// HALBORN

21.co - Wrapped Assets

Smart Contract Security Assessment

Prepared by: **Halborn**

Date of Engagement: **July 7th, 2023 - July 17th, 2023**

Visit: **Halborn.com**

DOCUMENT REVISION HISTORY

VERSION	MODIFICATION	DATE	AUTHOR
0.1	Draft Version	07/18/2023	Gabi Urrutia
1.0	Remediation Plan	08/16/2023	Gabi Urrutia

CONTACTS

CONTACT	COMPANY	EMAIL
Rob Behnke	Halborn	Rob.Behnke@halborn.com
Steven Walbroehl	Halborn	Steven.Walbroehl@halborn.com
Gabi Urrutia	Halborn	Gabi.Urrutia@halborn.com

EXECUTIVE OVERVIEW

1.1 INTRODUCTION

The project is a fork of the CMTA token (CMTAT).

The CMTAT is an open standard from the Capital Markets and Technology Association (CMTA), and the product of collaborative work by leading organizations in the Swiss finance and technology ecosystem.

21.co engaged Halborn to conduct a security assessment on their smart contracts beginning on July 7th, 2023 and ending on July 17th, 2023. The security assessment was scoped to the smart contracts provided in the amun/CMTAT GitHub repository. Commit hashes and further details can be found in the Scope section of this report.

1.2 ASSESSMENT SUMMARY

Halborn was provided 2 weeks for the engagement and assigned a team of full-time security engineers to verify the security of the smart contracts in scope. The security team consists of a blockchain and smart contract security experts with advanced penetration testing and smart contract hacking skills, and deep knowledge of multiple blockchain protocols.

The purpose of the assessments is to:

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

In summary, Halborn identified some security risks, which were mostly addressed by 21.co. The main ones were the following:

- Allowance confirmation was removed and standard ERC20 compliant `increaseAllowance()` was used instead.
- Pragma version was locked.

1.3 TEST APPROACH & 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.
- Scanning of solidity files for vulnerabilities, security hot-spots or bugs (MythX).
- Static Analysis of security for scoped contract, and imported functions (Slither).
- Testnet deployment (Foundry, Brownie).

2. 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.

2.1 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:

$$E = \prod m_e$$

2.2 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)	0
	Low (I:L)	0.25
	Medium (I:M)	0.5
	High (I:H)	0.75
	Critical (I:C)	1
Integrity (I)	None (I:N)	0
	Low (I:L)	0.25
	Medium (I:M)	0.5
	High (I:H)	0.75
	Critical (I:C)	1
Availability (A)	None (A:N)	0
	Low (A:L)	0.25
	Medium (A:M)	0.5
	High (A:H)	0.75
	Critical	1
Deposit (D)	None (D:N)	0
	Low (D:L)	0.25
	Medium (D:M)	0.5
	High (D:H)	0.75
	Critical (D:C)	1
Yield (Y)	None (Y:N)	0
	Low (Y:L)	0.25
	Medium: (Y:M)	0.5
	High: (Y:H)	0.75
	Critical (Y:H)	1

Impact I is calculated using the following formula:

$$I = max(m_I) + \frac{\sum m_I - max(m_I)}{4}$$

2.3 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.

Coefficient (C)	Coefficient Value	Numerical Value
Reversibility (r)	None (R:N)	1
	Partial (R:P)	0.5
	Full (R:F)	0.25
Scope (s)	Changed (S:C)	1.25
	Unchanged (S:U)	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

2.4 SCOPE

Code repositories:

1. Project Name

- Repository: amun/CMTAT
- Branch: wrapped-assets
- Commit ID: 27bb31952e76b4fa5b9d1c149fc2a83aec6cf13a
- Smart contracts in scope:
    1. contracts/modules/CMTAT_BASE.sol
        1. contracts/modules/security/AuthorizationModule.sol
        2. contracts/modules/wrapper/mandatory/BaseModule.sol
        3. contracts/modules/wrapper/mandatory/BurnModule.sol
        4. contracts/modules/wrapper/mandatory/ERC20BaseModule.sol
        5. contracts/modules/wrapper/mandatory/MintModule.sol
        6. contracts/modules/wrapper/mandatory/PauseModule.sol
- Remediations pull requests:
    1. (HAL-01) Current Allowance Confirmation Can Be Bypassed
    2. (HAL-04) Floating Pragma

Out-of-scope

- Third-party libraries and dependencies.
- Economic attacks.

3. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	0	0	2	2

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
(HAL-01) CURRENT ALLOWANCE CONFIRMATION CAN BE BYPASSED	Low (3.3)	SOLVED - 08/08/2023
(HAL-02) THE AUTHORIZATION MODULE DOES NOT FOLLOW SECURITY BEST PRACTICES	Low (2.0)	RISK ACCEPTED
(HAL-03) ALLOWANCE CAN BE MODIFIED WHILE CONTRACT IS PAUSED	Informational (0.0)	ACKNOWLEDGED
(HAL-04) FLOATING PRAGMA	Informational (0.0)	SOLVED - 08/08/2023

FINDINGS & TECH DETAILS

4.1 (HAL-01) CURRENT ALLOWANCE CONFIRMATION CAN BE BYPASSED - LOW (3.3)

Description:

The ERC20BaseModule contract has a non-standard approve function that requires the users to confirm their current allowance before being able to change it.

However, because this function does not override the standard approve function, this restriction can be easily bypassed by calling the standard public approve() function from the ERC20 module by OpenZeppelin.

It is also important to note that if this function is used to prevent the ERC20 race condition where a user can front-run the approve function and spend the allowance, the standard ERC20 already offers the increaseAllowance() method to circumvent this.

Code Location:

Listing 1: src/modules/wrapper/mandatory/ERC20BaseModule.sol (Line 100)

```
 97 function approve(
 98     address spender,
 99     uint256 amount,
100     uint256 currentAllowance
101 ) public virtual returns (bool) {
102     if (allowance(_msgSender(), spender) != currentAllowance) {
103         revert Errors.WrongAllowance(
104             allowance(_msgSender(), spender),
105             currentAllowance
106         );
107     }
108     super.approve(spender, amount);
109     return true;
110 }
```

BVSS:

AO:A/AC:L/AX:H/C:N/I:N/A:N/D:C/Y:N/R:N/S:U (3.3)

Recommendation:

Override the standard approve function or use increaseAllowance() if the goal is to prevent approve() front-running.

Remediation:

SOLVED: The 21.co team fixed the issue by removing the custom approve() function and using the increaseAllowance() instead in pull request #10.

4.2 (HAL-02) THE AUTHORIZATION MODULE DOES NOT FOLLOW SECURITY BEST PRACTICES - LOW (2.0)

Description:

The AuthorizationModule contract inherits from OpenZeppelin's AccessControl library. However, this library does not follow some considered security best practices, for example, the DEFAULT_ADMIN_ROLE is also its own admin, meaning it has permissions to grant and revoke this role.

BVSS:

AO:S/AC:L/AX:L/C:N/I:N/A:C/D:N/Y:N/R:N/S:U (2.0)

Recommendation:

Consider following security best practices and OpenZeppelin's recommendations, and use the AccessControlDefaultAdminRules extension to enforce additional security measures over this role.

Remediation:

RISK ACCEPTED: The 21.co team accepted the risk because they do not intend to use more than one admin role.

4.3 (HAL-03) ALLOWANCE CAN BE MODIFIED WHILE CONTRACT IS PAUSED - INFORMATIONAL (0.0)

Description:

The PauseModule contract introduces contract pausing functionality into the CMTAT ERC20 token. This prevents users from being able to perform transfers while the contract is paused.

However, this is not enforced in the functions that allow to change a user’s allowance.

BVSS:

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

Recommendation:

Add a check to prevent the users from being able to execute the functions that change allowance while the contract is paused.

Remediation:

ACKNOWLEDGED: The 21.co team acknowledged the issue because they want users to be able to change the allowance while the contract is paused in case of a hack.

