Seascape - MSCP Token & Vesting

Smart Contract Security Audit

Prepared by: **Halborn**

Date of Engagement: **November 12th, 2021 - November 19th, 2021**

Visit: **Halborn.com**

DOCUMENT REVISION HISTORY

VERSION	MODIFICATION	DATE	AUTHOR
0.1	Document Creation	11/17/2021	Ferran.Celades
0.2	Document Edits	11/17/2021	Ferran.Celades
0.3	Draft Review	11/17/2021	Gabi Urrutia
1.0	Remediation Plan	11/18/2021	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
Ferran Celades	Halborn	Ferran.Celades@halborn.com

EXECUTIVE OVERVIEW

1.1 INTRODUCTION

Seascape engaged Halborn to conduct a security audit on their MSCP Token and vesting contracts beginning on November 12th, 2021 and ending on November 19th, 2021. The security assessment was scoped to the smart contracts provided in the Github repository blocklords/moonscape-smartcontracts/contracts/riverboat

1.2 AUDIT SUMMARY

The team at Halborn was provided two weeks for the engagement and assigned a full time security engineer to audit the security of the smart contract. The security engineer is a blockchain and smart-contract security expert with advanced penetration testing, smart-contract hacking, and deep knowledge of multiple blockchain protocols.

The purpose of this audit is to:

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

In summary, Halborn identified few security risks that were mostly addressed by the Seascape 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
- 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

