Quantstamp Contract Security Certificate

November 14th 2020 — Quantstamp Verified

SIDC

This smart contract audit was prepared by Quantstamp, the protocol for securing smart contracts.

Executive Summary

Type	SIDC
Auditors	Kacper Bąk, Senior Research Engineer Alex Murashkin, Senior Software Engineer Martin Derka, Senior Research Engineer
Timeline	2020-11-8 through 2020-11-14
Languages	Solidity
Methods	Architecture Review, Computer-Aided Verification, Manual Review
Source Code	SIDC

Summary of Findings

ID	Description	Severity	Status
QSP-1	Compiler version	Low	Resolved
QSP-2	Constructor writing problem	Low	Resolved
QSP-3	Return criteria	Low	Resolved
QSP-4	Event standards	Low	Resolved
QSP-5	False recharge	Low	Resolved
QSP-6	Condition competition for the approve authorization function	Medium Risk	Resolved
QSP-7	Cyclic Dos problem	Low	Acknowledged
QSP-8	Cycle safety	Low	Acknowledged
QSP-9	Arithmetic overflow	Medium Risk	Unresolved
QSP-10	Coin burning overflow problem	Medium Risk	Resolved
QSP-11	Reentrant loopholes	Low	Acknowledged
QSP-12	call injection	Low	Acknowledged
QSP-13	Access control	Low	Acknowledged
QSP-14	Replay attack	Low	Acknowledged
QSP-15	Address initialization problem	Low	Resolved
QSP-16	Judgment function problem	Low	Resolved
QSP-17	Balance judgment	Low	Resolved
QSP-18	Transfer function problem	Low	Acknowledged
QSP-19	Design issues for external calls to code	Low	Acknowledged
QSP-20	Error handling	Low	Resolved
QSP-21	Weakly random number problem	Low	Acknowledged
QSP-22	Variable coverage problem	Low	Acknowledged
QSP-23	Syntactic property problem	Low	Acknowledged
QSP-24	Data privacy issues	Low	Resolved
QSP-25	Data reliability	Low	Acknowledged
QSP-26	Gas consumption optimization	Medium Risk	Acknowledged
QSP-27	Contract user	Low	Acknowledged
QSP-28	logging	Medium Risk	Acknowledged
QSP-29	The callback function	Low	Acknowledged
QSP-30	Owner permission problem	Low	Resolved
QSP-31	User authentication problem	Low	Acknowledged
QSP-32	Conditional competition problem	Low	Acknowledged
QSP-33	Uninitialized storage pointer	Low	Acknowledged
QSP-34	Not added to get ETH balance in contract	? Undetermined	Acknowledged
QSP-35	Problem of parameters in tokenRecipient function	? Undetermined	Unresolved

Assessment

Findings

QSP-1 coding specification issues

i. Compiler version

Severity: Low Risk

Status: Resolved

Description: The contract has been followed, pragma solidity >=0.4.22 <0.6.0; https://etherscan.io/solcbuginfo

ii. Constructor writing problem

Severity: *Low Risk*

Status: Resolved

Description: The contract has been followed.

```
constructor(
        uint256 initialSupply,
        string memory tokenName,
        string memory tokenSymbol
    ) public {
        totalSupply = initialSupply * 10 ** uint256(decimals);  // Update total supply with the decimal amount
        balanceOf[msg.sender] = totalSupply;                // Give the creator all initial tokens
        name = tokenName;                                   // Set the name for display purposes
        symbol = tokenSymbol;                               // Set the symbol for display purposes
    }
```

Use the correct constructor for each version of the compiler, otherwise the contract owner may change.

After version 0.4.22, the constructor keyword was introduced as a constructor declaration, but function was not required.

