Contract Audit
Smart Fast

Smart Contract
Security Audit Report

Number: 0506134615934

Date: 2021-05-06

Welcome to SmartFast!

0x01 Summary Information

The SmartFast (SF, for short) platform received this smart contract security audit application and audited the contract in May 2021.

It is necessary to declare that SF only issues this report in respect of facts that have occurred or existed before the issuance of this report, and undertakes corresponding responsibilities for this. For the facts that occur or exist in the future, SF is unable to judge the security status of its smart contract, and will not be responsible for it. The security audit analysis and other content made in this report are based on the documents and information provided to smart analysis team by the information provider as of the issuance of this report (referred to as "provided information"). SF hypothesis: There is no missing, tampered, deleted or concealed information in the mentioned information. If the information that has been mentioned is missing, tampered with, deleted, concealed or reflected does not match the actual situation, SmartFast shall not be liable for any losses and adverse effects caused thereby.

Table 1 Contract audit information

Project	Description
Contract name	Test
Contract type	Ethereum contract
Code language	Solidity
Contract files	arbitrary_send.sol
Contract address	
Auditors	Smart analysis team
Audit time	2021-05-06 13:46:15
Audit tool	SmartFast (SF)

Table 1 shows the relevant information of this contract audit in detail. The details and results of the contract security audit will be introduced in detail below.

0x02 Contract Audit Results

2.1 Vulnerability Distribution

The severity of vulnerabilities in this security audit is distributed according to the level of danger:

Table 2 Overview of contract audit vulnerability distribution

Vulnerability security distribution				
High	Medium	Low	Info	Opt
4	13	2	24	15

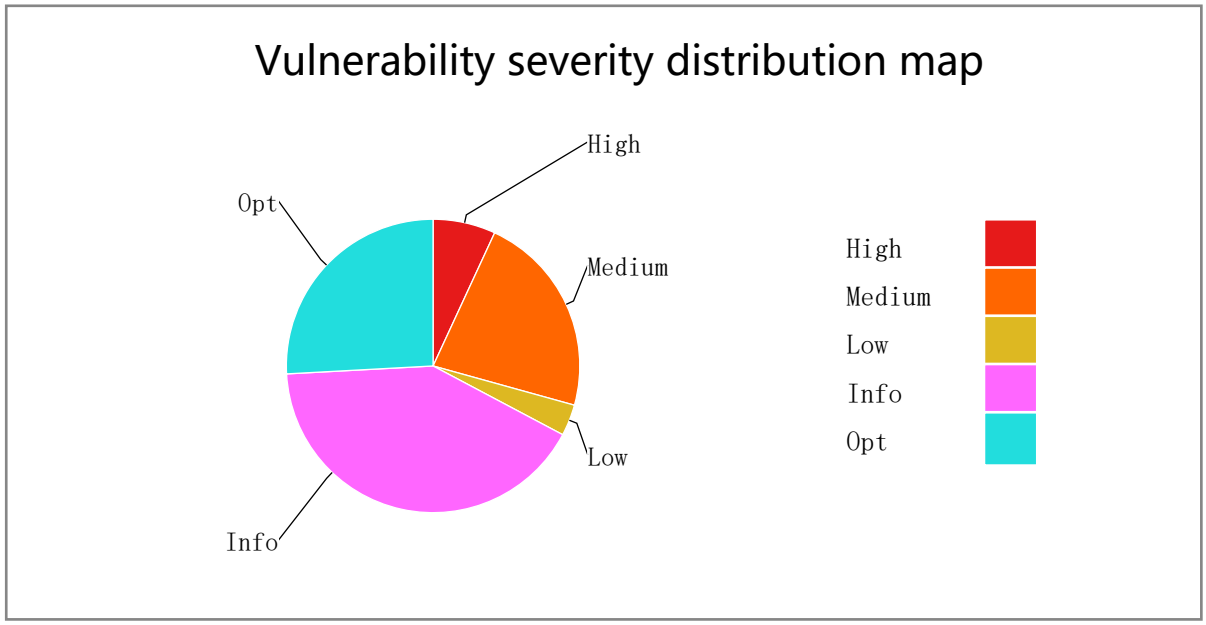

Figure 1 Vulnerability security distribution map

This security audit found 4 High-severity vulnerabilities, 13 Medium-severity vulnerabilities, 2 Low-severity vulnerabilities, 15 Optimization-severity vulnerabilities, and 24 places that need attention.

2.2 Audit Results

There are 119 test items in this security audit, and the test items are as follows (other unknown security vulnerabilities are not included in the scope of responsibility of this audit):

Table 3 Contract audit items

ID	Pattern	Description	Severity	Confidence	Status/Num
1	abiencoderv2-array	ABI encoding error	High	exactly	Pass
2	array-by-reference	Storage parameter usage	High	exactly	Pass
3	multiple-constructors	Multiple constructors in a contract	High	exactly	Pass

4	names-reused	Check contract name reuse	High	exactly	Pass
5	public-mappings-nested	Struct sets the nested structure of struct	High	exactly	Pass
6	rtlo	Overwritten from right to left	High	exactly	Pass
7	shadowing-state	Check state variable hiding	High	exactly	Pass
8	suicidal	Check if anyone can break the contract	High	exactly	Pass
9	uninitialized-state	Check for uninitialized state variables	High	exactly	Pass
10	uninitialized-storage	Check for uninitialized storage variables	High	exactly	Pass
11	unprotected-upgrade	Break the logic of the contract	High	exactly	Pass
12	visibility	Check visibility level error	High\Info	exactly	Info:1
13	redundant-fallback	Check for redundant fallback functions	High\Opt	exactly	Pass
14	arbitrary-send	Check if Ether can be sent to any address	High	probably	High:4
15	continue-in-loop	Continue causes an infinite loop	High	probably	Pass
16	controlled-array-length	Length is allocated directly	High	probably	Pass
17	controlled-delegatecall	The delegate address out of control	High	probably	Pass
18	incorrect-constructor	Check the constructor name error	High	probably	Pass
19	parity-multisig-bug	Check for multi-signature vulnerabilities	High	probably	Pass
20	reentrancy-eth	Re-entry vulnerabilities (Ethereum theft)	High	probably	Pass
21	storage-array	Signed integer array problem	High	probably	Pass
22	weak-prng	Modulo problem of parameters	High	probably	Pass
23	assert-violation	Check for incorrect use of assertions	Medium	exactly	Pass
24	constructor-return	The use of return in the constructor	Medium	exactly	Pass
25	default-return-value	Function returns only default value	Medium	exactly	Pass
26	enum-conversion	Scope of enumerated types	Medium	exactly	Pass
27	erc1155-interface	Check the wrong ERC1155 interface	Medium	exactly	Pass
28	erc1410-interface	Check the wrong ERC1410 interface	Medium	exactly	Pass
29	erc20-interface	Check for wrong ERC20 interface	Medium	exactly	Pass
30	erc223-interface	Check the wrong ERC223 interface	Medium	exactly	Pass
31	erc621-interface	Check the wrong ERC621 interface	Medium	exactly	Pass
32	erc721-interface	Check the wrong ERC721 interface	Medium	exactly	Pass
33	erc777-interface	Check the wrong ERC777 interface	Medium	exactly	Pass
34	erc875-interface	Check the wrong ERC875 interface	Medium	exactly	Pass
35	incorrect-equality	Check the strict equality of danger	Medium	exactly	Pass

36	incorrect-signature	Check for incorrect function signatures	Medium	exactly	Pass
37	locked-ether	Whether the contract ether is locked	Medium	exactly	Pass
38	mapping-deletion	Delete the mapped structure problem	Medium	exactly	Pass
39	shadowing-abstract	State variables from the abstract contract	Medium	exactly	Pass
40	tautology	Check for tautologies or contradictions	Medium	exactly	Pass
41	boolean-cst	Check for misuse of Boolean constants	Medium	probably	Pass
42	constant-function-state	Check Constant function to change state	Medium	probably	Pass
43	divide-before-multiply	Imprecise order of arithmetic operations	Medium	probably	Pass
44	erc20-approve	ERC-20 advance attack (TOD)	Medium	probably	Pass
45	function-problem	Abnormal termination of contract function	Medium	probably	Pass
46	mul-var-len-arguments	Hash collisions with multiple variable	Medium	probably	Pass
47	reentrancy-no-eth	Re-entry vulnerabilities (no ether theft)	Medium	probably	Pass
48	reused-constructor	Constructor conflict problem in the contract	Medium	probably	Pass
49	tx-origin	Check the dangerous use of tx.origin	Medium	probably	Pass
50	typographical-error	Check for writing errors (=+)	Medium	probably	Pass
51	unchecked-lowlevel	Check for uncensored low-level calls	Medium	probably	Pass
52	unchecked-send	Check unreviewed send	Medium	probably	Medium:13
53	uninitialized-local	Check for uninitialized local variables	Medium	probably	Pass
54	unused-return	Check if there is an unused return value	Medium	probably	Pass
55	writeto-arbitrarystorage	It can be written to any storage location	Medium	probably	Pass
56	costly-loop	Check for too expensive loops	Medium	possibly	Pass
57	shift-parameter-mixup	Check reversible shift operation	Medium	possibly	Pass
58	shadowing-builtin	Check the hiding of built-in symbols	Low	exactly	Pass
59	shadowing-function	Check function hiding	Low	exactly	Pass
60	shadowing-local	Check local variable hiding	Low	exactly	Pass
61	transfer-to-zeroaddress	The withdrawal address is 0x0	Low	exactly	Pass
62	uninitialized-fptr-cst	Uninitialized pointers in constructor	Low	exactly	Pass
63	variable-scope	Check the declaration of variables	Low	exactly	Pass
64	void-cst	Calls to unimplemented constructors	Low	exactly	Pass
65	incorrect-modifier	Restore abnormal Modifier	Low	exactly	Pass
66	assemblycall-rewrite	Assemblycall covers the input	Low	probably	Pass
67	block-other-parameters	Hazardous use variables (block.number etc.)	Low	probably	Pass

68	calls-loop	Check the external call in the loop	Low	probably	Pass
69	events-access	The loss of key access control parameters	Low	probably	Pass
70	events-maths	The loss of key arithmetic parameters	Low	probably	Pass
71	extcodesize-invoke	Check Extcodesize call	Low	probably	Pass
72	fllback-outofgas	The fallback function is too complicated	Low	probably	Pass
73	incorrect-blockhash	Incorrect use of Blockhash function	Low	probably	Pass
74	incorrect-inheritance-order	The inherited variables conflict	Low	probably	Pass
75	integer-overflow	Check for integer overflow	Low	probably	Low:1
76	missing-zero-check	Check the use of zero addresses	Low	probably	Low:1
77	reentrancy-benign	Reentrant vulnerabilities (continuous calls)	Low	probably	Pass
78	reentrancy-events	Reentrant vulnerabilities (events)	Low	probably	Pass
79	timestamp	The dangerous use of block.timestamp	Low	probably	Pass
80	signature-malleability	The signature contains an existing signature	Low	possibly	Pass
81	assembly	Unsafe use of assembly	Info	exactly	Pass
82	assert-state-change	Incorrect use of assert()	Info	exactly	Pass
83	delete-dynamic-arrays	The deletion of the dynamic storage array	Info	exactly	Pass
84	deprecated-standards	Solidity deprecated instructions	Info	exactly	Pass
85	erc20-indexed	ERC20 event parameter is missing indexed	Info	exactly	Pass
86	erc20-throw	ERC20 throws an exception	Info	exactly	Pass
87	length-manipulation	Unsafe operation to check array length	Info	exactly	Pass
88	low-level-calls	Check low-level calls	Info	exactly	Info:13
89	msgvalue-equals-zero	The judgment of msg.value and zero	Info	exactly	Pass
90	naming-convention	The naming follows the Solidity format	Info	exactly	Info:9
91	pragma	Undeclared multiple compiled versions	Info	exactly	Pass
92	solc-version	Check for incorrect Solidity version	Info	exactly	Info:1
93	unimplemented-functions	Functions overloaded in the contract	Info	exactly	Pass
94	upgrade-050	Code for Solidity 0.5.x upgrade	Info	exactly	Pass
95	function-init-state	State variables initialized by functions	Info	exactly	Pass
96	complex-function	Check complex functions	Info	probably	Pass
97	hardcoded	Check the legitimacy of the address	Info	probably	Pass
98	overpowered-role	The permissions are too concentrated	Info	probably	Pass
99	reentrancy-limited-events	Reentrancy vulnerabilities (limited events)	Info	probably	Pass

