Smart Electrum

Token Vesting

Smart Contract Audit
Final Report

November 10, 2021

Introduction

1. About Smart Electrum

Smart Electrum would like to create a wide and unique ecosystem that allows our users to freely use cryptocurrencies in their daily life.
Smart Electrum Ecosystem will allow users to easily exchange cryptocurrencies, together with a decentralized market where products and services. Smart Electrum is aimed at businesses and individuals involved in virtually any type of transaction.

Hundreds of millions of people have attained modern energy access over the last two decades through distribution networks.
This means that more people on Earth than ever before are now connected to ever-growing and interconnected electricity networks. This creates an enormous appetite for innovative new peer-to-peer (P2P) energy transaction platforms.

Smart Electrum would like to change the market by introducing elements of real value to it so that each token holder can experience the feeling of having something more — a digital record, a special solution that will be able to change the world.

Visit https://smartelectrum.io/ to learn more about.

2. About ImmuneBytes

ImmuneBytes is a security start-up to provide professional services in the blockchain space. The team has hands-on experience in conducting smart contract audits, penetration testing, and security consulting. ImmuneBytes's security auditors have worked on various A-league projects and have a great understanding of DeFi projects like AAVE, Compound, 0x Protocol, Uniswap, dydx.

The team has been able to secure 75+ blockchain projects by providing security services on different frameworks. ImmuneBytes team helps start-up with a detailed analysis of the system ensuring security and managing the overall project.

Visit http://immunebytes.com/ to know more about the services.

Documentation Details

The Smart Electrum team has provided the following doc for the purpose of audit:

1. https://smartelectrum.io/assets/Whitepaper.pdf

Audit Process & Methodology

ImmuneBytes team has performed thorough testing of the project starting with analyzing the code design patterns in which we reviewed the smart contract architecture to ensure it is structured and safe use of third-party smart contracts and libraries.

Our team then performed a formal line-by-line inspection of the Smart Contract in order to find any potential issues like Signature Replay Attacks, Unchecked External Calls, External Contract Referencing, Variable Shadowing, Race conditions, Transaction-ordering dependence, timestamp dependence, DoS attacks, and others.
In the Unit testing phase, we run unit tests written by the developer in order to verify the functions work as intended. In Automated Testing, we tested the Smart Contract with our in-house developed tools to identify vulnerabilities and security flaws.

The code was audited by a team of independent auditors which includes -

1. Testing the functionality of the Smart Contract to determine proper logic has been followed throughout.
2. Analyzing the complexity of the code by thorough, manual review of the code, line-by-line.
3. Deploying the code on testnet using multiple clients to run live tests.
4. Analyzing failure preparations to check how the Smart Contract performs in case of bugs and vulnerabilities.
5. Checking whether all the libraries used in the code are on the latest version.
6. Analyzing the security of the on-chain data.

Audit Details

- Project Name: Smart Electrum
- Contracts Name: TokenVesting.sol
- Languages: Solidity(Smart contract)
- Github link for initial audit: https://gist.github.com/WojcikMM/314004e0d4de055bd488390866f8ac4a
- Github link for final audit: https://gist.github.com/WojcikMM/314004e0d4de055bd488390866f8ac4a
- Platforms and Tools: Remix IDE, Truffle, Ganache, Solhint, Contract Library, Slither, SmartCheck

Audit Goals

The focus of the audit was to verify that the smart contract system is secure, resilient, and working according to its specifications. The audit activities can be grouped into the following three categories:

1. Security: Identifying security-related issues within each contract and within the system of contracts.
2. Sound Architecture: Evaluation of the architecture of this system through the lens of established smart contract best practices and general software best practices.
3. Code Correctness and Quality: A full review of the contract source code. The primary areas of focus include:
    a. Correctness
    b. Readability
    c. Sections of code with high complexity
    d. Quantity and quality of test coverage

Security Level References

Every issue in this report was assigned a severity level from the following:

High severity issues will bring problems and should be fixed.

Medium severity issues could potentially bring problems and should eventually be fixed.

Low severity issues are minor details and warnings that can remain unfixed but would be better fixed at some point in the future.

Issues	High	Medium	Low
Open	-	-	-
Closed	1	-	3

High Severity Issues

1. **Contract TokenVesting**
   function *updateVestingUsers*.

   We have created the TokenVest test contract with one user on Kovan and updatedVestingUsers with the other user and another user is able to update it successfully and the behavior is not intended and raises various security bugs.

