// Security Assessment

02.04.2026 - 02.05.2026

Looping Strategy Swapper Contract

Blueprint Finance

HALBORN

Looping Strategy Swapper Contract - Blueprint Finance

Prepared by: HALBORN

Last Updated 02/13/2026

Date of Engagement: February 4th, 2026 - February 5th, 2026

Summary

100% OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

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

TABLE OF CONTENTS

1. INTRODUCTION

`Blueprint Finance` engaged `Halborn` to perform a security assessment of their smart contracts from February 4th to February 5th 2026. 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 audit scope consists of two standalone swap modules enabling bidirectional conversions between `syrupUSDC` vault shares and USDC. `SwapModuleUniV3` uses ERC4626 deposits and Uniswap V3 two-hop routing via WETH. `SwapModuleSyrupUSDCV4` uses ERC4626 deposits and Uniswap V4 swaps via Permit2. Both are permissionless periphery contracts with no owner controls or protocol integration.

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 are to:

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

In summary, `Halborn` did not identify any significant security risks. However, some enhancements were recommended to improve code clarity, consistency, or maintainability. These were successfully addressed by the `Blueprint Finance team`:

- `Replace balance-based accounting with delta tracking and add a token rescue function.`
- `Add constructor validation for fee and tickSpacing parameters.`

3. TEST APPROACH AND METHODOLOGY

`Halborn` conducted a combination of manual code review and automated security testing to balance efficiency, timeliness, practicality, and accuracy within the scope of this assessment. While manual testing is crucial for identifying flaws in logic, processes, and implementation, automated testing enhances coverage of smart contracts and quickly detects deviations from established security best practices.

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

- Research into the platform's architecture, purpose and use.
- Manual code review and walkthrough of smart contracts to identify any logical issues.
- Comprehensive assessment of the safety and usage of critical Solidity variables and functions within scope that could lead to arithmetic-related vulnerabilities.
- Local testing using custom scripts.
- Fork testing against main networks.
- Static security analysis of scoped contracts, and imported functions (`Slither`).

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

FILE

(a) Submitted File: earn-v2-core-fe9a6e281ef37d02c881d34fd641c94db2e939fe (1).zip

(b) Items in scope:

- /earn-v2-core-fe9a6e281ef37d02c881d34fd641c94db2e939fe/src/periphery/auxiliary/SwapModuleUniV3.sol
- /earn-v2-core-fe9a6e281ef37d02c881d34fd641c94db2e939fe/src/periphery/auxiliary/SwapModuleUniV4AndVaultDeposit.sol
- /earn-v2-core-fe9a6e281ef37d02c881d34fd641c94db2e939fe/src/periphery/interface/IUniswapV4.sol
- /earn-v2-core-fe9a6e281ef37d02c881d34fd641c94db2e939fe/src/periphery/interface/ISwapModule.sol

REMEDIATION COMMIT ID:

- 505b7b5
- d9a77fb

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
2

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
BALANCE-BASED ACCOUNTING IN V4 SWAP PATHS ENABLES OUTPUT AND REFUND INFLATION VIA PRE-EXISTING TOKENS	INFORMATIONAL	SOLVED - 02/05/2026
MISSING POOL PARAMETER VALIDATION IN V4 CONSTRUCTOR ALLOWS DEPLOYMENT WITH INVALID CONFIGURATION	INFORMATIONAL	SOLVED - 02/05/2026

7. FINDINGS & TECH DETAILS

7.1 BALANCE-BASED ACCOUNTING IN V4 SWAP PATHS ENABLES OUTPUT AND REFUND INFLATION VIA PRE-EXISTING TOKENS

// INFORMATIONAL

Description

The `SwapModuleSyrupUSDCV4` contract uses `balanceOf(address(this))` to determine swap output amounts and refund values, rather than computing deltas from the actual swap operation. This approach causes any tokens already present on the module to be included in the output or refund delivered to the next caller, enabling unintended token extraction.

BVSS

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

Recommendation

Consider replacing the balance-based output accounting with delta tracking that records the module's balance before the swap and computes only the difference afterward. Additionally, consider including a zapper function to rescue stuck tokens.

Remediation Comment

SOLVED: The **Blueprint Finance team** solved this finding in the specified commit by replacing absolute balance checks with before-and-after balance deltas, ensuring only tokens received from the actual swap operation are counted in outputs and refunds.

Remediation Hash

https://github.com/Blueprint-Finance/earn-v2-core/commit/505b7b5441177e1af3364c909a324ac58b2f7a54

7.2 MISSING POOL PARAMETER VALIDATION IN V4 CONSTRUCTOR ALLOWS DEPLOYMENT WITH INVALID CONFIGURATION

// INFORMATIONAL

Description

The `SwapModuleSyrupUSDCV4` constructor accepts pool parameters (fee and tickSpacing) without validating that they are non-zero or form a valid Uniswap V4 pool configuration. By contrast, the SwapModuleUniV3 constructor explicitly validates fee parameters before deployment. This inconsistency may result in deploying a non-functional swap module that permanently fails all swap operations.

BVSS

AO:A/AC:L/AX:M/R:P/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

Consider adding parameter validation to the V4 constructor to match the defensive pattern established in the V3 implementation.

Remediation Comment

SOLVED: The **Blueprint Finance team** solved this finding in the specified commit by adding constructor validation checks that reject `fee == 0` and `tickSpacing == 0`, preventing deployment with invalid pool parameters.

Remediation Hash

https://github.com/Blueprint-Finance/earn-v2-core/commit/d9a77fb9d71e7a22151a39d62c8568888d4fb759

Halborn strongly recommends conducting a follow-up assessment of the project either within six months or immediately following any material changes to the codebase, whichever comes first. This approach is crucial for maintaining the project's integrity and addressing potential vulnerabilities introduced by code modifications.