// Private Smart Contract Security Assessment 07.22.2024 - 07.26.2024

V1
Eclipse Labs

HALBORN

V1 - Eclipse Labs

Prepared by: HALBORN

Last Updated 08/15/2024

Date of Engagement by: July 22nd, 2024 - July 26th, 2024

Summary

100% OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
4	0	0	0	1	3

TABLE OF CONTENTS

1. Introduction

`Eclipse` engaged Halborn to conduct a security reassessment on their smart contract beginning on July 22nd, 2024 and ending on July 26th, 2024. A security assessment on smart contracts was performed on the scoped smart contracts provided to the Halborn team.

2. Assessment Summary

The team at Halborn was provided one week for the engagement and assigned a full-time security engineer to verify 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 security risks that were accepted and acknowledged by the `Eclipse team`.

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.
- Testnet deployment (Foundry).

Out-Of-Scope

- External libraries and financial-related attacks.

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:

EXPLOITABILIY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Origin (AO)	Arbitrary (AO:A) Specific (AO:S)	1 0.2

EXPLOITABILIY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
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: syzygy

(b) Assessed Commit ID: 0fc9b79

(c) Items in scope:

- src/v1/Treasury.sol
- src/v1/Mailbox.sol
- src/v1/EtherBridge.sol
- src/v1/CommonContainer.sol

Out-of-Scope:

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

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	0	0	1	3

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
INSECURE RANDOMNESS IN GENERATERANDOMNONCE FUNCTION	LOW	RISK ACCEPTED
USE OF REVERT STRINGS INSTEAD OF CUSTOM ERRORS	INFORMATIONAL	ACKNOWLEDGED

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
CENTRALIZATION RISK: OWNERSHIP AND PAUSABLE	INFORMATIONAL	ACKNOWLEDGED
CENTRALIZATION RISK: EMERGENCY WITHDRAWAL	INFORMATIONAL	ACKNOWLEDGED

7. FINDINGS & TECH DETAILS

7.1 INSECURE RANDOMNESS IN GENERATERANDOMNONCE FUNCTION

// LOW

Description

The `generateRandomNonce` function uses predictable block properties to generate the nonce. This can be manipulated on `receiveMessage` function and set it as used on mapping `$.receiveNonces[secParam.nonce] = true`. In the future, when the nonce will match this encoded value, it is used before and the DoS is made on the function. However, since the `receiveMessage` function has proper access control, `onlyTrustedRelayer` the severity has been decreased. Finally, since the `lastNonce` is well-know when the contract has been initialized, then it is predictable the various block properties combination(block.number, block.timestamp and block.coinbase).

```
254 /// @dev Generates a pseudo-random nonce based on block properties and ch
255     /// This function combines previous nonce state with the hash of the
256     /// current block timestamp, and block difficulty to produce a new no
257     /// resistant to manipulation and harder to predict in advance given
258     /// @return Returns a 64-bit nonce generated from various entropy sou
259     function generateRandomNonce() internal returns (uint64) {
260         StorageV1 storage $ = _getStorageV1();
261         // Combine unpredictable block properties with the last nonce to
262         $.lastNonce = uint256(
263             keccak256(abi.encodePacked($.lastNonce, blockhash(block.numbe
264         );
265         return uint64($.lastNonce); // Return the least significant 64 bi
266     }
```

