

# Smart Contract Audit Report

Security status

**Safe**



Principal tester: Knownsec blockchain security team

## Version Summary

Content	Date	Version
Editing Document	2021/09/03	V1.0

## Report Information

Title	Version	Document Number	Type
<b>Gladiator Smart Contract Audit Report</b>	V1.0	f80543fc97904895ab36d47883d 4237a c9648fba980a447ea3ee17f547f 25406	Open to project team

## Copyright Notice

Knownsec only issues this report for facts that have occurred or existed before the issuance of this report, and assumes corresponding responsibilities for this.

Knownsec is unable to determine the security status of its smart contracts and is not responsible for the facts that will occur or exist in the future. The security audit analysis and other content made in this report are only based on the documents and information provided to us by the information provider as of the time this report is issued. Knownsec's assumption: There is no missing, tampered, deleted or concealed information. If the information provided is missing, tampered with, deleted, concealed or reflected in the actual situation, Knownsec shall not be liable for any losses and adverse effects caused thereby.

## Table of Contents

<b>1. Introduction .....</b>	<b>- 6 -</b>
<b>2. Code vulnerability analysis .....</b>	<b>- 8 -</b>
2.1 Vulnerability Level Distribution .....	- 8 -
2.2 Audit Result .....	- 9 -
<b>3. Analysis of code audit results .....</b>	<b>- 12 -</b>
3.1. Token Function 【PASS】 .....	- 12 -
3.2. Transfer function 【PASS】 .....	- 13 -
<b>4. Basic code vulnerability detection .....</b>	<b>- 15 -</b>
4.1. Compiler version security 【PASS】 .....	- 15 -
4.2. Redundant code 【PASS】 .....	- 15 -
4.3. Use of safe arithmetic library 【PASS】 .....	- 15 -
4.4. Not recommended encoding 【PASS】 .....	- 16 -
4.5. Reasonable use of require/assert 【PASS】 .....	- 16 -
4.6. Fallback function safety 【PASS】 .....	- 16 -
4.7. tx.origin authentication 【PASS】 .....	- 17 -
4.8. Owner permission control 【PASS】 .....	- 17 -
4.9. Gas consumption detection 【PASS】 .....	- 17 -
4.10. call injection attack 【PASS】 .....	- 18 -
4.11. Low-level function safety 【PASS】 .....	- 18 -
4.12. Vulnerability of additional token issuance 【PASS】 .....	- 18 -
4.13. Access control defect detection 【PASS】 .....	- 19 -

4.14.	Numerical overflow detection 【PASS】 .....	- 19 -
4.15.	Arithmetic accuracy error 【PASS】 .....	- 20 -
4.16.	Incorrect use of random numbers 【PASS】 .....	- 20 -
4.17.	Unsafe interface usage 【PASS】 .....	- 21 -
4.18.	Variable coverage 【PASS】 .....	- 21 -
4.19.	Uninitialized storage pointer 【PASS】 .....	- 22 -
4.20.	Return value call verification 【PASS】 .....	- 22 -
4.21.	Transaction order dependency 【PASS】 .....	- 23 -
4.22.	Timestamp dependency attack 【PASS】 .....	- 24 -
4.23.	Denial of service attack 【PASS】 .....	- 24 -
4.24.	Fake recharge vulnerability 【PASS】 .....	- 25 -
4.25.	Reentry attack detection 【PASS】 .....	- 25 -
4.26.	Replay attack detection 【PASS】 .....	- 26 -
4.27.	Rearrangement attack detection 【PASS】 .....	- 26 -
<b>5.</b>	<b>Appendix A: Vulnerability rating standard .....</b>	<b>- 27 -</b>
<b>6.</b>	<b>Appendix B: Introduction to auditing tools .....</b>	<b>- 28 -</b>
6.1.	Manticore.....	- 28 -
6.2.	Oyente .....	- 28 -
6.3.	securify.sh.....	- 28 -
6.4.	Echidna.....	- 29 -
6.5.	MAIAN .....	- 29 -
6.6.	ethersplay.....	- 29 -

6.7.	ida-evm.....	- 29 -
6.8.	Remix-ide .....	- 29 -
6.9.	Knownsec Penetration Tester Special Toolkit .....	- 30 -

Knownsec

# 1. Introduction

The effective test time of this report is from **September 1, 2021** to **September 3, 2021**. During this period, the security and standardization of **the smart contract code of the Gladiator** will be audited and used as the statistical basis for the report.

