// HALBORN

Stater - Vesting Contract

Smart Contract Security Audit

Prepared by: **Halborn**

Date of Engagement: **March 25, 2021**

Visit: **Halborn.com**

DOCUMENT REVISION HISTORY

VERSION	MODIFICATION	DATE	AUTHOR
0.1	Document Creation	03/24/2021	Gabi Urrutia
0.2	Document Additions	03/27/2021	Gabi Urrutia
0.3	Document Review	03/30/2021	Nishit Majithia
1.0	Final Version	03/30/2021	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

EXECUTIVE OVERVIEW

1.1 INTRODUCTION

Stater is a Lending platform for staking to receive fee discounts for borrowing or lending or hodl NFT and receive community drops and other perks inside our ecosystem.

The security assessment was scoped to the smart contract TokenVesting.sol . An audit of the security risk and implications regarding the changes introduced by the development team at Stater prior to its production release shortly following the assessments deadline.

Overall, due to time and resource constraints, only testing and verification of essential properties were performed to achieve objectives and deliverable set in the scope. It is important to remark that using the outdated TokenVesting contract could be dangerous if a new vulnerability is discovered.

1.2 AUDIT SUMMARY

The team at Halborn was provided a week timeframe 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 experience in advanced penetration testing, smart contract hacking, and has a deep knowledge in multiple blockchain protocols.

The purpose of this audit to achieve the following:
- Ensure that smart contract functions are intended.
- Identify potential security issues with the smart contracts.

In summary, Halborn identified 6 security risks, and recommends performing further testing to validate extended safety and correctness in context to the whole set of contracts. External threats, such as economic attacks, oracle attacks, and inter-contract functions and calls should be validated for expected logic and state.

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 the smart contract audit.
While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of smart contracts and can quickly identify items that 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 read 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.
- Scanning of solidity files for vulnerabilities, security hotspots, or bugs. (MythX)
- Static Analysis of security for scoped contract, and imported functions. (Slither)
- Security Testing Exploitation (teEther)
- Testnet deployment (Truffle, Ganache)

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:
Code related to TokenVesting.sol smart contract.

Specific commit of contract:

1eb378a13e48a44d9c02e3bc64d64375c16b3e80

OUT-OF-SCOPE:
Other smart contracts in the repository, external libraries and economics attacks.

2. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	0	1	3	2

LIKELIHOOD

IMPACT

🟨	🟨	🟥	🟥	🟪
🟩 (HAL-02) (HAL-03)	🟨 (HAL-01)	🟨	🟥	🟥
🟩 (HAL-04)	🟩	🟨	🟨	🟥
⬜ (HAL-06)	🟩	🟩	🟨	🟨
⬜ (HAL-05)	⬜	🟩	🟩	🟨

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
TOKENVESTING CONTRACT OUTDATED	Medium	-
LOW LEVEL CALLS	Low	-
USE OF BLOCK.TIMESTAMP	Low	-
PRAGMA VERSION DEPRECATED	Low	-
POSSIBLE MISUSE OF PUBLIC FUNCTIONS	Informational	-
USE OF INLINE ASSEMBLY	Informational	-
STATIC ANALYSIS REPORT	-	-
AUTOMATED SECURITY SCAN	-	-
SECURITY TESTING EXPLOITATION	-	-

FINDINGS & TECH DETAILS

3.1 (HAL-01) TOKENVESTING CONTRACT OUTDATED - MEDIUM

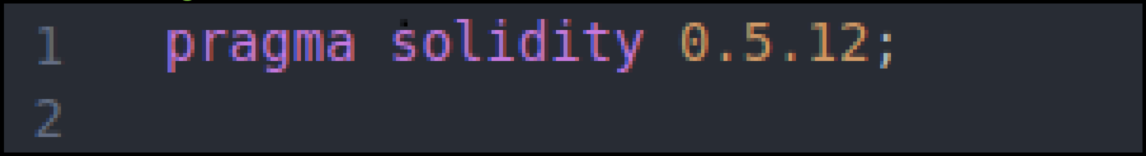
Description:

TokenVesting is used by Stater team with pragma 0.5.12. However, the contract was removed from OpenZeppelin contracts package. In addition, the latest version of the contract before being removed uses the pragma version ˆ0.6.0.

References: https://blog.openzeppelin.com/props-token-contracts-audit-2/
https://raw.githubusercontent.com/OpenZeppelin/openzeppelin-contracts/956d6632d9538f63c22d1eb71cda15f9848b56b7/contracts/drafts/TokenVesting.sol

Code Location:

TokenVesting.sol: Line #1



```
pragma solidity 0.5.12;
```

Risk Level:

Likelihood - 2
Impact - 4

Recommendations:

If possible, it is recommended other alternatives to use the token locking capability such as the use of EIP-1132 (Extending ERC20 with token locking capability) and Time-locked Wallets. If not possible, it is recommended to use the latest version.

