PUBLIC

Security Audit

of ROCKZ's Smart Contracts

December 5, 2018

Produced for

ROCKZ

by

CHAINSECURITY

Table Of Content

Foreword

We first and foremost thank ROCKZ for giving us the opportunity to audit their smart contracts. This documents outlines our methodology, limitations, and results.

– ChainSecurity

Executive Summary

The ROCKZ smart contracts have been analyzed under different aspects, with a variety of tools for automated security analysis of Ethereum smart contracts and expert manual review.

Overall, we found that ROCKZ has clean, well-documented code. Some crucial security vulnerabilities were uncovered during the audit and several design issues were raised. These were successfully fixed by ROCKZ before deployment.

Audit Overview

Scope of the Audit

The scope of the audit is limited to the following source code files. All of these source code files were received on October 3, 2018 and an updated version on December 4, 2018:

File	SHA-256 checksum
RockzToken.sol	`8102ed5a8cb189351131e77cf4f080610c241609446faafc9d704860ef494a39`

Depth of Audit

The scope of the security audit conducted by CHAINSECURITY was restricted to:

- Scan the contracts listed above for generic security issues using automated systems and manually inspect the results.
- Manual audit of the contracts listed above for security issues.

Terminology

For the purpose of this audit, we adopt the following terminology. For security vulnerabilities, we specify the *likelihood*, *impact* and *severity* (inspired by the OWASP risk rating methodology[1]).

Likelihood represents the likelihood of a security vulnerability to be encountered or exploited in the wild.

Impact specifies the technical and business related consequences of an exploit.

Severity is derived based on the likelihood and the impact calculated previously.

We categorize the findings into 4 distinct categories, depending on their severities:

- L Low: can be considered as less important
- M Medium: should be fixed
- H High: we strongly suggest to fix it before release
- C Critical: needs to be fixed before release

These severities are derived from the likelihood and the impact using the following table, following a standard approach in risk assessment.

	IMPACT		
LIKELIHOOD	High	Medium	Low
High	C	H	M
Medium	H	M	L
Low	M	L	L

[1] https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology

During the audit concerns might arise or tools might flag certain security issues. If our careful inspection reveals no security impact, we label it as ✓ No Issue. If during the course of the audit process, an issue has been addressed technically, we label it as ✓ Fixed, while if it has been addressed otherwise by improving documentation or further specification, we label it as ✓ Addressed. Finally, if an issue is meant to be fixed in the future without immediate changes to the code, we label it as ✓ Acknowledged.

Findings that are labelled as either ✓ Fixed or ✓ Addressed are resolved and therefore pose no security threat. Their severity is still listed, but just to give the reader a quick overview what kind of issues were found during the audit.

Limitations

Security auditing cannot uncover all existing vulnerabilities, and even an audit in which no vulnerabilities are found is not a guarantee for a secure smart contract. However, auditing allows to discover vulnerabilities that were overlooked during development and areas where additional security measures are necessary.

In most cases, applications are either fully protected against a certain type of attack, or they lack protection against it completely. Some of the issues may affect the entire smart contract application, while some lack protection only in certain areas. We therefore carry out a source code review trying to determine all locations that need to be fixed. Within the customer-determined timeframe, ChainSecurity has performed auditing in order to discover as many vulnerabilities as possible.

System Overview

Token Name & Symbol	ROCKZ COIN, RKZ
Decimals	2
Token Generation	Pre-Minted, Mintable
Tokens issued	Uncapped
Token Type	ERC20 compatible
Owner Rewards	None
Vesting	None

Table 1: Facts about the RKZ token.

In the following we describe the ROCKZ COIN (RKZ). Table 1 gives the general overview.

Token Overview

ROCKZ COIN is a standard ERC20 token.

Extra Token Features

Mintable A dedicated minter role is responsible for minting tokens. All minted tokens are transferred to the minter's balance.

Burnable Tokens can be burnt, by the same minter address. The minter can only burn tokens from their own balance.