The scope of this smart contract security audit does not include external contract calls, new attack methods that may appear in the future, and code after contract upgrades or tampering. (With the development of the project, the smart contract may add a new pool, New functional modules, new external contract calls, etc.), does not include front-end security and server security.

In this audit report, engineers conducted a comprehensive analysis of the common vulnerabilities of smart contracts (Chapter 3). **The smart contract code of the Gladiator** is comprehensively assessed as **SAFE**.

**Results of this smart contract security audit: SAFE**

Since the testing is under non-production environment, all codes are the latest version. In addition, the testing process is communicated with the relevant engineer, and testing operations are carried out under the controllable operational risk to avoid production during the testing process, such as: Operational risk, code security risk.

## Report information of this audit:

### Report Number:

f80543fc97904895ab36d47883d4237a (OEC)

c9648fba980a447ea3ee17f547f25406 (ETH)

### Report query address link:

<https://attest.im/attestation/searchResult?qurey=f80543fc97904895ab36d47883d4237a>

<https://attest.im/attestation/searchResult?qurey=c9648fba980a447ea3ee17f547f25406>

## Target information of the Gladiator audit:

Target information	
Project name	Crypto Gladiator Shards(CGS)
Contract address	0x81fDe2721F556e402296B2a57E1871637c27D5E8
Code type	Token code, Ethereum smart contract code, OKExChain smart contract code
Code language	Solidity

#### Contract documents and hash:

Contract documents	MD5
CGSToken. sol	6d4e27fd6d8d9b7457bc2cf13266da45

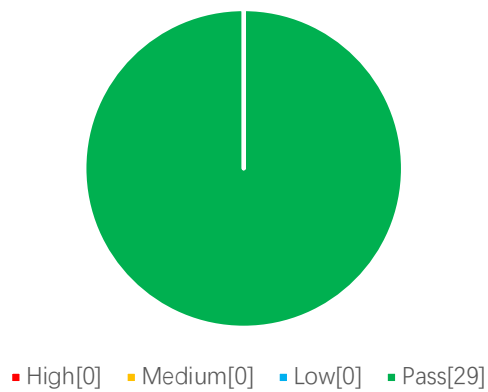
## 2. Code vulnerability analysis

### 2.1 Vulnerability Level Distribution

Vulnerability risk statistics by level:

Vulnerability risk level statistics table			
High	Medium	Low	Pass
0	0	0	29

Risk level distribution





## 2.2 Audit Result

Result of audit			
Audit Target	Audit	Status	Audit Description
Business security testing	Token function	Pass	After testing, there is no such safety vulnerability.
	Transfer function	Pass	After testing, there is no such safety vulnerability.
Basic code vulnerability detection	Compiler version security	Pass	After testing, there is no such safety vulnerability.
	Redundant code	Pass	After testing, there is no such safety vulnerability.
	Use of safe arithmetic library	Pass	After testing, there is no such safety vulnerability.
	Not recommended encoding	Pass	After testing, there is no such safety vulnerability.
	Reasonable use of require/assert	Pass	After testing, there is no such safety vulnerability.
	fallback function safety	Pass	After testing, there is no such safety vulnerability.
	tx.origin authentication	Pass	After testing, there is no such safety vulnerability.
	Owner permission control	Pass	After testing, there is no such safety vulnerability.
	Gas consumption detection	Pass	After testing, there is no such safety vulnerability.

