Pangolin - AllocationVester Contract

Smart Contract Security Audit

Prepared by: **Halborn**

Date of Engagement: **March 20th, 2022 - March 30th, 2022**

Visit: **Halborn.com**

DOCUMENT REVISION HISTORY

VERSION	MODIFICATION	DATE	AUTHOR
0.1	Document Creation	03/24/2022	Francisco González
0.2	Document Updates	03/28/2022	Francisco González
0.3	Document Updates	03/28/2022	István Böhm
0.4	Draft Review	03/28/2022	Gabi Urrutia
1.0	Remediation Plan	03/29/2022	Francisco González
1.1	Remediation Plan Review	03/30/2022	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
István Böhm	Halborn	Istvan.Bohm@halborn.com
Francisco González	Halborn	Francisco.Villarejo@halborn.com

EXECUTIVE OVERVIEW

1.1 INTRODUCTION

Pangolin engaged Halborn to conduct a security audit on their AllocationVester smart contract beginning on March 20th, 2022 and ending on March 30th, 2022. The security assessment was scoped to the AllocationVester smart contract provided in the exchange-contracts GitHub repository pangolindex/exchange-contracts.

1.2 AUDIT SUMMARY

The team at Halborn was provided a week for the engagement and assigned two full-time security engineers to audit the security of the smart contract. The security engineers are blockchain and smart-contract security experts 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 all functions in the AllocationVester smart contract are intended.
- Identify potential security issues in the AllocationVester smart contract.

In summary, Halborn identified few security risks that were mostly addressed by the Pangolin team.

1.3 TEST APPROACH & METHODOLOGY

Halborn performed a combination of manual and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of this audit. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of the bridge code and can quickly identify items that do not follow security best practices. The following phases and associated tools were used throughout the term of the audit:

- Research into architecture and purpose
- Smart contract manual code review and walkthrough
- Graphing out functionality and contract logic/connectivity/functions (solgraph)
- Manual assessment of use and safety for the critical Solidity variables and functions in scope to identify any arithmetic related vulnerability classes
- Manual testing by custom scripts
- Scanning of solidity files for vulnerabilities, security hotspots or bugs. (MythX)
- Static Analysis of security for scoped contract, and imported functions. (Slither)
- Testnet deployment (Brownie, Remix IDE)

RISK METHODOLOGY:

Vulnerabilities or issues observed by Halborn are ranked based on the risk assessment methodology by measuring the **LIKELIHOOD** of a security incident and the **IMPACT** should an incident occur. This framework works for communicating the characteristics and impacts of technology vulnerabilities. The quantitative model ensures repeatable and accurate measurement while enabling users to see the underlying vulnerability characteristics that were 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:

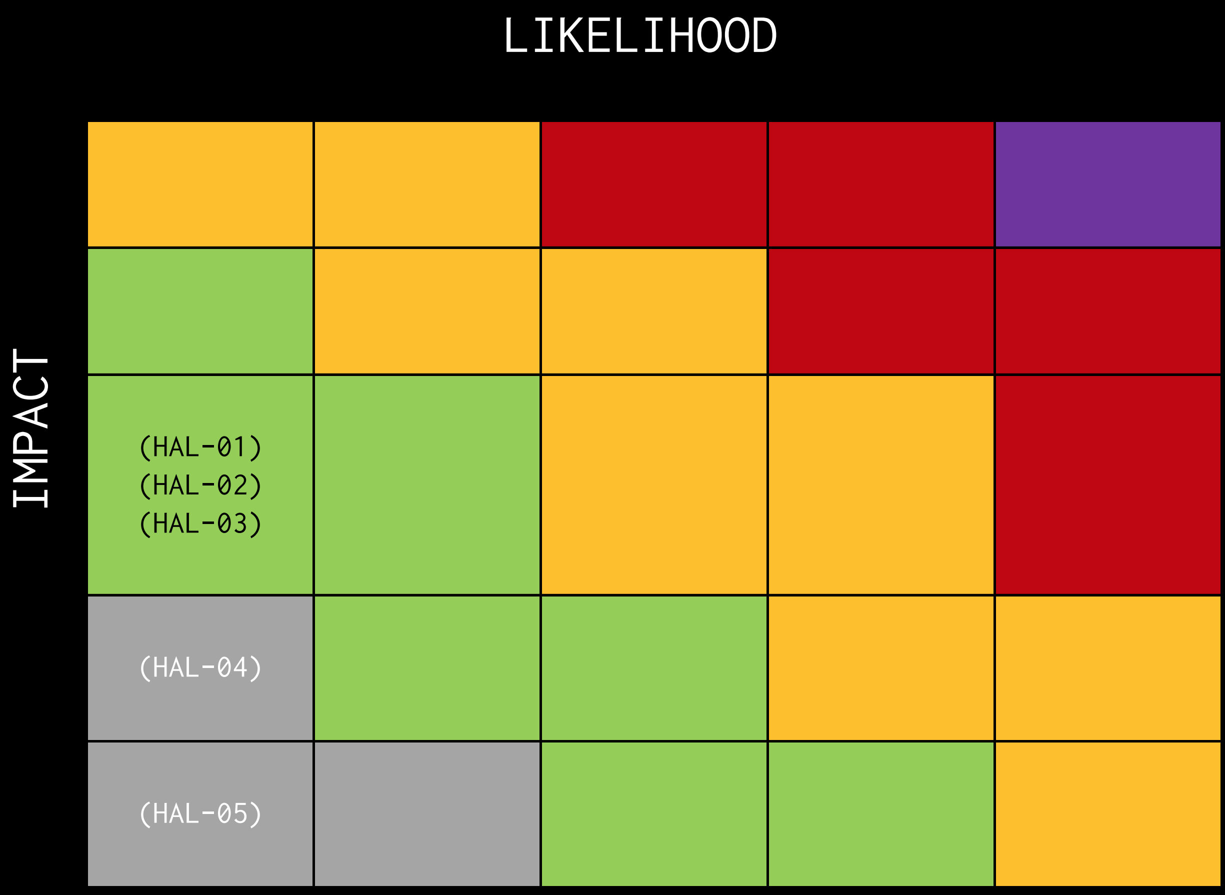
The security assessment was scoped to the following smart contract:

- AllocationVester.sol

Commit ID: 404631bb50e9d04935dd79930e9769ff73764bfd

2. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	0	0	3	2



SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
HAL01 - DOS WITH BLOCK GAS LIMIT	Low	SOLVED - 3/29/2022
HAL02 - INACCURATE REWARD RATE CALCULATION	Low	SOLVED - 3/29/2022
HAL03 - FLOATING PRAGMA	Low	FUTURE RELEASE
HAL04 - MISSING EVENTS EMITTING	Informational	SOLVED - 3/29/2022
HAL05 - MISSING ZERO ADDRESS CHECK	Informational	SOLVED - 3/29/2022

FINDINGS & TECH DETAILS

3.1 (HAL-01) DOS WITH BLOCK GAS LIMIT - LOW

Description:

The setAllocations() function is used to set the token allocations for each recipient included in the function call:

Listing 1: setAllocations() function (Line 118)

```
105 function setAllocations(
106     address[] memory accounts,
107     uint[] memory allocations,
108     uint[] memory durations
109 ) external onlyOwner {
110     uint length = accounts.length;
111     require(length != 0, "empty array");
112     require(
113         length == allocations.length && length == durations.length,
114         "varying-length arrays"
115     );
116 
117     uint balance = token.balanceOf(address(this));
118     for (uint i; i < length; ++i) {
119         address account = accounts[i];
120         uint allocation = allocations[i];
121         uint duration = durations[i];
122         Member storage member = members[account];
123 
124         require(account != address(0), "bad recipient");
125 
126         // check the member's remaining harvest
127         if (member.reserve != 0) {
128             // stash pending rewards of the member so it remains claimable
129             member.stash = pendingHarvest(account);
130             // free non-stashed reserves of the member from the reserves
131             reserve -= (member.reserve - member.stash);
132             // free non-stashed tokens from member's reserves
133             member.reserve = member.stash;
```

```
        }

        // check the member's new allocation
        if (allocation != 0) {
            require(duration >= MIN_DURATION, "short vesting duration");

            // lock tokens as reserve and ensure sufficient balance
            reserve += allocation;
            require(balance >= reserve, "low balance");

            // add vesting info for the member
            member.reserve += allocation;
            member.rate = allocation / duration;
            member.lastUpdate = block.timestamp;

            // add the member to the set
            _membersAddresses.add(account);
        }

        emit AllocationSet(account, allocation, duration);
    }
}
```

Since the length of recipients is not limited, in case there are too many recipients, the block gas limit could be reached, causing miners to not respond to all setAllocations() calls, thus blocking the main purpose of the smart contract.

Risk Level:

Likelihood - 1
Impact - 3

Recommendation:

Consider limiting the number of recipients in the setAllocations() function of the AllocationVester contract with a require statement.

