// Security Assessment

03.26.2025 - 03.28.2025

Morpho Strategy (Auto Compounding)

Blueprint Finance

HALBORN

Morpho Strategy (Auto Compounding) - Blueprint Finance

Prepared by: HALBORN

Last Updated 04/10/2025

Date of Engagement: March 26th, 2025 - March 28th, 2025

Summary

100% OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

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

TABLE OF CONTENTS

1. INTRODUCTION

`Blueprint Finance` engaged Halborn to conduct a security assessment on their smart contracts beginning on March 26th, 2025 and ending on March 28th, 2025. The security assessment was scoped to the smart contracts provided to the Halborn team.

2. ASSESSMENT SUMMARY

The team at Halborn was provided 3 days for the engagement and assigned a security engineer to evaluate the security of the smart contract.

The security engineer is a blockchain and smart-contract security expert with advanced penetration testing, smart-contract hacking, and deep knowledge of multiple blockchain protocols.

The purpose of this assessment is to:

- Ensure that smart contract functions operate as intended.
- Identify potential security issues with the smart contracts.

In summary, Halborn identified some improvements to reduce the likelihood and impact of risks, which have been addressed by the `Blueprint Finance team`. The main ones were the following:

- `Strengthen slippage handling using oracle-based pricing or by allowing user-defined thresholds.`
- `Add proper token approval logic to the 'addRewardToken' function to ensure compatibility and consistency with the initializer.`

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, purpose, and use of the platform.
- Smart contract manual code review and walkthrough to identify any logic issue.
- Thorough assessment of safety and usage of critical Solidity variables and functions in scope that could lead to arithmetic related vulnerabilities.
- Manual testing by custom scripts.
- Graphing out functionality and contract logic/connectivity/functions (`solgraph`).
- Static Analysis of security for scoped contract, and imported functions. (`Aderyn`).
- Local or public testnet deployment (`Foundry` , `Remix IDE`).ontent goes here.

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
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 (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
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

SEVERITY	SCORE VALUE RANGE
Informational	0 - 1.9

5. SCOPE

FILES AND REPOSITORY

(a) Repository: sc_earn-v1

(b) Assessed Commit ID: 6b0d0dd

(c) Items in scope:

- MorphoVaultStrategy.sol
- StrategyBase.sol

Out-of-Scope: concreteMultiStrategyVault.sol, WithdrawalQueueHelper.sol, MultiStrategyVaultHelper.sol, UniswapV3HelperV1.sol, third party dependencies and economic attacks. All code modifications not directly related to the scope in this report. (e.g., new features).

REMEDIATION COMMIT ID:

- 516b7d3
- 285a57e
- 55a86d4
- 69cdaee

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

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW
0	1	1	2

INFORMATIONAL
0

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
INCORRECT SLIPPAGE PROTECTION FOR TOKEN SWAPS	HIGH	SOLVED - 04/08/2025
MISSING TOKEN APPROVAL IN ADDREWARDTOKEN FUNCTION	MEDIUM	SOLVED - 04/08/2025
INCONSISTENT FEE COLLECTION DUE TO UNVERIFIED TOKEN TRANSFERS	LOW	SOLVED - 04/08/2025
MISSING APPROVAL RESET IN REMOVEREWARDTOKEN FUNCTION	LOW	SOLVED - 04/08/2025

7. FINDINGS & TECH DETAILS

7.1 INCORRECT SLIPPAGE PROTECTION FOR TOKEN SWAPS

// HIGH

Description

The **MorphoVaultStrategy** contract implements token swapping functionality that is vulnerable to sandwich attacks due to incorrect implementation of slippage protection. The issue occurs in the `_swapExactTokenToToken()` function where the minimum output amount calculation happens within the same transaction as the swap execution:

```
function _swapExactTokenToToken(address tokenIn, address tokenOut, uint256 amountIn, uint256 minAmountOut)
internal returns (uint256 swapAmountOut)
{
    if (tokenIn == tokenOut || amountIn == 0) {
        return 0;
    }
    // Approve Uniswap router to spend MORPHO tokens
    IERC20(tokenIn).forceApprove(uniswapRouter, amountIn);
    if (minAmountOut == 0) {
        //E @AUDIT compute expected output with quoter
        uint256 expectedOutput =
        UniswapV3HelperV1.getExpectedOutput(IQuoterV2(uniswapQuoter), tokenIn, tokenOut, amountIn, poolFee);
        // Calculate minimum output with slippage
        minAmountOut = expectedOutput.mulDiv(100_00 - MAX_SLIPPAGE, 100_00, Math.Rounding.Floor);
    }
    // Create swap parameters
    SwapParams memory params = SwapParams({
        tokenIn: tokenIn,
        tokenOut: tokenOut,
        amountIn: amountIn,
        minAmountOut: minAmountOut,
        poolFee: poolFee,
        recipient: address(this)
    });
    // Perform the swap
    try UniswapV3HelperV1.swapExactInputSingle(ISwapRouter(uniswapRouter), params) returns (uint256 swapAmount) {
        swapAmountOut = swapAmount;
    } catch {}
    // Reset approval
    IERC20(tokenIn).forceApprove(uniswapRouter, 0);
}
```