100	reentrancy-limited-gas	Reentry (send and transfer, with eth)	Info	probably	Pass
101	reentrancy-limited-gas-no-eth	Reentrance (send and transfer, no eth)	Info	probably	Pass
102	similar-names	Detect similar variables	Info	probably	Pass
103	too-many-digits	Too many number symbols	Info	probably	Pass
104	private-not-hidedata	Check the use of private visibility	Info	possibly	Pass
105	safemath	Check the use of SafeMath	Info	possibly	Pass
106	array-instead-bytes	The byte array can be replaced with bytes	Opt	exactly	Pass
107	boolean-equal	Check comparison with boolean constant	Opt	exactly	Pass
108	code-no-effects	Check for invalid codes	Opt	exactly	Pass
109	constable-states	State variables can be declared as constants	Opt	exactly	Pass
110	event-before-revert	Check if event is called before revert	Opt	exactly	Pass
111	external-function	Public functions can be declared as external	Opt	exactly	Opt:15
112	extra-gas-inloops	Check for additional gas consumption	Opt	exactly	Pass
113	missing-inheritance	Detect lost inheritance	Opt	exactly	Pass
114	redundant-statements	Detect the use of invalid sentences	Opt	exactly	Pass
115	return-struct	Multiple return values (struct)	Opt	exactly	Pass
116	revert-require	Check Revert in if operation	Opt	exactly	Pass
117	send-transfer	Check Transfe to replace Send	Opt	exactly	Pass
118	unused-state	Check unused state variables	Opt	exactly	Pass
119	costly-operations-loop	Expensive operations in the loop	Opt	probably	Pass

0x03 Contract Code

3.1 Code and Vulnerability Lables

In the corresponding position of each contract code, security vulnerabilities and coding specification issues have been marked in the form of comments. The comment labels start with //StFt. For details, please refer to the following contract code content.

```
contract Test{

  address destination;
  address owner;

  mapping (address => uint) balances;

  constructor(){ //StFt //visibility
    balances[msg.sender] = 0;
```

```
owner = msg.sender;
}

function direct() public{ //bad //StFt //external-function
    msg.sender.send(address(this).balance); //StFt //arbitrary-send, unchecked-send, low-level-calls
}

function init() public{ //StFt //external-function
    destination = msg.sender;
}

modifier isowner() {
address aa = msg.sender;
require(owner == aa);
_;
}

function indirect() public{ //bad //StFt //external-function
    destination.send(address(this).balance); //StFt //arbitrary-send, unchecked-send, low-level-calls
}

function directceshi_1() public{ //StFt //naming-convention, external-function
require(owner == msg.sender);
    destination.send(address(this).balance); //StFt //unchecked-send, low-level-calls
}

function directceshi_2() isowner public{ //StFt //naming-convention, external-function
    destination.send(address(this).balance); //StFt //unchecked-send, low-level-calls
}

function directceshi_3() public{ //StFt //naming-convention, external-function
    destination.send(msg.value); //StFt //unchecked-send, low-level-calls
}

function directceshi_4() public{ //StFt //naming-convention, external-function
    destination.send(balances[msg.sender]); //StFt //unchecked-send, low-level-calls
}

function directceshi_5() public{ //StFt //naming-convention, external-function
    destination.send(0); //StFt //unchecked-send, low-level-calls
}

function directceshi_6() public{ //StFt //naming-convention, external-function
uint avalue = 0;
    destination.send(avalue); //StFt //unchecked-send, low-level-calls
}

// these are legitimate calls
// and should not be detected
function repay() payable public{ //StFt //external-function
    msg.sender.transfer(msg.value);
}

function withdraw() public{ //StFt //external-function
    uint val = balances[msg.sender];
    msg.sender.send(val); //StFt //unchecked-send, low-level-calls
```

```
    }

    function withdraw_direct() public{ //StFt //naming-convention, external-function
        uint val = msg.value;
        msg.sender.send(val); //StFt //unchecked-send, low-level-calls
    }

    function withdraw_inderect() public{ //bad //StFt //naming-convention, external-function
        uint val = address(this).balance;
        msg.sender.send(val); //StFt //arbitrary-send, unchecked-send, low-level-calls
    }

    function indirect_ceshi1() public{ //bad //StFt //naming-convention, external-function
    address cc = msg.sender;
    cc = owner;
    require(cc == owner);
    address dd = msg.sender; //StFt //missing-zero-check
        uint val = address(this).balance;
        dd.send(val); //StFt //arbitrary-send, unchecked-send, low-level-calls
    }

    function buy() payable public{ //StFt //external-function
        uint value_send = msg.value;
        uint value_spent = 0 ; // simulate a buy of tokens
        uint remaining = value_send - value_spent; //StFt //integer-overflow
        msg.sender.send(remaining); //StFt //unchecked-send, low-level-calls
    }

}
```

0x04 Contract Audit Details

4.1 abiencoderv2-array

Vulnerability description

The solc 0.4.7-0.5.10 version contains a compiler error that causes the incorrect use of the ABI encoder.

Applicable scenarios

```
contract A {
    uint[2][3] bad_arr = [[1, 2], [3, 4], [5, 6]];
    /* Array of arrays passed to abi.encode is vulnerable */
    function bad() public {
        bytes memory b = abi.encode(bad_arr);
    }
}
```

When the compiler version is 0.4.7-0.5.10, because abi.encode (bad_arr) is used in bad(), calling bad() will incorrectly encode the array as [[1, 2], [2, 3], [3, 4]], and cause unexpected behavior.

Audit results:【Pass】

Security advice: none.

4.2 array-by-reference

Vulnerability description

Detect the function (reference) that passes the storage array to the storage function.

Applicable scenarios

```
contract Memory {
   uint[1] public x; // storage
   function f() public {
      f1(x); // update x
      f2(x); // do not update x
   }
   function f1(uint[1] storage arr) internal { // by reference
      arr[0] = 1;
   }
   function f2(uint[1] arr) internal { // by value
      arr[0] = 2;
   }
}
```

Bob calls f(). It stands to reason that at the end of the call, Bob x[0] is 2, but it is 1. Therefore, Bob used the contract incorrectly.

Audit results:【Pass】

Security advice: none.

4.3 multiple-constructors

Vulnerability description

Detect multiple constructor definitions in the same contract (using old and new schemes).

Applicable scenarios

```
contract A {
   uint x;
   constructor() public {
      x = 0;
   }
   function A() public {
      x = 1;
   }
   function test() public returns(uint) {
      return x;
   }
}
```

In Solidity 0.4.22, a contract with two constructor schemes will be compiled at the same time. The first constructor will take precedence over the second, which may be unexpected. Therefore, special attention should be paid to Solidity 0.4.22.

Audit results:【Pass】

Security advice: none.

4.4 names-reused

Vulnerability description

If a code base has two contracts with similar names, the compilation artifact will not contain one of the contracts with duplicate names.

Applicable scenarios

```
contract B{
   constructor() {
   }
   function ceshi1() {}
}
contract B{
   constructor() {
   }
   function ceshi2() {}
}
```

Bob's code base has two contracts called "ERC20". When the code base is running, only one of the two contracts will be compiled in "build/contracts". As a result, the second contract cannot be analyzed.

Audit results:【Pass】

Security advice: none.

4.5 public-mappings-nested

Vulnerability description

Before Solidity 0.5, public maps with nested structures (struct sets of struct) returned incorrect values.

Applicable scenarios

```
contract TestNestedStructInMapping {
   // The struct that is nested.
   struct structNested {
      uint dummy;
   }
   // The struct that holds the nested struct.
   struct structMain {
      structNested gamePaymentsSummary;
   }
   // The map that maps a game ID to a specific game.
   mapping(uint256 => structMain) public s_mapOfNestedStructs;
}
```

It does work in 0.4.25. However, adding another variable to the structure still does not make 0.4.25 error, but the generated code is wrong-it only returns 32 bytes instead of 64 bytes. Bob interacts with a publicly mapped contract with a nested

structure. The value returned by the mapping is incorrect (64 bytes are returned as 32bytes), destroying Bob's use.

Audit results:【Pass】

Security advice: none.

4.6 rtlo

Vulnerability description

Malicious actors can use Unicode characters (U+202E) overlaid from right to left to force the rendering of RTL text and enable users to achieve the true intention of confusing the contract.

Applicable scenarios

```
contract Token
{
   address payable o; // owner
   mapping(address => uint) tokens;
   function withdraw() external returns(uint)
   {
      uint amount = tokens[msg.sender];
      address payable d = msg.sender;
      token/*noitanitsed*/ d, o/*□□□
            /*value */, amount);
   }
   function _withdraw(address payable fee_receiver, address payable destination, uint value) internal
   {
      fee_receiver.transfer(1);
      destination.transfer(value);
   }
}
```

When calling `_withdraw`, `Token` uses right-to-left overwriting characters (the small box above contains overwriting characters). As a result, the fee was erroneously sent to `msg.sender`, and the token balance was sent to the owner.

Audit results:【Pass】

Security advice: none.

4.7 shadowing-state

Vulnerability description

Robustness allows ambiguous naming of state variables when using inheritance. When there are multiple definitions at the contract and functional level, state variable hiding may also occur within a single contract.

Applicable scenarios

```
contract BaseContract{
    address owner;
    modifier isOwner(){
        require(owner == msg.sender);
        _;
    }
}
contract DerivedContract is BaseContract{
    address owner;
    constructor(){
        owner = msg.sender;
    }
    function withdraw() isOwner() external{
        msg.sender.transfer(this.balance);
    }
}
```

The `owner` of `BaseContract` will not be specified, and the modifier `isOwner` has no effect. '''

Audit results:【Pass】

Security advice: none.

4.8 suicidal

Vulnerability description

Due to lack of access control or insufficient access control, malicious parties can self-destruct the contract. Calling selfdestruct/suicide lacks protection.

Applicable scenarios

```
contract Suicidal{
    function kill() public{
        selfdestruct(msg.sender);
    }
}
```

Bob calls the "kill" function and breaks the contract.

Audit results:【Pass】

Security advice: none.

4.9 uninitialized-state

Vulnerability description

Uninitialized state variables can lead to intentional or unintentional vulnerabilities.

Applicable scenarios

```
contract Uninitialized{
    address destination;
    function transfer() payable public{
```

```
        destination.transfer(msg.value);
    }
}
```

Bob calls "transfer". As a result, the ether is sent to the address "0x0" and is lost.

Audit results:【Pass】

Security advice: none.

4.10 uninitialized-storage

Vulnerability description

Local variables are not initialized. The initialized storage variable will be used as a reference to the first state variable, and key variables can be overwritten.

Applicable scenarios

```
contract Uninitialized{
    address owner = msg.sender;
    struct St{
        uint a;
    }
    function func() {
        St st;
        st.a = 0x0;
    }
}
```

Bob calls `func`. As a result, "owner" is overwritten with "0".

Audit results:【Pass】

Security advice: none.

4.11 unprotected-upgrade

Vulnerability description

Detect logical contracts that can be broken.

Applicable scenarios

```
contract Buggy is Initializable{
    address payable owner;

    function initialize() external initializer{
        require(owner == address(0));
        owner = msg.sender;
    }
    function kill() external{
        require(msg.sender == owner);
        selfdestruct(owner);
    }
}
```

Buggy is a renewable contract. Anyone can call initialize on the logic contract and destroy the contract.

