PYRA TOKEN AUDIT

August 2019

BLOCKCHAIN CONSILIUM

Contents

Introduction

We first thank Pyramidion Cryptocurrency for giving us opportunity to audit their smart contract. This document outlines our methodology, audit details, and results.

Pyramidion Cryptocurrency asked us to review their PYRA token smart contract (ETH mainnet address: `0x774be8Aa7482E2d4a4961ECa756C73D662689dF1`). Blockchain Consilium reviewed the system from a technical perspective looking for bugs, issues and vulnerabilities in their code base. The Audit is valid for `0x774be8Aa7482E2d4a4961ECa756C73D662689dF1` Ethereum smart contract. The audit is not valid for any other versions of the smart contract. Read more below.

Audit Summary

This code is clean, thoughtfully written and in general well architected. The code conforms closely to the documentation and specification. *We loved reading it.*

The code is based on OpenZeppelin in many cases. In general, OpenZeppelin's codebase is good, and this is a relatively safe start.

Overall, the code is well commented and clear on what it is supposed to do for each function. The visibility and state mutability of all the functions are clearly specified, and there are no confusions.

Audit Result	✓ **PASSED**
High Severity Issues	None
Moderate Severity Issues	None
Low Severity Issues	None

Overview

The project has one Solidity file for the PYRA ERC20 Token Smart Contract, the PyramidionCryptocurrency.sol file that contains about 273 lines of Solidity code. We manually reviewed each line of code in the smart contract. All the functions and state variables are well commented using the NatSpec documentation for the functions which is good to understand quickly how everything is supposed to work.

Nice Features:
The contract provides a good suite of functionality that will be useful for the entire contract AND It **USES** SafeMath library to check for overflows and underflows, which protects against overflow and underflow attacks. All the ERC20 functions are included; it is a valid ERC20 token and in addition has some extra functionality for transferring some token fees to owner Ethereum wallet address.

Methodology:

Blockchain Consilium manually reviewed the smart contract line-by-line, keeping in mind industry best practices and known attacks, looking for any potential issues and vulnerabilities, and areas where improvements are possible.

We also used automated tools like slither for analysis and reviewing the smart contract. The raw output of these tools is included in the Appendix. These tools often give false-positives, and any issues reported by them but not included in the issue list can be considered not valid.

Classification / Issue Types Definition:

1. **High Severity:** which presents a significant security vulnerability or failure of the contract across a range of scenarios, or which may result in loss of funds.
2. **Moderate Severity:** which affects the desired outcome of the contract execution or introduces a weakness that can be exploited. It may not result in loss of funds but breaks the functionality or produces unexpected behaviour.
3. **Low Severity:** which does not have a material impact on the contract execution and is likely to be subjective.

The smart contract is considered to pass the audit, as of the audit date, if no high severity or moderate severity issues are found.

Attacks & Issues considered while auditing

In order to check for the security of the contract, we reviewed each line of code in the smart contract considering several known Smart Contract Attacks & known issues.

- **Overflows and underflows:**
  An overflow happens when the limit of the type variable uint256 , 2 ** 256, is exceeded. What happens is that the value resets to zero instead of incrementing more.

For instance, if we want to assign a value to a uint bigger than 2 ** 256 it will simple go to 0—this is dangerous.

On the other hand, an underflow happens when you try to subtract 0 minus a number bigger than 0.For example, if you subtract 0 - 1 the result will be = 2 ** 256 instead of -1.

This is quite dangerous. This contract **DOES** check for overflows and underflows by using **OpenZeppelin's** *SafeMath*.

- **Reentrancy Attack:**
  One of the major dangers of calling external contracts is that they can take over the control flow, and make changes to your data that the calling function wasn't expecting. This class of bug can take many forms, and both of the major bugs that led to the DAO's collapse were bugs of this sort.

  This smart contract includes an external call for `transferAnyERC20Token` function, accessible only by owner, the owner must check the smart contract address before executing this function to be sure whether they're executing it on a valid token, and since it does not make any state changes after calling external functions, and it follows *checks-effects-interactions* smart contract development pattern, thus it *is not vulnerable* to re-entrancy attack.

- **Replay attack:**
  The replay attack consists of making a transaction on one blockchain like the original Ethereum's blockchain and then repeating it on another blockchain like the Ethereum's classic blockchain. The ether is transferred like a normal transaction from a blockchain to another. Though it's no longer a problem because since the version 1.5.3 of *Geth* and 1.4.4 of *Parity* both implement the attack protection EIP 155 by Vitalik Buterin.
  So the people that will use the contract depend on their own ability to be updated with those programs to keep themselves secure.

