

# Metasecurelabs analysis report

metasecurelabs.io

November 9, 2022

## 1 Introduction

### 1.1 storage\_signed\_integer\_array

**SWC\_ID:**

**Description:** solc versions 0.4.7–0.5.10 contain a compiler bug leading to incorrect values in signed integer arrays.

**Example:**

```
contract A {
    int[3] ether\textunderscore balances; // storage signed integer array
    function bad0() private {
        // ...
        ether\textunderscore balances = [\textendash 1, \textendash 1, \textendash 1];
        // ...
    }
}
```

bad0() uses a (storage\textendash allocated) signed integer array state variable to stor

```
}
}
```

**DASP** : Unknown unknowns  
**Found**: false

### 1.2 modifier\_like\_Sol\_keyword

**SWC\_ID:**

**Description:** A contract may contain modifier that looks similar to Solidity keyword

**Example:**

```
contract Contract{
    modifier public() {
    }
```

```

        function doSomething() public {
            require(owner == msg.sender);
            owner = newOwner;
        }
    }
}

```

public is a modifier meant to look like a Solidity keyword.

```

    }
}
DASP : Unknown Unknowns
Found: false

```

### 1.3 arbitrary\_from\_in\_transferFrom

#### SWC ID:

**Description:**Something wrong happens when msg.sender is not used as ‘from’ in transferFrom.

#### Example:

```

function a(address from, address to, uint256 amount) public {
    erc20.transferFrom(from, to, am);
}

```

Alice approves this contract to spend her ERC20 tokens. Bob can call a and specify Alice

```

    }
}
DASP : Unknown unknowns
Found: false

```

### 1.4 arithmetic

#### SWC ID:

**Description:**This bug type consists of various arithmetic bugs: integer overflow/underflow, division issues, . \* Integer overflow/underflow. An overflow/underflow happens when an arithmetic operation reaches the maximum or minimum size of a type. For instance if a number is stored in the uint8 type, it means that the number is stored in a 8 bits unsigned number ranging from 0 to  $2^8-1$ . In computer programming, an integer overflow occurs when an arithmetic operation attempts to create a numeric value that is outside of the range that can be represented with a given number of bits either larger than the maximum or lower than the minimum representable value. \* Division issues. Some wrong

will happen when integer or float numbers are divided by zero. \* Type deduction overflow. In Solidity, when declaring a variable as type var, the compiler uses type deduction to automatically infer the smallest possible type from the first expression that is assigned to the variable. Thus, the deduced type may not be appropriate, and it can incur overflow bugs later (see the example).

**Example:**

```
Integer overflow/underflow
/*
 * @source: https://capturetheether.com/challenges/math/token\textendash sale/
 * @author: Steve Marx
 */
pragma solidity \textsuperscript0.4.21;
contract TokenSaleChallenge {
    mapping(address => uint256) public balanceOf;
    uint256 constant PRICE\textunderscore PER\textunderscore TOKEN = 1 ether;

    function TokenSaleChallenge(address \textunderscore player) public payable {
        require(msg.value == 1 ether);
    }

    function isComplete() public view returns (bool) {
        return address(this).balance < 1 ether;
    }

    function buy(uint256 numTokens) public payable {
        require(msg.value == numTokens * PRICE\textunderscore PER\textunderscore TOKEN);

        balanceOf[msg.sender] += numTokens;
    }
}

Division issues
contract Division {

    /*function unsigned\textunderscore division(uint32 x, uint32 y) returns (int r) {
        //if (y == 0) { throw; }
        r = x / y;
    }*/

    function signed\textunderscore division(int x, int y) returns (int) {
        //if ((y == 0) || ((x == \textendash 2**255) \& \& (y == \textendash 1))) { throw
        return x / y;
    }
}
```

Type deduction overflow

```
contract For\textunderscore Test {
...
function Test () payable public {
  if ( msg . value > 0.1 ether ) {
    uint256 multi = 0;
    uint256 amountToTransfer = 0;
    for ( var i = 0; i < 2* msg . value ; i ++ ) {
      multi = i *2;
      if ( multi < amountToTransfer ) {
        break ;
        amountToTransfer = multi ;
      }
      msg.sender.transfer( amountToTransfer );
    }
  }
}

}
DASP : Arithmetic
Found: true
```

## 1.5 storage\_ABIEncoderV2\_array

**SWC\_ID:**

**Description:** solc versions 0.4.7–0.5.9 contain a compiler bug leading to incorrect ABI encoder usage.

**Example:**

```
contract A {
  uint[2][3] bad\textunderscore 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\textunderscore arr);
  }
}
```

abi.encode(bad\textunderscore arr) in a call to bad() will incorrectly encode the array

```
}
}
DASP : Unknown unknowns
Found: false
```

## 1.6 dead\_code

### SWC\_ID:

**Description:**In Solidity, it's possible to write code that does not produce the intended effects. Currently, the solidity compiler will not return a warning for effect-free code. This can lead to the introduction of "dead" code that does not properly performing an intended action.