   Contract created -
   https://kovan.etherscan.io/address/0x199725D12F85430AdFC1642C8723adaAE12018f3

   Tx hash for updatedVestingUsers with other user -
   https://kovan.etherscan.io/tx/0x73575937f4ef4c00302e0cb59ecf66f039f47a394a1345fdd3837aa4b7d7646b

   We suggest to wrap updateVestingUsers with onlyOwner modifier and the rights to update VestingUsers should be given to the owner.

```
function updateVestingUsers(VestingUserInput[] memory vestingUsers) external
onlyOwner{
        for (uint i = 0; i < vestingUsers.length; i++) {
            VestingUserInput memory vestingUser = vestingUsers[i];
            _vestingUsers[vestingUser.userAddress] =
VestingUser(vestingUser.userAddress, vestingUser.salesStagesBalance, 0);
        }
    }
```

   Amended (November 10th 2021): Issue was fixed by the **Smart Electrum** team and is no longer present in the code.

Medium severity issues

No issues were found.

Low Severity Issues

1. The pragma versions used in the contracts are not locked. Consider using the latest versions among 0.8.0 for deploying the contracts and libraries as it does not compile for any other version and can be confusing for a developer. Solidity source files indicate the versions of the compiler they can be compiled with.

   pragma solidity >=0.7.0 <0.9.0; // bad: compiles between 0.7.0 and 0.9.0
   pragma solidity 0.8.0; // good : compiles w 0.8.0 only but not the latest version

   We recommend using the locked version 0.8.0 and no explicit use of safeMath.
   Where the safeMath check is not needed, use an unchecked block.

   Amended (November 10th 2021): Issue was fixed by the **Smart Electrum** team and is no longer present in the code.

2. Function claimTokens() - line no 102

   The transfer return value is unchecked and in this case of failure or returns false. The function call will not fail.
   So we recommend checking the transfer return value and failing in the case of returning false.

   Changes Suggested:

```
require(_erc20Token.transfer(msg.sender, tokensToClaim)),
“ERC20Token: Transfer failed”);
```

   Amended (November 10th 2021): Issue was fixed by the **Smart Electrum** team and is no longer present in the code.

3. In contract TokenVesting - Line no 158 -163
   function _getTimeframesCount

   From:

```
if (timeframesCount <= maxTimeframes) {
            return timeframesCount;
        } else {
            return maxTimeframes;
```

```
        }
```

To:

```
return (timeframesCount <= maxTimeframes) ? timeframesCount :
maxTimeframes;
```

Reason:
We recommend that ternary operations should be opted over if-else. This will reduce gas price and contract size.

Amended (November 10th 2021): Issue was fixed by the **Smart Electrum** team and is no longer present in the code.

Recommendations

1. Contract TokenVesting, line 140, 154, 156

   We recommend that safe math in div operation is not needed as the denominator is constant and non-zero, so it will always pass the required checks of div in safe math.

   Amended (November 10th 2021): Issue was fixed by the **Smart Electrum** team and is no longer present in the code.

2. To improve the readability and consistency use uint256 instead of uint various places.

   Amended (November 10th 2021): Issue was fixed by the **Smart Electrum** team and is no longer present in the code.

