// HALBORN

Substance Exchange - Exchange V4

Smart Contract Security Assessment

Prepared by: **Halborn**

Date of Engagement: **August 3rd, 2023 - August 29th, 2023**

Visit: **Halborn.com**

DOCUMENT REVISION HISTORY

VERSION	MODIFICATION	DATE
0.1	Document Creation	08/25/2023
0.2	Document Updates	08/26/2023
0.3	Draft Review	08/30/2023
1.0	Remediation Plan	09/19/2023
1.1	Remediation Plan Review	09/19/2023

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

Substance Exchange is a Perpetual Decentralized Exchange where users can interact with futures and options and also can be Liquidity Providers earning from traders.

Substance Exchange engaged Halborn to conduct a security assessment on their smart contracts beginning on August 3rd, 2023 and ending on August 29th, 2023. The security assessment was scoped to the smart contracts provided in the Substance Exchange V3 GitHub repository. Commit hashes and further details can be found in the Scope section of this report.

1.2 ASSESSMENT SUMMARY

Halborn was provided about 4 weeks for the engagement and assigned a 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 some improvements to reduce the likelihood and impact of risks, which were mostly addressed by Substance Exchange .

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) 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

2.4 SCOPE

1. IN-SCOPE TREE & COMMIT :

Code repositories:

1. Substance Exchange V4

- Repository: SubstanceExchangeV1
- Commit ID: 70064da2385541ebb59e85dfc79d911bd5d4b19b

Out-of-scope:

- third-party libraries and dependencies.
- economic attacks.

2. REMEDIATION COMMIT IDs :

- 51d826b6ae84364b6b6aa2927ec4679235bf76eb
- f5d8a07b3d34ed4efb80a4cb2be5e217ffbb9b70
- e3bcd45831222c853af17b7b6f1774e68cab28c1
- e627f170380c479ae32b635096f43219f07bb267
- d9201004c50f3618dd8637f34225dba97b38116d

3. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
2	0	4	4	1

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
(HAL-01) INCONSISTENCY IN USER BALANCE CALCULATIONS ALLOWS FOR EXPLOITS IN WITHDRAWALS	Critical (10)	SOLVED - 08/25/2023
(HAL-02) INCORRECT TOKEN TRANSFER IN VESTING FUNCTION OF STAKING SMART CONTRACT	Critical (9.2)	SOLVED - 08/25/2023
(HAL-03) ORDERS CAN GET STUCK IN CANCELLED STATE DUE TO MISSING DEADLINE CHECK	Medium (5.6)	SOLVED - 09/16/2023
(HAL-04) GAS STIPEND IN USERWITHDRAWETH FUNCTION AFFECTING GNOSIS SAFE INTERACTIONS	Medium (5.6)	SOLVED - 08/25/2023
(HAL-05) INCOMPATIBILITY WITH REBASING/DEFLATIONARY/INFLATIONARY TOKENS	Medium (6.2)	FUTURE RELEASE
(HAL-06) NON-STANDARD ERC20 TOKENS WILL REVERT	Medium (5.6)	SOLVED - 08/25/2023
(HAL-07) IMPLEMENTATIONS CAN BE INITIALIZED	Low (2.5)	SOLVED - 08/28/2023
(HAL-08) CONTRACTS CANNOT BE PAUSED	Low (3.3)	SOLVED - 08/25/2023
(HAL-09) SOLIDITY 0.8.20 CAN BREAK THE COMPATIBILITY ON THE MULTICHAIN	Low (2.3)	SOLVED - 08/22/2023
(HAL-10) BROKEN FUNCTIONALITY IN stake FUNCTION PREVENTS DIRECT INTERACTIONS	Low (2.1)	RISK ACCEPTED
(HAL-11) MISSING/INCOMPLETE NATSPEC COMMENTS	Informational (0.0)	ACKNOWLEDGED

