BLAZE

INFORMATION SECURITY

Smart Contract Security Review
Annihilat.io ANNI Salary Tokens

CLIENT

Annihilat.io

BRAZIL +55 81 3071.7148
R. VISCONDE DE JEQUITINHONHA 279 / EMPRESARIAL TANCREDO NEVES / ROOM 701 / RECIFE-PE

PORTUGAL +351 22 120 1335
PRAÇA MOUZINHO DE ALBUQUERQUE 113 / 5TH FLOOR / PORTO

SUMMARY

1.0 DOCUMENT CONTROL

1.1 DOCUMENT CONTROL

Authors	Delivery Date	Pages	Version	Status
Victor Farias and Julio Fort	20/12/2017	16	0.9	Preliminary report
Victor Farias and Julio Fort	22/12/2017	17	1.0	Final report

1.2 DOCUMENT DISTRIBUTION

Name	Title	Organisation
Julio Fort	Director of Professional Services	Blaze Information Security
Vlad Smirnov	Founder	Annihilat.io

2.0 INTRODUCTION

This document presents the results of a Smart Contract Security Review for **Annihilat.io**. This engagement aimed to verify whether the smart contract only does what it is intended to do, and to discover security vulnerabilities that could negatively affect the Annihilat.io's ANNI token before the contract gets deployed into the blockchain network.

Annihilat.io is an idea of funding based on the concept of Initial Development Offering (IDO). Essentially, it works as a salary token for contributors of a project that raised capital in an IDO. Annihilat.io uses an Ethereum ERC20-based token and contracts were written in Solidity. Details about Annihilat.io and IDO can be found in the whitepaper: https://github.com/annihilatio/ido/blob/master/article.pdf

The analysis focused on vulnerabilities related to implementation and on issues caused by architecture and design errors, as well as inconsistencies between the documentation and the code.

For each code pattern non-compliant with the Ethereum token standard or to the contract specification, deviation of best practices and vulnerability discovered during the assessment, Blaze Information Security attributed a risk severity rating and, whenever possible, validated the existence of the vulnerability with a working exploit code.

The main objectives of the assessment were the following:

- Identify the main security-related issues present in the smart contract
- Assess the level of secure coding practices present in the project
- Obtain evidences for each vulnerability and, if possible, develop a working exploit

- Document, in a clear and easy to reproduce manner, all procedures used to replicate the issue
- Recommend mitigation factors and fixes for each defect identified in the analysis
- Provide context with a real risk scenario based on a realistic threat model

3.0 EXECUTIVE SUMMARY

The engagement was performed in a period of five business days, including report writing. The smart contract security review commenced on 12/12/2017 and ended on 18/12/2017, finishing with the preliminary version of this report.

On 21/12/2017 all findings reported by Blaze Information Security were fixed accordingly by Annihilat.io. The issues are no longer present in the code of the contracts and were fixed in commits b287f07393f8b5b67cbde5c1d70dfc31b9cd5aa1 **and** afdf264d030e8313fd65e1c8c236d2a958eff7c7.

The audit was done with the assistance of automated tools as well as subjected to manual review. The generated EVM code was not inspected in this assessment.

There were three issues discovered in the contracts audited in this engagement. Overall, the contracts under scope contained significant vulnerabilities that could lead to loss of tokens and cause a significant impact to the operations of Annihilat.io. They also lacked defensive security coding patterns and had other non-recommended Solidity programming practices.

The following table summarizes the issues found in the smart contracts under scope for this audit.

It is important to notice these vulnerabilities are no longer present, as they have been corrected by Annihilat.io and the fixes reviewed by the auditors.

POINT	TITLE	SEVERITY
1	Impossibility to trade ANNI tokens back into ETH will hold investor's funds	**CRITICAL**
2	Deviation from technical specifications of the contract and code for liquidating tokens	**MEDIUM**
3	Absence of explicit visibility in some function declarations	**LOW**

4.0 SCOPE

The scope of this security review is comprised of two smart contracts written in Solidity.

- Project name: **annihilatio**
- Commit: **8ad0a1bdc7dff7e40ad5cf61aea89deaa982eab5**

Filename	Lines of code
Token_flat.sol	345
Multisig_flat.sol	498

The code audited is open source and can be found at https://github.com/annihilatio/ido/tree/8ad0a1bdc7dff7e40ad5cf61aea89deaa982eab5/smart-contracts

5.0 METHODOLOGY - SMART CONTRACT SECURITY REVIEW

Our security-oriented smart contract review follows an organized methodology with the intent to identify the largest number of vulnerabilities in the contracts under scope from the perspective of a motivated, technically capable and persistent adversary.

Special attention is directed towards critical areas of the smart contract such as burning of tokens and functioning of the multi-signature. Our process also looks into other common implementation issues that lead to problems like reentrancy, mathematical overflows and underflows, gas-related denial of service, etc.

