CERTIK

Audit Report

Produced by CertiK

for MIR COIN

23RD Dec, 2019

CertiK Audit Report For MIR

Request Date: 2019-12-18
Revision Date: 2019-12-24

Contents

Disclaimer

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability) set forth in the Verification Services Agreement between CertiK and MIR(the "Company"), or the scope of services/verification, and terms and conditions provided to the Company in connection with the verification (collectively, the "Agreement"). This report provided in connection with the Services set forth in the Agreement shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Agreement. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes without CertiK's prior written consent.

About CertiK

CertiK is a technology-led blockchain security company founded by Computer Science professors from Yale University and Columbia University built to prove the security and correctness of smart contracts and blockchain protocols.

CertiK, in partnership with grants from IBM and the Ethereum Foundation, has developed a proprietary Formal Verification technology to apply rigorous and complete mathematical reasoning against code. This process ensures algorithms, protocols, and business functionalities are secured and working as intended across all platforms.

CertiK differs from traditional testing approaches by employing Formal Verification to mathematically prove blockchain ecosystem and smart contracts are hacker-resistant and bug-free. CertiK uses this industry-leading technology together with standardized test suites, static analysis, and expert manual review to create a full-stack solution for our partners across the blockchain world to secure 6.2B in assets.

For more information: https://certik.org/

Executive Summary

This report has been prepared for MIR to discover issues and vulnerabilities in the source code of their smart contracts. A comprehensive examination has been performed, utilizing CertiK's Formal Verification Platform, Static Analysis, and Manual Review techniques.

The auditing process pays special attention to the following considerations:

- Testing the smart contracts against both common and uncommon attack vectors.
- Assessing the codebase to ensure compliance with current best practices and industry standards.
- Ensuring contract logic meets the specifications and intentions of the client.
- Cross referencing contract structure and implementation against similar smart contracts produced by industry leaders.
- Thorough line-by-line manual review of the entire codebase by industry experts.

Vulnerability Classification

CertiK categorizes issues into three buckets based on overall risk levels:

Critical

Code implementation does not match specification, which could result in the loss of funds for contract owner or users.

Medium

Code implementation does not match the specification under certain conditions, which could affect the security standard by loss of access control.

Low

Code implementation does not follow best practices, or uses suboptimal design patterns, which could lead to security vulnerabilities further down the line.

Testing Summary

PASS

CERTIK *believes this smart contract passes security qualifications to be listed on digital asset exchanges.*

Dec 23, 2019

Type of Issues

CertiK's smart label engine applied 100% formal verification coverage on the source code. Our team of engineers has scanned the source code using proprietary static analysis tools and code-review methodologies. The following technical issues were found:

Title	**Description**	**Issues**	**SWC ID**
Integer Overflow and Underflow	An overflow/underflow occurs when an arithmetic operation reaches the maximum or minimum size of a type.	0	SWC-101
Function Incorrectness	Function implementation does not meet specification, leading to intentional or unintentional vulnerabilities.	0	
Buffer Overflow	An attacker can write to arbitrary storage locations of a contract if array of out bound happens	0	SWC-124
Reentrancy	A malicious contract can call back into the calling contract before the first invocation of the function is finished.	0	SWC-107
Transaction Order Dependence	A race condition vulnerability occurs when code depends on the order of the transactions submitted to it.	0	SWC-114
Insecure Randomness	Using block attributes to generate random numbers is unreliable, as they can be influenced by miners to some degree.	0	SWC-120
Delegatecall to Untrusted Callee	Calling untrusted contracts is very dangerous, so the target and arguments provided must be sanitized.	0	SWC-112
State Variable			

Default Visibility	Labeling the visibility explicitly makes it easier to catch incorrect assumptions about who can access the variable.	0	SWC-108
Function Default Visibility	Functions are public by default, meaning a malicious user can make unauthorized or unintended state changes if a developer forgot to set the visibility.	0	SWC-100
Uninitialized Variables	Uninitialized local storage variables can point to other unexpected storage variables in the contract.	0	SWC-109
Assertion Failure	The assert() function is meant to assert invariants. Properly functioning code should never reach a failing assert statement.	0	SWC-110
	0	SWC-111	
Unused Variables	Unused variables reduce code quality	0	

Vulnerability Details

Critical

No issue found.

Medium

No issue found.

Low

No issue found.

Manual Review Notes

Source Code SHA-256 Checksum

- **eosio.token.hpp**
  `1dcb6f8994956c1f53f9400324c0a23069fdd43b217565134fb4125a3cfc83ed`
- **eosio.token.cpp**
  `af27f8d968be0a4f00b725814dbf4e076cf0202b8fb81a27d437bef7d6246829`

Summary

CertiK worked closely with MIR to audit the design and implementation of its soon-to-be released smart contract. To ensure comprehensive protection, the source code was analyzed by the proprietary CertiK formal verification engine and manually reviewed by our smart contract experts and engineers. That end-to-end process ensures proof of stability as well as a hands-on, engineering-focused process to close potential loopholes and recommend design changes in accordance with best practices.

