Haqq - Pad

Smart Contract Security Assessment

Prepared by: **Halborn**

Date of Engagement: **January 2nd, 2023 - January 30th, 2023**

Visit: **Halborn.com**

DOCUMENT REVISION HISTORY

VERSION	MODIFICATION	DATE
0.1	Document Creation	01/29/2024
0.2	Document Updates	01/29/2024
0.3	Draft Version	01/30/2024
0.4	Draft Review	02/01/2024
0.5	Draft Review	02/02/2024
1.0	Remediation Plan	02/13/2024
1.1	Remediation Plan Review	02/13/2024
1.2	Remediation Plan Review	02/14/2024

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
Luis Quispe Gonzales	Halborn	Luis.Quispegonzales@halborn.com
Alexis Fabre	Halborn	Alexis.Fabre@halborn.com

EXECUTIVE OVERVIEW

1.1 INTRODUCTION

Haqq engaged Halborn to conduct a security assessment on their smart contracts beginning on January 2nd, 2023 and ending on January 30th, 2023. The security assessment was scoped to the smart contracts provided to the Halborn team.

1.2 ASSESSMENT SUMMARY

The team at Halborn was provided four weeks 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 improvements to reduce the likelihood and impact of risks, which have been partially addressed by the Haqq team:

- Increase the sale token treasury upon reception of new funds.
- Enhance vault to either support or refuse sales with the same token.

1.3 SCOPE

1. Solidity Smart Contracts

   (a) Repository: haqqpad-contracts

   (b) Security Assessment Frozen Commit ID : c5b45ae

   (c) Contracts in scope:

   - core/IndexBucket.sol.
   - core/Round.sol.
   - core/Staking.sol.
   - core/Vault.sol.
   - tokenFactory/Factory.sol.

Out-of-scope: External libraries and financial related attacks.

Remediation Commit IDs :

- e2156f6.
- 3e9a4f2.
- 7785575.
- dd3f4e5.
- dd4562f.
- 80a26f3.

Out-of-scope: Newest features after remediation commits are out-of-scope.

1.4 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.
- 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

EXECUTIVE OVERVIEW

3. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	1	0	1	6

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
(HAL-01) TOKEN COUNT INACCURACY IN VAULT FUNDING	High (8.4)	SOLVED - 02/02/2024
(HAL-02) VAULT DOES NOT SUPPORT MULTIPLE ROUNDS WITH THE SAME TOKEN	Low (2.5)	SOLVED - 02/02/2024
(HAL-03) ABSENT TOKEN VERIFICATION IN VAULT FUNDING	Informational (0.0)	SOLVED - 02/07/2024
(HAL-04) VAULT TREASURY CAN BE INCREASED WITHOUT COLLATERAL	Informational (0.0)	SOLVED - 02/02/2024
(HAL-05) REFACTORING PROPOSAL TO GROUP TRANSFERS DURING STAKING WITHDRAW	Informational (0.0)	SOLVED - 02/02/2024
(HAL-06) UNSTAKING ID INCONSISTENCY	Informational (0.0)	ACKNOWLEDGED
(HAL-07) TAX ROUNDING ISSUE IN WITHDRAWAL FUNCTIONS	Informational (0.0)	ACKNOWLEDGED
(HAL-08) OPTIMIZATION IN GETBACKALL FUNCTION	Informational (0.0)	SOLVED - 02/07/2024

FINDINGS & TECH DETAILS

4.1 (HAL-01) TOKEN COUNT INACCURACY IN VAULT FUNDING - HIGH (8.4)

Description:

The lockInVault function in the **Vault** contract has an issue where it overwrites the existing token count with the amount from the most recent transaction instead of increasing it. If called multiple times, there will be a mismatch between the real balance and the internal addressVaults [token][projectAddress] variable, and it will only be possible to withdraw the latest amount transferred:

- Project owner deposits 1000 tokens.
- Project owner decides to deposit 500 additional tokens.
- For some reason, project owner decides to withdraw the totality of the tokens, but only 500 of them are withdrawable.

Code Location:

The issue is located in the Vault contract's lockInVault function, specifically where addressVaults[token][atAddress] is set to amount without accumulating the previous balance:

Listing 1: src/core/Vault.sol (Line 201)

```
function lockInVault(
    address token,
    address atAddress,
    uint256 amount
) external {
    if (atAddress == address(0)) {
        revert Errors.InvalidAddress();
    }
    if (token == address(0)) {
        revert Errors.InvalidToken();
    }

200     IERC20(token).safeTransferFrom \newline (msg.sender, address(this), amount);
201     addressVaults[token][atAddress] = amount;
202 
203     emit TokensLocked(token, atAddress, amount);
204 }
```

In the takeFromVault function, the amount eligible to withdraw is tracked by the addressVaults[token][atAddress] variable that could have been overwritten in the lockInVault function:

Listing 2: src/core/Vault.sol (Line 247)

```
237 function takeFromVault(
238     address token,
239     address atAddress,
240     uint256 amount
241 ) public accessOnly {
242     uint256 balance = addressVaults[token][atAddress];
243     if (balance < amount) {
244         revert Errors.InsufficientBalance();
245     }
246 
247     addressVaults[token][atAddress] -= amount;
248 
249     emit TakenFromVault(token, atAddress, amount);
250 }
```

BVSS:

AO:A/AC:L/AX:M/C:N/I:N/A:N/D:C/Y:N/R:N/S:C (8.4)

Recommendation:

It is recommended to increment the addressVaults[token][atAddress] value by amount instead of overwriting it. This change ensures that each call to the function correctly adds to the total amount of tokens locked in the vault by a user.

Remediation plan:

SOLVED: the issue was fixed in commit e2156f6, incrementing the amount instead of overwriting it.

4.2 (HAL-02) VAULT DOES NOT SUPPORT MULTIPLE ROUNDS WITH THE SAME TOKEN - LOW (2.5)

Description:

In the **Vault** contract of the NFT Marketplace, there is a limitation where it cannot distinguish user-locked funds for different rounds if the same sale token is used. The _init function sets the sale token for a round but does not verify if an ongoing round is already using the same sale token. This oversight leads to a potential clash in managing funds for multiple rounds that use the same token.

Code Location:

The _init function does not prevent the setting of a sale token that is already in use by another active round:

Listing 3: src/core/Vault.sol (Line 175)

```
170 function _init(
171     uint256 id,
172     address saleToken,
173     uint256 amount
174 ) private {
175     vaults[id].saleToken = saleToken;
176     vaults[id].saleTokenTreasury += amount;
177
178     emit SaleTokenSetted(id, saleToken, amount);
179 }
```

The lockInVault function locks in funds without associating them with a specific round ID, creating ambiguity when the same token is used in multiple rounds:

Listing 4: src/core/Vault.sol

```
function lockInVault(
    address token,
    address atAddress,
    uint256 amount
) external {
    if (atAddress == address(0)) {
        revert Errors.InvalidAddress();
    }
    if (token == address(0)) {
        revert Errors.InvalidToken();
    }

    IERC20(token).safeTransferFrom \newline (msg.sender, address(
 ↳ this), amount);
    addressVaults[token][atAddress] = amount;

    emit TokensLocked(token, atAddress, amount);
}
```

BVSS:

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

Recommendation:

It is recommended to ensure clear management of funds for different rounds.

Remediation Plan:

SOLVED: the issue was fixed in commit 3e9a4f2, ensuring that a sale token can only be used once for a round.

4.3 (HAL-03) ABSENT TOKEN VERIFICATION IN VAULT FUNDING - INFORMATIONAL (0.0)

Description:

The lockInVault function in the **Vault** contract is designed to lock sale tokens into the vault before starting the sale. However, it currently lacks a verification step to ensure that the tokens being deposited are indeed the saleToken specified for a given round. This oversight could lead to inconsistencies or human errors where a wrong token might be sent to the vault. Identifying and rectifying such errors post-deposit would require additional administrative intervention.