References: https://eips.ethereum.org/EIPS/eip-1132
https://www.toptal.com/ethereum-smart-contract/time-locked-wallet-truffle-tutorial

3.2 (HAL-02) LOW LEVEL CALLS - LOW

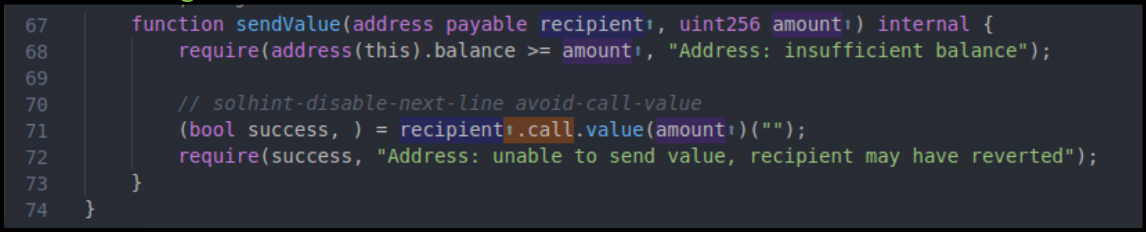
Description:

In Solidity, the use of low-level calls could cause unexpected behaviour if the call fails or an attacker provokes the call to fail. Then, it is a good practice not to use low-level calls in order to avoid a potentially exploitable behaviour. If the return value of a message call is checked before, execution will not resume because the called contract will throw an exception.

Reference: https://swcregistry.io/docs/SWC-104

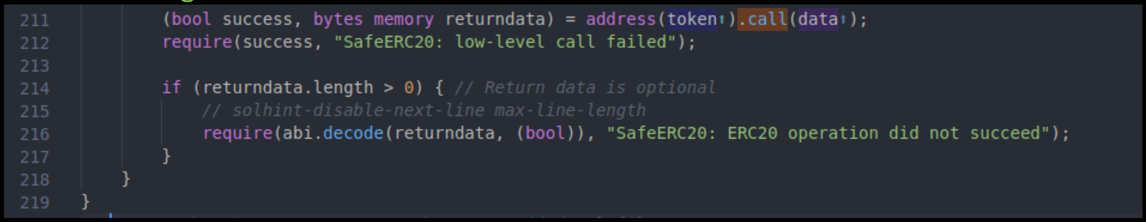
Code Location:

TokenVesting.sol: Line #71



```
67      function sendValue(address payable recipient, uint256 amount) internal {
68          require(address(this).balance >= amount, "Address: insufficient balance");
69
70          // solhint-disable-next-line avoid-call-value
71          (bool success, ) = recipient.call.value(amount)("");
72          require(success, "Address: unable to send value, recipient may have reverted");
73      }
74  }
```

TokenVesting.sol: Line #211



```
211             (bool success, bytes memory returndata) = address(token).call(data);
212             require(success, "SafeERC20: low-level call failed");
213
214             if (returndata.length > 0) { // Return data is optional
215                 // solhint-disable-next-line max-line-length
216                 require(abi.decode(returndata, (bool)), "SafeERC20: ERC20 operation did not succeed");
217             }
218         }
219     }
```

Risk Level:

Likelihood - 1
Impact - 4

Recommendations:

If possible, it is recommended not to use low-level calls in smart contracts. In that case, TokenVesting.sol is a flattened contract and includes some code from OpenZeppelin libraries in the contract code. In order to avoid the use of low-level calls in the contract, consider to import the OpenZeppelin libraries instead of using the flattened contract.

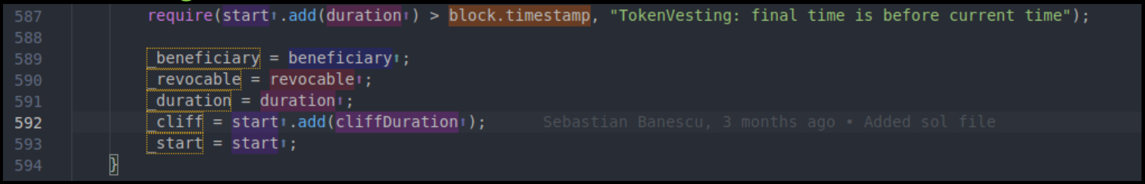
3.3 (HAL-03) USE OF BLOCK.TIMESTAMP - LOW

Description:

During a manual static review, the tester noticed the use of ‘‘now’’ in TokenVesting.sol. The contract developers should be aware that this does not mean current time. now is an alias for block.timestamp. The value of block.timestamp can be influenced by miners to a certain degree, so the testers should be warned that this may have some risk if miners collude on time manipulation to influence the price oracles. Miners can influence the timestamp by a tolerance of 900 seconds.

