// Security Assessment

03.21.2025 - 04.03.2025

Molecula Contracts

Molecula.io

HALBORN

Molecula Contracts - Molecula.io

Prepared by: HALBORN

Last Updated 04/30/2025

Date of Engagement: March 21st, 2025 - April 3rd, 2025

Summary

100% OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

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

TABLE OF CONTENTS

1. INTRODUCTION

`Molecula Protocol` engaged Halborn to conduct a security assessment on their smart contracts beginning on March 10th, 2025 and ending on April 3rd, 2025. The security assessment was scoped to the smart contracts provided to the Halborn team.

2. ASSESSMENT SUMMARY

The team at Halborn was provided 19 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 were either acknowledged and solved by the `Molecula Protocol team`, or marked as not applicable by Halborn after additional review. The main ones were the following:

- `Implement mechanisms to prevent front/back-running an oracle change.`
- `Implement a 2-Step ownership pattern.`

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. (`Slither` , `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
Informational	0 - 1.9

5. SCOPE

FILES AND REPOSITORY

(a) Repository: molecula-public

(b) Assessed Commit ID: aacc6e4

(c) Items in scope:

- common/ZeroValueChecker.sol
- common/rebase/structures/OperationStatus.sol
- common/rebase/structures/OperationInfo7540.sol
- common/rebase/RebaseERC20Permit.sol
- common/rebase/RebaseERC20.sol
- common/rebase/RebaseTokenCommon.sol
- common/mUSDLock.sol
- Nitrogen/RebaseToken.sol
- Nitrogen/AccountantAgent.sol
- core/MoleculaPool.sol
- core/SupplyManager.sol
- core/MoleculaPoolTreasury.sol

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

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

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	0	0	2	2

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
INCORRECT STATIC CALL IN MIGRATION FUNCTION BREAKS CROSS-CONTRACT MIGRATION	LOW	NOT APPLICABLE - 04/09/2025
SANDWICH ATTACK OPPORTUNITY IN ORACLE PRICE UPDATES	LOW	SOLVED - 04/09/2025
CENTRALIZATION OF PRIVILEGES	INFORMATIONAL	SOLVED - 04/09/2025
ABSENCE OF TWO-STEP OWNERSHIP TRANSFER PATTERN	INFORMATIONAL	ACKNOWLEDGED - 04/09/2025

7. FINDINGS & TECH DETAILS

7.1 INCORRECT STATIC CALL IN MIGRATION FUNCTION BREAKS CROSS-CONTRACT MIGRATION

// LOW

Description

In the `MoleculaPoolTreasury.migrate()` function, there's a issue when migrating from old MoleculaPoolTreasury contracts. The function calls `valueToRedeem()` on the old contract using a static call:

```
bytes memory result = oldMoleculaPool.functionStaticCall(
   abi.encodeWithSignature("valueToRedeem()")
);
// Get the old `valueToRedeem` in mUSD.
uint256 oldValueToRedeem = abi.decode(result, (uint256));
```

This approach only works with the original `MoleculaPool` contract where `valueToRedeem` is a state variable. In the newer `MoleculaPoolTreasury` implementation, this variable no longer exists in the same form. Instead, the `valueToRedeem` is stored within the token mapping:

```
// In MoleculaPoolTreasury
struct TokenInfo {
    TokenType tokenType;
    bool isBlocked;
    int8 n;
    uint32 arrayIndex;
    uint256 valueToRedeem; // This is now per-token
}
mapping(address => TokenInfo) public poolMap;
```

When attempting to migrate from one `MoleculaPoolTreasury` to another, the static call will fail or return 0, making the migration process incomplete.

BVSS

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

Recommendation

It is recommended to make it clear that this function is designated only for MoleculaPool and/or create another `migrate()` function specifically designed for new pool MoleculaPoolTreasury.

Remediation Comment

NOT APPLICABLE: Natspec comment in the interface mention that this function should only be used for old MoleculaPool to new one.