Audit results:【Pass】

Security advice: none.

4.12 visibility

Vulnerability description

The default function visibility level in the contract is public, and in interfaces-external, the default visibility level of state variables is internal. In the contract, the fallback function can be external or public. In the interface, all functions should be declared as external functions. Clearly define function visibility to prevent confusion. Specifically, before version 0.5.0: look for fallback functions without explicit visibility declaration and delete them, look for fallback functions that are neither external nor public, look for constructors with external or private visibility, in the interface Find internal and private functions; after version 0.5.0: Find non-external functions in the interface.

Applicable scenarios: 【Info:1】

For this pattern, the specific problems in the contract are as follows: (Informational)constructor(){ (tests/mini_dataset/arbitrary_send.sol)#8

Security advice

The visibility should be modified according to the requirements of the compiler.

4.13 redundant-fallback

Vulnerability description

Starting from Solidity 0.4.0, contracts that do not have a rollback function will automatically restore the payment, making the payment refusal to roll back redundant. But below 0.4, there is no fallback function that is prone to reentrance loopholes, which is very dangerous.

Applicable scenarios

```
pragma solidity 0.3.24;
contract Crowdsale {
}
```

A malicious user writes a fallback function to attack the contract.

Audit results:【Pass】

Security advice: none.

4.14 arbitrary-send

Vulnerability description

The call to the function that sends Ether to an arbitrary address has not been reviewed.

Applicable scenarios: 【High:4】

For this pattern, the specific problems in the contract are as follows:

(High)Test.direct() (tests/mini_dataset/arbitrary_send.sol#13-15) sends eth to arbitrary user

Dangerous calls:

- msg.sender.send(address(this).balance) (tests/mini_dataset/arbitrary_send.sol#14)

(High)Test.indirect() (tests/mini_dataset/arbitrary_send.sol#27-29) sends eth to arbitrary user

Dangerous calls:

- destination.send(address(this).balance) (tests/mini_dataset/arbitrary_send.sol#28)

(High)Test.withdraw_inderect() (tests/mini_dataset/arbitrary_send.sol#73-76) sends eth to arbitrary user

Dangerous calls:

- msg.sender.send(val) (tests/mini_dataset/arbitrary_send.sol#75)

(High)Test.indirect_ceshi1() (tests/mini_dataset/arbitrary_send.sol#78-85) sends eth to arbitrary user

Dangerous calls:

- dd.send(val) (tests/mini_dataset/arbitrary_send.sol#84)

Security advice

Ensure that no user can withdraw unauthorized funds.

4.15 continue-in-loop

Vulnerability description

continue will cause the loop judgment condition to fail to increment, resulting in an infinite loop.

Applicable scenarios

```
contract C {
  function f(uint a, uint b) public{
```

```
        uint a = 0;
        do {
            continue;
            a++;
        } while(a<10);
    }
}
```

Audit results:【Pass】

Security advice: none.

4.16 controlled-array-length

Vulnerability description

Detect direct allocation of array length.

Applicable scenarios

```
contract A {
    uint[] testArray; // dynamic size array
    function f(uint usersCount) public {
        // ...
        testArray.length = usersCount;
        // ...
    }
    function g(uint userIndex, uint val) public {
        // ...
        testArray[userIndex] = val;
        // ...
    }
}
```

Contract storage/state variables are indexed by 256-bit integers. Users can set the array length to 2 ** 256-1 to index all storage slots. In the above example, you can call function f to set the length of the array, and then call function g to control any storage slots needed. Please note that the storage slots here are indexed by the hash of the indexer. Nonetheless, all storage will still be accessible and can be controlled by an attacker.

Audit results:【Pass】

Security advice: none.

4.17 controlled-delegatecall

Vulnerability description

Delegate the call or call code to an address controlled by the user. The address of Delegatecall is not necessarily trusted, it is still a problem of access control, and the address is not checked.

Applicable scenarios

```
contract Delegatecall{
    function delegate(address to, bytes data){
        to.delegatecall(data);
    }
}
```

Bob calls `delegate` and delegates the execution of the malicious contract to him. As a result, Bob withdraws the funds from the contract and destroys the contract.

Audit results:【Pass】

Security advice: none.

4.18 incorrect-constructor

Vulnerability description

The constructor is a special function and can only be called once during contract creation. They usually perform key privileged operations, such as setting up contract owners. Before Solidity 0.4.22, the only way to define a constructor was to create a function with the same name as the contract class that contained it. If its name does not exactly match the contract name, the function intended to be the constructor will become a normal callable function. This behavior can sometimes cause security issues, especially when the smart contract code is reused with other names but the name of the constructor has not been changed accordingly.

Applicable scenarios

```
contract Incorrectconstructor {
    address owner;
    function Incorrectconstructo() {
        owner = msg.sender;
    }
    modifier ifowner()
    {
        require(msg.sender == owner);
        _;
    }
    function withdrawmoney() ifowner {
        msg.sender.transfer(address(this).balance);
    }
}
```

Audit results:【Pass】

Security advice: none.

4.19 parity-multisig-bug

Vulnerability description

Multi-signature vulnerability. Hackers can use the initWallet function to call the initMultiowned function to obtain the identity of the contract owner.

Applicable scenarios

```
contract WalletLibrary_bad is WalletEvents {
   function initWallet(address[] _owners, uint _required, uint _daylimit) {
      initDaylimit(_daylimit);
      initMultiowned(_owners, _required);
   } // kills the contract sending everything to `_to`.
   function initMultiowned(address[] _owners, uint _required) {
      m_numOwners = _owners.length + 1;
      m_owners[1] = uint(msg.sender);
      m_ownerIndex[uint(msg.sender)] = 1;
      for (uint i = 0; i < _owners.length; ++i)
      {
         m_owners[2 + i] = uint(_owners[i]);
         m_ownerIndex[uint(_owners[i])] = 2 + i;
      }
      m_required = _required;
   }
}
```

Audit results:【Pass】

Security advice: none.

4.20 reentrancy-eth

Vulnerability description

A reentrancy error was detected. This is the reentry of ether. Through re-entry, the account balance can be maliciously withdrawn, resulting in losses. Do not report re-reporting that does not involve Ether (please refer to "reentrancy-no-eth")

Applicable scenarios

```
function withdrawBalance(){
   // send userBalance[msg.sender] Ether to msg.sender
   // if mgs.sender is a contract, it will call its fallback function
   if( ! (msg.sender.call.value(userBalance[msg.sender])() ) ){
      throw;
   }
   userBalance[msg.sender] = 0;
}
```

Bob used the reentrance vulnerability to call `withdrawBalance` multiple times and withdrew more than he originally deposited into the contract.

Audit results:【Pass】

Security advice: none.

4.21 storage-array

Vulnerability description

The solc 0.4.7-0.5.10 version contains a compiler error that causes the values in the signed integer array to be incorrect.

Applicable scenarios

```
contract A {
   int[3] ether_balances; // storage signed integer array
   function bad0() private {
      // ...
      ether_balances = [-1, -1, -1];
      // ...
   }
}
```

bad0() uses a (storage-allocated) signed integer array state variable to store the ether balance of three accounts. -1 should indicate uninitialized values, but a Solidity error made them 1 and these values can be used by the account.

Audit results:【Pass】

Security advice: none.

4.22 weak-prng

Vulnerability description

The PRNG is weaker due to the modulus of block.timestamp, now or blockhash. These may be affected by miners to some extent, so they should be avoided.

Applicable scenarios

```
contract Game {
   uint reward_determining_number;
   function guessing() external{
    reward_determining_number = uint256(block.blockhash(10000)) % 10;
   }
}
```

Eve is a miner. Eve calls guessing() and reorders the blocks containing transactions. As a result, Eve won the game.

Audit results:【Pass】

Security advice: none.

4.23 assert-violation

Vulnerability description

The Solidity assert() function is used to declare invariants. Normally running code will never reach a failed assert statement. An attainable assertion may mean one of two things: 1. There is an error in the contract that allows it to enter an invalid state. 2. The assert statement is used

incorrectly, for example to verify input. ; CWE-670: Always incorrect controlflow implementation. The Assert condition can be false, and the definitionhere is wider. It is best to use requere. Generally used at the end of thefunction to verify whether there is overflow or something.

Applicable scenarios

```
contract Assertviolation [
   function bad(uint a, uint b){
      assert(a>b);
   }
}
```

Audit results:【Pass】

Security advice: none.

4.24 constructor-return

Vulnerability description

The return statement is used in the constructor of the contract. With return, the deployment process will be different from the intuitive process. For example, the deployed bytecode may not include functions implemented in the source code.

Applicable scenarios

```
pragma solidity 0.4.24;
contract HoneyPot {
   bytes internal constant ID = hex"60203414600857005B60008080803031335AF100";
   constructor () public payable {
      bytes memory contract_identifier = ID;
      assembly { return(add(0x20, contract_identifier), mload(contract_identifier)) }
   }
   function withdraw() public payable {
      require(msg.value >= 1 ether);
      msg.sender.transfer(address(this).balance);
   }
}
```

Audit results:【Pass】

Security advice: none.

4.25 default-return-value

Vulnerability description

If a function is declared to have a return value, but no return value is given to it in the end, a default return value will be generated, and the default return value and the return value after actual execution may be different.

Applicable scenarios

```
contract C{
    function bad_return() public returns(bool flag){
        address aa = msg.sender;
    }
    function bad_return1() public returns(bool){
        address aa = msg.sender;
    }
}
```

Audit results:【Pass】

Security advice: none.

4.26 enum-conversion

Vulnerability description

Detect out-of-range enumeration conversion (solc <0.4.5).

Applicable scenarios

```
pragma solidity 0.4.2;
contract Test{
    enum E{a}
    function bug(uint a) public returns(E){
        return E(a);
    }
}
```

An attacker can trigger unexpected behavior by calling bug(1).

Audit results:【Pass】

Security advice: none.

4.27 erc1155-interface

Vulnerability description

The return value of the "ERC1155" function is incorrect. Interacting with these functions, the contract of solidity version> 0.4.22 will not be executed because of the lack of return value.

Applicable scenarios

```
contract Token{
    function balanceOf(address _owner, uint256 _id) external view returns (bool);
    //...
}
```

Token.balanceOf does not return the expected uint256. Bob deploys the token. Alice creates a contract to interact with, but assumes that the interface is the correct `ERC1155`. Alice's contract cannot interact with Bob's contract.

Audit results:【Pass】

Security advice: none.

4.28 erc1410-interface

Vulnerability description

The return value of the "ERC1410" function is incorrect. Interacting with these functions, the contract of solidity version> 0.4.22 will not be executed because of the lack of return value.

Applicable scenarios

```
contract Token{
   function isOperator(address _operator, address _tokenHolder) external view returns (uint256);
   //...
}
```

Token.isOperator does not return the expected boolean value. Bob deploys the token. Alice creates a contract to interact with, but uses the correct `ERC1410` interface implementation. Alice's contract cannot interact with Bob's contract.

Audit results:【Pass】

Security advice: none.

4.29 erc20-interface

Vulnerability description

The return value of the "ERC20" function is incorrect. Interacting with these functions, the contract of solidity version> 0.4.22 will not be executed because of the lack of return value.

Applicable scenarios

```
contract Token{
   function transfer(address to, uint value) external;
   //...
}
```

Token.transfer does not return the expected boolean value. Bob deploys the token. Alice creates a contract to interact with, but uses the correct `ERC20` interface implementation. Alice's contract cannot interact with Bob's contract.

Audit results:【Pass】

Security advice: none.

4.30 erc223-interface

Vulnerability description

The return value of the "ERC223" function is incorrect. Interacting with these functions, the contract of solidity version> 0.4.22 will not be executed because of the lack of return value.