- **Short address attack:**
  This attack affects ERC20 tokens, was discovered by the Golem team and consists of the following:

  A user creates an Ethereum wallet with a trailing 0, which is not hard because it's only a digit. For instance: `0xiofa8d97756as7df5sd8f75g8675ds8gsdg0`
  Then he buys tokens by removing the last zero:
  Buy 1000 tokens from account `0xiofa8d97756as7df5sd8f75g8675ds8gsdg`. If the contract has enough amount of tokens and the buy function doesn't check the length of the address of the sender, the Ethereum's virtual machine will just add zeroes to the transaction until the address is complete.

The virtual machine will return 256000 for each 1000 tokens bought. This is abug of the virtual machine.

Here is a **fix for short address attacks**

```
modifier onlyPayloadSize(uint size) {
    assert(msg.data.length >= size + 4);
    _;
}
function transfer(address _to, uint256 _value) onlyPayloadSize(2 * 32) {
    // do stuff
}
```

Whether or not it is appropriate for token contracts to mitigate the short-address attack is a contentious issue among smart-contract developers. Many, including those behind the OpenZeppelin project, have explicitly chosen not to do so. Blockchain Consilium doesn't consider short address attack an issue of the smart contract at the token level.

This contract **does not** implement an `onlyPayloadSize(uint numwords)` modifier for `transfer`, `transferFrom`, `approve`, `increaseApproval`, and `decreaseApproval` functions, it probably assumes that checks for short address attacks are handled at a higher layer (which generally are), and since the `onlyPayloadSize()` modifier started causing some bugs restricting the flexibility of the smart contracts, it's alright not to check for short address attacks at the Token Contract level to allow for some more flexibility for dAPP coding, but the checks for short address attacks must be done at some layer of coding (e.g. for buys and sells, the exchange can do it - almost all well-known exchanges check for short address attacks after the Golem Team discovered it), this contract *does not prevent short address attack*, so the *checks for short address attack must be done while buying or selling or coding a DAPP using PYRA where necessary*.

You can read more about the attack here: ERC20 Short Address Attacks.

- **Approval Double-spend:**
  Imagine that Parul approves Arun to spend 100 tokens. Later, Parul decides to approve Arun to spend 150 tokens instead. If Arun is monitoring pending transactions, then when he sees Parul's new approval he can attempt to quickly spend 100 tokens, racing to get his transaction mined in before Parul's new approval arrives. If his transaction beats Parul's, then he can spend another 150 tokens after Parul's transaction goes through.

  This issue is a consequence of the ERC20 standard, which specifies that `approve()` takes a replacement value but no prior value. Preventing the attack while complying with ERC20 involves some compromise: users should

set the approval to zero, make sure Arun hasn't snuck in a spend, then set the new value. In general, this sort of attack is possible with functions which do not encode enough prior state; in this case Parul's baseline belief of Arun's outstanding spent token balance from the Arun allowance.

It's possible for `approve()` to enforce this behaviour without API changes in the ERC20 specification:

```
if ((_value != 0) && (approved[msg.sender][_spender] != 0)) return false;
```

However, this is just an attempt to modify user behaviour. If the user does attempt to change from one non-zero value to another, then the double spend can still happen, since the attacker will set the value to zero.

If desired, a nonstandard function can be added to minimize hassle for users. The issue can be fixed with minimal inconvenience by taking a change value rather than a replacement value:

```
function increaseApproval (address _spender, uint256 _addedValue)
returns (bool success) {
  uint oldValue = approved[msg.sender][_spender];
  approved[msg.sender][_spender] = safeAdd(oldValue, _addedValue);
  return true;
}
```

Even if this function is added, it's important to keep the original for compatibility with the ERC20 specification.

Likely impact of this bug is low for most situations. This contract implements an `increaseApproval` and a `decreaseApproval` function, both of which takes the change in value instead of taking the new value, which is really *nice*.

For more, see this discussion on GitHub:
https://github.com/ethereum/EIPs/issues/20#issuecomment263524729

- **Accidental Token Loss**

    - Token Smart Contracts should prevent transferring tokens to the token smart contract address if there's no good reason to not prevent, or if there's no way to take out tokens held by the token smart contract. The PYRA smart contract prevents transferring of PYRA to PYRA smart contract address. If someone accidentally sends PYRA to the PYRA smart contract, their PYRA won't be locked up because the transaction will fail.

    - One more issue is when other ERC20 Tokens are transferred to the PYRA smart contract, there would be no way to take them out, and this is already solved by implementing the "Any Token Transfer" feature.

Issues Found

High Severity Issues

No high severity issues were found in the smart contract.

Moderate Severity Issues

No moderate severity issues were found in the smart contract.

Low Severity Issues

No low severity issues were found in the smart contract.

