==
File: SWC-100.md
==

Title

Function Default Visibility

Relationships

- CWE-710: Improper Adherence to Coding Standards
- EthTrust Security Levels:
- **[Q] Code Linting**

Description

Functions that do not have a function visibility type specified are `public` by default. This can lead to a vulnerability if a developer forgot to set the visibility and a malicious user is able to make unauthorized or unintended state changes.

Remediation

Functions can be specified as being `external`, `public`, `internal` or `private`. It is recommended to make a conscious decision on which visibility type is appropriate for a function. This can dramatically reduce the attack surface of a contract system.

References

- Ethereum Smart Contract Best Practices - Visibility
- SigmaPrime - Visibility

Samples

visibility_not_set.sol

```solidity / * *@source: https://github.com/sigp/solidity-security-blog#visibility * @author: SigmaPrime * Modified by Gerhard Wagner* /

pragma solidity ^0.4.24;

contract HashForEther {

```
function withdrawWinnings() {
    // Winner if the last 8 hex characters of the address are 0.
```
```

````
    require(uint32(msg.sender) == 0);
    _sendWinnings();
 }

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

}
````

~~~

Comments

The function declarations in lines 11 and 17 do not set the visibility of the functions. At least for Solidity 0.4.24 (as specified in the `pragma` statement), this means they will default to being treated as `public`. This allows anyone to call the `_sendWinings()` function and take the money.

Instead, the fixed version below restricts the `_sendWinnings()` function visibility to `internal`, so it can only be activated by the `WithdrawWinnings()` function that enforces a check whether the sender actually met the presumed conditions to receive the money.

visibility_not_set_fixed.sol

```solidity / * *@source: https://github.com/sigp/solidity-security-blog#visibility * @author: SigmaPrime * Modified by Gerhard Wagner* /

pragma solidity ^0.4.24;

contract HashForEther {

~~~
function withdrawWinnings() public {
    // Winner if the last 8 hex characters of the address are 0.
    require(uint32(msg.sender) == 0);
    _sendWinnings();
 }

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

}
~~~

```

==
File: SWC-101.md
==
~~~

Title

Integer Overflow and Underflow

Relationships

- CWE-682: Incorrect Calculation
- EthTrust Security Levels:
- **[S] No Overflow/Underflow**
- **[M] Safe Overflow/Underflow**
- **[M] Documented Defensive Coding**

Description

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.

Remediation

It is recommended to use vetted safe math libraries for arithmetic operations consistently throughout the smart contract system.

References

- Ethereum Smart Contract Best Practices - Insecure Arithmetic

Samples

tokensalechallenge.sol

```solidity / * *@source: https://capturetheether.com/challenges/math/token-sale/ * @author: Steve Marx* /

pragma solidity ^0.4.21;

contract TokenSaleChallenge { mapping(address => uint256) public balanceOf; uint256 constant PRICE_PER_TOKEN = 1 ether;

```
function TokenSaleChallenge(address _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_PER_TOKEN);

    balanceOf[msg.sender] += numTokens;
}

function sell(uint256 numTokens) public {
    require(balanceOf[msg.sender] >= numTokens);

    balanceOf[msg.sender] -= numTokens;
    msg.sender.transfer(numTokens * PRICE_PER_TOKEN);
}
```

} ```

integer_overflow_mapping_sym_1.sol

```solidity //Single transaction overflow

pragma solidity ^0.4.11;

contract IntegerOverflowMappingSym1 { mapping(uint256 => uint256) map;

```
function init(uint256 k, uint256 v) public {
    map[k] -= v;
}
```

}

```

integer_overflow_mapping_sym_1_fixed.sol

```solidity //Single transaction overflow //Safe version

pragma solidity ^0.4.16;

contract IntegerOverflowMappingSym1 { mapping(uint256 => uint256) map;

```
function init(uint256 k, uint256 v) public {
    map[k] = sub(map[k], v);
}

//from SafeMath
function sub(uint256 a, uint256 b) internal pure returns (uint256) {
    require(b <= a);//SafeMath uses assert here
```
```

~~~
    return a - b;
}

}
~~~

```

## integer_overflow_minimal.sol

```solidity //Single transaction overflow //Post-transaction effect: overflow escapes to publicly-readable storage

pragma solidity ^0.4.19;

contract IntegerOverflowMinimal { uint public count = 1;

~~~
function run(uint256 input) public {
    count -= input;
}

}
~~~

```

integer_overflow_minimal_fixed.sol

```solidity //Single transaction overflow //Post-transaction effect: overflow escapes to publicly-readable storage //Safe version

pragma solidity ^0.4.19;

contract IntegerOverflowMinimal { uint public count = 1;

~~~
function run(uint256 input) public {
    count = sub(count,input);
}

//from SafeMath
function sub(uint256 a, uint256 b) internal pure returns (uint256) {
    require(b <= a);//SafeMath uses assert here
    return a - b;
}

}
~~~

```

integer_overflow_mul.sol

```solidity //Single transaction overflow //Post-transaction effect: overflow escapes to publicly-readable storage

pragma solidity ^0.4.19;

contract IntegerOverflowMul { uint public count = 2;

```
function run(uint256 input) public {
    count *= input;
}
```

}

\`\`\`

integer_overflow_mul_fixed.sol

\`\`\`solidity //Single transaction overflow //Post-transaction effect: overflow escapes to publicly-readable storage //Safe version

pragma solidity ^0.4.19;

contract IntegerOverflowMul { uint public count = 2;

```
function run(uint256 input) public {
    count = mul(count, input);
}

//from SafeMath
function mul(uint256 a, uint256 b) internal pure returns (uint256) {
  // Gas optimization: this is cheaper than requiring 'a' not being zero,
  // benefit is lost if 'b' is also tested.
  // See: https://github.com/OpenZeppelin/openzeppelin-solidity/pull/522
  if (a == 0) {
    return 0;
  }

  uint256 c = a * b;
  require(c / a == b);

  return c;
}
```

}

\`\`\`

integer_overflow_multitx_multifunc_feasible.sol

\`\`\`solidity / * *@source: https://github.com/ConsenSys/evm-analyzer-benchmark-suite* * *@author: Suhabe Bugrara* /

//Multi-transactional, multi-function //Arithmetic instruction reachable

pragma solidity ^0.4.23;

contract IntegerOverflowMultiTxMultiFuncFeasible { uint256 private initialized = 0; uint256 public count = 1;

```
function init() public {
    initialized = 1;
}

function run(uint256 input) {
    if (initialized == 0) {
        return;
    }

    count -= input;
}

}
```

\`\`\`

integer_overflow_multitx_multifunc_feasible_fixed.sol

\`\`\`solidity / * *@source: https://github.com/ConsenSys/evm-analyzer-benchmark-suite * @author: Suhabe Bugrara* /

//Multi-transactional, multi-function //Arithmetic instruction reachable (Safe)

pragma solidity ^0.4.23;

contract IntegerOverflowMultiTxMultiFuncFeasible { uint256 private initialized = 0; uint256 public count = 1;

```
function init() public {
    initialized = 1;
}

function run(uint256 input) {
    if (initialized == 0) {
        return;
    }

    count = sub(count, input);
}

//from SafeMath
function sub(uint256 a, uint256 b) internal pure returns (uint256) {
    require(b <= a);//SafeMath uses assert here
    return a - b;
}

}
```

\`\`\`

integer_overflow_multitx_onefunc_feasible.sol

```solidity / * *@source: https://github.com/ConsenSys/evm-analyzer-benchmark-suite * @author: Suhabe Bugrara* /

//Multi-transactional, single function //Arithmetic instruction reachable

pragma solidity ^0.4.23;

contract IntegerOverflowMultiTxOneFuncFeasible { uint256 private initialized = 0; uint256 public count = 1;

```
function run(uint256 input) public {
    if (initialized == 0) {
        initialized = 1;
        return;
    }

    count -= input;
}
```

}

```

integer_overflow_multitx_onefunc_feasible_fixed.sol

```solidity / * *@source: https://github.com/ConsenSys/evm-analyzer-benchmark-suite * @author: Suhabe Bugrara* /

//Multi-transactional, single function //Arithmetic instruction reachable (Safe)

pragma solidity ^0.4.23;

contract IntegerOverflowMultiTxOneFuncFeasible {

```
uint256 private initialized = 0;
uint256 public count = 1;

function run(uint256 input) public {
    if (initialized == 0) {
        initialized = 1;
        return;
    }

    count = sub(count, input);
}

//from SafeMath
function sub(uint256 a, uint256 b) internal pure returns (uint256) {
    require(b <= a);//SafeMath uses assert here
```
```

```
    return a - b;
}
```

}

integer_overflow_multitx_onefunc_infeasible.sol

```solidity / * *@source: https://github.com/ConsenSys/evm-analyzer-benchmark-suite * @author: Suhabe Bugrara* /

//Multi-transactional, single function //Overflow infeasible because arithmetic instruction not reachable

pragma solidity ^0.4.23;

contract IntegerOverflowMultiTxOneFuncInfeasible { uint256 private initialized = 0; uint256 public count = 1;

```
function run(uint256 input) public {
    if (initialized == 0) {
        return;
    }

    count -= input;
}
```

}

```

overflow_simple_add.sol

```solidity pragma solidity 0.4.24;

contract Overflow_Add { uint public balance = 1;

```
function add(uint256 deposit) public {
    balance += deposit;
}
```

}

```

overflow_simple_add_fixed.sol

```solidity pragma solidity ^0.4.24;

contract Overflow_Add { uint public balance = 1;
function add(uint256 deposit) public {
    balance = add(balance, deposit);
}

//from SafeMath
function add(uint256 a, uint256 b) internal pure returns (uint256) {
  uint256 c = a + b;
  require(c >= a);

  return c;
}

}
```

\`\`\`

BECToken.sol

\`\`\`solidity pragma solidity ^0.4.16;

/* * *@title SafeMath * @dev Math operations with safety checks that throw on error* / library SafeMath { function mul(uint256 a, uint256 b) internal constant returns (uint256) { uint256 c = a * b; require(a == 0 || c / a == b); return c; }

function div(uint256 a, uint256 b) internal constant returns (uint256) { // require(b > 0); // Solidity automatically throws when dividing by 0 uint256 c = a / b; // require(a == b * c + a % b); // There is no case in which this doesn't hold return c; }

function sub(uint256 a, uint256 b) internal constant returns (uint256) { require(b <= a); return a - b; }

function add(uint256 a, uint256 b) internal constant returns (uint256) { uint256 c = a + b; require(c >= a); return c; } }

/* * *@title ERC20Basic * @dev Simpler version of ERC20 interface * @dev see https://github.com/ethereum/EIPs/issues/179* / contract ERC20Basic { uint256 public totalSupply; function balanceOf(address who) public constant returns (uint256); function transfer(address to, uint256 value) public returns (bool); event Transfer(address indexed from, address indexed to, uint256 value); }

/* * *@title Basic token * @dev Basic version of StandardToken, with no allowances.* / contract BasicToken is ERC20Basic { using SafeMath for uint256;

mapping(address => uint256) balances;

/* * *@dev transfer token for a specified address * @param _to The address to transfer to. * @param _value The amount to be transferred.* / function transfer(address _to, uint256 _value) public returns (bool) { require(_to != address(0)); require(_value > 0 && _value <= balances[msg.sender]);

```
// SafeMath.sub will throw if there is not enough balance.
balances[msg.sender] = balances[msg.sender].sub(_value);
balances[_to] = balances[_to].add(_value);
Transfer(msg.sender, _to, _value);
return true;
```

}

/ * @dev Gets the balance of the specified address. * @param _owner The address to query the the balance of. * @return An uint256 representing the amount owned by the passed address.* / function balanceOf(address _owner) public constant returns (uint256 balance) { return balances[_owner]; } }

/ * @title ERC20 interface * @dev see https://github.com/ethereum/EIPs/issues/20* / contract ERC20 is ERC20Basic { function allowance(address owner, address spender) public constant returns (uint256); function transferFrom(address from, address to, uint256 value) public returns (bool); function approve(address spender, uint256 value) public returns (bool); event Approval(address indexed owner, address indexed spender, uint256 value); }

/ * @title Standard ERC20 token * * @dev Implementation of the basic standard token. * @dev https://github.com/ethereum/EIPs/issues/20 * @dev Based on code by FirstBlood: https://github.com/Firstbloodio/token/blob/master/smart_contract/FirstBloodToken.sol* / contract StandardToken is ERC20, BasicToken {

mapping (address => mapping (address => uint256)) internal allowed;

/ * @dev Transfer tokens from one address to another * @param _from address The address which you want to send tokens from * @param _to address The address which you want to transfer to * @param _value uint256 the amount of tokens to be transferred* / function transferFrom(address _from, address _to, uint256 _value) public returns (bool) { require(_to != address(0)); require(_value > 0 && _value <= balances[_from]); require(_value <= allowed[_from][msg.sender]);