3. Contract TokenVesting, line 100

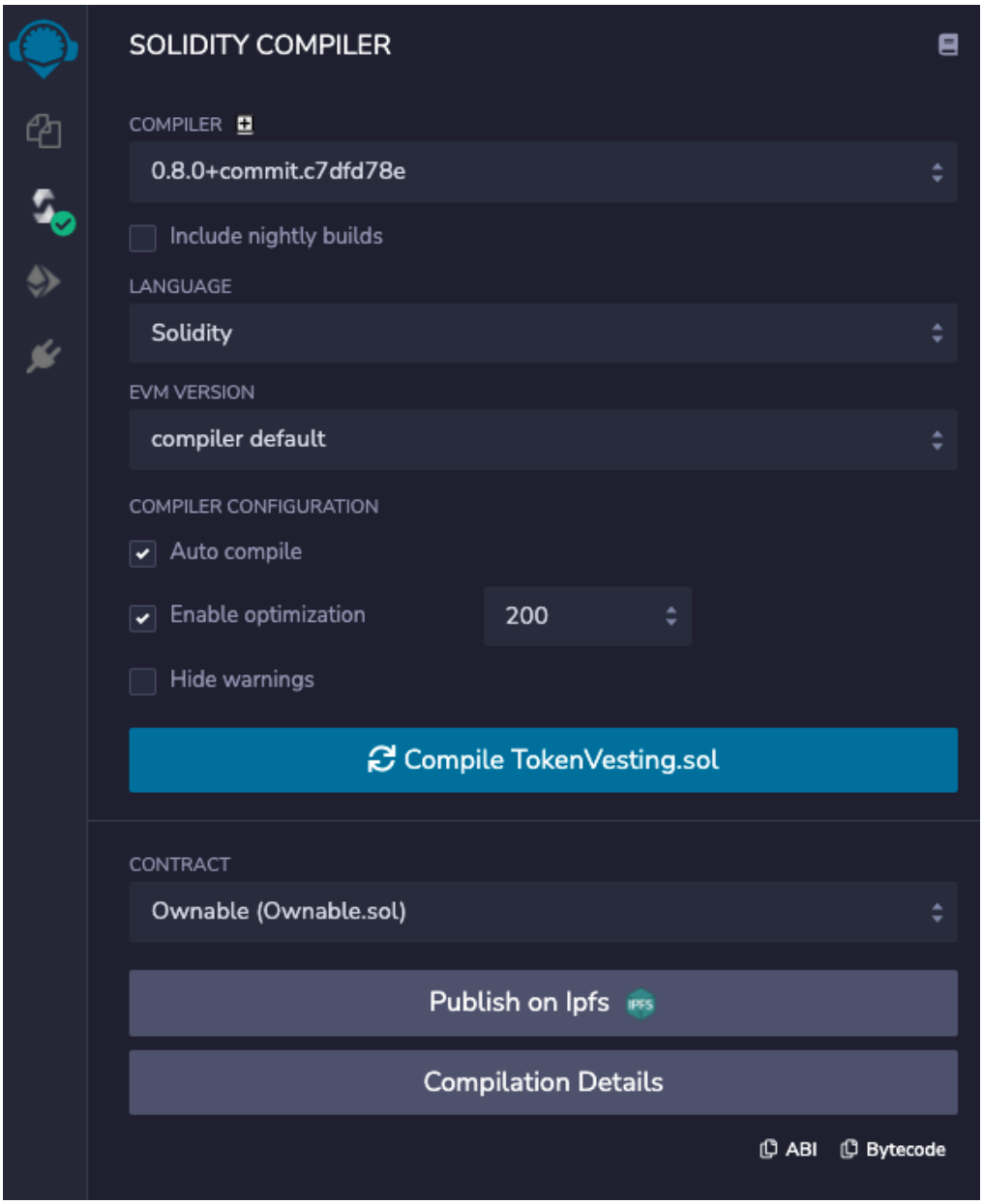
   We recommend using safe math for the arithmetic operation. Safe math adds an extra layer of security over overflow/underflow and it's always recommended to use when overflow/underflow conditions are encountered.

   Amended (November 10th 2021): Issue was fixed by the **Smart Electrum** team and is no longer present in the code.

Automated Audit

Remix Compiler Warnings

All contracts were compiled successfully on 0.8.0 in remix without any errors or warnings.



Contract Library

Contract-library contains the most complete, high-level decompiled representation of all Ethereum smart contracts, with security analysis applied to these in real-time.

Analysis was performed on kovan:
TokenVesting Contract
0x199725D12F85430AdFC1642C8723adaAE12018f3

Analysis summary can be accessed here:
https://contract-library.com/contracts/Kovan/0x199725D12F85430AdFC1642C8723adaAE12018f3

