ZIRCASH Token Contract Audit Report

September, 2018

Authored by

ZIRCASH Team
&
Blockchain Specialist

Contents

1. Introduction

This is the ZIRCASH Token Contract Audit report with the extensive code coverage and vulnerabilities .

1.1 Safe Audits Of Internally Written Code

A. Audit is done only when the team delivered the code as fit for purpose. This is known as a 'Chinese Wall' in some circles, and we followed these tight security procedures internally; No one had any access, even read-only access to the repository until the audit began.

B. These constraints might seem extreme, but we are very focused on maintaining safety for all clients, and the caution is warranted for any software that will be asked to secure customer funds.

1.2 Long term goals for the codebase

A. ERC20 contracts are the standard to implement – the API is short, and the general functionality encapsulated is minimal. However, in the last year a number of small-scale vulnerabilities and 'gotchas' have been published or discovered.

B. It can be said that overall the codebase is excellent, well engineered, and succeeds in these goals.

1.3 Methodology

A. Code was double audited, first working through our internal list of known attacks on smart contracts and ERC20 contracts and verified that the contracts were not vulnerable to these attacks. Attacks considered include recursive calls, over and underflow errors, replay attacks, reordering attacks, the so-called "double cross" and "single cross" attacks and Solar storm style process injection attacks.

B. It was then re-audited the files line by line, usually bottom up to maintain as little 'flow' that might cause complacency while reading as possible and worked to read and audit each small block of code carefully. Nevertheless we may have missed one or more small or large problems; this is why we publish the smart contract code for community review.

2. Files Audited

The files audited have been published at
https://github.com/ZIRCASH/Token-Information/blob/master/ZRS%20Smart%20Contract.sol

The commit hash of the code evaluated is **7487b8714f7249134d5ce2eeaaf7c263b70bf02f**.

Audited file name:

- **ZRS Smart Contract.sol**

3. Disclaimer

A. The audit makes no statements or warranties about utility 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 full bugfree status.
B. The audit documentation is for to evaluate safety of the code.

4. Executive Summary

Overall this is a super ambitious ERC20 contract that is well conceived and well executed. This contract is a good model for a modern ERC20 contract with various protections built in; as of September 2018, it is *best in class* for features delivered.

The contract provides for the following interesting features and use cases:

1. Ability to create tokens upto total supply as mentioned in whitepaper, which transaction is already done during creation of this document.
2. Resistance to recently published attacks on allowances and short message addresses
3. Ability to allow and give approval for delegated transfer.
4. Ability for locking and unlocking transfer for addresses, owning the tokens.
5. Approval mechanics for ownership transfers, minimizing ‘fat finger’ risks that could lock the contract.

In summary, We can consider this contract safe for use and built with the long term in mind.

5. Vulnerability Discussions :

The initial internal audit turned up one critical vulnerabilities and one moderate one. This audit aim to provide the amended code to mitigate these vulnerabilities.

5.1 Critical

A number of common critical vulnerabilities **do not** appear in this repository.

5.2 Moderate

A number of common minor vulnerabilities **do not** appear in this repository.

5.3 Minor

I found one “Minor” bug, See discussion on the Owner’s Rights during and after the initial token sale period. In particular, claim tokens mechanics are worth paying attention to.
(see 5.4)

Overall the safety of these contracts will increase when owner is set to a multisignature wallet.

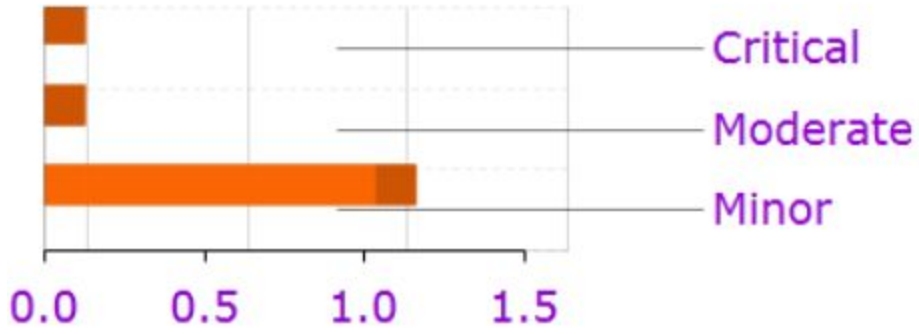

Figure 1: Vulnerability count

5.4 Minor: Owner's Rights

The owner has two sets of rights that are unusual in comparison to other ERC20 contracts: these revolve around the Ownership change mechanics and the claim Tokens method.

