// HALBORN

EasyFi Access Vesting Contract

Smart Contract Security Audit

Prepared by: **Halborn**

Date of Engagement: **June 10th - 17th 2021**

Visit: **Halborn.com**

DOCUMENT REVISION HISTORY

VERSION	MODIFICATION	DATE	AUTHOR
0.1	Document Creation	06/10/2021	Khaled Sakr
0.2	Document Edits	05/13/2021	Khaled Sakr
1.0	Final Draft	05/17/2021	Gabi Urrutia
1.1	Remediation Plan	07/01/2021	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
Khaled Sakr	Halborn	Khaled.Sakr@halborn.com

EXECUTIVE OVERVIEW

1.1 INTRODUCTION

AccessVesting is a token vesting implementation done by EasyFi team to lock tokens inside a smart contract for specific time, until released again by it's beneficiary when cliff period is reached.

EasyFi engaged Halborn to conduct a security audit on their accessVesting smart contract beginning on June 10th, 2021 and ending June 17th, 2021. The security assessment was scoped to the smart contract provided in the Github repository EasyFi Access Smart Contract.

Though this security audit's outcome is satisfactory, only the most essential aspects were tested and verified to achieve objectives and deliverable set in the scope due to time and resource constraints. It is essential to note the use of the best practices for secure smart-contract development.

1.2 AUDIT SUMMARY

The team at Halborn was provided one week for the engagement and assigned a full time security engineer to audit the security of the smart contract. The security engineer is 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 to achieve the following:

- Ensure that smart contract functions are intended.
- Identify potential security issues with the smart contracts.

In summary, Halborn identified few 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.
- Static Analysis of security for scoped contract, and imported functions.(Slither)
- 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.

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 access/accessVesting.sol smart contract.

Specific commit of contract:

6d3548851c6499c2a1ea12f8a7393b0b4f34304d

Fixed commit:
b8c142e343352b86074abd3428fdd3328ec01138

OUT-OF-SCOPE: External libraries and economics attacks

2. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	0	1	3	4

LIKELIHOOD

IMPACT

(HAL-02) (HAL-03)				
(HAL-07)	(HAL-04)		(HAL-01)	
(HAL-05) (HAL-06) (HAL-08)				

EXECUTIVE OVERVIEW

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
TOKENVESTING CONTRACT OUTDATED	Medium	SOLVED - 07/01/2021
PRAGMA VERSION DEPRECATED	Low	SOLVED - 07/01/2021
FLOATING PRAGMA AND VERSION MISMATCH	Low	SOLVED - 07/01/2021
USE OF BLOCK.TIMESTAMP	Low	NOT APPLICABLE
POSSIBLE MISUSE OF RELEASE FUNCTION	Low	ACKNOWLEDGED
MISSING RE-ENTRANCY PROTECTION	Informational	ACKNOWLEDGED
NO TEST COVERAGE	Informational	SOLVED 06/24/2021
DOCUMENTATION	Informational	SOLVED 06/24/2021

FINDINGS & TECH DETAILS

3.1 (HAL-01) TOKENVESTING CONTRACT OUTDATED - MEDIUM

Description:

TokenVesting is used by Stater team with pragma 0.5.17. However, the contract was removed from OpenZeppelin contracts package. In addition, the latest version of the contract before being removed uses the pragma version 0.6.0.

References:
Vesting Discussion
Last TokenVesting version

Code Location:

accessVesting.sol: Lines #204 #362

Listing 1: accessVesting.sol (Lines 204)

```
202 // File: browser/TokenVesting.sol
203
204 pragma solidity 0.5.17;
205
206
```

Listing 2: accessVesting.sol (Lines 362)

```
360 // File: browser/VestingFactory.sol
361
362 pragma solidity 0.5.17;
363
364
```

Risk Level:

Likelihood - 4
Impact - 2

Recommendation:

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:
(eip-1132)[https://eips.ethereum.org/EIPS/eip-1132]
(Time Locked WalletTruffle Tutorial)[https://www.toptal.com/ethereum-smart-contract/time-locked-wallettruffle-tutorial]

Remediation Plan:

SOLVED: EasyFi team updated the pragma to the version 0.7.6.