Code Location:

Listing 5: src/core/Vault.sol

```
function lockInVault(
    address token,
    address atAddress,
    uint256 amount
) external {
    if (atAddress == address(0)) {
        revert Errors.InvalidAddress();
    }
    if (token == address(0)) {
        revert Errors.InvalidToken();
    }

    IERC20(token).safeTransferFrom \newline (msg.sender, address(this), amount);
    addressVaults[token][atAddress] = amount;

    emit TokensLocked(token, atAddress, amount);
}
```

BVSS:

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

Recommendation:

It is recommended to verify that the `lockInVault` function parameter `token` matches the `saleToken` for a given round.

Remediation Plan:

SOLVED: the issue was fixed in commit 7785575, adding an administrator access control. The Haqq team also mentioned that:

> Human errors are unlikely in this case, since the transaction is generated programmatically and sent via ui.

4.4 (HAL-04) VAULT TREASURY CAN BE INCREASED WITHOUT COLLATERAL - INFORMATIONAL (0.0)

Description:

In the **Vault** contract of the NFT Marketplace, there is a potential issue in the increaseSaleTokenAmount function. This function permits any external caller to increase the saleTokenTreasury for a given round without actually transferring any collateral (tokens) into the vault. However, this action doesn't directly pose an exploitable threat, as no method allows for an attacker to withdraw these unbacked funds, it can lead to misleading information regarding the actual collateral held in the vault.

Code Location:

The saleTokenTreasury is increased without a corresponding transfer of tokens into the contract:

Listing 6: src/core/Vault.sol (Line 288)

```
279 function increaseSaleTokenAmount(
280     uint256 id,
281     uint256 amount
282 ) external availableRoundOnly(id) {
283     address saleToken = vaults[id].saleToken;
284     if (saleToken == address(0)) {
285         revert Errors.TokenNotSet();
286     }
287
288     vaults[id].saleTokenTreasury += amount;
289     emit SaleTokenDeposited(id, saleToken, amount);
290 }
```

BVSS:

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

Recommendation:

It is recommended to implement an access control on the increaseSaleTokenAmount or to make sure that every saleTokenTreasury is backed by an underlying token.

Remediation Plan:

SOLVED: the issue was fixed in commit dd3f4e5, adding an administrator access control.

4.5 (HAL-05) REFACTORING PROPOSAL TO GROUP TRANSFERS DURING STAKING WITHDRAW - INFORMATIONAL (0.0)

Description:

In the **Staking** contract, the withdrawAllUnbonded function currently processes multiple sendValue calls within a loop for each unbonded stake. This approach leads to multiple transactions, which can be inefficient and cost more gas.

Code Location:

The contract iterates over each unbonded stake and performs a sendValue call to transfer the unbonded amount to the user:

Listing 7: src/core/Staking.sol (Line 258)

```
246 function withdrawAllUnbonded() external {
247     uint256 i = pendingUnstakesForUser[msg.sender].length;
248     for (i; i > 0; ) {
249         // Never underflow.
250         unchecked { --i; }
251 
252         if (block.timestamp > pendingUnstakesForUser[msg.sender][i].unbondTimestamp) {
253             uint256 toWithdraw = _withdrawUnbonded(i);
254             uint256 tax = toWithdraw * UNSTAKING_TAX / PRECISION;
255 
256             totalFrozen -= toWithdraw;
257             taxTreasury += tax;
258             payable(msg.sender).sendValue(toWithdraw - tax);
259 
260             emit UnbondedWithdrawn(msg.sender, i, toWithdraw, tax);
261         }
262     }
263 }
```

BVSS:

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

Recommendation:

It is recommended to aggregate all transfers together into a unique call per user.

Remediation Plan:

SOLVED: the issue was fixed in commit dd4562f, aggregating all transfers together into a unique sendValue call.

4.6 (HAL-06) UNSTAKING ID INCONSISTENCY - INFORMATIONAL (0.0)

Description:

In the **Staking** contract, there is an issue with the handling of unstaking IDs during the process of unlocking funds. When a user initiates unstaking, an Unstaked event is emitted with an unstakingId. However, this ID can change during the process of unlocking funds. When a user withdraws an unstaked amount that has exceeded the unbonding period, the unstake item is removed from the list, and the last item in the list takes its place, changing the unstakingId of that item. This can lead to confusion and potential inconsistencies in tracking which unstakes have been withdrawn:

- user unstakes (t0) -> [unstaket0]
- user unstakes (t1) -> [unstaket0, unstaket1]
- user unstakes (t2) -> [unstaket0, unstaket1, unstaket2]
- user calls withdrawAllUnbonded after waiting enough time to withdraw unstake0:
- The unstakeId 0 is available to withdraw
- The list switches id2 with id0 and removes the last item
- [unstaket0, unstaket1, unstaket2] becomes [unstaket2, unstaket0]
- unstake2 has taken the place of unstaket0 and therefore is not referred anymore by unstakingId of 2, but instead has to be withdrawn using id of 0.
- user unstakes (t3) -> the list has 2 items so the next unstake id is 2
- Now the list is [unstaket2, unstaket0, unstaket3] but unstaket2 and unstaket3 both have emitted the Unstaked(id: 2) event.

Even though there could be a confusion with the emitted events, it will always be possible to finalize the unlock by simply considering unstakeId as the position in the list rather than a property of the unlocks.

Code Location:

The unstake function creates a new unstake and appends it to the pendingUnstakesForUser. The emitted unstake id is equal to the length of the unstake list, so it will not be unique if the list changes size once some unlocks are finalized:

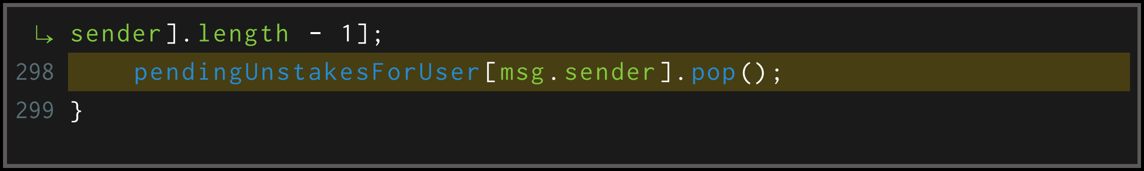
Listing 8: src/core/Staking.sol (Lines 234,240)

```
222 function unstake(
223     uint256 amountETH
224 ) external {
225     if (availableETHBalanceForUser[msg.sender] < amountETH) {
226         revert Errors.NotEnoughFreeBalance();
227     }
228
229     totalStakedByUser[msg.sender] -= amountETH;
230     availableETHBalanceForUser[msg.sender] -= amountETH;
231     totalStaked -= amountETH;
232     totalFrozen += amountETH;
233
234     uint256 unstakingId = pendingUnstakesForUser[msg.sender].
 ↳ length;
235     pendingUnstakesForUser[msg.sender].push(Unstake({
236         amountETH: amountETH,
237         unbondTimestamp: block.timestamp + UNBONDING_PERIOD
238     }));
239
240     emit Unstaked(msg.sender, amountETH, unstakingId, block.
 ↳ timestamp + UNBONDING_PERIOD);
241 }
```

The unstake list is popped here:

Listing 9: src/core/Staking.sol (Line 298)

```
293 function _withdrawUnbonded(
294     uint256 unstakingId
295 ) private returns (uint256 toSend) {
296     toSend = pendingUnstakesForUser[msg.sender][unstakingId].
 ↳ amountETH;
297     pendingUnstakesForUser[msg.sender][unstakingId] =
 ↳ pendingUnstakesForUser[msg.sender][pendingUnstakesForUser[msg.
```



```
↳ sender].length - 1];
        pendingUnstakesForUser[msg.sender].pop();
}
```

BVSS:

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

Recommendation:

It is recommended to better track the unlock once it changes place in the list, and consider that different unlocks can share the same unstake id.

Remediation Plan:

ACKNOWLEDGED: the Haqq team acknowledged that finding, also mentioning that:

> This is part of the protocol logic. The event specifies unstakingId to understand the number of objects in the array. When using the withdrawUnbonded function, unstakingId is passed correctly, according to the actual location of the elements in the array.