FINDINGS & TECH DETAILS

4.1 (HAL-01) INCONSISTENCY IN USER BALANCE CALCULATIONS ALLOWS FOR EXPLOITS IN WITHDRAWALS - CRITICAL(10)

Description:

The smart contract for SubstanceExchangeV1 has an inconsistency in how userBalance and lockUserBalance are calculated and checked. Specifically, the userWithdraw function does not consider the user's locked balance

- **Order Creation:** Users can create an order using the makeOrder function.
- **Locking User Balance:** When an order is made, the lockUserBalance function is called, which checks the user's available balance through the getAvailableToken function.
- **User Withdrawal:** Users can withdraw their balance using the userWithdraw function.
- **Inconsistency:** The userWithdraw function does not consider the user's locked balance.

Code Location:

UserBalance.sol#L148

Listing 1

```
1     function lockUserBalance(
2         address _token,
3         address _user,
4         uint256 _amount
5     ) external isManager {
6         _validTokenAddress(_token);
7         if (getAvailableToken(_token, _user) < _amount) {
8             revert UserBalance__InsufficientAvailableTokenAmount()
```

```
↳ ;
 9          }
10          lockedBalance[_user][_token] += _amount;
11          _emitUserBalanceUpdate(_user, _token);
12      }
```

Proof Of Concept:

Step 1 : Firstly, the users can create an order with makeOrder function.
Step 2 : On the lockUserBalance call, the user's available balance is checked through getAvailableToken function.
Step 3 : After the creating an order, the user can call userWithdraw function.
Step 4 : However, the user is locked balance is not considered on the userWithdraw function. Even if the user does not have any balance, user can call userClaimOptionProfit and _cancelOrder.

BVSS:

AO:A/AC:L/AX:M/C:N/I:C/A:C/D:C/Y:N/R:N/S:U (10)

Recommendation:

Modify the userWithdraw function to consider the user's locked balance before allowing a withdrawal.

Remediation Plan:

SOLVED: The Substance Exchange team solved the issue by deleting the lockUserBalance variable.

Commit ID: 51d826b6ae84364b6b6aa2927ec4679235bf76eb

4.2 (HAL-02) INCORRECT TOKEN TRANSFER IN VESTING FUNCTION OF STAKING SMART CONTRACT - CRITICAL(9.2)

Description:

In the vest function, the rewardToken is being transferred instead of the stakingToken. This could lead to a critical issue where users receive the wrong token upon vesting, which is not what they initially staked. This discrepancy could lead to financial loss for the users and could severely undermine the trust in the staking platform.

Code Location:

StakingReward.sol#L151

Listing 2

```
    function vest(uint256 tokenId) external {
        // Anyone can HELP the fully locked NFT to begin vesting
        // if (ownerOf(tokenId) != msgSender()) revert
↳ StakingReward__NotOwner();
        _updateReward();
        StakedPosition storage position = positions[tokenId];
        if (position.unlockTime > block.timestamp) revert
↳ StakingReward__NotVesting();
        if (position.lastClaimTime > 0) revert
↳ StakingReward__AlreadyVested();

        position.lastClaimTime = block.timestamp;
        position.reward = earned(tokenId);
        totalShare -= position.share;
        rewardToken.transfer(address(exchangeWallet), position.
↳ amount);
        exchangeWallet.increaseBalance(address(rewardToken),
↳ ownerOf(tokenId), position.amount);

```

```
15          emit StartVesting(tokenId, position.reward);
16      }
```

Proof Of Concept:

- Users will receive rewardToken instead of their original stakingToken upon vesting.
- This could lead to financial imbalances within the smart contract, affecting the overall tokenomics.

BVSS:

AO:A/AC:L/AX:M/C:N/I:H/A:H/D:C/Y:N/R:N/S:U (9.2)

Recommendation:

Modify the vest function to transfer the stakingToken back to the user instead of the rewardToken.

Remediation Plan:

SOLVED: The Substance Exchange team solved the issue by sending the correct token.