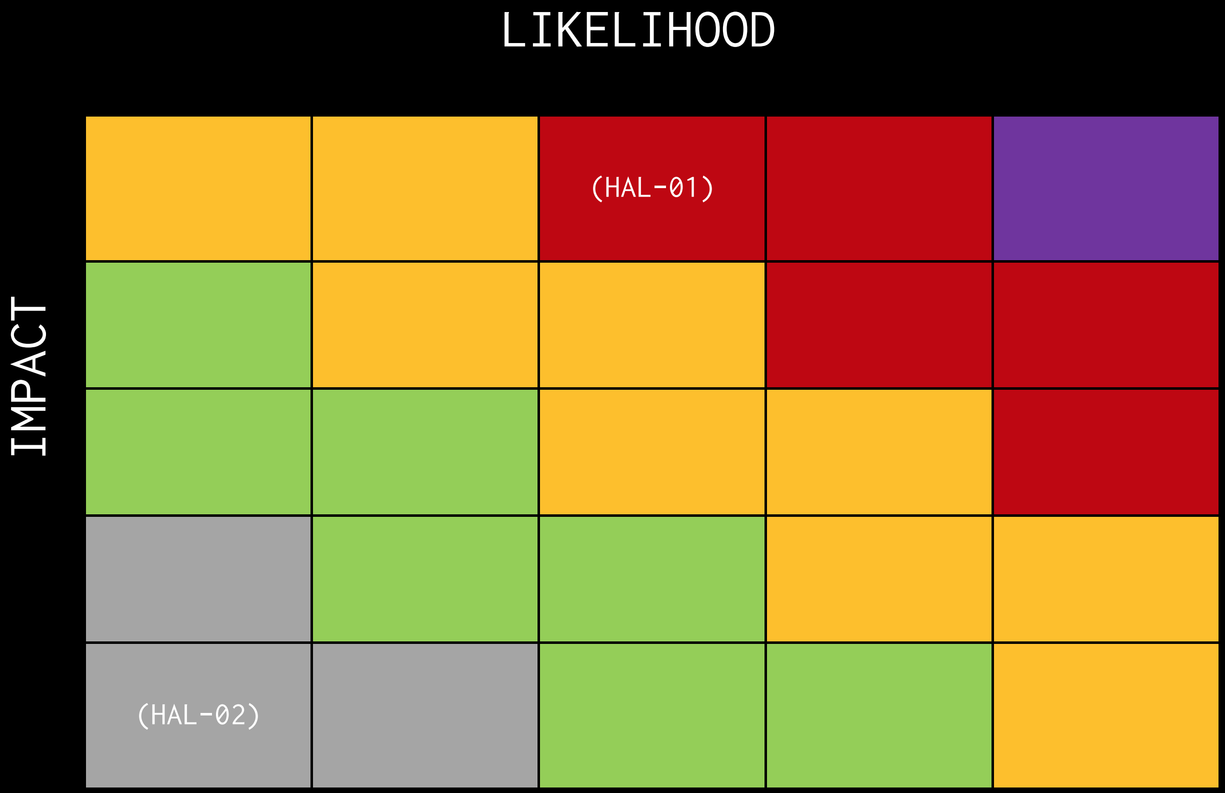
1.4 SCOPE

The security assessment was scoped to the following
blocklords/moonscape-smartcontracts/contracts/riverboat
- MscpToken.sol
- MscpVesting.sol
- All contracts inherited by these contracts

Commit ID: 9558f291d70c35828176823165316f128ccfb024

2. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	1	0	0	1



SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
(HAL-01) INVESTOR MULTI-WITHDRAW IF ADDED AGAIN	High	SOLVED IN THE DEPLOYMENT
(HAL-02) UNUSED CODE	Informational	SOLVED - 11/18/2021

FINDINGS & TECH DETAILS

3.1 (HAL-01) INVESTOR MULTI-WITHDRAW IF ADDED AGAIN - HIGH

Description:

The addInvestor function, called only by the owner, does not check if the investor does have claimed already the tokens. The only performed check is the remainingCoins == 0 which will be true after the vesting period finishes (150 days for strategic and 300 days for private).

When the vesting period finishes, the owner can call addInvestor, this will reset the remainingCoins for the investor. Having the reset the amount of remaining tokens allows the investor to withdraw again the max amount of tokens. This is possible since the getDuration function will return the total duration (startTime did not change), allowing to re-claim the total tokens again. This process can be repeated.

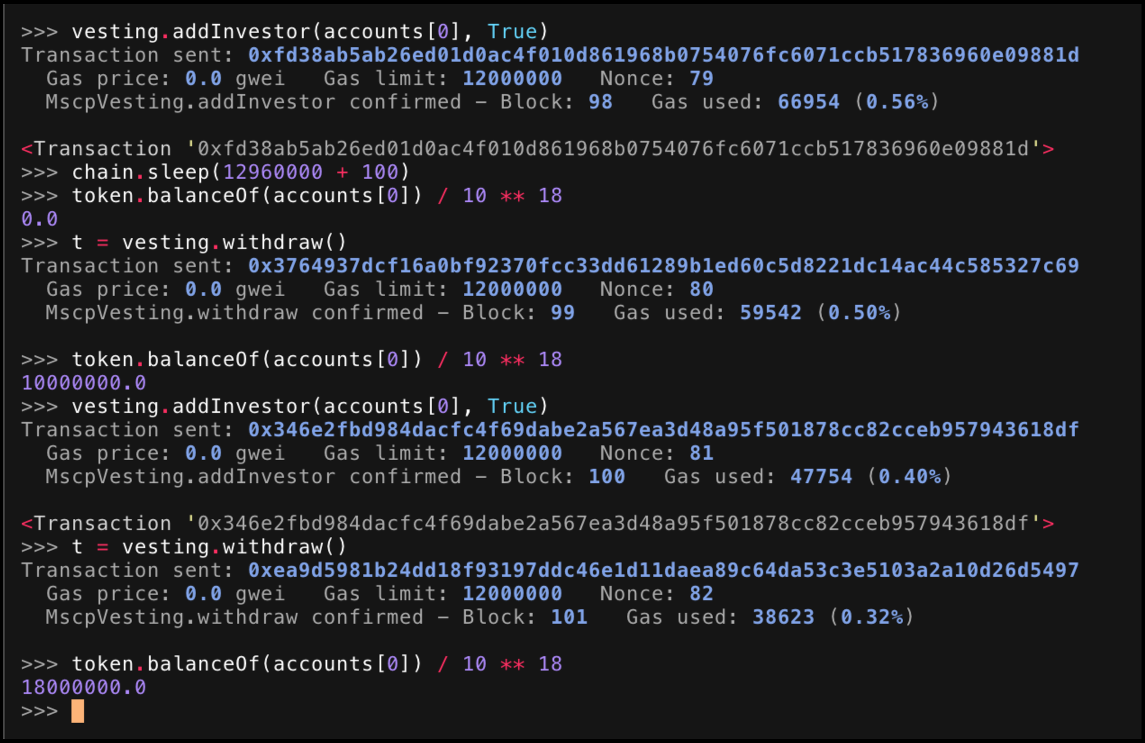
POC:

- Add an investor with addInvestor
- Wait for the vesting period to finish (150/300 days)
- Withdraw the tokens using the investor account
- Owner can call again the addInvestor function
- Withdraw the tokens again

Code Location:

Listing 1: contracts/MscpVesting (Lines 51)

```
50 function addInvestor (address _investor, bool _strategicInvestor)
       external onlyOwner {
51     require(balances[_investor].remainingCoins == 0, "investor
           already has allocation");
52
53     if(_strategicInvestor){
```



```
>>> vesting.addInvestor(accounts[0], True)
Transaction sent: 0xfd38ab5ab26ed01d0ac4f010d861968b0754076fc6071ccb517836960e09881d
  Gas price: 0.0 gwei   Gas limit: 12000000   Nonce: 79
  MscpVesting.addInvestor confirmed - Block: 98   Gas used: 66954 (0.56%)

<Transaction '0xfd38ab5ab26ed01d0ac4f010d861968b0754076fc6071ccb517836960e09881d'>
>>> chain.sleep(12960000 + 100)
>>> token.balanceOf(accounts[0]) / 10 ** 18
0.0
>>> t = vesting.withdraw()
Transaction sent: 0x3764937dcf16a0bf92370fcc33dd61289b1ed60c5d8221dc14ac44c585327c69
  Gas price: 0.0 gwei   Gas limit: 12000000   Nonce: 80
  MscpVesting.withdraw confirmed - Block: 99   Gas used: 59542 (0.50%)

>>> token.balanceOf(accounts[0]) / 10 ** 18
10000000.0
>>> vesting.addInvestor(accounts[0], True)
Transaction sent: 0x346e2fbd984dacfc4f69dabe2a567ea3d48a95f501878cc82cceb957943618df
  Gas price: 0.0 gwei   Gas limit: 12000000   Nonce: 81
  MscpVesting.addInvestor confirmed - Block: 100   Gas used: 47754 (0.40%)

<Transaction '0x346e2fbd984dacfc4f69dabe2a567ea3d48a95f501878cc82cceb957943618df'>
>>> t = vesting.withdraw()
Transaction sent: 0xea9d5981b24dd18f93197ddc46e1d11daea89c64da53c3e5103a2a10d26d5497
  Gas price: 0.0 gwei   Gas limit: 12000000   Nonce: 82
  MscpVesting.withdraw confirmed - Block: 101   Gas used: 38623 (0.32%)

>>> token.balanceOf(accounts[0]) / 10 ** 18
18000000.0
>>> 
```

Figure 1: PoC showing the double withdraw

```
            balances[_investor].remainingCoins = TOTAL_STRATEGIC;
            balances[_investor].strategicInvestor = true;
        } else
            balances[_investor].remainingCoins = TOTAL_PRIVATE;

        emit InvestorModified(_investor, balances[_investor].
            remainingCoins);
}
```

Risk Level:

Likelihood - 3
Impact - 5

Recommendation:

Check if the investor already has claimedBonus tokens. If this value is different from zero, it means that the investor is already vested.

Remediation Plan:

SOLVED IN THE DEPLOYMENT: The issue will be solved in the deployment implementing a multi-signature wallet.

3.2 (HAL-02) UNUSED CODE - INFORMATIONAL

Description:

The burn function on the MscpToken does contain unnecessary code.

Code Location:

Listing 2: contracts/MscpToken.sol (Lines 97)

```
    function burn(uint256 amount) public onlyBridge {
        require(false, "Only burnFrom is allowed");
    }
```

Risk Level:

Likelihood - 1
Impact - 1

Recommendation:

It is recommended to remove the onlyBridge modifier and directly raise “Only burnFrom is allowed”.

Remediation Plan:

SOLVED: The code was removed by the Seascape team.

MANUAL TESTING

4.1 Introduction

Halborn performed different manual tests in all the contracts, trying to find logic flaws and vulnerabilities that were not detected by the automatic tools.