Applicable scenarios

```
contract Token{
    function name() constant returns (uint _name);
    //...
}
```

Token.name does not return the expected boolean value. Bob deploys the token. Alice creates a contract to interact with, but uses the correct `ERC223` interface implementation. Alice's contract cannot interact with Bob's contract.

Audit results:【Pass】

Security advice: none.

4.31 erc621-interface

Vulnerability description

The return value of the "ERC621" function is incorrect. Interacting with these functions, the contract of solidity version> 0.4.22 will not be executed because of the lack of return value.

Applicable scenarios

```
contract Token{
    function decreaseSupply(uint value, address from) external;
    //...
}
```

Token.decreaseSupply does not return the expected boolean value. Bob deploys the token. Alice creates a contract to interact with, but uses the correct `ERC621` interface implementation. Alice's contract cannot interact with Bob's contract.

Audit results:【Pass】

Security advice: none.

4.32 erc721-interface

Vulnerability description

The return value of the "ERC721" function is incorrect. Interacting with these functions, the contract of solidity version> 0.4.22 will not be executed because of the lack of return value.

Applicable scenarios

```
contract Token{
    function ownerOf(uint256 _tokenId) external view returns (bool);
    //...
}
```

Token.ownerOf does not return the expected boolean value. Bob deploys the token. Alice creates a contract to interact with, but uses the correct `ERC721` interface implementation. Alice's contract cannot interact with Bob's contract.

Audit results:【Pass】

Security advice: none.

4.33 erc777-interface

Vulnerability description

The return value of the "ERC777" function is incorrect. Interacting with these functions, the contract of solidity version> 0.4.22 will not be executed because of the lack of return value.

Applicable scenarios

```
contract Token{
    function defaultOperators() public view returns (address);
    //...
}
```

Token.defaultOperators does not return the expected boolean value. Bob deploys the token. Alice creates a contract to interact with, but uses the correct `ERC777` interface implementation. Alice's contract cannot interact with Bob's contract.

Audit results:【Pass】

Security advice: none.

4.34 erc875-interface

Vulnerability description

The return value of the "ERC875" function is incorrect. Interacting with these functions, the contract of solidity version> 0.4.22 will not be executed because of the lack of return value.

Applicable scenarios

```
contract Token{
    function balanceOf(address _owner) public view returns (string _balances);
    //...
}
```

Token.balanceOf does not return the expected boolean value. Bob deploys the token. Alice creates a contract to interact with, but uses the correct `ERC875` interface implementation. Alice's contract cannot interact with Bob's contract.

Audit results:【Pass】

Security advice: none.

4.35 incorrect-equality

Vulnerability description

Using strict equality (== and !=), an attacker can easily manipulate these equality. Specifically: the opponent can forcefully send Ether to any address through selfdestruct() or through mining, thereby invalidating the strict judgment.

Applicable scenarios

```
contract Crowdsale{
   function fund_reached() public returns(bool){
      return this.balance == 100 ether;
   }
}
```

Crowdsale relies on fund_reached to know when to stop the sale of tokens. Bob sends 0.1 ether. As a result, fund_reached is always false, and crowdsale is always true.

Audit results:【Pass】

Security advice: none.

4.36 incorrect-signature

Vulnerability description

In Solidity, the definition of a function signature is: a basic prototype canonical expression without a data position specifier, that is, a function name with a bracketed parameter type list. The parameter types are separated by a comma without spaces. This means uint256 and int256 should be used instead of uint or int. For example: in bytes4 (keccak256()), signatrue should be unit256 or int256, not uint or int, otherwise the length is not enough.

Applicable scenarios

```
pragma solidity ^0.5.1;
contract Signature {
   function callFoo(address addr, uint value) public returns (bool) {
      bytes memory data = abi.encodeWithSignature("foo(uint)", value);
      (bool status, ) = addr.call(data);
      return status;
   }
}
```

Audit results:【Pass】

Security advice: none.

4.37 locked-ether

Vulnerability description

A contract programmed to receive ether (with the payable logo) should implement the method of withdrawing ether, that is, call transfer

(recommended), send or call.value at least once.

Applicable scenarios

```
pragma solidity 0.4.24;
contract Locked{
   function receive() payable public{}
}
```

All Ether sent to "Locked" will be lost.

Audit results:【Pass】

Security advice: none.

4.38 mapping-deletion

Vulnerability description

Deleting in a structure that contains a mapping will not delete the mapping (see Solidity documentation). The remaining data can be used to break the contract.

Applicable scenarios

```
struct BalancesStruct{
   address owner;
   mapping(address => uint) balances;
}
mapping(address => BalancesStruct) public stackBalance;
function remove() internal{
    delete stackBalance[msg.sender];
}
```

Remove() deletes one item of stackBalance. The mapped balances will never be deleted, so remove() will not work properly.

Audit results:【Pass】

Security advice: none.

4.39 shadowing-abstract

Vulnerability description

Detect hidden state variables in abstract contracts. Unlike shadowing-state, hidden state variables that are not used in the parent contract are detected here.

Applicable scenarios

```
contract BaseContract{
   address owner;
}
contract DerivedContract is BaseContract{
   address owner;
}
```

The owner of BaseContract is in the hidden variable of DerivedContract.

Audit results:【Pass】

Security advice: none.

4.40 tautology

Vulnerability description

Detecting tautology or contradictory expressions means that if, while, require, and assert conditions are always true or false.

Applicable scenarios

```
contract A {
   function f(uint x) public {
      // ...
      if (x >= 0) { // bad -- always true
         // ...
      }
      // ...
   }
   function g(uint8 y) public returns (bool) {
      // ...
      return (y < 512); // bad!
      // ...
   }
}
```

x is uint256, so x>= 0 will always be true. y is uint8, so y <512 will always be true.

Audit results:【Pass】

Security advice: none.

4.41 boolean-cst

Vulnerability description

Detect abuse of Boolean constants. Bool variable is used incorrectly, here is the operation of bool variable.

Applicable scenarios

```
contract A {
   function f(uint x) public {
      // ...
      if (false) { // bad!
         // ...
      }
      // ...
   }
   function g(bool b) public returns (bool) {
      // ...
      return (b || true); // bad!
      // ...
```

```
  }
}
```

The Boolean constants in the code have very few legal uses. Other uses (as conditions in complex expressions) indicate the persistence of errors or error codes.

Audit results:【Pass】

Security advice: none.

4.42 constant-function-state

Vulnerability description

Functions declared as constant/pure/view will change their state. Constant/pure/view was not enforced before Solidity 0.5. Starting from Solidity 0.5, calls to constant/pure/view functions use the STATICCALL opcode, which is restored when the state is modified. As a result, calls to incorrectly labeled functions may catch contracts compiled with Solidity 0.5.

Applicable scenarios

```
contract Constant{
  uint counter;
  function get() public view returns(uint){
    counter = counter +1;
    return counter
  }
}
```

Constant is deployed as Solidity 0.4.25. Bob wrote a smart contract that interacts with Constant in Solidity 0.5.0. All calls to get will be restored, thus destroying Bob's smart contract execution.

Audit results:【Pass】

Security advice: none.

4.43 divide-before-multiply

Vulnerability description

Entities only support integers, so division is often truncated; performing multiplication before division can sometimes avoid loss of precision, and entity integer division may be truncated. As a result, performing multiplication before division may reduce accuracy.

Applicable scenarios

```
contract A {
  function f(uint n) public {
    coins = (oldSupply / n) * interest;
  }
}
```

If n is greater than oldSupply, coins will be zero. For example, oldSupply = 5; n = 10, interest = 2, coins will be zero. If (oldSupply * interest / n) is used, the coin will be 1. In general, it is usually best to rearrange the arithmetic to perform the multiplication before the division, unless a smaller type of restriction makes this operation dangerous.

Audit results:【Pass】

Security advice: none.

4.44 erc20-approve

Vulnerability description

The approve function of ERC-20 is vulnerable to attack. Using a preemptive attack, the approved tokens can be spent before changing the quota value. This attack is also a type of transaction sequence dependence (TOD).

Applicable scenarios

```
pragma solidity ^0.4.5;
contract StandardToken is ERC20, BasicToken {
  ...
  function approve(address _spender, uint256 _value) public returns (bool) {
    allowed[msg.sender][_spender] = _value;
    Approval(msg.sender, _spender, _value);
    return true;
  }
  ...
}
```

Audit results:【Pass】

Security advice: none.

4.45 function-problem

Vulnerability description

The function will always end in an abnormal state such as revert(), and it cannot return after the normal execution, indicating that there is a problem with the function design.

Applicable scenarios

```
contract Functionproblem {
  address owner;
  function bad() { //bad
    revert();
  }
}
```

Audit results:【Pass】

Security advice: none.

4.46 mul-var-len-arguments

Vulnerability description

abi.encodePacked() In some cases, using multiple variable length parameters may cause hash collisions. Since abi.encodePacked() all elements are packed in order, regardless of whether they belong to an array, you can move elements between arrays, and as long as all elements are in the same order, it will return the same encoding. In the case of signature verification, an attacker can effectively bypass authorization by modifying the position of the element in the previous function call, thereby exploiting this vulnerability. Replay authentication through abi.encodePacked(), because in encodePacked(a,b), if both a and b are variable-length arrays, you can add the last one of a to the first one of b, so that the total order is not Change, you can generate the same value.

Applicable scenarios

```
contract Mulvarlenarguments {
    function addUsers(
        address[] calldata admins,
        address[] calldata regularUsers,
        bytes calldata signature
    )
        external
    {
        bytes32 hash = keccak256(abi.encodePacked(admins, regularUsers));
        address signer = hash.toEthSignedMessageHash().recover(signature);
    }
}
```

Audit results:【Pass】

Security advice: none.

4.47 reentrancy-no-eth

Vulnerability description

Check that there is reentrancy (there is a situation that reads first, writes later), but there is no transfer of eth.

Applicable scenarios

```
function bug(){
    require(not_called);
    if( ! (msg.sender.call() ) ){
        throw;
    }
    not_called = False;
}
```

Audit results:【Pass】

Security advice: none.

4.48 reused-constructor

Vulnerability description

Check whether the same basic constructor is called with parameters from two different positions in the same inheritance hierarchy.

Applicable scenarios

```
pragma solidity ^0.4.0;
contract A{
   uint num = 5;
   constructor(uint x) public{
      num += x;
   }
}
contract B is A{
   constructor() A(2) public { /* ... */ }
}
contract C is A {
   constructor() A(3) public { /* ... */ }
}
contract D is B, C {
   constructor() public { /* ... */ }
}
contract E is B {
   constructor() A(1) public { /* ... */ }
}
```

The constructor of A is called multiple times in D and E: D inherits from B and C, and both constitute A. E only inherits from B, but B and E construct A.

Audit results:【Pass】

Security advice: none.

4.49 tx-origin

Vulnerability description

If a legitimate user interacts with a malicious contract, the protection based on tx.origin will be abused by the malicious contract.

Applicable scenarios

```
contract TxOrigin {
   address owner = msg.sender;
   function bug() {
      require(tx.origin == owner);
   }
}
```

Bob is the owner of TxOrigin. Bob calls Eve's contract. Eve's contract is called TxOrigin and bypasses the protection of tx.origin.

Audit results:【Pass】

Security advice: none.

4.50 typographical-error

Vulnerability description

For example, when the intent of a defined operation is to sum a number with a variable (+=), but a typographical error is accidentally defined in the wrong way (= +), a typographical error occurs, and this happens to be effective Operator. Instead of calculating the sum, initialize the variables again.

Applicable scenarios

```
pragma solidity ^0.4.24;
contract TypoOneCommand {
    uint numberOne = 1;
    string numberstring = "";
    function alwaysOne() public {
        numberOne =+ 1; //bad
    }
    function alwaysOne_bad() public {
        numberOne =- 1; //bad
    }
}
```

Audit results:【Pass】

Security advice: none.

4.51 unchecked-lowlevel

Vulnerability description

The low-level call to the external contract failed, and the return value was not judged. When sending ether at the same time, please check the return value and handle the error.