For example, it's easy to miss the trailing parentheses in `msg.sender.call.value(xx)("")`;;, which could lead to a function proceeding without transferring funds to `msg.sender`. Also, internal functions could be 'dead' when they are not invoked.

### Example:

```
pragma solidity ^0.5.0;

contract DepositBox {
    mapping(address => uint) balance;

    // Accept deposit
    function deposit(uint amount) public payable {
        require(msg.value == amount, 'incorrect amount');
        // Should update user balance
        balance[msg.sender] = amount;
    }
}

}

}
DASP : Unknown unknowns
Found: false
```

## 1.7 func\_modifying\_storage\_array\_by\_value

### SWC\_ID:

**Description:**Arrays passed to a function that expects reference to a storage array.

### Example:

```
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()`. Bob assumes that at the end of the call `x[0]` is 2, but it is 1. As a result

```

}
}
DASP : Unknown Unknowns
Found: false

```

## 1.8 strict\_balance\_equality

### SWC\_ID:

**Description:** Contracts can behave erroneously when they strictly assume a specific Ether balance. It is always possible to forcibly send ether to a contract (without triggering its fallback function), using `selfdestruct`, or by mining to the account. In the worst case scenario this could lead to DOS conditions that might render the contract unusable.

#### Example:

```

if (address(this).balance == 42 ether ) {
    /* ... */
}

secure alternative:

if (address(this).balance >= 42 ether ) {
    /* ... */
}

}
}
DASP : Unknown unknowns
Found: false

```

## 1.9 overpowered\_role

### SWC\_ID:

**Description:** This function is callable only from one address. Therefore, the system depends heavily on this address. In this case, there are scenarios that

may lead to undesirable consequences for investors, e.g. if the private key of this address becomes compromised.

**Example:**

```
pragma solidity 0.4.25;

contract Crowdsale {

    address public owner;

    uint rate;
    uint cap;

    constructor() {
        owner = msg.sender;
    }

    function setRate(uint rate) public onlyOwner {
        rate = rate;
    }

    function setCap(uint cap) public {
        require (msg.sender == owner);
        cap = cap;
    }
}

}
}
DASP : Unknown unknowns
Found: false
```

## 1.10 erc20\_event\_not\_indexed

**SWC\_ID:**

**Description:**Events defined by the ERC20 specification that should have some parameters as indexed.

**Example:**

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

    // ...
}
```

Transfer and Approval events should have the 'indexed' keyword on their two first parameters

```
}
}
DASP : Unknown unknowns
Found: false
```

### 1.11 unused\_retval

#### SWC ID:

**Description:**The return value of an external call is not stored in a local or state variable.

#### Example:

```
contract MyConc{
    using SafeMath for uint;
    function my\textunderscore func(uint a, uint b) public{
        a.add(b);
    }
}
```

MyConc calls add of SafeMath, but does not store the result in a. As a result, the computation is wasted.

```
}
}
DASP : Unknown Unknowns
Found: true
```

### 1.12 extra\_gas\_in\_loops

#### SWC ID:

**Description:**State variable, .balance, or .length of non-memory array is used in the condition of for or while loop. In this case, every iteration of loop consumes extra gas.

#### Example:

In the following example, limiter variable is accessed on every for\textendash loop iteration.

```
pragma solidity 0.4.25;

contract NewContract {
    uint limiter = 100;

    function longLoop() {
```



```

        for(uint i = 0; i < limiter; i++) {
            /* ... */
        }
    }
}

```

```

}
}
DASP : Unknown unknowns
Found: false

```

### 1.13 uninitialized\_state\_variable

#### SWC ID:

**Description:**Some unexpected error may happen when state variables are not uninitialized.

#### Example:

```

contract Uninitialized{
    address destination;

    function transfer() payable public{
        destination.transfer(msg.value);
    }
}

```

```

}
}
DASP : Unknown unknowns
Found: true

```

### 1.14 pre-declare\_usage\_of\_local

#### SWC ID:

**Description:**Using a variable before the declaration is stepped over (either because it is later declared, or declared in another scope).

#### Example:

```

contract C {
    function f(uint z) public returns (uint) {
        uint y = x + 9 + z; // 'z' is used pre\textendash 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
    for (uint i = 0; i < max; i++) {
        x += 1;
    }

    return x;
}
}

}
}
DASP : Unknown unknowns
Found: false

```

### 1.15 race\_condition

#### SWC ID:

**Description:** Since miners always get rewarded via gas fees for running code on behalf of externally owned addresses (EOA), users can specify higher fees to have their transactions mined more quickly. Since the Ethereum blockchain is public, everyone can see the contents of others' pending transactions. This means if a given user is revealing the solution to a puzzle or other valuable secret, a malicious user can steal the solution and copy their transaction with higher fees to preempt the original solution. If developers of smart contracts are not careful, this situation can lead to practical and devastating front-running attacks.