Best Practices in ROCKZ's project

Projects of good quality follow best practices. In doing so, they make audits more meaningful, by allowing efforts to be focused on subtle and project-specific issues rather than the fulfillment of general guidelines.

Avoiding code duplication is a good example of a good engineering practice which increases the potential of any security audit.

We now list a few points that should be enforced in any good project that aims to be deployed on the Ethereum blockchain. The corresponding box is ticked when ROCKZ's project fitted the criterion when the audit started.

Hard Requirements

These requirements ensure that the ROCKZ's project can be audited by CHAINSECURITY.

- [x] The code is provided as a Git repository to allow the review of future code changes.
- [x] Code duplication is minimal, or justified and documented.
- [ ] Libraries are properly referred to as package dependencies, including the specific version(s) that are compatible with ROCKZ's project. No library file is mixed with ROCKZ's own files.
- [x] The code compiles with the latest Solidity compiler version. If ROCKZ uses an older version, the reasons are documented.
- [x] There are no compiler warnings, or warnings are documented.

Soft Requirements

Although these requirements are not as important as the previous ones, they still help to make the audit more valuable to ROCKZ.

- [x] There are migration scripts.
- [x] There are tests.
- [ ] The tests are related to the migration scripts and a clear separation is made between the two.
- [x] The tests are easy to run for CHAINSECURITY, using the documentation provided by ROCKZ.
- [x] The test coverage is available or can be obtained easily.
- [x] The output of the build process (including possible flattened files) is not committed to the Git repository.
- [x] The project only contains audit-related files, or, if not possible, a meaningful separation is made between modules that have to be audited and modules that CHAINSECURITY should assume correct and out of scope.
- [x] There is no dead code.
- [x] The code is well documented.
- [x] The high-level specification is thorough and allow a quick understanding of the project without looking at the code.
- [x] Both the code documentation and the high-level specification are up to date with respect to the code version CHAINSECURITY audits.
- [x] There are no getter functions for public variables, or the reason why these getters are in the code is given.
- [x] Function are grouped together according either to the Solidity guidelines[2], or to their functionality.

[2] https://solidity.readthedocs.io/en/latest/style-guide.html#order-of-functions

Security Issues

In the following, we discuss our investigation into security issues. Therefore, we highlight whenever we found specific issues but also mention what vulnerability classes do not appear, if relevant.

`approve` race condition L ✓ Fixed

ROCKZ does not mention the possibility of a race condition on the `approve` method of its ROCKZ COIN contract. Through reordering of transactions a successful attack can be performed, allowing to spend an allowance higher than thought[3]. ROCKZ should follow best practices and provide an appropriate comment before the function, as is done in e.g. the OpenZeppelin library[4].

In addition, two extra functions - `increaseAllowance` and `decreaseAllowance` can be introduced as is currently common and recommended by the OpenZeppelin reference implementation[5].

Likelihood Low
Impact Medium

Fixed: ROCKZ added the corresponding comment warning about this.

Wrong `require` condition C ✓ Fixed

While refactoring the code to use explicit `require` statements instead of the `if(...)revert` pattern, ROCKZ has forgotten to switch the sign inside the condition. For example consider the two version of the `onlyOwner` modifier:

```
modifier onlyOwner {
    if (msg.sender != owner) revert();
    _;
}
```

and

```
modifier onlyOwner {
    require(msg.sender != owner);
    _;
}
```

The first version, which implements the correct logic, means that if the `msg.sender` is different from the `owner`, then the transaction will revert.

However, the semantics of the `require` operator will cause a `revert` only if the condition does not evaluate to true. Therefore, as of the current implementation, all functions in the `RockzToken` contract will behave in the opposite way of what is expected whenever a `require` is executed. For example everyone besides the `owner` will be allowed to `mint` tokens.

Likelihood High
Impact High

Fixed: ROCKZ successfully fixed the conditions inside `require` statements.