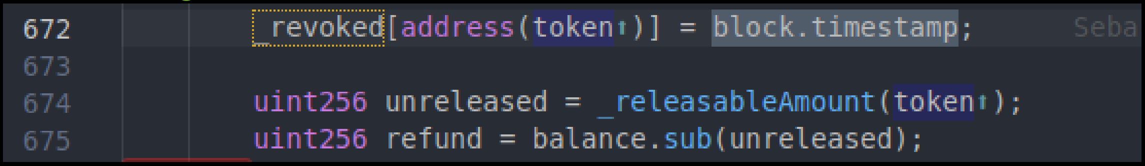
Code Location:

TokenVesting.sol: Line #587



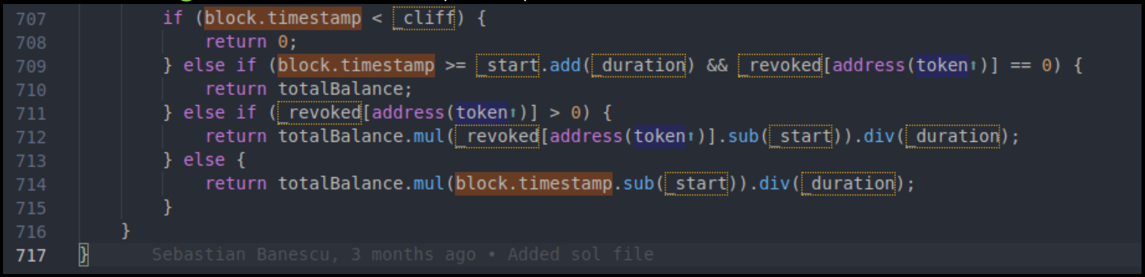
```
587         require(start.add(duration) > block.timestamp, "TokenVesting: final time is before current time");
588
589         _beneficiary = beneficiary;
590         _revocable = revocable;
591         _duration = duration;
592         _cliff = start.add(cliffDuration);       Sebastian Banescu, 3 months ago • Added sol file
593         _start = start;
594     }
```

TokenVesting.sol: Line #672



```
672         _revoked[address(token)] = block.timestamp;        Seba
673
674         uint256 unreleased = _releasableAmount(token);
675         uint256 refund = balance.sub(unreleased);
```

TokenVesting.sol: Line #707, 709, 714



```
707         if (block.timestamp < _cliff) {
708             return 0;
709         } else if (block.timestamp >= _start.add(_duration) && _revoked[address(token)] == 0) {
710             return totalBalance;
711         } else if (_revoked[address(token)] > 0) {
712             return totalBalance.mul(_revoked[address(token)].sub(_start)).div(_duration);
713         } else {
714             return totalBalance.mul(block.timestamp.sub(_start)).div(_duration);
715         }
716     }
717 }       Sebastian Banescu, 3 months ago • Added sol file
```

Recommendation:

Use block.number instead of block.timestamp or now reduce the influence of miners. Check if the timescale of the project occurs across years, days and months rather than seconds. If possible, it is recommended to use Oracles.

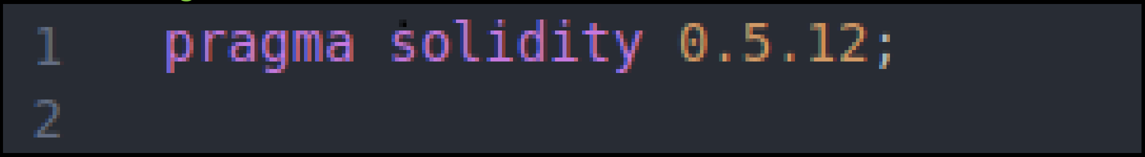
3.4 (HAL-04) PRAGMA VERSION DEPRECATED - LOW

Description:

The current version in use for the contracts is pragma 0.5.12. While this version is still functional, and most security issues safely implemented by mitigating contracts with other utility contracts such as SafeMath.sol and ReentrancyGuard.sol, the risk to the long-term sustainability and integrity of the solidity code increases.

Code Location:

TokenVesting.sol: Line #1



```
pragma solidity 0.5.12;

```

Risk Level:

Likelihood - 1
Impact - 3

Recommendations:

At the time of this audit, the current version is already at 0.8.2. When possible, use the most updated and tested pragma versions to take advantage of new features that provide checks and accounting, as well as prevent insecure use of code. (0.6.12)

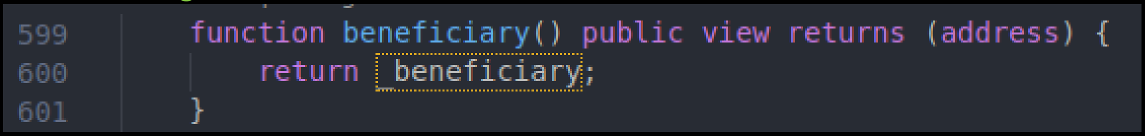
3.5 (HAL-05) POSSIBLE MISUSE OF PUBLIC FUNCTIONS - INFORMATIONAL

Description:

In public functions, array arguments are immediately copied array 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, the function expects its arguments being located in memory when the compiler generates the code for an internal function.