#### Example:

In this example, one can front\textendash run transactions to claim his/her reward before

```
pragma solidity \textsuperscript{0.4.16};
```

```

contract EthTxOrderDependenceMinimal {
    address public owner;
    bool public claimed;
    uint public reward;

    function EthTxOrderDependenceMinimal() public {
        owner = msg.sender;
    }

    function setReward() public payable {
        require (!claimed);
    }
}

```

```

        require(msg.sender == owner);
        owner.transfer(reward);
        reward = msg.value;
    }

    function claimReward(uint256 submission) {
        require (!claimed);
        require(submission < 10);
        msg.sender.transfer(reward);
        claimed = true;
    }
}

}
}
DASP : Front Running
Found: true

```

### 1.16 unchecked\_calls

#### SWC\_ID:

**Description:** The return value of a message call is not checked. Execution will resume even if the called contract throws an exception. If the call fails accidentally or an attacker forces the call to fail, this may cause unexpected behaviour in the subsequent program logic.

#### Example:

```

pragma solidity 0.4.25;

contract ReturnValue {

    checked
    function callchecked(address callee) public {
        require(callee.call());
    }

    function callnotchecked(address callee) public {
        callee.call();
    }
}

}

}
DASP : Unchecked Low Level Calls
Found: true

```

### 1.17 locked\_money

#### SWC ID:

**Description:** Contracts programmed to receive ether should implement a way to withdraw it, i.e., call transfer (recommended), send, or call.value at least once..

#### Example:

In the following example, contracts programmed to receive ether does not call transfer, ,

```
pragma solidity 0.4.25;
```

```
contract BadMarketPlace {  
    function deposit() payable {  
        require(msg.value > 0);  
    }  
}
```

```
}
```

```
}
```

**DASP** : Unknown unknowns

**Found:** false

### 1.18 incorrect\_ERC20\_interface

#### SWC ID:

**Description:** Incorrect return values for ERC20 functions. A contract compiled with Solidity < 0.4.22 interacting with these functions will fail to execute them, as the return value is missing.

#### Example:

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

Token.transfer does not return a boolean. Bob deploys the token. Alice creates a contract

```
}
```

```
}
```

**DASP** : Unknown Unknowns

**Found:** false

## 1.19 unused\_function\_should\_be\_external

### SWC\_ID:

**Description:** A function with public visibility modifier that is not called internally. Changing visibility level to external increases code readability. Moreover, in many cases functions with external visibility modifier spend less gas comparing to functions with public visibility modifier.

### Example:

In the following example, functions with both public and external visibility modifiers are

```
contract Token {

    mapping (address => uint256) internal \textunderscore balances;

    function transfer\textunderscore public(address to, uint256 value) public {
        require(value <= \textunderscore balances[msg.sender]);

        \textunderscore balances[msg.sender] \textendash = value;
        \textunderscore balances[to] += value;
    }

    function transfer\textunderscore external(address to, uint256 value) external {
        require(value <= \textunderscore balances[msg.sender]);

        \textunderscore balances[msg.sender] \textendash = value;
        \textunderscore balances[to] += value;
    }
}
```

The second function requires less gas.

```
}
}
DASP : Unknown unknowns
Found: true
```

## 1.20 uninitialized\_func\_pointer

### SWC\_ID:

**Description:** this.balance will include the value sent by msg.value, which might lead to incorrect computation.

### Example:

```
contract Bug{
    function buy() public payable{
```

```

        uint minted = msg.value * (1000 / address(this).balance);
        // ...
    }
}

```

buy is meant to compute a price that changes a ratio over the contract's balance. .balance

```

}
}
DASP : Unknown unknowns
Found: false

```

## 1.21 reentrancy

### SWC ID:

**Description:**One of the major dangers of calling external contracts is that they can take over the control flow. In the reentrancy attack (a.k.a. recursive call attack), a malicious contract calls back into the calling contract before the first invocation of the function is finished. This may cause the different invocations of the function to interact in undesirable ways.