4.4 (HAL-04) FLOATING PRAGMA - INFORMATIONAL (0.0)

Description:

Contracts should be deployed with the same compiler version and flags that they have been tested with thoroughly. Locking the pragma helps to ensure that contracts do not accidentally get deployed using, for example, an outdated compiler version that might introduce bugs that affect the contract system negatively.

Recommendation:

Set the pragma to a fixed version.

Remediation:

SOLVED: The 21.co team fixed the issue by locking the pragma version in pull request #12.

MANUAL TESTING

In the manual testing phase, the following scenarios were simulated. The scenarios listed below were selected based on the severity of the vulnerabilities Halborn was testing the program for.

5.1 ACCESS CONTROL AND ROLE MANAGEMENT

Description:

Proper access control on privileged functions was tested. All functions were reviewed to ensure that no sensitive functionality was left unprivileged.

Results:

- Only users with the MINTER_ROLE or theDEFAULT_ADMIN_ROLE were able to mint new tokens.
- Only users with the BURNER_ROLE or the DEFAULT_ADMIN_ROLE were able to mint new tokens.
- Only users with the PAUSER_ROLE or the DEFAULT_ADMIN_ROLE were able to pause token transfers.
- Only users with the DEFAULT_ADMIN_ROLE were able to change token id, terms, information, or flag of the token.

5.2 PROPER ERC20 FUNCTIONALITY

Description:

The CMTA token was reviewed to ensure it follows the ERC20 token standard.

Results:

- The 3 argument approve function does not follow the standard.

- All functions from the EIP20 standard were available in the contract.

5.3 REENTRANCY

Description:

All the functions were reviewed for external calls and possible reentrancy vulnerabilities.

Results:

No function was considered to be vulnerable to potential reentrancy attacks.

5.4 PAUSABLE FUNCTIONALITY

Functions were tested to ensure that no changes in the contract’s state could be performed while the contract is paused.

Results:

- Transactions could not be performed while contract is paused.
- Allowance could be changed while the contract is paused.

AUTOMATED TESTING

6.1 STATIC ANALYSIS REPORT

Description:

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 assessed all findings identified by the Slither software, however, findings with severity Information and Optimization are not included in the below results for the sake of report readability.

Results:

Slither results for CMTAT_BASE.sol

Finding	Impact
MathUpgradeable.mulDiv(uint256,uint256,uint256) (openzeppelin-contracts-upgradeable/contracts/utils/math/MathUpgradeable.sol#55-135) performs a multiplication on the result of a division: - denominator = denominator / twos (openzeppelin-contracts-upgradeable/contracts/utils/math/MathUpgradeable.sol#102) - inverse *= 2 - denominator * inverse (openzeppelin-contracts-upgradeable/contracts/utils/math/MathUpgradeable.sol#123)	Medium

Finding	Impact
MathUpgradeable.mulDiv(uint256,uint256,uint256) (openzeppelin-contracts-upgradeable/contracts/utils/math/MathUpgradeable.sol#55-135) performs a multiplication on the result of a division: - denominator = denominator / twos (openzeppelin-contracts-upgradeable/contracts/utils/math/MathUpgradeable.sol#102) - inverse *= 2 - denominator * inverse (openzeppelin-contracts-upgradeable/contracts/utils/math/MathUpgradeable.sol#121)	Medium
MathUpgradeable.mulDiv(uint256,uint256,uint256) (openzeppelin-contracts-upgradeable/contracts/utils/math/MathUpgradeable.sol#55-135) performs a multiplication on the result of a division: - denominator = denominator / twos (openzeppelin-contracts-upgradeable/contracts/utils/math/MathUpgradeable.sol#102) - inverse *= 2 - denominator * inverse (openzeppelin-contracts-upgradeable/contracts/utils/math/MathUpgradeable.sol#125)	Medium
MathUpgradeable.mulDiv(uint256,uint256,uint256) (openzeppelin-contracts-upgradeable/contracts/utils/math/MathUpgradeable.sol#55-135) performs a multiplication on the result of a division: - denominator = denominator / twos (openzeppelin-contracts-upgradeable/contracts/utils/math/MathUpgradeable.sol#102) - inverse *= 2 - denominator * inverse (openzeppelin-contracts-upgradeable/contracts/utils/math/MathUpgradeable.sol#124)	Medium