```
balances[_from] = balances[_from].sub(_value);
balances[_to] = balances[_to].add(_value);
allowed[_from][msg.sender] = allowed[_from][msg.sender].sub(_value);
Transfer(_from, _to, _value);
return true;
```

}

/ * @dev Approve the passed address to spend the specified amount of tokens on behalf of msg.sender. * * Beware that changing an allowance with this method brings the risk that someone may use both the old * and the new allowance by unfortunate transaction ordering. One possible solution to mitigate this * race condition is to first reduce the spender's allowance to 0 and set the desired value afterwards: * https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729 * @param _spender The address which will spend the funds. * @param _value The amount of tokens to be spent. /*

function approve(address _spender, uint256 _value) public returns (bool) { allowed[msg.sender][_spender] = _value; Approval(msg.sender, _spender, _value); return true; }

/* * *@dev Function to check the amount of tokens that an owner allowed to a spender. * @param _owner address The address which owns the funds. * @param _spender address The address which will spend the funds. * @return A uint256 specifying the amount of tokens still available for the spender.* / function allowance(address _owner, address _spender) public constant returns (uint256 remaining) { return allowed[_owner][_spender]; } }

/* * *@title Ownable * @dev The Ownable contract has an owner address, and provides basic authorization control * functions, this simplifies the implementation of* "*user permissions*"*.* / contract Ownable { address public owner;

event OwnershipTransferred(address indexed previousOwner, address indexed newOwner);

/* * *@dev The Ownable constructor sets the original* `owner` *of the contract to the sender * account.* / function Ownable() { owner = msg.sender; }

/* * *@dev Throws if called by any account other than the owner.* / modifier onlyOwner() { require(msg.sender == owner); _; }

/* * *@dev Allows the current owner to transfer control of the contract to a newOwner. * @param newOwner The address to transfer ownership to.* / function transferOwnership(address newOwner) onlyOwner public { require(newOwner != address(0)); OwnershipTransferred(owner, newOwner); owner = newOwner; }

}

/* * *@title Pausable * @dev Base contract which allows children to implement an emergency stop mechanism.* / contract Pausable is Ownable { event Pause(); event Unpause();

bool public paused = false;

/* * *@dev Modifier to make a function callable only when the contract is not paused.* / modifier whenNotPaused() { require(!paused); _; }

/* * *@dev Modifier to make a function callable only when the contract is paused.* / modifier whenPaused() { require(paused); _; }

/* * *@dev called by the owner to pause, triggers stopped state* / function pause() onlyOwner whenNotPaused public { paused = true; Pause(); }

/* * *@dev called by the owner to unpause, returns to normal state* / function unpause() onlyOwner whenPaused public { paused = false; Unpause(); } }

/ * @title Pausable token * * @dev StandardToken modified with pausable transfers. /

contract PausableToken is StandardToken, Pausable {

function transfer(address _to, uint256 _value) public whenNotPaused returns (bool) { return super.transfer(_to, _value); }

function transferFrom(address _from, address _to, uint256 _value) public whenNotPaused returns (bool) { return super.transferFrom(_from, _to, _value); }

function approve(address _spender, uint256 _value) public whenNotPaused returns (bool) { return super.approve(_spender, _value); }

function batchTransfer(address[] _receivers, uint256 _value) public whenNotPaused returns (bool) { uint cnt = _receivers.length; uint256 amount = uint256(cnt) * _value; require(cnt > 0 && cnt <= 20); require(_value > 0 && balances[msg.sender] >= amount);

```
balances[msg.sender] = balances[msg.sender].sub(amount);
for (uint i = 0; i < cnt; i++) {
    balances[_receivers[i]] = balances[_receivers[i]].add(_value);
    Transfer(msg.sender, _receivers[i], _value);
}
return true;
```

} }

/ * @title Bec Token * * @dev Implementation of Bec Token based on the basic standard token. */ contract BecToken is PausableToken { / * Public variables of the token * The following variables are OPTIONAL vanities. One does not have to include them. * They allow one to customise the token contract & in no way influences the core functionality. * Some wallets/interfaces might not even bother to look at this information. */ string public name = "BeautyChain"; string public symbol = "BEC"; string public version = '1.0.0'; uint8 public decimals = 18;

```
/**
 * @dev Function to check the amount of tokens that an owner allowed to a
 */
function BecToken() {
  totalSupply = 7000000000 * (10**(uint256(decimals)));
  balances[msg.sender] = totalSupply;    // Give the creator all initial t
}

function () {
    //if ether is sent to this address, send it back.
    revert();
}
```

} ```

==

File: SWC-102.md

==

Title

Outdated Compiler Version

Relationships

- CWE-937: Using Components with Known Vulnerabilities
- EEA EthTrust Security Levels:
- **Level [S]** Improved Compilers
- **Level [S]** Compiler Security Bugs
- **Level [M]** Compiler Bugs and Overriding Requirements
- **Recommended Practice** Use the Latest Compiler

Description

Using an outdated compiler version can be problematic especially if there are publicly disclosed bugs and issues that affect the current compiler version.

Remediation

It is recommended to use a recent version of the Solidity compiler.

References

- Solidity Release Notes
- Etherscan Solidity Bug Info

Samples

version_0_4_13.sol

```solidity pragma solidity 0.4.13;

contract OutdatedCompilerVersion { uint public x = 1; }

```

Comments

As of August 2023 the current version of the compiler is 0.8.21. There are several dozen compiler bugs that have been fixed between that and version 0.4.13, each of which can lead to data being corrupted, contracts not functioning as expected, or unexpected vulnerabilities in contracts. There have also been significant improvements in compiler capabilities to protect against errors.

See also the sections **Level [S]** Compiler Security Bugs, **Level [S]** Improved Compilers, and **Level [M]** Compiler Bugs and Overriding Requirements

==
File: SWC-103.md
==

Title

Floating Pragma

Relationships

- CWE-664: Improper Control of a Resource Through its Lifetime
- EEA EthTrust Security Levels:
- **Level [S]** Improved Compilers
- **Level [S]** Compiler Security Bugs
- **Level [M]** Compiler Bugs and Overriding Requirements
- **Recommended Practice** Use the Latest Compiler

Description

Contracts should be deployed with the same compiler version and flags that they have been tested with thoroughly. Locking the pragma helps to ensure that contracts do not accidentally get deployed using, for example, an outdated compiler version that might introduce bugs that affect the contract system negatively.

Remediation

Lock the pragma version and also consider known bugs (https://github.com/ethereum/solidity/releases) for the compiler version that is chosen.

Pragma statements can be allowed to float when a contract is intended for consumption by other developers, as in the case with contracts in a library or EthPM package. Otherwise, the developer would need to manually update the pragma in order to compile locally.

References

- Ethereum Smart Contract Best Practices - Locking Pragmas

Samples

floating_pragma.sol

```solidity pragma solidity ^0.4.0;
```

contract PragmaNotLocked { uint public x = 1; }

floating_pragma_fixed.sol

```solidity pragma solidity 0.4.25;

contract PragmaFixed { uint public x = 1; }

```

no_pragma.sol

```solidity

contract PragmaNotLocked { uint public x = 1; }

```

semver_floating_pragma.sol

```solidity pragma solidity >=0.4.0 < 0.6.0; pragma solidity >=0.4.0<0.6.0; pragma solidity >=0.4.14 <0.6.0; pragma solidity >0.4.13 <0.6.0; pragma solidity 0.4.24 - 0.5.2; pragma solidity >=0.4.24 <=0.5.3 ~0.4.20; pragma solidity <0.4.26; pragma solidity ~0.4.20; pragma solidity ^0.4.14; pragma solidity 0.4.*; pragma solidity 0.*; pragma solidity *; pragma solidity 0.4; pragma solidity 0;

contract SemVerFloatingPragma { }

```

semver_floating_pragma_fixed.sol

```solidity pragma solidity 0.4.25; // or pragma solidity =0.4.25;

contract SemVerFloatingPragmaFixed { }

```

==
File: SWC-104.md
==

Title

Unchecked Call Return Value

Relationships

- CWE-252: Unchecked Return Value
- EthTrust Security Levels:
- **[M] Handle External Call Returns**

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.

Remediation

If you choose to use low-level call methods, make sure to handle the possibility that the call will fail by checking the return value.

References

- Ethereum Smart Contract Best Practices - Handle errors in external calls

Samples

unchecked_return_value.sol

```solidity pragma solidity 0.4.25;

contract ReturnValue {

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

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

```

==
File: SWC-105.md
==

Title

Unprotected Ether Withdrawal

Relationships

- CWE-284: Improper Access Control
- EthTrust Security Levels:
- **[M] Protect Self-destruction**
- **[Q] Enforce Least Privilege**

Description

Due to missing or insufficient access controls, malicious parties can withdraw some or all Ether from the contract account.

This bug is sometimes caused by unintentionally exposing initialization functions. By wrongly naming a function intended to be a constructor, the constructor code ends up in the runtime byte code and can be called by anyone to re-initialize the contract.

Remediation

Implement controls so withdrawals can only be triggered by authorized parties or according to the specs of the smart contract system.

References

- Rubixi smart contract

Samples

tokensalechallenge.sol

``` solidity / * *@source: https://capturetheether.com/challenges/math/token-sale/ * @author: Steve Marx* /

pragma solidity ^0.4.21;

contract TokenSaleChallenge { mapping(address => uint256) public balanceOf; uint256 constant PRICE_PER_TOKEN = 1 ether;

```
function TokenSaleChallenge(address _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_PER_TOKEN);
```
```

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

function sell(uint256 numTokens) public {
    require(balanceOf[msg.sender] >= numTokens);

    balanceOf[msg.sender] -= numTokens;
    msg.sender.transfer(numTokens * PRICE_PER_TOKEN);
}
```

} ```

rubixi.sol

```solidity pragma solidity ^0.4.22;

contract Rubixi {

```
    //Declare variables for storage critical to contract
    uint private balance = 0;
    uint private collectedFees = 0;
    uint private feePercent = 10;
    uint private pyramidMultiplier = 300;
    uint private payoutOrder = 0;

    address private creator;

    //Sets creator
    function DynamicPyramid() {
            creator = msg.sender;
    }

    modifier onlyowner {
            if (msg.sender == creator) _;
    }

    struct Participant {
            address etherAddress;
            uint payout;
    }

    Participant[] private participants;

    //Fallback function
    function() {
            init();
    }

    //init function run on fallback
    function init() private {
            //Ensures only tx with value of 1 ether or greater are process
            if (msg.value < 1 ether) {
```
```

```
                        collectedFees += msg.value;
                        return;
                }

                uint _fee = feePercent;
                //50% fee rebate on any ether value of 50 or greater
                if (msg.value >= 50 ether) _fee /= 2;

                addPayout(_fee);
        }

        //Function called for valid tx to the contract
        function addPayout(uint _fee) private {
                //Adds new address to participant array
                participants.push(Participant(msg.sender, (msg.value * pyramid

                //These statements ensure a quicker payout system to later pyr
                if (participants.length == 10) pyramidMultiplier = 200;
                else if (participants.length == 25) pyramidMultiplier = 150;

                // collect fees and update contract balance
                balance += (msg.value * (100 - _fee)) / 100;
                collectedFees += (msg.value * _fee) / 100;

                //Pays earlier participiants if balance sufficient
                while (balance > participants[payoutOrder].payout) {
                        uint payoutToSend = participants[payoutOrder].payout;
                        participants[payoutOrder].etherAddress.send(payoutToSe

                        balance -= participants[payoutOrder].payout;
                        payoutOrder += 1;
                }
        }

        //Fee functions for creator
        function collectAllFees() onlyowner {
                if (collectedFees == 0) throw;

                creator.send(collectedFees);
                collectedFees = 0;
        }

        function collectFeesInEther(uint _amt) onlyowner {
                _amt *= 1 ether;
                if (_amt > collectedFees) collectAllFees();

                if (collectedFees == 0) throw;

                creator.send(_amt);
                collectedFees -= _amt;
        }
        function collectPercentOfFees(uint _pcent) onlyowner {
                if (collectedFees == 0 || _pcent > 100) throw;

                uint feesToCollect = collectedFees / 100 * _pcent;
                creator.send(feesToCollect);
                collectedFees -= feesToCollect;
        }

        //Functions for changing variables related to the contract
        function changeOwner(address _owner) onlyowner {
                creator = _owner;
        }

        function changeMultiplier(uint _mult) onlyowner {
                if (_mult > 300 || _mult < 120) throw;

                pyramidMultiplier = _mult;
        }

        function changeFeePercentage(uint _fee) onlyowner {
                if (_fee > 10) throw;

                feePercent = _fee;
        }

        //Functions to provide information to end-user using JSON interface or
        function currentMultiplier() constant returns(uint multiplier, string
                multiplier = pyramidMultiplier;
                info = 'This multiplier applies to you as soon as transaction
        }

        function currentFeePercentage() constant returns(uint fee, string info
                fee = feePercent;
                info = 'Shown in % form. Fee is halved(50%) for amounts equal
        }

        function currentPyramidBalanceApproximately() constant returns(uint py
                pyramidBalance = balance / 1 ether;
                info = 'All balance values are measured in Ethers, note that d
        }

        function nextPayoutWhenPyramidBalanceTotalsApproximately() constant re
                balancePayout = participants[payoutOrder].payout / 1 ether;
        }

        function feesSeperateFromBalanceApproximately() constant returns(uint
                fees = collectedFees / 1 ether;
        }

        function totalParticipants() constant returns(uint count) {
                count = participants.length;
        }
```

```
    function numberOfParticipantsWaitingForPayout() constant returns(uint
            count = participants.length - payoutOrder;
    }

    function participantDetails(uint orderInPyramid) constant returns(addr
            if (orderInPyramid <= participants.length) {
                    Address = participants[orderInPyramid].etherAddress;
                    Payout = participants[orderInPyramid].payout / 1 ether
            }
    }
```

}

multiowned_not_vulnerable.sol

```solidity pragma solidity ^0.4.23;

/* * *@title MultiOwnable* / contract MultiOwnable { address public root; mapping (address => address) public owners; // owner => parent of owner

/* * *@dev The Ownable constructor sets the original `owner` of the contract to the sender * account.* / constructor() public { root = msg.sender; owners[root] = root; }

/* * *@dev Throws if called by any account other than the owner.* / modifier onlyOwner() { require(owners[msg.sender] != 0); _; }

/* * *@dev Adding new owners * Note that the "onlyOwner" modifier is used here.* / function newOwner(address _owner) onlyOwner external returns (bool) { require(_owner != 0); owners[_owner] = msg.sender; return true; }

/* * *@dev Deleting owners* / function deleteOwner(address _owner) onlyOwner external returns (bool) { require(owners[_owner] == msg.sender || (owners[_owner] != 0 && msg.sender == root)); owners[_owner] = 0; return true; } }

contract TestContract is MultiOwnable {

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

function() payable { }

}