If not, the constructor is compiled into a normal function that can be called by anyone, causing the owner permission to be stolen and more serious consequences.

iii. Return criteria

Severity: *Low Risk*

Status: Resolved

Description: The contract has been followed. And according to the ERC20 specification, transfer and approve functions should return bool values, and return value codes should be added.

iv. Event standards

Severity: *Low Risk*

Status: Resolved

Description: The contract has been followed. And following the ERC20 specification, the transfer and approve functions are required to trigger corresponding events.

v. False recharge

Severity: *Low Risk*

Status: Resolved

Description: The contract has been followed. And in the transfer function, you need to use the require function to throw an error in the judgment of balance and transfer amount, otherwise the transaction will be wrongly judged as successful

QSP-2 Design defects

i. Condition competition for the approve authorization function

Severity: *Medium Risk*

Status: Resolved

Description: Conditional competition should be avoided in the approve function. Before modifying the allowance, change it to 0 and then to _value.

The reason for this loophole is that in order to encourage miners to dig, miners can decide which deals to package themselves. In order to get a bigger profit, miners will generally choose to package a bigger deal with gas price, rather than relying on the transaction sequence before and after.

By setting 0, you can mitigate some of the hazards of conditional contests. The contract manager can check the log to determine if a conditional contest has occurred. The greater significance of this repair is to remind users of the approve function that the operation of the function is to some extent irreversible.

```
function approve(address _spender, uint256 _value) public returns (bool success){
    allowance[msg.sender][_spender] = _value;
    return true
```

This code can lead to conditional competition.

Should be added to approve

```
require((_value == 0) || (allowance[msg.sender][_spender] == 0));
```

Change the allowance first to 0 and then to the corresponding number

ii. Cyclic Dos problem

i) Circular consumption problem

Severity: *Low Risk*

Status: Acknowledged

Description: In the contract, it is not recommended to use too many cycles

In Ethereum, each transaction consumes a certain amount of GAS, while the actual consumption is determined by the complexity of the transaction. The higher the number of cycles, the higher the complexity of the transaction, and when the maximum allowable gas consumption is exceeded, the transaction will fail.

ii) Cycle safety

Severity: *Low Risk*

Status: Acknowledged

Description: In the contract, the number of cycles should be avoided as much as possible because the attacker may use too large a loop to complete the Dos attack.

When a user needs to transfer money to multiple accounts at the same time, we need to traverse the transfer to the target account list, which can lead to a Dos attack.

In the above situation, recommending the use of descriptive funds to let a user retrieve his or her token, rather than sending it to a corresponding account, is a procedural reducing harm.

The above code, if it controls function calls, can construct huge loops that consume gas, causing Dos problems

QSP-3 Coding security issues

i. Overflow problem

i) Arithmetic overflow

Severity: *Medium Risk*

Status: Unresolved

Description: When adding, subtracting, multiplying, and dividing are called, the safeMath library should be used instead, otherwise it is easy to cause arithmetic overflow and inevitable loss.

balances[msg.sender] - _value >= 0, The judgment can be circumvented by downflow.

The usual fix is to use Openzeppelin -safeMath, but you can limit it by judging different variables, but it's hard to limit multiplications and exponents.

```
function sell(uint256 amount) public {
        address myAddress = address(this);
        require(myAddress.balance >= amount * sellPrice);     // checks if the contract has enough ether to buy
        _transfer(msg.sender, address(this), amount);          // makes the transfers
        msg.sender.transfer(amount * sellPrice);               // sends ether to the seller. It's important to do this last to avoid recursion attacks
    }
```

ii) Coin burning overflow problem

Severity: *Medium Risk*

Status: Resolved

Description: In the coinage function, upper limit should be set for totalSupply to avoid malicious coinage issuance due to arithmetic overflow and other loopholes.

```
constructor(
        uint256 initialSupply,
        string memory tokenName,
        string memory tokenSymbol
    ) public {
        totalSupply = initialSupply * 10 ** uint256(decimals);   // Update total supply with the decimal amount
        balanceOf[msg.sender] = totalSupply;                // Give the creator all initial tokens
        name = tokenName;                                   // Set the name for display purposes
        symbol = tokenSymbol;                               // Set the symbol for display purposes
    }
```

There is no restriction on totalSupply in the above code, which may cause the index arithmetic to overrun.