Applicable scenarios

```
contract MyConc{
    function my_func(address payable dst) public payable{
        dst.call.value(msg.value)("");
    }
}
```

The return value of the low-level call is not checked, so if the call fails, the ether will be locked in the contract. If you use low-level calls to block block operations, consider logging the failed calls.

Audit results:【Pass】

Security advice: none.

4.52 unchecked-send

Vulnerability description

Similar to unchecked-lowlevel, it is explained here that the return value of send and Highlevelcall is not checked.

Applicable scenarios: 【Medium:13】

For this pattern, the specific problems in the contract are as follows:

(Medium)Test.buy() (tests/mini_dataset/arbitrary_send.sol#87-92) unchecks send return value:

- msg.sender.send(remaining) (tests/mini_dataset/arbitrary_send.sol#91)

(Medium)Test.direct() (tests/mini_dataset/arbitrary_send.sol#13-15) unchecks send return value:

- msg.sender.send(address(this).balance) (tests/mini_dataset/arbitrary_send.sol#14)

(Medium)Test.directceshi_1() (tests/mini_dataset/arbitrary_send.sol#31-34) unchecks send return value:

- destination.send(address(this).balance) (tests/mini_dataset/arbitrary_send.sol#33)

(Medium)Test.directceshi_2() (tests/mini_dataset/arbitrary_send.sol#36-38) unchecks send return value:

- destination.send(address(this).balance) (tests/mini_dataset/arbitrary_send.sol#37)

(Medium)Test.directceshi_3() (tests/mini_dataset/arbitrary_send.sol#40-42) unchecks send return value:

- destination.send(msg.value) (tests/mini_dataset/arbitrary_send.sol#41)

(Medium)Test.directceshi_4() (tests/mini_dataset/arbitrary_send.sol#44-46) unchecks send return value:

- destination.send(balances[msg.sender]) (tests/mini_dataset/arbitrary_send.sol#45)

(Medium)Test.directceshi_5() (tests/mini_dataset/arbitrary_send.sol#48-50) unchecks send return value:

- destination.send(0) (tests/mini_dataset/arbitrary_send.sol#49)

(Medium)Test.directceshi_6() (tests/mini_dataset/arbitrary_send.sol#52-55) unchecks send return value:

- destination.send(avalue) (tests/mini_dataset/arbitrary_send.sol#54)

(Medium)Test.indirect() (tests/mini_dataset/arbitrary_send.sol#27-29) unchecks send return value:

- destination.send(address(this).balance) (tests/mini_dataset/arbitrary_send.sol#28)

(Medium)Test.indirect_ceshi1() (tests/mini_dataset/arbitrary_send.sol#78-85) unchecks send return value:

- dd.send(val) (tests/mini_dataset/arbitrary_send.sol#84)

(Medium)Test.withdraw() (tests/mini_dataset/arbitrary_send.sol#63-66) unchecks send return value:

- msg.sender.send(val) (tests/mini_dataset/arbitrary_send.sol#65)

(Medium)Test.withdraw_direct() (tests/mini_dataset/arbitrary_send.sol#68-71) unchecks send return value:

- msg.sender.send(val) (tests/mini_dataset/arbitrary_send.sol#70)

(Medium)Test.withdraw_inderect() (tests/mini_dataset/arbitrary_send.sol#73-76) unchecks send return value:

- msg.sender.send(val) (tests/mini_dataset/arbitrary_send.sol#75)

Security advice

Make sure to check or record the return value of send.

4.53 uninitialized-local

Vulnerability description

Check the local variables that are not initialized.

Applicable scenarios

```
contract Uninitialized is Owner{
  function withdraw() payable public onlyOwner{
    address to;
    to.transfer(this.balance)
  }
}
```

Bob calls transfer. As a result, all ether was sent to the address "0x0" and lost.

Audit results:【Pass】

Security advice: none.

4.54 unused-return

Vulnerability description

The return value of the call is not stored in a local variable or state variable, that is, the calling function may not have any effect.

Applicable scenarios

```
contract MyConc{
    using SafeMath for uint;
    function my_func(uint a, uint b) public{
        a.add(b);
    }
}
```

MyConc will call SafeMath's add, but will not store the result in a. As a result, the calculation has no effect.

Audit results:【Pass】

Security advice: none.

4.55 writeto-arbitrarystorage

Vulnerability description

The data of the smart contract (for example, the owner of the storage contract) is permanently stored in a storage location (ie, key or address) at the EVM level. The contract is responsible for ensuring that only authorized users or contract accounts can write to sensitive storage locations. If an attacker can write to any storage location of the contract, the authorization check can be easily bypassed. This may allow an attacker to destroy the storage space. For example, by overwriting a field that stores the address of the contract owner.

Applicable scenarios

```
contract Map {
    address public owner;
    uint256[] map;
    function set(uint256 key, uint256 value) public {
        if (map.length <= key) {
            map.length = key + 1;
        }
        map[key] = value;
    }
}
```

Audit results:【Pass】

Security advice: none.

4.56 costly-loop

Vulnerability description

Ethereum is a very resource-constrained environment. The price of each calculation step is several orders of magnitude higher than the price of the centralized provider. In addition, Ethereum miners impose limits on the total amount of natural gas consumed in the block. If array.length is large enough, the function exceeds the gas limit, and the transaction that calls the function will never be confirmed. If external participants influence array.length, this will become a security issue.

Applicable scenarios

```
pragma solidity 0.4.24;
contract PriceOracle {
    address internal owner;
    address[] public subscribers;
    mapping(address => uint) balances;
    uint internal constant PRICE = 10**15;
    function subscribe() payable external{
        subscribers.push(msg.sender);
        balances[msg.sender] += msg.value;
    }
    function setPrice(uint price) external {
        require(msg.sender == owner);
        bytes memory data = abi.encodeWithSelector(SIGNATURE, price);
        for (uint i = 0; i < subscribers.length; i++) {
            if(balances[subscribers[i]] >= PRICE) {
                balances[subscribers[i]] -= PRICE;
                subscribers[i].call.gas(50000)(data);
            }
        }
    }
}
```

Audit results:【Pass】

Security advice: none.

4.57 shift-parameter-mixup

Vulnerability description

Check whether the value in the shift operation is inverted.

Applicable scenarios

```
contract C {
    function f() internal returns (uint a) {
        assembly {
            a := shr(a, 8)
        }
    }
}
```

The shift statement shifts the constant 8 one bit to the right.

Audit results:【Pass】

Security advice: none.

4.58 shadowing-builtin

Vulnerability description

Use local variables, state variables, functions, modifiers or events to detect hidden built-in symbols.

Applicable scenarios

```
pragma solidity ^0.4.24;
contract Bug {
   uint now; // Overshadows current time stamp.
   function assert(bool condition) public {
      // Overshadows built-in symbol for providing assertions.
   }
   function get_next_expiration(uint earlier_time) private returns (uint) {
      return now + 259200; // References overshadowed timestamp.
   }
}
```

now is defined as a state variable, and the built-in symbol now is hidden. The assert function eclipses the built-in assert function. Such use of these built-in symbols may lead to unexpected results.

Audit results:【Pass】

Security advice: none.

4.59 shadowing-function

Vulnerability description

Detect hidden functions.

Applicable scenarios

```
contract BaseContract{
   function aa(uint a,uint b) returns (uint) {
   return a;
   }
}
contract DerivedContract is BaseContract{
   function aa(uint a,uint b) returns (uint) {
   return b;
   }
}
```

The aa function of BaseContract does not work.

Audit results:【Pass】

Security advice: none.

4.60 shadowing-local

Vulnerability description

Detect hidden local variables.

Applicable scenarios

```
pragma solidity ^0.4.24;
contract Bug {
   uint owner;
   function sensitive_function(address owner) public {
      // ...
      require(owner == msg.sender);
   }
   function alternate_sensitive_function() public {
      address owner = msg.sender;
      // ...
      require(owner == msg.sender);
   }
}
```

The sensitive_function.owner hides the Bug.owner. As a result, using owner in sensitive_function may be incorrect.

Audit results:【Pass】

Security advice: none.

4.61 transfer-to-zeroaddress

Vulnerability description

In sensitive functions such as transfer, transferFrom, and transferOwnership, user operations are irreversible. Therefore, it is recommended that developers add a non-zero check for the target address in the implementation of these functions to avoid user misuse and cause loss of user permissions and property damage. If it is transferred to 0x00, it will be difficult for the ether to revert, which will cause the ether to be lost.

Applicable scenarios

```
contract Transfertozeroaddress {
   address owner;
   function bad() {
      address aa = 0x0;
      aa.transfer(msg.value);
   }
   function good() {
      address aa = 0x0;
      if(aa != 0x0) {revert();}
      aa.transfer(msg.value);
   }
}
```

Audit results:【Pass】

Security advice: none.

4.62 uninitialized-fptr-cst

Vulnerability description

Solc versions 0.4.5-0.4.26 and 0.5.0-0.5.8 contain a compiler error that causes unexpected behavior when calling uninitialized function pointers in the constructor.

Applicable scenarios

```
contract bad0 {
   constructor() public {
      /* Uninitialized function pointer */
      function(uint256) internal returns(uint256) a;
      a(10);
   }
}
```

Calling a(10) will cause unexpected behavior because the function pointer a is not initialized in the constructor.

Audit results:【Pass】

Security advice: none.

4.63 variable-scope

Vulnerability description

Check the possible usage of the variable before closing the declaration (because it was declared later, or declared in another scope).

Applicable scenarios

```
contract C {
   function f(uint z) public returns (uint) {
      uint y = x + 9 + z; // 'z' is used pre-declaration
      uint x = 7;
      if (z % 2 == 0) {
         uint max = 5;
         // ...
      }
      // 'max' was intended to be 5, but it was mistakenly declared in a scope and not assigned (so it is
zero).
      for (uint i = 0; i < max; i++) {
         x += 1;
      }
      return x;
   }
}
```

In the above case, the variable x will be used before the declaration, which may lead to unintended consequences. In addition, the for loop uses the variable max, which is declared in a previous scope that may not have been reached before. If the user incorrectly uses the variable before any expected statement assignment, it may lead to unintended consequences. It may also indicate that the user intends to reference other

variables. That is, max may not be declared normally, and unexpected situations willoccur during subsequent calls.

Audit results:【Pass】

Security advice: none.

4.64 void-cst

Vulnerability description

Detect calls to unimplemented constructors. The calling contract does not declare a constructor.

Applicable scenarios

```
contract A{}
contract B is A{
   constructor() public A(){}
}
```

When reading the definition of B's constructor, we can assume that A() starts the contract, but no code is executed.

Audit results:【Pass】

Security advice: none.

4.65 incorrect-modifier

Vulnerability description

If the modifier does not execute _ or restore, the execution of the function will return to the default value, which may mislead the caller.

Applicable scenarios

```
contract A{
   modidfier myModifier(){
      if(false){
       _;
      }
   }
   function set() myModifier returns(uint){
      return 0;
   }
}
```

myModifier will cause the set function to fail to execute.

Audit results:【Pass】

Security advice: none.

4.66 assemblycall-rewrite

Vulnerability description

It is dangerous to use the inline assembly instruction assemblyCall of the CALL series, which will overwrite the input with the output. If an arbitrary address is called a return value, the return value may be different from the expected value.

Applicable scenarios

```
contract MixinSignatureValidator {
    function isValidWalletSignature(
        bytes32 hash,
        address walletAddress,
        bytes signature
    )internal view returns (bool isValid){
        assembly {
            let cdStart := add(calldata, 32)
            let success := staticcall(
                gas, // forward all gas
                walletAddress, // address of Wallet contract
                cdStart, // pointer to start of input
                mload(calldata), // length of input
                cdStart, // write output over input
                32 // output size is 32 bytes
            )
        }
        return isValid;
    }
}
```

Audit results:【Pass】

Security advice: none.