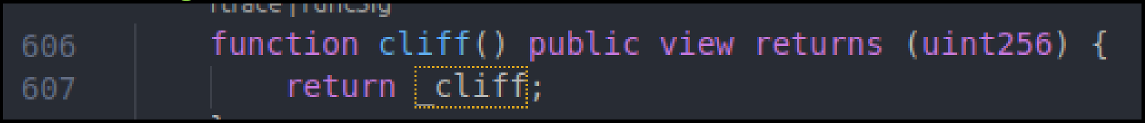
Code Location:

TokenVesting.sol: Line #599



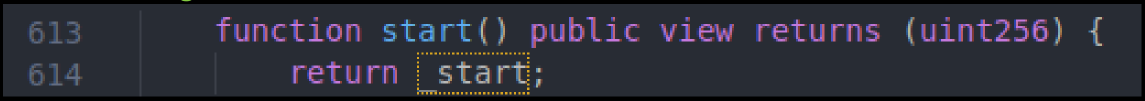
```
599     function beneficiary() public view returns (address) {
600         return _beneficiary;
601     }
```

TokenVesting.sol: Line #606



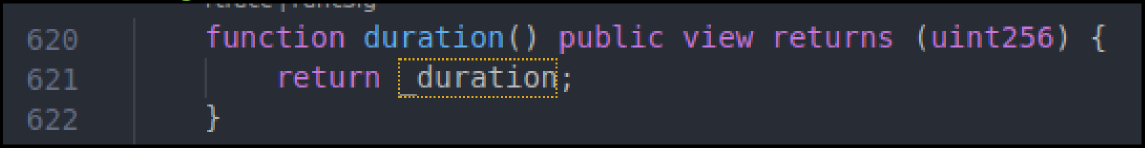
```
606     function cliff() public view returns (uint256) {
607         return _cliff;
```

TokenVesting.sol: Line #613



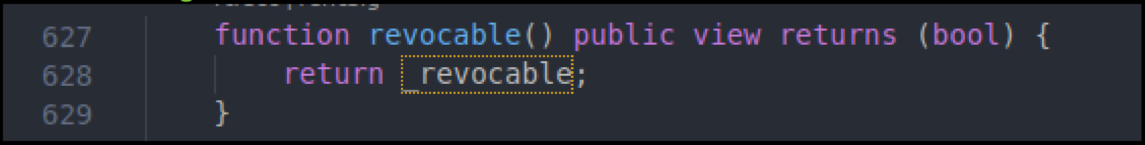
```
613     function start() public view returns (uint256) {
614         return _start;
```

TokenVesting.sol: Line #620



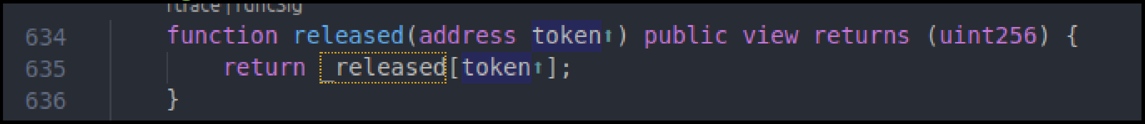
```
620     function duration() public view returns (uint256) {
621         return _duration;
622     }
```

TokenVesting.sol: Line #627



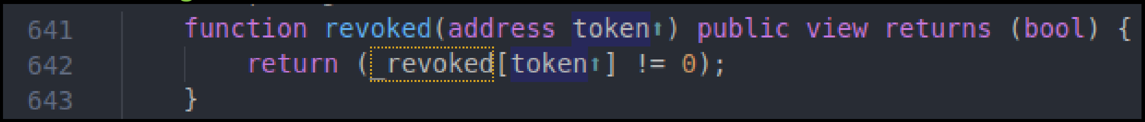
```
    function revocable() public view returns (bool) {
        return _revocable;
    }
```