Blaze's smart contract review methodology involves automated and manual audit techniques.

The applications are subjected to a round of dynamic analysis using tools like linters, program profilers and source code security scanners.

The contracts have their source code manually inspected for security flaws. This type of analysis has the ability to detect issues that are missed by automated scanners and static analyzers, as it can discover edge-cases and business logic-related problems.

6.0 TECHNICAL SUMMARY

6.1 DESCRIPTION OF THE SMART CONTRACTS

6.1.1 MULTISIG WALLET

In order to safely store the funds and balance consensus among the owners, annihilat.io uses a multi-signature wallet to control the token. This contract can be used to change the configuration of the token, for example to set new values for the share that goes to the investor, founders and to the project, call TGE to go live, etc.

6.1.2 TOKEN

This contract is the ANNI token per se. It contains all functions related to minting, transference of tokens among wallets, function to check balance of a wallet, burning of tokens, and other functionalities outlined in the token specification.

6.2 OBSERVATIONS ABOUT MULTISIG_FLAT.SOL

There were numerous errors when trying to run Multisig_flat.sol in automated tools. Blaze Information Security suggests the developers should review the code of the contract to understand why most Solidity security tools could not parse it.

7.0 VULNERABILITIES

7.1 IMPOSSIBILITY TO TRADE ANNI TOKENS BACK INTO ETH WILL HOLD INVESTOR'S FUNDS

SEVERITY	CRITICAL
AFFECTED POINTS	Token_flat.sol

DESCRIPTION

Fixed	Comments
Yes	Fixed in commit afdf264d030e8313fd65e1c8c236d2a958eff7c7

The contract has a function for token holders to convert their ANNI tokens into Ether (ETH).

This function, known as **burn()** is expected to work by first transferring the amount requested in tokens to a “burn address” and subsequently transferring ETH to whoever called the function.

Nevertheless, during the code review Blaze Information Security noticed that the **transfer()** function does not allow a transfer to a zero address, so the balance of the sender is never updated, and an exception will take place due to a missing condition to satisfy a **require()**, hence his or hers can call the function to burn tokens multiple times but no action will happen.

From Token_Flat.sig:

line 135:

```
address constant public burnAddress = 0x0;
```

[...]

line 217:

```
function burn(uint _amount)
    public
    isNotTgeLive
    noAnyReentrancy
    returns(bool _success)

    {

        require(balances[msg.sender] >= _amount);

        transfer(burnAddress, amount);            // here
a transfer is supposed to happen to the burn address

        msg.sender.transfer(amount);

        Burn(msg.sender, _amount);

        return true;

    }
```

[...]

line 59:

```
function transfer(address _to, uint value) isNotFrozenOnly
onlyPayloadSize(2 * 32) returns (bool success) {

        require(_to != address(0));               // the
burn address is 0×0, require() will not be satisfied and
will throw

        require(value <= balances[msg.sender]); // this
line will never be executed
```

Given the code constructs above, when a **burn()** function is called it will attempt to execute a transfer like **transfer(0×0, _amount)** and the statement **require(_to != address(0));** will not complete as expected.

The impact of this issue is severe, as investors that hold ANNI tokens will never be able to convert their tokens back to ETH.

SOLUTION

There may be different solutions to this issue. Blaze recommends not calling the transfer function in **burn()** but instead verify the amount and deduct it from the account:

```
require(value <= balances[msg.sender]);

balances[msg.sender] = balances[msg.sender].sub(value);
```

7.2 DEVIATION FROM TECHNICAL SPECIFICATIONS OF THE CONTRACT AND CODE FOR LIQUIDATING TOKENS

SEVERITY	MEDIUM
AFFECTED POINTS	Token_flat.sol

DESCRIPTION

Fixed	Comments
Yes	Fixed in commit b287f07393f8b5b67cbde5c1d70dfc31b9cd5aa1

According to the specifications of the smart contract (https://github.com/annihilatio/ido/blob/master/SMART-CONTRACT-SPECS.md#burning):

```
The tokens can be liquidated at any time by a token
holder, at this stage tokens are burnt and the token
contract sends the same amount of ETH to token holder.
Token holder obviously can not burn more tokens than he
owns. Also as tokens are burnt total supply is decreased
by the same number of tokens.
```

From Token_flat.sol:

```
/// @dev Burn tokens to burnAddress from msg.sender wallet

    /// @param _amount Amount of tokens

    function burn(uint _amount)

    public

    isNotTgeLive

    noAnyReentrancy

    returns(bool _success)

    {

        require(balances[msg.sender] >= _amount);

        transfer(burnAddress, amount);

        msg.sender.transfer(amount);
        Burn(msg.sender, _amount);

        return true;

    }
```

According to the code above the token liquidation (burn event) cannot be called anytime, as opposed to what the documentation says, but only when TGE (Token Generation Event) is not live.