5.4.1 A great tackle to approve Ownership Changes

A. It is great to have an approval mechanic for ownership change, but it is worth talking about the security trade-offs. In the very unlikely case of racing an attacker who has compromised the owner key but not changed the owner yet, requiring approval from a new owner address may slow down seizure of the contract from the attacker. It is hard to imagine an attacker that is sophisticated enough to engage as the owner after key compromise but not sophisticated enough to change the owner at the first chance, though.

B. In exchange for this very slight reduction in security, you get what seems to me to be a big benefit - nobody, including an attacker with the owner key – can lock the contract by changing the owner to an invalid address. You are always guaranteed that the new owner can sign transactions that are function calls. I think this is a good trade-off, and recommend this idea in general.

5.4.3 Claim Tokens:

Note the initial sale does not have any situation in which tokens or ETH may be claimable. A full discussion of these mechanisms is beyond the scope of the audit, but readers are encouraged to engage to implement that.

Summary (Suggestive but not Mandatory):

1. Multisig-wallet address for owner.
2. Claim Token functionality will be a value add depending on business situations.

6. Line by Line Comments

Included below are the line by line comments and notes as part of the audit process.

File Name: ZIRCASH Smart Contract.sol

Line 1: Solidity Version

Recent version of Solidity is used 0.4.24. It is a great way to tackle old vulnerabilities which present in old versions. In the below table the known bugs are mentioned which are tackled classically by using latest stable version of solidity. In other terms we can say this contract is free from the below bugs:


```
[
  {
    "name": "ZeroFunctionSelector",
    "summary": "It is possible to craft the name of a function such that it is executed instead of the fallback function in very specific circumstances.",
    "description": "If a function has a selector consisting only of zeros, is payable and part of a contract that does not have a fallback function and at most five external functions in total, this function is called instead of the fallback function if Ether is sent to the contract without data.",
    "fixed": "0.4.18",
    "severity": "very low"
  },
  {
    "name": "DelegateCallReturnValue",
    "summary": "The low-level .delegatecall() does not return the execution outcome, but converts the value returned by the functioned called to a boolean instead.",
    "description": "The return value of the low-level .delegatecall() function is taken from a position in memory, where the call data or the return data resides. This value is interpreted as a boolean and put onto the stack. This means if the called function returns at least 32 zero bytes, .delegatecall() returns false even if the call was successful.",
    "introduced": "0.3.0",
    "fixed": "0.4.15",
    "severity": "low"
  },
  {
    "name": "ECRecoverMalformedInput",
    "summary": "The ecrecover() builtin can return garbage for malformed input.",
    "description": "The ecrecover precompile does not properly signal failure for malformed input (especially in the 'v' argument) and thus the Solidity function can return data that was previously present in the return area in memory.",
    "fixed": "0.4.14",
    "severity": "medium"
  },
  {
    "name": "SkipEmptyStringLiteral",
    "summary": "If \"\" is used in a function call, the following function arguments will not be correctly passed to the function.",
    "description": "If the empty string literal \"\" is used as an argument in a function call, it is skipped by the encoder. This has the effect that the encoding of all arguments following this is shifted left by 32 bytes and thus the function call data is corrupted.",
    "fixed": "0.4.12",
```

```
        "severity": "low"
    },
    {
        "name": "ConstantOptimizerSubtraction",
        "summary": "In some situations, the optimizer replaces certain numbers in the code with routines that compute different numbers.",
        "description": "The optimizer tries to represent any number in the bytecode by routines that compute them with less gas. For some special numbers, an incorrect routine is generated. This could allow an attacker to e.g. trick victims about a specific amount of ether, or function calls to call different functions (or none at all).",
        "link": "https://blog.ethereum.org/2017/05/03/solidity-optimizer-bug/",
        "fixed": "0.4.11",
        "severity": "low",
        "conditions": {
            "optimizer": true
        }
    },
    {
        "name": "IdentityPrecompileReturnIgnored",
        "summary": "Failure of the identity precompile was ignored.",
        "description": "Calls to the identity contract, which is used for copying memory, ignored its return value. On the public chain, calls to the identity precompile can be made in a way that they never fail, but this might be different on private chains.",
        "severity": "low",
        "fixed": "0.4.7"
    },
    {
        "name": "OptimizerStateKnowledgeNotResetForJumpdest",
        "summary": "The optimizer did not properly reset its internal state at jump destinations, which could lead to data corruption.",
        "description": "The optimizer performs symbolic execution at certain stages. At jump destinations, multiple code paths join and thus it has to compute a common state from the incoming edges. Computing this common state was simplified to just use the empty state, but this implementation was not done properly. This bug can cause data corruption.",