The correct way of writing is as follows:

```
constructor(
        string memory tokenName,
        string memory tokenSymbol
    ) public {
        totalSupply = 21000000* 10 ** uint256(decimals);  // Update total supply with the decimal
        amount
            .....
        }
```

ii. Reentrant loopholes

Severity: *Low Risk*

Status: Acknowledged

Description: Avoid using CALL to trade in smart contracts and avoid re-entry loopholes.

Intelligent contracts provide three ways to trade Ethereum: call, send and transfer. The biggest difference of CALL is that there is no restriction on GAS, while the other two will report out of gas when gas is insufficient.

There are several characteristics of reentry vulnerabilities. 1. The call function is used as the transfer function. 2. GAS is no restriction on the call function. 3. The balance is deducted after the transfer. 4. Add () to the call to execute the fallback function.

iii. call injection

Severity: *Low Risk*

Status: Acknowledged

Description: When the call function is called, strict permission control should be done, or the dead call function should be written directly.

In EVM design, if the call parameter data is 0xdeadbeef(assuming a function name) + 0x0000000000...01, in which case you call the function.

Call injection can lead to token theft, permission bypass, private functions can be called through call injection, and even some high-authority functions.
When delegatecall must call other contracts within the contract, you can use the keyword Library to ensure that the contract is stateless and non-self-destructing. By forcing the contract to be stateless, the complexity of the storage environment can be alleviated to some extent and the attacker can be prevented from attacking the contract by modifying the state.

iv. Access control

Severity: *Low Risk*

Status: Acknowledged

Description: Reasonable permissions should be set for different functions in the contract.

Check whether the functions in the contract have correctly used keywords such as public and private for visibility modification, and check whether the contract has correctly defined and used modifier to restrict access to key functions, so as to avoid problems caused by overstepping authority.

v. Replay attack

Severity: *Low Risk*

Status: Acknowledged

Description: If the requirement of delegation management is involved in the contract, attention should be paid to the non-reusability of verification to avoid replay attack.

In the asset management system, there are often cases of entrusted management in which the principal gives the assets to the agent for management and the principal pays a certain fee to the agent.This business scenario is also common in smart contracts.

QSP-4 Coding design problem

i. Address initialization problem

Severity: *Low Risk*

Status: Resolved

Description: The contract has been followed. And in functions involving addresses, it is recommended to add require(_to!= Address (0)), which can effectively avoid unnecessary loss caused by user's wrong operation or unknown error.

Because EVM entry during the compilation of contract code initializes the address 0, if developers at some address code to initialize the variable, but did not give the initial value, or the user is in when launched an operation, wrong operation not give address variable, but need to do to the variables in the code below, may lead to unnecessary security risks.

Such checks can be the simplest way to avoid unknown errors, short address attacks and other problems.

ii. Judgment function problem

Severity: *Low Risk*

Status: Resolved

Description: The contract has been followed. And to determine the condition, use the require function instead of the assert function, because assert can cause all the remaining gas to be consumed, which otherwise would behave the same way.

It is worth noting that assert does enforce consistency. For checks on fixed variables, a assert can be used to avoid unknown problems because it forces the termination of the contract and invalidates it. Under some fixed conditions, a assert is more useful.

iii. Balance judgment

Severity: *Low Risk*

Status: Resolved

Description: The contract has been followed. And Don't assume the balance is zero when the contract is created. You can force a transfer.

Be careful about writing invariants to check account balances, as an attacker can force Wei to be sent to any account, even if the Fallback function throw is not available.

An attacker can use 1Wei to create the contract and then call selfdestruct(victimAddress) to destroy it. This will force the balance to be transferred to the target without code execution and cannot be prevented.