The auditing team understands this issue does not bring any negative security impact to the contract per se, but it is certainly a deviation from the intended functionality outlined in the technical specifications of the smart contract.

SOLUTION

Consider removing the modifier *isNotTgeLive* from the function burn(). If the code is actually what reflects the business logic, change the documentation to reflect it accurately.

7.3 ABSENCE OF EXPLICIT VISIBILITY IN SOME FUNCTION DECLARATIONS

SEVERITY	LOW
AFFECTED POINTS	Token_flat.sol

DESCRIPTION

Fixed	Comments
Yes	Fixed in commit b287f07393f8b5b67cbde5c1d70dfc31b9cd5aa1

The audit revealed that with the exception of two functions, **_finishTge()** and **_mint(uint,uint,uint)**, both marked as *internal*, all other functions of the contracts are *public* as some of them have not been explicitly labelled. Many of these functions are state changing.

By default Solidity marks as *public* all non-labelled functions, making them being callable by external agents in the network. In order to restrict this behavior, a developer should use the labels *internal* or *private* to prevent them for being called from the outside.

While Blaze Information Security noticed there were different checks in the functions to prevent abuse from external parties calling them, not labelling functions explictly is considered a bad programming practice and should be avoided.

This recommendation hopefully will also provide the development team with an opportunity to review the visibility of the functions and reconsider their current label.

A function profiling of the audited contracts, including the visibility status of each function, can be found in the Appendix B of this report.

REFERENCE

- https://consensys.github.io/smart-contract-best-practices/recommendations/

SOLUTION

Add explicit visibility of functions and state variables.

8.0 ADDITIONAL REMARKS

- At MultiSigWallet contract notNull() modifier could be moved from addTransaction(address destination, uint value, bytes data) to functions **submitTransaction()**, **setLiveTx()** and **setFinishedTx()** to verify it at an earlier stage.

- At MultiSigWallet function **isConfirmed()** does not explicitly return false.

- At MultiSigWallet in the loop inside **getTransactionIds()** function, the var 'i' could be initialized with the value of variable 'from' instead 0 to save gas.

- At MultiSigWallet could the "IToken token" variable be changed through function "setToken" by any owner without a election? This seems to defeat the purpose of multi-signature and the idea of reaching a consensus in order to perform an action in the wallet.

- Both contracts start with the following code construct:

```
pragma solidity ^0.4.15;
```

According to the best practices outlined in https://consensys.github.io/smart-contract-best-practices/recommendations/#lock-pragmas-to-specific-compiler-version it should be:

```
pragma solidity 0.4.15;
```

9.0 CONCLUSION

The ultimate goal of a security assessment is to bring the opportunity to better illustrate the risk of an organization and help make it understand and validate its security posture against potential threats to its business.

With that in mind, Blaze Information Security provides the following recommendations that we believe should be adopted as next steps to further enhance the security posture of the Annihilat.io smart contracts:

- Fix all issues presented in the report and consider the observations made in the remarks section;
- Perform another round of audit to verify the fixes;
- Consider establishing a bug bounty program, as it is becoming increasingly common among companies in the smart contract and blockchain field.

Blaze Information Security would like to thank the team of Annihilat.io for their support and assistance during the entire engagement. We sincerely hope to work with Annihilat.io again in the near future.

10.0 APPENDIX A - VULNERABILITY CRITERIA CLASSIFICATION

Below the risk rating criteria used to classify the vulnerabilities discussed in this report:

Severity	Description
CRITICAL	Leads to the compromise of the system and the data it handles. Can be exploited by an unskilled attacker using publicly available tools and exploits. Must be addressed immediately.
HIGH	Usually leads to the compromise of the system and the data it handles.
MEDIUM	Does not lead to the immediate compromise of the system but when chained with other issues can bring serious security risks. Nevertheless, it is advisable to fix them accordingly.
LOW	Do not pose an immediate risk and even when chained with other vulnerabilities are less likely to cause serious impact.

11.0 APPENDIX B - AUTOMATED ANALYSIS RESULTS

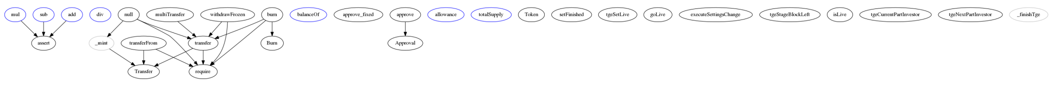
11.1 SOL-FUNCTION-PROFILER

Contract	Function	Visibility	Constant	Returns	Modifiers
ERC20	transfer(address,uint)	public	false	success	isNotFrozenOnly,onlyPayloadSize
ERC20	transferFrom(address,address,uint)	public	false	success	isNotFrozenOnly,onlyPayloadSize
ERC20	balanceOf(address)	public	true	balance	
ERC20	approve_fixed(address,uint,uint)	public	false	success	isNotFrozenOnly,onlyPayloadSize
ERC20	approve(address,uint)	public	false	success	isNotFrozenOnly,onlyPayloadSize
ERC20	allowance(address,address)	public	true	remaining	
ERC20	totalSupply()	public	true	totalSupply	
Token	Token(address,address)	public	false		
Token	()	public	false		
Token	setFinished()	public	false		only,isNotFrozenOnly,isTgeLive
Token	tgeSetLive()	public	false		only,isNotTgeLive,isNotFrozenOnly

