HALBORN

Floin - Floin Smart Contracts

Smart Contract Security Audit

Prepared by: **Halborn**

Date of Engagement: **February 21st, 2023 - March 2nd, 2023**

Visit: **Halborn.com**

DOCUMENT REVISION HISTORY

VERSION	MODIFICATION	DATE	AUTHOR
0.1	Document Creation	03/01/2023	Ataberk Yavuzer
0.2	Document Edits	03/01/2023	Ataberk Yavuzer
0.3	Draft Review	03/02/2023	Grzegorz Trawinski
0.4	Draft Review	03/02/2023	Piotr Cielas
0.5	Draft Review	03/02/2023	Gabi Urrutia
1.0	Remediation Plan	03/13/2023	Ataberk Yavuzer
1.1	Remediation Plan Edits	03/13/2023	Ataberk Yavuzer
1.2	Remediation Plan Review	03/15/2023	Gabi Urrutia

CONTACTS

CONTACT	COMPANY	EMAIL
Rob Behnke	Halborn	Rob.Behnke@halborn.com
Steven Walbroehl	Halborn	Steven.Walbroehl@halborn.com
Gabi Urrutia	Halborn	Gabi.Urrutia@halborn.com
Ataberk Yavuzer	Halborn	Ataberk.Yavuzer@halborn.com
Grzegorz Trawinski	Halborn	Grzegorz.Trawinski@halborn.com
Piotr Cielas	Halborn	Piotr.Cielas@halborn.com

EXECUTIVE OVERVIEW

1.1 INTRODUCTION

Floin is a decentralized platform built on top of the Ethereum blockchain that aims to provide a scalable and low-cost solution for DeFi transactions. Floin is designed to address the scalability and high transaction fees issues currently faced by the Ethereum network, by leveraging the power of layer-2 scaling solutions like Optimistic Rollups and Plasma.

Floin engaged Halborn to conduct a security audit on their smart contracts beginning on February 21st, 2023 and ending on March 2nd, 2023. The security assessment was scoped to the smart contracts provided to the Halborn team.

1.2 AUDIT SUMMARY

The team at Halborn was provided two weeks for the engagement and assigned a full-time security engineer to audit 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 audit is to:

- Ensure that smart contract functions operate as intended
- Identify potential security issues with the smart contracts

In summary, Halborn identified some security risks that should be addressed by the Floin team.

1.3 TEST APPROACH & METHODOLOGY

Halborn performed a combination of manual and automated security testing to balance efficiency, timeliness, practicality, and accuracy regarding

the scope of the smart contract audit. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of smart contracts and can quickly identify items that don't follow security best practices. The following phases and associated tools were used throughout the term of the audit:

- 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.
- Dynamic Analysis (foundry)
- Static Analysis(slither, MythX)

RISK METHODOLOGY:

Vulnerabilities or issues observed by Halborn are ranked based on the risk assessment methodology by measuring the **LIKELIHOOD** of a security incident and the **IMPACT** should an incident occur. This framework works for communicating the characteristics and impacts of technology vulnerabilities. The quantitative model ensures repeatable and accurate measurement while enabling users to see the underlying vulnerability characteristics that were used to generate the Risk scores. For every vulnerability, a risk level will be calculated on a scale of 5 to 1 with 5 being the highest likelihood or impact.

RISK SCALE - LIKELIHOOD

5 - Almost certain an incident will occur.
4 - High probability of an incident occurring.
3 - Potential of a security incident in the long term.
2 - Low probability of an incident occurring.
1 - Very unlikely issue will cause an incident.

RISK SCALE - IMPACT

5 - May cause devastating and unrecoverable impact or loss.
4 - May cause a significant level of impact or loss.
3 - May cause a partial impact or loss to many.
2 - May cause temporary impact or loss.
1 - May cause minimal or un-noticeable impact.

The risk level is then calculated using a sum of these two values, creating a value of 10 to 1 with 10 being the highest level of security risk.

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL

10 - CRITICAL
9 - 8 - HIGH
7 - 6 - MEDIUM
5 - 4 - LOW
3 - 1 - VERY LOW AND INFORMATIONAL

1.4 SCOPE

1. Floin - Floin Smart Contracts Security Audit Test Scope
    (a) Repository: Floin Smart Contracts
    (b) Commit ID: ba553936c09d86e579d1bcc97167f25d0a5214b1
2. In-Scope:
    (a) FLTKStaking.sol
    (b) Crowdsale.sol
    (c) VestingSplitter.sol
    (d) FLTK.sol
3. Out-of-Scope:
    (a) MockToken.sol

2. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
1	1	3	6	3

LIKELIHOOD

IMPACT

				(HAL-01)
			(HAL-02)	
(HAL-09) (HAL-11)	(HAL-06) (HAL-07)	(HAL-03)		
	(HAL-08) (HAL-10)		(HAL-04) (HAL-05)	
(HAL-12) (HAL-13) (HAL-14)				

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
(HAL-01) ANY USER CAN PURCHASE CROWDSALE TOKENS FOR FREE	Critical	SOLVED - 15/3/2023
(HAL-02) REWARD MODIFICATION LEADS TO REWARDS INFLATION	High	NOT APPLICABLE
(HAL-03) USING FIXED PRICE FOR EUR/USD PAIR CAN AFFECT CROWDSALE TOKEN PRICES	Medium	SOLVED - 15/3/2023
(HAL-04) MISCALCULATION OF DEPOSIT ID	Medium	NOT APPLICABLE
(HAL-05) DEPOSITFOR FUNCTION USES A DANGEROUS MODIFIER	Medium	SOLVED - 9/3/2023
(HAL-06) LATESTROUNDDATA MIGHT RETURN STALE RESULTS	Low	SOLVED - 9/3/2023
(HAL-07) INCOMPATIBILITY WITH TRANSFER-ON-FEE OR DEFLATIONARY TOKENS	Low	RISK ACCEPTED
(HAL-08) CENTRALIZED FUNCTIONALITY	Low	RISK ACCEPTED
(HAL-09) A PAUSER CAN BRICK CROWDSALE CONTRACT	Low	SOLVED - 9/3/2023
(HAL-10) LACK OF TWO-STEP OWNERSHIP PATTERN	Low	SOLVED - 9/3/2023
(HAL-11) SHARES VARIABLE DOES NOT HAVE AN UPPER BOUND	Low	NOT APPLICABLE
(HAL-12) MISSING REENTRANCY GUARD	Informational	SOLVED - 9/3/2023
(HAL-13) REVERT MESSAGES LONGER THAN 32 BYTES COST EXTRA GAS	Informational	SOLVED - 9/3/2023
(HAL-14) FOR LOOP OPTIMIZATIONS	Informational	SOLVED - 9/3/2023

FINDINGS & TECH DETAILS

3.1 (HAL-01) ANY USER CAN PURCHASE CROWDSALE TOKENS FOR FREE - CRITICAL

Description:

The Crowdsale contract enables swapping payment token for purchasable tokens. The amount of tokens to be paid is calculated with the following formula when calling the buyTokens() function:

Listing 1: Price Formula

```
uint256 payment = tokens * price * uint256(usdPrice) / 10 **
↳ priceFeedDecimals / 10 ** tokenDecimals;
```

The problem is, the payment can return zero if (tokens * price * uint256(usdPrice)) is less than (10 ** priceFeedDecimals / 10 ** tokenDecimals). Therefore, any user can purchase crowdsale tokens for free as they find an optimal value for tokens variable.