It is worth noting that during the packaging process, an attacker can transfer money before the contract is created through conditional competition, so that the balance is not zero at the time the contract is created.

iv. Transfer function problem

Severity: *Low Risk*

Status: Acknowledged

Description: When a transaction is completed, transfer instead of Send is recommended by default.

When the target of transfer or send function is the contract, the fallback function of the contract will be called. However, when the fallback function fails, transfer will throw an error and automatically roll back, while send will return false. Therefore, it is necessary to determine the return type when using SEND, otherwise it may lead to the situation that the transfer fails but the balance decreases.

v. Design issues for external calls to code

Severity: *Low Risk*

Status: Acknowledged

Description: Use pull rather than push for external contracts.

When making an external call, there will always be a failure, intentionally or unintentionally. In order to avoid unknown losses, it is possible to change the external operation to the user to fetch.

vi. Error handling

Severity: *Low Risk*

Status: Resolved

Description: The contract has been followed. And when the contract involves methods such as call that operate at the bottom of address, do reasonable error handling.

You need to check the validity of the address before calling such a function.

vii. Weakly random number problem

Severity: *Low Risk*

Status: Acknowledged

Description: The way random Numbers are generated on a smart contract requires more consideration.

viii. Variable coverage problem

Severity: *Low Risk*

Status: Acknowledged

Description: Avoiding the Array variable key in the contract can be controlled.

QSP-5 Potential coding problems

i. Syntactic property problem

Severity: *Low Risk*

Status: Acknowledged

Description: Take care of the down rounding problem of integer division in intelligent contract.

In a smart contract, all integer divisions are rounded down to the nearest integer, and when we need more precision, we need to use a multiplier to increase this number.

If the problem appears explicitly in the code, the compiler will issue a problem warning and will not be able to continue compiling, but if it does, it will be rounded down.

ii. Data privacy issues

Severity: *Low Risk*

Status: Resolved

Description: The contract has been followed.

Note that all data on the chain is public。

In the contract, all data, including private variables, are public, and no private data can be stored on the chain.

iii. Data reliability

Severity: *Low Risk*

Status: Acknowledged

Description: The contract should not include the timestamp in the code because it is vulnerable to interference by miners, and should use constant data such as block.Height.

iv. Gas consumption optimization

Severity: *Medium Risk*

Status: Acknowledged

Description: For some functions and variables that do not involve state change, constant can be added to avoid the consumption of gas.

Try to avoid duplicate variable definitions. The following variables have been re-assigned in the constructor.

```
using SafeMath for uint;
    // Public variables of the token
    string public name = ”SUPER ID CHAIN”;
    string public symbol = ”SIDC”;
    uint8 public decimals = 6;
    // 18 decimals is the strongly suggested default, avoid changing it
    uint256 public totalSupply = 60000 * 10 ** uint256(decimals);
```

v. Contract user

Severity: *Low Risk*

Status: Acknowledged

Description: In the contract, the transaction target should be considered as far as possible when the contract, to avoid all kinds of malicious use。

vi. logging

Severity: *Medium Risk*

Status: Acknowledged

Description: For the convenience of operation and maintenance monitoring, in addition to transfer, authorization and other

functions, other operations also need to add detailed Event records, such as transfer of administrator privileges and other special main functions.

```
function transferOwnership(address newOwner) onlyOwner public {
  require(newOwner != address(0));
  owner = newOwner;
    emit OwnershipTransferred(owner, newowner);
}
```

vii. The callback function

Severity: *Low Risk*

Status: Acknowledged

Description: The Fallback function is defined in the contract and made as simple as possible.