[3] For details see the ERC20 standard: https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md#approve
[4] Approve comment: https://github.com/OpenZeppelin/openzeppelin-solidity/blob/af42c39e6cb835b220c2ce8ef6247f074707baf7/contracts/token/ERC20/ERC20.sol#L66-L70
[5] ERC20 contract: https://github.com/OpenZeppelin/openzeppelin-solidity/blob/af42c39e6cb835b220c2ce8ef6247f074707baf7/contracts/token/ERC20/ERC20.sol

Missing zero address checks M ✓ Fixed

Rockz's Rockz Coin implements the `approve` and `transfer` functions which are defined in the `ERC20` interface. However, following best practices, a check should be introduced to prevent users from accidentally burning tokens:

```
require(_spender != address(0));
```

and

```
require(_to != address(0));
```

Likelihood Medium
Impact Medium

Fixed: Rockz implemented the additional checks.

Locked tokens in `RockzToken` contract L ✓ Fixed

Any ERC20 tokens sent to the `RockzToken` contract by mistake will be permanently locked inside it since there is no functionality to handle them. Rockz should be aware of this issues as numerous historic cases have shown that accidental ERC20 transfers to token contracts occur frequently.

Although `RockzToken` implements the ERC223 `transfer` function, Rockz Coins can still get locked in the cases when users are calling the ERC20 `transferFrom()`.

Likelihood Low
Impact Medium

Fixed: This issue does not persist anymore, as Rockz Coins can (generally) not be sent to contracts anymore. However, ChainSecurity remarks that in order to prevent transfers to just its own contract, a simple check for the receiver not being the token contract address would be sufficient.

Trust Issues

The issues described in this section are not security issues but describe functionality which is not fixed inside the smart contract and hence requires additional trust into ROCKZ, including in ROCKZ's ability to deal with such powers appropriately.

`centralMinter` has no restrictions H ✓ Addressed

The `centralMinter` role meets the specification requirements, meaning they can mint tokens to themselves. However the number of tokens that can be minted is unlimited. ROCKZ's ROCKZ COIN has no maximum cap on the amount of tokens that can be minted and there are no checks in place to prevent the minter from abusing its power.

Additionally, the specification does not mention what will trigger the minting process and if or what such relevant parameters could be. CHAINSECURITY notes that there is no enforcement of any guarantees that might have been given elsewhere.

A similar trust issue is also present for the burning process, as the minter may or may not burn tokens. This leaves potential users and investors in an uncertain situation about the RKZ supply.

Addressed: ROCKZ added a section to their whitepaper(8.3, page 37[6]) elaborating the trust model.

`owner` can mint tokens L ✓ Addressed

As defined in the specification, the `owner` must not be able to mint tokens. While this is not directly possible with the current implementation, the `owner` has full control over the `centralMinter` role. Therefore the `owner` can set themselves to the `centralMinter` and mint arbitrary amounts at will.

Even if there is a `require` that explicitly prevents the owner from becoming a minter, the owner can always circumvent this by creating a new Ethereum account and setting it as the current minter. While there is no direct remedy for this issue, CHAINSECURITY wanted to raise awareness about the trust placed in certain roles.

Addressed: See comment on previous trust issue.

[6] `https://s3.eu-central-1.amazonaws.com/alprockz-docs/RockzWhitePaperEnglish_v3.pdf`

Design Issues

The points listed here are general recommendations about the design and style of ROCKZ's project. They highlight possible ways for ROCKZ to further improve the code.

Usage of `extcodesize` to verify EOAs and block transfers H ✓ Fixed

ROCKZ makes use of the `extcodesize` assembly instruction to check for contract size and to not allow externally owned accounts (EOAs) to receive tokens. However, such checks are not secure and can be easily circumvented by calling from the constructor of a contract account[7].

While in ROCKZ's case no security risk is incurred as the main intention was to prevent locked ETH on its own contract, CHAINSECURITY wants to raise awareness that such limitations effectively prevent the ROCKZ COIN to be used for anything but transfer from one private person to another, as no exchange, wallet, multi-signature account or other smart contract will be able to pass the checks in `transfer` or `transferFrom` and hence the token will be never sent.

CHAINSECURITY doubts that this is the desired behavior and hence raises this issue.