Remediation Plan:

SOLVED: The Pangolin Team solved this issue by implementing the above recommendation. The maximum number of recipients has been set to 40, preventing gas usage from increasing too much.

Commit ID: **9563309e6aaf1d0fc930a94cb0fc903d7017d40a**

3.2 (HAL-02) INACCURATE REWARD RATE CALCULATION - LOW

Description:

AllocationVester.sol contract calculates the reward unlock rate by dividing the total amount of tokens to be unlocked by the total vesting period (in seconds), which can go from eight weeks (4838400 seconds) to infinity. However, when the amount of tokens is a number of the same order as the total vesting period, rounding Solidity to zero will introduce inaccuracies in the result of the uint division:

```
Setting 10 PNG allocation to user1 for one year (31557600 seconds) --> contract_AllocationVester.setAllocations([user1], [10*10**18], [ONE_YEAR], {'from': owner})

Transaction sent: 0xe891b5fb85a563f57826cab3dce2fb02c63a1647f4662aee88e57e9820f40ead
  Gas price: 0.0 gwei   Gas limit: 800000000   Nonce: 3
  AllocationVester.setAllocations confirmed   Block: 14450216   Gas used: 176394 (0.02%)

Token unlock rate per second of user1 -> 3.1688087814e-07 PNG per second

Setting 31557599 tokens allocation to user2 for one year (31557600 seconds) --> contract_AllocationVester.setAllocations([user2], [31557599], [ONE_YEAR], {'from': owner})

Transaction sent: 0x9bc5916a8a49ae6fb1d774fea1d0e627d91febc90cec50e9e0d1640b7e177838
  Gas price: 0.0 gwei   Gas limit: 800000000   Nonce: 4
  AllocationVester.setAllocations confirmed   Block: 14450217   Gas used: 127170 (0.02%)

Token unlock rate per second of user2 -> 0.0 PNG per second
```

The accuracy of the reward unlock rate depends primarily on the ERC20 token to be distributed, which is associated with the AllocationVester contract during its deployment. The use of the token contracts with high decimal values such as Png (18) is considered safe, but the use of contracts with lower decimal values such as USDT (6) may result in an inaccurate rate calculation, which can undermine user trust.

For example, allocating 30 USDT for one year would result in an unlock rate of 0, and it would take 5% more time to unlock 1000 USDT allocated for two years.

However, it has been noted that incorrectly set allocations can be easily overridden by the contract owner, if needed, causing locked funds to be returned to the contract reserve.

Code Location:

Listing 2: Reward rate calculation (Line 146)

```
144 // add vesting info for the member
145 member.reserve += allocation;
146 member.rate = allocation / duration;
147 member.lastUpdate = block.timestamp;
```

Risk Level:

Likelihood - 1
Impact - 3

Recommendation:

Consider checking the decimals of the token to be distributed in the AllocationVester constructor.

Also, before deployment, consider reviewing the use cases of the contract and the associated token to ensure the currently implemented reward calculation algorithm does not have perceptible rounding errors.

Remediation Plan:

SOLVED: The Pangolin Team solved this issue by multiplying the token allocation by a coefficient that will add 11 decimals of precision when calculating the reward unlock rate, which is considered enough for this contract.

Commit ID: **69fa24b3b55c0c7b5bc566354e9b1b36eb3f6272**

3.3 (HAL-03) FLOATING PRAGMA - LOW

Description:

AllocationVester.sol contract uses the floating pragma ^0.8.0. The contract should be deployed with the same compiler version and flags that they have been tested with thoroughly. Locking the pragma helps to ensure that contracts do not accidentally get deployed using, for example, either an outdated compiler version that might introduce bugs that affect the contract system negatively or a pragma version too new which has not been extensively tested.

Code Location:

Listing 3: Floating pragma (Line 3)

```
3 pragma solidity ^0.8.0;
```

Risk Level:

Likelihood - 1
Impact - 3

Recommendation:

Consider locking the pragma version with known bugs for the compiler version. When possible, do not use floating pragma in the final live deployment. Specifying a fixed compiler version ensures that the bytecode produced does not vary between builds. This is especially important if you rely on bytecode-level verification of the code.

Remediation Plan:

PENDING: The Pangolin Team acknowledged this issue and will address it in future releases.

3.4 (HAL-04) MISSING EVENTS EMITTING - INFORMATIONAL

Description:

It has been observed that important functionalities are missing emitting events. The following functions should emit events in the AllocationVester contract:

Fuctions with events missing:

- withdraw()
- harvest()

Risk Level:

Likelihood - 1
Impact - 2

Recommendation:

Consider emitting an event when calling related functions on the list above.