```

multiowned_vulnerable.sol

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

/* * *@title MultiOwnable* / contract MultiOwnable { address public root; mapping (address => address) public owners; // owner => parent of owner

/* * *@dev The Ownable constructor sets the original owner of the contract to the sender* * *account.* / constructor() public { root = msg.sender; owners[root] = root; }

/* * *@dev Throws if called by any account other than the owner.* / modifier onlyOwner() { require(owners[msg.sender] != 0); _; }

/* * *@dev Adding new owners* * *Note that the "onlyOwner" modifier is missing here.* / function newOwner(address _owner) external returns (bool) { require(_owner != 0); owners[_owner] = msg.sender; return true; }

/* * *@dev Deleting owners* / function deleteOwner(address _owner) onlyOwner external returns (bool) { require(owners[_owner] == msg.sender || (owners[_owner] != 0 && msg.sender == root)); owners[_owner] = 0; return true; } }

contract TestContract is MultiOwnable {

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

function() payable { }

} ```

simple_ether_drain.sol

```solidity pragma solidity ^0.4.22;

contract SimpleEtherDrain {

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

function () public payable { }

}

```

wallet_01_ok.sol

```solidity pragma solidity ^0.4.24;

/ *User can add pay in and withdraw Ether. Nobody can withdraw more Ether than they paid in.* /

contract Wallet { address creator;

```
mapping(address => uint256) balances;

constructor() public {
    creator = msg.sender;
```
```

~~~
}

function deposit() public payable {
    assert(balances[msg.sender] + msg.value > balances[msg.sender]);
    balances[msg.sender] += msg.value;
}

function withdraw(uint256 amount) public {
    require(amount <= balances[msg.sender]);
    msg.sender.transfer(amount);
    balances[msg.sender] -= amount;
}

function refund() public {
    msg.sender.transfer(balances[msg.sender]);
    balances[msg.sender] = 0;
}

// In an emergency the owner can migrate  allfunds to a different address.

function migrateTo(address to) public {
    require(creator == msg.sender);
    to.transfer(this.balance);
}

}
~~~

```

## wallet_02_refund_nosub.sol

```solidity pragma solidity ^0.4.24;

*/ User can add pay in and withdraw Ether. Unfortunately the developer forgot set the user's balance to 0 when refund() is called. An attacker can pay in a small amount of Ether and call refund() repeatedly to empty the contract. /*

contract Wallet { address creator;

~~~
mapping(address => uint256) balances;

constructor() public {
    creator = msg.sender;
}

function deposit() public payable {
    assert(balances[msg.sender] + msg.value > balances[msg.sender]);
    balances[msg.sender] += msg.value;
}

function withdraw(uint256 amount) public {
~~~

```
    require(amount <= balances[msg.sender]);
    msg.sender.transfer(amount);
    balances[msg.sender] -= amount;
}

function refund() public {
    msg.sender.transfer(balances[msg.sender]);
}

// In an emergency the owner can migrate allfunds to a different address.

function migrateTo(address to) public {
    require(creator == msg.sender);
    to.transfer(this.balance);
}

}
```

\`\`\`

wallet_03_wrong_constructor.sol

\`\`\`solidity pragma solidity ^0.4.24;

/ *User can add pay in and withdraw Ether. The constructor is wrongly named, so anyone can become 'creator' and withdraw all funds.* /

contract Wallet { address creator;

```
mapping(address => uint256) balances;

function initWallet() public {
    creator = msg.sender;
}

function deposit() public payable {
    assert(balances[msg.sender] + msg.value > balances[msg.sender]);
    balances[msg.sender] += msg.value;
}

function withdraw(uint256 amount) public {
    require(amount <= balances[msg.sender]);
    msg.sender.transfer(amount);
    balances[msg.sender] -= amount;
}

// In an emergency the owner can migrate  allfunds to a different address.

function migrateTo(address to) public {
    require(creator == msg.sender);
    to.transfer(this.balance);
}
```

}

```

## wallet_04_confused_sign.sol

```solidity pragma solidity ^0.4.24;

*/ User can add pay in and withdraw Ether. Unfortunately, the developer was drunk and used the wrong comparison operator in "withdraw()" Anybody can withdraw arbitrary amounts of Ether :() /*

contract Wallet { address creator;

~~~
mapping(address => uint256) balances;

constructor() public {
    creator = msg.sender;
}

function deposit() public payable {
    assert(balances[msg.sender] + msg.value > balances[msg.sender]);
    balances[msg.sender] += msg.value;
}

function withdraw(uint256 amount) public {
    require(amount >= balances[msg.sender]);
    msg.sender.transfer(amount);
    balances[msg.sender] -= amount;
}

// In an emergency the owner can migrate  allfunds to a different address.

function migrateTo(address to) public {
    require(creator == msg.sender);
    to.transfer(this.balance);
}
~~~

}

```

==
File: SWC-106.md
==

Title

Unprotected SELFDESTRUCT Instruction
```

Relationships

- CWE-284: Improper Access Control
- EthTrust Security Levels:
- **[M] Protect Self-destruction**
- **[Q] Enforce Least Privilege**

Description

Due to missing or insufficient access controls, malicious parties can self-destruct the contract.

Remediation

Consider removing the self-destruct functionality unless it is absolutely required. If there is a valid use-case, it is recommended to implement a multisig scheme so that multiple parties must approve the self-destruct action.

References

- Parity "I accidentally killed it" bug

Samples

WalletLibrary.sol

```solidity //sol Wallet // Multi-sig, daily-limited account proxy/wallet. // @authors: // Gav Wood g@ethdev.com // inheritable "property" contract that enables methods to be protected by requiring the acquiescence of either a // single, or, crucially, each of a number of, designated owners. // usage: // use modifiers onlyowner (just own owned) or onlymanyowners(hash), whereby the same hash must be provided by // some number (specified in constructor) of the set of owners (specified in the constructor, modifiable) before the // interior is executed.

pragma solidity ^0.4.9;

contract WalletEvents { // EVENTS

// this contract only has six types of events: it can accept a confirmation, in which case // we record owner and operation (hash) alongside it. event Confirmation(address owner, bytes32 operation); event Revoke(address owner, bytes32 operation);

// some others are in the case of an owner changing. event OwnerChanged(address oldOwner, address newOwner); event OwnerAdded(address newOwner); event OwnerRemoved(address oldOwner);
```

// the last one is emitted if the required signatures change event RequirementChanged(uint newRequirement);

// Funds has arrived into the wallet (record how much). event Deposit(address _from, uint value); // Single transaction going out of the wallet (record who signed for it, how much, and to whom it's going). event SingleTransact(address owner, uint value, address to, bytes data, address created); // Multi-sig transaction going out of the wallet (record who signed for it last, the operation hash, how much, and to whom it's going). event MultiTransact(address owner, bytes32 operation, uint value, address to, bytes data, address created); // Confirmation still needed for a transaction. event ConfirmationNeeded(bytes32 operation, address initiator, uint value, address to, bytes data); }

contract WalletAbi { // Revokes a prior confirmation of the given operation function revoke(bytes32 _operation) external;

// Replaces an owner `_from` with another `_to`. function changeOwner(address _from, address _to) external;

function addOwner(address _owner) external;

function removeOwner(address _owner) external;

function changeRequirement(uint _newRequired) external;

function isOwner(address _addr) constant returns (bool);

function hasConfirmed(bytes32 _operation, address _owner) external constant returns (bool);

// (re)sets the daily limit. needs many of the owners to confirm. doesn't alter the amount already spent today. function setDailyLimit(uint _newLimit) external;

function execute(address _to, uint _value, bytes _data) external returns (bytes32 o_hash); function confirm(bytes32 _h) returns (bool o_success); }

contract WalletLibrary is WalletEvents { // TYPES

// struct for the status of a pending operation. struct PendingState { uint yetNeeded; uint ownersDone; uint index; }

// Transaction structure to remember details of transaction lest it need be saved for a later call. struct Transaction { address to; uint value; bytes data; }

// MODIFIERS

// simple single-sig function modifier. modifier onlyowner { if (isOwner(msg.sender)) *; } // multi-sig function modifier: the operation must have an intrinsic hash in order // that later attempts can be realised as the same underlying operation and // thus count as confirmations. modifier onlymanyowners(bytes32 _operation) { if (confirmAndCheck(_operation))* ; }

// METHODS

// gets called when no other function matches function() payable { // just being sent some cash? if (msg.value > 0) Deposit(msg.sender, msg.value); }

// constructor is given number of sigs required to do protected "onlymanyowners" transactions // as well as the selection of addresses capable of confirming them. function initMultiowned(address[] _owners, uint _required) only_uninitialized { 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; }

// Revokes a prior confirmation of the given operation function revoke(bytes32 _operation) external { uint ownerIndex = m_ownerIndex[uint(msg.sender)]; // make sure they're an owner if (ownerIndex == 0) return; uint ownerIndexBit = 2**ownerIndex; var pending = m_pending[_operation]; if (pending.ownersDone & ownerIndexBit > 0) { pending.yetNeeded++; pending.ownersDone -= ownerIndexBit; Revoke(msg.sender, _operation); } }

// Replaces an owner `_from` with another `_to`. function changeOwner(address _from, address _to) onlymanyowners(sha3(msg.data)) external { if (isOwner(_to)) return; uint ownerIndex = m_ownerIndex[uint(_from)]; if (ownerIndex == 0) return;

```
clearPending();
m_owners[ownerIndex] = uint(_to);
m_ownerIndex[uint(_from)] = 0;
m_ownerIndex[uint(_to)] = ownerIndex;
OwnerChanged(_from, _to);
```

}

function addOwner(address _owner) onlymanyowners(sha3(msg.data)) external { if (isOwner(_owner)) return;

```
clearPending();
if (m_numOwners >= c_maxOwners)
  reorganizeOwners();
if (m_numOwners >= c_maxOwners)
  return;
m_numOwners++;
m_owners[m_numOwners] = uint(_owner);
m_ownerIndex[uint(_owner)] = m_numOwners;
OwnerAdded(_owner);
```

}

function removeOwner(address _owner) onlymanyowners(sha3(msg.data)) external { uint ownerIndex = m_ownerIndex[uint(_owner)]; if (ownerIndex == 0) return; if (m_required > m_numOwners - 1) return;

```
m_owners[ownerIndex] = 0;
m_ownerIndex[uint(_owner)] = 0;
clearPending();
reorganizeOwners(); //make sure m_numOwner is equal to the number of owner
OwnerRemoved(_owner);

}
```

function changeRequirement(uint _newRequired) onlymanyowners(sha3(msg.data)) external { if (_newRequired > m_numOwners) return; m_required = _newRequired; clearPending(); RequirementChanged(_newRequired); }

// Gets an owner by 0-indexed position (using numOwners as the count) function getOwner(uint ownerIndex) external constant returns (address) { return address(m_owners[ownerIndex + 1]); }

function isOwner(address _addr) constant returns (bool) { return m_ownerIndex[uint(_addr)] > 0; }

function hasConfirmed(bytes32 _operation, address _owner) external constant returns (bool) { var pending = m_pending[_operation]; uint ownerIndex = m_ownerIndex[uint(_owner)];

```
// make sure they're an owner
if (ownerIndex == 0) return false;

// determine the bit to set for this owner.
uint ownerIndexBit = 2**ownerIndex;
return !(pending.ownersDone & ownerIndexBit == 0);

}
```

// constructor - stores initial daily limit and records the present day's index. function initDaylimit(uint _limit) only_uninitialized { m_dailyLimit = _limit; m_lastDay = today(); } // (re)sets the daily limit. needs many of the owners to confirm. doesn't alter the amount already spent today. function setDailyLimit(uint _newLimit) onlymanyowners(sha3(msg.data)) external { m_dailyLimit = _newLimit; } // resets the amount already spent today. needs many of the owners to confirm. function resetSpentToday() onlymanyowners(sha3(msg.data)) external { m_spentToday = 0; }

// throw unless the contract is not yet initialized. modifier only_uninitialized { if (m_numOwners > 0) throw; _; }

// constructor - just pass on the owner array to the multiowned and // the limit to daylimit function initWallet(address[] _owners, uint _required, uint _daylimit) only_uninitialized { initDaylimit(_daylimit); initMultiowned(_owners, _required); }

// kills the contract sending everything to `_to`. function kill(address _to) onlymanyowners(sha3(msg.data)) external { suicide(_to); }

// Outside-visible transact entry point. Executes transaction immediately if below daily spend limit. // If not, goes into multisig process. We provide a hash on return to allow the sender to provide // shortcuts for the other confirmations (allowing them to avoid replicating the _to, _value // and _data arguments). They still get the option of using them if they want, anyways. function execute(address _to, uint _value, bytes _data) external onlyowner returns (bytes32 o_hash) { // first, take the opportunity to check that we're under the daily limit. if ((_data.length == 0 && underLimit(_value)) || m_required == 1) { // yes - just execute the call. address created; if (_to == 0) { created = create(_value, _data); } else { if (!_to.call.value(_value)(_data)) throw; } SingleTransact(msg.sender, _value, _to, _data, created); } else { // determine our operation hash. o_hash = sha3(msg.data, block.number); // store if it's new if (m_txs[o_hash].to == 0 && m_txs[o_hash].value == 0 && m_txs[o_hash].data.length == 0) { m_txs[o_hash].to = _to; m_txs[o_hash].value = _value; m_txs[o_hash].data = _data; } if (!confirm(o_hash)) { ConfirmationNeeded(o_hash, msg.sender, _value, _to, _data); } } }

function create(uint _value, bytes _code) internal returns (address o_addr) { / *assembly { o_addr := create(_value, add(_code, 0x20), mload(_code)) jumpi(invalidJumpLabel, iszero(extcodesize(o_addr))) }* / }