4.67 block-other-parameters

Vulnerability description

Contracts usually require access to time values to perform certain types of functions. block.number can let you know the current time or time increment, but in most cases it is not safe to use them. block.number The block time of Ethereum is usually about 14 seconds, so the time increment between blocks can be predicted. However, the lockout time is not fixed and may change due to various reasons (for example, fork reorganization and difficulty coefficient). Since the block time is variable, block.number should not rely on accurate time calculations. The ability to generate random numbers is very useful in various applications. An obvious example is a gambling DApp, where a pseudo-random number generator is used to select the winner. However, creating a sufficiently powerful source of randomness in Ethereum is very challenging. Using blockhash, block.difficulty and other

areas is also unsafe because they are controlled by miners. If the stakes arehigh, the miner can mine a large number of blocks by renting hardware in ashort period of time, select the block that needs to obtain the block hashvalue to win, and then discard all other blocks.

Applicable scenarios

```
contract Otherparameters{
    event Number(uint);
    event Coinbase(address);
    event Difficulty(uint);
    event Gaslimit(uint);
    function bad0() external{
        require(block.number == 20);
        require(block.coinbase == msg.sender);
        require(block.difficulty == 20);
        require(block.gaslimit == 20);
    }
}
```

The randomness of Bob's contract depends on block.number and so on. Eve is a miner who manipulates block.number and so on to use Bob's contract.

Audit results:【Pass】

Security advice: none.

4.68 calls-loop

Vulnerability description

Check that the key access control ETH is transmitted cyclically. If at least one address cannot receive ETH (for example, it is a contract with a default fallback function), the entire transaction will be restored. Loss of parameters.

Applicable scenarios

```
contract CallsInLoop{
    address[] destinations;
    constructor(address[] newDestinations) public{
        destinations = newDestinations;
    }
    function bad() external{
        for (uint i=0; i < destinations.length; i++){
            destinations[i].transfer(i);
        }
    }
}
```

If one of the destination addresses is restored by the rollback function, bad() will restore all, so all the work is wasted.

Audit results:【Pass】

Security advice: none.

4.69 events-access

Vulnerability description

Detect missing events of key access control parameters.

Applicable scenarios

```
contract C {
    modifier onlyAdmin {
        if (msg.sender != owner) throw;
        _;
    }
    function updateOwner(address newOwner) onlyAdmin external {
        owner = newOwner;
    }
}
```

updateOwner() has no events, so it is difficult to track these extremely privileged operations off-chain.

Audit results:【Pass】

Security advice: none.

4.70 events-maths

Vulnerability description

Detect missing events of key arithmetic parameters.

Applicable scenarios

```
contract C {
    modifier onlyOwner {
        if (msg.sender != owner) throw;
        _;
    }
    function setBuyPrice(uint256 newBuyPrice) onlyOwner public {
        buyPrice = newBuyPrice;
    }
    function buy() external {
     ... // buyPrice is used to determine the number of tokens purchased
    }
}
```

setBuyPrice does not record events, so it is difficult to track changes in the purchase price off-chain.

Audit results:【Pass】

Security advice: none.

4.71 extcodesize-invoke

Vulnerability description

The extcodesize is zero when the contract is deployed, and the attacker can call the victim contract in his own constructor. At this time, using extcodesize to verify is invalid.

Applicable scenarios

```
pragma solidity ^0.4.23;
contract ExtCodeSize {
    // This contract would be 'hacked' if the address saved here is a contract address
    address public thisIsNotAContract;
    function aContractCannotCallThis() public {
        uint codeSize;
        assembly { codeSize := extcodesize(caller) }
        // If extcodesize returns 0, it means the caller's code length is 0, so, it is not a contract...
        // or maybe not
        require(codeSize == 0);
        thisIsNotAContract = msg.sender;
    }
}
```

Audit results:【Pass】

Security advice: none.

4.72 fllback-outofgas

Vulnerability description

The fallback function of the contract is usually used to receive an eth transfer (restore the withdrawal operation after the transfer fails), but if too complicated logic is implemented in the fallback, the gas may be exhausted, resulting in unsuccessful transfer.

Applicable scenarios

```
contract C {
    function f(uint a, uint b) public{
    uint a = 0;
    do {
            continue;
            a++;
        } while(a<10);
    }
}
```

Audit results:【Pass】

Security advice: none.

4.73 incorrect-blockhash

Vulnerability description

The blockhash function only returns the non-zero value of the last 256 blocks. In addition, for the current block, it always returns 0, that is, blockhash (block number) is always equal to 0. The hash of the current block

cannot be queried, only the most recent 256 blocks can be queried, otherwiseonly the value 0 is returned.

Applicable scenarios

```
pragma solidity 0.4.25;
contract MyContract {
    function currentBlockHash() public view returns(bytes32) {
        return blockhash(block.number);
    }
}
```

Audit results:【Pass】

Security advice: none.

4.74 incorrect-inheritance-order

Vulnerability description

The entity supports multiple inheritance, which means that a contract can inherit multiple contracts. Multiple inheritance introduces an ambiguity problem called diamond: if two or more basic contracts define the same function, which one should be called in the sub-contract? Solidity resolves this ambiguity by using reverse C3 linearization, which sets a priority between the underlying contracts. In this way, basic contracts have different priorities, so the order of inheritance is important. Ignoring the order of inheritance may lead to unexpected behavior. Pay attention to the order of contract inheritance, because the inherited contract may have overlapping variables or functions, and the inheritance order determines the level of the contract, which in turn determines the overlapping variables and functions in which contract to use.

Applicable scenarios

```
contract A {
    address owner;
}
contract B {
    address owner;
}
contract C is B,A{}
```

Audit results:【Pass】

Security advice: none.

4.75 integer-overflow

Vulnerability description

When an arithmetic operation reaches the maximum or minimum size of the type, overflow/underflow will occur. For example, if a number is stored in the uint8 type, it means that the number is stored as an 8-bit unsigned number, ranging from 0 to 2^8-1. In computer programming, when an arithmetic operation attempts to create a value, an integer overflow occurs, and the value can be represented by a given number of bits-greater than the maximum value or less than the minimum value.

Applicable scenarios: 【Low:1】

For this pattern, the specific problems in the contract are as follows:

(Low)Test.buy() (tests/mini_dataset/arbitrary_send.sol#87-92) may have integer overflow

Possible nodes:

- remaining = value_send - value_spent (tests/mini_dataset/arbitrary_send.sol#90)

Security advice

Use Safemath to perform integer arithmetic or verify calculated values.

4.76 missing-zero-check

Vulnerability description

Check the verification of the zero address.

Applicable scenarios: 【Low:1】

For this pattern, the specific problems in the contract are as follows:

(Low)Test.indirect_ceshi1().dd (tests/mini_dataset/arbitrary_send.sol#82) lacks a zero-check on :

- dd.send(val) (tests/mini_dataset/arbitrary_send.sol#84)

Security advice

Check that the address is not zero.

4.77 reentrancy-benign

Vulnerability description

A reentrancy error is detected. The main explanation here is that the effect of the reentrancy is the same as calling the function twice.

Applicable scenarios

```
function callme(){
  if( ! (msg.sender.call()() ) ){
```

```
        throw;
    }
    counter += 1
}
```

callme contains reentrant vulnerabilities. But reentrancy is benign, because its use has the same effect as two consecutive calls.

Audit results:【Pass】

Security advice: none.

4.78 reentrancy-events

Vulnerability description

A reentrancy error is detected, only reentrance vulnerabilities that can lead to out-of-sequence events are reported here.

Applicable scenarios

```
function bug(Called d){
    counter += 1;
    d.f();
    emit Counter(counter);
}
```

If d() is reentered, Counter events will be displayed in the wrong order, which may cause problems for third parties.

Audit results:【Pass】

Security advice: none.

4.79 timestamp

Vulnerability description

There is a strict comparison with block.timestamp or now in the contract, and miners can benefit from block.timestamp.

Applicable scenarios

```
contract Timestamp{
    event Time(uint);
    modifier onlyOwner {
        require(block.timestamp == 0);
        _;
    }
    function bad0() external{
        require(block.timestamp == 0);
    }
}
```

Bob's contract relies on the randomness of block.timestamp. Eve is a miner who manipulates block.timestamp to take advantage of Bob's contract.

Audit results:【Pass】

Security advice: none.

4.80 signature-malleability

Vulnerability description

The implementation of the cryptographic signature system in the Ethereum contract usually assumes that the signature is unique, but the signature can be changed without having the private key, and the signature is still valid. The EVM specification defines several so-called "pre-compiled" contracts, one of which ecrecover performs elliptic curve public key recovery. A malicious user can slightly modify the three values v, r and s to create other valid signatures. If the signature is part of the signed message hash, the system that performs signature verification at the contract level may be attacked. Malicious users can create valid signatures to replay previously signed messages. The signature cannot contain the existing signature, which is vulnerable to attack.

Applicable scenarios

```
function transfer(bytes _signature,address _to,uint256 _value,uint256 _gasPrice,uint256 _nonce) public
returns (bool){
    bytes32 txid = keccak256(abi.encodePacked(getTransferHash(_to, _value, _gasPrice, _nonce),
_signature)); //bad
    require(!signatureUsed[txid]);
    address from = recoverTransferPreSigned(_signature, _to, _value, _gasPrice, _nonce);
    require(balances[from] > _value);
    balances[from] -= _value;
    balances[_to] += _value;
    signatureUsed[txid] = true;
}
```

Audit results:【Pass】

Security advice: none.

4.81 assembly

Vulnerability description

Inline assembly is to access the Ethereum virtual machine from a low level, which leads to the abandonment of several important security features of Solidity.

Applicable scenarios

```
function recover(bytes32 hash, bytes sig) public pure returns (address) {
    bytes32 r;
    bytes32 s;
    uint8 v;
    // Divide the signature in r, s and v variables
    assembly {
```

```
        r := mload(add(sig, 32))
        s := mload(add(sig, 64))
        v := byte(0, mload(add(sig, 96)))
    }
}
```

Audit results:【Pass】

Security advice: none.

4.82 assert-state-change

Vulnerability description

Wrong use of assert(). See Solidity best practices.

Applicable scenarios

```
contract A {
 uint s_a;
 function bad() public {
   assert((s_a += 1) > 10);
 }
}
```

The assert in Bad() increments the state variable s_a when checking the condition.

Audit results:【Pass】

Security advice: none.

4.83 delete-dynamic-arrays

Vulnerability description

Applying delete or .length=0 to dynamically sized storage arrays may cause Out-of-Gas exceptions. Because it will traverse all the data in the array, the gas may be exceeded.

Applicable scenarios

```
contract C {
    uint[] amounts;
    address payable[] addresses;
    function collect(address payable to) external payable {
        amounts.push(msg.value);
        addresses.push(to);
    }
    function pay() external {
        uint length = amounts.length;
        delete amounts;
        delete addresses;
    }
}
```

Audit results:【Pass】

Security advice: none.

4.84 deprecated-standards

Vulnerability description

Several functions and operators in Solidity are not recommended. Using them will reduce code quality. In the new major version of the Solidity compiler, deprecated functions and operators may cause side effects and compilation errors. Structures deprecated by Solidity after 0.5.0: years, sha3, suicide, throw and constant functions.

Applicable scenarios

```
contract ContractWithDeprecatedReferences {
   // Deprecated: Change block.blockhash() -> blockhash()
   bytes32 globalBlockHash = block.blockhash(0);
   // Deprecated: Change constant -> view
   function functionWithDeprecatedThrow() public constant {
      // Deprecated: Change msg.gas -> gasleft()
      if(msg.gas == msg.value) {
         // Deprecated: Change throw -> revert()
         throw;
      }
   }
   // Deprecated: Change constant -> view
   function functionWithDeprecatedReferences() public constant {
      // Deprecated: Change sha3() -> keccak256()
      bytes32 sha3Result = sha3("test deprecated sha3 usage");
      // Deprecated: Change callcode() -> delegatecall()
      address(this).callcode();
      // Deprecated: Change suicide() -> selfdestruct()
      suicide(address(0));
   }
}
```

Audit results:【Pass】

Security advice: none.

