// Security Assessment

04.28.2025 - 05.01.2025

Mamo Contracts

Moonwell

Mamo Contracts - Moonwell

Prepared by: HALBORN

Last Updated 05/20/2025

Date of Engagement: April 28th, 2025 - May 1st, 2025

Summary

100% OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
5	0	0	1	0	4

TABLE OF CONTENTS

1. Introduction

`Moonwell` engaged `Halborn` to conduct a security assessment on their smart contracts beginning on April 28th, 2025 and ending on May 2nd, 2025. The security assessment was scoped to the smart contracts provided in the **moonwell-fi/mamo-contracts/** GitHub repository. Commit hash and further details can be found in the Scope section of this report.

2. Assessment Summary

`Halborn` was provided 4 (four) days for the engagement, and assigned one full-time security engineer to review the security of the smart contracts in scope. The engineer is a blockchain and smart contract security expert with advanced penetration testing and smart contract hacking skills, and deep knowledge of multiple blockchain protocols.

The purpose of the assessment is to:

- Identify potential security issues within the smart contracts.
- Ensure that smart contract functionality operates as intended.

In summary, Halborn identified some improvements to reduce the likelihood and impact of risks, which were mostly acknowledged by the `Moonwell team`. The main ones were the following:

- `Move up the call to MamoStrategyRegistry.updateStrategyOwner, so it happens before super.transferOwnership is called.`
- `Assign critical roles to multi-signature wallets with transparent signing thresholds to reduce single-key compromise risk. Alternatively, implement a time-lock for sensitive operations.`

3. Test Approach And Methodology

`Halborn` performed a combination of manual and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of this assessment. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of the code and can quickly identify items that do not follow the security best practices. The following phases and associated tools were used during the assessment:

- Research into architecture and purpose.
- Smart contract manual code review and walkthrough.
- Graphing out functionality and contract logic/connectivity/functions (`solgraph`).
- Manual assessment of use and safety for the critical Solidity variables and functions in scope to identify any arithmetic related vulnerability classes.
- Manual testing by custom scripts.
- Static Analysis of security for scoped contract, and imported functions (`slither`).
- Testnet deployment (`Foundry`).

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 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.

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 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

FILES AND REPOSITORY

(a) Repository: mamo-contracts

(b) Assessed Commit ID: 0ced5c9

(c) Items in scope:

- src/token/ConfigurablePause.sol
- src/token/WormholeBridgeAdapter.sol
- src/token/WormholeTrustedSender.sol
- src/token/MintLimits.sol
- src/token/Mamo2.sol
- src/token/ConfigurablePauseGuardian.sol
- src/token/xERC20BridgeAdapter.sol
- src/token/Mamo.sol
- src/token/xERC20.sol
- src/ERC20MoonwellMorphoStrategy.sol
- src/SlippagePriceChecker.sol
- src/libraries/RateLimitedMidpointLibrary.sol
- src/libraries/RateLimitMidpointCommonLibrary.sol
- src/libraries/Math.sol
- src/libraries/GPv2Order.sol
- src/BaseStrategy.sol
- src/USDCStrategyFactory.sol
- src/ERC1967Proxy.sol
- src/MamoStrategyRegistry.sol

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

REMEDIATION COMMIT ID:

0 0 1 0

INFORMATIONAL
4

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
OWNERSHIP MISMATCH BLOCKS UPDATING STRATEGY OWNER	MEDIUM	SOLVED - 05/09/2025
MULTIPLE ACCESS-CONTROLLED FUNCTIONS POSE CENTRALIZATION RISK	INFORMATIONAL	ACKNOWLEDGED - 05/05/2025
MISSING CHECKS FOR ADDRESS(0) ASSIGNMENTS	INFORMATIONAL	ACKNOWLEDGED - 05/05/2025
RATE LIMIT BYPASS IN CROSSCHAINMINT	INFORMATIONAL	ACKNOWLEDGED - 05/09/2025
MISLEADING NATSPEC COMMENTS IN ERC20MOONWELLMORPHOSTRATEGY CONTRACT	INFORMATIONAL	ACKNOWLEDGED - 05/05/2025

7. FINDINGS & TECH DETAILS

7.1 OWNERSHIP MISMATCH BLOCKS UPDATING STRATEGY OWNER

// MEDIUM

Description

There is a logical mismatch between `BaseStrategy.transferOwnership` and `MamoStrategyRegistry.updateStrategyOwner` that effectively blocks any intended ownership transfer on previously deployed strategies.

In the `transferOwnership` function of `BaseStrategy` contract, the call to `super.transferOwnership(newOwner)` immediately updates the contract's owner to `newOwner`. Subsequently, the `updateStrategyOwner` in the `MamoStrategyRegistry` contract is called, also passing `newOwner` as address parameter.