Listing 3: accessVesting.sol (Lines 3)

```
1 // SPDX-License-Identifier: UNLICENSED
2
3 pragma solidity 0.7.6;
4
5 /**
6
```

3.2 (HAL-02) PRAGMA VERSION DEPRECATED - LOW

Description:

The current version in use for the contracts is pragma =0.5.17 and 0.5.2 for accessVesting.sol. 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:

Listing 4: accessVesting.sol (Lines 80)

```
76 // File: browser/Address.sol
77
78 // File: openzeppelin-solidity/contracts/utils/Address.sol
79
80 pragma solidity ^0.5.2;
81
82 /**
83  * Utility library of inline functions on addresses
84  */
```

Listing 5: accessVesting.sol (Lines 204)

```
202 // File: browser/TokenVesting.sol
203
204 pragma solidity 0.5.17;
205
206
207
208
209
210 /**
211  * @title TokenVesting
```

```
212  * @dev A token holder contract that can release its token balance
         gradually like a
213  * typical vesting scheme, with a cliff and vesting period.
       Optionally revocable by the
214  * owner.
215  */
216 contract TokenVesting {
```

Risk Level:

Likelihood - 1
Impact - 3

Recommendations:

At the time of this audit, the current version is already at 0.8.6. 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 (0.6.12-0.7.6).

Remediation Plan:

SOLVED: EasyFi team updated the pragma to the version 0.7.6.

Listing 6: accessVesting.sol (Lines 3)

```
1 // SPDX-License-Identifier: UNLICENSED
2
3 pragma solidity 0.7.6;
4
5 /**
6
```

3.3 (HAL-03) FLOATING PRAGMA AND VERSION MISMATCH - LOW

Description:

Smart contract accessVesting.sol uses the floating pragma ^0.5.2 in some of the libraries inside, while using floating pragma ^0.5.17 in other libraries inside the smart contract, Contracts should be deployed with the same compiler version and flags used during development and testing. Locking the **pragma** helps to ensure that contracts do not accidentally get deployed using another pragma. For example, an outdated pragma version might introduce bugs that affect the contract system negatively or recently released pragma versions may have unknown security vulnerabilities.

Code Location:

Listing 7: version 0.5.2 (Lines 80)

```
76 // File: browser/Address.sol
77
78 // File: openzeppelin-solidity/contracts/utils/Address.sol
79
80 pragma solidity ^0.5.2;
81
82 /**
83  * Utility library of inline functions on addresses
84  */
```

Listing 8: version 0.5.17 (Lines 204)

```
202 // File: browser/TokenVesting.sol
203
204 pragma solidity 0.5.17;
```

Risk Level:

Likelihood - 1
Impact - 3

Recommendations:

Consider locking the pragma version. It is not recommended to use a floating pragma in production. Apart from just locking the pragma version in the code, the sign (>=) need to be removed. It is possible to lock the pragma by fixing the version both in truffle-config.js for Truffle framework or in hardhat.config.js for HardHat framework.

Remediation Plan:

SOLVED: EasyFi team locked the pragma to the version 0.7.6.

Listing 9: accessVesting.sol (Lines 3)

```
1 // SPDX-License-Identifier: UNLICENSED
2
3 pragma solidity 0.7.6;
4
5 /**
6
```

3.4 (HAL-04) USE OF BLOCK.TIMESTAMP - LOW

Description:

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

Code Location:

accessVesting.sol Lines #253 #351 #353 #356 #416 #417 #424

```
249     constructor (address beneficiary, uint256 amount, uint256 start, uint256 cliffDuration,
250         require(beneficiary != address(0));
251         require(cliffDuration <= duration);
252         require(duration > 0);
253         require(start.add(duration) > block.timestamp);
254
```

```
351         if (block.timestamp < _cliff) {
352             return 0;
353         } else if (block.timestamp >= _start.add(_duration)) {
354             return totalVestingAmount;
355         } else {
356             return totalVestingAmount.mul(block.timestamp.sub(_start)).div(_duration);
```

```
416             if (cliffTime > block.timestamp) {
417                 cliff = cliffTime.sub(block.timestamp);
418             }
419
420
421             TokenVesting vesting = new TokenVesting(
422                 msg.sender,
423                 amount,
424                 block.timestamp,
```

Risk Level:

Likelihood - 2
Impact - 2

Recommendation:

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

Remediation Plan:

NOT APPLICABLE: EasyFi team assumes the use of block.timestamp is safe since the timescales is higher than 900 seconds.