Finding	Impact
MathUpgradeable.mulDiv(uint256,uint256,uint256) (openzeppelin-contracts-upgradeable/contracts/utils/math/MathUpgradeable.sol#55-135) performs a multiplication on the result of a division: - denominator = denominator / twos (openzeppelin-contracts-upgradeable/contracts/utils/math/MathUpgradeable.sol#102) - inverse *= 2 - denominator * inverse (openzeppelin-contracts-upgradeable/contracts/utils/math/MathUpgradeable.sol#126)	Medium
MathUpgradeable.mulDiv(uint256,uint256,uint256) (openzeppelin-contracts-upgradeable/contracts/utils/math/MathUpgradeable.sol#55-135) performs a multiplication on the result of a division: - prod0 = prod0 / twos (openzeppelin-contracts-upgradeable/contracts/utils/math/MathUpgradeable.sol#105) - result = prod0 * inverse (openzeppelin-contracts-upgradeable/contracts/utils/math/MathUpgradeable.sol#132)	Medium
MathUpgradeable.mulDiv(uint256,uint256,uint256) (openzeppelin-contracts-upgradeable/contracts/utils/math/MathUpgradeable.sol#55-135) performs a multiplication on the result of a division: - denominator = denominator / twos (openzeppelin-contracts-upgradeable/contracts/utils/math/MathUpgradeable.sol#102) - inverse = (3 * denominator) extasciicircum 2 (openzeppelin-contracts-upgradeable/contracts/utils/math/MathUpgradeable.sol#117)	Medium

Finding	Impact
MathUpgradeable.mulDiv(uint256,uint256,uint256) (openzeppelin-contracts-upgradeable/contracts/utils/math/MathUpgradeable.sol#55-135) performs a multiplication on the result of a division: - denominator = denominator / twos (openzeppelin-contracts-upgradeable/contracts/utils/math/MathUpgradeable.sol#102) - inverse *= 2 - denominator * inverse (openzeppelin-contracts-upgradeable/contracts/utils/math/MathUpgradeable.sol#122)	Medium
End of table for CMTAT_BASE.sol	

Results summary:

The findings obtained as a result of the Slither scan were reviewed, and they were not included in the report because they were determined false positives.

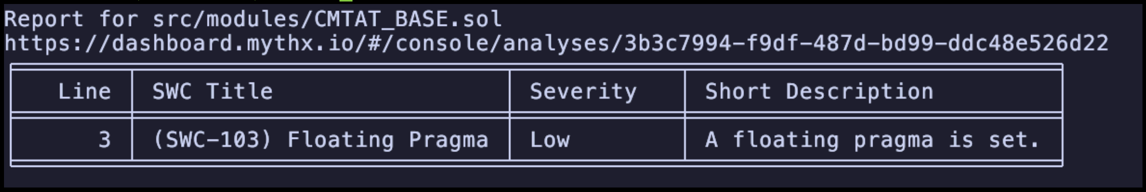
6.2 AUTOMATED SECURITY SCAN

Description:

Halborn used automated security scanners to assist with detection of well-known security issues and to identify low-hanging fruits on the targets for this engagement. Among the tools used was MythX, a security analysis service for Ethereum smart contracts. MythX performed a scan on the smart contracts and sent the compiled results to the analyzers in order to locate any vulnerabilities.

Results:

contracts/modules/CMTAT_BASE.sol



```
Report for src/modules/CMTAT_BASE.sol
https://dashboard.mythx.io/#/console/analyses/3b3c7994-f9df-487d-bd99-ddc48e526d22
╒════════╤═══════════════════════════╤════════════╤═══════════════════════════╕
│   Line │ SWC Title                 │ Severity   │ Short Description         │
╞════════╪═══════════════════════════╪════════════╪═══════════════════════════╡
│      3 │ (SWC-103) Floating Pragma │ Low        │ A floating pragma is set. │
╘════════╧═══════════════════════════╧════════════╧═══════════════════════════╛
```

The findings obtained as a result of the MythX scan were examined, and they were included in the report.

THANK YOU FOR CHOOSING

// HALBORN