#### Example:

```

/*
 * @source: http://blockchain.unica.it/projects/ethereum/textendash_survey/attacks.htmlsin
 * @author: \textendash
 * @vulnerable\textunderscore at\textunderscore lines: 19
 */

pragma solidity \textsuperscript0.4.2;

contract SimpleDAO {
    mapping (address => uint) public credit;

    function donate(address to) payable {
        credit[to] += msg.value;
    }

    function withdraw(uint amount) {
        if (credit[msg.sender]>= amount) {
            // <yes> <report> REENTRANCY
            bool res = msg.sender.call.value(amount)();
            credit[msg.sender]\textendash =amount;
        }
    }
}

```

```

}
}
DASP : Reentrancy
Found: true

```

## 1.22 visibility

### SWC\_ID:

**Description:**The default function visibility level in contracts is public, in interfaces –external, and the state variable default visibility level is internal. In contracts, the fallback function can be external or public. In interfaces, all the functions should be declared as external. Explicitly define function visibility to prevent confusion. Additionally, the visibility of state variables could be a problem. labeling the visibility explicitly makes it easier to catch incorrect assumptions about who can access the variable.

#### Example:

In this example, a specific modifier, such as public, is not used when declaring a function.

```
function foo();
```

Preferred alternatives:

```
function foo() public;
function foo() internal;
```

```

}
}
DASP : Unknown Unknowns
Found: true

```

## 1.23 state\_variable\_shadowing

### SWC\_ID:

**Description:**Solidity allows for ambiguous naming of state variables when inheritance is used. Contract A with a variable x could inherit contract B that also has a state variable x defined. This would result in two separate versions of x, one of them being accessed from contract A and the other one from contract B. In more complex contract systems this condition could go unnoticed and subsequently lead to security issues.

#### Example:

```
pragma solidity 0.4.25;

contract Tokensale {
    uint public hardcap = 10000 ether;
```

```

    function Tokensale() {}

    function fetchCap() public constant returns(uint) {
        return hardcap;
    }
}

contract Presale is Tokensale {
    //uint hardcap = 1000 ether;
    //If the hardcap variables were both needed we would have to rename one to fix this.
    function Presale() Tokensale() {
        hardcap = 1000 ether;
    }
}

}
}
DASP : Unknown Unknowns
Found: false

```

## 1.24 call\_without\_data

### SWC\_ID:

**Description:**Using low-level call function with no arguments provided.

### Example:

In the following example, call function is used for ETH transfer:  
pragma solidity 0.4.24;

```

contract MyContract {

    function withdraw() {
        if (msg.sender.call.value(1)()) {
            /*...*/
        }
    }
}

}
}
DASP : Unknown unknowns
Found: false

```



## 1.25 incorrect\_modifier

### SWC ID:

**Description:** If a modifier does not execute `_or` revert, the execution of the function will return the default value, which can be misleading for the caller.

### Example:

```
modifier myModif(){
    if(..){
        \textunderscore ;
    }
}
function get() myModif returns(uint){}
```

If the condition in `myModif` is false, the execution of `get()` will return 0.

```
}
}
DASP : Unknown unknowns
Found: false
```

## 1.26 builtin\_symbol\_shadowing

### SWC ID:

**Description:** Something wrong may happen when built-in symbols are shadowed by local variables, state variables, functions, modifiers, or events.

### Example:

```
pragma solidity \textsuperscript{0.4.24};

contract Bug {
    uint now; // Overshadows current time stamp.

    function assert(bool condition) public {
        // Overshadows built\textendash in symbol for providing assertions.
    }

    function get\textunderscore next\textunderscore expiration(uint earlier\textunderscore timestamp)
        return now + 259200; // References overshadowed timestamp.
    }
}

}
}
DASP : Unknown unknowns
Found: false
```

## 1.27 address\_hardcoded

### SWC\_ID:

**Description:**The contract contains unknown address. This address might be used for some malicious activity. Please check hardcoded address and it's usage.

### Example:

In the following contract, the address is specified in the source code:

```
pragma solidity 0.4.24;
contract C {
    function f(uint a, uint b) pure returns (address) {
        address public multisig = 0xf64B584972FE6055a770477670208d737Fff282f;
        return multisig;
    }
}
```

Do not forget to check the contract at the address 0xf64B584972FE6055a770477670208d737Ff.

```
}
}
DASP : Unknown unknowns
Found: false
```

## 1.28 wrong\_signature

### SWC\_ID:

**Description:**In Solidity, the function signature is defined as the canonical expression of the basic prototype without data location specifier, i.e. the function name with the parenthesised list of parameter types. Parameter types are split by a single comma –no spaces are used. This means one should use uint256 and int256 instead of uint or int.

### Example:

This code uses incorrect function signature:

```
pragma solidity \textsuperscript{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;
    }
}
```

Use "foo(uint256)" instead.

```
}
}
DASP : Unknown Unknowns
Found: false
```

### 1.29 msg.value\_in\_loop

#### SWC ID:

**Description:**It is error-prone to use msg.value inside a loop.

#### Example:

```
contract MsgValueInLoop{
    mapping (address => uint256) balances;

    function bad(address[] memory receivers) public payable {
        for (uint256 i=0; i < receivers.length; i++) {
            balances[receivers[i]] += msg.value;
        }
    }
}
```

msg.value should be tracked through a local variable and decrease its amount on every it

```
}
}
DASP : Unknown unknowns
Found: false
```

### 1.30 right\_to\_left\_char

#### SWC ID:

**Description:**Malicious actors can use the Right-To-Left-Override unicode character to force RTL text rendering and confuse users as to the real intent of a contract.