Proof of Concept:

Listing 2: Crowdsale.t.sol - Test Case

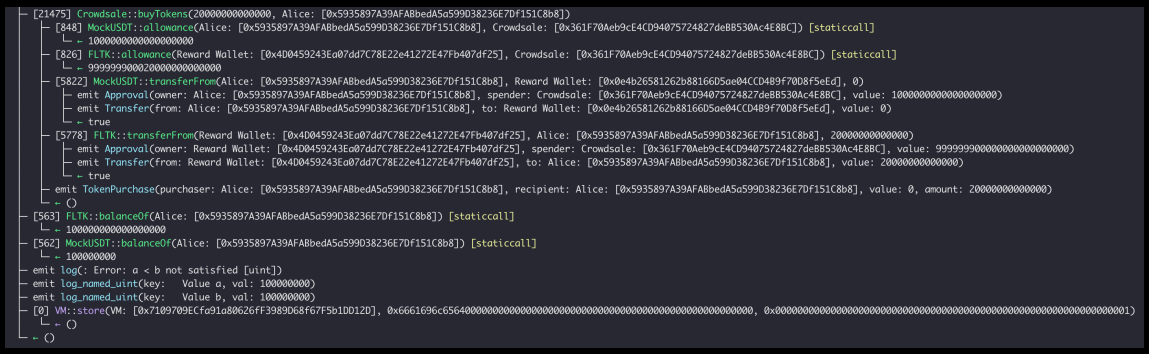
```
function testFail_crowdsaleBuyTokensForFreePoC() public {
    vm.startPrank(user1);
    uint256 allowedBalance = 1e18;
    mockUSDT.approve(address(crowdsale), allowedBalance);

    uint256 preFltkBalance = floinToken.balanceOf(user1);
    uint256 preUsdtBalance = mockUSDT.balanceOf(user1);

    for (uint i; i < 5000; ) {
        crowdsale.buyTokens(2e13, user1);
        unchecked {
            ++i;
        }
    }
```

```
15
16      uint256 postFltkBalance = floinToken.balanceOf(user1);
17      uint256 postUsdtBalance = mockUSDT.balanceOf(user1);
18
19      assertGt(postFltkBalance - preFltkBalance, 0);
20      assertLt(postUsdtBalance, preUsdtBalance);
21  }
```

Screenshots:



Code Location:

Listing 3: Crowdsale.sol (Lines 116,126)

```
109 function buyTokens(uint256 tokens, address recipient) public
 ↳ whenNotPaused onlyWhileOpen {
110     require(msg.sender == recipient || initiatorWhitelist[msg.
 ↳ sender], "Crowdsale::buyTokens: Not allowed.");
111     require(tokens > 0, "Crowdsale::buyTokens: tokens cannot be 0"
 ↳ );
112
113     int256 usdPrice = getLatestPrice();
114     require(usdPrice > 0, "Crowdsale::buyTokens: negative usd
 ↳ price");
115
116     uint256 payment = tokens * price * uint256(usdPrice) / 10 **
 ↳ priceFeedDecimals / 10 ** tokenDecimals;
117
118     uint256 allowance = paymentToken.allowance(msg.sender, address
 ↳ (this));
```

```
119     require(allowance >= payment, "Crowdsale::buyTokens: Allowance
 ↳  too low");
120
121     uint256 remaining = remainingTokens();
122     require(remaining >= tokens, "Crowdsale::buyTokens: Not enough
 ↳  tokens left");
123
124     paymentRaised += payment;
125
126     paymentToken.safeTransferFrom(msg.sender, paymentCollector,
 ↳ payment);
127     token.safeTransferFrom(tokenWallet, recipient, tokens);
128
129     emit TokenPurchase(msg.sender, recipient, payment, tokens);
130 }
```

Risk Level:

Likelihood - 5
Impact - 5

Recommendation:

Consider correcting the price formula by taking decimals of token and price feed into account.

Remediation Plan:

SOLVED: The Floin team solved the issue by introducing the new payment calculation function (getPayment()) into their code.

Commit ID: 822726a467a789a621736790b4cf44e2e62a48c9

3.2 (HAL-02) REWARD MODIFICATION LEADS TO REWARDS INFLATION - HIGH

Description:

The Staking contract implements a rewarding feature to help to distribute extra rewards to protocol users. The reward speed parameter is set inside the constructor when deploying the contract. The contract owner can modify rewarding parameters by calling the set() function.

However, the set() function tries to update pool rewards, since it also calls the updatePoolRewards() function. Therefore, rewards are miscalculated, and all stake owners get inflated rewards.

Proof of Concept:

Listing 4: FLTKStaking.t.sol - Test Case

```
function testFail_miscalculatedRewardsSetFunctionPoC() public {
    vm.startPrank(user1);
    uint256[] memory depositIds = new uint256[](2);
    stakingToken.increaseAllowance(address(fltkStaking), 3e18);
    depositIds[0] = fltkStaking.deposit(1);
    skip(1001);
    depositIds[1] = fltkStaking.deposit(2e18);
    //fltkStaking.claimRewards(depositIds); // 1001
    uint256 calculatedReward = fltkStaking.pendingRewardForUser(
↳ depositIds, user1);
    vm.stopPrank();

    vm.prank(deployer);
    fltkStaking.set(0, 1, 1e3);
    skip(1);
    vm.prank(user1);
    fltkStaking.claimRewards(depositIds); // 1001000

    uint256 finalRewardBalance = rewardToken.balanceOf(user1);
    assertEq(finalRewardBalance, calculatedReward);
}
```

Screenshots:

Code Location:

Listing 5: FLTKStaking.sol (Line 358)

```
351 function set(uint256 _rewardsPerSecond, uint256 _depositBucketSize
 ↳ , uint256 _rewardSlashingInterval)
352 public
353 onlyOwner
354 {
355     require(_depositBucketSize > 0, "FLTKStaking::set: Deposit
 ↳ bucket size has to be greater than 0");
356     require(_rewardSlashingInterval > 0, "FLTKStaking::set: Reward
 ↳  slashing interval has to greater than 0");
357
358     updatePoolRewards();
359     rewardsPerSecond = _rewardsPerSecond;
360     depositBucketSize = _depositBucketSize;
361     rewardSlashingInterval = _rewardSlashingInterval;
362 }
```

Risk Level:

Likelihood - 4
Impact - 4

Recommendation:

Consider correcting the updatePoolRewards() function for set() call since it inflates rewards.