Fixed: ROCKZ fixed the issue by removing all occurrences of `extcodesize` where it is used to verify that the recipient is not a contract.

Calls to untrusted contracts M ✓ Fixed

Both `transfer` functions make a call to an external contract. However, the address of the external contract is passed as an argument which means that the caller of `transfer` has full control over what that address is. Therefore arbitrary code can be executed, as ROCKZ has no guarantees over the implementation of the external `tokenFallback` function.

```
ERC223ReceivingContract receiver = ERC223ReceivingContract(_to);
receiver.tokenFallback(msg.sender, _value, empty);
```

If the `tokenFallback` uses a `delegatecall`, the attacker can then act in the name of the ROCKZ COIN contract since the original `msg.sender` is preserved. While indeed all checks for the `msg.sender` can be passed, there can be no state changes made as the context for code execution during the `delegatecall` remains the attacker's and events emitted still have the origin of the attacker's contract.

Nonetheless, it is bad practice to allow calls to untrusted contracts, especially if these can be arbitrary externally provided addresses. Additionally, CHAINSECURITY wants to remark that the ERC223 is in draft status for over a year[8] and that the OpenZeppelin pull request is on hold[9].

Given the previous and the fact that the use case for ERC223, avoiding any accidental token transfers, is not given because of issue 4., CHAINSECURITY recommends ROCKZ to either avoid the ERC223 approach or implement it such that it fully avoids locked tokens.

Fixed: ROCKZ removed the calls.

Conditional `revert` used instead of `require` L ✓ Fixed

ROCKZ does not use `require`, meaning that instead of:

```
require(condition);
```

the following is used:

```
if(!condition) revert();
```

[7]https://consensys.github.io/smart-contract-best-practices/recommendations/#avoid-using-extcodesize-to-check-for-externally-owned-accounts

[8]The original proposal was opened in March 2017: `https://github.com/ethereum/EIPs/issues/223`

[9]Originally submitted December 2017: `https://github.com/OpenZeppelin/openzeppelin-solidity/pull/609`

This choice goes against common practices in Solidity and no explicit reason is provided by ROCKZ for this design choice. Using `require` would be more concise and explicit.

Fixed: ROCKZ adapted `require` clauses instead.

OpenZeppelin `SafeMath` library modification L ✓ Fixed

ROCKZ changed all `require` statements to a custom `asserta` function:

```
function asserta(bool assertion) internal pure {
    if (!assertion) {
        revert();
    }
}
```

As no explicit reason is provided for this choice and CHAINSECURITY does not see a direct benefit in such a code change, the original version of the `SafeMath` library should be used and cleanly imported instead of being copied into the source code.

Fixed: ROCKZ reverted the changes made.

Deprecated keyword `constant` L ✓ Fixed

The functions `decimals`, `allowance` and `balanceOf` use the deprecated `constant` keyword.

`constant` is obsolete and replaced by `view` for functions not modifying storage and `pure` for functions not even reading any state information.

Fixed: ROCKZ does not use the deprecated keyword anymore.

Redundant check in `transferFrom` L ✓ Fixed

An `if` statement in the beginning of the `transferFrom` function checks if the balance of the receiver would increase after adding the amount of tokens.

```
120 function transferFrom(address _owner, address _to, uint256 _value) public
        returns (bool success) {
121      if (balances[_owner] < _value) revert();
122      if (balances[_to] + _value < balances[_to]) revert();
```

contracts/RockzToken.sol

However this check is redundant as later in the function the balance of the receiver is increased via a `SafeMath` addition operator which ensures no overflow is possible.

Fixed: The redundant check was removed.

Outdated Solidity version 0.4.24 L ✓ Fixed

The ROCKZ contract does not make use of the newest compiler. Without a documented reason, the newest compiler version should be used, which currently would be the version 0.4.25.

Fixed: The compiler version was updated.

