
        vm.stopPrank();

        // Step 4: Alice transfers shares to Bob (bypassing whitelist check)
        vm.prank(alice);
        concreteStandardVault.transfer(bob, sharesReceived);

        // Verify Bob now has shares without being whitelisted
        assertEq(concreteStandardVault.balanceOf(bob), sharesReceived);
        assertEq(concreteStandardVault.balanceOf(alice), 0);
        assertFalse(hook.isWhitelisted(bob)); // Bob is still NOT whitelisted

        // Step 5: Bob can now withdraw assets even though he's not whitelisted
        // This bypasses the whitelist restriction!
        uint256 bobShares = concreteStandardVault.balanceOf(bob);
        vm.prank(bob);
        uint256 assetsWithdrawn = concreteStandardVault.redeem(bobShares, bob, bob);

        // Verify Bob successfully withdrew assets
        assertGt(assetsWithdrawn, 0);
        assertEq(concreteStandardVault.balanceOf(bob), 0);
        assertGt(asset.balanceOf(bob), 0);
}
```

Command

```
forge test --mt test_POC_whitelistBypassViaTransfer -vvv
```

Output

```
Ran 1 test for test/periphery/unit-test/WhitelistUserDepositHook.t.sol:WhitelistUserDepositHookTest
[PASS] test_POC_whitelistBypassViaTransfer() (gas: 260986)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 6.69ms (854.08µs CPU time)
```

```
Ran 1 test suite in 107.60ms (6.69ms CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

BVSS

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

Recommendation

Whitelist validation must be enforced within the ERC20 `transfer` and `transferFrom` logic.

Remediation Comment

ACKNOWLEDGED: The **Blueprint Finance team** acknowledged this finding, noting that this is expected behavior. Only whitelisted users are permitted to deposit into the vault. The restriction applies solely to the depositor; however, a whitelisted user may still gift or transfer their shares to a non-whitelisted address. This behavior is not tied to KYC status; it is purely a whitelist-based deposit control. The team also mentioned that this aligns with their previous implementations of the whitelist feature.

7.2 WHITELIST CHECK APPLIES TO MSG.SENDER INSTEAD OF RECEIVER, ALLOWING WHITELIST BYPASS

// INFORMATIONAL

Description

In the `ConcreteStandardVaultImpl` the `deposit()` function invokes `preDeposit()` using `_msgSender()` as the `sender` argument. The whitelist hook then checks whether this caller is whitelisted but does not validate the `receiver`, which is actually receiving vault shares.

As a result, a whitelisted address can deposit on behalf of a non-whitelisted address, allowing the non-whitelisted user to obtain shares and bypass the core restriction that only approved participants should receive or hold Earn V2 shares. This breaks the intended access-control model and allows indirect participation by unapproved users.

Code Location

In the `ConcreteStandardVaultImpl` the `deposit()` function not invokes `preDeposit()` using `receiver`.

```
186 function deposit(uint256 assets, address receiver)
187     public
188     virtual
189     override
190 {
191     ...
192     if (h.checkIsValid(HooksLib.PRE_DEPOSIT)) {
193         h.preDeposit(
194             _msgSender(),               // @audit uses caller instead of receiver
195             assets,
196             receiver,
197             totalAssetsBeforeDeposit
198         );
199     }
200     ...
201     _deposit(_msgSender(), receiver, assets, shares);
202 }
```

Proof of Concept

The following proof of concept demonstrates that the Earn V2 vault allows a non-whitelisted receiver to obtain shares and withdraw assets by passing a whitelisted caller as the `msg.sender`, while the actual receiver is unwhitelisted. The whitelist hook validates only the caller (`sender`) instead of the `receiver`, enabling a full bypass of the whitelist restriction.

The steps of the proof of concept are as follows:

- Whitelist Alice only, leaving Bob non-whitelisted.
- Bob attempts to deposit, which correctly reverts because he is not whitelisted.
- Alice performs a deposit but sets `receiver = Bob`.
- The hook checks `whitelist[msg.sender]` → Alice is whitelisted → passes.

- Shares are minted to Bob, who is not whitelisted.
- Bob now holds valid vault shares despite not being whitelisted.
- Bob redeems the shares and withdraws assets, fully bypassing the intended access control.