During the manual testing, multiple questions were considered while evaluating each of the defined functions:

- Can it be re-called changing admin/roles and permissions?
- Can somehow an external controlled contract call again the function during the execution of it? (Re-entrancy)
- Can a function be called twice in the same block causing issues?
- Do we control sensitive or vulnerable parameters?
- Does the function check for boundaries on the parameters and internal values? Bigger than zero or equal? Argument count, array sizes, integer truncation. . .
- Are the function parameters and variables controlled by external contracts?
- Can extended contracts cause issues on the extender contract?

The following graph, Figure 2, displays the inheritance on both audited contracts.

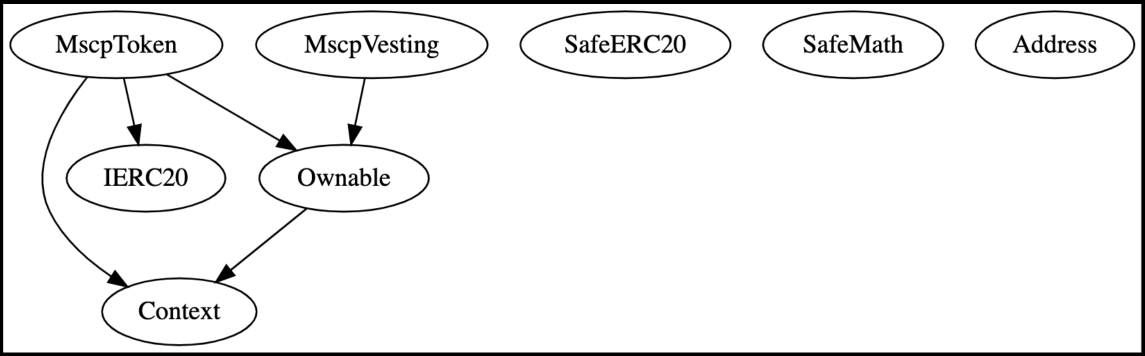


Figure 2: MscpToken and MscpVesting inheritance graph

The Figure 3 and Figure 4 show the call flow present on the contracts.