Commit ID: e627f170380c479ae32b635096f43219f07bb267

4.3 (HAL-03) ORDERS CAN GET STUCK IN CANCELLED STATE DUE TO MISSING DEADLINE CHECK - MEDIUM (5.6)

Description:

The makeOrder function in the smart contract allows for the creation of orders with various parameters, including a _deadline. However, there is no check to ensure that the _deadline is greater than the current block timestamp. As a result, during the execution of the fillOrder function, orders can be directly marked as cancelled if they don't meet this condition, leading to orders getting stuck in a cancelled state.

Code Location:

/core/option/Option.sol#L244-L245

Listing 3

```
function makeOrder(
    address _user,
    uint256 _option,
    uint256 _epoch,
    uint256 _batch,
    uint256 _productId,
    uint256 _maxPrice,
    uint256 _size,
    uint256 _deadline
) external payable onlyManager returns (uint256 cost) {
    if (_option >= nextOptionId) {
        revert Option__InvalidOption();
    }
    if (msg.value < minExecutionFee) {
        revert Option__InsufficientExecutionFee();
    }
    if (epochBatch[_option][_epoch] < _batch) {
        revert Option__InvalidBatch();
    }
```

```
20      if (_productId >= optionProduct[_option][_epoch][_batch].
↳ length) {
21          revert Option__InvalidProductId();
22      }
23      if (_deadline > strikeTimeRecord[_option][_epoch][_batch]) {
24          revert Option__InvalidDeadline();
25      }
26      cost = _maxPrice * _size;
27      // last order is currentNonce - 1 ?
28      // so using a 0 index here
29      uint256 nonce = traderNonce[_user]++;
30      traderOrder[_user][nonce] = Struct.OptionOrder({
31          optionId: _option,
32          epochId: _epoch,
33          batch: _batch,
34          productId: _productId,
35          maxPrice: _maxPrice,
36          size: _size,
37          deadline: _deadline,
38          executionFee: msg.value,
39          valid: true
40      });
41      emit MakeOptionOrder(_user, nonce, _option, _epoch, _batch,
↳ _productId, _maxPrice, _size, _deadline, msg.value);
42  }
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:M/Y:L/R:N/S:U (5.6)

Recommendation:

Modify the makeOrder function to include a check that ensures the _deadline is greater than the current block timestamp.

Remediation Plan:

SOLVED: The Substance Exchange team solved the issue by adding a deadline check.

Commit ID: d9201004c50f3618dd8637f34225dba97b38116d

4.4 (HAL-04) GAS STIPEND IN USERWITHDRAWETH FUNCTION AFFECTING GNOSIS SAFE INTERACTIONS - MEDIUM (5.6)

Description:

The userWithdrawETH function in the smart contract uses the transfer method to send Ether, which has a gas stipend of 2300 by default. However, Gnosis Safe contracts require more than 2600 gas for successful execution, as documented here. This discrepancy in gas requirements makes it impossible for Gnosis Safe contracts to interact with the userWithdrawETH function.

Code Location:

UserBalance.sol#L128

Listing 4

```
1     function userWithdrawETH(uint256 _amount) external {
2         address user = msgSender();
3         userBalance[user][address(weth)] -= _amount;
4         weth.withdraw(_amount);
5         payable(user).transfer(_amount);
6         emit Withdraw(user, address(weth), _amount);
7         _emitUserBalanceUpdate(user, address(weth));
8     }
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:M/Y:L/R:N/S:U (5.6)

Recommendation:

Replace the transfer function with a low-level call to allow for a customizable gas stipend.

Remediation Plan:

SOLVED: The Substance Exchange team solved the issue by changing the function.

Commit ID: 51d826b6ae84364b6b6aa2927ec4679235bf76eb

4.5 (HAL-05) INCOMPATIBILITY WITH REBASING/DEFLATIONARY/INFLATIONARY TOKENS - MEDIUM (6.2)

Description:

The protocol does not appear to support rebasing/deflationary/inflationary tokens whose balance changes during transfers or over time. The necessary checks include at least verifying the amount of tokens transferred to contracts before and after the actual transfer to infer any fees/interest.