#### Example:

```
/*
 * @source: https://youtu.be/P\textunderscore Mtd5Fc\textunderscore 3E
 * @author: Shahar Zini
 */
pragma solidity \textsuperscript0.5.0;

contract GuessTheNumber
```

```

{
    uint \textunderscore secretNumber;
    address payable \textunderscore owner;
    event success(string);
event wrongNumber(string);

    function guess(uint n) payable public
    {
        require(msg.value == 1 ether);

        uint p = address(this).balance;
        checkAndTransferPrize(/*The prize/*rebmun desseug*/n , p/*
                               /*The user who should benefit *//,msg.sender);
    }

    function checkAndTransferPrize(uint p, uint n, address payable guesser) internal retu
    {
        if(n == \textunderscore secretNumber)
        {
            guesser.transfer(p);
            emit success("You guessed the correct number!");
        }
        else
        {
            emit wrongNumber("You've made an incorrect guess!");
        }
    }
}

}
}
DASP : Unknown Unknowns
Found: false

```

### 1.31 `local_variable_shadowing`

#### **SWC ID:**

**Description:** Something wrong may happen when local variables shadowing state variables or other local variables.

#### **Example:**

```

pragma solidity \textsuperscript0.4.24;

contract Bug {
    uint owner;

```

```

function sensitive\textunderscore function(address owner) public {
    // ...
    require(owner == msg.sender);
}

function alternate\textunderscore sensitive\textunderscore function() public {
    address owner = msg.sender;
    // ...
    require(owner == msg.sender);
}
}

sensitive\textunderscore function.owner shadows Bug.owner. As a result, the use of owner

}
}
DASP : Unknown unknowns
Found: false

```

### 1.32 use\_after\_delete

#### SWC\_ID:

**Description:**Using values of variables after they have been explicitly deleted may lead to unexpected behavior or compromise.

#### Example:

```

mapping(address => uint) public balances;
function f() public {
    delete balances[msg.sender];
    msg.sender.transfer(balances[msg.sender]);
}

```

`balances[msg.sender]` is deleted before it's sent to the caller, leading the transfer to

```

}
}
DASP : Unknown unknowns
Found: false

```

### 1.33 incorrect\_shift\_in\_assembly

#### SWC\_ID:

**Description:**The values in a shift operation could be reversed (in a wrong order)

#### Example:

```

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

}
}
DASP : Unknown Unknowns
Found: false

```

### 1.34 deprecated\_standards

#### SWC\_ID:

**Description:** Several functions and operators in Solidity are deprecated. Using them leads to reduced code quality. With new major versions of the Solidity compiler, deprecated functions and operators may result in side effects and compile errors. Deprecated Alternative suicide(address) selfdestruct(address) block.blockhash(uint) blockhash(uint) sha3(...) keccak256(...) callcode(...) delegatecall(...) throw revert() msg.gas gasleft constant view var corresponding type name

#### Example:

```

pragma solidity 0.4.24;

contract BreakThisHash {
    bytes32 hash;
    uint birthday;
    constructor(bytes32 \textunderscore hash) public payable {
        hash = \textunderscore hash;
        birthday = now;
    }

    function kill(bytes password) external {
        if (sha3(password) != hash) {
            throw;
        }
        suicide(msg.sender);
    }

    function hashAge() public constant returns(uint) {
        return(now \textendash birthday);
    }
}

```

Use keccak256, selfdestruct, revert() instead.

```
}
}
DASP : Unknown unknowns
Found: false
```

### 1.35 costly\_ops\_in\_loop

**SWC\_ID:**

**Description:** Costly operations inside a loop might waste gas, so optimizations are justified.

**Example:**

```
contract CostlyOperationsInLoop{

    uint loop\textunderscore count = 100;
    uint state\textunderscore variable=0;

    function bad() external{
        for (uint i=0; i < loop\textunderscore count; i++){
            state\textunderscore variable++;
        }
    }

    function good() external{
        uint local\textunderscore variable = state\textunderscore variable;
        for (uint i=0; i < loop\textunderscore count; i++){
            local\textunderscore variable++;
        }
        state\textunderscore variable = local\textunderscore variable;
    }
}
```

Incrementing state\textunderscore variable in a loop incurs a lot of gas because of expensive

```
}
}
DASP : Unknown Unknowns
Found: false
```