	<b>call injection attack</b>	<b>Pass</b>	After testing, there is no such safety vulnerability.
	<b>Low-level function safety</b>	<b>Pass</b>	After testing, there is no such safety vulnerability.
	<b>Vulnerability of additional token issuance</b>	<b>Pass</b>	After testing, there is no such safety vulnerability.
	<b>Access control defect detection</b>	<b>Pass</b>	After testing, there is no such safety vulnerability.
	<b>Numerical overflow detection</b>	<b>Pass</b>	After testing, there is no such safety vulnerability.
	<b>Arithmetic accuracy error</b>	<b>Pass</b>	After testing, there is no such safety vulnerability.
	<b>Wrong use of random number detection</b>	<b>Pass</b>	After testing, there is no such safety vulnerability.
	<b>Unsafe interface use</b>	<b>Pass</b>	After testing, there is no such safety vulnerability.
	<b>Variable coverage</b>	<b>Pass</b>	After testing, there is no such safety vulnerability.
	<b>Uninitialized storage pointer</b>	<b>Pass</b>	After testing, there is no such safety vulnerability.
	<b>Return value call verification</b>	<b>Pass</b>	After testing, there is no such safety vulnerability.
	<b>Transaction order dependency detection</b>	<b>Pass</b>	After testing, there is no such safety vulnerability.
	<b>Timestamp dependent attack</b>	<b>Pass</b>	After testing, there is no such safety vulnerability.
	<b>Denial of service attack detection</b>	<b>Pass</b>	After testing, there is no such safety vulnerability.

	<b>Fake recharge vulnerability detection</b>	Pass	After testing, there is no such safety vulnerability.
	<b>Reentry attack detection</b>	Pass	After testing, there is no such safety vulnerability.
	<b>Replay attack detection</b>	Pass	After testing, there is no such safety vulnerability.
	<b>Rearrangement attack detection</b>	Pass	After testing, there is no such safety vulnerability.

### 3. Analysis of code audit results

#### 3.1. Token Function **【PASS】**

**Audit analysis:** The contract token function is implemented by the CGSToken contract class, inherited from ERC20, and conforms to the ERC20 token standard.

```
contract CGSToken is StandardToken { //knownsec// CGS token contract

    /* Public variables of the token */
    string public name; //Name: eg Simon Bucks
    uint8 public decimals; // The largest number of decimal places , How many
decimals to show. ie. There could 1000 base units with 3 decimals. Meaning 0.980 SBX = 980 base
units. It's like comparing 1 wei to 1 ether.
    string public symbol; //token abbreviation: eg SBX
    string public version = 'H0.1'; //Version

    function CGSToken(uint256 _initialAmount, string _tokenName, uint8 _decimalUnits, string
    _tokenSymbol) {
        balances[msg.sender] = _initialAmount; // The initial number of tokens is given to the
message sender
        totalSupply = _initialAmount; // Set initial total
        name = _tokenName; // token name
        decimals = _decimalUnits; // Decimal places
        symbol = _tokenSymbol; // token abbreviation
    }

    /* Approves and then calls the receiving contract */

    function approveAndCall(address _spender, uint256 _value, bytes _extraData) returns (bool
    success) {
        allowed[msg.sender][_spender] = _value;
        Approval(msg.sender, _spender, _value);
        //call the receiveApproval function on the contract you want to be notified. This crafts the

```

function signature manually so one doesn't have to include a contract in here just for this.

```
//receiveApproval(address _from, uint256 _value, address _tokenContract, bytes
_extraData)
```

*//it is assumed that when does this that the call \*should\* succeed, otherwise one would use vanilla approve instead.*

```
require(_spender.call(bytes4(bytes32(sha3("receiveApproval(address,uint256,address,bytes)"))),
msg.sender, _value, this, _extraData));

return true;

}

}
```