        "severity": "medium",
        "introduced": "0.4.5",
        "fixed": "0.4.6",
        "conditions": {
            "optimizer": true
        }
    },
    {
        "name": "HighOrderByteCleanStorage",
        "summary": "For short types, the high order bytes were not cleaned properly and could overwrite existing data.",
        "description": "Types shorter than 32 bytes are packed together into the same 32 byte storage slot, but storage writes always write 32 bytes. For some types, the higher order bytes were not cleaned properly, which made it sometimes possible to overwrite a variable in storage when writing to another one.",
        "link":
"https://blog.ethereum.org/2016/11/01/security-alert-solidity-variables-can-overwritten-storage/",
        "severity": "high",
        "introduced": "0.1.6",
        "fixed": "0.4.4"
    },
    {
        "name": "OptimizerStaleKnowledgeAboutSHA3",
```

```
        "summary": "The optimizer did not properly reset its knowledge about SHA3 operations resulting in some hashes (also used for storage variable positions) not being calculated correctly.",
        "description": "The optimizer performs symbolic execution in order to save re-evaluating expressions whose value is already known. This knowledge was not properly reset across control flow paths and thus the optimizer sometimes thought that the result of a SHA3 operation is already present on the stack. This could result in data corruption by accessing the wrong storage slot.",
        "severity": "medium",
        "fixed": "0.4.3",
        "conditions": {
          "optimizer": true
        }
    },
    {
        "name": "LibrariesNotCallableFromPayableFunctions",
        "summary": "Library functions threw an exception when called from a call that received Ether.",
        "description": "Library functions are protected against sending them Ether through a call. Since the DELEGATECALL opcode forwards the information about how much Ether was sent with a call, the library function incorrectly assumed that Ether was sent to the library and threw an exception.",
        "severity": "low",
        "introduced": "0.4.0",
        "fixed": "0.4.2"
    },
    {
        "name": "SendFailsForZeroEther",
        "summary": "The send function did not provide enough gas to the recipient if no Ether was sent with it.",
        "description": "The recipient of an Ether transfer automatically receives a certain amount of gas from the EVM to handle the transfer. In the case of a zero-transfer, this gas is not provided which causes the recipient to throw an exception.",
        "severity": "low",
        "fixed": "0.4.0"
    },
    {
        "name": "DynamicAllocationInfiniteLoop",
        "summary": "Dynamic allocation of an empty memory array caused an infinite loop and thus an exception.",
        "description": "Memory arrays can be created provided a length. If this length is zero, code was generated that did not terminate and thus consumed all gas.",
        "severity": "low",
        "fixed": "0.3.6"
    },
    {
        "name": "OptimizerClearStateOnCodePathJoin",
        "summary": "The optimizer did not properly reset its internal state at jump destinations, which could lead to data corruption.",
        "description": "The optimizer performs symbolic execution at certain stages. At jump destinations, multiple code paths join and thus it has to compute a common state from the incoming edges. Computing this common state was not done correctly. This bug can cause data corruption, but it is probably quite hard to use for targeted attacks.",
        "severity": "low",
        "fixed": "0.3.6",
        "conditions": {
          "optimizer": true
        }
    },
    {
        "name": "CleanBytesHigherOrderBits",
```

```
        "summary": "The higher order bits of short bytesNN types were not cleaned before comparison.",
        "description": "Two variables of type bytesNN were considered different if their higher order bits, which are not part of the actual value, were different. An attacker might use this to reach seemingly unreachable code paths by providing incorrectly formatted input data.",
        "severity": "medium/high",
        "fixed": "0.3.3"
    },
    {
        "name": "ArrayAccessCleanHigherOrderBits",
        "summary": "Access to array elements for arrays of types with less than 32 bytes did not correctly clean the higher order bits, causing corruption in other array elements.",
        "description": "Multiple elements of an array of values that are shorter than 17 bytes are packed into the same storage slot. Writing to a single element of such an array did not properly clean the higher order bytes and thus could lead to data corruption.",
        "severity": "medium/high",
        "fixed": "0.3.1"
    },
    {
        "name": "AncientCompiler",
        "summary": "This compiler version is ancient and might contain several undocumented or undiscovered bugs.",
        "description": "The list of bugs is only kept for compiler versions starting from 0.3.0, so older versions might contain undocumented bugs.",
        "severity": "high",
        "fixed": "0.3.0"
    }
]
```


Line 18: Safemath Contract

Latest zeppelin Safemath Library is used from the open source project named 'OpenZeppelin'.
https://github.com/OpenZeppelin/zeppelin-solidity/blob/master/contracts/math/SafeMath.so
It is highly appreciable to use SafeMath contract and reuse it. Most of the successful ICO contracts implement the safety using the same.