// confirm a transaction through just the hash. we use the previous transactions map, m_txs, in order // to determine the body of the transaction from the hash provided. function confirm(bytes32 _h) onlymanyowners(_h) returns (bool o_success) { if (m_txs[_h].to != 0 || m_txs[_h].value != 0 || m_txs[_h].data.length != 0) { address created; if (m_txs[_h].to == 0) { created = create(m_txs[_h].value, m_txs[_h].data); } else { if (!m_txs[_h].to.call.value(m_txs[_h].value)(m_txs[_h].data)) throw; }

```
  MultiTransact(msg.sender, _h, m_txs[_h].value, m_txs[_h].to, m_txs[_h].d
  delete m_txs[_h];
  return true;
}
```

}

// INTERNAL METHODS

function confirmAndCheck(bytes32 _operation) internal returns (bool) { // determine what index the present sender is: uint ownerIndex = m_ownerIndex[uint(msg.sender)]; // make sure they're an owner if (ownerIndex == 0) return;

```
var pending = m_pending[_operation];
// if we're not yet working on this operation, switch over and reset the c
if (pending.yetNeeded == 0) {
  // reset count of confirmations needed.
  pending.yetNeeded = m_required;
  // reset which owners have confirmed (none) - set our bitmap to 0.
  pending.ownersDone = 0;
  pending.index = m_pendingIndex.length++;
  m_pendingIndex[pending.index] = _operation;
}
// determine the bit to set for this owner.
uint ownerIndexBit = 2**ownerIndex;
// make sure we (the message sender) haven't confirmed this operation prev
if (pending.ownersDone & ownerIndexBit == 0) {
  Confirmation(msg.sender, _operation);
  // ok - check if count is enough to go ahead.
  if (pending.yetNeeded <= 1) {
    // enough confirmations: reset and run interior.
    delete m_pendingIndex[m_pending[_operation].index];
    delete m_pending[_operation];
    return true;
  }
  else
  {
    // not enough: record that this owner in particular confirmed.
    pending.yetNeeded--;
    pending.ownersDone |= ownerIndexBit;
  }
}

}
```

function reorganizeOwners() private { uint free = 1; while (free < m_numOwners) { while (free < m_numOwners && m_owners[free] != 0) free++; while (m_numOwners > 1 && m_owners[m_numOwners] == 0) m_numOwners--; if (free < m_numOwners && m_owners[m_numOwners] != 0 && m_owners[free] == 0) { m_owners[free] = m_owners[m_numOwners]; m_ownerIndex[m_owners[free]] = free; m_owners[m_numOwners] = 0; } } }

// checks to see if there is at least `_value` left from the daily limit today. if there is, subtracts it and // returns true. otherwise just returns false. function underLimit(uint _value) internal onlyowner returns (bool) { // reset the spend limit if we're on a different day to last time. if (today() > m_lastDay) { m_spentToday = 0; m_lastDay = today(); } // check to see if there's enough left - if so, subtract and return true. // overflow protection // dailyLimit check if (m_spentToday + _value >= m_spentToday && m_spentToday + _value <= m_dailyLimit) { m_spentToday += _value; return true; } return false; }

// determines today's index. function today() private constant returns (uint) { return now / 1 days; }

function clearPending() internal { uint length = m_pendingIndex.length;

```
for (uint i = 0; i < length; ++i) {
  delete m_txs[m_pendingIndex[i]];

  if (m_pendingIndex[i] != 0)
    delete m_pending[m_pendingIndex[i]];
}

delete m_pendingIndex;
```

}

// FIELDS address constant _walletLibrary = 0xcafecafecafecafecafecafecafecafecafecafe;

// the number of owners that must confirm the same operation before it is run. uint public m_required; // pointer used to find a free slot in m_owners uint public m_numOwners;

uint public m_dailyLimit; uint public m_spentToday; uint public m_lastDay;

// list of owners uint[256] m_owners;

uint constant c_maxOwners = 250; // index on the list of owners to allow reverse lookup mapping(uint => uint) m_ownerIndex; // the ongoing operations. mapping(bytes32 => PendingState) m_pending; bytes32[] m_pendingIndex;

// pending transactions we have at present. mapping (bytes32 => Transaction) m_txs; }

simple_suicide.sol

```solidity pragma solidity ^0.4.22;

contract SimpleSuicide {

function sudicideAnyone() { selfdestruct(msg.sender); }

}

```

suicide_multitx_feasible.sol

```solidity pragma solidity ^0.4.23;

contract SuicideMultiTxFeasible { uint256 private initialized = 0; uint256 public count = 1;

~~~
function init() public {
    initialized = 1;
}

function run(uint256 input) {
    if (initialized == 0) {
        return;
    }

    selfdestruct(msg.sender);
}
~~~
```

}

suicide_multitx_infeasible.sol

```solidity pragma solidity ^0.4.23;

contract SuicideMultiTxFeasible { uint256 private initialized = 0; uint256 public count = 1;

```
function init() public {
    initialized = 1;
}

function run(uint256 input) {
    if (initialized != 2) {
        return;
    }

    selfdestruct(msg.sender);
}
```

}

```

```
============================================================
```
File: SWC-107.md
```
============================================================
```

Title

Reentrancy

Relationships

CWE-841: Improper Enforcement of Behavioral Workflow

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.

Remediation

The best practices to avoid Reentrancy weaknesses are:

- Make sure all internal state changes are performed before the call is executed. This is known as the Checks-Effects-Interactions pattern
- Use a reentrancy lock (ie. OpenZeppelin's ReentrancyGuard).

References

- Ethereum Smart Contract Best Practices - Reentrancy

Samples

modifier_reentrancy.sol

```solidity pragma solidity ^0.5.0;

contract ModifierEntrancy {

mapping (address => uint) public tokenBalance; string constant name = "Nu Token"; Bank bank;

constructor() public{ bank = new Bank(); }

//If a contract has a zero balance and supports the token give them some token function airDrop() hasNoBalance supportsToken public{ tokenBalance[msg.sender] += 20; }

//Checks that the contract responds the way we want modifier supportsToken() { require(keccak256(abi.encodePacked("Nu Token")) == bank.supportsToken()); _; }

//Checks that the caller has a zero balance modifier hasNoBalance { require(tokenBalance[msg.sender] == 0); _; } }

contract Bank{

```
function supportsToken() external returns(bytes32) {
    return keccak256(abi.encodePacked("Nu Token"));
}
```

}

```

modifier_reentrancy_fixed.sol

```solidity pragma solidity ^0.5.0;
```

contract ModifierEntrancy { mapping (address => uint) public tokenBalance; string constant name = "Nu Token"; Bank bank; constructor() public{ bank = new Bank(); }

//If a contract has a zero balance and supports the token give them some token function airDrop() supportsToken hasNoBalance public{ // In the fixed version supportsToken comes before hasNoBalance tokenBalance[msg.sender] += 20; }

//Checks that the contract responds the way we want modifier supportsToken() { require(keccak256(abi.encodePacked("Nu Token")) == bank.supportsToken()); *; } //Checks that the caller has a zero balance modifier hasNoBalance { require(tokenBalance[msg.sender] == 0);* ; } }

contract Bank{

```
function supportsToken() external returns(bytes32){
    return(keccak256(abi.encodePacked("Nu Token")));
}
```

}

simple_dao.sol

```solidity /* *@source: http://blockchain.unica.it/projects/ethereum-survey/attacks.html#simpledao * @author: Atzei N., Bartoletti M., Cimoli T * Modified by Josselin Feist* / pragma solidity 0.4.24;

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

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

function withdraw(uint amount) public{ if (credit[msg.sender]>= amount) { require(msg.sender.call.value(amount)()); credit[msg.sender]-=amount; } }

function queryCredit(address to) view public returns(uint){ return credit[to]; } }

```

simple_dao_fixed.sol

```solidity /* *@source: http://blockchain.unica.it/projects/ethereum-survey/attacks.html#simpledao * @author: Atzei N., Bartoletti M., Cimoli T * Modified by Bernhard Mueller, Josselin Feist* / pragma solidity 0.4.24;

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

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

function withdraw(uint amount) public { if (credit[msg.sender]>= amount) { credit[msg.sender]-=amount; require(msg.sender.call.value(amount) ()); } }

function queryCredit(address to) view public returns (uint){ return credit[to]; } }

~~~

==
File: SWC-108.md
==

Title

State Variable Default Visibility

Relationships

CWE-710: Improper Adherence to Coding Standards

Description

Labeling the visibility explicitly makes it easier to catch incorrect assumptions about who can access the variable.

Remediation

Variables can be specified as being `public`, `internal` or `private`. Explicitly define visibility for all state variables.

References

- Ethereum Smart Contract Best Practices - Visibility

Samples

storage.sol

```solidity pragma solidity 0.4.24;

contract TestStorage {

~~~
uint storeduint1 = 15;
uint constant constuint = 16;
uint32 investmentsDeadlineTimeStamp = uint32(now);

bytes16 string1 = "test1";
```
```

````
bytes32 private string2 = "test1236";
string public string3 = "lets string something";

mapping (address => uint) public uints1;
mapping (address => DeviceData) structs1;

uint[] uintarray;
DeviceData[] deviceDataArray;

struct DeviceData {
    string deviceBrand;
    string deviceYear;
    string batteryWearLevel;
}

function testStorage() public  {
    address address1 = 0xbccc714d56bc0da0fd33d96d2a87b680dd6d0df6;
    address address2 = 0xaee905fdd3ed851e48d22059575b9f4245a82b04;

    uints1[address1] = 88;
    uints1[address2] = 99;

    DeviceData memory dev1 = DeviceData("deviceBrand", "deviceYear", "wear

    structs1[address1] = dev1;

    uintarray.push(8000);
    uintarray.push(9000);

    deviceDataArray.push(dev1);
}

} ```
````

==

File: SWC-109.md

==

Title

Uninitialized Storage Pointer

Relationships

[CWE-824: Access of Uninitialized Pointer]()

Description

Uninitialized local storage variables can point to unexpected storage locations in the contract, which can lead to intentional or unintentional vulnerabilities.

Remediation

Check if the contract requires a storage object as in many situations this is actually not the case. If a local variable is sufficient, mark the storage location of the variable explicitly with the `memory` attribute. If a storage variable is needed then initialise it upon declaration and additionally specify the storage location `storage`.

Note: As of compiler version 0.5.0 and higher this issue has been systematically resolved as contracts with uninitialised storage pointers do no longer compile.

References

- SigmaPrime - Uninitialised Storage Pointers

Samples

crypto_roulette.sol

```solidity / * *@source: https://github.com/thec00n/smart-contract-honeypots/blob/master/CryptoRoulette.sol* / pragma solidity ^0.4.19;

// CryptoRoulette // // Guess the number secretly stored in the blockchain and win the whole contract balance! // A new number is randomly chosen after each try. // // To play, call the play() method with the guessed number (1-20). Bet price: 0.1 ether

contract CryptoRoulette {

```
uint256 private secretNumber;
uint256 public lastPlayed;
uint256 public betPrice = 0.1 ether;
address public ownerAddr;

struct Game {
    address player;
    uint256 number;
}
Game[] public gamesPlayed;

function CryptoRoulette() public {
    ownerAddr = msg.sender;
    shuffle();
```
```

```
}

function shuffle() internal {
    // randomly set secretNumber with a value between 1 and 20
    secretNumber = uint8(sha3(now, block.blockhash(block.number-1))) % 20
}

function play(uint256 number) payable public {
    require(msg.value >= betPrice && number <= 10);

    Game game;
    game.player = msg.sender;
    game.number = number;
    gamesPlayed.push(game);

    if (number == secretNumber) {
        // win!
        msg.sender.transfer(this.balance);
    }

    shuffle();
    lastPlayed = now;
}

function kill() public {
    if (msg.sender == ownerAddr && now > lastPlayed + 1 days) {
        suicide(msg.sender);
    }
}

function() public payable { }
```

} ```

crypto_roulette_fixed.sol

```solidity / * *@source: https://github.com/thec00n/smart-contract-honeypots/blob/master/CryptoRoulette.sol* / pragma solidity ^0.4.19;

// CryptoRoulette // // Guess the number secretly stored in the blockchain and win the whole contract balance! // A new number is randomly chosen after each try. // // To play, call the play() method with the guessed number (1-20). Bet price: 0.1 ether

contract CryptoRoulette {

```
uint256 private secretNumber;
uint256 public lastPlayed;
uint256 public betPrice = 0.1 ether;
address public ownerAddr;

struct Game {
```
```

```
    address player;
    uint256 number;
}
Game[] public gamesPlayed;

function CryptoRoulette() public {
    ownerAddr = msg.sender;
    shuffle();
}

function shuffle() internal {
    // randomly set secretNumber with a value between 1 and 20
    secretNumber = uint8(sha3(now, block.blockhash(block.number-1))) % 20
}

function play(uint256 number) payable public {
    require(msg.value >= betPrice && number <= 10);

    Game memory game;
    game.player = msg.sender;
    game.number = number;
    gamesPlayed.push(game);

    if (number == secretNumber) {
        // win!
        msg.sender.transfer(this.balance);
    }

    shuffle();
    lastPlayed = now;
}

function kill() public {
    if (msg.sender == ownerAddr && now > lastPlayed + 1 days) {
        suicide(msg.sender);
    }
}

function() public payable { }

}
```

==
File: SWC-110.md
==

Title

Assert Violation

Relationships

CWE-670: Always-Incorrect Control Flow Implementation

Description

The Solidity `assert()` function is meant to assert invariants. Properly functioning code should *never* reach a failing assert statement. A reachable assertion can mean one of two things:

1. A bug exists in the contract that allows it to enter an invalid state;
2. The `assert` statement is used incorrectly, e.g. to validate inputs.