Remediation Plan:

NOT APPLICABLE: This finding is not applicable, as the test scenario in the **Proof of Concept** section describes that the _rewardSlashingInterval variable is decreasing from 1e6 to 1e3. According to the reward formula, the **denumerator** will be less than the previous value. In this case, more rewards are expected as a result.

3.3 (HAL-03) FIXED PRICE FOR EUR/USD PAIR CAN AFFECT CROWDSALE TOKEN PRICES - MEDIUM

Description:

The payment calculation formula of Crowdsale contract uses fixed price for EUR/USD pair. The price variable represents the price of EUR/USD but this variable is immutable. It means the price will be defined during contract deployment. The problem is EUR/USD price can change over time.

If Euro loses value against United States Dollar, then, users will get less Crowdsale tokens than they should have.

Proof of Concept:

Before Value Loss:

Listing 6: EUR/USD prices before value loss

```
Chainlink Oracle EUR/USD price: 106690000
Chainlink Oracle USDC/USD price: 98991589
Floin Fixed EUR/USD price: 106690000
```

Alice decides to buy 200_000_000_000 Crowdsale FLTK tokens:

- (200_000_000_000 * 106690000 * 98991589)/ (10**8)/ (10**18)
- As a result, Alice will get 200_000_000_000 FLTK Crowdsale Tokens for 21.

After Value Loss:

Listing 7: EUR/USD prices after value loss

```
Chainlink Oracle EUR/USD price: 96690000
Chainlink Oracle USDC/USD price: 98991589
Floin Fixed EUR/USD price: 106690000
```

Alice buys 200_000_000_000 Crowdsale FLTK tokens:

- (200_000_000_000 * 106690000 * 98991589)/ (10**8)/ (10**18)
- As a result, Alice will get 200_000_000_000 FLTK Crowdsale Tokens for 21.
- However, actual price calculation should have been:
  - (200_000_000_000 * 96690000 * 98991589)/ (10**8)/ (10**18)
  - As a result, Alice should have get 200_000_000_000 FLTK Crowdsale Tokens for 19.

Code Location:

Listing 8: Crowdsale.sol (Line 102)

```
84 constructor(
85     address _paymentCollector,
86     address _tokenWallet,
87     address _priceFeed,
88     ERC20 _token,
89     ERC20 _paymentToken,
90     uint256 _price,
91     uint256 _openingTime,
92     uint256 _closingTime
93 ) {
94     if (_paymentCollector == address(0)) revert AddressZero();
95     if (_tokenWallet == address(0)) revert AddressZero();
96     if (address(_token) == address(0)) revert AddressZero();
97     if (address(_paymentToken) == address(0)) revert AddressZero()
   ↳ ;
98     if (_priceFeed == address(0)) revert AddressZero();
99     if (_price == 0) revert InvalidPrice();
100    if (_closingTime <= block.timestamp) revert InvalidClosingTime
```

```
 ↳ ();
101
102     price = _price;
```

Listing 9: Crowdsale.sol (Line 131)

```
131 uint256 payment = tokens * price * uint256(usdPrice) / 10 **
 ↳ priceFeedDecimals / 10 ** tokenDecimals;
```

Risk Level:

Likelihood - 3
Impact - 3

Recommendation:

In order not to be affected by this vulnerability, the instant value of this specific pair can be fetched over price oracles and this value can be used in the code.

Remediation Plan:

SOLVED: The Floin team solved the issue by introducing the new payment calculation function (getPayment()) into their code. The contract will get the EUR/USD price information from Chainlink Price Oracle with this new change.

Commit ID: 822726a467a789a621736790b4cf44e2e62a48c9

3.4 (HAL-04) MISCALCULATION OF DEPOSIT ID - MEDIUM

Description:

In the FLTKStaking.sol contract, when a user tries to stake their assets by calling the deposit function, that function calculates deposit ID with the following formula:

Listing 10: Deposit ID Calculation Formula

```
1 uint256 userDepositId = block.timestamp / depositBucketSize *
↳ depositBucketSize;
```

Division before multiplication is used in the formula above. Therefore, there is a precision loss and userDepositId might be miscalculated.

Proof of Concept:

1. Assume that depositBucketSize variable is 100.
2. Assume that block.timestamp is 1677687861.
3. The calculation will return 1677687800 as depositId. However, if we try to execute the multiplication before division, it was going to return 1677687861 which is the correct time.

Screenshots:

```
» uint256 currentTime = 1677687861
» uint256 depositBucketSize = 100
» uint256 userDepositId
» userDepositId = currentTime / depositBucketSize * depositBucketSize
» userDepositId
1677687800
```

```
» currentTime = 1677687861
» depositBucketSize = 100
» userDepositId = currentTime * depositBucketSize / depositBucketSize
» userDepositId
1677687861
```

Code Location:

Listing 11: FLTKStaking.sol (Line 152)

```
148 function deposit(uint256 _amount) public returns (uint256) {
149     require(_amount > 0, "FLTKStaking::deposit: Amount cannot be 0
 ↳ ");
150
151     address _userAddress = _getUser();
152     uint256 userDepositId = block.timestamp / depositBucketSize *
 ↳ depositBucketSize;
153     UserDeposit storage user = userDeposit[_userAddress][
 ↳ userDepositId];
154     uint256[] memory _depositIds = new uint256[](1);
155     _depositIds[0] = userDepositId;
```

Risk Level:

Likelihood - 4
Impact - 2

Recommendation:

Consider executing the multiplication before the division operation to prevent miscalculating deposit IDs.

Remediation Plan:

NOT APPLICABLE: At the end of the discussion with the Floin team, it was decided that this finding was not a vulnerability, but an intentional design. The loss of precision here is intentionally done to include depositors in the same group.

3.5 (HAL-05) DEPOSITFOR FUNCTION USES A DANGEROUS MODIFIER - MEDIUM

Description:

The depositFor function uses a risky impersonateUser modifier to deposit on behalf of another user. If someone gives allowance to the FLTKStaking contract, any other user can spend these allowances by calling the depositFor function by providing a depositor address.

An example scenario:

1. Alice decides to stake 10e18 Staking Tokens on FLTKStaking contract.
2. She calls stakingtoken.approve(address(FLTKStaking), 10e18) function to increase allowance.
3. She stakes 3e18 Staking Tokens and decides not to stake the other portion of the contract.
4. Bob calls depositFor(address(Alice), 7e18) function with the given parameters.
5. As a result, Alice's Staking Tokens will be deposited into FLTKStaking contract.

