Genesis Vision Smart Contracts Security Analysis

Abstract

In this report we consider the security of the Genesis Vision project. Our task is to find and describe security issues in the Smart Contracts of the platform.

Procedure

In our analysis we considered Genesis Vision whitepaper and Smart Contracts code (version with latest commit 1ecf99d on 28 Aug 2017). We used several publicly available automated Solidity analysis tools. Also, we scanned project's Smart Contracts with our own tool. All the issues found by tools were manually checked (rejected or confirmed). Contracts were manually analyzed, their logic was checked and compared with the one described in the whitepaper.

Automated Analysis

We used several publicly available automated Solidity analysis tools. Here are the combined results of their analysis.

Tool	Issue type	Number of issues
Oyente	Gas requirement of function	26
Securify	Transactions May Affect Ether Receiver	18
Securify	Gas-dependent Reentrancy	3
Securify	Reentrancy With Constant Gas	1
Securify	Unchecked Transaction Data Length	2
Securify	Unhandled Exception	4
Securify	Missing Input Validation	7
Securify	Locked Ether	8
SmartDec Analyzer	Reentrancy external call	7
SmartDec Analyzer	Style guide violation	6
SmartDec Analyzer	Timestamp dependence	2
SmartDec Analyzer	Visibility	49
SmartDec Analyzer	Pragmas version	1
SmartDec Analyzer	Using var	7
SmartDec Analyzer	Integer division	4
SmartDec Analyzer	Malicious libraries	1
SmartDec Analyzer	Locked money	11
SmartDec Analyzer	Unchecked math	46
SmartDec Analyzer	Keeping secrets	3

All the issues found by tools were manually checked (rejected or confirmed). Cases when these issues lead to actual bugs or vulnerabilities are described in the next section.

Manual Analysis

Contracts were completely manually analyzed, their logic was checked and compared with the one described in the whitepaper. Besides, the results of automated analysis were manually verified and taken into account.
The analysis showed that the project does not contain any serious vulnerabilities. No discrepancies were found between the Smart Contracts and the whitepaper.
However, here are the list of minor issues that are to be fixed.

Warning: empty method

GVTTeamAllocator.sol, line 16:

```
	function GVTTeamAllocator() {
		unlockedAt = now + 12 * 30 days;
		owner = msg.sender;
		//ToDo Fill allocations table
		//allocations[0x0] = 50; 50% of team tokens
	}
```

The constructor of the `GVTTeamAllocator` contract is not finished yet. The method should be implemented before the deploy. Besides, it is highly recommended to check the project for todo's before the deploy.

Warning: same addresses

2_deploy_contracts.js, line 5:

```
	const team = accounts[0];
	const gvAgent = accounts[0];
	const migrationMaster = accounts[0];
```

The three constants: `team`, `gvAgent` and `migrationMaster` are assigned the same value `accounts[0]`. The variables should be assigned their actual values before the deploy.

Warning: misleading check

ICO.sol, line 156:

```
	require(icoState == IcoState.Running || icoState ==
		IcoState.RunningForOptionsHolders);
```

This check is in the `buyTokensInternal` method, which means it should be passed when the ICO is running. On the other hand, the
`icoState == IcoState.RunningForOptionsHolders`

condition implies that the check can also be passed during the presale for option holders. This does not lead to an exploitable vulnerability since the method is private. However, the check is misleading and thus should be removed.

Warning: unchecked math

In GVOptionProgram.sol, line 105:

```
remainingCents = usdCents - (executedTokens / optionPerCent);
```

and in many other places of code unchecked math operations are used. In the current version of code it does not lead to any vulnerabilities. However, in future versions some bugs may appear. It is recommended to consider using `SafeMath` library. This will increase the amount of gas needed but will improve the system security.

Warning: misleading comment

ICO.sol, line 100:

```
uint totalAmount = mintedTokens * 4 / 3; // 125% tokens
```

The number of minted tokens is multiplied by `4/3` while comment says it is multiplied by `5/4`. The comment is misleading and should be changed.

Checked vulnerabilities

The problems that we have found in Genesis Vision are described above. We should also describe vulnerabilities that we searched and haven't found in Genesis Vision, and thus the probability of their appearance in the code is low.