4.7 (HAL-07) TAX ROUNDING ISSUE IN WITHDRAWAL FUNCTIONS - INFORMATIONAL (0.0)

Description:

In the **Staking** contract, the withdrawUnbonded and withdrawAllUnbonded functions are designed to apply an unstaking tax on the withdrawal amount. This tax is calculated using a ratio of UNSTAKING_TAX / PRECISION, which translates to 2 / 100 or 1 / 50. Due to this ratio, any withdrawal amount below 50 results in no tax being applied. This is because the tax calculation, being an integer division, rounds down to zero for amounts less than 50.

This behavior is also found in the **Round** contract, in the _sendTokens function.

This tax evasion scheme would be only worth for the attacker if the cost of a transaction is lower than the amount of saved tokens, which is highly unlikely.

Code Location:

The tax is computed in the **Staking** contract, in the withdrawAllUnbonded function:

Listing 10: src/core/Staking.sol (Line 254)

```
246 function withdrawAllUnbonded() external {
247     uint256 i = pendingUnstakesForUser[msg.sender].length;
248     for (i; i > 0; ) {
249         // Never underflow.
250         unchecked { --i; }
251
252         if (block.timestamp > pendingUnstakesForUser[msg.sender][i].unbondTimestamp) {
253             uint256 toWithdraw = _withdrawUnbonded(i);
254             uint256 tax = toWithdraw * UNSTAKING_TAX / PRECISION;
```

```
            totalFrozen -= toWithdraw;
            taxTreasury += tax;
            payable(msg.sender).sendValue(toWithdraw - tax);

            emit UnbondedWithdrawn(msg.sender, i, toWithdraw, tax);
        }
    }
}
```

But also in the **Round** contract, in the _sendTokens function:

Listing 11: src/core/Round.sol (Lines 643,647)

```
629 function _sendTokens(
630     uint256 roundId,
631     bool factoryUsed,
632     bool fillIndexBucket
633 ) private {
634     /**
635      * @dev If the round ends with a `FAIL` status function will still work correctly
636      * and only return their own tokens to the project if the token address has been set.
637      */
638     uint256 income = boughtAllocation[false][roundId] + boughtAllocation[true][roundId];
639     uint256 soldTokens = income * _roundMainDetails[roundId].usdTokenPrice / PRICE_DECIMALS;
640     uint256 residue = _roundMainDetails[roundId].saleTokensAmount + _roundMainDetails[roundId].additionalSaleTokensAmount - soldTokens;
641     address projectAddress = _roundMainDetails[roundId].projectAddress;
642
643     uint256 tax = income * TAX_PERCENTAGE / TARGET;
644
645     /// If the round ends with a `FAIL` status `factoryUsed` will always be `false`.
646     if (factoryUsed) {
647         tax += income * FACTORY_TAX_PERCENTAGE / TARGET;
648
```

```
        DeployedToken memory tokenData = IFactory(factory).getTokenData(_roundMainDetails[roundId].roundToken);
        /// Transfer the owner's rights in the token contract to the project.
        ITransferOwnership(tokenData.token).transferOwnership(projectAddress);
        emit TokenOwnershipTransferred(roundId, tokenData.token, projectAddress);

        /// If the token contract is upgradeable and uses a proxy,
        /// then transfer the proxyAdmin contract owner rights to the project.
        if (tokenData.proxyAdmin != address(0)) {
            ITransferOwnership(tokenData.proxyAdmin).transferOwnership(projectAddress);
            emit TokenOwnershipTransferred(roundId, tokenData.proxyAdmin, projectAddress);
        }
    }
    uint256 toIndexBucket = fillIndexBucket ? income * INDEX_BUCKET_PERCENTAGE / TARGET : 0;
    income -= tax + toIndexBucket;
    IVault(vault).takeTax(roundId, tax);

    if (toIndexBucket > 0) {
        IVault(vault).withdrawUsdt(roundId, address(this), toIndexBucket);
        IERC20(usdt).safeApprove(indexBucket, toIndexBucket);
        IIndexBucket(indexBucket).deposit(usdt, roundId, toIndexBucket);
    }
    if (income > 0) {
        IVault(vault).withdrawUsdt(roundId, projectAddress, income);
        emit IncomeSended(roundId, projectAddress, income);
    }
    if (residue > 0 && _roundMainDetails[roundId].roundToken != address(0)) {
        IVault(vault).withdrawSaleToken(roundId, projectAddress, residue);
        emit SaleTokensReturned(roundId, projectAddress, residue);
    }
}
```

BVSS:

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

Recommendation:

This finding is mostly informative about the necessary rounding during integer divisions. It could be possible to apply a minimal taxed amount if the truncation is considered a problem.

Remediation Plan:

ACKNOWLEDGED: the Haqq team acknowledged that finding, also mentioning that:

> Solidity only works with integers, so if the numerator is less than the denominator, the result of division will be 0. In this case, this is the intended behavior. In addition, this behavior is absolutely disadvantageous for the user because withdrawing funds at 49 wei is disadvantageous in terms of gas consumption.

4.8 (HAL-08) OPTIMIZATION IN GETBACKALL FUNCTION - INFORMATIONAL (0.0)

Description:

In the **Round** contract, the _getBackAll function handles the return of unallocated USDT to users. Currently, this function calls IVault(vault).withdrawUsdt within a loop for each allocation amount, leading to multiple withdrawal transactions. A more efficient approach would be to aggregate the total USDT amount to be withdrawn and perform a single withdrawal transaction at the end of the loop.

Code Location:

Listing 12: src/core/Round.sol (Line 927)

```
function _getBackAll(
    uint256 roundId,
    address user,
    bool isLottery
) private {
    uint256[] memory positions = queueUserPositions[isLottery][roundId][user];
    uint256 passedIndex = queuePassedIndex[isLottery][roundId];
    uint256 totalAmount;
    uint256 i = positions.length;
    /// Looping in reverse order, because `pop()` is used and
    /// the `queueUserPositions` array can change length during the loop.
    for (i; i > 0;) {
        /// Never underflow.
        unchecked { --i; }
        if (positions[i] < passedIndex) {
            break;
        } else {
            uint256 amount = _roundsQueue[isLottery][roundId][positions[i]].allocationAmount;
            queueUserPositions[isLottery][roundId][user].pop();
```

```
                allocatedUSDTToUser[roundId][user] -= amount;
                IVault(vault).withdrawUsdt(roundId, user, amount); // would be more optimised to call once
                totalAmount += amount;

                if (!isLottery && amount > 0) {
                    earlyAccessRight[_roundPhaseDetails[roundId].timebuffRoundId][user].available = true;
                }
            }
        }
        if (totalAmount > 0) {
            emit BroughtBackAll(roundId, isLottery, user, totalAmount);
        }
}
```

BVSS:

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

Recommendation:

It is recommended to aggregate all withdrawals together into a unique call per user.

Remediation Plan:

SOLVED: the issue was fixed in commit 80a26f3, aggregating all transfers together into a unique withdrawUsdt call.

AUTOMATED TESTING

5.1 STATIC ANALYSIS REPORT

Description:

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.

Slither results:

Due to the large outputs, only high and critical issues are shown for the aux directory and only medium and above for any other contract.

Slither results for aux	
Finding	**Impact**
Round.endPledge(uint256,bool,uint256) (contracts/core/Round.sol#294-333) uses a Boolean constant improperly: - roundsQueue[false][roundId].push(nullQueue) (contracts/core/Round.sol#329) Round.endPledge(uint256,bool,uint256) (contracts/core/Round.sol#294-333) uses a Boolean constant improperly: - roundsQueue[true][roundId].push(nullQueue) (contracts/core/Round.sol#330) Round.buyAllocation(uint256,uint256) (contracts/core/Round.sol#340-429) uses a Boolean constant improperly: - boughtAllocation[false][roundId] + boughtAllocation[true][roundId] + amountUSDT > rounds[roundId].fundraiserGoal (contracts/core/Round.sol#390-391)	Medium