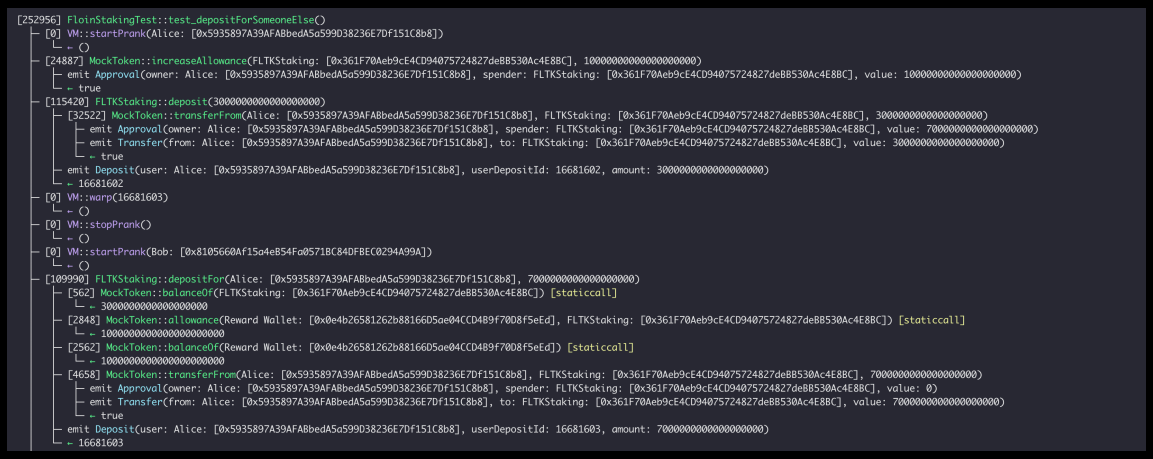
Proof of Concept:

Listing 12: FLTKStaking.t.sol - Test Case

```
function test_depositForSomeoneElse() public {
    vm.startPrank(user1);
    stakingToken.increaseAllowance(address(fltkStaking), 10e18);
    fltkStaking.deposit(3e18);
    skip(1);
    vm.stopPrank();

    vm.startPrank(user2);
    fltkStaking.depositFor(address(user1), 7e18);
    vm.stopPrank();
}
```

Screenshots:



Code Location:

Listing 13: FLTKStaking.sol (Line 175)

```
175 function depositFor(address _user, uint256 _amount) external
 ↳ impersonateUser(_user) returns (uint256) {
176     return deposit(_amount);
177 }
```

Listing 14: FLTKStaking.sol (Line 372)

```
371 modifier impersonateUser(address _user) {
372     _userContext = _user;
373     _;
374     _userContext = address(0);
375 }
```

Risk Level:

Likelihood - 4
Impact - 2

Recommendation:

It is suggested to remove the dangerous impersonateUser modifier and depositFor() function to prevent this attack to occur.

Remediation Plan:

SOLVED: The Floin team solved the issue by removing the depositFor function from their contract to solve this finding.

Commit ID: 5af6890d9dd9513c759541428ecbc2cd88ac6f56

3.6 (HAL-06) LATESTROUNDDATA MIGHT RETURN STALE RESULTS - LOW

Description:

Across these contracts, Chainlink's latestrounddata API is being used. However, there is only a check on updatedAt variable. This could lead to stale prices according to the Chainlink documentation:

- getrounddata-return-values

Code Location:

Listing 15: Crowdsale.sol (Line 179)

```
177 function getLatestPrice() public view returns (int) {
178     (
179     /* uint80 roundId */
180     ,
181     int256 answer,
182     /* uint256 startedAt */
183     ,
184     /* uint256 updatedAt */
185     ,
186     /* uint80 answeredInRound */
187     ) = priceFeed.latestRoundData();
188
189     return answer;
190 }
191
```

Risk Level:

Likelihood - 2
Impact - 3

Recommendation:

It is recommended to apply following checks too:

Listing 16: additional checks

```
require(answeredInRound >= roundID, "stale price");
require(updatedAt > 0, "incomplete round");
```

Remediation Plan:

SOLVED: The Floin team solved the issue by applying the above recommendation to solve this finding.

Commit ID: db6bd3edb7802165d8f934e2a0cac7bce5bb4bed

3.7 (HAL-07) INCOMPATIBILITY WITH TRANSFER-ON-FEE OR DEFLATIONARY TOKENS - LOW

Description:

Some ERC20 tokens make modifications to the standard implementations of their ERC20's `transfer` or `balanceOf` functions.
One type of such token is deflationary tokens that charge a fee on every `transfer()` and `transferFrom()`.
The protocol does not have incompatibility with fee-on-transfer tokens. When repaying debt to the treasury, the pre-fee amount is deducted, but the received amount might be lesser.

As a result, token transfers for these assets may fail silently. It is mostly a problem for `FLTKStaking` contract since the users are expected to get exact amount at stake during `withdraw()` call. The `deposit()` call will update user stakes with the exact amount as well.

Code Location:

Listing 17: FLTKStaking.sol (Lines 163,165)

```
148 function deposit(uint256 _amount) public returns (uint256) {
149     require(_amount > 0, "FLTKStaking::deposit: Amount cannot be 0
 ↳ ");
150
151     address _userAddress = _getUser();
152     uint256 userDepositId = block.timestamp / depositBucketSize *
 ↳ depositBucketSize;
153     UserDeposit storage user = userDeposit[_userAddress][
 ↳ userDepositId];
154     uint256[] memory _depositIds = new uint256[](1);
155     _depositIds[0] = userDepositId;
156
157     if (user.rewardDebt == 0) {
158         depositIds[_userAddress].push(userDepositId);
159     }
```

```
160
161     updatePoolRewards();
162     _saveRewards(_depositIds);
163     stakingToken.safeTransferFrom(_userAddress, address(this),
 ↳ _amount);
164
165     user.amount += _amount;
166     user.rewardDebt = user.amount * accTokenPerShare / UNITS;
167     emit Deposit(_userAddress, userDepositId, _amount);
168     return userDepositId;
169 }
```

Risk Level:

Likelihood - 2
Impact - 3

Recommendation:

If support for fee-on-transfer tokens is desired, consider tracking balances before and after the transfer to compute the actual received amount.

Remediation Plan:

RISK ACCEPTED: The Floin team accepted the risk of this finding as they **will not be using** any fee-on-transfer tokens in their protocol.

3.8 (HAL-08) CENTRALIZED FUNCTIONALITY - LOW

Description:

It was determined that the setInitiator function of Crowdsale function is a centralized function. With this function, the contract owner can decide who can participate in Crowdsale earlier than other users in the protocol.

It is a good practice to decentralized contract governance by avoiding excessive use of centralized functions.