TokenVesting.sol: Line #634



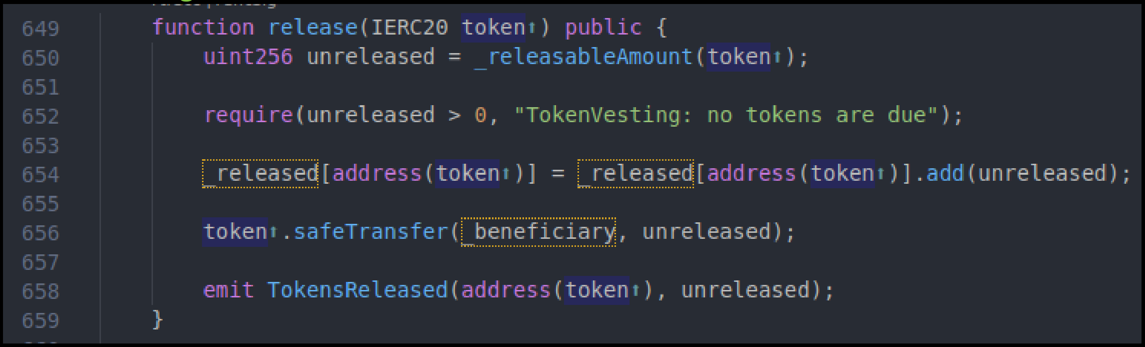
```
    function released(address token) public view returns (uint256) {
        return _released[token];
    }
```

TokenVesting.sol: Line #641



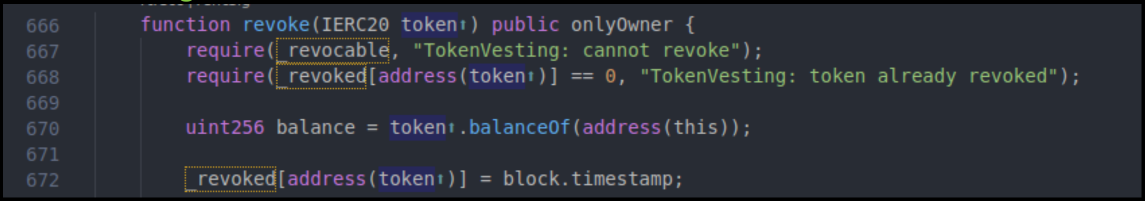
```
    function revoked(address token) public view returns (bool) {
        return (_revoked[token] != 0);
    }
```

TokenVesting.sol: Line #649



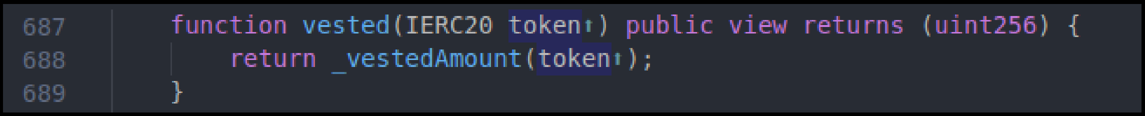
```
    function release(IERC20 token) public {
        uint256 unreleased = _releasableAmount(token);

        require(unreleased > 0, "TokenVesting: no tokens are due");

        _released[address(token)] = _released[address(token)].add(unreleased);

        token.safeTransfer(_beneficiary, unreleased);

        emit TokensReleased(address(token), unreleased);
    }
```

TokenVesting.sol: Line #666



```
    function revoke(IERC20 token) public onlyOwner {
        require(_revocable, "TokenVesting: cannot revoke");
        require(_revoked[address(token)] == 0, "TokenVesting: token already revoked");

        uint256 balance = token.balanceOf(address(this));

        _revoked[address(token)] = block.timestamp;
```

TokenVesting.sol: Line #687



```
    function vested(IERC20 token) public view returns (uint256) {
        return _vestedAmount(token);
    }
```

Risk Level:

Likelihood - 1

Impact - 1

Recommendations:

Consider as much as possible declaring external variables instead of public variables. As for best practice, 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, all can access to public functions while external functions only can be accessed externally.

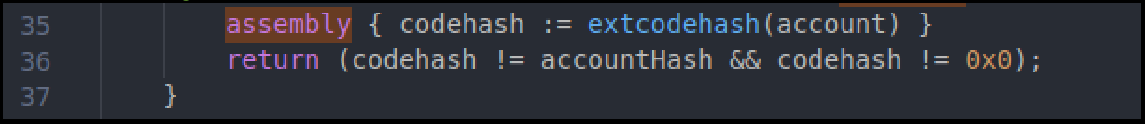
3.6 (HAL-06) USE OF INLINE ASSEMBLY - INFORMATIONAL

Description:

Inline assembly is a way to access the Ethereum Virtual Machine at a low level. This discards several important safety features of Solidity, and the static compiler. Due to the fact that the EVM is a stack machine, it is often hard to address the correct stack slot and provide arguments to opcodes at the correct point on the stack. Solidity's inline assembly tries to facilitate that and other issues arising when writing manual assembly. Assembly is much more difficult to write because the compiler does not perform checks, so the developer of the contract should be aware of this warning.