Finding	Impact
Round.endFundraising(EndFundraisingData,DeployData) (contracts/core/Round.sol#442-618) uses a Boolean constant improperly: - boughtAllocation[true][data.roundId] = 0 (contracts/core/Round.sol#490) Round.endFundraising(EndFundraisingData,DeployData) (contracts/core/Round.sol#442-618) uses a Boolean constant improperly: - boughtAllocation[false][data.roundId] = 0 (contracts/core/Round.sol#467) Round.endFundraising(EndFundraisingData,DeployData) (contracts/core/Round.sol#442-618) uses a Boolean constant improperly: - queuePassedIndex[true][data.roundId] = roundsQueue[true][data.roundId].length (contracts/core/Round.sol#555) Round.endFundraising(EndFundraisingData,DeployData) (contracts/core/Round.sol#442-618) uses a Boolean constant improperly: - totalBoughtAllocation = boughtAllocation[false][data.roundId] + boughtAllocation[true][data.roundId] (contracts/core/Round.sol#523-524) Round.endFundraising(EndFundraisingData,DeployData) (contracts/core/Round.sol#442-618) uses a Boolean constant improperly: - boughtAllocation[true][data.roundId] = 0 (contracts/core/Round.sol#468) Round.endFundraising(EndFundraisingData,DeployData) (contracts/core/Round.sol#442-618) uses a Boolean constant improperly: - queuePassedIndex[false][data.roundId] = roundsQueue[false][data.roundId].length (contracts/core/Round.sol#556) Round.endFundraising(EndFundraisingData,DeployData) (contracts/core/Round.sol#442-618) uses a Boolean constant improperly: - boughtAllocation[false][data.roundId] = 0 (contracts/core/Round.sol#489)	Medium

Finding	Impact
Round.endFundraising(EndFundraisingData,DeployData) (contracts/core/Round.sol#442-618) uses a Boolean constant improperly: - totalBoughtAllocation = boughtAllocation[true][data.roundId] + boughtAllocation[false][data.roundId] (contracts/core/Round.sol#460-461) Round._sendTokens(uint256,bool,bool) (contracts/core/Round.sol#626-675) uses a Boolean constant improperly: - income = boughtAllocation[false][roundId] + boughtAllocation[true][roundId] (contracts/core/Round.sol#635) Round._excludeUsersFromQueue (uint256,uint256,bool,uint256,uint256) (contracts/core/Round.sol#687-728) uses a Boolean constant improperly: - boughtAllocation[true][roundId] = 0 (contracts/core/Round.sol#701) Round._excludeUsersFromQueue (uint256,uint256,bool,uint256,uint256) (contracts/core/Round.sol#687-728) uses a Boolean constant improperly: - queuePassedIndex[false][roundId] = roundsQueue[false][roundId].length (contracts/core/Round.sol#704) Round._excludeUsersFromQueue (uint256,uint256,bool,uint256,uint256) (contracts/core/Round.sol#687-728) uses a Boolean constant improperly: - queuePassedIndex[false][roundId] = roundsQueue[false][roundId].length (contracts/core/Round.sol#719) Round._excludeUsersFromQueue (uint256,uint256,bool,uint256,uint256) (contracts/core/Round.sol#687-728) uses a Boolean constant improperly: - totalBoughtAllocation = tempBoughtAllocation + boughtAllocation[true][roundId] (contracts/core/Round.sol#695) Round._excludeUsersFromQueue (uint256,uint256,bool,uint256,uint256) (contracts/core/Round.sol#687-728) uses a Boolean constant improperly:	Medium

Finding	Impact
- queuePassedIndex[true][roundId] = 0 (contracts/core/Round.sol#700) Round._excludeUsersFromQueue (uint256,uint256,bool,uint256,uint256) (contracts/core/Round.sol#687-728) uses a Boolean constant improperly: - tempBoughtAllocation = boughtAllocation[false][roundId] (contracts/core/Round.sol#694) Round._calcToClaim(uint256,address,uint256) (contracts/core/Round.sol#842-860) performs a multiplication on the result of a division: - totalAmount = totalBoughtAllocation * rounds[roundId].usdTokenPrice / PRICE_DECIMALS (contracts/core/Round.sol#847) - totalVested = totalAmount * roundPhases[roundId].vestingTge / TARGET (contracts/core/Round.sol#848) Round._calcToClaim(uint256,address,uint256) (contracts/core/Round.sol#842-860) performs a multiplication on the result of a division: - monthsHavePassed = (block.timestamp - roundPhases[roundId].vestingStartedAt) / VESTING_INTERVAL (contracts/core/Round.sol#852) - totalVested += (totalAmount - totalVested) * monthsHavePassed / totalMonths (contracts/core/Round.sol#854) Reentrancy in Round._getBackAll(uint256,address,bool) (contracts/core/Round.sol#904-935): External calls: - IVault(vault).withdrawUsdt(roundId,user,amount) (contracts/core/Round.sol#924) State variables written after the call(s): - allocatedToUser[roundId][user] -= amount (contracts/core/Round.sol#923) Round.allocatedToUser (contracts/core/Round.sol#106) can be used in cross function reentrancies: - Round._exclude(uint256,uint256,uint256,bool) (contracts/core/Round.sol#744-799) - Round._getBackAll(uint256,address,bool) (contracts/core/Round.sol#904-935) - Round.allocatedToUser (contracts/core/Round.sol#106)	Medium

Finding	Impact
- Round.buyAllocation(uint256,uint256) (contracts/core/Round.sol#340-429) - Round.claim(uint256) (contracts/core/Round.sol#806-831) - Round.getBack(uint256,address,bool) (contracts/core/Round.sol#946-974) - queueUserPositions[isLottery][roundId][user].pop() (contracts/core/Round.sol#922) Round.queueUserPositions (contracts/core/Round.sol#104) can be used in cross function reentrancies: - Round._getBackAll(uint256,address,bool) (contracts/core/Round.sol#904-935) - Round.buyAllocation(uint256,uint256) (contracts/core/Round.sol#340-429) - Round.getBack(uint256,address,bool) (contracts/core/Round.sol#946-974) - Round.getQueueUserPositions(uint256,address,bool) (contracts/core/Round.sol#1161-1167) - Round.queueUserPositions (contracts/core/Round.sol#104) - Round.totalBoughtAllocationByUser(uint256,address,bool) (contracts/core/Round.sol#1069-1085) - timebuffs[roundPhases[roundId].timebuffRoundId][user].available = true (contracts/core/Round.sol#928) Round.timebuffs (contracts/core/Round.sol#99) can be used in cross function reentrancies: - Round._excludeUsersFromQueue (uint256,uint256,bool,uint256, uint256) (contracts/core/Round.sol#687-728) - Round._getBackAll(uint256,address,bool) (contracts/core/Round.sol#904-935) - Round.buyAllocation(uint256,uint256) (contracts/core/Round.sol#340-429) - Round.getBack(uint256,address,bool) (contracts/core/Round.sol#946-974) - Round.timebuffs (contracts/core/Round.sol#99)	Medium

Finding	Impact
Reentrancy in Round.buyAllocation(uint256,uint256) (contracts/core/Round.sol#340-429): External calls: - IERC20Upgradeable(usdt).safeTransferFrom(msg.sender,address(this),amountUSDT) (contracts/core/Round.sol#406) - IERC20Upgradeable(usdt).safeApprove(vault,amountUSDT) (contracts/core/Round.sol#407) - IVault(vault).depositUsdt(roundId,amountUSDT) (contracts/core/Round.sol#408) State variables written after the call(s): - allocatedToUser[roundId][msg.sender] += amountUSDT (contracts/core/Round.sol#409) Round.allocatedToUser (contracts/core/Round.sol#106) can be used in cross function reentrancies: - Round._exclude(uint256,uint256,uint256,bool) (contracts/core/Round.sol#744-799) - Round._getBackAll(uint256,address,bool) (contracts/core/Round.sol#904-935) - Round.allocatedToUser (contracts/core/Round.sol#106) - Round.buyAllocation(uint256,uint256) (contracts/core/Round.sol#340-429) - Round.claim(uint256) (contracts/core/Round.sol#806-831) - Round.getBack(uint256,address,bool) (contracts/core/Round.sol#946-974) - boughtAllocation[isLottery][roundId] += amountUSDT (contracts/core/Round.sol#419) Round.boughtAllocation (contracts/core/Round.sol#103) can be used in cross function reentrancies: - Round._exclude(uint256,uint256,uint256,bool) (contracts/core/Round.sol#744-799) - Round._excludeUsersFromQueue(uint256,uint256,bool,uint256,uint256) (contracts/core/Round.sol#687-728) - Round._sendTokens(uint256,bool,bool) (contracts/core/Round.sol#626-675)	Medium