**Recommendation:** nothing.

## 3.2. Transfer function **【PASS】**

**Audit analysis:** The transfer function of the contract is jointly implemented by the transfer and transferFrom functions for the transfer of tokens.

```
function transfer(address _to, uint256 _value) returns (bool success) {//knownsec// Transfer

    // The default totalSupply will not exceed the maximum value (2^256-1).

    // If new tokens will be generated over time, you can use the following sentence to avoid
overflow exceptions

    require(balances[msg.sender] >= _value && balances[_to] + _value > balances[_to]);
    //require(balances[msg.sender] >= _value);

    balances[msg.sender] -= _value;// Subtract the number of tokens _value from the account of
the message sender

    balances[_to] += _value;// Add the number of tokens to the receiving account _value

    Transfer(msg.sender, _to, _value);// Trigger a currency transfer transaction event

    return true;

}
```

```
function transferFrom(address _from, address _to, uint256 _value) returns
(bool success) {//knownsec// Authorized transfer

    require(balances[_from] >= _value && allowed[_from][msg.sender] >=
        _value && balances[_to] + _value > balances[_to]);
    //require(balances[_from] >= _value && allowed[_from][msg.sender] >= _value);
    balances[_to] += _value;// The receiving account increases the number of tokens _value
    balances[_from] -= _value; // Spending account _from minus the number of tokens _value
    allowed[_from][msg.sender] -= _value;// The amount that the message sender can transfer
    from the account _from is reduced by _value
    Transfer(_from, _to, _value);// Trigger a currency transfer transaction event
    return true;
}
```

**Recommendation:** nothing.

## 4. Basic code vulnerability detection

---

### 4.1. Compiler version security **【PASS】**

Check whether a safe compiler version is used in the contract code implementation.

**Audit result:** After testing, the compiler version ^0.4.8 is formulated in the smart contract code, and there is no security issue.

**Recommendation:** nothing.

### 4.2. Redundant code **【PASS】**

Check whether the contract code implementation contains redundant code.

**Audit result:** After testing, the security problem does not exist in the smart contract code.

**Recommendation:** nothing.

### 4.3. Use of safe arithmetic library **【PASS】**

Check whether the SafeMath safe arithmetic library is used in the contract code implementation.

**Audit result:** After testing, the SafeMath safe arithmetic library is not used in the smart contract code, but there is no arithmetic overflow, so it is passed.

**Recommendation:** nothing.

#### 4.4. Not recommended encoding **【PASS】**

Check whether there is an encoding method that is not officially recommended or abandoned in the contract code implementation

**Audit result:** After testing, the security problem does not exist in the smart contract code.

**Recommendation:** nothing.

#### 4.5. Reasonable use of require/assert **【PASS】**

Check the rationality of the use of require and assert statements in the contract code implementation.

**Audit result:** After testing, the security problem does not exist in the smart contract code.

**Recommendation:** nothing.

#### 4.6. Fallback function safety **【PASS】**

Check whether the fallback function is used correctly in the contract code implementation.

**Audit result:** After testing, the security problem does not exist in the smart contract code.

**Recommendation:** nothing.



#### 4.7. tx.origin authentication **【PASS】**

tx.origin is a global variable of Solidity that traverses the entire call stack and returns the address of the account that originally sent the call (or transaction). Using this variable for authentication in a smart contract makes the contract vulnerable to attacks like phishing.

**Audit result:** After testing, the security problem does not exist in the smart contract code.

**Recommendation:** nothing.

#### 4.8. Owner permission control **【PASS】**

Check whether the owner in the contract code implementation has excessive authority. For example, arbitrarily modify other account balances, etc.

**Audit result:** After testing, the security problem does not exist in the smart contract code.

**Recommendation:** nothing.

#### 4.9. Gas consumption detection **【PASS】**

Check whether the consumption of gas exceeds the maximum block limit.

