// Smart Contract Security Assessment 10.23.2024 - 10.25.2024

Cube Contracts

Layer3

HALBORN

Cube Contracts - Layer3

Prepared by: HALBORN

Last Updated 10/30/2024

Date of Engagement by: October 23rd, 2024 - October 25th, 2024

Summary

100% ⓘ OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

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

TABLE OF CONTENTS

1. Introduction

Layer3 engaged Halborn to conduct a security assessment on smart contracts beginning on *10/23/2024* and ending on *10/25/2024*. The security assessment was scoped to the smart contracts provided to the Halborn team.

2. Assessment Summary

The team at Halborn dedicated 3 working days for the engagement and assigned one full-time 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 minor improvements to reduce the likelihood and impact of risks, which were mostly addressed by the `Layer3 team`. The main ones were the following:

- Returning excess funds while minting using the native token.
- Best Practices and Validations.

3. Test Approach And Methodology

Halborn performed a combination of manual, semi-automated and automated security testing to balance efficiency, timeliness, practicality, and accuracy regarding 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 security best practices. The following phases and associated tools were used throughout the term of the assessment:

- Research into architecture and purpose.
- Smart contract manual code review and walk-through.
- Manual assessment of use and safety for the critical Solidity variables and functions in scope to identify any vulnerability classes
- Manual testing by custom scripts.
- Static Analysis of security for scoped contract, and imported functions. (`Slither`)
- Local deployment and testing (`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:

EXPLOITABILIY 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

EXPLOITABILIY 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 (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I: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

FILES AND REPOSITORY

(a) Repository: cubes

(b) Assessed Commit ID: 4b1b3d7

(c) Items in scope:

- src/CUBE.sol

Out-of-Scope:

REMEDIATION COMMIT ID:

- 79ac4dc

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

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	0	1	3	0

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
POTENTIAL DOS DUE TO REVERTS ON ZERO AMOUNT TREASURY TRANSFERS	MEDIUM	SOLVED - 10/28/2024
EXCESS ETH NOT REFUNDED IN NATIVE PAYMENT MINTING	LOW	RISK ACCEPTED - 10/28/2024
ZERO ADDRESS VALIDATION MISSING IN INITIALIZE FUNCTION	LOW	SOLVED - 10/28/2024

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
INCORRECT BALANCE REPORTING IN WITHDRAWAL EVENT	LOW	SOLVED - 10/28/2024

7. FINDINGS & TECH DETAILS

7.1 POTENTIAL DOS DUE TO REVERTS ON ZERO AMOUNT TREASURY TRANSFERS

// MEDIUM

Description

In the `_processL3Payouts` function, when the total fee payments to recipients equal the full price, the contract attempts to transfer 0 tokens to the treasury. This can cause the entire minting transaction to fail for tokens that revert on zero-value transfers.

```
(bool success, bytes memory returnData) = s_l3Token.call(
    abi.encodeWithSelector(TRANSFER_ERC20, msg.sender, s_treasury, data.pri
);
```

Impact:

- When `totalAmount` equals `data.price`, the transfer amount becomes 0
- Many ERC20 tokens (e.g., LEND, ZRX, BNB) revert to 0-value transfers
- This causes the entire `mintCube` transaction to fail
- Affects cases where fee recipients' total BPS equals 10000 (100%)

BVSS

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

Recommendation

Consider skipping 0 transfers:

```
uint256 treasuryAmount = data.price - totalAmount;
    if (treasuryAmount > 0) {
        (bool success, bytes memory returnData) = s_l3Token.call(
            abi.encodeWithSelector(TRANSFER_ERC20, msg.sender, s_treasury,
        );
        if (!success || (returnData.length > 0 && !abi.decode(returnData, (
            revert CUBE__ERC20TransferFailed();
        }
    }
```

Remediation

SOLVED: The suggested mitigation was implemented by **Layer3 team.**

Remediation Hash

79ac4dc538d3a75a7356e4953acce6e60898446c

7.2 EXCESS ETH NOT REFUNDED IN NATIVE PAYMENT MINTING

// LOW

Description

The `mintCube()` function accepts native ETH payments but fails to return excess ETH sent by users above the required price.

```
function mintCube(CubeData calldata cubeData, bytes calldata signature)
    external
    payable
    nonReentrant
{
    if (!s_isMintingActive) {
        revert CUBE__MintingIsNotActive();
    }
    if (cubeData.isNative) {
        if (msg.value < cubeData.price) {
            revert CUBE__FeeNotEnough();
        }
    }
    // ... rest of the function
}
```

- Users who accidentally send more ETH than the mint price will have their excess funds trapped in the contract
- The only way to recover these funds is through the admin's `withdraw()` function
- Users must trust and rely on admin intervention to recover their funds

BVSS

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

Recommendation

Implement excess ETH return functionality in the `mintCube()` function:

```
function mintCube(CubeData calldata cubeData, bytes calldata signature)
    external
    payable
    nonReentrant
{
    if (!s_isMintingActive) {
        revert CUBE__MintingIsNotActive();
    }
```

```
    if (cubeData.isNative) {
        // Check for minimum required payment
        if (msg.value < cubeData.price) {
            revert CUBE__FeeNotEnough();
        }

        // Store original msg.value for refund calculation
        uint256 excessAmount = msg.value - cubeData.price;

        // Process mint with exact price
        _mintCube(cubeData, signature);

        // Return excess ETH if any
        if (excessAmount > 0) {
            (bool success,) = msg.sender.call{value: excessAmount}("");
            if (!success) {
                revert CUBE__RefundFailed();
            }
        }
    } else {
        _mintCube(cubeData, signature);
    }
}
```