Remediation Plan:

SOLVED: The Pangolin Team solved this issue by defining new events that will be emitted every time a reward pickup or withdrawal occurs.

Commit ID: **d8137ff12bfbf6c083939b8d487e6dd47a70a3ea**

3.5 (HAL-05) MISSING ZERO ADDRESS CHECK - INFORMATIONAL

Description:

The constructor of the AllocationVester contract is missing address validation. The distributionToken parameter should be checked to be non-zero. This is considered a best practice.

Code Location:

Listing 4: Missing zero address check (Line 61)

```
60     constructor(IERC20 distributionToken) {
61         token = distributionToken;
62     }
```

Risk Level:

Likelihood - 1
Impact - 1

Recommendation:

It is recommended to validate that each address inputs in the constructor and other critical functions are non-zero.

Remediation Plan:

SOLVED: The Pangolin Team solved this issue by implementing a zero check address in the contract constructor.

Commit ID: **2a5bb1afe0e1cb0aab67b9f8e75970a3ad75c992**

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.

AUTOMATED TESTING

Slither Results:

```
AllocationVester.harvest() (contracts/allocation-vester/AllocationVester.sol#67-87) uses a dangerous strict equality:
	- member.reserve == 0 (contracts/allocation-vester/AllocationVester.sol#84)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#dangerous-strict-equalities

AllocationVester.setAllocations(address[],uint256[],uint256[]).i (contracts/allocation-vester/AllocationVester.sol#118) is a local variable never initialized
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#uninitialized-local-variables

AllocationVester.harvest() (contracts/allocation-vester/AllocationVester.sol#67-87) ignores return value by _membersAddresses.remove(account) (contracts/allocation-vester/All
ocationVester.sol#84)
AllocationVester.setAllocations(address[],uint256[],uint256[]) (contracts/allocation-vester/AllocationVester.sol#105-155) ignores return value by _membersAddresses.add(accoun
t) (contracts/allocation-vester/AllocationVester.sol#150)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#unused-return

Claimable.transferOwnership(address).newOwner (contracts/Claimable.sol#14) lacks a zero-check on :
		- _pendingOwner = newOwner (contracts/Claimable.sol#15)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#missing-zero-address-validation

AllocationVester.harvest() (contracts/allocation-vester/AllocationVester.sol#67-87) uses timestamp for comparisons
	Dangerous comparisons:
	- require(bool,string)(amount != 0,no pending harvest) (contracts/allocation-vester/AllocationVester.sol#73)
	- member.reserve == 0 (contracts/allocation-vester/AllocationVester.sol#84)
AllocationVester.withdraw(uint256) (contracts/allocation-vester/AllocationVester.sol#93-96) uses timestamp for comparisons
	Dangerous comparisons:
	- require(bool,string)(unreserved() >= amount,low balance) (contracts/allocation-vester/AllocationVester.sol#94)
AllocationVester.setAllocations(address[],uint256[],uint256[]) (contracts/allocation-vester/AllocationVester.sol#105-155) uses timestamp for comparisons
	Dangerous comparisons:
	- require(bool,string)(balance >= reserve,low balance) (contracts/allocation-vester/AllocationVester.sol#142)
AllocationVester.pendingHarvest(address) (contracts/allocation-vester/AllocationVester.sol#173-180) uses timestamp for comparisons
	Dangerous comparisons:
	- amount > member.reserve (contracts/allocation-vester/AllocationVester.sol#179)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#block-timestamp

Address.isContract(address) (node_modules/@openzeppelin/contracts/utils/Address.sol#27-37) uses assembly
	- INLINE ASM (node_modules/@openzeppelin/contracts/utils/Address.sol#33-35)
Address.verifyCallResult(bool,bytes,string) (node_modules/@openzeppelin/contracts/utils/Address.sol#196-216) uses assembly
	- INLINE ASM (node_modules/@openzeppelin/contracts/utils/Address.sol#208-211)
EnumerableSet.values(EnumerableSet.AddressSet) (node_modules/@openzeppelin/contracts/utils/structs/EnumerableSet.sol#274-283) uses assembly
	- INLINE ASM (node_modules/@openzeppelin/contracts/utils/structs/EnumerableSet.sol#278-280)
EnumerableSet.values(EnumerableSet.UintSet) (node_modules/@openzeppelin/contracts/utils/structs/EnumerableSet.sol#347-356) uses assembly