Code Location:

Listing 18: Crowdsale.sol (Line 155)

```
153 function setInitiator(address address_, bool allowed) public
 ↳ onlyOwner {
154     require(initiatorWhitelist[address_] != allowed, "Crowdsale::
 ↳ setInitiator: Already set");
155     initiatorWhitelist[address_] = allowed;
156 }
```

Risk Level:

Likelihood - 2
Impact - 2

Recommendation:

Consider removing the `setInitiator` function to avoid centralization problems.

Remediation Plan:

RISK ACCEPTED: The Floin team accepted the risk as the finding will only pose a risk if the Owner address is compromised.

3.9 (HAL-09) A PAUSER CAN BRICK CROWDSALE CONTRACT - LOW

Description:

A malicious or compromised Pauser can call pause() and renouncePauser() to brick the contract and all assets can be frozen in the Crowdsale contract.

Proof of Concept:

Listing 19: Crowdsale.t.sol - Test Case

```
1  function test_pauserCanBrickContractPoC() public {
2      vm.startPrank(deployer);
3      crowdsale.pause();
4      crowdsale.renounceOwnership();
5      vm.stopPrank();
6
7      vm.startPrank(user1);
8      mockUSDT.approve(address(crowdsale), 100e6);
9      crowdsale.buyTokens(1e18, address(0));
10     vm.stopPrank();
11 }
```

Screenshots:

```
[53419] FloinCrowdSaleTest::test_pauserCanBrickContractPoC()
  ├─ [0] VM::startPrank(Deployer: [0x7121207b118BbaCF0340A989527474Bd4495c3C6])
  │   └─ ← ()
  ├─ [6716] Crowdsale::pause()
  │   ├─ emit Paused(account: Deployer: [0x7121207b118BbaCF0340A989527474Bd4495c3C6])
  │   └─ ← ()
  ├─ [2205] Crowdsale::renounceOwnership()
  │   ├─ emit OwnershipTransferred(previousOwner: Deployer: [0x7121207b118BbaCF0340A989527474Bd4495c3C6], newOwner: 0x0000000000000000000000000000000000000000)
  │   └─ ← ()
  ├─ [0] VM::stopPrank()
  │   └─ ← ()
  ├─ [0] VM::startPrank(Alice: [0x5935897A39AFABbedA5a599D38236E7Df151C8b8])
  │   └─ ← ()
  ├─ [24628] MockUSDT::approve(Crowdsale: [0x361F70Aeb9cE4CD94075724827deBB530Ac4E8BC], 100000000)
  │   ├─ emit Approval(owner: Alice: [0x5935897A39AFABbedA5a599D38236E7Df151C8b8], spender: Crowdsale: [0x361F70Aeb9cE4CD94075724827deBB530Ac4E8BC], value: 100000000)
  │   └─ ← true
  ├─ [643] Crowdsale::buyTokens(1000000000000000000, 0x0000000000000000000000000000000000000000)
  │   └─ ← "Pausable: paused"
  └─ ← "Pausable: paused"
```

Code Location:

Listing 20: Crowdsale.sol (Line 13)

```
13 contract Crowdsale is Ownable, Pausable {
```

Risk Level:

Likelihood - 1
Impact - 3

Recommendation:

Consider overriding renouncePauser function to prevent renouncing address to address(0). Furthermore, transferOwnership() function is not recommended to use. A two-Step Ownership Change pattern should be followed.

Check: (HAL-10) LACK OF TWO-STEP OWNERSHIP PATTERN finding for more information.

Remediation Plan:

SOLVED: The Floin team solved the issue by implementing the Ownable2Step library as Access Control and overriding the renounceOwnership() function.

Commit ID: a93f8e0bfefb338f5efe62d2540794ba5878eeac

3.10 (HAL-10) LACK OF TWO-STEP OWNERSHIP PATTERN - LOW

Description:

To change the owner address, the current contract owner can call the Ownable.transferOwnership() function and set a new address and this new address assumes the role immediately.
If the new address is inactive or not willing to act in the role, there is no way to restore access to that role. Therefore, the owner role can be lost.

Code Location:

- Crowdsale.sol#L13
- VestingSplitter.sol#L12
- FLTKStaking.sol#L9

Risk Level:

Likelihood - 2
Impact - 2

Recommendation:

It is recommended to use the following Ownable2Step library instead of Ownable library:

Ownable2Step.sol

Remediation Plan:

SOLVED: The Floin team solved the issue by implementing the Ownable2Step library.

Commit ID: 32e2830c4d7eab3c62f1e439458601b68f2e5c2b

3.11 (HAL-11) SHARES VARIABLE DOES NOT HAVE AN UPPER BOUND - LOW

Description:

The shares_ variable of _addPayee() function does not have any upper bound. If these share values are used as percentages, an upper limit should be used. Not using the upper limit in such functions may cause overflow issues.

Code Location:

- VestingSplitter.sol#L143

Risk Level:

Likelihood - 1
Impact - 3

Recommendation:

Consider implementing an upper bound for shares_ variable to prevent any overflow issue.

Remediation Plan:

NOT APPLICABLE: The finding is not applicable since the shares_ variable does not actually represent the percentage. Therefore, the overflow is not realistic for this case.

3.12 (HAL-12) MISSING REENTRANCY GUARD - INFORMATIONAL

Description:

To protect against cross-function re-entrancy attacks, it may be necessary to use a mutex. Functions decorated with such modifiers cannot be exploited with recursive calls. OpenZeppelin has its own mutex implementation called ReentrancyGuard which provides a nonReentrant modifier that guards the decorated function with a mutex against re-entrancy attacks.

Some tokens (ERC777) are known to have callback functions. It is recommended to add **Reentrancy Guard** as a best-practice to protect against Reentrancy vulnerabilities that may occur from these tokens.

Code Location:

Listing 21: Functions with missing Reentrancy Guard

```
Crowdsale.buyTokens()
FLTKStaking.deposit()
FLTKStaking.withdraw()
FLTKStaking.emergencyWithdraw()
FLTKStaking.claimRewards()
VestingSplitter.release()
```

Risk Level:

Likelihood - 1
Impact - 1

Recommendation:

The functions listed in the Code Location section are missing nonReentrant modifiers. It is recommended to add the necessary nonReentrant modifier to prevent the introduction of future re-entrancy vulnerabilities.

- OpenZeppelin - Reentrancy Guard

Remediation Plan:

SOLVED: The Floin team solved the issue by implementing OpenZeppelin ReentrancyGuard to the above functions.

Commit ID: fce3f5c31f4c15e7bfc8f9b1204948b48aa7bd13

3.13 (HAL-13) REVERT MESSAGES LONGER THAN 32 BYTES COST EXTRA GAS - INFORMATIONAL

Description:

If message data can fit into 32 bytes, it is always a better idea to use BYTES32 datatype rather than bytes or strings as using that variable is much cheaper in Solidity.