3.5 (HAL-05) POSSIBLE MISUSE OF RELEASE FUNCTION - INFORMATIONAL

Description:

When a beneficiary wants to vest tokens, the smart contract creates another contract which is used to hold the vested token, whenever a beneficiary wants to release this tokens given specific conditions are met , he can call the release() function to transfer the tokens back to the beneficiary.
It was observed that the release function is public and there is no checks that the function caller is the actual beneficiary or not, there are no loss of funds as the tokens are sent back to the beneficiary and not the user who called the function (msg.sender)

Code Location:

Listing 10: accessVesting.sol (Lines 319)

```
316  /**
317      * @notice Transfers vested tokens to beneficiary.
318      */
319     function release() public {
320         uint256 unreleased = _releasableAmount();
321
322
323         require(unreleased > 0);
324
325         uint256 sTokensToRelease = unreleased.mul(multiplier).div
                (10000);
326
327         _released[mainToken] = _released[mainToken].add(unreleased
                );
328         _released[secondaryToken] = _released[secondaryToken].add(
                sTokensToRelease);
329
330
331         IERC20(mainToken).safeTransfer(_beneficiary, unreleased);
```

```
332         IERC20(secondaryToken).safeTransferFrom(factory,
                _beneficiary, sTokensToRelease);
333
334         emit TokensReleased(unreleased, sTokensToRelease);
335     }
336
```

Risk Level:

Likelihood - 1
Impact - 1

Recommendations:

Consider adding onlyBeneficiary which contains below sample code.

Listing 11: onlyBeneficiary Modifier (Lines 2)

```
1   modifier onlyBeneficiary() {
2     require(msg.sender == beneficiary)
3     _
4   }
```

Remediation Plan:

ACKNOWLEDGED: EasyFi considers that this is intended behavior to be able to allow beneficiary to release their tokens even from another address, at the end the tokens are only released to the beneficiary address.

3.6 (HAL-06) MISSING RE-ENTRANCY PROTECTION - INFORMATIONAL

Description:

To protect against cross-function reentrancy attacks, it may be necessary to use a mutex. By using this lock, an attacker can no longer exploit the withdraw function with a recursive call. OpenZeppelin has it's own mutex implementation called ReentrancyGuard which provides a modifier to any function called nonReentrant that guards the function with a mutex against reentrancy attacks.

Risk Level:

Likelihood - 1
Impact - 1

Recommendation:

In the accessVesting.sol contract, function like release(), is missing nonReentrant guard. Use the nonReentrant modifier to avoid introducing future vulnerabilities.

Remediation Plan:

ACKNOWLEDGED: EasyFi team accepts the risk and nonReentrant modifier will not be added due to the fact that all state changes are happening before the transfer calls

3.7 (HAL-07) NO TEST COVERAGE - INFORMATIONAL

Description:

Unlike traditional software, smart contracts can not be modified unless deployed using a proxy contract. Because of the permanence, unit tests and functional testing are recommended to ensure the code works correctly before deployment. Mocha and Chai are valuable tools to perform unit tests in smart contracts. Mocha is a Javascript testing framework for creating synchronous and asynchronous unit tests, and Chai is a library with assertion functionality such as assert or expect and should be used to develop custom unit tests.

References:
https://github.com/mochajs/mocha
https://github.com/chaijs/chai
https://docs.openzeppelin.com/learn/writing-automated-tests

Risk Level:

Likelihood - 1
Impact - 2

Recommendation:

We recommend performing as many test cases as possible to cover all conceivable scenarios in the smart contract.

Remediation Plan:

SOLVED: Unit tests were added to the test directory including many test cases such as:
- makings transfers exceeding the current balance
- releasing tokens before the end of vesting period.

- Tesitng the vest() function which creates another contract of type tokenVesting.

Listing 12: Sample Unit Test Added (Lines 42,43,44)

```
28 it("Expect Revert - Vest", async function () {
29         await expect(
30             access.connect(addr2).vest(1000)
31         ).to.be.revertedWith("ERC20: transfer amount exceeds
               balance");
32
33         await network.provider.send("evm_increaseTime", [5184001])
34         await network.provider.send("evm_mine")
35
36         await expect(
37             access.connect(addr2).vest(1000)
38         ).to.be.revertedWith("release time is before current time"
               );
39
40         let maxVesting = await access.maxVesting();
41         maxVesting = maxVesting + 1;
42         await expect(
43             access.connect(addr2).vest(maxVesting)
44         ).to.be.revertedWith("Breaching max vesting limit");
45     })
46
```

3.8 (HAL-08) DOCUMENTATION - INFORMATIONAL

Description:

The documentation provided by the EasyFi team is not complete. For instance, the documentation included in the GitHub repository should include a walkthrough to deploy and test the smart contracts.