This proves that using `msg.sender` for whitelist validation, rather than the actual `receiver`, allows unapproved addresses to participate in Earn V2 by leveraging a whitelisted intermediary.

```
function test_POC_whitelistBypassViaReceiverParameter_deposit() public {
        // Step 1: Whitelist Alice only (Bob is NOT whitelisted)
        address[] memory users = new address[](1);
        users[0] = alice;
        vm.prank(hookOwner);
        hook.whitelistUsers(users);

        // Verify whitelist status
        assertTrue(hook.isWhitelisted(alice));
        assertFalse(hook.isWhitelisted(bob));

        // Step 2: Bob tries to deposit directly - should fail (not whitelisted)
        vm.startPrank(bob);
        asset.approve(address(concreteStandardVault), 500 ether);
        vm.expectRevert(abi.encodeWithSelector(WhitelistUserDepositHook.NotWhitelisted.selecto
        concreteStandardVault.deposit(500 ether, bob);
        vm.stopPrank();

        // Step 3: Alice deposits but passes Bob as the receiver parameter
        // The hook checks whitelist[sender] = whitelist[alice] which passes
        // But shares are minted to Bob (receiver) who is NOT whitelisted!
        vm.startPrank(alice);
        asset.approve(address(concreteStandardVault), 500 ether);
        uint256 shares = concreteStandardVault.deposit(500 ether, bob); // Bob is the receiver
        vm.stopPrank();

        // Step 4: Verify Bob received shares even though he's NOT whitelisted
        assertEq(concreteStandardVault.balanceOf(bob), shares);
        assertEq(concreteStandardVault.balanceOf(alice), 0); // Alice has no shares
        assertFalse(hook.isWhitelisted(bob)); // Bob is still NOT whitelisted

        // Step 5: Bob can now redeem/withdraw the shares he received
        // This completely bypasses the whitelist restriction!
        vm.prank(bob);
        uint256 assetsWithdrawn = concreteStandardVault.redeem(shares, bob, bob);

        // Verify Bob successfully withdrew assets
        assertGt(assetsWithdrawn, 0);
        assertEq(concreteStandardVault.balanceOf(bob), 0);
        assertGt(asset.balanceOf(bob), 0);
    }
```

Command

```
forge test --mt test_POC_whitelistBypassViaReceiverParameter_deposit -vvv
```

Output

```
Ran 1 test for test/periphery/unit-test/WhitelistUserDepositHook.t.sol:WhitelistUserDepositHoo
[PASS] test_POC_whitelistBypassViaReceiverParameter_deposit() (gas: 252904)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 7.62ms (873.25µs CPU time)

Ran 1 test suite in 108.92ms (7.62ms CPU time): 1 tests passed, 0 failed, 0 skipped (1 total t
```

BVSS

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

Recommendation

The receiver must be validated, not only the caller.

Remediation Comment

ACKNOWLEDGED: The **Blueprint Finance team** acknowledged this finding, noting that this is expected behavior. Only whitelisted users are permitted to deposit into the vault. The restriction applies solely to the depositor; however, a whitelisted user may still gift or transfer their shares to a non-whitelisted address. This behavior is not tied to KYC status; it is purely a whitelist-based deposit control. The team also mentioned that this aligns with their previous implementations of the whitelist feature.

7.3 POTENTIAL OUT-OF-GAS RISK IN BATCH WHITELISTING AND UNWHITELISTING

// INFORMATIONAL

Description

In the `WhitelistUserDepositHook` the `whitelistUsers()` and `unWhitelistUsers()` functions, iterate over a user-supplied array and apply whitelist changes in a `for` loop without any upper bound. Since the size of the array is unrestricted, a large input can cause the transaction to exceed the block gas limit, resulting in an out-of-gas revert.

BVSS

AO:A/AC:L/AX:H/R:N/S:U/C:N/A:L/I:N/D:N/Y:N (0.8)

Recommendation

A reasonable maximum batch size should be implemented to prevent accidental out-of-gas failures.

Remediation Comment

ACKNOWLEDGED: The **Blueprint Finance team** acknowledged this finding, noting that they are already aware of this behavior. Since whitelisted users are finite and remain well within bounded limits, and given the limited associated risk (primarily gas cost), they are comfortable removing the additional check.

7.4 MISSING ZERO-ADDRESS VALIDATION IN WHITELISTING

// INFORMATIONAL

Description

In the `WhitelistUserDepositHook` the `_whitelistUser()` function allows `address(0)` to be added to or removed from the whitelist without any restriction.

BVSS

AO:A/AC:L/AX:H/R:N/S:U/C:N/A:L/I:N/D:N/Y:N (0.8)

Recommendation

Add validation to reject zero-address entries.

Remediation Comment

SOLVED: The **Blueprint Finance team** solved the issue in the specified commit id.

Remediation Hash

354f1fbf010540a16dff6d46319131f4ef4cd40a

7.5 MISSING CHECK FOR ALREADY WHITELISTED USERS

// INFORMATIONAL

Description

In the `WhitelistUserDepositHook` the `_whitelistUser()` function blindly sets the whitelist value without checking whether the address is already in the desired state. This results in redundant writes to storage.

BVSS

AO:A/AC:L/AX:H/R:N/S:U/C:N/A:L/I:N/D:N/Y:N (0.8)

Recommendation

Unnecessary writes should be prevented by verifying the current state prior to applying updates.

Remediation Comment

SOLVED: The **Blueprint Finance team** solved the issue in the specified commit id.

Remediation Hash

b8df5c0fc133a4c1588bd812cdb65aca6cd8de24