The **UniswapV3HelperV1** library defines a `MAX_SLIPPAGE` value but applies this to the quote received in the same transaction:

```
uint256 public constant UNISWAPV3_MAX_SLIPPAGE = 300; // 3% slippage

function getExpectedOutput(IQuoterV2 quoterV2, address tokenIn, address tokenOut, uint256 amountIn, uint24 poolFee)
external
returns (uint256 amountOut)
{
    IQuoterV2.QuoteExactInputSingleParams memory quoteExactInputSingleParams = IQuoterV2.QuoteExactInputSingleParams({
        tokenIn: tokenIn,
        tokenOut: tokenOut,
        amountIn: amountIn,
        fee: poolFee,
        sqrtPriceLimitX96: 0
    });
    validateFeeTier(poolFee);
```

```
    (amountOut,,,) = quoterV2.quoteExactInputSingle(quoteExactInputSingleParams);
}
```

When the strategy performs swaps (particularly during auto-compounding via `_autoCompoundRewards()`), attackers can:

1. Front-run the transaction to manipulate the pool price upward
2. Allow the victim transaction to execute with slippage calculated on the already manipulated price
3. Back-run the transaction to profit from the price difference

This results in direct value extraction from users' rewards and reduces the overall effectiveness of the yield optimization strategy. The issue is particularly severe because:

1. Auto-compounding operations happen regularly and predictably
2. The reward token (MORPHO) and base asset pair likely has limited liquidity, making price manipulation easier
3. The cumulative impact increases over time as rewards continually leak value

Proof of Concept

This test can be added to `morphoVaultStrategy.t.sol`:

```
function test_sandwichAttack_vulnerability() public {
        // Setup: deposit assets and prepare for auto-compounding
        uint256 depositAmount = 10 ether;
        _mintAsset(depositAmount, hazel);
        vm.prank(hazel);
        asset.approve(address(strategy), depositAmount);
        vm.prank(hazel);
        strategy.deposit(depositAmount, hazel);

        // Fund strategy with MORPHO tokens
        uint256 morphoAmount = 2 ether;
        deal(address(morphoToken), address(strategy), morphoAmount);

        // Enable auto-compounding
        vm.prank(configurator);
        strategy.setAutoCompoundingEnabled(true);
        vm.prank(configurator);
        strategy.setMinRewardAmountForCompounding(0.5 ether);

        // Record initial state
        uint256 initialAssets = strategy.totalAssets();

        // ATTACK SIMULATION
        // 1. Front-run: Manipulate price seen by quoter
        uint256 fairPrice = morphoAmount / 100; // 0.02 ETH for 2 MORPHO tokens
        uint256 manipulatedPrice = fairPrice / 2; // 50% worse price

        // Mock the quoter to return the manipulated price
        vm.mockCall(
            UNISWAP_QUOTER,
            abi.encodeWithSelector(IQuoterV2.quoteExactInputSingle.selector),
            abi.encode(manipulatedPrice, 0, 0, 0)
        );

        // Mock the approvals
        vm.mockCall(
            address(morphoToken),
            abi.encodeWithSelector(IERC20.approve.selector, UNISWAP_ROUTER, morphoAmount),
            abi.encode(true)
        );
```

```
        // Mock the router to return the manipulated price (swapped amount)
        vm.mockCall(
            UNISWAP_ROUTER,
            abi.encodeWithSelector(ISwapRouter.exactInputSingle.selector),
            abi.encode(manipulatedPrice)
        );

        // Update balances to simulate the swap
        uint256 stratWethBefore = asset.balanceOf(address(strategy));
        deal(WETH, address(strategy), stratWethBefore + manipulatedPrice);
        deal(address(morphoToken), address(strategy), 0);

        // 2. Execute victim transaction
        vm.prank(compounder);
        strategy.compoundRewards();

        // 3. Analyze the attack impact
        uint256 finalAssets = strategy.totalAssets();
        uint256 slippageThreshold = manipulatedPrice * (10000 - 300) / 10000; // 3% slippage on manipulated price

        console.log("Initial assets:", initialAssets);
        console.log("Final assets:", finalAssets);
        console.log("Fair price:", fairPrice);
        console.log("Manipulated price (what the strategy received):", manipulatedPrice);
        console.log("Slippage threshold (min acceptable to strategy):", slippageThreshold);
        console.log("Loss percentage from attack:", ((fairPrice - manipulatedPrice) * 100) / fairPrice, "%");

        // Verify the vulnerability - slippage protection was ineffective
        assertLt(slippageThreshold, fairPrice * 50 / 100, "Slippage protection threshold far too low");
        assertEq(finalAssets - initialAssets, manipulatedPrice, "Strategy accepted manipulated price");

        // Key point: The slippage protection calculated in _swapExactTokenToToken was based on the
        // already manipulated price, making it ineffective against sandwich attacks
        console.log("VULNERABILITY: Slippage was calculated based on already manipulated price!");
        console.log("Slippage threshold is only", (slippageThreshold * 100) / fairPrice, "% of fair value");
    }
```