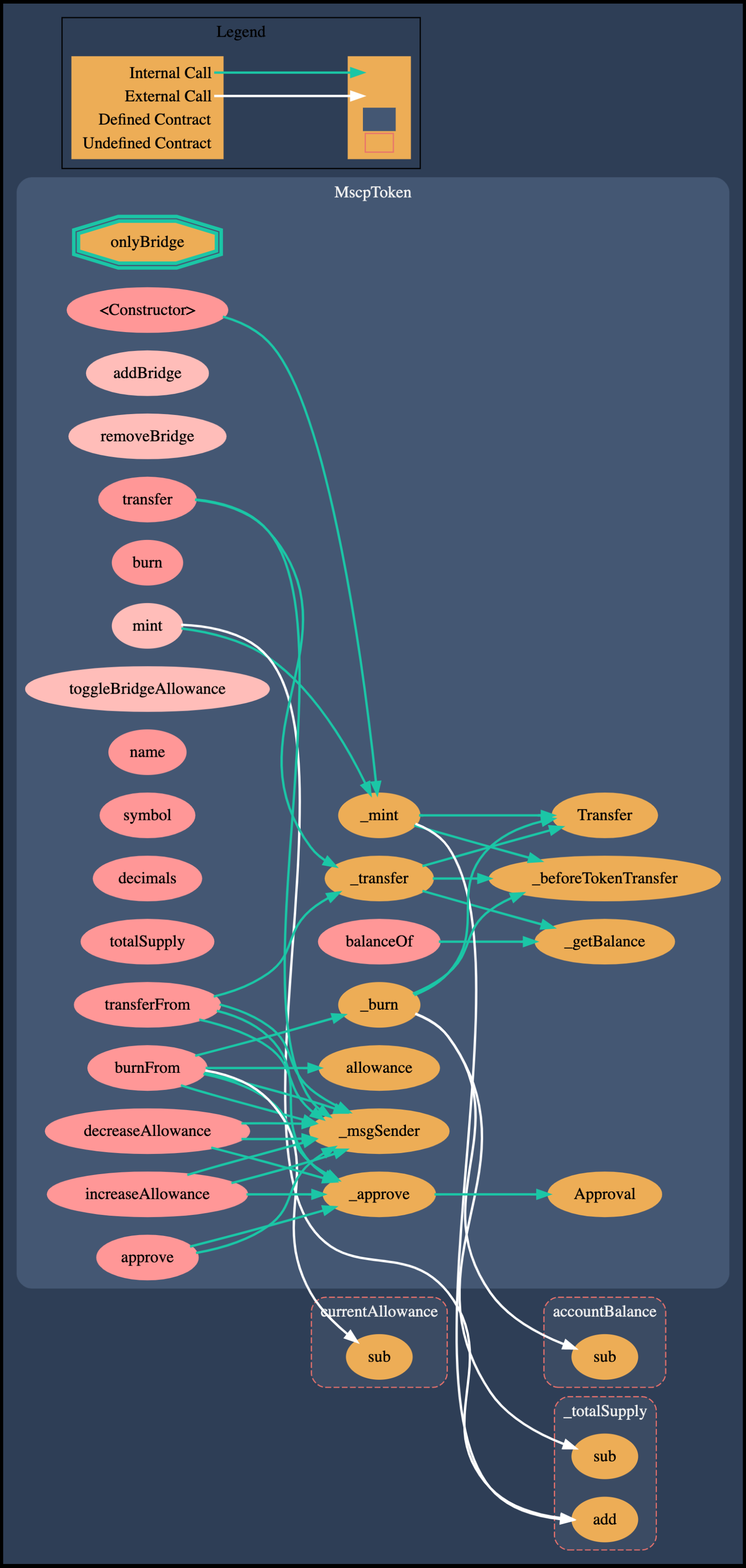


Figure 3: MscpToken call graph

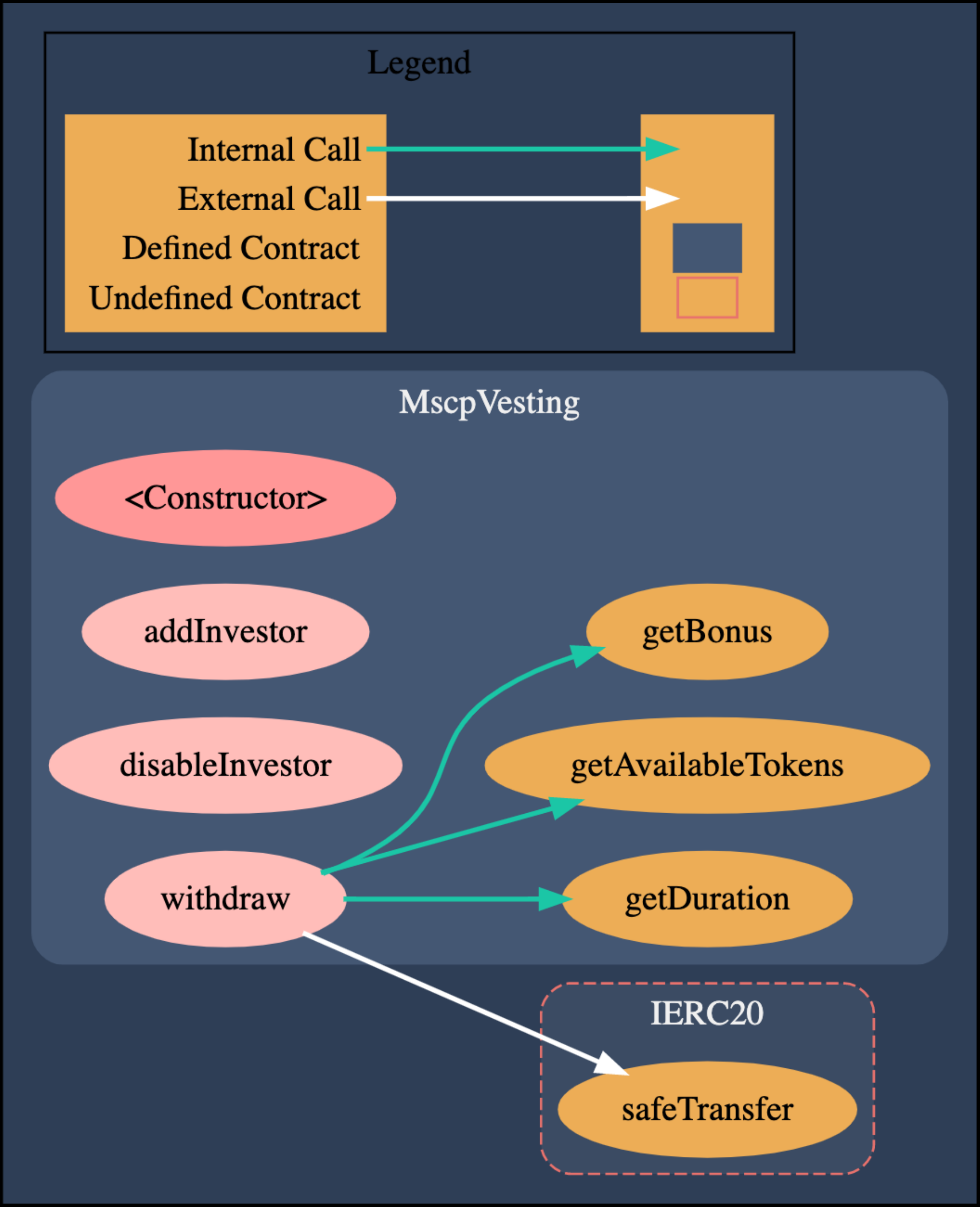


Figure 4: MscpVesting call graph

4.2 Stateful testing

The MscpToken was fully verified for being ERC20 compliant before the stateful testing.

[[attachments/token_erc20_compliant.png]]