Remediation

Consider whether the condition checked in the `assert()` is actually an invariant. If not, replace the `assert()` statement with a `require()` statement.

If the exception is indeed caused by unexpected behaviour of the code, fix the underlying bug(s) that allow the assertion to be violated.

References

- The use of revert(), assert(), and require() in Solidity, and the new REVERT opcode in the EVM

Samples

assert_constructor.sol

\`\`\`solidity / * *@source: https://github.com/ConsenSys/evm-analyzer-benchmark-suite * @author: Suhabe Bugrara* /

pragma solidity ^0.4.19;

contract AssertConstructor { function AssertConstructor() public { assert(false); } }

\`\`\`

assert_minimal.sol

\`\`\`solidity / * *@source: https://github.com/ConsenSys/evm-analyzer-benchmark-suite * @author: Suhabe Bugrara* /

pragma solidity ^0.4.19;

contract AssertMinimal { function run() public { assert(false); } }

```

## assert_multitx_1.sol

```solidity / * *@source: https://github.com/ConsenSys/evm-analyzer-benchmark-suite * @author: Suhabe Bugrara* /

pragma solidity ^0.4.19;

contract AssertMultiTx1 { uint256 private param;

~~~
function AssertMultiTx1(uint256 _param) public {
    require(_param > 0);
    param = _param;
}

function run() {
    assert(param > 0);
}
~~~

}

```

assert_multitx_2.sol

```solidity / * *@source: https://github.com/ConsenSys/evm-analyzer-benchmark-suite * @author: Suhabe Bugrara* /

pragma solidity ^0.4.19;

contract AssertMultiTx2 { uint256 private param;

~~~
function AssertMultiTx2(uint256 _param) public {
    param = 0;
}

function run() {
    assert(param > 0);
}

function set(uint256 _param) {
    param = _param;
}
~~~

}

```

constructor_create.sol

```solidity / * *@source: ChainSecurity * @author: Anton Permenev* /

pragma solidity ^0.4.25;

contract ConstructorCreate{ B b = new B();

```
function check(){
    assert(b.foo() == 10);
}
```

}

contract B{

```
function foo() returns(uint){
    return 11;
}
```

}

```` ``` ````

constructor_create_argument.sol

```` ``` ````solidity / * *@source: ChainSecurity* * *@author: Anton Permenev* / pragma solidity ^0.4.22*;*

contract ConstructorCreateArgument{ B b = new B(11);

```
function check(){
    assert(b.foo() == 10);
}
```

}

contract B{

```
uint x_;
constructor(uint x){
    x_ = x;
}

function foo() returns(uint){
    return x_;
}
```

}

```` ``` ````

constructor_create_modifiable.sol

```` ``` ````solidity / * *@source: ChainSecurity* * *@author: Anton Permenev* * *Assert violation with 2 message calls:* * *- B.set_x(X): X != 10* * *- ContructorCreateModifiable.check()* /

pragma solidity ^0.4.22;

contract ContructorCreateModifiable{ B b = new B(10);

```
function check(){
    assert(b.foo() == 10);
}
```

}

contract B{

```
uint x_;
constructor(uint x){
    x_ = x;
}
```

```
function foo() returns(uint){
    return x_;
}
```

```
function set_x(uint x){
    x_ = x;
}
```

```
}
```

\`\`\`

gas_model.sol

\`\`\`solidity / * *@source: ChainSecurity* * *@author: Anton Permenev* / pragma solidity ^0.4.21;

contract GasModel{ uint x = 100; function check(){ uint a = gasleft(); x = x + 1; uint b = gasleft(); assert(b > a); } }

\`\`\`

gas_model_fixed.sol

\`\`\`solidity / * *@source: ChainSecurity* * *@author: Anton Permenev* / pragma solidity ^0.4.21;

contract GasModelFixed{ uint x = 100; function check(){ uint a = gasleft(); x = x + 1; uint b = gasleft(); assert(b < a); } }

\`\`\`

mapping_perfomance_2.sol

\`\`\`solidity / * *@source: ChainSecurity* * *@author: Anton Permenev* / pragma solidity ^0.4.22;

contract MappingPerformance2sets{

```
mapping(bytes32=>uint) m0;
mapping(bytes32=>uint) m1;
mapping(bytes32=>uint) m2;
mapping(bytes32=>uint) m3;
mapping(bytes32=>uint) m4;
mapping(bytes32=>uint) m5;
uint b;

constructor(){
    b = 10;
}

function set(bytes32 a, uint cond){
    if(cond == 0){
        m0[a] = 5;
    }else if(cond == 1){
        m1[a] = 5;
    }else if(cond == 2){
        m2[a] = 5;
    }else if(cond == 3){
        m3[a] = 5;
    }else if(cond == 4){
        m4[a] = 5;
    }
}
function check(bytes32 a0, uint cond0,
              bytes32 a1, uint cond1, bytes32 a){
                  set(a0, cond0);
                  set(a1, cond1);
                  assert(m5[a] == 0);
}
```

}

~~~

mapping_performance_1.sol

```solidity / * *@source: ChainSecurity* * *@author: Anton Permenev* / pragma solidity ^0.4.22;

contract MappingPerformance1set{

~~~
mapping(bytes32=>uint) m0;
mapping(bytes32=>uint) m1;
mapping(bytes32=>uint) m2;
mapping(bytes32=>uint) m3;
mapping(bytes32=>uint) m4;
mapping(bytes32=>uint) m5;
uint b;
```
```

~~~
constructor(){
    b = 10;
}

function set(bytes32 a, uint cond){
    if(cond == 0){
        m0[a] = 5;
    }else if(cond == 1){
        m1[a] = 5;
    }else if(cond == 2){
        m2[a] = 5;
    }else if(cond == 3){
        m3[a] = 5;
    }else if(cond == 4){
        m4[a] = 5;
    }
}
function check(bytes32 a0, uint cond0, bytes32 a){
                  set(a0, cond0);
                  assert(m5[a] == 0);
}

}
~~~

```

## out-of-bounds-exception.sol

```solidity pragma solidity ^0.5.0;

contract OutOfBoundsException {

~~~
uint256[] private array;

function getArrayElement(uint256 idx) public returns (uint256) {
    return array[idx];
}

}
~~~

```

return_memory.sol

```solidity /* * @source: https://forum.zeppelin.solutions/t/using-automatic-analysis-tools-with-makerdao-contracts/1021/3 * Author: Dan Guido / Trail of Bits * Slightly modified by Bernhard Mueller

- An assertion violation is possible in 3 transactions: *
- etch(addr)
- lookup(slate, addr)

- checkAnInvariant()
- Whereby slate == Keccak(addr) *
- Ideally tools should output the correct transaction trace. */

pragma solidity ^0.5.0;

contract ReturnMemory { mapping(bytes32=>address) public slates; bool everMatched = false;

```
function etch(address yay) public returns (bytes32 slate) {
    bytes32 hash = keccak256(abi.encodePacked(yay));
    slates[hash] = yay;
    return hash;
}
```

```
function lookup(bytes32 slate, address nay) public {
   if (nay != address(0x0)) {
     everMatched = slates[slate] == nay;
   }
}
```

```
function checkAnInvariant() public returns (bool) {
    assert(!everMatched);
}
```

} ```

runtime_create_user_input.sol

```solidity / * *@source: ChainSecurity* * *@author: Anton Permenev* / pragma solidity ^0.4.22;

contract RuntimeCreateUserInput{

```
function check(uint x){
    B b = new B(x);
    assert(b.foo() == 10);
}
```

}

contract B{

```
uint x_;
constructor(uint x){
    x_ = x;
}
```

```
function foo() returns(uint){
    return x_;
}
```
```

}

```

## runtime_user_input_call.sol

```solidity / * *@source: ChainSecurity* * *@author: Anton Permenev* / pragma solidity ^0.4.19;

contract RuntimeUserInputCall{

~~~
function check(address b){
    assert(B(b).foo() == 10);
}
~~~

}

contract B{ function foo() returns(uint); }

```

sha_of_sha_2_mappings.sol

```solidity / * *@source: ChainSecurity* * *@author: Anton Permenev* / pragma solidity ^0.4.22;

contract ShaOfSha2Mappings{

~~~
mapping(bytes32=>uint) m;
mapping(bytes32=>uint) n;

constructor(){
    m[keccak256(abi.encode("AAA", msg.sender))] = 100;
}

function check(address a){
    assert(n[keccak256(abi.encode("BBB", a))] == 0);
}
~~~

}

```

sha_of_sha_collision.sol

```solidity / * *@source: ChainSecurity* * *@author: Anton Permenev* * *Assert violation with 2 message calls:* * *- set(66)* * *- check(0x4100000000000000000000000000000000000000000000000000000000000000*
pragma solidity ^0.4.22;

contract ShaOfShaCollission{

```
mapping(bytes32=>uint) m;

function set(uint x){
    m[keccak256(abi.encodePacked("A", x))] = 1;
}
function check(uint x){
    assert(m[keccak256(abi.encodePacked(x, "B"))] == 0);
}

}
```

\`\`\`

sha_of_sha_concrete.sol

\`\`\`solidity / * *@source: ChainSecurity * @author: Anton Permenev* / pragma solidity ^0.4.22*;*

contract ShaOfShaConcrete{

```
mapping(bytes32=>uint) m;
uint b;

constructor(){
    b = 1;
}

function check(uint x){
    assert(m[keccak256(abi.encodePacked(x, "B"))] == 0);
}

}
```

\`\`\`

token-with-backdoor.sol

\`\`\`solidity / * *@source: TrailofBits workshop at TruffleCon 2018 * @author: Josselin Feist (adapted for SWC by Bernhard Mueller) * Assert violation with 3 message calls: * - airdrop() * - backdoor() * - test_invariants()* / pragma solidity ^0.4.22*;*

contract Token{

```
mapping(address => uint) public balances;
function airdrop() public{
    balances[msg.sender] = 1000;
}

function consume() public{
    require(balances[msg.sender]>0);
    balances[msg.sender] -= 1;
}

function backdoor() public{
    balances[msg.sender] += 1;
}
```

function test_invariants() { assert(balances[msg.sender] <= 1000); } }

two_mapppings.sol

```solidity pragma solidity ^0.4.22;

contract TwoMappings{

```
mapping(uint=>uint) m;
mapping(uint=>uint) n;

constructor(){
    m[10] = 100;
}

function check(uint a){
    assert(n[a] == 0);
}
```

}

```

simpledschief.sol

```solidity / * *@source: https://forum.zeppelin.solutions/t/using-automatic-analysis-tools-with-makerdao-contracts/1021/3 * Author: Vera Bogdanich Espina / Zeppelin Solutions * * A simplified version of the MakerDAO DSChief contract. * Tools should output the correct transaction trace (see source link). /*

contract SimpleDSChief { mapping(bytes32=>address) public slates; mapping(address=>bytes32) public votes; mapping(address=>uint256) public approvals; mapping(address=>uint256) public deposits;

```
function lock(uint wad) public {
    deposits[msg.sender] = add(deposits[msg.sender], wad);
    addWeight(wad, votes[msg.sender]);
}

function free(uint wad) public {
    deposits[msg.sender] = sub(deposits[msg.sender], wad);
    subWeight(wad, votes[msg.sender]);
}
```
```

````
function voteYays(address yay) public returns (bytes32){
    bytes32 slate = etch(yay);
    voteSlate(slate);

    return slate;
}

function etch(address yay) public returns (bytes32 slate) {
    bytes32 hash = keccak256(abi.encodePacked(yay));

    slates[hash] = yay;

    return hash;
}

function voteSlate(bytes32 slate) public {
    uint weight = deposits[msg.sender];
    subWeight(weight, votes[msg.sender]);
    votes[msg.sender] = slate;
    addWeight(weight, votes[msg.sender]);
}

function addWeight(uint weight, bytes32 slate) internal {
    address yay = slates[slate];
    approvals[yay] = add(approvals[yay], weight);
}

function subWeight(uint weight, bytes32 slate) internal {
    address yay = slates[slate];
    approvals[yay] = sub(approvals[yay], weight);
}

function add(uint x, uint y) internal pure returns (uint z) {
    require((z = x + y) >= x);
}

function sub(uint x, uint y) internal pure returns (uint z) {
    require((z = x - y) <= x);
}
````

function checkAnInvariant() public { bytes32 senderSlate = votes[msg.sender]; address option = slates[senderSlate]; uint256 senderDeposit = deposits[msg.sender];

````
    assert(approvals[option] >= senderDeposit);
}

} ```
````

==
File: SWC-111.md
==

Title

Use of Deprecated Solidity Functions

Relationships

CWE-477: Use of Obsolete Function

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.

Remediation

Solidity provides alternatives to the deprecated constructions. Most of them are aliases, thus replacing old constructions will not break current behavior. For example, `sha3` can be replaced with `keccak256`.

| 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 |

References

- List of global variables and functions, as of Solidity 0.4.25
- Error handling: Assert, Require, Revert and Exceptions
- View functions
- Untyped declaration is deprecated as of Solidity 0.4.20
- Solidity compiler changelog

Samples

deprecated_simple.sol

```solidity pragma solidity ^0.4.24;

contract DeprecatedSimple {
```

```
// Do everything that's deprecated, then commit suicide.