Some revert messages are longer than 32 bytes. In this case, they will cost extra gas.

Code Location:

- VestingSplitter.sol#L35
- VestingSplitter.sol#L37
- VestingSplitter.sol#L38
- VestingSplitter.sol#L101
- VestingSplitter.sol#L111
- VestingSplitter.sol#L144
- VestingSplitter.sol#L145
- FLTKStaking.sol#L67
- FLTKStaking.sol#L68
- FLTKStaking.sol#L69
- FLTKStaking.sol#L70
- FLTKStaking.sol#L71
- FLTKStaking.sol#L149
- FLTKStaking.sol#L183
- FLTKStaking.sol#L207
- FLTKStaking.sol#L324
- FLTKStaking.sol#L326
- FLTKStaking.sol#L333
- FLTKStaking.sol#L336
- FLTKStaking.sol#L355

- FLTKStaking.sol#L356
- FLTKStaking.sol#L365
- Crowdsale.sol#L110
- Crowdsale.sol#L111
- Crowdsale.sol#L114
- Crowdsale.sol#L119
- Crowdsale.sol#L122
- Crowdsale.sol#L154

Risk Level:

Likelihood - 1
Impact - 1

Recommendation:

Check all revert strings and consider using 32 bytes at maximum for revert messages or other strings in the protocol to optimize gas usage instead of these revert messages.

Remediation Plan:

SOLVED: The Floin team solved the issue by implementing **Custom Errors** instead of reverting messages.

Commit ID: 2ff5b47d2207bf76623598403089657a7c3dd574

3.14 (HAL-14) FOR LOOP OPTIMIZATIONS - INFORMATIONAL

Description:

It has been observed all `for` loops in the protocol are not optimized. Suboptimal for loops can cost too much gas.

These for loops can be optimized with the suggestions above:

1. In Solidity (pragma 0.8.0 and later), adding the `unchecked` keyword for arithmetical operations can reduce gas usage on contracts where underflow/underflow is unrealistic. It is possible to save gas by using this keyword on multiple code locations.
2. In all for loops, the `index` variable is incremented using `+=`. It is known that, in loops, using `++i` costs less gas per iteration than `+=`. This also affects incremented variables within the loop code block.
3. Do not initialize `index` variables with `0` Solidity already initializes these `uint` variables as zero.

Code Location:

- FLTKStaking.sol#L191
- FLTKStaking.sol#L219
- FLTKStaking.sol#L245
- FLTKStaking.sol#L280
- FLTKStaking.sol#L300
- VestingSplitter.sol#L44
- VestingSplitter.sol#L121

Risk Level:

Likelihood - 1
Impact - 1

Recommendation:

It is recommended to apply the following pattern for Solidity pragma version 0.8.0 and later.

Listing 22: Optimization Example

```
for (uint256 i; i < arrayLength; ) {
    . . .
    unchecked {
     ++i
    }
```

Remediation Plan:

SOLVED: The Floin team solved the issue by optimizing the for loops according to the **Recommendation** section.

Commit ID: 28fbe0738d0fcfd216489429a458c0481d24d44b

AUTOMATED TESTING

4.1 STATIC ANALYSIS SCAN

Description:

Halborn used automated testing techniques to enhance coverage of certain areas of the scoped contract. Slither, a Solidity static analysis framework, was used for static analysis. After Halborn verified all the contracts in the repository and was able to compile them correctly into their ABI and binary formats. 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.