Finding	Impact
- Round.boughtAllocation (contracts/core/Round.sol#103) - Round.buyAllocation(uint256,uint256) (contracts/core/Round.sol#340-429) - Round.endFundraising(EndFundraisingData,DeployData) (contracts/core/Round.sol#442-618) Reentrancy in Vault.depositSaleToken(uint256,uint256) (contracts/core/Vault.sol#257-270): External calls: - IERC20Upgradeable(saleToken).safeTransferFrom (msg.sender,address(this),amount) (contracts/core/Vault.sol#266) State variables written after the call(s): - vaults[id].saleTokenTreasury += amount (contracts/core/Vault.sol#267) Vault.vaults (contracts/core/Vault.sol#52) can be used in cross function reentrancies: - Vault._init(uint256,address,uint256) (contracts/core/Vault.sol#169-178) - Vault.availableRoundOnly(uint256) (contracts/core/Vault.sol#87-94) - Vault.depositSaleToken(uint256,uint256) (contracts/core/Vault.sol#257-270) - Vault.depositUsdt(uint256,uint256) (contracts/core/Vault.sol#322-330) - Vault.increaseSaleTokenAmount(uint256,uint256) (contracts/core/Vault.sol#278-289) - Vault.initRound(uint256,address,uint256) (contracts/core/Vault.sol#123-138) - Vault.lateInit(uint256,address,uint256) (contracts/core/Vault.sol#147-161) - Vault.takeTax(uint256,uint256) (contracts/core/Vault.sol#359-368) - Vault.vaults (contracts/core/Vault.sol#52) - Vault.withdrawSaleToken(uint256,address,uint256) (contracts/core/Vault.sol#300-314) - Vault.withdrawUsdt(uint256,address,uint256) (contracts/core/Vault.sol#341-350) Reentrancy in Vault.depositUsdt(uint256,uint256) (contracts/core/Vault.sol#322-330):	Medium

Finding	Impact
External calls: - IERC20Upgradeable(usdt).safeTransferFrom(msg.sender,address(this),amount) (contracts/core/Vault.sol#326) State variables written after the call(s): - vaults[id].usdtTreasury += amount (contracts/core/Vault.sol#327) Vault.vaults (contracts/core/Vault.sol#52) can be used in cross function reentrancies: - Vault._init(uint256,address,uint256) (contracts/core/Vault.sol#169-178) - Vault.availableRoundOnly(uint256) (contracts/core/Vault.sol#87-94) - Vault.depositSaleToken(uint256,uint256) (contracts/core/Vault.sol#257-270) - Vault.depositUsdt(uint256,uint256) (contracts/core/Vault.sol#322-330) - Vault.increaseSaleTokenAmount(uint256,uint256) (contracts/core/Vault.sol#278-289) - Vault.initRound(uint256,address,uint256) (contracts/core/Vault.sol#123-138) - Vault.lateInit(uint256,address,uint256) (contracts/core/Vault.sol#147-161) - Vault.takeTax(uint256,uint256) (contracts/core/Vault.sol#359-368) - Vault.vaults (contracts/core/Vault.sol#52) - Vault.withdrawSaleToken(uint256,address,uint256) (contracts/core/Vault.sol#300-314) - Vault.withdrawUsdt(uint256,address,uint256) (contracts/core/Vault.sol#341-350) Reentrancy in Round.endFundraising(EndFundraisingData,DeployData) (contracts/core/Round.sol#442-618): External calls: - _excludeUsersFromQueue(data.roundId,rounds[data.roundId].fundraiserGoal,false,0,data.startLoteryQueueIndex) (contracts/core/Round.sol#516-522) - IVault(vault).withdrawUsdt(roundId,user,extraAllocation) (contracts/core/Round.sol#796) - IVault(vault).takeFromVault(roundToken,rounds[data.roundId].projectAddress,data.additionalSaleTokensAmount) (contracts/core/Round.sol#531-535)	Medium

Finding	Impact
- IVault(vault).increaseSaleTokenAmount (data.roundId,data.additionalSaleTokensAmount) (contracts/core/Round.sol#536) State variables written after the call(s): - rounds[data.roundId].additionalSaleTokensAmount += data.additionalSaleTokensAmount (contracts/core/Round.sol#539) Round.rounds (contracts/core/Round.sol#96) can be used in cross function reentrancies: - Round._calcToClaim(uint256,address,uint256) (contracts/core/Round.sol#842-860) - Round._sendTokens(uint256,bool,bool) (contracts/core/Round.sol#626-675) - Round.buyAllocation(uint256,uint256) (contracts/core/Round.sol#340-429) - Round.checkEnded(uint256) (contracts/core/Round.sol#1108-1114) - Round.checkPledge(uint256) (contracts/core/Round.sol#1091-1101) - Round.claim(uint256) (contracts/core/Round.sol#806-831) - Round.endFundraising(EndFundraisingData,DeployData) (contracts/core/Round.sol#442-618) - Round.endPledge(uint256,bool,uint256) (contracts/core/Round.sol#294-333) - Round.getBack(uint256,address,bool) (contracts/core/Round.sol#946-974) - Round.getBackAll(uint256,address) (contracts/core/Round.sol#885-895) - Round.getRoundInfo(uint256) (contracts/core/Round.sol#1134-1141) - Round.increasePhaseDuration(uint256,uint256) (contracts/core/Round.sol#982-994) - Round.startRound(InitRoundData) (contracts/core/Round.sol#203-282) Reentrancy in Round.endFundraising(EndFundraisingData,DeployData) (contracts/core/Round.sol#442-618): External calls: - _excludeUsersFromQueue (data.roundId,rounds[data.roundId].fundraiserGoal,false,0, data.startLoteryQueueIndex) (contracts/core/Round.sol#516-522) - IVault(vault).withdrawUsdt(roundId,user,extraAllocation) (contracts/core/Round.sol#796)	Medium

Finding	Impact
- IVault(vault).takeFromVault(roundToken, rounds[data.roundId].projectAddress,data.additionalSaleTokensAmount) (contracts/core/Round.sol#531-535) - IVault(vault).increaseSaleTokenAmount(data.roundId,data.additionalSaleTokensAmount) (contracts/core/Round.sol#536) - _excludeUsersFromQueue (data.roundId,fundraiserGoal,true,data.startQueueIndex,0) (contracts/core/Round.sol#545-551) - IVault(vault).withdrawUsdt(roundId,user,extraAllocation) (contracts/core/Round.sol#796) State variables written after the call(s): - _excludeUsersFromQueue (data.roundId,fundraiserGoal,true,data.startQueueIndex,0) (contracts/core/Round.sol#545-551) - allocatedToUser[roundId][user] -= extraAllocation (contracts/core/Round.sol#793) Round.allocatedToUser (contracts/core/Round.sol#106) can be used in cross function reentrancies: - Round._exclude(uint256,uint256,uint256,bool) (contracts/core/Round.sol#744-799) - Round._getBackAll(uint256,address,bool) (contracts/core/Round.sol#904-935) - Round.allocatedToUser (contracts/core/Round.sol#106) - Round.buyAllocation(uint256,uint256) (contracts/core/Round.sol#340-429) - Round.claim(uint256) (contracts/core/Round.sol#806-831) - Round.getBack(uint256,address,bool) (contracts/core/Round.sol#946-974) - _excludeUsersFromQueue (data.roundId,fundraiserGoal,true,data.startQueueIndex,0) (contracts/core/Round.sol#545-551) - boughtAllocation[true][roundId] = 0 (contracts/core/Round.sol#701) - boughtAllocation[isLottery][roundId] = allowedAllocation (contracts/core/Round.sol#786) Round.boughtAllocation (contracts/core/Round.sol#103) can be used in cross function reentrancies: - Round._exclude(uint256,uint256,uint256,bool) (contracts/core/Round.sol#744-799)	Medium