4.85 erc20-indexed

Vulnerability description

The address parameters of the "Transfer" and "Approval" events of the ERC-20 token standard shall include indexed.

Applicable scenarios

```
contract ERC20Bad {
   // ...
   event Transfer(address from, address to, uint value);
   event Approval(address owner, address spender, uint value);
   // ...
}
```

According to the definition of the ERC20 specification, the first two parameters of the Transfer and Approval events should carry the indexed keyword. If these keywords are

not included, the parameter data will be excluded from the bloom filter of thetransaction/block. Therefore, external tools searching for these parameters may ignorethem and fail to index the logs in this token contract.

Audit results:【Pass】

Security advice: none.

4.86 erc20-throw

Vulnerability description

The function of the ERC-20 token standard should be thrown in the following special circumstances: if there are not enough tokens in the _from account balance to spend, it should be thrown; unless the _from account deliberately authorizes the sending of messages through some mechanism Otherwise, transferFrom should be thrown.

Applicable scenarios

```
contract SomeToken {
    mapping(address => uint256) balances;
    event Transfer(address indexed _from, address indexed _to, uint256 _value);
    function transfer(address _to, uint _value) public returns (bool) {
        if (_value > balances[msg.sender] || _value > balances[_to] + _value) {
            return false;
        }
        balances[msg.sender] = balances[msg.sender] - _value;
        balances[_to] = balances[_to] + _value;
        emit Transfer(msg.sender, _to, _value);
        return true;
    }
}
```

Audit results:【Pass】

Security advice: none.

4.87 length-manipulation

Vulnerability description

The length of the dynamic array changes directly. In this case, huge arrays may appear, and storage overlap attacks (conflicts with other data in the storage) may result. The operations of "length" are: =, +=, -=, *=, /=, --, etc.

Applicable scenarios

```
pragma solidity 0.4.24;
contract dataStorage {
    uint[] public data;
    function writeData(uint[] _data) external {
        for(uint i = data.length; i < _data.length; i++) {
```

```
            data.length++;
            data[i]=_data[i];
        }
    }
}
```

Audit results:【Pass】

Security advice: none.

4.88 low-level-calls

Vulnerability description

Label low-level methods such as call, delegatecall, and callcode, because these methods are easily exploited by attackers.

Applicable scenarios: 【Info:13】

For this pattern, the specific problems in the contract are as follows:

(Informational)Low level call in Test.direct()
(tests/mini_dataset/arbitrary_send.sol#13-15):
- msg.sender.send(address(this).balance)
(tests/mini_dataset/arbitrary_send.sol#14)

(Informational)Low level call in Test.indirect()
(tests/mini_dataset/arbitrary_send.sol#27-29):
- destination.send(address(this).balance)
(tests/mini_dataset/arbitrary_send.sol#28)

(Informational)Low level call in Test.directceshi_1()
(tests/mini_dataset/arbitrary_send.sol#31-34):
- destination.send(address(this).balance)
(tests/mini_dataset/arbitrary_send.sol#33)

(Informational)Low level call in Test.directceshi_2()
(tests/mini_dataset/arbitrary_send.sol#36-38):
- destination.send(address(this).balance)
(tests/mini_dataset/arbitrary_send.sol#37)

(Informational)Low level call in Test.directceshi_3()
(tests/mini_dataset/arbitrary_send.sol#40-42):
- destination.send(msg.value) (tests/mini_dataset/arbitrary_send.sol#41)

(Informational)Low level call in Test.directceshi_4()
(tests/mini_dataset/arbitrary_send.sol#44-46):

- destination.send(balances[msg.sender])
(tests/mini_dataset/arbitrary_send.sol#45)

(Informational)Low level call in Test.directceshi_5()
(tests/mini_dataset/arbitrary_send.sol#48-50):

- destination.send(0) (tests/mini_dataset/arbitrary_send.sol#49)

(Informational)Low level call in Test.directceshi_6()
(tests/mini_dataset/arbitrary_send.sol#52-55):

- destination.send(avalue) (tests/mini_dataset/arbitrary_send.sol#54)

(Informational)Low level call in Test.withdraw()
(tests/mini_dataset/arbitrary_send.sol#63-66):

- msg.sender.send(val) (tests/mini_dataset/arbitrary_send.sol#65)

(Informational)Low level call in Test.withdraw_direct()
(tests/mini_dataset/arbitrary_send.sol#68-71):

- msg.sender.send(val) (tests/mini_dataset/arbitrary_send.sol#70)

(Informational)Low level call in Test.withdraw_inderect()
(tests/mini_dataset/arbitrary_send.sol#73-76):

- msg.sender.send(val) (tests/mini_dataset/arbitrary_send.sol#75)

(Informational)Low level call in Test.indirect_ceshi1()
(tests/mini_dataset/arbitrary_send.sol#78-85):

- dd.send(val) (tests/mini_dataset/arbitrary_send.sol#84)

(Informational)Low level call in Test.buy()
(tests/mini_dataset/arbitrary_send.sol#87-92):

- msg.sender.send(remaining) (tests/mini_dataset/arbitrary_send.sol#91)

Security advice

Avoid low-level calls. Check whether the call is successful. If the call is to sign a contract, please check whether the code exists.

4.89 msgvalue-equals-zero

Vulnerability description

msg.value==0 The check condition is meaningless in most cases.

Applicable scenarios

```
contract A{
   address owner;
   mapping(address => uint256) balances;
   constructor() {
      owner = msg.sender;
```

```
    }
    function B() return (uint256){
        if(masg.value == 0) {
            return 0;
        }
        balances[msg.sender] += msg.value;
        return balances[msg.sender];
    }
}
```

Audit results:【Pass】

Security advice: none.

4.90 naming-convention

Vulnerability description

Check whether the naming written in the contract is standardized, because the naming is messy and not easy to understand and manage.

Applicable scenarios: 【Info:9】

For this pattern, the specific problems in the contract are as follows:

(Informational)Function Test.directceshi_1()
(tests/mini_dataset/arbitrary_send.sol#31-34) is not in mixedCase

(Informational)Function Test.directceshi_2()
(tests/mini_dataset/arbitrary_send.sol#36-38) is not in mixedCase

(Informational)Function Test.directceshi_3()
(tests/mini_dataset/arbitrary_send.sol#40-42) is not in mixedCase

(Informational)Function Test.directceshi_4()
(tests/mini_dataset/arbitrary_send.sol#44-46) is not in mixedCase

(Informational)Function Test.directceshi_5()
(tests/mini_dataset/arbitrary_send.sol#48-50) is not in mixedCase

(Informational)Function Test.directceshi_6()
(tests/mini_dataset/arbitrary_send.sol#52-55) is not in mixedCase

(Informational)Function Test.withdraw_direct()
(tests/mini_dataset/arbitrary_send.sol#68-71) is not in mixedCase

(Informational)Function Test.withdraw_inderect()
(tests/mini_dataset/arbitrary_send.sol#73-76) is not in mixedCase

(Informational)Function Test.indirect_ceshi1()
(tests/mini_dataset/arbitrary_send.sol#78-85) is not in mixedCase

Security advice

Please follow Solidity [naming conventions](https://solidity.readthedocs.io/en/v0.4.25/style-guide.html#naming-conventions).

4.91 pragma

Vulnerability description

The use of different Solidity versions (more than two) in the contract will make the compiler not able to compile according to our ideas.

Applicable scenarios

```
pragma solidity ^0.4.23;
pragma solidity ^0.4.24;
```

Audit results:【Pass】

Security advice: none.

4.92 solc-version

Vulnerability description

Solidity source files indicate the version of the compiler that can be compiled with them. It is recommended to indicate a clear version, because future versions of the compiler may handle certain language constructs in ways that developers cannot foresee.

Applicable scenarios: 【Info:1】

For this pattern, the specific problems in the contract are as follows:

(Informational)solc-0.4.24 is not recommended for deployment

Security advice

Use a simple pragma version that allows any of these versions. Consider using the latest version of Solidity for testing. Solidity version recommended: 0.5.11-0.5.13, 0.5.15-0.5.17, 0.6.8, 0.6.10-0.6.11.

4.93 unimplemented-functions

Vulnerability description

Detect functions that are not implemented on most derivative contracts.

Applicable scenarios

```
interface BaseInterface {
    function f1() external returns(uint);
    function f2() external returns(uint);
}
interface BaseInterface2 {
    function f3() external returns(uint);
}
```

```
contract DerivedContract is BaseInterface, BaseInterface2 {
    function f1() external returns(uint){
        return 42;
    }
}
```

DerivedContract does not implement BaseInterface.f2 or BaseInterface2.f3. As a result, the contract will not be compiled correctly. All unrealized functions must be realized on the contract to be used.

Audit results:【Pass】

Security advice: none.

4.94 upgrade-050

Vulnerability description

Check the code update for the Solidity 0.5.0 version. For example: .call() with more than one parameter, keccak256(...) with more than one parameter, etc.

Applicable scenarios

```
contract Token {
    uint totalSupply;
    function Token() {
        totalSupply = +1e18;
    }
    function () payable {}
}
```

Audit results:【Pass】

Security advice: none.

4.95 function-init-state

Vulnerability description

Detect non-fixed variables (the execution result in the function is related to the execution order in the function) to initialize the state variable. The different initialization sequence of the state variable may lead to different initialization values.

Applicable scenarios

```
contract StateVaribleInitFromDynamicFunction {
    uint public v_al = setval(); // Initialize from function (sets to 77)
    uint public w_al = 7;
    uint public x_al = setval(); // Initialize from function (sets to 88)

    constructor(){
    }
    function setval() public returns(uint) {
        if(w_val == 0) {
```

```
            return 4;
        }
        return 5;
    }
}
```

The value of v_al is initialized to 4, and the value of x_al is initialized to 5. However, the caller believes that these two values should be the same. Therefore, this may cause other errors to occur.

Audit results:【Pass】

Security advice: none.

4.96 complex-function

Vulnerability description

Complicated functions will consume more gas. When gas is greater than the set gas limit, the transaction will fail to execute, so every call will fail.

Applicable scenarios

```
contract Complex {
    function a() {
        int numberOfSides = 7;
        string shape;
        uint i0 = 0;
        uint i1 = 0;
        uint i2 = 0;
        uint i3 = 0;
        uint i4 = 0;
        uint i5 = 0;
        uint i6 = 0;
        uint i7 = 0;
        uint i8 = 0;
        uint i9 = 0;
        uint i10 = 0;
        ...
    }
}
```

Bob's call to function a() will exceed the gas limit, and the call will never succeed.

Audit results:【Pass】

Security advice: none.

4.97 hardcoded

Vulnerability description

The contract contains an unknown address, which may be used for some malicious activities. Need to check the hard-coded address and its purpose. The address length is prone to errors, and the length of the address is not

enough, it will not report an error, so it is very dangerous to write amistake. Here is an identification.

Applicable scenarios

```
contract C {
    function f(uint a, uint b) pure returns (address) {
        address public multisig = 0xf64B584972FE6055a770477670208d737Fff282f;
        return multisig;
    }
}
```

Audit results:【Pass】

Security advice: none.

4.98 overpowered-role

Vulnerability description

This function can only be called from one address, so the system relies heavily on this address. In this case, it may cause undesirable consequences for investors. For example, if the private key of the address is compromised, the account will become unusable and the contract will not function properly.

Applicable scenarios

```
contract Crowdsale {
    address public owner;
    uint rate;
    constructor() {
        owner = msg.sender;
    }
    function setRate(_rate) public onlyOwner {
        rate = _rate;
    }
}
```

Audit results:【Pass】

Security advice: none.