Token	burn(uint)	public	false	_success	isNotTgeLive,noAnyReentrancy
Token	multiTransfer(address,uint)	public	false	uint	isNotFrozenOnly
Token	goLive()	public	false	bool	only,isNotFrozenOnly
Token	withdrawFrozen()	public	false		isFrozenOnly,noAnyReentrancy
Token	executeSettingsChange(uint,uint,uint)	public	false	success	only,isNotTgeLive,isNotFrozenOnly
Token	tgeStageBlockLeft()	public	false	uint	isTgeLive
Token	isLive()	public	false	bool	
Token	tgeCurrentPartInvestor()	public	false	uint	isTgeLive
Token	tgeNextPartInvestor()	public	false	uint	isTgeLive
Token	_finishTge()	internal	false		
Token	_mint(uint,uint,uint)	internal	false		

11.2 SOLGRAPH

- Token_flat.sol



- Multisig_flat.sol

solgraph could not generate a call graph of this contract due to numerous syntax errors.

11.3 OYENTE

- Token_flat.sol

```
INFO:root:Contract Token_flat.sol:Base:

INFO:oyente.symExec:Running, please wait…

INFO:oyente.symExec:        ==== Results ===

INFO:oyente.symExec:         EVM code
coverage:          100.0%

INFO:oyente.symExec:         Callstack bug:
False

INFO:oyente.symExec:         Money concurrency bug:
False
```

```
INFO:oyente.symExec:          Time dependency
bug:           False

INFO:oyente.symExec:          Reentrancy bug:
False

INFO:root:Contract Token_flat.sol:ERC20:

INFO:oyente.symExec:Running, please wait…

INFO:oyente.symExec:        ==== Results ===

INFO:oyente.symExec:          EVM code
coverage:           99.9%

INFO:oyente.symExec:          Callstack bug:
False

INFO:oyente.symExec:          Money concurrency bug:
False

INFO:oyente.symExec:          Time dependency
bug:           False

INFO:oyente.symExec:          Reentrancy bug:
False

INFO:root:Contract Token_flat.sol:SafeMath:

INFO:oyente.symExec:Running, please wait…

INFO:oyente.symExec:        ==== Results ===

INFO:oyente.symExec:          EVM code
coverage:           100.0%

INFO:oyente.symExec:          Callstack bug:
False

INFO:oyente.symExec:          Money concurrency bug:
False

INFO:oyente.symExec:          Time dependency
bug:           False

INFO:oyente.symExec:          Reentrancy bug:
False

INFO:root:Contract Token_flat.sol:Token:

INFO:oyente.symExec:Running, please wait…
```

```
INFO:oyente.symExec:        ==== Results ===

INFO:oyente.symExec:          EVM code
coverage:          76.1%

INFO:oyente.symExec:          Callstack bug:
False

INFO:oyente.symExec:          Money concurrency bug:
False

INFO:oyente.symExec:          Time dependency
bug:            False

INFO:oyente.symExec:          Reentrancy bug:
False

INFO:oyente.symExec:        == Analysis Completed ==

INFO:oyente.symExec:        == Analysis Completed ==

INFO:oyente.symExec:        == Analysis Completed ==

INFO:oyente.symExec:        == Analysis Completed ==
```

- Multisig_flat.sol

oyente's analysis could not be completed.

11.4 MYTH

- Token_flat.sol

```
The scan was completed successfully. No issues were
detected.
```

- MultiSigWallet.sol

```
== CALL with gas to dynamic address ==

Type: Warning

Contract: MultiSigWallet

Function name: executeTransaction(uint256)

PC address: 10750

The function executeTransaction(uint256) contains a
function call to an address provided as a function
argument. The available gas is forwarded to the called
contract. Make sure that the logic of the calling
contract is not adversely affected if the called
contract misbehaves (e.g. reentrancy).
```

```
== Integer Underflow ==

Type: Warning

Contract: MultiSigWallet

Function name: removeOwner(address)

PC address: 3328

A possible integer underflow exists in the function
removeOwner(address).

The SUB instruction at address 3328 may result in a
value < 0.

----------

++++ Debugging info ++++

(storage_3) - (1).]
```