AUTOMATED TESTING

Finding	Impact
- Round._excludeUsersFromQueue(uint256,uint256,bool,uint256,uint256) (contracts/core/Round.sol#687-728) - Round._sendTokens(uint256,bool,bool) (contracts/core/Round.sol#626-675) - Round.boughtAllocation (contracts/core/Round.sol#103) - Round.buyAllocation(uint256,uint256) (contracts/core/Round.sol#340-429) - Round.endFundraising(EndFundraisingData,DeployData) (contracts/core/Round.sol#442-618) - _excludeUsersFromQueue(data.roundId,fundraiserGoal,true,data.startQueueIndex,0) (contracts/core/Round.sol#545-551) - roundsQueue[isLottery][roundId][index].allocationAmount = userAllocationResidue (contracts/core/Round.sol#787) - roundsQueue[isLottery][roundId][index].totalUserAllocation -= extraAllocation (contracts/core/Round.sol#790) Round.roundsQueue (contracts/core/Round.sol#101) can be used in cross function reentrancies: - Round._exclude(uint256,uint256,uint256,bool) (contracts/core/Round.sol#744-799) - Round._excludeUsersFromQueue(uint256,uint256,bool,uint256,uint256) (contracts/core/Round.sol#687-728) - Round._getBackAll(uint256,address,bool) (contracts/core/Round.sol#904-935) - Round.buyAllocation(uint256,uint256) (contracts/core/Round.sol#340-429) - Round.endFundraising(EndFundraisingData,DeployData) (contracts/core/Round.sol#442-618) - Round.endPledge(uint256,bool,uint256) (contracts/core/Round.sol#294-333) - Round.getBack(uint256,address,bool) (contracts/core/Round.sol#946-974) - Round.getRoundQueue(uint256,bool) (contracts/core/Round.sol#1122-1127) - Round.totalBoughtAllocationByUser(uint256,address,bool) (contracts/core/Round.sol#1069-1085)	Medium

Finding	Impact
- _excludeUsersFromQueue (data.roundId,fundraiserGoal,true,data.startQueueIndex,0) (contracts/core/Round.sol#545-551) - timebuffs[roundPhases[roundId].timebuffRoundId][user].available = true (contracts/core/Round.sol#714) Round.timebuffs (contracts/core/Round.sol#99) can be used in cross function reentrancies: - Round._excludeUsersFromQueue (uint256,uint256,bool,uint256,uint256) (contracts/core/Round.sol#687-728) - Round._getBackAll(uint256,address,bool) (contracts/core/Round.sol#904-935) - Round.buyAllocation(uint256,uint256) (contracts/core/Round.sol#340-429) - Round.getBack(uint256,address,bool) (contracts/core/Round.sol#946-974) - Round.timebuffs (contracts/core/Round.sol#99) Reentrancy in Round.endFundraising(EndFundraisingData,DeployData) (contracts/core/Round.sol#442-618): External calls: - _excludeUsersFromQueue (data.roundId,rounds[data.roundId].fundraiserGoal,false,0, data.startLoteryQueueIndex) (contracts/core/Round.sol#516-522) - IVault(vault).withdrawUsdt(roundId,user,extraAllocation) (contracts/core/Round.sol#796) - IVault(vault).takeFromVault (roundToken,rounds[data.roundId].projectAddress, data.additionalSaleTokensAmount) (contracts/core/Round.sol#531-535) - IVault(vault).increaseSaleTokenAmount (data.roundId,data.additionalSaleTokensAmount) (contracts/core/Round.sol#536) - _excludeUsersFromQueue (data.roundId,fundraiserGoal,true,data.startQueueIndex,0) (contracts/core/Round.sol#545-551) - IVault(vault).withdrawUsdt(roundId,user,extraAllocation) (contracts/core/Round.sol#796) - roundToken = IFactory(factory).deployToken(address(this),toMint,deployData) (contracts/core/Round.sol#570)	Medium

Finding	Impact
State variables written after the call(s): - rounds[data.roundId].roundToken = roundToken (contracts/core/Round.sol#575) Round.rounds (contracts/core/Round.sol#96) can be used in cross function reentrancies: - Round._calcToClaim(uint256,address,uint256) (contracts/core/Round.sol#842-860) - Round._sendTokens(uint256,bool,bool) (contracts/core/Round.sol#626-675) - Round.buyAllocation(uint256,uint256) (contracts/core/Round.sol#340-429) - Round.checkEnded(uint256) (contracts/core/Round.sol#1108-1114) - Round.checkPledge(uint256) (contracts/core/Round.sol#1091-1101) - Round.claim(uint256) (contracts/core/Round.sol#806-831) - Round.endFundraising(EndFundraisingData,DeployData) (contracts/core/Round.sol#442-618) - Round.endPledge(uint256,bool,uint256) (contracts/core/Round.sol#294-333) - Round.getBack(uint256,address,bool) (contracts/core/Round.sol#946-974) - Round.getBackAll(uint256,address) (contracts/core/Round.sol#885-895) - Round.getRoundInfo(uint256) (contracts/core/Round.sol#1134-1141) - Round.increasePhaseDuration(uint256,uint256) (contracts/core/Round.sol#982-994) - Round.startRound(InitRoundData) (contracts/core/Round.sol#203-282) Reentrancy in Round.endFundraising(EndFundraisingData,DeployData) (contracts/core/Round.sol#442-618): External calls: - _excludeUsersFromQueue (data.roundId,rounds[data.roundId].fundraiserGoal,false,0, data.startLoteryQueueIndex) (contracts/core/Round.sol#516-522) - IVault(vault).withdrawUsdt(roundId,user,extraAllocation) (contracts/core/Round.sol#796) - IVault(vault).takeFromVault (roundToken,rounds[data.roundId].projectAddress, data.additionalSaleTokensAmount) (contracts/core/Round.sol#531-535)	Medium

Finding	Impact
- IVault(vault).increaseSaleTokenAmount(data.roundId,data.additionalSaleTokensAmount) (contracts/core/Round.sol#536) - _excludeUsersFromQueue(data.roundId,fundraiserGoal,true,data.startQueueIndex,0) (contracts/core/Round.sol#545-551) - IVault(vault).withdrawUsdt(roundId,user,extraAllocation) (contracts/core/Round.sol#796) - roundToken = IFactory(factory).deployToken(address(this),toMint,deployData) (contracts/core/Round.sol#570) - IERC20Upgradeable(roundToken).safeApprove(vault,toMint) (contracts/core/Round.sol#576-579) - IVault(vault).lateInit(data.roundId,roundToken,toMint) (contracts/core/Round.sol#580-584) State variables written after the call(s): - roundPhases[data.roundId].vestingDuration = data.vestingDuration (contracts/core/Round.sol#591) Round.roundPhases (contracts/core/Round.sol#97) can be used in cross function reentrancies: - Round._calcToClaim(uint256,address,uint256) (contracts/core/Round.sol#842-860) - Round._excludeUsersFromQueue(uint256,uint256,bool,uint256,uint256) (contracts/core/Round.sol#687-728) - Round._getBackAll(uint256,address,bool) (contracts/core/Round.sol#904-935) - Round.buyAllocation(uint256,uint256) (contracts/core/Round.sol#340-429) - Round.checkPledge(uint256) (contracts/core/Round.sol#1091-1101) - Round.endFundraising(EndFundraisingData,DeployData) (contracts/core/Round.sol#442-618) - Round.endPledge(uint256,bool,uint256) (contracts/core/Round.sol#294-333) - Round.getBack(uint256,address,bool) (contracts/core/Round.sol#946-974) - Round.getRoundInfo(uint256) (contracts/core/Round.sol#1134-1141) - Round.increasePhaseDuration(uint256,uint256) (contracts/core/Round.sol#982-994) - Round.startRound(InitRoundData) (contracts/core/Round.sol#203-282)	Medium