Fallback is called when there is a problem with the execution of the contract (such as when there is no matching function), and when the send or transfer function is called, only 2300GAS is used for Fallback function after failure. 2300 GAS is only allowed to execute a set of bytecode instructions, which needs to be written carefully so as not to run out of GAS.

viii. Owner permission problem

Severity: *Low Risk*

Status: Resolved

Description: The contract has been followed. **Avoid excessive owner permissions.**

The owner has too much authority in some contracts, so he can operate all kinds of data in the contracts at will, including modifying rules, transferring money at will, casting and burning COINS at will. Once security problems occur, serious results may be caused.

As for the authority of the owner, several requirements should be followed: 1. After the contract is created, no one can change the contract rules, including the size of the rule parameters. 2. Only the owner is allowed to withdraw the balance from the contract.

ix. User authentication problem

Severity: *Low Risk*

Status: Acknowledged

Description: Do not use tx.origin as an authentication right in your contract.

tx.origin represents the initial address. If user A calls contract C through contract B, for contract C, tx.origin is user A and msg.sender is contract B. For authentication, this is very dangerous, and it represents a possible phishing attack.

x. Conditional competition problem

Severity: *Low Risk*

Status: Acknowledged

Description: Avoid reliance on the order of transactions in the contract.

In smart contracts, it is often easy to rely on the order of transactions, such as the King rule or the last winner rule. These are design rules that are strongly dependent on the transaction order, but the underlying rules of Ethereum itself are based on the maximization of miners' interests. In a certain limit case, as long as the attacker pays enough price, he can control the transaction order to a certain extent. Developers should avoid this problem.

xi. Uninitialized storage pointer

Severity: *Low Risk*

Status: Acknowledged

Description: Avoid initializing struct variables in functions.

A special data structure is allowed to be struct in solidity, and local variables inside the function are stored in storage or memory by default.

Existing in storage and memory, which are two different concepts, solidity allows a pointer to an uninitialized reference, while an uninitialized local stroage would cause variables to point to other storage variables, causing variable overwrites, or something more serious.

QSP-6 Others

i. Not added to get ETH balance in contract

Severity: *Undetermined Risk*

Status: Acknowledged

Description: This problem may result in user error driving into the ETH irrevocably.

Recommendation: Add the appropriate extract ETH function to avoid irreversible situations.

ii. Problem of parameters in tokenRecipient function

Severity: *Undetermined Risk*

Status: Unresolved

Description:

```
interface tokenRecipient { function receiveApproval(address _from, uint256 _value, address _token, bytes _extraData) external; }
```

The above function may cause some version compilations to fail.

The data location must be ”calldata” for the parameter in the external function, but the above _extraData does not provide it.

```
interface tokenRecipient { function receiveApproval(address _from, uint256 _value, address _token, bytes calldata _extraData) external; }
```

Audit result

The SIDC intelligent contract has no vulnerabilities to attack and is relatively secure.We will follow up the contract upgrade and the follow-up test plan.

About Quantstamp

Quantstamp is a Y Combinator-backed company that helps to secure smart contracts at scale using computer-aided reasoning tools, with a mission to help boost adoption of this exponentially growing technology.

Quantstamp's team boasts decades of combined experience in formal verification, static analysis, and software verification. Collectively, our individuals have over 500 Google scholar citations and numerous published papers. In its mission to proliferate development and adoption of blockchain applications, Quantstamp is also developing a new protocol for smart contract verification to help smart contract developers and projects worldwide to perform cost-effective smart contract security audits.

To date, Quantstamp has helped to secure hundreds of millions of dollars of transaction value in smart contracts and has assisted dozens of blockchain projects globally with its white glove security auditing services. As an evangelist of the blockchain ecosystem, Quantstamp assists core infrastructure projects and leading community initiatives such as the Ethereum Community Fund to expedite the adoption of blockchain technology.

Finally, Quantstamp's dedication to research and development in the form of collaborations with leading academic institutions such as National University of Singapore and MIT (Massachusetts Institute of Technology) reflects Quantstamp's commitment to enable world-class smart contract innovation.