The initialize function used as first instance the `lastNonce` than could be predictable the various block combinations used for the next ones. Consider expecting a deterministic first nonce in order to be unpredictable.

```
115 /// @dev Initializes the mailbox contract
116     /// @param params the variable which contains all the fields needed t
117     function initialize(InitParams calldata params)
118         public
119         addressInitialized(params.initialOwner)
120         addressInitialized(params.trustedRelayer)
121         addressArrayInitialized(params.modules)
122         bytes32ArrayInitialized(params.ids)
```

```
123             bytesInitialized(bytes(params.ethereumChainId))
124             bytesInitialized(bytes(params.eclipseChainId))
125             initializer
126         {
127             require(params.ids.length == params.modules.length, "Module ids a
128
129             __AccessControl_init();
130             __ReentrancyGuard_init();
131             __UUPSUpgradeable_init();
132
133             // Grant roles and update modules by id
134             StorageV1 storage $ = _getStorageV1();
135             _grantRole(DEFAULT_ADMIN_ROLE, params.initialOwner);
136             _grantRole(TRUSTED_RELAYER_ROLE, params.trustedRelayer);
137
138             for (uint256 i = 0; i < params.modules.length; i++) {
139                 bytes32 id = params.ids[i];
140                 address module = params.modules[i];
141                 _grantRole(MODULE_ROLE, module);
142                 $.modulesById[id] = module;
143             }
144
145             // Store chain ids and initial nonce seed
146             $.ethereumChainId = generateChainId(params.ethereumChainId);
147             $.eclipseChainId = generateChainId(params.eclipseChainId);
148             $.lastNonce = generateRandomNonce();
149         }
```

Proof of Concept

The following `Foundry` test was used in order to prove the aforementioned issue:

```
// Functionality tests for `receiveMessage`
    function test_Mailbox_ReceiveMessageInsecurePseudoRandom() public {
        bytes memory message = abi.encodeWithSelector(MockModule.receiveCall.selector);
        uint256 init_value = 0;
        uint256 lastNonce = uint256(
            keccak256(abi.encodePacked(init_value, blockhash(block.number - 1),
block.timestamp, block.coinbase))
        ); //initialize contract is predictable.
        uint64 nonce = uint64(uint256(
            keccak256(abi.encodePacked(lastNonce, blockhash(block.number - 1),
block.timestamp+120, block.coinbase))
        ));
```

```
        bytes memory data = abi.encode(Mailbox.EclipseSecurityParam({nonce: nonce}));
        bytes memory chainId = abi.encode(keccak256(abi.encodePacked(eclipseChainId)));
        vm.startPrank(trustedRelayer);
        bool result = mailbox.receiveMessage(chainId, abi.encode(moduleIds[0]), message,
data);
        assertTrue(result);
        vm.warp(block.timestamp+120);
        mailbox.receiveMessage(chainId, abi.encode(moduleIds[0]), message, data);
//expected revert due to the nonce has been predicted, causing DoS.
    }
```

Evidence of the DoS:

BVSS

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

Recommendation

Consider to use additional entropy sources like user input, off-chain randomness, or oracle services.
References: Chainlink VRF.

Remediation Plan

RISK ACCEPTED: The **Eclipse team** accepted the risk of this issue.

7.2 USE OF REVERT STRINGS INSTEAD OF CUSTOM ERRORS

// INFORMATIONAL

Description

In Solidity smart contract development, replacing hard-coded revert message strings with the `Error()` syntax is an optimization strategy that can significantly reduce gas costs. Hard-coded strings, stored on the blockchain, increase the size and cost of deploying and executing contracts.
The `Error()` syntax allows for the definition of reusable, parameterized custom errors, leading to a more efficient use of storage and reduced gas consumption. This approach not only optimizes gas usage during deployment and interaction with the contract but also enhances code maintainability and readability by providing clearer, context-specific error information.

BVSS

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

Recommendation

It is recommended to replace hard-coded revert strings in require statements for custom errors, which can be done following the logic below.
1. Standard require statement (to be replaced):

```
require(condition, "Condition not met");
```

2. Declare the error definition to state

```
error ConditionNotMet();
```

3. As currently is not possible to use custom errors in combination with require statements, the standard syntax is:

```
if (!condition) revert ConditionNotMet();
```

More information about this topic in the official Solidity documentation.

Remediation Plan

ACKNOWLEDGED: The **Eclipse team** acknowledged this issue.