Finding	Impact
- roundPhases[data.roundId].vestingTge = data.vestingTge (contracts/core/Round.sol#592) Round.roundPhases (contracts/core/Round.sol#97) can be used in cross function reentrancies: - Round._calcToClaim(uint256,address,uint256) (contracts/core/Round.sol#842-860) - Round._excludeUsersFromQueue(uint256,uint256,bool,uint256,uint256) (contracts/core/Round.sol#687-728) - Round._getBackAll(uint256,address,bool) (contracts/core/Round.sol#904-935) - Round.buyAllocation(uint256,uint256) (contracts/core/Round.sol#340-429) - Round.checkPledge(uint256) (contracts/core/Round.sol#1091-1101) - Round.endFundraising(EndFundraisingData,DeployData) (contracts/core/Round.sol#442-618) - Round.endPledge(uint256,bool,uint256) (contracts/core/Round.sol#294-333) - Round.getBack(uint256,address,bool) (contracts/core/Round.sol#946-974) - Round.getRoundInfo(uint256) (contracts/core/Round.sol#1134-1141) - Round.increasePhaseDuration(uint256,uint256) (contracts/core/Round.sol#982-994) - Round.startRound(InitRoundData) (contracts/core/Round.sol#203-282) - roundPhases[data.roundId].vestingCliff = data.vestingCliff (contracts/core/Round.sol#593) Round.roundPhases (contracts/core/Round.sol#97) can be used in cross function reentrancies: - Round._calcToClaim(uint256,address,uint256) (contracts/core/Round.sol#842-860) - Round._excludeUsersFromQueue(uint256,uint256,bool,uint256,uint256) (contracts/core/Round.sol#687-728) - Round._getBackAll(uint256,address,bool) (contracts/core/Round.sol#904-935) - Round.buyAllocation(uint256,uint256) (contracts/core/Round.sol#340-429) - Round.checkPledge(uint256) (contracts/core/Round.sol#1091-1101)	Medium

Finding	Impact
- Round.endFundraising(EndFundraisingData,DeployData) (contracts/core/Round.sol#442-618) - Round.endPledge(uint256,bool,uint256) (contracts/core/Round.sol#294-333) - Round.getBack(uint256,address,bool) (contracts/core/Round.sol#946-974) - Round.getRoundInfo(uint256) (contracts/core/Round.sol#1134-1141) - Round.increasePhaseDuration(uint256,uint256) (contracts/core/Round.sol#982-994) - Round.startRound(InitRoundData) (contracts/core/Round.sol#203-282) - roundPhases[data.roundId].timebuffRoundId = nextRoundId (contracts/core/Round.sol#610) Round.roundPhases (contracts/core/Round.sol#97) can be used in cross function reentrancies: - Round._calcToClaim(uint256,address,uint256) (contracts/core/Round.sol#842-860) - Round._excludeUsersFromQueue (uint256,uint256,bool,uint256,uint256) (contracts/core/Round.sol#687-728) - Round._getBackAll(uint256,address,bool) (contracts/core/Round.sol#904-935) - Round.buyAllocation(uint256,uint256) (contracts/core/Round.sol#340-429) - Round.checkPledge(uint256) (contracts/core/Round.sol#1091-1101) - Round.endFundraising(EndFundraisingData,DeployData) (contracts/core/Round.sol#442-618) - Round.endPledge(uint256,bool,uint256) (contracts/core/Round.sol#294-333) - Round.getBack(uint256,address,bool) (contracts/core/Round.sol#946-974) - Round.getRoundInfo(uint256) (contracts/core/Round.sol#1134-1141) - Round.increasePhaseDuration(uint256,uint256) (contracts/core/Round.sol#982-994) - Round.startRound(InitRoundData) (contracts/core/Round.sol#203-282) - roundPhases[data.roundId].vestingStartedAt = block.timestamp + roundPhases[data.roundId].vestingCliff (contracts/core/Round.sol#611)	Medium

Finding	Impact
Round.roundPhases (contracts/core/Round.sol#97) can be used in cross function reentrancies: - Round._calcToClaim(uint256,address,uint256) (contracts/core/Round.sol#842-860) - Round._excludeUsersFromQueue (uint256,uint256,bool,uint256,uint256) (contracts/core/Round.sol#687-728) - Round._getBackAll(uint256,address,bool) (contracts/core/Round.sol#904-935) - Round.buyAllocation(uint256,uint256) (contracts/core/Round.sol#340-429) - Round.checkPledge(uint256) (contracts/core/Round.sol#1091-1101) - Round.endFundraising(EndFundraisingData,DeployData) (contracts/core/Round.sol#442-618) - Round.endPledge(uint256,bool,uint256) (contracts/core/Round.sol#294-333) - Round.getBack(uint256,address,bool) (contracts/core/Round.sol#946-974) - Round.getRoundInfo(uint256) (contracts/core/Round.sol#1134-1141) - Round.increasePhaseDuration(uint256,uint256) (contracts/core/Round.sol#982-994) - Round.startRound(InitRoundData) (contracts/core/Round.sol#203-282) - rounds[data.roundId].phase = Phase.VESTING (contracts/core/Round.sol#609) Round.rounds (contracts/core/Round.sol#96) can be used in cross function reentrancies: - Round._calcToClaim(uint256,address,uint256) (contracts/core/Round.sol#842-860) - Round._sendTokens(uint256,bool,bool) (contracts/core/Round.sol#626-675) - Round.buyAllocation(uint256,uint256) (contracts/core/Round.sol#340-429) - Round.checkEnded(uint256) (contracts/core/Round.sol#1108-1114) - Round.checkPledge(uint256) (contracts/core/Round.sol#1091-1101) - Round.claim(uint256) (contracts/core/Round.sol#806-831)	Medium

Finding	Impact
- Round.endFundraising(EndFundraisingData,DeployData) (contracts/core/Round.sol#442-618) - Round.endPledge(uint256,bool,uint256) (contracts/core/Round.sol#294-333) - Round.getBack(uint256,address,bool) (contracts/core/Round.sol#946-974) - Round.getBackAll(uint256,address) (contracts/core/Round.sol#885-895) - Round.getRoundInfo(uint256) (contracts/core/Round.sol#1134-1141) - Round.increasePhaseDuration(uint256,uint256) (contracts/core/Round.sol#982-994) - Round.startRound(InitRoundData) (contracts/core/Round.sol#203-282) Reentrancy in Round.getBackAll(uint256,address) (contracts/core/Round.sol#885-895): External calls: - _getBackAll(roundId,user,false) (contracts/core/Round.sol#893) - IVault(vault).withdrawUsdt(roundId,user,amount) (contracts/core/Round.sol#924) - _getBackAll(roundId,user,true) (contracts/core/Round.sol#894) - IVault(vault).withdrawUsdt(roundId,user,amount) (contracts/core/Round.sol#924) State variables written after the call(s): - _getBackAll(roundId,user,true) (contracts/core/Round.sol#894) - allocatedToUser[roundId][user] -= amount (contracts/core/Round.sol#923) Round.allocatedToUser (contracts/core/Round.sol#106) can be used in cross function reentrancies: - Round._exclude(uint256,uint256,uint256,bool) (contracts/core/Round.sol#744-799) - Round._getBackAll(uint256,address,bool) (contracts/core/Round.sol#904-935) - Round.allocatedToUser (contracts/core/Round.sol#106) - Round.buyAllocation(uint256,uint256) (contracts/core/Round.sol#340-429) - Round.claim(uint256) (contracts/core/Round.sol#806-831)	Medium

Finding	Impact
- Round.getBack(uint256,address,bool) (contracts/core/Round.sol#946-974) - _getBackAll(roundId,user,true) (contracts/core/Round.sol#894) - queueUserPositions[isLottery][roundId][user].pop() (contracts/core/Round.sol#922) Round.queueUserPositions (contracts/core/Round.sol#104) can be used in cross function reentrancies: - Round._getBackAll(uint256,address,bool) (contracts/core/Round.sol#904-935) - Round.buyAllocation(uint256,uint256) (contracts/core/Round.sol#340-429) - Round.getBack(uint256,address,bool) (contracts/core/Round.sol#946-974) - Round.getQueueUserPositions(uint256,address,bool) (contracts/core/Round.sol#1161-1167) - Round.queueUserPositions (contracts/core/Round.sol#104) - Round.totalBoughtAllocationByUser(uint256,address,bool) (contracts/core/Round.sol#1069-1085) - _getBackAll(roundId,user,true) (contracts/core/Round.sol#894) - timebuffs[roundPhases[roundId].timebuffRoundId][user].available = true (contracts/core/Round.sol#928) Round.timebuffs (contracts/core/Round.sol#99) can be used in cross function reentrancies: - Round._excludeUsersFromQueue(uint256,uint256,bool,uint256,uint256) (contracts/core/Round.sol#687-728) - Round._getBackAll(uint256,address,bool) (contracts/core/Round.sol#904-935) - Round.buyAllocation(uint256,uint256) (contracts/core/Round.sol#340-429) - Round.getBack(uint256,address,bool) (contracts/core/Round.sol#946-974) - Round.timebuffs (contracts/core/Round.sol#99) Reentrancy in Vault.lateInit(uint256,address,uint256) (contracts/core/Vault.sol#147-161): External calls: - IERC20Upgradeable(saleToken).safeTransferFrom(msg.sender,address(this),amount) (contracts/core/Vault.sol#159) State variables written after the call(s): - _init(id,saleToken,amount) (contracts/core/Vault.sol#160)	Medium