7.2 SANDWICH ATTACK OPPORTUNITY IN ORACLE PRICE UPDATES

// LOW

Description

The `RebaseERC20.sol` contract allows the owner to update the oracle address using the `setOracle` function. When this function is called, it changes the source from which token price/share information is derived. This creates an opportunity for sandwich attacks where users can exploit the price difference between the old and new oracle by executing trades immediately before and after the oracle update:

```
function setOracle(address oracleAddress) public onlyOwner checkNotZero(oracleAddress) {
    oracle = oracleAddress;
}
```

The issue arises because:

1. The `setOracle` transaction is visible in the mempool before being executed
2. The token price calculation depends directly on the oracle address via `convertToShares()` and `convertToAssets()`
3. There's no protection mechanism to prevent trades immediately before and after oracle updates

When the owner submits a transaction to update the oracle, an attacker observing the mempool can:

1. Submit a transaction with higher gas to execute before the oracle update
2. Submit another transaction to execute after the oracle update
3. Profit from any difference in valuations between the two oracles

BVSS

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

Recommendation

It is recommended to stop the protocol when doing this , pausing mechanisms should be used before and after the price update (in separate transaction).

Remediation Comment

SOLVED: The **Molecula team** has pausing system in place to protect this from happening that will be used during such updates.

7.3 CENTRALIZATION OF PRIVILEGES

// INFORMATIONAL

Description

The Molecula Protocol suffers from excessive centralization of control, giving contract owners extensive powers that could compromise the security and integrity of the protocol. Multiple contracts in the system rely heavily on `onlyOwner` access controls, allowing privileged accounts to manipulate critical protocol parameters, mint/burn tokens, and modify system states without restrictions.

Key centralization issues include:

1. In `RebaseERC20.sol`, the owner can arbitrarily mint and burn shares to/from any account
2. In `MoleculaPoolTreasury.sol`, the owner can add or remove tokens from the pool, block tokens, and manipulate the whitelist
3. In `SupplyManager.sol`, the owner can set agents, distribute yield, and change critical parameters

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 implement decentralization measures/multisigs wallets to mitigate these centralization risks.

Remediation Comment

SOLVED: The **Molecula team** implements a decentralized MoleculaRouter to manage minting/burning operations, and owner privileges will be transitioned to DAO governance.

7.4 ABSENCE OF TWO-STEP OWNERSHIP TRANSFER PATTERN

// INFORMATIONAL

Description

Multiple contracts including **RebaseERC20.sol**, **MoleculaPoolTreasury.sol**, and **SupplyManager.sol** inherit from OpenZeppelin's Ownable rather than **Ownable2Step**:

```
// MoleculaPoolTreasury.sol
contract MoleculaPoolTreasury is Ownable, IMoleculaPool, ZeroValueChecker {
    // ...
}

// SupplyManager.sol
contract SupplyManager is Ownable, ISupplyManager, IOracle, ZeroValueChecker {
    // ...
}
```

The single-step ownership transfer mechanism creates risk of permanently losing administrative control if the owner address is incorrectly specified during transfer.

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 implement OpenZeppelin's **Ownable2Step** pattern instead of the basic Ownable for all privileged contracts.

Remediation Comment

ACKNOWLEDGED: The **Molecula team** acknowledged the 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.

```
→  solidity git:(main) ✗ slither . --exclude-low --exclude-informational
'npx hardhat clean' running (wd: /Users/liliancariou/Desktop/Halborn/audits/molecula-public/blockchain/solidity)
'npx hardhat clean --global' running (wd: /Users/liliancariou/Desktop/Halborn/audits/molecula-public/blockchain/solidity)
'npx hardhat compile --force' running (wd: /Users/liliancariou/Desktop/Halborn/audits/molecula-public/blockchain/solidity)
INFO:Slither:. analyzed (90 contracts with 63 detectors), 0 result(s) found
→  solidity git:(main) ✗
```

All issues identified by 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.