- `BaseStrategy.sol`

```
    function transferOwnership(address newOwner) public override onlyOwner {
        super.transferOwnership(newOwner);
        mamoStrategyRegistry.updateStrategyOwner(newOwner);
    }
```

In the `updateStrategyOwner` the following check is performed:

- `MamoStrategyRegistry.sol`

```
        // Check if the caller is the current owner of the strategy
        require(isUserStrategy(currentOwner, strategy), "Not authorized to update strategy owner");
```

Since `currentOwner` is read from `Ownable(strategy).owner()`, it already returns `newOwner`, whereas the registry's `_userStrategies` mapping still references the old owner for that strategy.

This causes the check to fail and revert with "Not authorized to update strategy owner", **block ownership transfers for any strategy.** In the current implementation, users are unable to transfer ownership of existing strategies.

Remediation Comment

SOLVED: The `Moonwell` team has solved the issue as recommended.

Remediation Hash

https://github.com/moonwell-fi/mamo-contracts/commit/d986b8923b96fb15bc9feb6f5951a12348d1dbe3

7.2 MULTIPLE ACCESS-CONTROLLED FUNCTIONS POSE CENTRALIZATION RISK

// INFORMATIONAL

Description

A variety of contracts throughout the codebase (e.g., `Mamo.sol`, `Mamo2.sol`, `MamoStrategyRegistry.sol`, `ConfigurablePauseGuardian.sol`) rely on role-based or ownership-based access control for critical functions. Examples include:

- **MamoStrategyRegistry.sol**: `whitelistImplementation()`, `addStrategy()`, `upgradeStrategy()`, restricted by `onlyRole(DEFAULT_ADMIN_ROLE)` or `onlyRole(BACKEND_ROLE)`.
- **ConfigurablePauseGuardian.sol**: `pause()`, `unpause()`, restricted by a designated pause guardian.
- **Mamo.sol**: Administrative functions (`setBufferCap()`, `addBridges()`, etc.) guarded by `onlyOwner`.

While these controls are standard for protocol management, they also centralize power in specific addresses or roles.

In case these privileged accounts be compromised or behave maliciously, they could alter system parameters, pause the protocol, or upgrade implementations in ways that deviate from user interests.

BVSS

AO:S/AC:L/AX:L/R:N/S:U/C:N/A:H/I:H/D:N/Y:N (1.9)

Recommendation

Multi-Sig Governance: Assign critical roles to multi-signature wallets with transparent signing thresholds to reduce single-key compromise risk.

Time-Locked Upgrades: If feasible, subject major changes (e.g., upgrades) to a timelock, allowing users to exit or react to unexpected modifications.

Auditable Roles: Publicly document and track all role assignments. Make it easy for community members

7.3 MISSING CHECKS FOR ADDRESS(0) ASSIGNMENTS

// INFORMATIONAL

Description

In some locations throughout the codebase, addresses are assigned to state variables without verifying that they are non-zero. For example, when setting roles, updating guardians, or configuring critical contract addresses, there is no explicit `require(newAddress != address(0))` check.

If an unintentional zero address is used, it may disrupt the intended functionality—particularly for calls that assume a valid contract or EOA (e.g., ERC20 or bridging adapters).

```
Found in:
- Base Strategy (Line: 99)
- ConfigurablePauseGuardian (Line: 86)
- WormholeBridgeAdapter (Line: 91)
- xERC20BridgeAdapter (Line: 55)
```

BVSS

AO:S/AC:L/AX:L/R:N/S:U/C:N/A:M/I:M/D:N/Y:N (1.3)

Recommendation

Whenever assigning an address to a state variable (especially for roles, ownership, or bridging logic), use confirm that the parameter passed to the setter is not the `address(0)`.

Remediation Comment

ACKNOWLEDGED: The `Moonwell` team has acknowledged this finding.

7.4 RATE LIMIT BYPASS IN CROSSCHAINMINT

// INFORMATIONAL

Description

In the `Mamo.sol` contract, there is a function named `crossChainMint` allowing the `SUPERCHAIN_TOKEN_BRIDGE` address to mint tokens by directly calling `_mint`. Unlike the normal mint function (which goes through `xERC20.mint` and enforces rate limits and a `maxSupply` check), `crossChainMint` **does not** apply these constraints.

As a result, if the sequencer mis-behaves, could exceed the intended cap of "1 billion tokens," bypassing both:

- **Rate-limit** enforced by the `MintLimits` system.
- `maxSupply` check that normally prevents minting above a certain threshold.

This creates a discrepancy between the declared maximum supply and the actual token logic, undermining claims of a hard supply cap and subverting protections that limit inflation.

BVSS

AO:S/AC:L/AX:L/R:N/S:U/C:N/A:N/I:M/D:N/Y:N (1.0)

Recommendation

Route through Standard mint: Modify `crossChainMint` and `crossChainBurn` to invoke `xERC20.mint` (or `super.mint`) and `xERC20.burn` (or `super.burn`) so it inherits all rate-limit and max-supply checks.

