// Private Smart Contract Security Assessment 08.27.2024 - 08.29.2024

HexTrustUSDv2

Hex Trust

HALBORN

HexTrustUSDv2 - Hex Trust

Prepared by: HALBORN

Last Updated 09/04/2024

Date of Engagement by: August 27th, 2024 - August 29th, 2024

Summary

100% ⓘ OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

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

TABLE OF CONTENTS

1. Introduction

Hex Trust engaged Halborn to conduct a security assessment of their changes for the HexTrustUSDv2 **beginning on August 27th and ending on August 29th**. The security assessment was scoped to the smart contracts changes in the GitHub PR. Commit hash and further details can be found in the Scope section of this report.

2. Assessment Summary

Halborn was provided 3 days for the engagement and assigned 1 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 security findings that were addressed and acknowledged by the Hex Trust 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.
- 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`).

3.1 Out-Of-Scope

- External libraries and financial-related attacks.
- New features/implementations after/with the **remediation commit IDs**. The review of the new cancellation mechanism and the deployment scripts was later conducted, uncovering two informational risk findings.

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: hex-stablecoin-usd

(b) Assessed Commit ID: aa3af65

(c) Items in scope:

- contracts/HexTrustUSDV2.sol
- contracts/AccessControlDefaultAdminRulesUpgradeable.sol
- contracts/BlacklistableWithRolesUpgradeable.sol
- contracts/ERC20WithRolesUpgradeable.sol
- layerzero/oapp/contracts/oft/OFTCoreUpgradeable.sol
- layerzero/oapp/contracts/oft/OFTAdapterUpgradeable.sol
- layerzero/oapp/contracts/oft/OFTUpgradeable.sol
- layerzero/oapp/contracts/oapp/OAppCoreUpgradeable.sol
- layerzero/oapp/contracts/oapp/OAppReceiverUpgradeable.sol
- layerzero/oapp/contracts/oapp/OAppSenderUpgradeable.sol
- layerzero/oapp/contracts/oapp/OAppUpgradeable.sol
- layerzero/oapp/contracts/oapp/libs/OAppOptionsType3Upgradeable.sol
- layerzero/oapp/contracts/precrime/OAppPreCrimeSimulatorUpgradeable.sol
- layerzero/oapp/contracts/precrime/PreCrimeUpgradeable.sol
- layerzero/oapp/contracts/precrime/extensions/PreCrimeE1Upgradeable.sol
- layerzero/oapp/contracts/auth/OAuth.sol
- ----------CHANGES----------
- For the new file HexTrustUSDV2.sol:
- - Extends OFTCoreUpgradeable.sol
- - Upgradability control update: enable for default admin & enable upgrade when contract is paused (by _authorizeUpgrade)
- - Add logic related to OFT
- - Modify based on https://github.com/LayerZero-Labs/LayerZero-v2/blob/main/packages/layerzero-v2/evm/oapp/contracts/oft/OFT.sol
- - Extends contracts under contracts/layerzero folder
- [Change] AccessControlDefaultAdminRulesUpgradeable.sol
- - add isDefaultAdmin() / onlyRoleOrDefaultAdmin() / onlyDefaultAdmin()
- [Change] BlacklistableWithRolesUpgradeable.sol
- - notBlacklistMember rename to notBlacklisterRole
- [Change] ERC20WithRolesUpgradeable.sol
- - burnBlackFunds can only be called by BURNER_ROLE instead of BLACKLISTER_ROLE
- ---
- For the layerzero/oapp folder:
- - Enhanced to upgradeable contracts
- - Remove all ownable logic

- - Enhanced to allow child contract to custom admin related logic by extending layerzero/auth/OAuth.sol
- - OApp add setEndpointDelegate() & endpointDelegate() for update & query from tagged endpoint. Not limited by initializing only

Out-of-Scope: Third party dependencies.

REMEDIATION COMMIT ID:

- 30d8c1d

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

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	0	0	0	4

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
FLOATING PRAGMA IN SOLIDITY CONTRACTS	INFORMATIONAL	ACKNOWLEDGED
UNUSED MODIFIER	INFORMATIONAL	ACKNOWLEDGED
USE OF REVERT STRINGS INSTEAD OF CUSTOM ERROR	INFORMATIONAL	SOLVED - 09/02/2024
OUTDATED NATSPEC	INFORMATIONAL	SOLVED - 09/02/2024

7. FINDINGS & TECH DETAILS

7.1 FLOATING PRAGMA IN SOLIDITY CONTRACTS

// INFORMATIONAL

Description

The files in scope started using a floating pragma version `^0.8.20` in the PR in scope, which means that the code can be compiled by any compiler version that is greater than or equal to 0.8.20, and less than 0.9.0.
It is recommended that contracts should be deployed with the same compiler version and flags used during development and testing. Locking the pragma helps to ensure that contracts do not accidentally get deployed using another pragma. For example, an outdated pragma version might introduce bugs that affect the contract system negatively.