Here are the results:

```
Ran 1 test for test/strategies/MorphoVaulStrategy.t.sol:MorphoVaultStrategyTest
[PASS] test_sandwichAttack_vulnerability() (gas: 1708415)
Logs:
  Initial assets: 9999999999999999999
  Final assets: 10009999999999999999
  Fair price: 20000000000000000
  Manipulated price (what the strategy received): 10000000000000000
  Slippage threshold (min acceptable to strategy): 9700000000000000
  Loss percentage from attack: 50 %
  VULNERABILITY: Slippage was calculated based on already manipulated price!
  Slippage threshold is only 48 % of fair value

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 38.08s (32.48s CPU time)

Ran 1 test suite in 38.09s (38.08s CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:H/Y:N (7.5)

Recommendation

It is recommended the following approaches:

1. Allow users to submit minimum amount of shares they want.
2. Use a TWAP from Uniswap or external oracle to determine fair prices.

3. Implement a separate function that only estimates swap output without executing the swap.

Remediation Comment

SOLVED: The **Blueprint Finance team** solved the issue by adding an oracle, checking the quoter price. Even if for a price feed, the same grace period is used, which is not recommended, the current fix is good.

Remediation Hash

516b7d3e2310540924ae351b91925dd41b303ee3

7.2 MISSING TOKEN APPROVAL IN ADDREWARDTOKEN FUNCTION

// MEDIUM

Description

The `addRewardToken` function in the `StrategyBase.sol` contract fails to execute the necessary token approval, despite a comment indicating this intention.
While the initialization function properly approves tokens, the subsequent method to add new reward tokens does not:

```
// In __StrategyBase_init function - proper approval is made
if (!rewardTokens_[i].token.approve(address(this), type(uint256).max)) revert ERC20ApproveFail();
```

```
// In addRewardToken function - missing approval
function addRewardToken(RewardToken calldata rewardToken_) external onlyOwner nonReentrant {
    // Ensure the reward token address is not zero, not already approved, and its parameters are correctly initialized.
    if (address(rewardToken_.token) == address(0)) {
        revert InvalidRewardTokenAddress();
    }
    if (rewardTokenApproved[address(rewardToken_.token)]) {
        revert RewardTokenAlreadyApproved();
    }
    if (rewardToken_.accumulatedFeeAccounted != 0) {
        revert AccumulatedFeeAccountedMustBeZero();
    }

    // Add the reward token to the list and approve it for unlimited spending by the strategy.
    rewardTokens.push(rewardToken_);
    rewardTokenApproved[address(rewardToken_.token)] = true;
    // Missing approval call here
}
```

The function correctly adds the token to the `rewardTokens` array and sets `rewardTokenApproved[address(rewardToken_.token)]` to true, but fails to execute the actual ERC20 approval that would allow the contract to transfer these tokens.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:M/I:N/D:N/Y:N (5.0)

Recommendation

It is recommended to modify the `addRewardToken` function to include the same approval mechanism present in the initializer function.

Remediation Comment

SOLVED: The **Blueprint Finance team** solved the issue by adding an approval when adding a token.

Remediation Hash

285a57e6377ddefb806654ada3501507023d419d

7.3 INCONSISTENT FEE COLLECTION DUE TO UNVERIFIED TOKEN TRANSFERS

// LOW

Description

In the `harvestRewards` function of the `StrategyBase` contract, there's a vulnerability related to how reward token fee transfers are handled. The function uses `TokenHelper.attemptSafeTransfer()` with `false` as the fourth parameter, which means it doesn't revert on failed transfers but only returns a boolean success indicator:

```
if (TokenHelper.attemptSafeTransfer(address(rewardAddress), feeRecipient, collectedFee, false)) {
    rewardTokens[i].accumulatedFeeAccounted += collectedFee;
    netReward = claimedBalance - collectedFee;
    emit Harvested(_vault, netReward);
}
```

The issue is that the code unconditionally trusts this success indicator without verifying if the actual amount transferred matches the intended amount. Some tokens might return `true` even when they transfer fewer tokens than requested or none at all, especially non-standard ERC20 implementations.

BVSS

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

Recommendation

It is recommended to enforce the strict validation of token transfers using `balanceBefore` / `balanceAfter` pattern.

Remediation Comment

SOLVED: The **Blueprint Finance team** solved the issue by adding the recommended pattern.

Remediation Hash

55a86d4ebd736412e4a919aa0de3a7302b3d593d

7.4 MISSING APPROVAL RESET IN REMOVEREWARDTOKEN FUNCTION

// LOW

Description

The `removeRewardToken` function in `StrategyBase.sol` removes tokens from the reward system but fails to reset the previously granted unlimited token approvals, creating a potential security risk:

```
function removeRewardToken(RewardToken calldata rewardToken_) external onlyOwner {
    // Ensure the reward token is approved before attempting removal.
    if (!rewardTokenApproved[address(rewardToken_.token)]) {
        revert RewardTokenNotApproved();
    }

    rewardTokens[_getIndex(address(rewardToken_.token))] = rewardTokens[rewardTokens.length - 1];
    rewardTokens.pop();

    // Mark the reward token as not approved.
    rewardTokenApproved[address(rewardToken_.token)] = false;
    // Missing approval reset
}
```

While the function correctly updates the internal tracking state by setting `rewardTokenApproved[address(rewardToken_.token)]` to `false`, it does not reset the actual ERC20 approval that was previously set to `type(uint256).max` during either the initialization or token addition process (if FIND-02 is remediated):

```
// Approve the strategy to spend the reward token without limit
if (!rewardTokens_[i].token.approve(address(this), type(uint256).max)) revert ERC20ApproveFail();
```

This creates a discrepancy between the contract's understanding of its approvals and the actual on-chain approval state.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:L/D:N/Y:N (2.5)

Recommendation

It is recommended to reset token approvals to zero when removing a reward token.

Remediation Comment

SOLVED: The **Blueprint Finance team** solved the issue by removing the approval.

Remediation Hash

69cdaee797e1d00b8c4b5814e8941e5021ef1370

8. AUTOMATED TESTING

Halborn used automated testing techniques to enhance the coverage of certain areas of the smart contracts in scope. Among the tools used was `Aderyn` , a Solidity static analysis framework.
After Halborn verified the smart contracts in the repository and was able to compile them correctly into their abis and binary format, Slither was run against the contracts. This tool can statically verify mathematical relationships between Solidity variables to detect invalid or inconsistent usage of the contracts' APIs across the entire code-base.

```
# Aderyn Analysis Report

This report was generated by [Aderyn](https://github.com/Cyfrin/aderyn), a static analysis tool built by [Cyfrin](https://cyfrin.io), a blockchain security company. This report i
s not a substitute for manual audit or security review. It should not be relied upon for any purpose other than to assist in the identification of potential security vulnerabilit
ies.
# Table of Contents

- [Summary](#summary)
  - [Files Summary](#files-summary)
  - [Files Details](#files-details)
  - [Issue Summary](#issue-summary)
- [High Issues](#high-issues)
  - [H-1: Return value of the function call is not checked.](#h-1-return-value-of-the-function-call-is-not-checked)
- [Low Issues](#low-issues)
  - [L-1: Centralization Risk for trusted owners](#l-1-centralization-risk-for-trusted-owners)
  - [L-2: Define and use `constant` variables instead of using literals](#l-2-define-and-use-constant-variables-instead-of-using-literals)
  - [L-3: Event is missing `indexed` fields](#l-3-event-is-missing-indexed-fields)
  - [L-4: Large literal values multiples of 10000 can be replaced with scientific notation](#l-4-large-literal-values-multiples-of-10000-can-be-replaced-with-scientific-notation)
  - [L-5: Dead Code](#l-5-dead-code)
  - [L-6: State variable could be declared immutable](#l-6-state-variable-could-be-declared-immutable)
```

All issues identified by `Aderyn` were proved to be false positives or have been added to the issue list in this report.

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.