We have scanned Genesis Vision Smart Contracts for commonly known and more specific vulnerabilities. Here are some of the commonly known vulnerabilities that we considered (the full list includes them but is not limited to them).

Reentrancy (not found)

Any interaction from a contract **A** with another contract **B** and any transfer of Ether hands over control to the contract **B**. This makes it possible for **B** to call back into **A** before this interaction is completed. Furthermore, you also have to take multi-contract situations into account. The called contract (**B**) could modify the state of third (**C**) contract you depend on.
https://media.readthedocs.org/pdf/solidity/develop/solidity.pdf

Timestamp Dependence (not found)

The timestamp of the block can be manipulated by the miner, and so should not be used for critical components of the contract. Block numbers and average block time can be used to estimate time, but this is not future proof as block times may change.
https://github.com/ethereum/wiki/wiki/Safety#timestamp-dependence

Gas Limit and Loops (not found)

Loops that do not have a fixed number of iterations, for example, loops that depend on storage values are insecure. Due to the block gas limit, transactions can only consume a certain amount of gas. Either explicitly or just due to normal operation, the number of iterations in a loop can grow large enough, so required amount of gas exceed block gas limit. This cause the complete contract to be stalled at a certain point. This may not apply to constant functions that are only executed to read data from the blockchain. Still, such functions may be called by other contracts as part of on-chain operations and stall those.
http://solidity.readthedocs.io/en/develop/security-considerations.html#gas-limit-and-loops

DoS with (Unexpected) Throw (not found)

Vulnerability of this type are intended to make the contract unavailable to achieve the purpose for which it is designed. In this case it is due to the unexpected throw.
https://github.com/ethereum/wiki/wiki/Safety#dos-with-unexpected-throw

DoS with Block Gas Limit (not found)

Each Ethereum block can process a certain amount of computation. If you try to go over that, your transaction will fail. This can lead to problems even in the absence of an intentional attack. However, it's especially bad if an attacker can manipulate the amount of gas needed.
https://github.com/ethereum/wiki/wiki/Safety#dos-with-block-gas-limit

Transaction-Ordering Dependence (not found)

Since a transaction is in the mempool for a short while, one can know what actions will occur, before it is included in a block. This can be troublesome for things like decentralized markets, where a transaction to buy some tokens can be seen, and a market order implemented before the other transaction gets included.
https://github.com/ethereum/wiki/wiki/Safety#transaction-ordering-dependence-tod

tx.origin (not found)

Using `tx.origin` for authorization is insecure.
http://solidity.readthedocs.io/en/develop/security-considerations.html#tx-origin

Exception disorder (not found)

In Solidity there are several situations where an exception may be raised, e.g. if (i) the execution runs out of gas; (ii) the call stack reaches its limit; (iii) the command throw is executed. However, Solidity is not uniform in the way it handles exceptions: there are two different behaviours, which depend on how contracts call each others. The irregularity in how exceptions are handled may affect the security of contracts.
https://eprint.iacr.org/2016/1007.pdf

Gasless send (not found)

When using the function `send` to transfer ether to a contract, it is possible to incur in an out-of-gas exception. This may be quite unexpected by programmers, because transferring ether is not generally associated to executing code. The reason behind this exception is subtle. This is due to the fact that function `C.send (amount)` is compiled in the same way of a `call` with empty signature.
https://eprint.iacr.org/2016/1007.pdf

Conclusion

In this report we have considered the security of Genesis Vision Smart Contracts. We used several publicly available automated Solidity analysis tools as well as our own Smart Contracts Security tool. All the issues found by tools were manually checked (rejected or confirmed). Besides, contracts were completely manually analysed.
Smart Contracts logic was checked and compared with the one described in the whitepaper. No discrepancies were found.
The analysis showed high code quality and security of the project. However, several minor issues are to be addressed. The list of commonly known vulnerabilities that the project does not contain can be found above.
After the described issues are fixed, the Smart Contracts of the platform will be secure.

Audit conducted successfully:

SmartDec	ZERION
Sergey Pavlin [illegible]	Evgeny Yurtaev [illegible]