Add Check: If a direct call is necessary, at least replicate the checks inside `mint/burn` functions, verifying `(totalSupply + amount <= maxSupply)` and calling `depleteBuffer()` for the `SUPERCHAIN_TOKEN_BRIDGE` address.

Document Intended Behavior: If you truly want to allow unlimited cross-chain minting for certain bridges, explicitly state this in the docs. However, that negates the premise of a "fixed supply."

7.5 MISLEADING NATSPEC COMMENTS IN ERC20MOONWELLMORPHOSTRATEGY CONTRACT

// INFORMATIONAL

Description

The ERC20MoonwellMorphoStrategy.sol contract contains two instances of misleading or incorrect NatSpec documentation that contradict the actual implementation of the functions:

- In the setFeeRecipient function (line ~300), the NatSpec comment states:

```
// @dev Only callable by the strategy owner
```

However, the function implements the onlyBackend modifier instead, allowing only the backend address to update the fee recipient, not the strategy owner.

- In the deposit function (line ~342), the NatSpec comment states:

```
   * Only callable by the user who owns this strategy
```

However, the function has no access control modifier, making it callable by any external address, not just the strategy owner.

BVSS

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

Recommendation

It is recommended to update the NatSpec comments to accurately reflect the actual implementation.

Remediation Comment

ACKNOWLEDGED: The `Moonwell` team has acknowledged this finding.

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 Slither, 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.

```
INFO:Detectors:
WormholeBridgeAdapter._bridgeOut(address,uint256,uint256,address) (src/token/WormholeBridgeAdapter.sol#172-191) sends eth to arbitrary user
        Dangerous calls:
        - wormholeRelayer.sendPayloadToEvm{value: cost}(targetChainId,targetAddress[targetChainId],abi.encode(to,amount),0,gasLimit) (src/token/WormholeBridgeAdapte
r.sol#181-188)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#functions-that-send-ether-to-arbitrary-destinations
INFO:Detectors:
Math.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#144-223) has bitwise-xor operator ^ instead of the exponentiation ope
rator **:
         - inverse = (3 * denominator) ^ 2 (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#205)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-exponentiation
INFO:Detectors:
ERC20MoonwellMorphoStrategy (src/ERC20MoonwellMorphoStrategy.sol#26-505) is an upgradeable contract that does not protect its initialize functions: ERC20MoonwellMor
phoStrategy.initialize(ERC20MoonwellMorphoStrategy.InitParams) (src/ERC20MoonwellMorphoStrategy.sol#137-172). Anyone can delete the contract with: UUPSUpgradeable.u
pgradeToAndCall(address,bytes) (lib/openzeppelin-contracts-upgradeable/contracts/proxy/utils/UUPSUpgradeable.sol#92-95)Reference: https://github.com/crytic/slither/
wiki/Detector-Documentation#unprotected-upgradeable-contract
INFO:Detectors:
Math.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#144-223) performs a multiplication on the result of a division:
        - denominator = denominator / twos (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#190)
        - inverse = (3 * denominator) ^ 2 (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#205)
Math.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#144-223) performs a multiplication on the result of a division:
        - denominator = denominator / twos (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#190)
        - inverse *= 2 - denominator * inverse (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#209)
Math.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#144-223) performs a multiplication on the result of a division:
        - denominator = denominator / twos (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#190)
        - inverse *= 2 - denominator * inverse (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#210)
Math.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#144-223) performs a multiplication on the result of a division:
        - denominator = denominator / twos (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#190)
        - inverse *= 2 - denominator * inverse (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#211)
Math.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#144-223) performs a multiplication on the result of a division:
        - denominator = denominator / twos (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#190)
        - inverse *= 2 - denominator * inverse (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#212)
Math.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#144-223) performs a multiplication on the result of a division:
        - denominator = denominator / twos (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#190)
        - inverse *= 2 - denominator * inverse (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#213)
Math.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#144-223) performs a multiplication on the result of a division:
        - denominator = denominator / twos (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#190)
        - inverse *= 2 - denominator * inverse (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#214)
Math.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#144-223) performs a multiplication on the result of a division:
        - prod0 = prod0 / twos (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#193)
        - result = prod0 * inverse (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#220)
Math.invMod(uint256,uint256) (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#243-289) performs a multiplication on the result of a division:
        - quotient = gcd / remainder (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#265)
        - (gcd,remainder) = (remainder,gcd - remainder * quotient) (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#267-274)
SlippagePriceChecker.getExpectedOutFromChainlink(address[],bool[],uint256[],uint256,address,address) (src/SlippagePriceChecker.sol#213-258) performs a multiplicatio
n on the result of a division:
        - _expectedOutFromChainlink = _expectedOutFromChainlink / (10 ** (_fromTokenDecimals - _toTokenDecimals)) (src/SlippagePriceChecker.sol#254)
        - _amountIntoThisIteration = _expectedOutFromChainlink (src/SlippagePriceChecker.sol#240)
```

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