SECURITY AUDIT OF SMART CONTRACT

OCTOBER 12, 2017

INTRODUCTION

We thank you for giving us the opportunity to audit your smart contract code. In this report we consider the security of the Access project. Our task was to find and describe potential security issues in the smart contracts of the platform. Contracts were manually analyzed, their logic was checked. The analysis showed that the project does not contain any serious vulnerability. However, a list of issues that need to be fixed was found. This document outlines the results of the security audit.

PROCEDURE

Our analysis contained following steps:

- Project analysis
- Manual analysis of smart contracts
- Check of logic and its comparison with the requirements of the project
- Manual test of smart contracts
- Defining detected errors
- Detection of vulnerabilities
- Report of the work done

AUDIT DETAILS

The project includes 28 files (*.sol, *.js). 9 *.sol files have been manually tested and analyzed. 14 test files were written.

Reviewed files:

- AccessOptionProgram.sol
- AccessOptionToken.sol
- AccessTTeamAllocator.sol
- StandardToken.sol
- AccessToken.sol
- ERC20Basic.sol
- BasicToken.sol
- ERC20.sol
- ICO.sol

TERMINOLOGY

We categorized the findings into 4 categories basing on their vulnerability:

- **Low** – less important issue, must be analyzed.
- **Medium** – important issue, needs to be analyzed and fixed.
- **High** – important issue that might cause vulnerabilities, needs to be analyzed and fixed.
- **Critical** – serious bug causes, needs to be analyzed and fixed.

LIMITATIONS

Smart Contract security audit cannot cover all existing vulnerabilities. Even if the audit did not detect any vulnerabilities , there is no guarantee that smart contracts will be secure in the future. In most cases smart contracts are protected from certain types of attacks. We performed an extensive smart contract audit in order to discover as many vulnerabilities as possible. Audit is not a legally binding document and it doesn't guarantee anything. That it's just a discussion document.

MANUAL ANALYSIS

All Smart contracts were completely manually analyzed. The analysis shows that the project does not contain any serious vulnerability. However, a list of minor issues that should be fixed, was found. Code is being tested for work, bugs, stability issues.

Warning: medium

Contracts specifying a pragma version with the symbol (^) up front which tells the compiler to use any version of solidity bigger than 0.4.11.

```
1. pragma solidity ^0.4.11;
```

***.sol**

Recommendations: set a fixed version, since there could be major changes between versions that would make your code unstable.

Warning: low

Contract defining the unsigned integer variable ”*tokenForAllocation*” in singular which isn’t correct since it stores the total amount of tokens for allocation.

```
1. uint public tokenForAllocation;
```

AccessTTeamAllocator.sol

Recommendations: change ”*tokensForAllocation*” to ”*tokenForAllocation*”.

Warning: low

Contract defines an unused the unsigned integer variable. Defining variable costs some gas every time when it is executed.

```
1. uint tokenAmount;
```

AccessOptionProgram.sol

Recommendations: remove it if you're not using it in the code.

Warning: medium

Contract uses "*assert( )*" instead of "*require( )*" at the beginning of the function "*init( )*"

```
1. function init(address token) {
2.     gvt = ERC20Basic(token);
3.     assert(msg.sender == owner);
4. }
```

AccessTTeamAllocator.sol

Recommendations: functions "*require( )*" and "*assert( )*" are identical, but "*assert( )*" is used to validate a state after making changes in current function, while "*require( )*" is used at beginning of the function to verify the input data is well formatted; Change, "*require(msg.sender == owner)*" to "*assert(msg.sender == owner)*".

Warning: low

The function "*totalSupply()*" isn't constant. Function doesn't modify the state of the code.

```
1. function totalSupply( ) public constant returns (uint) {
2.     return totalSupply;
3. }
```

AccessToken.sol

Recommendations: this function must be constant to save some gas every time it executes in code.

Warning: medium

Contract defines an unused function "*totalSupply( )*". Defining function cost some gas every time when it executes.

```
1. function totalSupply( ) public constant returns (uint) {
2.     return totalSupply;
3. }
```

AccessToken.sol

Recommendations: remove it if you're not using it in the code.

Warning: high

The method "*initOptionProgram( )*" isn't initial. This function can be executed by anyone.

```
1. function initOptionProgram( ) public {
2.     if (optionProgram == address(0)) {
3.         optionProgram = new GVOptionProgram(this, gvAgent, team);
4.     }
5. }
```

ico.sol

Recommendations: since you don't want anybody to execute it outside the contract, this method should include modifier "*accessTeamOnly*" or be "*Initial*". Method "*accessTeamOnly*" defines the user permission by following address.