	- INLINE ASM (node_modules/@openzeppelin/contracts/utils/structs/EnumerableSet.sol#351-353)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#assembly-usage

AllocationVester.setAllocations(address[],uint256[],uint256[]) (contracts/allocation-vester/AllocationVester.sol#105-155) has costly operations inside a loop:
	- reserve -= (member.reserve - member.stash) (contracts/allocation-vester/AllocationVester.sol#131)
AllocationVester.setAllocations(address[],uint256[],uint256[]) (contracts/allocation-vester/AllocationVester.sol#105-155) has costly operations inside a loop:
	- reserve += allocation (contracts/allocation-vester/AllocationVester.sol#141)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#costly-operations-inside-a-loop

Address.functionCall(address,bytes) (node_modules/@openzeppelin/contracts/utils/Address.sol#80-82) is never used and should be removed
Address.functionCallWithValue(address,bytes,uint256) (node_modules/@openzeppelin/contracts/utils/Address.sol#109-115) is never used and should be removed
Address.functionDelegateCall(address,bytes) (node_modules/@openzeppelin/contracts/utils/Address.sol#169-171) is never used and should be removed
Address.functionDelegateCall(address,bytes,string) (node_modules/@openzeppelin/contracts/utils/Address.sol#179-188) is never used and should be removed
Address.functionStaticCall(address,bytes) (node_modules/@openzeppelin/contracts/utils/Address.sol#142-144) is never used and should be removed
Address.functionStaticCall(address,bytes,string) (node_modules/@openzeppelin/contracts/utils/Address.sol#152-161) is never used and should be removed
Address.sendValue(address,uint256) (node_modules/@openzeppelin/contracts/utils/Address.sol#55-60) is never used and should be removed
Context._msgData() (node_modules/@openzeppelin/contracts/utils/Context.sol#21-23) is never used and should be removed
EnumerableSet._at(EnumerableSet.Set,uint256) (node_modules/@openzeppelin/contracts/utils/structs/EnumerableSet.sol#130-132) is never used and should be removed
EnumerableSet._length(EnumerableSet.Set) (node_modules/@openzeppelin/contracts/utils/structs/EnumerableSet.sol#116-118) is never used and should be removed
EnumerableSet.add(EnumerableSet.Bytes32Set,bytes32) (node_modules/@openzeppelin/contracts/utils/structs/EnumerableSet.sol#158-160) is never used and should be removed
EnumerableSet.add(EnumerableSet.UintSet,uint256) (node_modules/@openzeppelin/contracts/utils/structs/EnumerableSet.sol#297-299) is never used and should be removed
EnumerableSet.at(EnumerableSet.AddressSet,uint256) (node_modules/@openzeppelin/contracts/utils/structs/EnumerableSet.sol#262-264) is never used and should be removed
EnumerableSet.at(EnumerableSet.Bytes32Set,uint256) (node_modules/@openzeppelin/contracts/utils/structs/EnumerableSet.sol#196-198) is never used and should be removed
EnumerableSet.at(EnumerableSet.UintSet,uint256) (node_modules/@openzeppelin/contracts/utils/structs/EnumerableSet.sol#335-337) is never used and should be removed
EnumerableSet.contains(EnumerableSet.AddressSet,address) (node_modules/@openzeppelin/contracts/utils/structs/EnumerableSet.sol#241-243) is never used and should be removed
EnumerableSet.contains(EnumerableSet.Bytes32Set,bytes32) (node_modules/@openzeppelin/contracts/utils/structs/EnumerableSet.sol#175-177) is never used and should be removed
EnumerableSet.contains(EnumerableSet.UintSet,uint256) (node_modules/@openzeppelin/contracts/utils/structs/EnumerableSet.sol#314-316) is never used and should be removed
EnumerableSet.length(EnumerableSet.AddressSet) (node_modules/@openzeppelin/contracts/utils/structs/EnumerableSet.sol#248-250) is never used and should be removed
EnumerableSet.length(EnumerableSet.Bytes32Set) (node_modules/@openzeppelin/contracts/utils/structs/EnumerableSet.sol#182-184) is never used and should be removed
EnumerableSet.length(EnumerableSet.UintSet) (node_modules/@openzeppelin/contracts/utils/structs/EnumerableSet.sol#321-323) is never used and should be removed
EnumerableSet.remove(EnumerableSet.Bytes32Set,bytes32) (node_modules/@openzeppelin/contracts/utils/structs/EnumerableSet.sol#168-170) is never used and should be removed
EnumerableSet.remove(EnumerableSet.UintSet,uint256) (node_modules/@openzeppelin/contracts/utils/structs/EnumerableSet.sol#307-309) is never used and should be removed