7.3 CENTRALIZATION RISK: OWNERSHIP AND PAUSABLE

// INFORMATIONAL

Description

Since `pause` and `unpause` functions are only accessible by the owner, the contract has a single point of failure if the owner account is compromised.

```
71  function pause() external virtual onlyOwner {
72          _pause();
73      }
74
75      function unpause() external virtual onlyOwner {
76          _unpause();
77      }
```

Score

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

Recommendation

Consider implementing multi-signature authorization for sensitive functions like `pause`, `unpause`, and `emergencyWithdraw`.

Remediation Plan

ACKNOWLEDGED: The **Eclipse team** acknowledged this issue.

7.4 CENTRALIZATION RISK: EMERGENCY WITHDRAWAL

// INFORMATIONAL

Description

The `emergencyWithdraw` function in the `Treasury` contract allows the owner to withdraw any amount of Ether from the contract at any time. This function can pose a centralization risk for several reasons:
Centralization Risks

1. **Single Point of Failure:** The `emergencyWithdraw` function is only accessible by the contract owner. If the owner's private key is compromised, the attacker can drain all funds from the contract.
2. **Potential for Abuse:** The owner has unilateral control over the funds. This central control can be abused if the owner acts maliciously or irresponsibly.
3. **Lack of Oversight:** There is no built-in mechanism to prevent or limit the owner's withdrawals. This lack of oversight can be problematic if the owner decides to withdraw funds for personal gain rather than for legitimate emergencies.

Score

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

Recommendation

To mitigate these centralization risks, several strategies can be implemented:
1. **Multi-Signature Wallets:** Use a multi-signature wallet for the contract ownership. This requires multiple parties to approve transactions, reducing the risk of a single point of failure.

```
address[] public owners;
uint256 public requiredApprovals;

modifier onlyOwners() {
    require(isOwner(msg.sender), "Not an owner");
    _;
}

function isOwner(address account) public view returns (bool) {
    for (uint256 i = 0; i < owners.length; i++) {
        if (owners[i] == account) {
            return true;
        }
    }
    return false;
}
```

2.Time-Locked Withdrawals: Implement a time lock for the `emergencyWithdraw` function. This would require a waiting period before the funds can be withdrawn, allowing users to exit or object to the transaction.

```
uint256 public withdrawalTimestamp;
uint256 public constant TIMELOCK = 1 days;

function emergencyWithdraw(uint256 amount) external onlyOwner {
    require(block.timestamp >= withdrawalTimestamp, "Withdrawal is timelocked");
    require(address(this).balance >= amount, "Insufficient funds");
    withdrawalTimestamp = block.timestamp + TIMELOCK;
    emit EmergencyTreasuryWithdraw(msg.sender, amount);
    (bool success,) = msg.sender.call{value: amount}("");
    require(success, "Failed to send Ether");
}

function startEmergencyWithdrawal() external onlyOwner {
    withdrawalTimestamp = block.timestamp + TIMELOCK;
}
```

3.Withdrawal Limits: Implement withdrawal limits to prevent large amounts of Ether from being withdrawn at once.

```
uint256 public constant MAX_WITHDRAWAL_AMOUNT = 10 ether;

function emergencyWithdraw(uint256 amount) external onlyOwner {
    require(amount <= MAX_WITHDRAWAL_AMOUNT, "Withdrawal amount exceeds the limit");
    require(address(this).balance >= amount, "Insufficient funds");
    emit EmergencyTreasuryWithdraw(msg.sender, amount);
    (bool success,) = msg.sender.call{value: amount}("");
    require(success, "Failed to send Ether");
}
```

Remediation Plan

ACKNOWLEDGED: The **Eclipse team** acknowledged this issue.

8. AUTOMATED TESTING

Halborn used automated testing techniques to enhance the coverage of certain areas of the scoped contracts. Among the tools used was Slither, a Solidity static analysis framework. After Halborn verified all the contracts in the repository and was able to compile them correctly into their ABI and binary formats, Slither was run on the all-scoped 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. It has been observed than every issues is marked as low.

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.