It helps in the following way:
1. Reusable to all basic 4 operations in the contract.
2. It mitigates **Overflow** and **Underflow** attacks.

Line 54: Token Contract

ERC 20 standard is implemented as an interface type contract, where the functions are declared but not implemented. It is good approach to make the contract modular and readable.
All ERC 20 compatible methods and events are present in the contract.

Line 100: Checks in Transfer function

This is nice; it protects from many errors and attacks.
require(_to != address(0)) -> Can not send to a **empty or null address**.

(accounts [msg.sender] < _value) return false -> great check to mitigate **spending more amount** than existing in the account.

(_value > 0 && msg.sender != _to) -> Classical checking of positive balance and the sender can not send the same fund to itself, it nicely tackles the **double spending** attack.

Line 122: Checks in Transfer from function

This is nice; it protects from many errors and attacks.
require(_to != address(0)) -> Can not send to a **empty or null address**.

(accounts [_from] < _value) return false -> Great check to mitigate **spending more amount** than existing in the account.

(_value > 0 && _from != _to) -> Classical checking of positive balance and the sender can not send the same fund to itself, it nicely tackles the **double spending** attack.

(allowances [_from][msg.sender] < _value) return false -> Spender can not send more amount than
allocated via approval method.

Line 143 and 277: resistance to approve racing

require(allowance (msg.sender, _spender) == 0 || _value == 0) -> This is good! It keeps an approver safe from getting raced to a double spend on approval. This could prove to be a very valuable protection for heavy use of the ERC20 approval mechanism.

Line 190: Max Token Count

The value mentioned is same as in whitrepaper. Constant keyword makes it unalterable once the contract is deployed..

Line 195: Owner declaration

Owner variable is declared as private which is a classic way to restrict the access of the owner variable from public. It mitigates major vulnerability of ownership change.

Line 231, 232, 233: Decimal, Symbol and Name

The fields mentioned are same as mentioned in Whitepaper.

Line 288: Create Tokens

It is the main contract to generate tokens during TGE. It can only be called by the owner which is obvious best security and necessary.

(_value > safeSub (MAX_TOKEN_COUNT, tokenCount)) return false -> This check ensures that during TGE total generated tokens can not increase the max supply for TGE. [TGE aka Token Generation Event]

Line 315: Transfer Ownership

Would be nice to stop a foot fault / attack by making sure at least that newOwner isn't 0x0. That doesn't stop someone from making up a malicious newOwner but it would stop a fat finger function call.

Line 325, 338: Freeze and UnFreeze transfer of Tokens

This is a very good secure functionality to freeze or unfreeze the transfers of tokens. It can only be called by the owner. If there is some open risk with the tokens from some exchanges then owner can freeze the transfer and trading to mitigate the risk of loss of tokens.

Line 355: Refund Tokens

The function is implemented nicely, this function implementation is not present on most of other ICO contracts. But ZIRCASH really does a nice job by thinking about other tokens. If user mistakenly send any other tokens to this contract address, then that will not be locked in the contract rather refund tokens function can send the tokens back to the user. It proves how ZIRCASH considers transparency and created most of the functions to mitigate this type of mistakes.