function useDeprecated() public constant {

    bytes32 blockhash = block.blockhash(0);
    bytes32 hashofhash = sha3(blockhash);

    uint gas = msg.gas;

    if (gas == 0) {
        throw;
    }

    address(this).callcode();

    var a = [1,2,3];

    var (x, y, z) = (false, "test", 0);

    suicide(address(0));
}

function () public {}

}
```

\`\`\`

deprecated_simple_fixed.sol

\`\`\`solidity pragma solidity ^0.4.24;

contract DeprecatedSimpleFixed {

```
function useDeprecatedFixed() public view {

    bytes32 bhash = blockhash(0);
    bytes32 hashofhash = keccak256(bhash);

    uint gas = gasleft();

    if (gas == 0) {
        revert();
    }

    address(this).delegatecall();

    uint8[3] memory a = [1,2,3];

    (bool x, string memory y, uint8 z) = (false, "test", 0);

    selfdestruct(address(0));
}

function () external {}

}
```

```

================================================================
File: SWC-112.md
================================================================
```

Title

Delegatecall to Untrusted Callee

Relationships

CWE-829: Inclusion of Functionality from Untrusted Control Sphere

Description

There exists a special variant of a message call, named `delegatecall` which is identical to a message call apart from the fact that the code at the target address is executed in the context of the calling contract and `msg.sender` and `msg.value` do not change their values. This allows a smart contract to dynamically load code from a different address at runtime. Storage, current address and balance still refer to the calling contract.

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.

Remediation

Use `delegatecall` with caution and make sure to never call into untrusted contracts. If the target address is derived from user input ensure to check it against a whitelist of trusted contracts.

References

- Solidity Documentation - Delegatecall / Callcode and Libraries
- How to Secure Your Smart Contracts: 6 Solidity Vulnerabilities and how to avoid them (Part 1) - Delegate Call
- Solidity Security: Comprehensive list of known attack vectors and common anti-patterns - Delegatecall

Samples

proxy.sol

```solidity
pragma solidity ^0.4.24;

contract Proxy {

address owner;

constructor() public { owner = msg.sender;
}

function forward(address callee, bytes _data) public
{ require(callee.delegatecall(_data)); }

}
```

proxy_fixed.sol

```solidity
pragma solidity ^0.4.24;

contract Proxy {

address callee; address owner;

modifier onlyOwner { require(msg.sender == owner); _; }

constructor() public { callee = address(0x0); owner = msg.sender; }

function setCallee(address newCallee) public onlyOwner { callee =
newCallee; }

function forward(bytes _data) public { require(callee.delegatecall(_data)); }

}
```

proxy_pattern_false_positive.sol

```solidity
pragma solidity ^0.4.24;

contract proxy{ address owner;

function proxyCall(address _to, bytes _data) external { require( !
_to.delegatecall(_data)); } function withdraw() external{ require(msg.sender
== owner); msg.sender.transfer(address(this).balance); } }

/* You can't use proxyCall to change the owner address as either:
```

1) the delegatecall reverts and thus does not change owner 2) the delegatecall does not revert and therefore will cause the proxyCall to revert and preventing owner from changing

This false positive may seem like a really edge case, however since you can revert data back to proxy this patern is useful for proxy architectures */ ```

==
File: SWC-113.md
==

Title

DoS with Failed Call

Relationships

[CWE-703: Improper Check or Handling of Exceptional Conditions]

Description

External calls can fail accidentally or deliberately, which can cause a DoS condition in the contract. To minimize the damage caused by such failures, it is better to isolate each external call into its own transaction that can be initiated by the recipient of the call. This is especially relevant for payments, where it is better to let users withdraw funds rather than push funds to them automatically (this also reduces the chance of problems with the gas limit).

Remediation

It is recommended to follow call best practices:

- Avoid combining multiple calls in a single transaction, especially when calls are executed as part of a loop
- Always assume that external calls can fail
- Implement the contract logic to handle failed calls

References

- [Ethereum Smart Contract Best Practices - Favor pull over push for external calls]

Samples

send_loop.sol

```solidity / * *@source: https://consensys.github.io/smart-contract-best-practices/known_attacks/#dos-with-unexpected-revert * @author: ConsenSys Diligence * Modified by Bernhard Mueller* /

pragma solidity 0.4.24;

contract Refunder {

address[] private refundAddresses; mapping (address => uint) public refunds;

```
constructor() {
    refundAddresses.push(0x79B483371E87d664cd39491b5F06250165e4b184);
    refundAddresses.push(0x79B483371E87d664cd39491b5F06250165e4b185);
}

// bad
function refundAll() public {
    for(uint x; x < refundAddresses.length; x++) { // arbitrary length ite
        require(refundAddresses[x].send(refunds[refundAddresses[x]])); //
    }
}
```

}

```

```
==============================================================
```
File: SWC-114.md
```
==============================================================
```

Title

Transaction Order Dependence

Relationships

[CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')]

Description

The Ethereum network processes transactions in blocks, with new blocks getting confirmed approximately every 17 seconds. Miners review the transactions they have received and select which ones to include in a block, based on who has paid a high enough gas price to be included. Additionally, when transactions are sent to the Ethereum network, they are forwarded to each node for processing. Thus, a person who is running an Ethereum node can tell which transactions are going to occur before they are finalized. A race condition vulnerability occurs when code depends on the order of the transactions submitted to it.

The simplest example of a race condition is when a smart contract gives a reward for submitting information. Suppose a contract will give out 1 token to the first person who solves a math problem. Alice solves the problem and submits the answer to the network with a standard gas price. Eve runs an Ethereum node and can see the answer to the math problem in the transaction that Alice submitted to the network. So, Eve submits the answer to the network with a much higher gas price, and thus it gets processed and committed before Alice's transaction. Eve receives one token, and Alice gets nothing, even though it was Alice who worked to solve the problem. A common way this occurs in practice is when a contract rewards people for calling out bad behavior in a protocol by giving a bad actor's deposit to the person who proved they were misbehaving.

The race condition that happens most frequently on the network today is the race condition in the ERC20 token standard. The ERC20 token standard includes a function called 'approve', which allows an address to approve another address to spend tokens on their behalf. Assume that Alice has approved Eve to spend n of her tokens, then Alice decides to change Eve's approval to m tokens. Alice submits a function call to approve with the value n for Eve. Eve runs an Ethereum node, so she knows that Alice is going to change her approval to m. Eve then submits a transferFrom request, sending n of Alice's tokens to herself, but gives it a much higher gas price than Alice's transaction. The transferFrom executes first so gives Eve n tokens and sets Eve's approval to zero. Then Alice's transaction executes and sets Eve's approval to m. Eve then sends those m tokens to herself as well. Thus, Eve gets n + m tokens, even though she should have gotten at most max(n,m).

Remediation

A possible way to remedy race conditions in the submission of information in exchange for a reward is called a commit reveal hash scheme. Instead of submitting the answer, the party who has the answer submits hash(salt, address, answer) [salt being some number of their choosing]; the contract stores this hash and the sender's address. To claim the reward, the sender then submits a transaction with the salt, and answer. The contract hashes (salt, msg.sender, answer) and checks the hash produced against the stored hash. If the hash matches, the contract releases the reward.

The best fix for the ERC20 race condition is to add a field to the inputs of approve, which is the expected current value, and to have approve revert if Eve's current allowance is not what Alice indicated she was expecting. However, this means that your contract no longer conforms to the ERC20 standard. If it is important to your project to have the contract conform to

ERC20, you can add a safe approve function. From the user's perspective, it is possible to mitigate the ERC20 race condition by setting approvals to zero before changing them.

References

General Article on Race Conditions ERC20 Race Condition

Samples

ERC20.sol

```solidity pragma solidity ^0.4.24;

/* *Taken from the OpenZeppelin github * @title SafeMath * @dev Math operations with safety checks that revert on error* / library SafeMath {

/* * *@dev Multiplies two numbers, reverts on overflow.* / function mul(uint256 a, uint256 b) internal pure returns (uint256) { // Gas optimization: this is cheaper than requiring 'a' not being zero, but the // benefit is lost if 'b' is also tested. // See: https://github.com/OpenZeppelin/openzeppelin-solidity/pull/522 if (a == 0) { return 0; }

```
uint256 c = a * b;
require(c / a == b);

return c;
```

}

/* * *@dev Integer division of two numbers truncating the quotient, reverts on division by zero.* / function div(uint256 a, uint256 b) internal pure returns (uint256) { require(b > 0); // Solidity only automatically asserts when dividing by 0 uint256 c = a / b; // assert(a == b * c + a % b); // There is no case in which this doesn't hold

```
return c;
```

}

/* * *@dev Subtracts two numbers, reverts on overflow (i.e. if subtrahend is greater than minuend).* / function sub(uint256 a, uint256 b) internal pure returns (uint256) { require(b <= a); uint256 c = a - b;

```
return c;
```

}

/* * *@dev Adds two numbers, reverts on overflow.* / function add(uint256 a, uint256 b) internal pure returns (uint256) { uint256 c = a + b; require(c >= a);
```

```
return c;
```

}

/ * @dev Divides two numbers and returns the remainder (unsigned integer modulo), * reverts when dividing by zero.* / function mod(uint256 a, uint256 b) internal pure returns (uint256) { require(b != 0); return a % b; } }

contract ERC20 {

event Transfer(address indexed from, address indexed to, uint256 value); event Approval(address indexed owner, address indexed spender, uint256 value); using SafeMath for *;

mapping (address => uint256) private _balances;

mapping (address => mapping (address => uint256)) private _allowed;

uint256 private _totalSupply;

constructor(uint totalSupply){ _balances[msg.sender] = totalSupply; }

function balanceOf(address owner) public view returns (uint256) { return _balances[owner]; }

function allowance(address owner, address spender) public view returns (uint256) { return _allowed[owner][spender]; }

function transfer(address to, uint256 value) public returns (bool) { require(value <= _balances[msg.sender]); require(to != address(0));

```
_balances[msg.sender] = _balances[msg.sender].sub(value);
_balances[to] = _balances[to].add(value);
emit Transfer(msg.sender, to, value);
return true;
```

}

function approve(address spender, uint256 value) public returns (bool) { require(spender != address(0));

```
_allowed[msg.sender][spender] = value;
emit Approval(msg.sender, spender, value);
return true;
```

}

function transferFrom(address from, address to, uint256 value) public returns (bool) { require(value <= _balances[from]); require(value <= _allowed[from][msg.sender]); require(to != address(0));

```
_balances[from] = _balances[from].sub(value);
_balances[to] = _balances[to].add(value);
_allowed[from][msg.sender] = _allowed[from][msg.sender].sub(value);
```

```
emit Transfer(from, to, value);
return true;
```

} }

eth_tx_order_dependence_minimal.sol

```solidity / * *@source: https://github.com/ConsenSys/evm-analyzer-benchmark-suite * @author: Suhabe Bugrara /*

pragma solidity ^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;
}
```

}

```

==
File: SWC-115.md
==

Title

Authorization through tx.origin

Relationships

- CWE-477: Use of Obsolete Function
- EEA EthTrust Security Levels:
- **[S] No `tx.origin`**
- **[Q] Verify `tx.origin` Usage**

Description

`tx.origin` is a global variable in Solidity which returns the address of the account that sent the transaction. Using the variable for authorization could make a contract vulnerable if an authorized account calls into a malicious contract. A call could be made to the vulnerable contract that passes the authorization check since `tx.origin` returns the original sender of the transaction which in this case is the authorized account.

Remediation

`tx.origin` should not be used for authorization. Use `msg.sender` instead.

References

- Solidity Documentation - tx.origin
- Ethereum Smart Contract Best Practices - Avoid using tx.origin
- SigmaPrime - Visibility

Samples

mycontract.sol

```solidity / * *@source: https://consensys.github.io/smart-contract-best-practices/recommendations/#avoid-using-txorigin * @author: Consensys Diligence*
* *Modified by Gerhard Wagner* /

pragma solidity 0.4.24;

contract MyContract {

```
address owner;

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

function sendTo(address receiver, uint amount) public {
    require(tx.origin == owner);
    receiver.transfer(amount);
}
```
```

} ```

mycontract_fixed.sol

```solidity / * *@source: https://consensys.github.io/smart-contract-best-practices/recommendations/#avoid-using-txorigin * @author: Consensys Diligence * Modified by Gerhard Wagner /*

pragma solidity 0.4.25;

contract MyContract {

```
address owner;

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

function sendTo(address receiver, uint amount) public {
  require(msg.sender == owner);
  receiver.transfer(amount);
}
```

}

```

```
===================================================
```
File: SWC-116.md
```
===================================================
```

Title

Block values as a proxy for time

Relationships

- CWE-829: Inclusion of Functionality from Untrusted Control Sphere
- **[M] Document Special Code Use**
- **[M] Don't Misuse Block Data**

Description

Contracts often need access to time values to perform certain types of functionality. Values such as `block.timestamp`, and `block.number` can give you a sense of the current time or a time delta, however, they are not safe to use for most purposes.

In the case of `block.timestamp`, developers often attempt to use it to trigger time-dependent events. As Ethereum is decentralized, nodes can synchronize time only to some degree. Moreover, malicious miners can alter the timestamp of their blocks, especially if they can gain advantages by doing so. However, miners can't set a timestamp smaller than the previous one (otherwise the block will be rejected), nor can they set the timestamp too far ahead in the future. Taking all of the above into consideration, developers can't rely on the preciseness of the provided timestamp.

As for `block.number`, considering the block time on Ethereum is generally about 14 seconds, it's possible to predict the time delta between blocks. However, block times are not constant and are subject to change for a variety of reasons, e.g. fork reorganisations and the difficulty bomb. Due to variable block times, `block.number` should also not be relied on for precise calculations of time.

Remediation

Developers should write smart contracts with the notion that block values are not precise, and the use of them can lead to unexpected effects. Alternatively, they may make use of oracles.