Score

Impact:

Likelihood:

Recommendation

Lock the pragma version to the same version used during development and testing.

Remediation Progress

ACKNOWLEDGED: The **Hex Trust team** made a business decision to acknowledge this finding and not alter the contracts.

7.2 UNUSED MODIFIER

// INFORMATIONAL

Description

The new `onlyDefaultAdmin` modifier in the `AccessControlDefaultAdminRulesUpgradeable` contract was added in the PR under review, but it is never used in the codebase.

Score

Impact:

Likelihood:

Recommendation

For better clarity, consider removing the unused code.

Remediation Progress

ACKNOWLEDGED: The **Hex Trust team** made a business decision to acknowledge this finding and not alter the contracts.

7.3 USE OF REVERT STRINGS INSTEAD OF CUSTOM ERROR

// 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.
Even though most of the reverts were found to be using custom errors, some were still using simple strings. See the following example of the `onlyDefaultAdmin()` modifier added in the PR under review:

```
/**
 * @dev Throws if called by any account other than the _currentDefaultAdmin
 */
modifier onlyDefaultAdmin() {
  require(isDefaultAdmin(_msgSender()), "AccessControl: caller is not the d
  _;
}
```

Score

Impact:

Likelihood:

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 Official Solidity Documentation.

Remediation Progress

SOLVED: The **Hex Trust team** solved this finding in commit `30d8c1d8b9808abc492f95e2608e063a53142093` by following the mentioned recommendation.

Remediation Hash

https://github.com/hextrust/hex-stablecoin-usd/commit/30d8c1d8b9808abc492f95e2608e063a53142093

7.4 OUTDATED NATSPEC

// INFORMATIONAL

Description

The PR in scope modified some LayzerZero contracts to be upgradeable and added the Upgradeable suffix to the name of each contract. However,
See the following two examples:

```
/**
 * @title OAppCore
 * @dev Abstract contract implementing the IOAppCore interface with basic OA
 */
abstract contract OAppCoreUpgradeable is IOAppCore, OAuth, Initializable {
```

and

```
/**
 * @title OAuthUpgradeable
 * @dev Abstract contract implementing the IOAuth interface which perform ac
 */
abstract contract OAuth {
```

Score

Impact:

Likelihood:

Recommendation

It is recommended to review the NatSpec comments of all modified contracts to make sure they reflect the new behavior and names.

Remediation Progress

SOLVED: The **Hex Trust team** solved this finding in commit `30d8c1d8b9808abc492f95e2608e063a53142093` by following the mentioned recommendation.

Remediation Hash

https://github.com/hextrust/hex-stablecoin-usd/commit/30d8c1d8b9808abc492f95e2608e063a53142093

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

Output

```
INFO:Detectors:
OFTCoreUpgradeable._removeDust(uint256) (layerzero/oapp/contracts/oft/OFTCoreUpgradeable.sol#345-347) performs a multiplication on the result of a division:
        - (_amountLD / decimalConversionRate) * decimalConversionRate (layerzero/oapp/contracts/oft/OFTCoreUpgradeable.sol#346)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#divide-before-multiply
INFO:Detectors:
Contract locking ether found:
        Contract $AccessControlDefaultAdminRulesUpgradeable (contracts-exposed/AccessControlDefaultAdminRulesUpgradeable.sol#17-118) has payable functions:
         - $AccessControlDefaultAdminRulesUpgradeable.constructor() (contracts-exposed/AccessControlDefaultAdminRulesUpgradeable.sol#24-25)
         - $AccessControlDefaultAdminRulesUpgradeable.receive() (contracts-exposed/AccessControlDefaultAdminRulesUpgradeable.sol#117)
        But does not have a function to withdraw the ether
Contract locking ether found:
        Contract $BlacklistableWithRolesUpgradeable (contracts-exposed/BlacklistableWithRolesUpgradeable.sol#22-155) has payable functions:
         - $BlacklistableWithRolesUpgradeable.constructor() (contracts-exposed/BlacklistableWithRolesUpgradeable.sol#29-30)
         - $BlacklistableWithRolesUpgradeable.receive() (contracts-exposed/BlacklistableWithRolesUpgradeable.sol#154)
        But does not have a function to withdraw the ether
Contract locking ether found:
        Contract $ERC20WithRolesUpgradeable (contracts-exposed/ERC20WithRolesUpgradeable.sol#27-200) has payable functions:
         - $ERC20WithRolesUpgradeable.constructor() (contracts-exposed/ERC20WithRolesUpgradeable.sol#34-35)
         - $ERC20WithRolesUpgradeable.receive() (contracts-exposed/ERC20WithRolesUpgradeable.sol#199)
        But does not have a function to withdraw the ether
Contract locking ether found:
```

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.