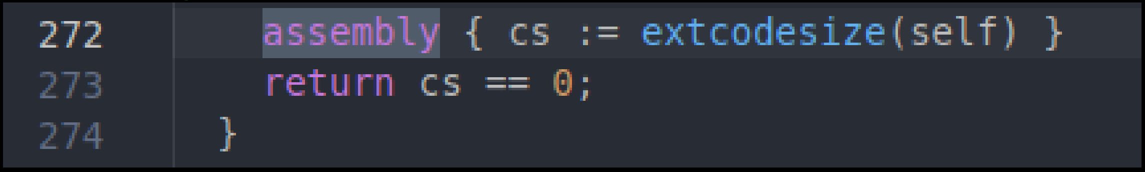
Code Location:

TokenVesting.sol: Line #35



```
35          assembly { codehash := extcodehash(account) }
36          return (codehash != accountHash && codehash != 0x0);
37      }
```

TokenVesting.sol: Line #272



```
272         assembly { cs := extcodesize(self) }
273         return cs == 0;
274     }
```

Risk Level:

Likelihood - 1
Impact - 2

Recommendations:

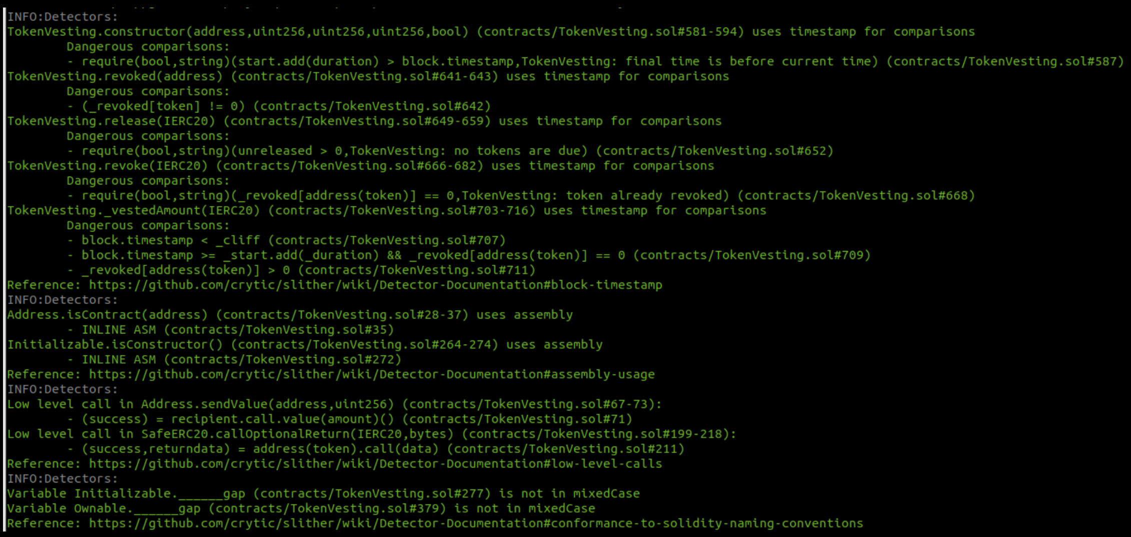
When possible, do not use inline assembly because it is a manner to access to the EVM (Ethereum Virtual Machine) at a low level. An attacker could bypass many important safety features of Solidity. In that case, TokenVesting.sol is a flattened contract and includes some code from OpenZeppelin libraries in the contract code. In order to avoid the use of assembly in the contract, consider to import the OpenZeppelin libraries instead of using the flattened contract.

3.7 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. This tool can statically verify mathematical relationships between Solidity variables to detect invalid or inconsistent usage of the contracts' APIs across the entire code-base.