### 1.36 function\_declared\_return\_but\_no\_return

**SWC\_ID:**

**Description:** Function doesn't initialize return value. As result default value will be returned.

**Example:**

In the following example, the function's signature only denotes the type of the return value.

```
pragma solidity 0.4.25;
```

```
contract NewContract {
    uint minimumBuy;

    function setMinimumBuy(uint256 newMinimumBuy) returns (bool){
        minimumBuy = newMinimumBuy;
    }
}
```

```
}
```

```
}
```

**DASP** : Unknown unknowns

**Found**: false

### 1.37 multiple\_constructor\_schemes

**SWC ID:**

**Description:**Multiple constructor definitions in the same contract (using new and old schemes).

**Example:**

```
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 both constructor schemes will compile. The first constructor

```
}
```

```
}
```

**DASP** : Unknown unknowns

**Found**: false



### 1.38 byte\_array\_instead\_bytes

**SWC\_ID:**

**Description:** Use bytes instead of byte[] for lower gas consumption.

**Example:**

In the following example, byte array is used:

```
pragma solidity 0.4.24;

contract C {
    byte[] someVariable;
    ...
}
```

Alternative:

```
pragma solidity 0.4.24;

contract C {
    bytes someVariable;
    ...
}
```

```
}
}
DASP : Unknown Unknowns
Found: false
```

### 1.39 short\_addresses

**SWC\_ID:**

**Description:** MISSING

**Example:**

MISSING

```
}
}
DASP : Unknown unknowns
Found: false
```

### 1.40 uninitialized\_storage\_pointer

**SWC\_ID:**

**Description:**An uninitialized storage variable will act as a reference to the first state variable, and can override a critical variable.

**Example:**

```
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 overridden to 0.

```
}
}
DASP : Unknown Unknowns
Found: false
```

#### 1.41 pausable\_modifier\_absence

**SWC\_ID:**

**Description:**ERC20 balance/allowance is modified without whenNotPaused modifier (in pausable contract).x

**Example:**

```
function buggyTransfer(address to, uint256 value) external returns (bool){
    balanceOf[msg.sender] \textendash = value;
    balanceOf[to] += value;
    return true;
}
```

In a pausable contract, buggyTransfer performs a token transfer but does not use Pausabl

```
}
}
DASP : Unknown unknowns
Found: false
```

## 1.42 useless\_compare

### SWC ID:

**Description:** A variable compared to itself is probably an error as it will always return true for `==`, `!=`, `==` and always false for `!`, `!` and `!=`. In addition, some comparison are also tautologies or contradictions.

#### Example:

```
function check(uint a) external returns(bool){
    return (a >= a);
}
```

```
}
```

```
}
```

**DASP** : Unknown unknowns

**Found**: false

## 1.43 benign\_reentrancy

### SWC ID:

**Description:** Some re-entrancy bugs have no adverse effect since its exploitation would have the same effect as two consecutive calls.

#### Example:

```
function callme(){
    if( ! (msg.sender.call()( ) ) ){
        throw;
    }
    counter += 1
}
```

`callme()` contains a benign reentrancy.

```
}
```

```
}
```

**DASP** : Unknown unknowns

**Found**: false

## 1.44 divide\_before\_multiply

### SWC ID:

**Description:** Solidity operates only with integers. Thus, if the division is done before the multiplication, the rounding errors can increase dramatically. Vulnerability type by SmartDec classification: Precision issues.

#### Example:

In the following example, amount variable is divided by DELIMITER and then multiplied by

```
pragma solidity 0.4.25;
```

```
contract MyContract {  
  
    uint constant BONUS = 500;  
    uint constant DELIMITER = 10000;  
  
    function calculateBonus(uint amount) returns (uint) {  
        return amount/DELIMITER*BONUS;  
    }  
}
```

```
}  
}  
}  
DASP : Unknown Unknowns  
Found: false
```

#### 1.45 should\_be\_pure

**SWC ID:**

**Description:**In Solidity, function that do not read from the state or modify it can be declared as pure.

**Example:**

Here is the example of correct pure\textendash function:

```
pragma solidity \textsuperscript0.4.16;
```

```
contract C {  
    function f(uint a, uint b) pure returns (uint) {  
        return a * (b + 42) + now;  
    }  
}
```

```
}  
}  
DASP : Unknown unknowns  
Found: false
```

#### 1.46 del\_structure\_containing\_mapping

**SWC ID:**

**Description:** A deletion in a structure containing a mapping will not delete the mapping (see the Solidity documentation). The remaining data may be used to compromise the contract.