Recommendation:

Consider updating the documentation in Github for greater ease when contracts are deployed and tested. Have a Non-Developer or QA resource work through the process to make sure it addresses any gaps in the set-up steps due to technical assumptions.

Remediation Plan:

SOLVED: Documentation has been added to the README.md

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. 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,uint256,address,address,uint256).beneficiary (contracts/access/accessVesting.sol#71) shadows:
        - TokenVesting.beneficiary() (contracts/access/accessVesting.sol#91-93) (function)
TokenVesting.constructor(address,uint256,uint256,uint256,uint256,address,address,uint256).start (contracts/access/accessVesting.sol#71) shadows:
        - TokenVesting.start() (contracts/access/accessVesting.sol#105-107) (function)
TokenVesting.constructor(address,uint256,uint256,uint256,uint256,address,address,uint256).duration (contracts/access/accessVesting.sol#71) shadows:
        - TokenVesting.duration() (contracts/access/accessVesting.sol#112-114) (function)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#local-variable-shadowing
INFO:Detectors:
VestingFactory.accruedAmount(address) (contracts/access/accessVesting.sol#266-272) has external calls inside a loop: amount = amount.add(TokenVesting(userVsVesting[user][i]).accruedAmount()) (contracts/access/accessVesting.sol#269)
VestingFactory.mainTokenBalance(address) (contracts/access/accessVesting.sol#274-280) has external calls inside a loop: amount = amount.add(TokenVesting(userVsVesting[user][i]).available()) (contracts/access/accessVesting.sol#277)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation/#calls-inside-a-loop
INFO:Detectors:
Reentrancy in TokenVesting.release() (contracts/access/accessVesting.sol#141-157):
        External calls:
        - IERC20(mainToken).safeTransfer(_beneficiary,unreleased) (contracts/access/accessVesting.sol#153)
        - IERC20(secondaryToken).safeTransferFrom(factory,_beneficiary,sTokensToRelease) (contracts/access/accessVesting.sol#154)
        Event emitted after the call(s):
        - TokensReleased(unreleased,sTokensToRelease) (contracts/access/accessVesting.sol#156)
Reentrancy in VestingFactory.vest(uint256) (contracts/access/accessVesting.sol#232-260):
        External calls:
        - IERC20(mainToken).safeTransferFrom(msg.sender,address(vesting),amount) (contracts/access/accessVesting.sol#255)
        - IERC20(secondaryToken).safeApprove(address(vesting),amount.mul(multipier).div(10000)) (contracts/access/accessVesting.sol#256)
        Event emitted after the call(s):
        - Vested(msg.sender,address(vesting)) (contracts/access/accessVesting.sol#258)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-3
INFO:Detectors:
TokenVesting.constructor(address,uint256,uint256,uint256,uint256,address,address,uint256) (contracts/access/accessVesting.sol#71-86) uses timestamp for comparisons
        Dangerous comparisons:
        - require(bool)(start.add(duration) > block.timestamp) (contracts/access/accessVesting.sol#75)
TokenVesting.release() (contracts/access/accessVesting.sol#141-157) uses timestamp for comparisons
        Dangerous comparisons:
        - require(bool)(unreleased > 0) (contracts/access/accessVesting.sol#145)
TokenVesting._vestedAmount() (contracts/access/accessVesting.sol#170-180) uses timestamp for comparisons
        Dangerous comparisons:
        - block.timestamp < _cliff (contracts/access/accessVesting.sol#173)
        - block.timestamp >= _start.add(_duration) (contracts/access/accessVesting.sol#175)
VestingFactory.vest(uint256) (contracts/access/accessVesting.sol#232-260) uses timestamp for comparisons
        Dangerous comparisons:
        - cliffTime > block.timestamp (contracts/access/accessVesting.sol#238)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#block-timestamp
INFO:Detectors:
Address.isContract(address) (contracts/access/Address.sol#20-31) uses assembly
        - INLINE ASM (contracts/access/Address.sol#29)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#assembly-usage
INFO:Detectors:
Different versions of Solidity is used in :
        - Version used: ['0.5.17', '^0.5.2']
        - ^0.5.2 (contracts/access/Address.sol#1)
        - ^0.5.2 (contracts/access/IERC20.sol#2)
        - ^0.5.2 (contracts/access/SafeERC20.sol#2)
        - ^0.5.2 (contracts/access/SafeMath.sol#1)
        - 0.5.17 (contracts/access/accessVesting.sol#2)
        - 0.5.17 (contracts/access/accessVesting.sol#184)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#different-pragma-directives-are-used
```

AUTOMATED TESTING