Code Location:

UserBalance.sol#L79C1-L86C6

Listing 5

```
    function userDeposit(address _token, uint256 _amount) external
↳ {
        _validTokenAddress(_token);
        address user = msgSender();
        IERC20(_token).safeTransferFrom(user, address(this),
↳ _amount);
        userBalance[user][_token] += _amount;
        emit Deposit(user, _token, _amount);
        _emitUserBalanceUpdate(user, _token);
    }
```

Proof Of Concept:

Step 1 : User A calls the userDeposit function to deposit 100 tokens into the contract.
Step 2 : Due to the deflationary nature of the token, a fee (e.g., 1%) is deducted during the transfer.
Step 3 : The contract, however, records a deposit of 100 tokens, although

only 99 tokens (100 tokens - 1% fee) were transferred due to the fee.

BVSS:

AO:A/AC:L/AX:L/C:N/I:M/A:N/D:M/Y:N/R:N/S:U (6.2)

Recommendation:

Ensure checking previous balance/after balance equals to the amount for any rebasing/inflation/deflation. Consider adding support in contracts for such tokens before accepting user-supplied tokens. Finally, consider supporting deflationary / rebasing / etc. tokens by extra checking the balances before/after or strictly inform your users not to use such tokens if they don't want to lose them.

Remediation Plan:

PENDING: The Substance Exchange team claims that they will not white-list any deflationary/inflationary token. The Team is committed to addressing this issue in a forthcoming release whenever they will white-list any deflationary/inflationary token.

4.6 (HAL-06) NON-STANDARD ERC20 TOKENS WILL REVERT - MEDIUM (5.6)

Description:

The library Option.sol contains the function to perform ERC20 tokens transfers in the protocol. However, this library uses the interface of IERC20 from OpenZeppelin which enforces the return value on transfer.

This pattern is not followed by all ERC20 tokens, for example USDT. If attempting to transfer these tokens, the contract will revert, preventing the transaction to be executed.

Code Location:

/core/option/Option.sol#L385

Listing 6

```
1    function transfer(
2         address _token,
3         address _to,
4         uint256 _amount
5     ) external onlyManager {
6         IERC20(_token).transfer(_to, _amount);
7     }
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:M/Y:L/R:N/S:U (5.6)

Recommendation:

It is recommended to use OpenZeppelin's SafeERC20 wrapper and the safeTransfer function to transfer the tokens.

Remediation Plan:

SOLVED: The Substance Exchange team solved the issue by using SafeERC20 wrapper.

Commit ID: f5d8a07b3d34ed4efb80a4cb2be5e217ffbb9b70

4.7 (HAL-07) IMPLEMENTATIONS CAN BE INITIALIZED - LOW (2.5)

Description:

The contracts are upgradable, inheriting from the Initializable contract. However, the current implementations are missing the _disableInitializers() function call in the constructors. Thus, an attacker can initialize the implementation. Usually, the initialized implementation has no direct impact on the proxy itself; however, it can be exploited in a phishing attack. In rare cases, the implementation might be mutable and may have an impact on the proxy.

BVSS:

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

Recommendation:

It is recommended to call _disableInitializers within the contract's constructor to prevent the implementation from being initialized.

Remediation Plan:

SOLVED: The contracts now implement the _disableInitializers() function call in the constructors.

Commit ID : e3bcd45831222c853af17b7b6f1774e68cab28c1

4.8 (HAL-08) CONTRACTS CANNOT BE PAUSED - LOW (3.3)

Description:

It was observed the contracts in scope lack the pause functionality. In case a vulnerability is discovered in any of the contracts or if any of the contracts is an object of an attack, protocol governance is unable of halting contract operations to manage the losses.

BVSS:

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

Recommendation:

Review the Pausable contract from OpenZeppelin and decorate mission-critical contract functions with the whenNotPaused modifier.

Remediation Plan:

SOLVED: The Substance Exchange team solved the issue by adding pause functionality.

Commit ID : 51d826b6ae84364b6b6aa2927ec4679235bf76eb

4.9 (HAL-09) SOLIDITY 0.8.20 CAN BREAK THE COMPATIBILITY ON THE MULTICHAIN - LOW (2.3)

Description:

Solidity version 0.8.20 introduced the support for PUSH 0, a feature not available in prior versions. While this might seem like a useful addition, it has potential compatibility issues across different types of blockchains. If a contract's pragmas are locked to Solidity version 0.8.20, certain blockchains may not support this new feature, thereby causing the contract functionality to break.

This issue might hinder the contract's ability to be deployed across multiple chains, restricting its interoperability and potentially leading to unexpected behavior in the smart contract on chains that don't support the PUSH 0 feature. It might also lead to failed transactions or unanticipated contract behavior, which could have severe financial implications and impact user trust.

Code Location:

Delegatable.sol#L3

Listing 7