Warning: medium

The constructor of "*AccessTTeamAllocator( )*" contract should be implemented before the deploy.

```
1. function AccessTTeamAllocator( ) {
2.     unlockedAt = now + 12 * 31 days;
3.     owner = msg.sender;
4. }
```

AccessTTeamAllocator.sol

Recommendations: check the project for todo's before the deploy.

Warning: low

Unchecked math operations are used. In the current version of solidity code it does not lead to any vulnerabilities.

```
1. remainingUSDCents = currencyUSDCents - (executedTokens / optionPerCent);
```

AccessOptionProgram.sol

Recommendations: use safe math library. It will increase the amount of gas needed but will improve the system's security in future updates.

Warning: high

This check is included in the "*purchaseTokensInternal( )*" method, which means it should be passed when the ICO is running. "*RunningOptionsHolders*" condition implies that the check can also be passed, during the presale for option holders.

```
1.  require(icoState == IcoState.Running || icoState == IcoState.RunningOptionsHolders);
```

ico.sol

Recommendations: this does not lead to vulnerability, because the method "*purchaseTokensInternal( )*" is private, but the check is misleading and should be removed.

CHECKED VULNERABILITIES

Vulnerability report will contain information about every vulnerability that smart contract was tested for. These are the vulnerabilities that we searched and haven't found in the smart contracts. All these vulnerability issues will be described and can be fixed via www.smartaudit24.com team. Smart contract that will be audited will be tested for these issues.

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, it's important 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)

Every block has it's own parameters such as timestamp. Timestamp can be manipulated by miners, and that way cause problems with synchronization. Timestamp should not be used for main components of the system.

https://github.com/ethereum/wiki/wiki/Safety#timestamp-dependence

GAS LIMIT AND LOOPS (not found)

Gas limit and loops are dangerous for a project. Gas limit detects amount of resources for completion of smart contract code. If smart contract will be looped then after some

time, depending on the given gas amount, it will stall. That means system will stop working because the limit of resources (gas) had been reached.

http://solidity.readthedocs.io/en/develop/security-considerations.html#gas-limit-and-loops

DOS WITH UNEXPECTED THROW (not found)

Dos attacks are very popular nowadays. They are meant to stop smart contract functioning. This can happen if smart contract's source code has vulnerabilities and they are not detected, repaired.

https://github.com/ethereum/wiki/wiki/Safety#dos-with-unexpected-throw

DOS WITH BLOCK GAS LIMIT (not found)

Blocks have not only timestamp parameters, but many more. One of them is the amount of computations. If limit of computations have been exceeded then iteration will fail. Every failed iteration will cost gas anyway, that's especially bad if hacker has access to gas price of smart contract. This can lead to stealing resources or exceeding gas limit.

https://github.com/ethereum/wiki/wiki/Safety#dos-with-block-gas-limit

TRANSACTION-ORDERING DEPENDENCE (not found)

While iterations are made they are being holded in queue. That means the more iterations are made, the longer time a has to wait. That isn't acceptable for systems with large amount of users. By checking for this vulnerability, it will prevent the problem in the future.

https://github.com/ethereum/wiki/wiki/Safety#transaction-ordering-dependence-tod

TX.ORIGIN (not found)

This isn't the most secure way to authorize.

http://solidity.readthedocs.io/en/develop/security-considerations.html#tx-origin

EXCEPTION DISORDER (not found)

There are situations when exception could be raised, for example, If iteration runs out of gas, all stack reaches the limit and when throw command is being executed. Security of contracts can possibly be affected by irregularity of how exceptions are being handled.

https://eprint.iacr.org/2016/1007.pdf

GASLESS SEND (not found)

If gas limit was reached and iteration was sent for completion without payment, a problem will occur. Main reason of this problem is that the developers didn't expect transferring "ether" at source code. Problem occurs when function C.send (amount) is compiling with empty signature.

https://eprint.iacr.org/2016/1007.pdf

CONCLUSION

In this report we have considered the security of smart contracts. The smart contracts have been analyzed under different aspects. All the issues were manually checked and fixed. The analysis showed a high quality of project, without critical vulnerabilities. The source code of the contracts is well documented and the methods are extensively commented. No critical vulnerabilities were found during this audit. The mechanism of crowdsale is quite simple, so it shouldn't bring any major issues. All expected described issues ware analyzed and fixed. This is a secure smart contract that will store safely the funds in a stage of a crowdfund.