**Audit result:** After testing, the security problem does not exist in the smart contract code.

**Recommendation:** nothing.

#### 4.10. call injection attack **【PASS】**

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

**Audit result:** After testing, the smart contract does not have this vulnerability.

**Recommendation:** nothing.

#### 4.11. Low-level function safety **【PASS】**

Check whether there are security vulnerabilities in the use of low-level functions (call/delegatecall) in the contract code implementation

The execution context of the call function is in the called contract; the execution context of the delegatecall function is in the contract that currently calls the function.

**Audit result:** After testing, the security problem does not exist in the smart contract code.

**Recommendation:** nothing.

#### 4.12. Vulnerability of additional token issuance **【PASS】**

Check whether there is a function that may increase the total amount of tokens in the token contract after initializing the total amount of tokens.

**Audit result:** After testing, the security problem does not exist in the smart contract code.

**Recommendation:** nothing.

#### 4.13. Access control defect detection **【PASS】**

Different functions in the contract should set reasonable permissions.

Check whether each function in the contract correctly uses keywords such as public and private for visibility modification, check whether the contract is correctly defined and use modifier to restrict access to key functions to avoid problems caused by unauthorized access.

**Audit result:** After testing, the security problem does not exist in the smart contract code.

**Recommendation:** nothing.

#### 4.14. Numerical overflow detection **【PASS】**

The arithmetic problems in smart contracts refer to integer overflow and integer underflow.

Solidity can handle up to 256-bit numbers ( $2^{256}-1$ ). If the maximum number increases by 1, it will overflow to 0. Similarly, when the number is an unsigned type, 0 minus 1 will underflow to get the maximum digital value.

Integer overflow and underflow are not a new type of vulnerability, but they are especially dangerous in smart contracts. Overflow conditions can lead to incorrect results, especially if the possibility is not expected, which may affect the reliability and safety of the program.

**Audit result:** After testing, the security problem does not exist in the smart contract code.

**Recommendation:** nothing.

#### 4.15. Arithmetic accuracy error **【PASS】**

As a programming language, Solidity has data structure design similar to ordinary programming languages, such as variables, constants, functions, arrays, functions, structures, etc. There is also a big difference between Solidity and ordinary programming languages-Solidity does not float Point type, and all the numerical calculation results of Solidity will only be integers, there will be no decimals, and it is not allowed to define decimal type data. Numerical calculations in the contract are indispensable, and the design of numerical calculations may cause relative errors. For example, the same level of calculations:  $5/2*10=20$ , and  $5*10/2=25$ , resulting in errors, which are larger in data. The error will be larger and more obvious.

**Audit result:** After testing, the security problem does not exist in the smart contract code.

**Recommendation:** nothing.

#### 4.16. Incorrect use of random numbers **【PASS】**

Smart contracts may need to use random numbers. Although the functions and variables provided by Solidity can access values that are obviously unpredictable, such as `block.number` and `block.timestamp`, they are usually more public than they

appear or are affected by miners. These random numbers are predictable to a certain extent, so malicious users can usually copy it and rely on its unpredictability to attack the function.

**Audit result:** After testing, the security problem does not exist in the smart contract code.

**Recommendation:** nothing.

#### 4.17. Unsafe interface usage **【PASS】**

Check whether unsafe interfaces are used in the contract code implementation.

**Audit result:** After testing, the security problem does not exist in the smart contract code.

**Recommendation:** nothing.

#### 4.18. Variable coverage **【PASS】**

Check whether there are security issues caused by variable coverage in the contract code implementation.

**Audit result:** After testing, the security problem does not exist in the smart contract code.

**Recommendation:** nothing.

#### 4.19. Uninitialized storage pointer **【PASS】**

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

The existence of storage (memory) and memory (memory) are two different concepts. Solidity allows pointers to point to an uninitialized reference, while uninitialized local storage will cause variables to point to other storage variables, leading to variable coverage, or even more serious As a consequence, you should avoid initializing struct variables in functions during development.