```
1 pragma solidity ^0.8.19;
```

BVSS:

AO:A/AC:M/AX:M/C:H/I:L/A:N/D:N/Y:N/R:P/S:C (2.3)

Recommendation:

Consider the compatibility across different blockchain environments when deciding on a Solidity version. If interoperability is a priority, it is recommended to either use a lower Solidity version that has widespread support or implement feature detection checks to avoid using features that are unsupported on a given chain.

Remediation Plan:

SOLVED: The Substance Exchange team solved the issue by locking pragma in the hardhat config file.

Commit ID : hardhat.config.ts#L45

4.10 (HAL-10) BROKEN FUNCTIONALITY IN stake FUNCTION PREVENTS DIRECT INTERACTIONS - LOW (2.1)

Description:

The stake function in the smart contract is intended to allow users to stake a certain amount of tokens for a specified lockup time. However, the function currently does not permit direct interactions, leading to broken functionality.

Code Location:

StakingReward.sol#L126

Listing 8

```
    function stake(uint256 amount, uint256 lockupTime) external {
        if (amount <= 0) revert StakingReward__CannotStakeZero();
        _updateReward();
        address user = msgSender();
        uint256 share = (amount * boostMultiplier(lockupTime)) /
↳ BOOST_PRECISION;
        exchangeWallet.transfer(stakingToken, user, address(this),
↳  amount);
        totalShare += share;
        uint256 tokenId = nextTokenId++;
        StakedPosition storage position = positions[tokenId];
        position.amount = amount;
        position.unlockTime = block.timestamp + lockupTime;
        position.rewardPerSharePaid = rewardPerShare();
        position.share = share;

        emit CreatePosition(tokenId, amount, share, lockupTime);

        _mint(user, tokenId);
    }
```

BVSS:

AO:A/AC:L/AX:M/C:N/I:L/A:L/D:N/Y:N/R:N/S:U (2.1)

Recommendation:

Review the stake function to ensure that it allows for direct interactions as intended.

Remediation Plan:

RISK ACCEPTED: The Substance Exchange team accepted the risk of the issue. The team claims that it is design by default.

4.11 (HAL-11) MISSING/INCOMPLETE NATSPEC COMMENTS - INFORMATIONAL (0.0)

Description:

The functions are missing `@param` for some of their parameters. Given that **NatSpec** is an important part of code documentation, this affects code comprehension, auditability, and usability.

Code Location:

SubstanceExchangeV1

BVSS:

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

Recommendation:

Consider adding in full **NatSpec** comments for all functions to have complete code documentation for future use.

Remediation Plan:

ACKNOWLEDGED: The Substance Exchange team acknowledged the issue.

4.12 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:

MythX did not identify any vulnerabilities in the contracts.

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

THANK YOU FOR CHOOSING

// HALBORN