References

- Safety: Timestamp dependence
- Ethereum Smart Contract Best Practices - Timestamp Dependence
- How do Ethereum mining nodes maintain a time consistent with the network?
- Solidity: Timestamp dependency, is it possible to do safely?
- Avoid using block.number as a timestamp

Samples

time_lock.sol

```solidity / * *@author: Kaden Zipfel* /

pragma solidity ^0.5.0;

contract TimeLock { struct User { uint amount; // amount locked (in eth) uint unlockBlock; // minimum block to unlock eth }

```
mapping(address => User) private users;

// Tokens should be locked for exact time specified
function lockEth(uint _time, uint _amount) public payable {
    require(msg.value == _amount, 'must send exact amount');
    users[msg.sender].unlockBlock = block.number + (_time / 14);
    users[msg.sender].amount = _amount;
}

// Withdraw tokens if lock period is over
```
```

```
function withdraw() public {
    require(users[msg.sender].amount > 0, 'no amount locked');
    require(block.number >= users[msg.sender].unlockBlock, 'lock period no

    uint amount = users[msg.sender].amount;
    users[msg.sender].amount = 0;
    (bool success, ) = msg.sender.call.value(amount)("");
    require(success, 'transfer failed');
}
```

}

\`\`\`

timed_crowdsale.sol

\`\`\`solidity pragma solidity ^0.5.0;

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(); } }

} \`\`\`

==
File: SWC-117.md
==

Title

Signature Malleability

Relationships

[CWE-347: Improper Verification of Cryptographic Signature]

Description

The implementation of a cryptographic signature system in Ethereum contracts often assumes that the signature is unique, but signatures can be altered without the possession of the private key and still be valid. The EVM specification defines several so-called ‘precompiled’ contracts one of them being `ecrecover` which executes the elliptic curve public key recovery. A malicious user can slightly modify the three values *v*, *r* and *s* to create other valid signatures. A system that performs signature verification on contract level might be susceptible to attacks if the signature is part of the signed message hash. Valid signatures could be created by a malicious user to replay previously signed messages.

Remediation

A signature should never be included into a signed message hash to check if previously messages have been processed by the contract.

References

Bitcoin Transaction Malleability CTF - Challenge

Samples

transaction_malleablity.sol

```solidity pragma solidity ^0.4.24;

contract transaction_malleablity{ mapping(address => uint256) balances; mapping(bytes32 => bool) signatureUsed;

constructor(address[] owners, uint[] init){ require(owners.length == init.length); for(uint i=0; i < owners.length; i ++){ balances[owners[i]] = init[i]; } }

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)); require(!signatureUsed[txid]);

```
  address from = recoverTransferPreSigned(_signature, _to, _value, _gasPri

  require(balances[from] > _value);
  balances[from] -= _value;
  balances[_to] += _value;

  signatureUsed[txid] = true;
}

function recoverTransferPreSigned(
    bytes _sig,
    address _to,
    uint256 _value,
    uint256 _gasPrice,
    uint256 _nonce)
  public
  view
returns (address recovered)
```
```

````
{
    return ecrecoverFromSig(getSignHash(getTransferHash(_to, _value, _gasP
}

function getTransferHash(
    address _to,
    uint256 _value,
    uint256 _gasPrice,
    uint256 _nonce)
  public
  view
returns (bytes32 txHash) {
    return keccak256(address(this), bytes4(0x1296830d), _to, _value, _gasP
}

function getSignHash(bytes32 _hash)
  public
  pure
returns (bytes32 signHash)
{
    return keccak256("\x19Ethereum Signed Message:\n32", _hash);
}

function ecrecoverFromSig(bytes32 hash, bytes sig)
  public
  pure
returns (address recoveredAddress)
{
    bytes32 r;
    bytes32 s;
    uint8 v;
    if (sig.length != 65) return address(0);
    assembly {
        r := mload(add(sig, 32))
        s := mload(add(sig, 64))
        v := byte(0, mload(add(sig, 96)))
    }
    if (v < 27) {
      v += 27;
    }
    if (v != 27 && v != 28) return address(0);
    return ecrecover(hash, v, r, s);
}

}
````

~~~

transaction_malleablity_fixed.sol

```solidity pragma solidity ^0.4.24;
~~~

```
contract transaction_malleablity{ mapping(address => uint256) balances;
mapping(bytes32 => bool) signatureUsed;

constructor(address[] owners, uint[] init){ require(owners.length ==
init.length); for(uint i=0; i < owners.length; i ++){ balances[owners[i]] =
init[i]; } }

function transfer( bytes _signature, address _to, uint256 _value, uint256
_gasPrice, uint256 _nonce) public returns (bool) { bytes32 txid =
getTransferHash(_to, _value, _gasPrice, _nonce); require(!
signatureUsed[txid]);

  address from = recoverTransferPreSigned(_signature, _to, _value, _gasPri

  require(balances[from] > _value);
  balances[from] -= _value;
  balances[_to] += _value;

  signatureUsed[txid] = true;
}

function recoverTransferPreSigned(
    bytes _sig,
    address _to,
    uint256 _value,
    uint256 _gasPrice,
    uint256 _nonce)
  public
  view
returns (address recovered)
{
    return ecrecoverFromSig(getSignHash(getTransferHash(_to, _value, _gasP
}

function getTransferHash(
    address _to,
    uint256 _value,
    uint256 _gasPrice,
    uint256 _nonce)
  public
  view
returns (bytes32 txHash) {
    return keccak256(address(this), bytes4(0x1296830d), _to, _value, _gasP
}

function getSignHash(bytes32 _hash)
  public
  pure
returns (bytes32 signHash)
{
    return keccak256("\x19Ethereum Signed Message:\n32", _hash);
}
function ecrecoverFromSig(bytes32 hash, bytes sig)
  public
  pure
returns (address recoveredAddress)
{
    bytes32 r;
    bytes32 s;
    uint8 v;
    if (sig.length != 65) return address(0);
    assembly {
        r := mload(add(sig, 32))
        s := mload(add(sig, 64))
        v := byte(0, mload(add(sig, 96)))
    }
    if (v < 27) {
      v += 27;
    }
    if (v != 27 && v != 28) return address(0);
    return ecrecover(hash, v, r, s);
}

}
```

==
File: SWC-118.md
==

Incorrect Constructor Name

Relationships

- CWE-665: Improper Initialization
- EthTrust Security Levels
- **[S] Use a Modern Compiler**
- **[Q] Code Linting**

Description

Constructors are special functions that are called only once during the contract creation. They often perform critical, privileged actions such as setting the owner of the contract. Before Solidity version 0.4.22, the only way of defining a constructor was to create a function with the same name as the contract class containing it. A function meant to become a constructor becomes a normal, callable function if its name doesn't exactly match the contract name. This behavior sometimes leads to security issues, in particular when smart contract code is re-used with a different name but the name of the constructor function is not changed accordingly.

Remediation

Solidity version 0.4.22 introduces a new `constructor` keyword that make a constructor definitions clearer. It is therefore recommended to upgrade the contract to a recent version of the Solidity compiler and change to the new constructor declaration.

References

- SigmaPrime - Constructors with Care

Samples

incorrect_constructor_name1.sol

``` solidity / * *@source: https://github.com/trailofbits/not-so-smart-contracts/blob/master/wrong_constructor_name/incorrect_constructor.sol * @author: Ben Perez * Modified by Gerhard Wagner* /

pragma solidity 0.4.24;

contract Missing{ address private owner;

```
modifier onlyowner {
    require(msg.sender==owner);
    _;
}

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

function () payable {}

function withdraw()
    public
    onlyowner
{
   owner.transfer(this.balance);
}

}
```

```

## incorrect_constructor_name1_fixed.sol

```solidity / * *@source: https://github.com/trailofbits/not-so-smart-contracts/blob/master/wrong_constructor_name/incorrect_constructor.sol * @author: Ben Perez * Modified by Gerhard Wagner* /

pragma solidity ^0.4.24;

contract Missing{ address private owner;

~~~
modifier onlyowner {
    require(msg.sender==owner);
    _;
}

constructor()
    public
{
    owner = msg.sender;
}

function () payable {}

function withdraw()
    public
    onlyowner
{
   owner.transfer(this.balance);
}
~~~

}

```

incorrect_constructor_name2.sol

```solidity / * *@source: https://github.com/trailofbits/not-so-smart-contracts/blob/master/wrong_constructor_name/incorrect_constructor.sol * @author: Ben Perez * Modified by Gerhard Wagner* /

pragma solidity 0.4.24;

contract Missing{ address private owner;

~~~
modifier onlyowner {
    require(msg.sender==owner);
    _;
}

function Constructor()
    public
{
~~~
```

~~~
    owner = msg.sender;
}

function () payable {}

function withdraw()
    public
    onlyowner
{
   owner.transfer(this.balance);
}

}
~~~

```

## incorrect_constructor_name2_fixed.sol

```solidity / *@source: https://github.com/trailofbits/not-so-smart-contracts/blob/master/wrong_constructor_name/incorrect_constructor.sol * @author: Ben Perez * Modified by Gerhard Wagner /*

pragma solidity ^0.4.24;

contract Missing{ address private owner;

~~~
modifier onlyowner {
    require(msg.sender==owner);
    _;
}

constructor()
    public
{
    owner = msg.sender;
}

function () payable {}

function withdraw()
    public
    onlyowner
{
   owner.transfer(this.balance);
}

}
~~~

```

==
File: SWC-119.md
==
```

# Title

Shadowing State Variables

## Relationships

- CWE-710: Improper Adherence to Coding Standards
- EthTrust Security Levels:
- **[Q] Implement as Documented**

## 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.

Shadowing state variables can also occur within a single contract when there are multiple definitions on the contract and function level.

## Remediation

Review storage variable layouts for your contract systems carefully and remove any ambiguities. Always check for compiler warnings as they can flag the issue within a single contract.

## References

- Issue on Solidity's Github - Shadowing of inherited state variables should be an error (override keyword)
- Issue on Solidity's Github - Warn about shadowing state variables

EthTrust Security Levels: - **[Q] Implement as Documented**

## Samples

### ShadowingInFunctions.sol

```solidity pragma solidity 0.4.24;

contract ShadowingInFunctions { uint n = 2; uint x = 3;

```
function test1() constant returns (uint n) {
    return n; // Will return 0
}
```
```

```
function test2() constant returns (uint n) {
    n = 1;
    return n; // Will return 1
}

function test3() constant returns (uint x) {
    uint n = 4;
    return n+x; // Will return 4
}

}
```

\`\`\`

TokenSale.sol

\`\`\`solidity pragma solidity 0.4.24;

contract Tokensale { uint hardcap = 10000 ether;

```
function Tokensale() {}

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

}
```

contract Presale is Tokensale { uint hardcap = 1000 ether;

```
function Presale() Tokensale() {}

}
```

\`\`\`

TokenSale_fixed.sol

\`\`\`solidity pragma solidity 0.4.25;

//We fix the problem by eliminating the declaration which overrides the prefered hardcap.

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; //We set the hardcap from the constructor for the Tokensale to be 1000 instead of 10000 } }

```

================================================================
File: SWC-120.md
================================================================

# Title

Weak Sources of Randomness from Chain Attributes

## Relationships

- CWE-330: Use of Insufficiently Random Values
- EthTrust Security Levels:
- **[M] Sources of Randomness**
- **[M] Document Special Code Use**

## Description

Ability to generate random numbers is very helpful in all kinds of applications. One obvious example is gambling DApps, where pseudo-random number generator is used to pick the winner. However, creating a strong enough source of randomness in Ethereum is very challenging. For example, use of `block.timestamp` is insecure, as a miner can choose to provide any timestamp within a few seconds and still get his block accepted by others. Use of `blockhash`, `block.difficulty` and other fields is also insecure, as they're controlled by the miner. If the stakes are high, the miner can mine lots of blocks in a short time by renting hardware, pick the block that has required block hash for him to win, and drop all others.