**Audit result:** After testing, the security problem does not exist in the smart contract code.

**Recommendation:** nothing.

#### 4.20. Return value call verification **【PASS】**

This problem mostly occurs in smart contracts related to currency transfer, so it is also called silent failed delivery or unchecked delivery.

In Solidity, there are transfer(), send(), call.value() and other currency transfer methods, which can all be used to send ETH to an address. The difference is: When the transfer fails, it will be thrown and the state will be rolled back; Only 2300gas will be passed for calling to prevent reentry attacks; false will be returned when send fails; only 2300gas will be passed for calling to prevent reentry attacks; false will be returned when call.value fails to be sent; all available gas will be passed for calling

(can be Limit by passing in gas\_value parameters), which cannot effectively prevent reentry attacks.

If the return value of the above send and call.value transfer functions is not checked in the code, the contract will continue to execute the following code, which may lead to unexpected results due to ETH sending failure.

**Audit result:** After testing, the security problem does not exist in the smart contract code.

**Recommendation:** nothing.

#### 4.21. Transaction order dependency **【PASS】**

Since miners always get gas fees through codes that represent externally owned addresses (EOA), users can specify higher fees for faster transactions. Since the Ethereum blockchain is public, everyone can see the content of other people's pending transactions. This means that if a user submits a valuable solution, a malicious user can steal the solution and copy its transaction at a higher fee to preempt the original solution.

**Audit result:** After testing, the security problem does not exist in the smart contract code.

**Recommendation:** nothing.

## 4.22. Timestamp dependency attack **【PASS】**

The timestamp of the data block usually uses the local time of the miner, and this time can fluctuate in the range of about 900 seconds. When other nodes accept a new block, it only needs to verify whether the timestamp is later than the previous block and The error with local time is within 900 seconds. A miner can profit from it by setting the timestamp of the block to satisfy the conditions that are beneficial to him as much as possible.

Check whether there are key functions that depend on the timestamp in the contract code implementation.

**Audit result:** After testing, the security problem does not exist in the smart contract code.

**Recommendation:** nothing.

## 4.23. Denial of service attack **【PASS】**

In the world of Ethereum, denial of service is fatal, and a smart contract that has suffered this type of attack may never be able to return to its normal working state.

There may be many reasons for the denial of service of the smart contract, including malicious behavior as the transaction recipient, artificially increasing the gas required for computing functions to cause gas exhaustion, abusing access control to access the private component of the smart contract, using confusion and negligence, etc. Wait.



**Audit result:** After testing, the security problem does not exist in the smart contract code.

**Recommendation:** nothing.

#### 4.24. Fake recharge vulnerability **【PASS】**

The transfer function of the token contract uses the if judgment method to check the balance of the transfer initiator (msg.sender). When balances[msg.sender] <value, enter the else logic part and return false, and finally no exception is thrown. We believe that only if/else this kind of gentle judgment method is an imprecise coding method in sensitive function scenarios such as transfer.

**Audit result:** After testing, the security problem does not exist in the smart contract code.

**Recommendation:** nothing.

#### 4.25. Reentry attack detection **【PASS】**

The **call.value()** function in Solidity consumes all the gas it receives when it is used to send ETH. When the **call.value()** function to send ETH occurs before the actual reduction of the sender's account balance, There is a risk of reentry attacks.

**Audit results:** After testing, the security problem does not exist in the smart contract code.

**Recommendation:** nothing.

## 4.26. Replay attack detection **【PASS】**

If the contract involves the need for entrusted management, attention should be paid to the non-reusability of verification to avoid replay attacks

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

**Audit results:** After testing, the smart contract does not have this vulnerability.

**Recommendation:** nothing.

## 4.27. Rearrangement attack detection **【PASS】**

A rearrangement attack refers to a miner or other party trying to "compete" with smart contract participants by inserting their own information into a list or mapping, so that the attacker has the opportunity to store their own information in the contract. in.