Slither

Slither, an open-source static analysis framework. This tool provides rich information about Ethereum smart contracts and has critical properties. While Slither is built as a security-oriented static analysis framework, it is also used to enhance the user's understanding of smart contracts, assist in code reviews, and detect missing optimizations.

```
INFO:Detectors:
TokenVesting.claimTokens() (contracts/TokenVesting.sol#83-103) ignores return value by
_erc20Token.transfer(msg.sender,tokensToClaim) (contracts/TokenVesting.sol#100)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#unchecked-transfer
INFO:Detectors:
TokenVesting.claimTokens() (contracts/TokenVesting.sol#83-103) uses a dangerous strict equality:
	- isInitialClaim = _getTimeframesCount(timeNow,vestingStartTime) == 0
(contracts/TokenVesting.sol#87)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#dangerous-strict-equalities
INFO:Detectors:
Reentrancy in TokenVesting.claimTokens() (contracts/TokenVesting.sol#83-103):
	External calls:
	- _erc20Token.transfer(msg.sender,tokensToClaim) (contracts/TokenVesting.sol#100)
	Event emitted after the call(s):
	- TokensClaimed(msg.sender,tokensToClaim,isInitialClaim) (contracts/TokenVesting.sol#102)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-3
INFO:Detectors:
TokenVesting.claimTokens() (contracts/TokenVesting.sol#83-103) uses timestamp for comparisons
	Dangerous comparisons:
	- isInitialClaim = _getTimeframesCount(timeNow,vestingStartTime) == 0
(contracts/TokenVesting.sol#87)
	- require(bool,string)(timeNow > vestingStartTime,Vesting not started yet.)
(contracts/TokenVesting.sol#89)
	- require(bool,string)(tokensToClaim > 0,You don't have tokens to claim.)
(contracts/TokenVesting.sol#96)
TokenVesting._getReleasedTokensAmount(uint256,uint256,uint256) (contracts/TokenVesting.sol#129-139) uses
timestamp for comparisons
	Dangerous comparisons:
```

```
        - timeNow < vestingStartTime (contracts/TokenVesting.sol#131)
TokenVesting._getTimeframesCount(uint256,uint256) (contracts/TokenVesting.sol#146-161) uses timestamp for
comparisons
        Dangerous comparisons:
        - timeNow < vestingStartTime (contracts/TokenVesting.sol#148)
        - timeframesCount <= maxTimeframes (contracts/TokenVesting.sol#156)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#block-timestamp
INFO:Detectors:
Different versions of Solidity is used:
        - Version used: ['>=0.7.0<0.9.0', '^0.8.0']
        - ^0.8.0 (node_modules/@openzeppelin/contracts/access/Ownable.sol#3)
        - ^0.8.0 (node_modules/@openzeppelin/contracts/token/ERC20/IERC20.sol#3)
        - ^0.8.0 (node_modules/@openzeppelin/contracts/utils/Context.sol#3)
        - ^0.8.0 (node_modules/@openzeppelin/contracts/utils/math/SafeMath.sol#3)
        - >=0.7.0<0.9.0 (contracts/TokenVesting.sol#1)
Reference:
https://github.com/crytic/slither/wiki/Detector-Documentation#different-pragma-directives-are-used
INFO:Detectors:
Pragma version^0.8.0 (node_modules/@openzeppelin/contracts/access/Ownable.sol#3) necessitates a version too
recent to be trusted. Consider deploying with 0.6.12/0.7.6
Pragma version^0.8.0 (node_modules/@openzeppelin/contracts/token/ERC20/IERC20.sol#3) necessitates a version
too recent to be trusted. Consider deploying with 0.6.12/0.7.6
Pragma version^0.8.0 (node_modules/@openzeppelin/contracts/utils/Context.sol#3) necessitates a version too
recent to be trusted. Consider deploying with 0.6.12/0.7.6
Pragma version^0.8.0 (node_modules/@openzeppelin/contracts/utils/math/SafeMath.sol#3) necessitates a
version too recent to be trusted. Consider deploying with 0.6.12/0.7.6
Pragma version>=0.7.0<0.9.0 (contracts/TokenVesting.sol#1) is too complex
solc-0.8.0 is not recommended for deployment
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity
INFO:Detectors:
renounceOwnership() should be declared external:
        - Ownable.renounceOwnership() (node_modules/@openzeppelin/contracts/access/Ownable.sol#53-55)
transferOwnership(address) should be declared external:
        - Ownable.transferOwnership(address)
(node_modules/@openzeppelin/contracts/access/Ownable.sol#61-64)
Reference:
https://github.com/crytic/slither/wiki/Detector-Documentation#public-function-that-could-be-declared-extern
al
INFO:Slither:. analyzed (5 contracts with 75 detectors), 15 result(s) found
```

All issues raised by slither are covered in the manual audit or are not relevant.

Concluding Remarks

While conducting the audits of the Smart Electrum smart contract, it was observed that the contracts contained only Low severity issues with a few areas of recommendations. No High or Medium severity was found.

Our auditors suggest that Low severity issues and recommendations should be resolved by Smart Electrum developers. The recommendations given will improve the operations of the smart contract.

- ***The Smart Electrum team has fixed the issues based on the auditor's recommendation.***

Disclaimer

ImmuneBytes's audit does not provide a security or correctness guarantee of the audited smart contract. Securing smart contracts is a multistep process, therefore running a bug bounty program as a complement to this audit is strongly recommended.

Our team does not endorse the Smart Electrum platform or its product nor this audit is investment advice.
Notes:

- Please make sure contracts deployed on the mainnet are the ones audited.
- Check for the code refactor by the team on critical issues.

ImmuneBytes