**Example:**

```
struct BalancesStruct{
    address owner;
    mapping(address => uint) balances;
}
mapping(address => BalancesStruct) public stackBalance;

function remove() internal{
    delete stackBalance[msg.sender];
}

}
}
DASP : Unknown unknowns
Found: false
```

#### 1.47 msg.value\_equals\_zero

**SWC ID:**

**Description:** The `msg.value == 0` condition check is meaningless in most cases.

**Example:**

```
msg.value == 0

}
}
DASP : Unknown unknowns
Found: false
```

#### 1.48 unused\_state\_variables

**SWC ID:**

**Description:** Unused variables are allowed in Solidity and they do not pose a direct security issue. It is best practice though to avoid them as they can: \* cause an increase in computations (and unnecessary gas consumption) \* indicate bugs or malformed data structures and they are generally a sign of poor code quality \* cause code noise and decrease readability of the code

**Example:**

```

pragma solidity >=0.5.0;
pragma experimental ABIEncoderV2;

import "./base.sol";

contract DerivedA is Base {
    // i is not used in the current contract
    A i = A(1);

    int internal j = 500;

    function call(int a) public {
        assign1(a);
    }

    function assign3(A memory x) public returns (uint) {
        return g[1] + x.a + uint(j);
    }

    function ret() public returns (int){
        return this.e();
    }

    int internal j = 500;
    function call(int a) public {
        assign1(a);
    }

    function assign3(A memory x) public returns (uint) {
        return g[1] + x.a + uint(j);
    }

    function ret() public returns (int){
        return this.e();
    }
}

}
}
DASP : Unknown unknowns
Found: false

```

#### 1.49 denial\_of\_service

**SWC ID:**

**Description:** Denial of service (DoS) is deadly in the world of Ethereum: while other types of applications can eventually recover, smart contracts can be taken offline forever by just one of these attacks. DoS can happen in the following cases: \* External calls can fail accidentally or deliberately, which can cause a DoS condition in the contract. Particularly, DoS would happen if there is a loop where external calls are not isolated. \* A large number of loops may consume gas, so it is possible that the function exceeds the block gas limit, and transactions calling it will never be confirmed. \* An inappropriate type inference in the loop (e.g., literal `-1`, `uint8`) may cause an infinite loop. \* Recursive external calls may consume a large number of callstacks, which may lead to DoS.

**Example:**

```
for (var i = 0; i < array.length; i++) { /* ... */
}
}
```

**DASP :** Denial of Services

**Found:** true

## 1.50 array\_length\_manipulation

**SWC ID:**

**Description:** The length of the dynamic array is changed directly. In the following case, the appearance of gigantic arrays is possible and it can lead to a storage overlap attack (collisions with other data in storage).

**Example:**

```
pragma solidity 0.4.24;

contract dataStorage {
    uint[] public data;

    function writeData(uint[] \textunderscore data) external {
        for(uint i = data.length; i < \textunderscore data.length; i++) {
            data.length++;
            data[i]=\textunderscore data[i];
        }
    }
}

}
```

**DASP :** Unknown Unknowns

**Found:** false

## 1.51 constant\_state\_variable

### SWC ID:

**Description:** There is a conflict if the same base constructor is called with arguments from two different locations in the same inheritance hierarchy.

### Example:

```
pragma solidity \textsuperscript{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 { /* ... */ }
}

}
```

**DASP** : Unknown unknowns

**Found:** false

## 1.52 access\_control

### SWC ID:

**Description:** Access Control issues are common in all programs, not just smart contracts. In fact, it's number 5 on the OWASP top 10. One usually accesses a contract's functionality through its public or external functions. While insecure visibility settings give attackers straightforward ways to access a contract's private values or logic, access control bypasses are sometimes more subtle. These vulnerabilities can occur in the following cases: \* Contracts use the deprecated tx.origin to validate callers \* Handling large authorization logic with lengthy require \* Making reckless use of delegatecall in proxy libraries or proxy contracts. Delegate calling into untrusted contracts is very dangerous, as the code at the target address can change any storage values of the caller and



has full control over the caller's balance. \* Due to missing or insufficient access controls, malicious parties can withdraw some or all Ether from the contract account. \* Due to missing or insufficient access controls, malicious parties can self-destruct the contract.

**Example:**

```
contract TestContract is MultiOwnable {

    function withdrawAll(){
        msg.sender.transfer(this.balance);
    }
}

}
DASP : Access control
Found: true
```

### 1.53 ignore

**SWC ID:**

**Description:**Other trivial bug types.

**Example:**

```
}
}
DASP : Unknown Unknowns
Found: true
```

### 1.54 controlled\_lowlevel\_call

**SWC ID:**

**Description:**Low-level call with a user-controlled data field

**Example:**

```
address token;

function call\textunderscore token(bytes data){
    token.call(data);
}
```

token' points to an ERC20 token. Bob uses call\textunderscore token to call the transfer

```

}
}
DASP : Unknown Unknowns
Found: true