```
INFO:Detectors:
Pragma version^0.5.2 (contracts/access/Address.sol#1) allows old versions
Pragma version^0.5.2 (contracts/access/IERC20.sol#2) allows old versions
Pragma version^0.5.2 (contracts/access/SafeERC20.sol#2) allows old versions
Pragma version^0.5.2 (contracts/access/SafeMath.sol#1) allows old versions
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity
INFO:Detectors:
Low level call in SafeERC20.callOptionalReturn(IERC20,bytes) (contracts/access/SafeERC20.sol#58-76):
        - (success,returndata) = address(token).call(data) (contracts/access/SafeERC20.sol#70)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#low-level-calls
INFO:Detectors:
Function TokenVesting._accruedAmount() (contracts/access/accessVesting.sol#133-135) is not in mixedCase
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#conformity-to-solidity-naming-conventions
INFO:Detectors:
beneficiary() should be declared external:
        - TokenVesting.beneficiary() (contracts/access/accessVesting.sol#91-93)
cliff() should be declared external:
        - TokenVesting.cliff() (contracts/access/accessVesting.sol#98-100)
start() should be declared external:
        - TokenVesting.start() (contracts/access/accessVesting.sol#105-107)
duration() should be declared external:
        - TokenVesting.duration() (contracts/access/accessVesting.sol#112-114)
available() should be declared external:
        - TokenVesting.available() (contracts/access/accessVesting.sol#119-121)
released(address) should be declared external:
        - TokenVesting.released(address) (contracts/access/accessVesting.sol#126-128)
_accruedAmount() should be declared external:
        - TokenVesting._accruedAmount() (contracts/access/accessVesting.sol#133-135)
release() should be declared external:
        - TokenVesting.release() (contracts/access/accessVesting.sol#141-157)
userContracts(address) should be declared external:
        - VestingFactory.userContracts(address) (contracts/access/accessVesting.sol#262-264)
accruedAmount(address) should be declared external:
        - VestingFactory.accruedAmount(address) (contracts/access/accessVesting.sol#266-272)
mainTokenBalance(address) should be declared external:
        - VestingFactory.mainTokenBalance(address) (contracts/access/accessVesting.sol#274-280)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#public-function-that-could-be-declared-external
INFO:Slither:contracts/access/accessVesting.sol analyzed (6 contracts with 46 detectors), 30 result(s) found
```

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 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. Only security-related findings are shown below.

Results:

Report for accessVesting.sol
https://dashboard.mythx.io/#/console/analyses/89d25bb5-15da-411b-aa5e-6980ce06c950

Line	SWC Title	Severity	Short Description
91	(SWC-000) Unknown	Medium	Function could be marked as external.
98	(SWC-000) Unknown	Medium	Function could be marked as external.
105	(SWC-000) Unknown	Medium	Function could be marked as external.
112	(SWC-000) Unknown	Medium	Function could be marked as external.
119	(SWC-000) Unknown	Medium	Function could be marked as external.
126	(SWC-000) Unknown	Medium	Function could be marked as external.
133	(SWC-000) Unknown	Medium	Function could be marked as external.
141	(SWC-000) Unknown	Medium	Function could be marked as external.
243	(SWC-123) Requirement Violation	Low	Requirement violation.
262	(SWC-000) Unknown	Medium	Function could be marked as external.
263	(SWC-128) DoS With Block Gas Limit	Low	Implicit loop over unbounded data structure.
266	(SWC-000) Unknown	Medium	Function could be marked as external.
268	(SWC-128) DoS With Block Gas Limit	Low	Loop over unbounded data structure.
274	(SWC-000) Unknown	Medium	Function could be marked as external.
276	(SWC-128) DoS With Block Gas Limit	Low	Loop over unbounded data structure.

THANK YOU FOR CHOOSING

// HALBORN