EnumerableSet.values(EnumerableSet.Bytes32Set) (node_modules/@openzeppelin/contracts/utils/structs/EnumerableSet.sol#208-210) is never used and should be removed
EnumerableSet.values(EnumerableSet.UintSet) (node_modules/@openzeppelin/contracts/utils/structs/EnumerableSet.sol#347-356) is never used and should be removed
SafeERC20.safeApprove(IERC20,address,uint256) (node_modules/@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol#45-58) is never used and should be removed
SafeERC20.safeDecreaseAllowance(IERC20,address,uint256) (node_modules/@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol#69-80) is never used and should be removed
SafeERC20.safeIncreaseAllowance(IERC20,address,uint256) (node_modules/@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol#60-67) is never used and should be removed
SafeERC20.safeTransferFrom(IERC20,address,address,uint256) (node_modules/@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol#29-36) is never used and should be removed
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#dead-code

Pragma version^0.8.0 (node_modules/@openzeppelin/contracts/access/Ownable.sol#4) allows old versions
Pragma version^0.8.0 (node_modules/@openzeppelin/contracts/token/ERC20/IERC20.sol#4) allows old versions
Pragma version^0.8.0 (node_modules/@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol#4) allows old versions
Pragma version^0.8.0 (node_modules/@openzeppelin/contracts/utils/Address.sol#4) allows old versions
Pragma version^0.8.0 (node_modules/@openzeppelin/contracts/utils/Context.sol#4) allows old versions
Pragma version^0.8.0 (node_modules/@openzeppelin/contracts/utils/structs/EnumerableSet.sol#4) allows old versions
Pragma version^0.8.0 (contracts/Claimable.sol#2) allows old versions
Pragma version^0.8.0 (contracts/allocation-vester/AllocationVester.sol#3) allows old versions
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity

Low level call in Address.sendValue(address,uint256) (node_modules/@openzeppelin/contracts/utils/Address.sol#55-60):
	- (success) = recipient.call{value: amount}() (node_modules/@openzeppelin/contracts/utils/Address.sol#58)
Low level call in Address.functionCallWithValue(address,bytes,uint256,string) (node_modules/@openzeppelin/contracts/utils/Address.sol#123-134):
	- (success,returndata) = target.call{value: value}(data) (node_modules/@openzeppelin/contracts/utils/Address.sol#132)
Low level call in Address.functionStaticCall(address,bytes,string) (node_modules/@openzeppelin/contracts/utils/Address.sol#152-161):
	- (success,returndata) = target.staticcall(data) (node_modules/@openzeppelin/contracts/utils/Address.sol#159)
Low level call in Address.functionDelegateCall(address,bytes,string) (node_modules/@openzeppelin/contracts/utils/Address.sol#179-188):
	- (success,returndata) = target.delegatecall(data) (node_modules/@openzeppelin/contracts/utils/Address.sol#186)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#low-level-calls

renounceOwnership() should be declared external:
	- Ownable.renounceOwnership() (node_modules/@openzeppelin/contracts/access/Ownable.sol#54-56)
transferOwnership(address) should be declared external:
	- Claimable.transferOwnership(address) (contracts/Claimable.sol#14-16)
	- Ownable.transferOwnership(address) (node_modules/@openzeppelin/contracts/access/Ownable.sol#62-65)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#public-function-that-could-be-declared-external
contracts/allocation-vester/AllocationVester.sol analyzed (8 contracts with 77 detectors), 58 result(s) found
```

- No major issues found by Slither.

4.2 AUTOMATED SECURITY SCAN

MYTHX:

Halborn used automated security scanners to assist with detecting well-known security issues and to identify low-hanging fruits on the targets for this engagement. MythX, a security analysis service for Ethereum smart contracts, is among the tools used. MythX was used to scan all the contracts and sent the compiled results to the analyzers to locate any vulnerabilities.

Results:

SphrVesting.sol

Report for contracts/allocation-vester/AllocationVester.sol
https://dashboard.mythx.io/#/console/analyses/3325127b-27b9-4309-bdd0-11bf62804954

Line	SWC Title	Severity	Short Description
3	(SWC-103) Floating Pragma	Low	A floating pragma is set.

- No major issues found by MythX.

THANK YOU FOR CHOOSING

HALBORN