Results:

```
Reentrancy in Crowdsale.buyTokens(uint256,address) (contracts/Crowdsale.sol#109-130):
        External calls:
        - paymentToken.safeTransferFrom(msg.sender,paymentCollector,payment) (contracts/Crowdsale.sol#126)
        - token.safeTransferFrom(tokenWallet,recipient,tokens) (contracts/Crowdsale.sol#127)
        Event emitted after the call(s):
        - TokenPurchase(msg.sender,recipient,payment,tokens) (contracts/Crowdsale.sol#129)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-3

Crowdsale.constructor(address,address,address,ERC20,ERC20,uint256,uint256,uint256) (contracts/Crowdsale.sol#69-99) uses timestamp for comparisons
        Dangerous comparisons:
        - require(bool,string)(_closingTime > block.timestamp,Crowdsale::closing time is invalid) (contracts/Crowdsale.sol#85)
Crowdsale.hasClosed() (contracts/Crowdsale.sol#162-164) uses timestamp for comparisons
        Dangerous comparisons:
        - block.timestamp > closingTime (contracts/Crowdsale.sol#163)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#block-timestamp

Address.verifyCallResult(bool,bytes,string) (node_modules/@openzeppelin/contracts/utils/Address.sol#201-221) uses assembly
        - INLINE ASM (node_modules/@openzeppelin/contracts/utils/Address.sol#213-216)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#assembly-usage

Different versions of Solidity are used:
        - Version used: ['0.8.17', '^0.8.0', '^0.8.1']
        - 0.8.17 (contracts/Crowdsale.sol#2)
        - ^0.8.0 (node_modules/@chainlink/contracts/src/v0.8/interfaces/AggregatorV3Interface.sol#2)
        - ^0.8.0 (node_modules/@openzeppelin/contracts/access/Ownable.sol#4)
        - ^0.8.0 (node_modules/@openzeppelin/contracts/security/Pausable.sol#4)
        - ^0.8.0 (node_modules/@openzeppelin/contracts/token/ERC20/ERC20.sol#4)
        - ^0.8.0 (node_modules/@openzeppelin/contracts/token/ERC20/IERC20.sol#4)
        - ^0.8.0 (node_modules/@openzeppelin/contracts/token/ERC20/extensions/IERC20Metadata.sol#4)
        - ^0.8.0 (node_modules/@openzeppelin/contracts/token/ERC20/extensions/draft-IERC20Permit.sol#4)
        - ^0.8.0 (node_modules/@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol#4)
        - ^0.8.1 (node_modules/@openzeppelin/contracts/utils/Address.sol#4)
        - ^0.8.0 (node_modules/@openzeppelin/contracts/utils/Context.sol#4)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#different-pragma-directives-are-used
```

```
Reentrancy in FLTKStaking.withdraw(uint256[],uint256) (contracts/FLTKStaking.sol#182-209):
        External calls:
        - _harvest(_depositIds) (contracts/FLTKStaking.sol#185)
                - returndata = address(token).functionCall(data,SafeERC20: low-level call failed) (node_modules/@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol#110)
                - (success,returndata) = target.call{value: value}(data) (node_modules/@openzeppelin/contracts/utils/Address.sol#137)
                - rewardToken.safeTransferFrom(rewardWallet,_getUser(),transferAmount) (contracts/FLTKStaking.sol#329)
        External calls sending eth:
        - _harvest(_depositIds) (contracts/FLTKStaking.sol#185)
                - (success,returndata) = target.call{value: value}(data) (node_modules/@openzeppelin/contracts/utils/Address.sol#137)
        State variables written after the call(s):
        - user.amount -= amountChunk (contracts/FLTKStaking.sol#196)
        - user.rewardDebt = user.amount * accTokenPerShare / UNITS (contracts/FLTKStaking.sol#197)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities

FLTKStaking.deposit(uint256) (contracts/FLTKStaking.sol#148-169) performs a multiplication on the result of a division:
        -userDepositId = block.timestamp / depositBucketSize * depositBucketSize (contracts/FLTKStaking.sol#152)
Math.mulDiv(uint256,uint256,uint256) (node_modules/@openzeppelin/contracts/utils/math/Math.sol#55-135) performs a multiplication on the result of a division:
        -denominator = denominator / twos (node_modules/@openzeppelin/contracts/utils/math/Math.sol#102)
        -inverse = (3 * denominator) ^ 2 (node_modules/@openzeppelin/contracts/utils/math/Math.sol#117)
Math.mulDiv(uint256,uint256,uint256) (node_modules/@openzeppelin/contracts/utils/math/Math.sol#55-135) performs a multiplication on the result of a division:
        -denominator = denominator / twos (node_modules/@openzeppelin/contracts/utils/math/Math.sol#102)
        -inverse *= 2 - denominator * inverse (node_modules/@openzeppelin/contracts/utils/math/Math.sol#121)
Math.mulDiv(uint256,uint256,uint256) (node_modules/@openzeppelin/contracts/utils/math/Math.sol#55-135) performs a multiplication on the result of a division:
        -denominator = denominator / twos (node_modules/@openzeppelin/contracts/utils/math/Math.sol#102)
        -inverse *= 2 - denominator * inverse (node_modules/@openzeppelin/contracts/utils/math/Math.sol#122)
Math.mulDiv(uint256,uint256,uint256) (node_modules/@openzeppelin/contracts/utils/math/Math.sol#55-135) performs a multiplication on the result of a division:
        -denominator = denominator / twos (node_modules/@openzeppelin/contracts/utils/math/Math.sol#102)
        -inverse *= 2 - denominator * inverse (node_modules/@openzeppelin/contracts/utils/math/Math.sol#123)
Math.mulDiv(uint256,uint256,uint256) (node_modules/@openzeppelin/contracts/utils/math/Math.sol#55-135) performs a multiplication on the result of a division:
        -denominator = denominator / twos (node_modules/@openzeppelin/contracts/utils/math/Math.sol#102)
        -inverse *= 2 - denominator * inverse (node_modules/@openzeppelin/contracts/utils/math/Math.sol#124)
Math.mulDiv(uint256,uint256,uint256) (node_modules/@openzeppelin/contracts/utils/math/Math.sol#55-135) performs a multiplication on the result of a division:
        -denominator = denominator / twos (node_modules/@openzeppelin/contracts/utils/math/Math.sol#102)
        -inverse *= 2 - denominator * inverse (node_modules/@openzeppelin/contracts/utils/math/Math.sol#125)
Math.mulDiv(uint256,uint256,uint256) (node_modules/@openzeppelin/contracts/utils/math/Math.sol#55-135) performs a multiplication on the result of a division:
        -denominator = denominator / twos (node_modules/@openzeppelin/contracts/utils/math/Math.sol#102)
        -inverse *= 2 - denominator * inverse (node_modules/@openzeppelin/contracts/utils/math/Math.sol#126)
Math.mulDiv(uint256,uint256,uint256) (node_modules/@openzeppelin/contracts/utils/math/Math.sol#55-135) performs a multiplication on the result of a division:
        -prod0 = prod0 / twos (node_modules/@openzeppelin/contracts/utils/math/Math.sol#105)
        -result = prod0 * inverse (node_modules/@openzeppelin/contracts/utils/math/Math.sol#132)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#divide-before-multiply

FLTKStaking.updatePoolRewards() (contracts/FLTKStaking.sol#92-112) uses a dangerous strict equality:
        - poolBalance == 0 (contracts/FLTKStaking.sol#100)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#dangerous-strict-equalities

Reentrancy in FLTKStaking.deposit(uint256) (contracts/FLTKStaking.sol#148-169):
        External calls:
        - stakingToken.safeTransferFrom(_userAddress,address(this),_amount) (contracts/FLTKStaking.sol#163)
        State variables written after the call(s):
        - user.amount += _amount (contracts/FLTKStaking.sol#165)
        - user.rewardDebt = user.amount * accTokenPerShare / UNITS (contracts/FLTKStaking.sol#166)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-1
```

```
Reentrancy in VestingSplitter.release(address) (contracts/VestingSplitter.sol#100-118):
        External calls:
        - vestingWallet.release(address(token)) (contracts/VestingSplitter.sol#107)
        State variables written after the call(s):
        - _released[account] += payment (contracts/VestingSplitter.sol#113)
        - _totalReleased += payment (contracts/VestingSplitter.sol#114)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-2

Reentrancy in VestingSplitter.release(address) (contracts/VestingSplitter.sol#100-118):
        External calls:
        - vestingWallet.release(address(token)) (contracts/VestingSplitter.sol#107)
        - SafeERC20.safeTransfer(token,account,payment) (contracts/VestingSplitter.sol#116)
        Event emitted after the call(s):
        - PaymentReleased(account,payment) (contracts/VestingSplitter.sol#117)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-3

VestingSplitter.release(address) (contracts/VestingSplitter.sol#100-118) uses timestamp for comparisons
        Dangerous comparisons:
        - vested > 0 (contracts/VestingSplitter.sol#106)
        - require(bool,string)(payment != 0,VestingSplitter: account is not due payment) (contracts/VestingSplitter.sol#111)
VestingSplitter._addPayee(address,uint256) (contracts/VestingSplitter.sol#143-156) uses timestamp for comparisons
        Dangerous comparisons:
        - require(bool,string)(block.timestamp < _start,VestingSplitter: Vesting has already started) (contracts/VestingSplitter.sol#144)
VestingWallet._vestingSchedule(uint256,uint64) (node_modules/@openzeppelin/contracts/finance/VestingWallet.sol#126-134) uses timestamp for comparisons
        Dangerous comparisons:
        - timestamp < start() (node_modules/@openzeppelin/contracts/finance/VestingWallet.sol#127)
        - timestamp > start() + duration() (node_modules/@openzeppelin/contracts/finance/VestingWallet.sol#129)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#block-timestamp

Address.verifyCallResult(bool,bytes,string) (node_modules/@openzeppelin/contracts/utils/Address.sol#201-221) uses assembly
        - INLINE ASM (node_modules/@openzeppelin/contracts/utils/Address.sol#213-216)
Math.mulDiv(uint256,uint256,uint256) (node_modules/@openzeppelin/contracts/utils/math/Math.sol#55-135) uses assembly
        - INLINE ASM (node_modules/@openzeppelin/contracts/utils/math/Math.sol#66-70)
        - INLINE ASM (node_modules/@openzeppelin/contracts/utils/math/Math.sol#86-93)
        - INLINE ASM (node_modules/@openzeppelin/contracts/utils/math/Math.sol#100-109)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#assembly-usage

Different versions of Solidity are used:
        - Version used: ['0.8.17', '^0.8.0', '^0.8.1']
        - 0.8.17 (contracts/VestingSplitter.sol#3)
        - ^0.8.0 (node_modules/@openzeppelin/contracts/access/Ownable.sol#4)
        - ^0.8.0 (node_modules/@openzeppelin/contracts/finance/VestingWallet.sol#3)
        - ^0.8.0 (node_modules/@openzeppelin/contracts/token/ERC20/IERC20.sol#4)
        - ^0.8.0 (node_modules/@openzeppelin/contracts/token/ERC20/extensions/draft-IERC20Permit.sol#4)
        - ^0.8.0 (node_modules/@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol#4)
        - ^0.8.1 (node_modules/@openzeppelin/contracts/utils/Address.sol#4)
        - ^0.8.0 (node_modules/@openzeppelin/contracts/utils/Context.sol#4)
        - ^0.8.0 (node_modules/@openzeppelin/contracts/utils/math/Math.sol#4)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#different-pragma-directives-are-used
```