Overall, we found MIR's smart contracts to follow good practices. With the final update of source code and delivery of the audit report, we conclude that the contract is structurally sound and not vulnerable to any classically known anti-patterns or security issues. The audit report itself is not necessarily a guarantee of correctness or trustworthiness, and we always recommend to seek multiple opinions, continually improve the codebase, and perform additional tests before the mainnet release.

Persistence Tables

The MIRCoin smart contract uses 2 multi-index tables to store data.

Table	Table name	Read by functions	Written by functions
`acounts`	`"acounts"`	`sub_balance()`, `add_balance()`, `open()`, `close()`, `get_balance()`	`sub_balance()`, `add_balance()`, `open()`
`stats`	`"stat"`	`create()`, `issue()`, `retire()`, `transfer()`, `open()`, `get_supply()`	`create()`, `issue()`, `retire()`

- Table `accounts`:

Property	Type	Key	Description
balance	eosio::asset	primary: balance.symbol.code().raw()	

- Table stats:

Property	Type	Key	Description
supply	eosio::asset	primary: supply.symbol.code().raw()	
max_supply	eosio::asset		
issuer	eosio::name		

Functions

The MIRCoin smart contract consists of 10 functions.

Function	Static	Access	Auth	Supply change	Balance change
create()		public	_self		
issue()		public	st.issuer	st.supply += quantity	
retire()		public	st.issuer	st.supply -= quantity	
transfer()		public	from		from.balance -= quantity to.balance += quantity
open()		public	ram_payer		
close()		public	owner		
get_supply()	✓	public			
get_balance()	✓	public			
sub_balance()		private			from.balance -= quantity
add_balance()		private			to.balance += quantity

Recommendations

Items in this section are not critical to the overall functionality of MIR's smart contracts; however, we leave it to the client's discretion to decide whether to address them before the final deployment of source codes. Recommendations are labeled CRITICAL, MAJOR, MINOR, INFO, and DISCUSSION in decreasing significance level.

eosio.token.cpp

- INFO Too many check() calls which might cause confusion. Recommend extracting similar code into separate functions and following the single responsibility principle. As an example, we can define a function

```
void is_valid_quantity (const asset& quantity, const string& memo) {
   check( quantity.is_valid(), "invalid quantity" );
   check( quantity.amount > 0, memo );
}
```

to replace the following `check()` calls:

```
void token::create(...) {
    ...
    check( maximum_supply.is_valid(), "invalid supply" );
    check( maximum_supply.amount > 0, "max-supply must be positive" );
    ...
}

void token::issue(...) {
    ...
    check( quantity.is_valid(), "invalid quantity" );
    check( quantity.amount > 0, "must issue positive quantity" );
    ...
}

void token::retire(...) {
    ...
    check( quantity.is_valid(), "invalid quantity" );
    check( quantity.amount > 0, "must retire positive quantity" );
    ...
}

void token::transfer(...) {
    ...
    check( quantity.is_valid(), "invalid quantity" );
    check( quantity.amount > 0, "must transfer positive quantity" );
    ...
}
```

Best practice

Smart contract development requires a particular engineering mindset. A failure in the initial construction can be catastrophic, and changing the project after the fact can be exceedingly difficult.
To ensure success and to avoid the challenges above smart contracts should here to best practices at their conception. Below, we summarized a checklist of key points that help to indicate a high overall quality of the current project. (✓ indicates satisfaction; × indicates unsatisfaction; − indicates inapplicablility)

Security

Identifying security related issues within each contract and within the system of contracts. Some of the commonly known vulnerabilities that were considered are listed below.

✓ No numerical overflows: All numerical calculations must be protected against potential overflow and underflow in arithmetic operations.

✓ Authorization checks: All sensitive actions in the contract require authorizations.

− Apply checks: Using the standard `EOSIO_DISPATCH` for the apply dispatching.

✓ Memory management: Proper use of pointers and references.

✓ Persistent data handling: Handling persistent data on RAM. Proper use of multi index table

- ✓ String parameter length: To ensure string parameters to actions are not unbounded in length the string parameters.
- ✓ Safe from rollback attacks.
- ✓ RAM fill up protection (RAM DoS): Storing all items in the user's RAM space rather than the contract RAM except for actions requiring the contract's `_self` permission.
- ✓ CPU bandwidth protection (CPU DoS): All actions in the contract are initiated by the caller of the contract therefore the caller would need to have enough staked CPU resources to perform the called action.
- ✓ Source code version: The contract code has been updated to compile with a recent version of the eosio compiler tools.

Code Correctness and Quality

The primary areas of focus include:

- ✓ Correctness
- ✓ Readability
- ✓ Sections of code with high complexity
- ✓ Quantity and quality of test coverage