Line by line comments

- Line 1:
  The compiler version is specified as 0.5.11, this means the code can be compiled with solidity compilers with only version 0.5.11, that's a good practise for saving the code such that there are no unexpected issues with compiling in future when the compiler versions will be different.

- Lines 3 to 31:
  SafeMath library is included for safe arithmetic operations.

- Lines 34 to 73:
  The `Ownable` smart contract is included such that while deploying, the contract creator becomes the owner of the smart contract. The smart contract owner gets the 0.01% token transfer fee for all token transfers.

- Lines 75 to 85:
  The ERC20 Basic Solidity Interface is included.

- Lines 88 to 139:
  A `BasicToken` Smart Contract is implemented with minimal token features, along with extra feature for token transfer fees. Such that for every token transfer, 0.01% transfer fees in tokens are sent out to the smart contract owner.

- Lines 141 to 150:
  The Standard ERC20 Interface is included.

- Lines 153 to 246:
  A Standard Token implantation is included along with all standard ERC20 functions, also including the two non-standard `increaseApproval` and `decreaseApproval` functions which are useful to mitigate approval double-spend attack.

- Lines 250 to 273:
  The `PyramidionCrytocurrency` smart contract is implemented, inheriting features from `StandardToken`. The smart contract assigns the token symbol "`PYRA`", name "`Pyramidion Cryptocurrency`", decimals `8` and total supply of `1,000,000,000,000` (1 Trillion PYRA). The constructor immediately sends the total supply tokens to the smart contract creator. The smart contract creator gets 0.01% token transfer fees for all token transfers.

Appendix

Smart Contract Summary

- Contract SafeMath
    - From SafeMath
        - add(uint256,uint256) (internal)
        - div(uint256,uint256) (internal)
        - mul(uint256,uint256) (internal)
        - sub(uint256,uint256) (internal)
- Contract Ownable
    - From Ownable
        - constructor() (public)
        - transferOwnership(address) (public)
- Contract ERC20Basic
    - From ERC20Basic
        - balanceOf(address) (public)
        - transfer(address,uint256) (public)
- Contract BasicToken
    - From BasicToken
        - balanceOf(address) (public)
        - getFee() (public)
        - transfer(address,uint256) (public)
        - calculateFee(uint256) (internal)
    - From Ownable
        - constructor() (public)
        - transferOwnership(address) (public)
- Contract ERC20
    - From ERC20Basic
        - balanceOf(address) (public)
        - transfer(address,uint256) (public)
    - From ERC20

        - allowance(address,address) (public)
        - approve(address,uint256) (public)
        - transferFrom(address,address,uint256) (public)

- Contract StandardToken
    - From BasicToken
        - balanceOf(address) (public)
        - getFee() (public)
        - transfer(address,uint256) (public)
        - calculateFee(uint256) (internal)
    - From StandardToken
        - allowance(address,address) (public)
        - approve(address,uint256) (public)
        - decreaseApproval(address,uint256) (public)
        - increaseApproval(address,uint256) (public)
        - transferFrom(address,address,uint256) (public)
    - From Ownable
        - constructor() (public)
        - transferOwnership(address) (public)

- Contract PyramidionCryptocurrency
    - From BasicToken
        - balanceOf(address) (public)
        - getFee() (public)
        - transfer(address,uint256) (public)
        - calculateFee(uint256) (internal)
    - From StandardToken
        - allowance(address,address) (public)
        - approve(address,uint256) (public)
        - decreaseApproval(address,uint256) (public)
        - increaseApproval(address,uint256) (public)
        - transferFrom(address,address,uint256) (public)
    - From Ownable
        - transferOwnership(address) (public)
    - From PyramidionCryptocurrency
        - constructor() (public)
        - transferAnyERC20Token(address,address,uint256) (public)