Results:



```
INFO:Detectors:
TokenVesting.constructor(address,uint256,uint256,uint256,bool) (contracts/TokenVesting.sol#581-594) uses timestamp for comparisons
        Dangerous comparisons:
        - require(bool,string)(start.add(duration) > block.timestamp,TokenVesting: final time is before current time) (contracts/TokenVesting.sol#587)
TokenVesting.revoked(address) (contracts/TokenVesting.sol#641-643) uses timestamp for comparisons
        Dangerous comparisons:
        - (_revoked[token] != 0) (contracts/TokenVesting.sol#642)
TokenVesting.release(IERC20) (contracts/TokenVesting.sol#649-659) uses timestamp for comparisons
        Dangerous comparisons:
        - require(bool,string)(unreleased > 0,TokenVesting: no tokens are due) (contracts/TokenVesting.sol#652)
TokenVesting.revoke(IERC20) (contracts/TokenVesting.sol#666-682) uses timestamp for comparisons
        Dangerous comparisons:
        - require(bool,string)(_revoked[address(token)] == 0,TokenVesting: token already revoked) (contracts/TokenVesting.sol#668)
TokenVesting._vestedAmount(IERC20) (contracts/TokenVesting.sol#703-716) uses timestamp for comparisons
        Dangerous comparisons:
        - block.timestamp < _cliff (contracts/TokenVesting.sol#707)
        - block.timestamp >= _start.add(_duration) && _revoked[address(token)] == 0 (contracts/TokenVesting.sol#709)
        - _revoked[address(token)] > 0 (contracts/TokenVesting.sol#711)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#block-timestamp
INFO:Detectors:
Address.isContract(address) (contracts/TokenVesting.sol#28-37) uses assembly
        - INLINE ASM (contracts/TokenVesting.sol#35)
Initializable.isConstructor() (contracts/TokenVesting.sol#264-274) uses assembly
        - INLINE ASM (contracts/TokenVesting.sol#272)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#assembly-usage
INFO:Detectors:
Low level call in Address.sendValue(address,uint256) (contracts/TokenVesting.sol#67-73):
        - (success) = recipient.call.value(amount)() (contracts/TokenVesting.sol#71)
Low level call in SafeERC20.callOptionalReturn(IERC20,bytes) (contracts/TokenVesting.sol#199-218):
        - (success,returndata) = address(token).call(data) (contracts/TokenVesting.sol#211)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#low-level-calls
INFO:Detectors:
Variable Initializable.______gap (contracts/TokenVesting.sol#277) is not in mixedCase
Variable Ownable.______gap (contracts/TokenVesting.sol#379) is not in mixedCase
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#conformance-to-solidity-naming-conventions
```

Slither found minor issues which were already detected by auditor.

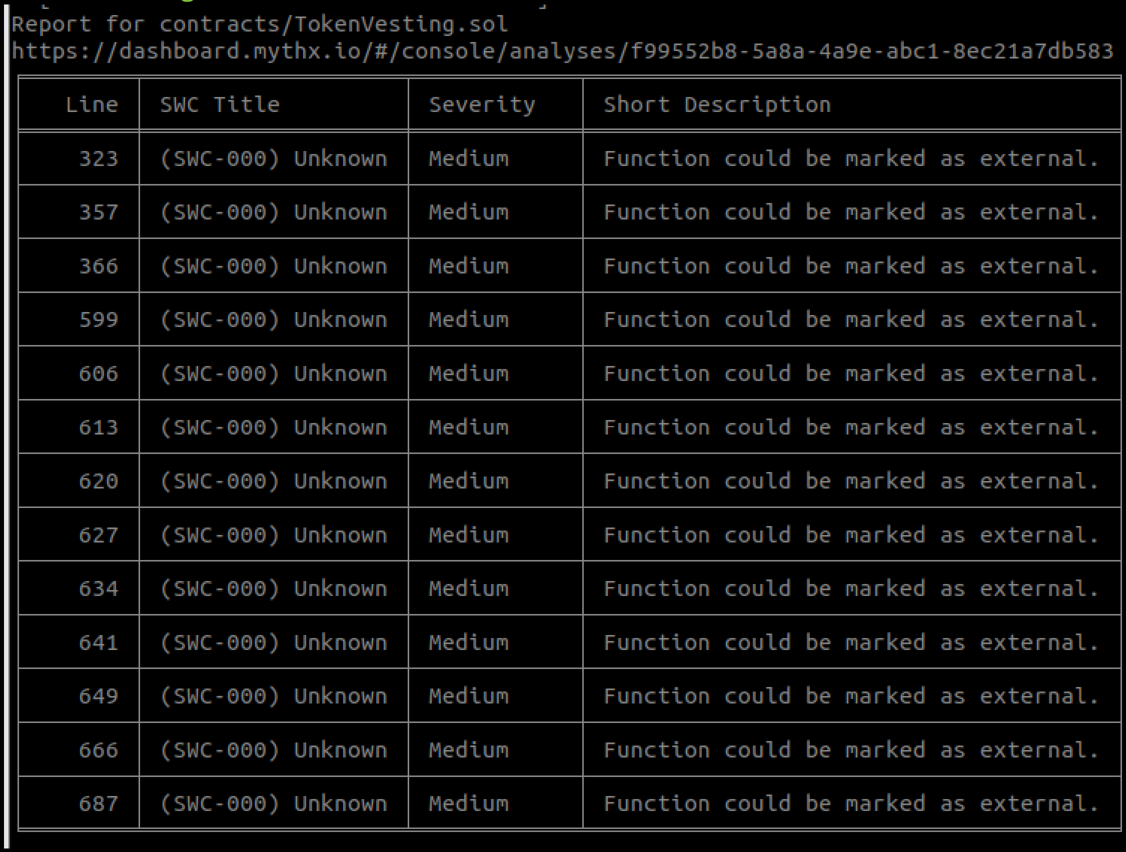
3.8 AUTOMATED SECURITY SCAN

Description:

Halborn used automated security scanners to assist with detection of well-known security issues, and to identify low-hanging fruit 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 testers machine and sent the compiled results to the analyzers to locate any vulnerabilities. In addition, security detections are only in scope.