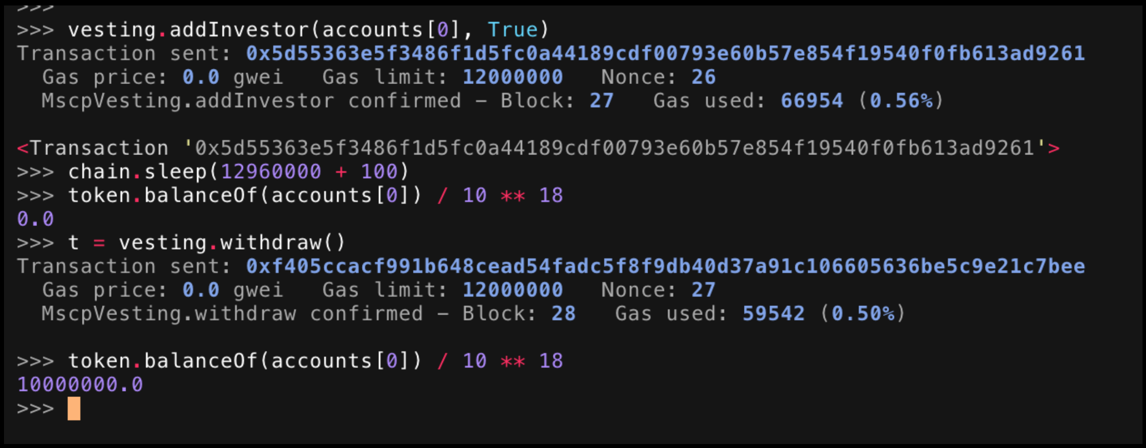
The token was tested against stateful testing scripts made to verify current existing token implementations such as OpenZeppelin ERC20 tokens ERC20-PBT

The contract was executed against:
- StateMachine
- MintingStateMachine (since mint functionality exists)

The results showed no issues at all.