As a result of the tests carried out with the Slither tool, some results were obtained and these results were reviewed by Halborn. Based on the results reviewed, some vulnerabilities were determined to be false positives and these results were not included in the report. The actual vulnerabilities found by Slither are already included in the report findings.

4.2 AUTOMATED SECURITY SCAN

Description:

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

Results:

Report for contracts/FLTKStaking.sol
https://dashboard.mythx.io/#/console/analyses/add4ab11-cc9e-44d7-b7df-953b5737ce92

Line	SWC Title	Severity	Short Description
13	(SWC-101) Integer Overflow and Underflow	Unknown	Arithmetic operation "**" discovered
48	(SWC-110) Assert Violation	Unknown	Public state variable with array type causing reacheable exception by default.
109	(SWC-101) Integer Overflow and Underflow	Unknown	Arithmetic operation "/" discovered
109	(SWC-101) Integer Overflow and Underflow	Unknown	Arithmetic operation "+=" discovered
109	(SWC-101) Integer Overflow and Underflow	Unknown	Arithmetic operation "*" discovered
123	(SWC-101) Integer Overflow and Underflow	Unknown	Arithmetic operation "-" discovered
123	(SWC-101) Integer Overflow and Underflow	Unknown	Arithmetic operation "*" discovered
152	(SWC-101) Integer Overflow and Underflow	Unknown	Arithmetic operation "/" discovered
152	(SWC-101) Integer Overflow and Underflow	Unknown	Arithmetic operation "*" discovered
155	(SWC-110) Assert Violation	Unknown	Out of bounds array access
165	(SWC-101) Integer Overflow and Underflow	Unknown	Arithmetic operation "+=" discovered
166	(SWC-101) Integer Overflow and Underflow	Unknown	Arithmetic operation "*" discovered
166	(SWC-101) Integer Overflow and Underflow	Unknown	Arithmetic operation "/" discovered
191	(SWC-101) Integer Overflow and Underflow	Unknown	Arithmetic operation "++" discovered
192	(SWC-110) Assert Violation	Unknown	Out of bounds array access
196	(SWC-101) Integer Overflow and Underflow	Unknown	Arithmetic operation "-=" discovered
197	(SWC-101) Integer Overflow and Underflow	Unknown	Arithmetic operation "*" discovered
197	(SWC-101) Integer Overflow and Underflow	Unknown	Arithmetic operation "/" discovered
198	(SWC-101) Integer Overflow and Underflow	Unknown	Arithmetic operation "-=" discovered
219	(SWC-101) Integer Overflow and Underflow	Unknown	Arithmetic operation "++" discovered
220	(SWC-110) Assert Violation	Unknown	Out of bounds array access
224	(SWC-101) Integer Overflow and Underflow	Unknown	Arithmetic operation "+=" discovered
245	(SWC-101) Integer Overflow and Underflow	Unknown	Arithmetic operation "++" discovered
246	(SWC-110) Assert Violation	Unknown	Out of bounds array access

Figure 1: MythX Result - 1

Report for contracts/Crowdsale.sol
https://dashboard.mythx.io/#/console/analyses/bdb57d26-22ee-4277-9e35-6f7e9c405f24

Line	SWC Title	Severity	Short Description
41	(SWC-108) State Variable Default Visibility	Low	State variable visibility is not set.
116	(SWC-101) Integer Overflow and Underflow	Unknown	Arithmetic operation "*" discovered
116	(SWC-101) Integer Overflow and Underflow	Unknown	Arithmetic operation "**" discovered
116	(SWC-101) Integer Overflow and Underflow	Unknown	Arithmetic operation "/" discovered
124	(SWC-101) Integer Overflow and Underflow	Unknown	Arithmetic operation "+=" discovered

Figure 2: MythX Result - 2

Report for contracts/VestingSplitter.sol
https://dashboard.mythx.io/#/console/analyses/40a05ff6-7419-4bd4-9b2b-7516e9301143

Line	SWC Title	Severity	Short Description
44	(SWC-101) Integer Overflow and Underflow	Unknown	Arithmetic operation "++" discovered
45	(SWC-110) Assert Violation	Unknown	Out of bounds array access
83	(SWC-110) Assert Violation	Unknown	Out of bounds array access
104	(SWC-101) Integer Overflow and Underflow	Unknown	Arithmetic operation "-" discovered
113	(SWC-101) Integer Overflow and Underflow	Unknown	Arithmetic operation "+=" discovered
114	(SWC-101) Integer Overflow and Underflow	Unknown	Arithmetic operation "+=" discovered
121	(SWC-101) Integer Overflow and Underflow	Unknown	Arithmetic operation "++" discovered
122	(SWC-110) Assert Violation	Unknown	Out of bounds array access
135	(SWC-101) Integer Overflow and Underflow	Unknown	Arithmetic operation "*" discovered
135	(SWC-101) Integer Overflow and Underflow	Unknown	Arithmetic operation "/" discovered
135	(SWC-101) Integer Overflow and Underflow	Unknown	Arithmetic operation "-" discovered
152	(SWC-101) Integer Overflow and Underflow	Unknown	Arithmetic operation "+=" discovered
153	(SWC-101) Integer Overflow and Underflow	Unknown	Arithmetic operation "+=" discovered

Figure 3: MythX Result - 3

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

THANK YOU FOR CHOOSING

// HALBORN