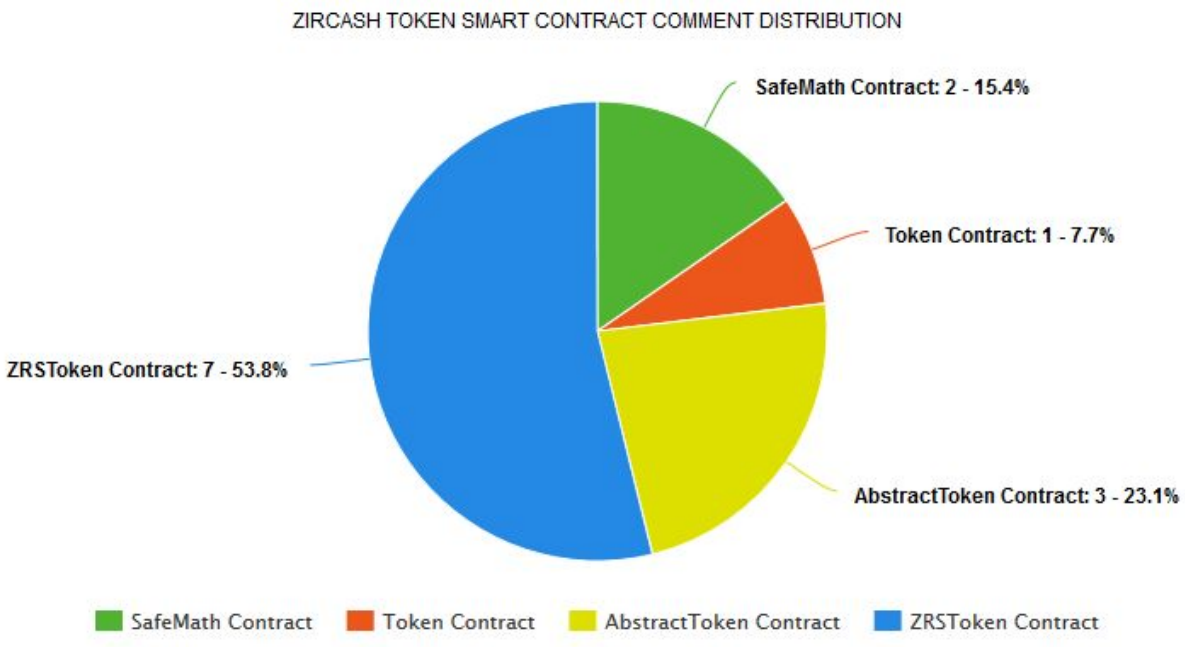

Figure 2: Comment Distribution

7. Notes on Some Vulnerabilities

7.1 Short Address Attack

The codebase fixes a common vulnerability in ERC20 tokens; some explanation on the vulnerability is included here.

This attack can be entirely prevented by doing a length check on msg.data. In the case of transfer(), the length should be 68:

assert(msg.data.length == 68);

Vulnerable functions include all those whose last two parameters are an address, followed by a value.

In ERC20 these functions include transferFrom and approve.

A general way to implement this is with a modifier (slightly modified from one suggested by redditor izqui9):

```
modifier onlyPayloadSize(uint numwords) {
    assert(msg.data.length == numwords * 32 + 4); _;
```

}

function transfer(address _to, uint256 _value) onlyPayloadSize(2) { }

If an exploit of this nature were to succeed, it would arguably be the fault of the exchange, or whoever else improperly constructed the offending transactions. However, we believe in defense in depth. It's easy and desirable to make tokens which cannot be stolen this way, even from poorly-coded exchanges.

Because dividend-paying tokens execute additional code on transfer we think this issue is worth looking at more carefully than most token contracts, and recommend that at very least the assert be added to the code.

Further explanation of this attack is here:
http://vessenes.com/the-erc20-short-address-attack-explained/

But Coding in this way are creating some problems with multisig wallet codings, so it is recommendable to list in secure exchanges which inherently checks the address and function arguments.

Most of the contract are not using the payload size checking on the Solidity level to be compliant with the solidity coding standard. So the contract has done the correct thing of not including the mechanism

7.2 Approval Double-spend

Imagine that Alice approves Mallory to spend 100 tokens. Later, Alice decides to approve Mallory to spend 150 tokens instead. If Mallory is monitoring pending transactions, then when he sees Alice's new approval he can attempt to quickly spend 100 tokens, racing to get his transaction mined in before Alice's new approval arrives. If his transaction beats Alice's, then he can spend another 150 tokens after Alice'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 Mallory 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 Alice's baseline belief of Mallory's outstanding spent token balance from the Mallory allowance.

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

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

This is already implemented nicely in the contract.

However, this is just an attempt to modify user behavior. If the user does attempt to change from one non-zero value to another, then the doublespend 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, but exchanges without preferred mining contracts may wish to consider carefully their approve workflow.

For more, see this discussion on github:
https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729

8. References

A. https://github.com/ConsenSys/smart-contract-best-practices
B. https://lightrains.com/blogs/smart-contract-best-practices-solidity
C. https://www.gitbook.com/book/alexxiong97/smart-contract-best-practices/details
D. https://blog.zeppelin.solutions/onward-with-ethereum-smart-contract-security-97a827e47702