4.3 Manual checks

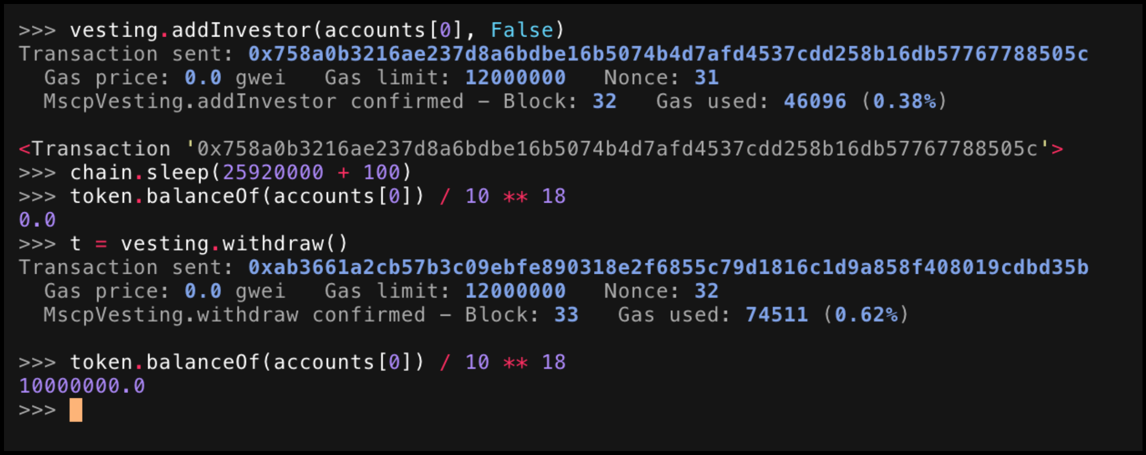
Both strategic and private should have the same amount at the end of the period. Checking for max withdrawal after the period finished for an strategic account, and a none strategic account (private):



```
>>> vesting.addInvestor(accounts[0], True)
Transaction sent: 0x5d55363e5f3486f1d5fc0a44189cdf00793e60b57e854f19540f0fb613ad9261
  Gas price: 0.0 gwei   Gas limit: 12000000   Nonce: 26
  MscpVesting.addInvestor confirmed - Block: 27   Gas used: 66954 (0.56%)

<Transaction '0x5d55363e5f3486f1d5fc0a44189cdf00793e60b57e854f19540f0fb613ad9261'>
>>> chain.sleep(12960000 + 100)
>>> token.balanceOf(accounts[0]) / 10 ** 18
0.0
>>> t = vesting.withdraw()
Transaction sent: 0xf405ccacf991b648cead54fadc5f8f9db40d37a91c106605636be5c9e21c7bee
  Gas price: 0.0 gwei   Gas limit: 12000000   Nonce: 27
  MscpVesting.withdraw confirmed - Block: 28   Gas used: 59542 (0.50%)

>>> token.balanceOf(accounts[0]) / 10 ** 18
10000000.0
>>> 
```



```
>>> vesting.addInvestor(accounts[0], False)
Transaction sent: 0x758a0b3216ae237d8a6bdbe16b5074b4d7afd4537cdd258b16db57767788505c
  Gas price: 0.0 gwei   Gas limit: 12000000   Nonce: 31
  MscpVesting.addInvestor confirmed - Block: 32   Gas used: 46096 (0.38%)

<Transaction '0x758a0b3216ae237d8a6bdbe16b5074b4d7afd4537cdd258b16db57767788505c'>
>>> chain.sleep(25920000 + 100)
>>> token.balanceOf(accounts[0]) / 10 ** 18
0.0
>>> t = vesting.withdraw()
Transaction sent: 0xab3661a2cb57b3c09ebfe890318e2f6855c79d1816c1d9a858f408019cdbd35b
  Gas price: 0.0 gwei   Gas limit: 12000000   Nonce: 32
  MscpVesting.withdraw confirmed - Block: 33   Gas used: 74511 (0.62%)

>>> token.balanceOf(accounts[0]) / 10 ** 18
10000000.0
>>> 
```

AUTOMATED TESTING

5.1 STATIC ANALYSIS REPORT

Description:

Halborn used automated testing techniques to enhance the coverage of certain areas of the scoped contracts. Among the tools used was Slither, a Solidity static analysis framework. After Halborn verified all the contracts in the repository and was able to compile them correctly into their ABI and binary formats, Slither was run on the all-scoped contracts. This tool can statically verify mathematical relationships between Solidity variables to detect invalid or inconsistent usage of the contracts' APIs across the entire code-base.

Slither results:

Slither did not reveal any security issue or concern.

THANK YOU FOR CHOOSING

HALBORN