```

## 1.55 dangerous\_enum\_conversion

### SWC\_ID:

**Description:**out-of-range enum conversion may occur (solc j 0.4.5).

### Example:

```

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

```

```

}
}
DASP : Unknown Unknowns
Found: false

```

## 1.56 should\_be\_view

### SWC\_ID:

**Description:**In Solidity, functions that do not read from the state or modify it can be declared as view.

### Example:

Here is the example of correct view\textendash function:

```

contract C {
    function f(uint a, uint b) view returns (uint) {
        return a * (b + 42) + now;
    }
}

```

```

}
}
DASP : Unknown unknowns
Found: false

```

## 1.57 uninitialized\_local\_variable

### SWC ID:

**Description:**Some unexpected error may happen when local variables are not uninitialized.

### Example:

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

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

```
}
}
DASP : Unknown unknowns
Found: false
```

## 1.58 reused\_base\_constructors

### SWC ID:

**Description:**There is a conflict if the same base constructor is called with arguments from two different locations in the same inheritance hierarchy.

### Example:

```
pragma solidity \textsuperscript{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 { /* ... */ }
    }

}
}
DASP : Unknown unknowns
Found: false

```

## 1.59 blockhash\_current

### SWC\_ID:

**Description:** blockhash function returns a non-zero value only for 256 last blocks. Besides, it always returns 0 for the current block, i.e. blockhash(block.number) always equals to 0.

### Example:

In the following example, currentBlockBlockhash function always returns 0:

```

pragma solidity 0.4.25;

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

}
}
DASP : Unknown unknowns
Found: false

```

## 1.60 payable\_func\_using\_delegatecall\_in\_loop

### SWC\_ID:

**Description:** The same msg.value amount may be incorrectly accredited multiple times when using delegatecall inside a loop in a payable function.

### Example:

```

contract DelegatecallInLoop{

    mapping (address => uint256) balances;

    function bad(address[] memory receivers) public payable {
        for (uint256 i = 0; i < receivers.length; i++) {
            address(this).delegatecall(abi.encodeWithSignature("addBalance(address)", re

```

```

    }
}

function addBalance(address a) public payable {
    balances[a] += msg.value;
}
}

}
}
DASP : Unknown Unknowns
Found: false

```

### 1.61 using\_send

#### SWC\_ID:

**Description:** The send function is called inside checks instead of using transfer. The recommended way to perform checked ether payments is `addr.transfer(x)`, which automatically throws an exception if the transfer is unsuccessful.

#### Example:

In the following example, the send function is used:

```

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

```

Preferred alternative:

```

addr.transfer(42 ether);

```

```

}
}
DASP : Unknown Unknowns
Found: false

```

### 1.62 time\_manipulation

#### SWC\_ID:

**Description:** From locking a token sale to unlocking funds at a specific time for a game, contracts sometimes need to rely on the current time. This is usually done via `block.timestamp` or its alias `now` in Solidity. But where does that value come from? From the miners! Because a transaction's miner has leeway

in reporting the time at which the mining occurred, good smart contracts will avoid relying strongly on the time advertised.

**Example:**

```
contract TimedCrowdsale
    event Finished();
    event notFinished();

    // Sale should finish exactly at January 1, 2019
    function isSaleFinished() private returns (bool) {
        return block.timestamp >= 1546300800;
    }

    function run() public {
        if (isSaleFinished()) {
            emit Finished();
        } else {
            emit notFinished();
        }
    }
}

}
DASP : Time Manipulation
Found: false
```

### 1.63 incorrect\_ERC721\_interface

**SWC\_ID:**

**Description:**Incorrect return values for ERC721 functions. A contract compiled with solidity < 0.4.22 interacting with these functions will fail to execute them, as the return value is missing.

**Example:**

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

}
DASP : Unknown unknowns
Found: false
```

## 1.64 redundant\_code

### SWC\_ID:

**Description:**Redundant statements may have no effect.

**Example:**

```
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 commented line references types/identifiers, but performs no action with them, so no

```
}
}
DASP : Unknown unknowns
Found: false
```

## 1.65 do\_while\_continue

### SWC\_ID:

**Description:**Prior to version 0.5.0, Solidity compiler handles continue inside do-while loop incorrectly: it ignores while condition.

**Example:**

The following loop is infinite:

```
do {
    continue;
} while(false);
```

```
}
}
DASP : Unknown Unknowns
Found: false
```

## 1.66 assert\_state\_change

### SWC\_ID:

**Description:**Incorrect use of assert(). See Solidity best practices.

### Example:

```
contract A {
    uint s\textunderscore a;

    function bad() public {
        assert((s\textunderscore a += 1) > 10);
    }
}
```

The assert in bad() increments the state variable s\textunderscore a while checking for .

```
}
```

```
}
```

**DASP** : Unknown Unknowns

**Found:** false