Remediation

RISK ACCEPTED: The **Layer3 team** decided to acknowledge this finding with the following remark: *"The excessive amount isn't returned deliberately and we made it like that from the beginning. If someone sends in a lot accidentally, we can return if they contact us."*

7.3 ZERO ADDRESS VALIDATION MISSING IN INITIALIZE FUNCTION

// LOW

Description

The `initialize()` function lacks zero address validation for the `_admin` parameter. If address(0) is passed as the `_admin` parameter, it will be granted the DEFAULT_ADMIN_ROLE.
Since this is an initializer function that can only be called once, this error cannot be corrected after deployment without a contract upgrade.

BVSS

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

Recommendation

Add zero address validation at the beginning of the initialize function.

```
function initialize(
    string memory _tokenName,
    string memory _tokenSymbol,
    string memory _signingDomain,
    string memory _signatureVersion,
    address _admin
) external initializer {
    if(_admin == address(0)) revert CUBE__InvalidAdminAddress();

    __ERC721_init(_tokenName, _tokenSymbol);
    __EIP712_init(_signingDomain, _signatureVersion);
    __AccessControl_init();
    __UUPSUpgradeable_init();
    __ReentrancyGuard_init();
    s_isMintingActive = true;

    _grantRole(DEFAULT_ADMIN_ROLE, _admin);
}
```

Remediation

SOLVED: The suggested mitigation was implemented by **Layer3 team.**

Remediation Hash

79ac4dc538d3a75a7356e4953acce6e60898446c

7.4 INCORRECT BALANCE REPORTING IN WITHDRAWAL EVENT

// LOW

Description

The `withdraw()` function emits an event with an incorrect balance parameter, leading to inaccurate transaction logging and potential monitoring issues. The vulnerability lies in using the contract's balance after the withdrawal for the event emission.

```
function withdraw() external onlyRole(DEFAULT_ADMIN_ROLE) {
    (bool success,) = msg.sender.call{value: address(this).balance}("");
    if (!success) {
        revert CUBE__WithdrawFailed();
    }
    emit ContractWithdrawal(address(this).balance); // @audit Incorrect
}
```

- The event uses `address(this).balance` after the withdrawal
- At this point, the balance is already 0
- The event will always emit a value of 0
- This makes it impossible to track actual withdrawal amounts

BVSS

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

Recommendation

Cache the balance before withdrawal:

```
function withdraw() external onlyRole(DEFAULT_ADMIN_ROLE) {
    uint256 withdrawAmount = address(this).balance;

    (bool success,) = msg.sender.call{value: withdrawAmount}("");
    if (!success) {
        revert CUBE__WithdrawFailed();
    }

    emit ContractWithdrawal(withdrawAmount);
}
```

Remediation

SOLVED: The suggested mitigation was implemented by **Layer3 team.**

Remediation Hash

79ac4dc538d3a75a7356e4953acce6e60898446c

8. AUTOMATED TESTING

Introduction

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.
The security team conducted a comprehensive review of all findings generated by the Slither static analysis tool.

```
INFO:Detectors:
CUBE._processNativePayouts(CUBE.CubeData) (src/CUBE.sol#483-530) sends eth to arbitrary user
        Dangerous calls:
        - (payoutSuccess,None) = recipient.call{value: referralAmount}() (src/CUBE.sol#513)
        - (success,None) = address(s_treasury).call{value: data.price - totalReferrals}() (src/CUBE.sol#526)
CUBE.withdraw() (src/CUBE.sol#683-689) sends eth to arbitrary user
        Dangerous calls:
        - (success,None) = msg.sender.call{value: address(this).balance}() (src/CUBE.sol#684)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#functions-that-send-ether-to-arbitrary-destinations
```

```
INFO:Detectors:
CUBE._processNativePayouts(CUBE.CubeData).totalReferrals (src/CUBE.sol#487) is a local variable never initialized
CUBE._processL3Payouts(CUBE.CubeData).totalAmount (src/CUBE.sol#426) is a local variable never initialized
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#uninitialized-local-variables
```

```
INFO:Detectors:
CUBE.setTreasury(address)._treasury (src/CUBE.sol#668) lacks a zero-check on :
                - s_treasury = _treasury (src/CUBE.sol#669)
CUBE.setL3TokenAddress(address)._l3 (src/CUBE.sol#676) lacks a zero-check on :
                - s_l3Token = _l3 (src/CUBE.sol#677)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#missing-zero-address-validation
INFO:Detectors:
Reentrancy in CUBE.withdraw() (src/CUBE.sol#683-689):
        External calls:
        - (success,None) = msg.sender.call{value: address(this).balance}() (src/CUBE.sol#684)
        Event emitted after the call(s):
        - ContractWithdrawal(address(this).balance) (src/CUBE.sol#688)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-3
```

After careful examination and consideration of the flagged issues, it was determined that within the project's specific context and scope, all were false positives. Including the reentrancy, as the call is being made to trusted address.

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.