Delayed event emission in `transfer` functions L ✓ Fixed

Both `transfer` functions suffer from delayed event emission.

```
function transfer(address _to, uint _value) public {
    uint codeLength;
    bytes memory empty;

    assembly {
    // Retrieve the size of the code on target address, this needs assembly .
        codeLength := extcodesize(_to)
    }

    balances[msg.sender] = balances[msg.sender].sub(_value);
    balances[_to] = balances[_to].add(_value);
    if (codeLength > 0) {
        ERC223ReceivingContract receiver = ERC223ReceivingContract(_to);
        receiver.tokenFallback(msg.sender, _value, empty);
    }
    emit Transfer(msg.sender, _to, _value, empty);
}
```

We discuss the case where the parameter `address _to` is an ERC223 enabled contract which implements the `tokenFallback` function.

Once `transfer()` is called, the balances of the sender and the receiver are updated:

```
balances[msg.sender] = balances[msg.sender].sub(_value);
balances[_to] = balances[_to].add(_value);
```

Next the `tokenFallback` function is called and the control flow enters the receiving contract. This contract forwards the received tokens immediately by using `transfer()`. As this will complete first, this transfer event is incorrectly fired first. Only after the execution has returned, the transfer event of the initial transfer will be fired.

The `Transfer` event may be fired immediately after the balances have been updated, before the call to the `ERC223ReceivingContract.tokenFallback()` which would mitigate this problem.

The correct order of the `Transfer` events may be of importance for dApps or monitoring services interacting with this token contract. Incorrectly ordered events may lead to unexpected behavior.

Fixed: Given that the external calls were removed, this issue does not apply anymore.

`transferFrom` does not call `tokenFallback` L ✓ Fixed

Both `transfer` functions call the receiver's `ERC223ReceivingContract.tokenFallback()` if the codesize of the receiver is greater than zero. This prevents accidental token transfers to contracts with no functionality to handle them.

`transferFrom()` however misses this check. Thus if someone sends tokens to one of these contracts via the `transferFrom` function, the fallback function does not get executed as opposed to when one sends tokens via one of the `transfer` functions.

This inconsistent behavior should be avoided.

Fixed: This issue does not persist anymore, as ROCKZ COINS can (generally) not be sent to contracts anymore.

Wrong order of `Transfer` and `Mint` events L ✓ Fixed

```
function mint(uint256 _amountToMint, bytes _data) public onlyMinter {
    balances[centralMinter] = balances[centralMinter].add(_amountToMint);
    totalSupply = totalSupply.add(_amountToMint);
    emit Transfer(this, centralMinter, _amountToMint, _data);
    emit Mint(centralMinter, centralMinter, _amountToMint, _data);
}
```

The `mint` function, fires the `Transfer` event before the `Mint` event. For an external observer, this suggest that non-existing tokens have been transferred before they actually got minted.

ROCKZ's code should adhere to the correct order of events.

Fixed: The order was changed.

Redundant getter functions for public variables L ✓ Fixed

Variables `name`, `decimals`, `symbol` and `totalSupply` are marked as `public` but have explicit getter functions. These are not needed as the Solidity compiler automatically creates getters for public state variables.

Therefore, either the visibility of the fields should be restricted to `private` or **`internal`**, or the getter functions removed[10].

Fixed: ROCKZ removed the corresponding getter functions.

[10]https://solidity.readthedocs.io/en/latest/contracts.html#getter-functions

Recommendations / Suggestions

✓ CHAINSECURITY's investigations showed that ROCKZ's test suite reaches a test coverage of less than 50%. Some important functions have never been tested:

```
function allowance()
function approve()
function transferFrom()
function transfer(address _to, uint _value, bytes _data)
```

CHAINSECURITY strongly recommends to improve the test suite and increase the code coverage of the tests.

✓ Both the `mint` and `burn` functions are supposed to work only with the `centralMinter` balance. However, one function refers to `centralMinter` and the other to `msg.sender`. Since only the `centralMinter` can call `burn()`, the `msg.sender` will always be the `centralMinter`. A consistent way of referring to that role should be used.

Post-audit comment: ROCKZ has fixed all of the above issues. Therefore, ROCKZ has to perform no more code changes with regards to these recommendations.

Disclaimer