Slither Results

```
> slither PYRA.sol

INFO:Detectors:
PyramidionCryptocurrency.transferAnyERC20Token (PYRA.sol#269-271) ignores return
value by external calls "ERC20(tokenAddr).transfer(_to,_amount)" (PYRA.sol#270)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#unused-
return
INFO:Detectors:
Ownable.transferOwnership (PYRA.sol#68-72) should be declared external
ERC20Basic.balanceOf (PYRA.sol#82) should be declared external
BasicToken.balanceOf (PYRA.sol#140-142) should be declared external
BasicToken.transfer (PYRA.sol#118-133) should be declared external
ERC20Basic.transfer (PYRA.sol#83) should be declared external
BasicToken.getFee (PYRA.sol#100-102) should be declared external
ERC20.allowance (PYRA.sol#150) should be declared external
StandardToken.allowance (PYRA.sol#220-222) should be declared external
StandardToken.transferFrom (PYRA.sol#175-196) should be declared external
ERC20.transferFrom (PYRA.sol#151) should be declared external
ERC20.approve (PYRA.sol#152) should be declared external
StandardToken.approve (PYRA.sol#208-212) should be declared external
StandardToken.increaseApproval (PYRA.sol#230-236) should be declared external
StandardToken.decreaseApproval (PYRA.sol#238-249) should be declared external
PyramidionCryptocurrency.transferAnyERC20Token (PYRA.sol#269-271) should be
declared external
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#public-
function-that-could-be-declared-as-external
INFO:Detectors:
Pragma version "0.5.11" allows old versions (PYRA.sol#1)
Reference: https://github.com/crytic/slither/wiki/Detector-
Documentation#incorrect-versions-of-solidity
INFO:Detectors:
Parameter '_owner' of BasicToken.balanceOf (PYRA.sol#140) is not in mixedCase
Parameter '_to' of BasicToken.transfer (PYRA.sol#118) is not in mixedCase
Parameter '_value' of BasicToken.transfer (PYRA.sol#118) is not in mixedCase
Parameter '_amount' of BasicToken.calculateFee (PYRA.sol#109) is not in mixedCase
Parameter '_owner' of StandardToken.allowance (PYRA.sol#220) is not in mixedCase
Parameter '_spender' of StandardToken.allowance (PYRA.sol#220) is not in mixedCase
Parameter '_from' of StandardToken.transferFrom (PYRA.sol#175) is not in mixedCase
Parameter '_to' of StandardToken.transferFrom (PYRA.sol#175) is not in mixedCase
Parameter '_value' of StandardToken.transferFrom (PYRA.sol#175) is not in
mixedCase
Parameter '_spender' of StandardToken.approve (PYRA.sol#208) is not in mixedCase
Parameter '_value' of StandardToken.approve (PYRA.sol#208) is not in mixedCase
Parameter '_spender' of StandardToken.increaseApproval (PYRA.sol#230) is not in
mixedCase
Parameter '_addedValue' of StandardToken.increaseApproval (PYRA.sol#230) is not in
mixedCase
Parameter '_spender' of StandardToken.decreaseApproval (PYRA.sol#238) is not in
mixedCase
Parameter '_subtractedValue' of StandardToken.decreaseApproval (PYRA.sol#238) is
not in mixedCase
Parameter '_to' of PyramidionCryptocurrency.transferAnyERC20Token (PYRA.sol#269)
is not in mixedCase
```

```
Parameter '_amount' of PyramidionCryptocurrency.transferAnyERC20Token
(PYRA.sol#269) is not in mixedCase
Constant 'PyramidionCryptocurrency.initialSupply' (PYRA.sol#260) is not in
UPPER_CASE_WITH_UNDERSCORES
Reference: https://github.com/crytic/slither/wiki/Detector-
Documentation#conformance-to-solidity-naming-conventions
INFO:Slither:PYRA.sol analyzed (7 contracts), 37 result(s) found
```

Purpose of the report

The Audits and the analysis described therein are created solely for Clients and published with their consent. The scope of our review is limited to a review of Solidity code and only the Solidity code we note as being within the scope of our review within this report. The Solidity language itself remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond the Solidity programming language that could present security risks. Cryptographic tokens and smart contracts are emergent technologies and carry with them high levels of technical risk and uncertainty.

The Audits are not an endorsement or indictment of any particular project or team, and the Audits do not guarantee the security of any particular project. This Report does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. No Report provides any warranty or representation to any Third-Party in any respect, including regarding the bugfree nature of code, the business model or proprietors of any such business model, and the legal compliance of any such business. No third party should rely on the Audits in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset. This Report does not constitute investment advice, is not intended to be relied upon as investment advice, is not an endorsement of this project or team, and it is not a guarantee as to the absolute security of the project. There is no owed duty to any Third-Party by virtue of publishing these Audits.

Disclaimer

The audit makes no statements or warranties about utility of the code, safety of the code, suitability of the business model, regulatory regime for the business model, or

any other statements about fitness of the contracts to purpose, or their bug free status. The audit documentation is for discussion purposes only.

THE CONTENT OF THIS AUDIT REPORT IS PROVIDED “AS IS”, WITHOUT REPRESENTATIONS AND WARRANTIES OF ANY KIND, AND BLOCKCHAIN CONSILIUM DISCLAIMS ANY LIABILITY FOR DAMAGE ARISING OUT OF, OR IN CONNECTION WITH, THIS AUDIT REPORT. COPYRIGHT OF THIS REPORT REMAINS WITH BLOCKCHAIN CONSILIUM.