Finding	Impact
- vaults[id].saleToken = saleToken (contracts/core/Vault.sol#174) - vaults[id].saleTokenTreasury += amount (contracts/core/Vault.sol#175) Vault.vaults (contracts/core/Vault.sol#52) can be used in cross function reentrancies: - Vault._init(uint256,address,uint256) (contracts/core/Vault.sol#169-178) - Vault.availableRoundOnly(uint256) (contracts/core/Vault.sol#87-94) - Vault.depositSaleToken(uint256,uint256) (contracts/core/Vault.sol#257-270) - Vault.depositUsdt(uint256,uint256) (contracts/core/Vault.sol#322-330) - Vault.increaseSaleTokenAmount(uint256,uint256) (contracts/core/Vault.sol#278-289) - Vault.initRound(uint256,address,uint256) (contracts/core/Vault.sol#123-138) - Vault.lateInit(uint256,address,uint256) (contracts/core/Vault.sol#147-161) - Vault.takeTax(uint256,uint256) (contracts/core/Vault.sol#359-368) - Vault.vaults (contracts/core/Vault.sol#52) - Vault.withdrawSaleToken(uint256,address,uint256) (contracts/core/Vault.sol#300-314) - Vault.withdrawUsdt(uint256,address,uint256) (contracts/core/Vault.sol#341-350) Reentrancy in Staking.selectRounds(uint256[],uint8[],uint8[]) (contracts/core/Staking.sol#151-216): External calls: - IRound(round).addRoundToUser(msg.sender,selectedRounds[i]) (contracts/core/Staking.sol#210) State variables written after the call(s): - freeUserBalance[msg.sender] -= ethExtraPay (contracts/core/Staking.sol#190) Staking.freeUserBalance (contracts/core/Staking.sol#74) can be used in cross function reentrancies: - Staking.freeUserBalance (contracts/core/Staking.sol#74) - Staking.selectRounds(uint256[],uint8[],uint8[]) (contracts/core/Staking.sol#151-216) - Staking.stake() (contracts/core/Staking.sol#138-143)	Medium

Finding	Impact
- Staking.unlock(address,uint256) (contracts/core/Staking.sol#307-330) - Staking.unstake(uint256) (contracts/core/Staking.sol#222-241) - roundValue[selectedRounds[i]] += tiers[selectedTiers[i]].allocationUSD - tiers[_userRoundTier].allocationUSD (contracts/core/Staking.sol#192-193) Staking.roundValue (contracts/core/Staking.sol#67) can be used in cross function reentrancies: - Staking.roundValue (contracts/core/Staking.sol#67) - Staking.selectRounds(uint256[],uint8[],uint8[]) (contracts/core/Staking.sol#151-216) - userRound[msg.sender][selectedRounds[i]] = lockData (contracts/core/Staking.sol#203) Staking.userRound (contracts/core/Staking.sol#71) can be used in cross function reentrancies: - Staking.getUserRound(address,uint256) (contracts/core/Staking.sol#365-370) - Staking.selectRounds(uint256[],uint8[],uint8[]) (contracts/core/Staking.sol#151-216) - Staking.unlock(address,uint256) (contracts/core/Staking.sol#307-330) - Staking.userRound (contracts/core/Staking.sol#71) - userRoundIndex[msg.sender][selectedRounds[i]] = userRounds[msg.sender].length (contracts/core/Staking.sol#206) Staking.userRoundIndex (contracts/core/Staking.sol#70) can be used in cross function reentrancies: - Staking.selectRounds(uint256[],uint8[],uint8[]) (contracts/core/Staking.sol#151-216) - Staking.unlock(address,uint256) (contracts/core/Staking.sol#307-330) - Staking.userRoundIndex (contracts/core/Staking.sol#70) - userRounds[msg.sender].push(selectedRounds[i]) (contracts/core/Staking.sol#207) Staking.userRounds (contracts/core/Staking.sol#69) can be used in cross function reentrancies: - Staking.getUserRounds(address) (contracts/core/Staking.sol#377-381) - Staking.selectRounds(uint256[],uint8[],uint8[]) (contracts/core/Staking.sol#151-216)	Medium

Finding	Impact
- Staking.unlock(address,uint256) (contracts/core/Staking.sol#307-330) - Staking.userRounds (contracts/core/Staking.sol#69) Reentrancy in Staking.withdrawAllUnbonded() (contracts/core/Staking.sol#246-263): External calls: - address(msg.sender).sendValue(toWithdraw - tax) (contracts/core/Staking.sol#258) State variables written after the call(s): - taxTreasury += tax (contracts/core/Staking.sol#257) Staking.taxTreasury (contracts/core/Staking.sol#63) can be used in cross function reentrancies: - Staking.taxTreasury (contracts/core/Staking.sol#63) - Staking.withdrawAllUnbonded() (contracts/core/Staking.sol#246-263) - Staking.withdrawTax(address) (contracts/core/Staking.sol#337-349) - Staking.withdrawUnbonded(uint256) (contracts/core/Staking.sol#269-287) - totalFrozen -= toWithdraw (contracts/core/Staking.sol#256) Staking.totalFrozen (contracts/core/Staking.sol#65) can be used in cross function reentrancies: - Staking.totalFrozen (contracts/core/Staking.sol#65) - Staking.unstake(uint256) (contracts/core/Staking.sol#222-241) - Staking.withdrawAllUnbonded() (contracts/core/Staking.sol#246-263) - Staking.withdrawUnbonded(uint256) (contracts/core/Staking.sol#269-287) - toWithdraw = _withdrawUnbonded(i) (contracts/core/Staking.sol#253) - unstakes[msg.sender][unstakingId] = unstakes[msg.sender][unstakes[msg.sender].length - 1] (contracts/core/Staking.sol#297) - unstakes[msg.sender].pop() (contracts/core/Staking.sol#298) Staking.unstakes (contracts/core/Staking.sol#76) can be used in cross function reentrancies: - Staking._withdrawUnbonded(uint256) (contracts/core/Staking.sol#293-299) - Staking.getUserUnstakes(address) (contracts/core/Staking.sol#388-392)	Medium

Finding	Impact
- Staking.unstake(uint256) (contracts/core/Staking.sol#222-241) - Staking.unstakes (contracts/core/Staking.sol#76) - Staking.withdrawAllUnbonded() (contracts/core/Staking.sol#246-263) - Staking.withdrawUnbonded(uint256) (contracts/core/Staking.sol#269-287)	Medium
Round.endFundraising(EndFundraisingData,DeployData).factoryUsed (contracts/core/Round.sol#559) is a local variable never initialized Round._exclude(uint256,uint256,uint256,bool).userData (contracts/core/Round.sol#765) is a local variable never initialized Round.buyAllocation(uint256,uint256).timebuff (contracts/core/Round.sol#351) is a local variable never initialized Round.endPledge(uint256,bool,uint256).nullQueue (contracts/core/Round.sol#328) is a local variable never initialized Round.startRound(InitRoundData).toDeposit (contracts/core/Round.sol#266) is a local variable never initialized Round.buyAllocation(uint256,uint256).delay (contracts/core/Round.sol#350) is a local variable never initialized	Medium
End of table for aux	

- As a result of the tests carried out with the Slither tool, some results were obtained and reviewed by Halborn. Based on the results reviewed, some vulnerabilities were determined to be false positives.

THANK YOU FOR CHOOSING

// HALBORN