**Audit results:** After testing, the security problem does not exist in the smart contract code.

**Recommendation:** nothing.

## 5. Appendix A: Vulnerability rating standard

<i>Smart contract vulnerability rating standards</i>	
Level	Level Description
<b>High</b>	<p>Vulnerabilities that can directly cause the loss of token contracts or user funds, such as: value overflow loopholes that can cause the value of tokens to zero, fake recharge loopholes that can cause exchanges to lose tokens, and can cause contract accounts to lose ETH or tokens. Access loopholes, etc.;</p> <p>Vulnerabilities that can cause loss of ownership of token contracts, such as: access control defects of key functions, call injection leading to bypassing of access control of key functions, etc.;</p> <p>Vulnerabilities that can cause the token contract to not work properly, such as: denial of service vulnerability caused by sending ETH to malicious addresses, and denial of service vulnerability caused by exhaustion of gas.</p>
<b>Medium</b>	<p>High-risk vulnerabilities that require specific addresses to trigger, such as value overflow vulnerabilities that can be triggered by token contract owners; access control defects for non-critical functions, and logical design defects that cannot cause direct capital losses, etc.</p>
<b>Low</b>	<p>Vulnerabilities that are difficult to be triggered, vulnerabilities with limited damage after triggering, such as value overflow vulnerabilities that require a large amount of ETH or tokens to trigger, vulnerabilities where attackers cannot directly profit after triggering value overflow, and the transaction sequence triggered by specifying high gas depends on the risk.</p>

## 6. Appendix B: Introduction to auditing tools

---

### 6.1. Manticore

Manticore is a symbolic execution tool for analyzing binary files and smart contracts. Manticore includes a symbolic Ethereum Virtual Machine (EVM), an EVM disassembler/assembler and a convenient interface for automatic compilation and analysis of Solidity. It also integrates Ethersplay, Bit of Traits of Bits visual disassembler for EVM bytecode, used for visual analysis. Like binary files, Manticore provides a simple command line interface and a Python for analyzing EVM bytecode API.

### 6.2. Oyente

Oyente is a smart contract analysis tool. Oyente can be used to detect common bugs in smart contracts, such as reentrancy, transaction sequencing dependencies, etc. More convenient, Oyente's design is modular, so this allows advanced users to implement and Insert their own detection logic to check the custom attributes in their contract.

### 6.3. securify.sh

Securify can verify common security issues of Ethereum smart contracts, such as disordered transactions and lack of input verification. It analyzes all possible execution paths of the program while fully automated. In addition, Securify also has a

specific language for specifying vulnerabilities, which makes Securify can keep an eye on current security and other reliability issues at any time.

## 6.4. Echidna

Echidna is a Haskell library designed for fuzzing EVM code.

## 6.5. MAIAN

MAIAN is an automated tool for finding vulnerabilities in Ethereum smart contracts. Maian processes the bytecode of the contract and tries to establish a series of transactions to find and confirm the error.

## 6.6. ethersplay

ethersplay is an EVM disassembler, which contains relevant analysis tools.

## 6.7. ida-evm

ida-evm is an IDA processor module for the Ethereum Virtual Machine (EVM).

## 6.8. Remix-ide

ida-evm is an IDA processor module for the Ethereum Virtual Machine (EVM).

## 6.9. Knownsec Penetration Tester Special Toolkit

Pen-Tester tools collection is created by KnownSec team. It contains plenty of Pen-Testing tools such as automatic testing tool, scripting tool, Self-developed tools etc.

Knownsec



Beijing KnownSec Information Technology Co., Ltd.

Advisory telephone +86(10)400 060 9587

E-mail [sec@knownsec.com](mailto:sec@knownsec.com)

Website [www.knownsec.com](http://www.knownsec.com)

Address wangjing soho T2-B2509,Chaoyang District, Beijing