Results:

TokenVesting.sol



```
Report for contracts/TokenVesting.sol
https://dashboard.mythx.io/#/console/analyses/f99552b8-5a8a-4a9e-abc1-8ec21a7db583
╒════════╤═══════════════════════╤════════════╤═══════════════════════════════════════╕
│   Line │ SWC Title             │ Severity   │ Short Description                     │
╞════════╪═══════════════════════╪════════════╪═══════════════════════════════════════╡
│    323 │ (SWC-000) Unknown     │ Medium     │ Function could be marked as external. │
├────────┼───────────────────────┼────────────┼───────────────────────────────────────┤
│    357 │ (SWC-000) Unknown     │ Medium     │ Function could be marked as external. │
├────────┼───────────────────────┼────────────┼───────────────────────────────────────┤
│    366 │ (SWC-000) Unknown     │ Medium     │ Function could be marked as external. │
├────────┼───────────────────────┼────────────┼───────────────────────────────────────┤
│    599 │ (SWC-000) Unknown     │ Medium     │ Function could be marked as external. │
├────────┼───────────────────────┼────────────┼───────────────────────────────────────┤
│    606 │ (SWC-000) Unknown     │ Medium     │ Function could be marked as external. │
├────────┼───────────────────────┼────────────┼───────────────────────────────────────┤
│    613 │ (SWC-000) Unknown     │ Medium     │ Function could be marked as external. │
├────────┼───────────────────────┼────────────┼───────────────────────────────────────┤
│    620 │ (SWC-000) Unknown     │ Medium     │ Function could be marked as external. │
├────────┼───────────────────────┼────────────┼───────────────────────────────────────┤
│    627 │ (SWC-000) Unknown     │ Medium     │ Function could be marked as external. │
├────────┼───────────────────────┼────────────┼───────────────────────────────────────┤
│    634 │ (SWC-000) Unknown     │ Medium     │ Function could be marked as external. │
├────────┼───────────────────────┼────────────┼───────────────────────────────────────┤
│    641 │ (SWC-000) Unknown     │ Medium     │ Function could be marked as external. │
├────────┼───────────────────────┼────────────┼───────────────────────────────────────┤
│    649 │ (SWC-000) Unknown     │ Medium     │ Function could be marked as external. │
├────────┼───────────────────────┼────────────┼───────────────────────────────────────┤
│    666 │ (SWC-000) Unknown     │ Medium     │ Function could be marked as external. │
├────────┼───────────────────────┼────────────┼───────────────────────────────────────┤
│    687 │ (SWC-000) Unknown     │ Medium     │ Function could be marked as external. │
╘════════╧═══════════════════════╧════════════╧═══════════════════════════════════════╛
```

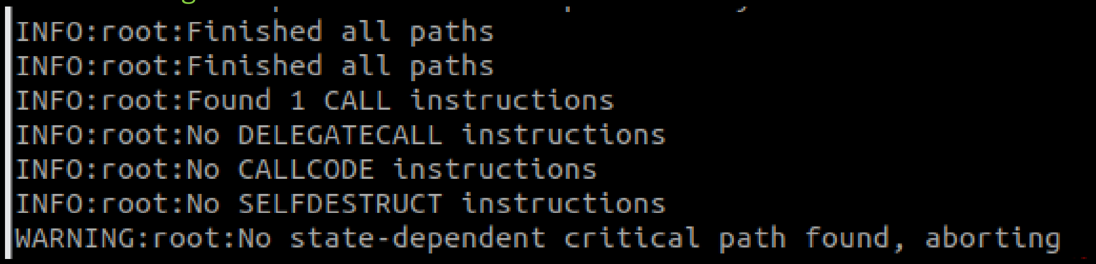
3.9 SECURITY TESTING EXPLOITATION

Description:

teEther is a tool to perform analysis and automatic exploitation over smart contracts. teEther try to exploit the most common vulnerabilities in the bytecode level.

Results:

TokenVesting.sol



```
INFO:root:Finished all paths
INFO:root:Finished all paths
INFO:root:Found 1 CALL instructions
INFO:root:No DELEGATECALL instructions
INFO:root:No CALLCODE instructions
INFO:root:No SELFDESTRUCT instructions
WARNING:root:No state-dependent critical path found, aborting
```

TeEther found a low-level call which was already detected in manual code review.

THANK YOU FOR CHOOSING

// HALBORN