4.99 reentrancy-limited-events

Vulnerability description

If a reentrancy error is detected, only reentry vulnerabilities that can cause out-of-sequence events in the restricted gas of read and transfer are reported here.

Applicable scenarios

```
function bug(Called d){
    uint sendeth = 0;
    msg.sender.send(sendeth):
    emit Counter(counter);
}
```

If you reenter "send", the "counter" events will be displayed in an incorrect order, which may cause problems for third parties.

Audit results:【Pass】

Security advice: none.

4.100 reentrancy-limited-gas

Vulnerability description

In the case of natural gas price changes, "send" and "transfer" cannot prevent re-entry, but only show that re-entry is more difficult.

Applicable scenarios

```
function callme(){
   msg.sender.transfer(balances[msg.sender]):
   balances[msg.sender] = 0;
}
```

In the case of natural gas price changes, "send" and "transfer" cannot prevent reentry.

Audit results:【Pass】

Security advice: none.

4.101 reentrancy-limited-gas-no-eth

Vulnerability description

Compared with reentrancy-limited-gas, the send and transfer reentry without eth transfer is detected here.

Applicable scenarios

```
function callme(){
   uint sendeth = 0;
   msg.sender.transfer(sendeth):
   balances[msg.sender] = balances[msg.sender] - sendeth;
}
```

In the case of natural gas price changes, "send" and "transfer" cannot prevent reentry.

Audit results:【Pass】

Security advice: none.

4.102 similar-names

Vulnerability description

Detect variables whose names are too similar.

Applicable scenarios

```
contract SimilarVariables {
    uint similarvariables1 = 1;
    uint similarvariables2 = 2;
    uint similarvariables3 = 3;
}
```

Similar variables make the contract difficult to read.

Audit results:【Pass】

Security advice: none.

4.103 too-many-digits

Vulnerability description

Words with many numbers are difficult to read and view, and variable names are easy to mislead people.

Applicable scenarios

```
contract MyContract{
    uint 1_ether = 10000000000000000000;
}
```

Although 1_ether looks like 1 ether, it is 10 ether. As a result, it is likely to be used incorrectly.

Audit results:【Pass】

Security advice: none.

4.104 private-not-hidedata

Vulnerability description

Contrary to common understanding, the private modifier does not make variables invisible, and miners can access the code and data of all contracts. Developers must solve the problem of Ethereum's lack of privacy. Although it is private, it can be viewed by miners.

Applicable scenarios

```
contract OpenWallet {
    struct Wallet {
        bytes32 password;
        uint balance;
    }
    mapping(uint => Wallet) private wallets;
    function replacePassword(uint _wallet, bytes32 _previous, bytes32 _new) public {
        require(_previous == wallets[_wallet].password);
        wallets[_wallet].password = _new;
    }
}
```

Audit results:【Pass】

Security advice: none.

4.105 safemath

Vulnerability description

SafeMath library is used. It is good to use SafeMath, but if it is modified, it will also cause some loopholes. It should be noted that we call for the use of the safemath library, but care should be taken not to modify it at will.

Applicable scenarios

```
pragma solidity 0.4.24;
import "../libraries/SafeMath.sol";
contract SafeSubAndDiv {
    using SafeMath for uint256;
    function sub(uint a, uint b) public returns(uint) {
        return(a.sub(b));
    }
}
```

Audit results:【Pass】

Security advice: none.

4.106 array-instead-bytes

Vulnerability description

byte[] can be converted to bytes to save gas resources.

Applicable scenarios

```
pragma solidity 0.4.24;
contract C {
    byte[] someVariable;
    ...
}
```

Audit results:【Pass】

Security advice: none.

4.107 boolean-equal

Vulnerability description

Check the comparison of Boolean constants. There is no need to compare with true and false, so it's superfluous (gas consumption).

Applicable scenarios

```
contract A {
    function f(bool x) public {
        // ...
        if (x == true) { // bad!
         // ...
        }
        // ...
    }
```

```
}
```

Boolean constants can be used directly without comparison with true or false.

Audit results:【Pass】

Security advice: none.

4.108 code-no-effects

Vulnerability description

In Solidity, you can write code that does not produce the desired effect. Currently, the solidity compiler will not return warnings for invalid codes. This can lead to the introduction of "dead" code that cannot perform the expected actions correctly. For example, it is easy to omit the parentheses msg.sender.call.value(address(this).balance)("");, which may cause the function to continue execution without transferring funds to msg.sender.

Applicable scenarios

```
pragma solidity ^0.5.0;
contract Wallet {
   mapping(address => uint) balance;
   // Withdraw funds from contract
   function withdraw(uint amount) public {
      require(amount <= balance[msg.sender], 'amount must be less than balance');
      uint previousBalance = balance[msg.sender];
      balance[msg.sender] = previousBalance - amount;
      // Attempt to send amount from the contract to msg.sender
      msg.sender.call.value(amount);
   }
}
```

Audit results:【Pass】

Security advice: none.

4.109 constable-states

Vulnerability description

Constant state variables should be declared as constants to save gas.

Applicable scenarios

```
contract B {
   address public mySistersAddress = 0x999999cf1046e68e36E1aA2E0E07105eDDD1f08E;
   function setUsed(uint a) public {
      if (msg.sender == MY_ADDRESS) {
         used = a;
      myFriendsAddress = 0xc0ffee254729296a45a3885639AC7E10F9d54980;
      }
   }
}
```

Audit results:【Pass】

Security advice: none.

4.110 event-before-revert

Vulnerability description

The event is called before the exception is thrown, and the Revert rollback will make the event waste gas.

Applicable scenarios

```
contract Callbeforerevert {
    address owner;
    event EventName(address bidder, uint amount);
    function bad() public {
        emit EventName(msg.sender, msg.value);
        revert();
    }
}
```

Audit results:【Pass】

Security advice: none.

4.111 external-function

Vulnerability description

Functions with public visibility modifiers are not called internally. Changing the visibility level to an external level can improve the readability of the code. In addition, in many cases, functions that use external visibility modifiers cost less gas than functions that use public visibility modifiers.

Applicable scenarios: 【Opt:15】

For this pattern, the specific problems in the contract are as follows:

(Optimization)direct() should be declared external:

- Test.direct() (tests/mini_dataset/arbitrary_send.sol#13-15)

(Optimization)init() should be declared external:

- Test.init() (tests/mini_dataset/arbitrary_send.sol#17-19)

(Optimization)indirect() should be declared external:

- Test.indirect() (tests/mini_dataset/arbitrary_send.sol#27-29)

(Optimization)directceshi_1() should be declared external:

- Test.directceshi_1() (tests/mini_dataset/arbitrary_send.sol#31-34)

(Optimization)directceshi_2() should be declared external:

- Test.directceshi_2() (tests/mini_dataset/arbitrary_send.sol#36-38)

(Optimization)directceshi_3() should be declared external:

- Test.directceshi_3() (tests/mini_dataset/arbitrary_send.sol#40-42)

(Optimization)directceshi_4() should be declared external:

- Test.directceshi_4() (tests/mini_dataset/arbitrary_send.sol#44-46)

(Optimization)directceshi_5() should be declared external:

- Test.directceshi_5() (tests/mini_dataset/arbitrary_send.sol#48-50)

(Optimization)directceshi_6() should be declared external:

- Test.directceshi_6() (tests/mini_dataset/arbitrary_send.sol#52-55)

(Optimization)repay() should be declared external:

- Test.repay() (tests/mini_dataset/arbitrary_send.sol#59-61)

(Optimization)withdraw() should be declared external:

- Test.withdraw() (tests/mini_dataset/arbitrary_send.sol#63-66)

(Optimization)withdraw_direct() should be declared external:

- Test.withdraw_direct() (tests/mini_dataset/arbitrary_send.sol#68-71)

(Optimization)withdraw_inderect() should be declared external:

- Test.withdraw_inderect() (tests/mini_dataset/arbitrary_send.sol#73-76)

(Optimization)indirect_ceshi1() should be declared external:

- Test.indirect_ceshi1() (tests/mini_dataset/arbitrary_send.sol#78-85)

(Optimization)buy() should be declared external:

- Test.buy() (tests/mini_dataset/arbitrary_send.sol#87-92)

Security advice

Use the "external" attribute for functions that are never called from within the contract.

4.112 extra-gas-inloops

Vulnerability description

Use non-memory array state variables .balance or .length under the condition of for loop or while loop. In this case, each iteration of the loop will consume additional gas.

Applicable scenarios

```
contract NewContract {
   uint[] ss;
   function longLoop() {
      for(uint i = 0; i < ss.length; i++) {
         uint a = ss[i];
         /* ... */
      }
```

```
    }
}
```

Audit results:【Pass】

Security advice: none.

4.113 missing-inheritance

Vulnerability description

Detect lost inheritance.

Applicable scenarios

```
interface ISomething {
    function f1() external returns(uint);
}
contract Something {
    function f1() external returns(uint){
        return 42;
    }
}
```

Some things should inherit from ISomething.

Audit results:【Pass】

Security advice: none.

4.114 redundant-statements

Vulnerability description

Detect the use of invalid statements.

Applicable scenarios

```
contract RedundantStatementsContract {
    constructor() public {
        uint; // Elementary Type Name
        bool; // Elementary Type Name
        RedundantStatementsContract; // Identifier
    }
    function test() public returns (uint) {
        uint; // Elementary Type Name
        assert; // Identifier
        test; // Identifier
        return 777;
    }
}
```

Each comment line references the type/identifier, but does nothing on it, so no code is generated for such statements, so it can be deleted.

Audit results:【Pass】

Security advice: none.

4.115 return-struct

Vulnerability description

Consider using struct instead of multiple return values for internal or private functions, it can improve the readability of the code. Function to pass multiple values using struct.

Applicable scenarios

```
pragma solidity 0.4.24;
contract TestContract {
   function test() internal returns(uint a, address b, bool c, int d) {
      a = 1;
      b = msg.sender;
      c = true;
      d = 2;
   }
}
```

Audit results:【Pass】

Security advice: none.

4.116 revert-require

Vulnerability description

if (condition) {revert(); or throw;} can be replaced by require(condition) to save resources and gas.

Applicable scenarios

```
contract Holder {
   uint public holdUntil;
   address public holder;
   function withdraw (uint a, uint b) external {
      if (now < holdUntil){
         revert();
      }
      holder.transfer(this.balance);
   }
}
```

Audit results:【Pass】

Security advice: none.

4.117 send-transfer

Vulnerability description

The recommended way to perform the check of Ether payment is addr.transfer(x). If the transfer fails, an exception is automatically raised.

Applicable scenarios

```
if(!addr.send(42 ether)) {
   revert();
}
```

Audit results:【Pass】

Security advice: none.

4.118 unused-state

Vulnerability description

Unused variables are allowed in Solidity, and they do not pose direct security issues. The best practice is to avoid them as much as possible: resulting in increased calculations (and unnecessary gas consumption) means errors or incorrect data structures, and usually means poor code quality leads to code noise and reduces code readability.

Applicable scenarios

```
contract A{
    address unused;
    address public unused2;
    address private unused3;
    address unused4;
    address used;
    function ceshi1 () external{
        unused3 = address(0);
    }
}
```

Audit results:【Pass】

Security advice: none.

4.119 costly-operations-loop

Vulnerability description

Expensive operations within the loop.

Applicable scenarios

```
contract CostlyOperationsInLoop{
    uint loop_count = 100;
    uint state_variable=0;
    function bad() external{
        for (uint i=0; i < loop_count; i++){
            state_variable++;
        }
    }
    function good() external{
     uint local_variable = state_variable;
     for (uint i=0; i < loop_count; i++){
        local_variable++;
     }
     state_variable = local_variable;
    }
}
```

Due to the expensive SSTOREs, the incremental state variables in the loop will generate a large amount of gas, which may result in insufficient gas.

Audit results:【Pass】

Security advice: none.

Thanks for using the SmartFast contract audit platform!