## Remediation

- Using external sources of randomness via oracles, and cryptographically checking the outcome of the oracle on-chain. e.g. Chainlink VRF. This approach does not rely on trusting the oracle, as a falsly generated random number will be rejected by the on-chain portion of the system.
- Using commitment scheme, e.g. RANDAO.
- Using external sources of randomness via oracles, e.g. Oraclize. Note that this approach requires trusting in oracle, thus it may be reasonable to use multiple oracles.
- Using Bitcoin block hashes, as they are more expensive to mine.
```

References

- How can I securely generate a random number in my smart contract?
- When can BLOCKHASH be safely used for a random number? When would it be unsafe?
- The Run smart contract

Samples

guess_the_random_number.sol

```solidity

*/ * @source: https://capturetheether.com/challenges/lotteries/guess-the-random-number/ * @author: Steve Marx /*

pragma solidity ^0.4.21;

contract GuessTheRandomNumberChallenge { uint8 answer;

```
function GuessTheRandomNumberChallenge() public payable {
    require(msg.value == 1 ether);
    answer = uint8(keccak256(block.blockhash(block.number - 1), now));
}

function isComplete() public view returns (bool) {
    return address(this).balance == 0;
}

function guess(uint8 n) public payable {
    require(msg.value == 1 ether);

    if (n == answer) {
        msg.sender.transfer(2 ether);
    }
}
```

}

```

### guess_the_random_number_fixed.sol

```solidity */ * @source: https://capturetheether.com/challenges/lotteries/guess-the-random-number/ * @author: Steve Marx /*

pragma solidity ^0.4.25;

contract GuessTheRandomNumberChallenge { uint8 answer; uint8 commitedGuess; uint commitBlock; address guesser;
```

~~~
function GuessTheRandomNumberChallenge() public payable {
    require(msg.value == 1 ether);
}

function isComplete() public view returns (bool) {
    return address(this).balance == 0;
}

//Guess the modulo of the blockhash 20 blocks from your guess
function guess(uint8 _guess) public payable {
    require(msg.value == 1 ether);
    commitedGuess = _guess;
    commitBlock = block.number;
    guesser = msg.sender;
}
function recover() public {
  //This must be called after the guessed block and before commitBlock+20'
  require(block.number > commitBlock + 20 && commitBlock+20 > block.number
  require(guesser == msg.sender);

  if(uint(blockhash(commitBlock+20)) == commitedGuess){
    msg.sender.transfer(2 ether);
  }
}

} ```
~~~

old_blockhash.sol

```solidity pragma solidity ^0.4.24;

//Based on the the Capture the Ether challange at https://capturetheether.com/challenges/lotteries/predict-the-block-hash/ //Note that while it seems to have a 1/2^256 chance you guess the right hash, actually blockhash returns zero for blocks numbers that are more than 256 blocks ago so you can guess zero and wait. contract PredictTheBlockHashChallenge {

~~~
struct guess{
  uint block;
  bytes32 guess;
}

mapping(address => guess) guesses;

constructor() public payable {
    require(msg.value == 1 ether);
}

function lockInGuess(bytes32 hash) public payable {
    require(guesses[msg.sender].block == 0);
    require(msg.value == 1 ether);
~~~
```

~~~
    guesses[msg.sender].guess = hash;
    guesses[msg.sender].block  = block.number + 1;
}

function settle() public {
    require(block.number > guesses[msg.sender].block);

    bytes32 answer = blockhash(guesses[msg.sender].block);

    guesses[msg.sender].block = 0;
    if (guesses[msg.sender].guess == answer) {
        msg.sender.transfer(2 ether);
    }
}

}
~~~

```

## old_blockhash_fixed.sol

```solidity pragma solidity ^0.4.24;

//Based on the the Capture the Ether challange at https://capturetheether.com/challenges/lotteries/predict-the-block-hash/ //Note that while it seems to have a 1/2^256 chance you guess the right hash, actually blockhash returns zero for blocks numbers that are more than 256 blocks ago so you can guess zero and wait. contract PredictTheBlockHashChallenge {

~~~
struct guess{
  uint block;
  bytes32 guess;
}

mapping(address => guess) guesses;

constructor() public payable {
    require(msg.value == 1 ether);
}

function lockInGuess(bytes32 hash) public payable {
    require(guesses[msg.sender].block == 0);
    require(msg.value == 1 ether);

    guesses[msg.sender].guess = hash;
    guesses[msg.sender].block  = block.number + 1;
}

function settle() public {
    require(block.number > guesses[msg.sender].block +10);
~~~

```
    //Note that this solution prevents the attack where blockhash(guesses[
    //Also we add ten block cooldown period so that a minner cannot use fo
    if(guesses[msg.sender].block - block.number < 256){
      bytes32 answer = blockhash(guesses[msg.sender].block);

      guesses[msg.sender].block = 0;
      if (guesses[msg.sender].guess == answer) {
          msg.sender.transfer(2 ether);
      }
    }
    else{
      revert("Sorry your lottery ticket has expired");
    }
}

}
```

**random_number_generator.sol**

```solidity pragma solidity ^0.4.25;

// Based on TheRun contract deployed at 0xcac337492149bDB66b088bf5914beDfBf78cCC18. contract RandomNumberGenerator { uint256 private salt = block.timestamp;

function random(uint max) view private returns (uint256 result) { // Get the best seed for randomness uint256 x = salt * 100 / max; uint256 y = salt * block.number / (salt % 5); uint256 seed = block.number / 3 + (salt % 300) + y; uint256 h = uint256(blockhash(seed)); // Random number between 1 and max return uint256((h / x)) % max + 1; } }

```

==
File: SWC-121.md
==

Title

Missing Protection against Signature Replay Attacks

Relationships

CWE-347: Improper Verification of Cryptographic Signature

Description

It is sometimes necessary to perform signature verification in smart contracts to achieve better usability or to save gas cost. A secure implementation needs to protect against Signature Replay Attacks by for example keeping track of all processed message hashes and only allowing new message hashes to be processed. A malicious user could attack a contract without such a control and get message hash that was sent by another user processed multiple times.

Remediation

In order to protect against signature replay attacks consider the following recommendations:

- Store every message hash that has been processed by the smart contract. When new messages are received check against the already existing ones and only proceed with the business logic if it's a new message hash.
- Include the address of the contract that processes the message. This ensures that the message can only be used in a single contract.
- Under no circumstances generate the message hash including the signature. The `ecrecover` function is susceptible to signature malleability (see also SWC-117).

References

- Medium - Replay Attack Vulnerability in Ethereum Smart Contracts Introduced by transferProxy()

==
File: SWC-122.md
==

Title

Lack of Proper Signature Verification

Relationships

CWE-345: Insufficient Verification of Data Authenticity

Description

It is a common pattern for smart contract systems to allow users to sign messages off-chain instead of directly requesting users to do an on-chain transaction because of the flexibility and increased transferability that this provides. Smart contract systems that process signed messages have to implement their own logic to recover the authenticity from the signed messages before they process them further. A limitation for such systems is that smart contracts can not directly interact with them because they can not sign messages. Some signature verification implementations attempt to solve this problem by assuming the validity of a signed message based on other methods that do not have this limitation. An example of such a method is to rely on `msg.sender` and assume that if a signed message originated from the sender address then it has also been created by the sender address. This can lead to vulnerabilities especially in scenarios where proxies can be used to relay transactions.

Remediation

It is not recommended to use alternate verification schemes that do not require proper signature verification through `ecrecover()`.

References

- Consensys Diligence 0x Audit Report - Insecure signature validator

==
File: SWC-123.md
==

Title

Requirement Violation

Relationships

CWE-573: Improper Following of Specification by Caller

Description

The Solidity `require()` construct is meant to validate external inputs of a function. In most cases, such external inputs are provided by callers, but they may also be returned by callees. In the former case, we refer to them as precondition violations. Violations of a requirement can indicate one of two possible issues:

1. A bug exists in the contract that provided the external input.
2. The condition used to express the requirement is too strong.

Remediation

If the required logical condition is too strong, it should be weakened to allow all valid external inputs.

Otherwise, the bug must be in the contract that provided the external input and one should consider fixing its code by making sure no invalid inputs are provided.

References

- The use of revert(), assert(), and require() in Solidity, and the new REVERT opcode in the EVM

Samples

requirement_simple.sol

```solidity pragma solidity ^0.4.25;

contract Bar { Foo private f = new Foo(); function doubleBaz() public view returns (int256) { return 2 * f.baz(0); } }

contract Foo { function baz(int256 x) public pure returns (int256) { require(0 < x); return 42; } }

```

requirement_simple_fixed.sol

```solidity pragma solidity ^0.4.25;

contract Bar { Foo private f = new Foo(); function doubleBaz() public view returns (int256) { return 2 * f.baz(1); //Changes the external contract to not hit the overly strong requirement. } }

contract Foo { function baz(int256 x) public pure returns (int256) { require(0 < x); //You can also fix the contract by changing the input to the uint type and removing the require return 42; } }

```

==
File: SWC-124.md
==

Title

Write to Arbitrary Storage Location

Relationships

CWE-123: Write-what-where Condition

Description

A smart contract's data (e.g., storing the owner of the contract) is persistently stored at some storage location (i.e., a key or address) on the EVM level. The contract is responsible for ensuring that only authorized user or contract accounts may write to sensitive storage locations. If an attacker is able to write to arbitrary storage locations of a contract, the authorization checks may easily be circumvented. This can allow an attacker to corrupt the storage; for instance, by overwriting a field that stores the address of the contract owner.

Remediation

As a general advice, given that all data structures share the same storage (address) space, one should make sure that writes to one data structure cannot inadvertently overwrite entries of another data structure.

References

- Entry to Underhanded Solidity Coding Contest 2017 (honorable mention)

Samples

arbitrary_location_write_simple.sol

\`\`\`solidity pragma solidity ^0.4.25;

contract Wallet { uint[] private bonusCodes; address private owner;

```
constructor() public {
    bonusCodes = new uint[](0);
    owner = msg.sender;
}

function () public payable {
}

function PushBonusCode(uint c) public {
    bonusCodes.push(c);
}

function PopBonusCode() public {
    require(0 <= bonusCodes.length);
    bonusCodes.length--;
}

function UpdateBonusCodeAt(uint idx, uint c) public {
    require(idx < bonusCodes.length);
    bonusCodes[idx] = c;
}

function Destroy() public {
    require(msg.sender == owner);
    selfdestruct(msg.sender);
}

}
```

```
```

arbitrary_location_write_simple_fixed.sol

```solidity pragma solidity ^0.4.25;

contract Wallet { uint[] private bonusCodes; address private owner;

```
constructor() public {
    bonusCodes = new uint[](0);
    owner = msg.sender;
}

function () public payable {
}

function PushBonusCode(uint c) public {
    bonusCodes.push(c);
}

function PopBonusCode() public {
    require(0 < bonusCodes.length);
    bonusCodes.length--;
}

function UpdateBonusCodeAt(uint idx, uint c) public {
    require(idx < bonusCodes.length); //Since you now have to push very co
    bonusCodes[idx] = c;
}

function Destroy() public {
    require(msg.sender == owner);
    selfdestruct(msg.sender);
}

}
```

```
```
```

### mapping_write.sol

```solidity pragma solidity ^0.4.24;

//This code is derived from the Capture the Ether https://capturetheether.com/challenges/math/mapping/

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;
}

function get(uint256 key) public view returns (uint256) {
    return map[key];
}
function withdraw() public{
  require(msg.sender == owner);
  msg.sender.transfer(address(this).balance);
}
```

}

```

==================================================================
File: SWC-125.md
==================================================================

# Title

Incorrect Inheritance Order

## Relationships

CWE-696: Incorrect Behavior Order

## Description

Solidity supports multiple inheritance, meaning that one contract can inherit several contracts. Multiple inheritance introduces ambiguity called Diamond Problem: if two or more base contracts define the same function, which one should be called in the child contract? Solidity deals with this ambiguity by using reverse C3 Linearization, which sets a priority between base contracts.

That way, base contracts have different priorities, so the order of inheritance matters. Neglecting inheritance order can lead to unexpected behavior.

## Remediation

When inheriting multiple contracts, especially if they have identical functions, a developer should carefully specify inheritance in the correct order. The rule of thumb is to inherit contracts from more /general/ to more /specific/.

## References

- Smart Contract Best Practices - Complex Inheritance
- Solidity docs - Multiple Inheritance and Linearization
- Solidity anti-patterns: Fun with inheritance DAG abuse

## Samples

### MDTCrowdsale.sol

```solidity / * *@source: https://github.com/Arachnid/uscc/blob/master/submissions-2017/philipdaian/MDTCrowdsale.sol * @author: Philip Daian* /

pragma solidity ^0.4.25;

//import "https://github.com/OpenZeppelin/openzeppelin-solidity/contracts/math/SafeMath.sol"; / * **@title SafeMath * @dev Math operations with safety checks that revert on error */ library SafeMath { /** * @dev Multiplies two numbers, reverts on overflow. */ function mul(uint256 a, uint256 b) internal pure returns (uint256) { // Gas optimization: this is cheaper than requiring 'a' not being zero, but the // benefit is lost if 'b' is also tested. // See: https://github.com/OpenZeppelin/openzeppelin-solidity/pull/522 if (a == 0) { return 0; }

```
    uint256 c = a * b;
    require(c / a == b);

    return c;
}

/**
* @dev Integer division of two numbers truncating the quotient, reverts on
*/
function div(uint256 a, uint256 b) internal pure returns (uint256) {
    // Solidity only automatically asserts when dividing by 0
    require(b > 0);
    uint256 c = a / b;
    // assert(a == b * c + a % b); // There is no case in which this doesn

    return c;
```
```

```
}

/**
* @dev Subtracts two numbers, reverts on overflow (i.e. if subtrahend is g
*/
function sub(uint256 a, uint256 b) internal pure returns (uint256) {
    require(b <= a);
    uint256 c = a - b;

    return c;
}

/**
* @dev Adds two numbers, reverts on overflow.
*/
function add(uint256 a, uint256 b) internal pure returns (uint256) {
    uint256 c = a + b;
    require(c >= a);

    return c;
}

/**
* @dev Divides two numbers and returns the remainder (unsigned integer mod
* reverts when dividing by zero.
*/
function mod(uint256 a, uint256 b) internal pure returns (uint256) {
    require(b != 0);
    return a % b;
}

}
```

//import "https://github.com/OpenZeppelin/openzeppelin-solidity/contracts/token/ERC20/ERC20Mintable.sol";

/* * *@title ERC20 interface * @dev see https://github.com/ethereum/EIPs/issues/20* / interface IERC20 { function totalSupply() external view returns (uint256);

```
function balanceOf(address who) external view returns (uint256);

function allowance(address owner, address spender) external view returns (

function transfer(address to, uint256 value) external returns (bool);

function approve(address spender, uint256 value) external returns (bool);

function transferFrom(address from, address to, uint256 value) external re

event Transfer(address indexed from, address indexed to, uint256 value);
event Approval(address indexed owner, address indexed spender, uint256 val

}
```

/ * @title Standard ERC20 token * * @dev Implementation of the basic standard token. * https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md * Originally based on code by FirstBlood: https://github.com/Firstbloodio/token/blob/master/smart_contract/FirstBloodToken.sol * * This implementation emits additional Approval events, allowing applications to reconstruct the allowance status for * all accounts just by listening to said events. Note that this isn't required by the specification, and other * compliant implementations may not do it.* / contract ERC20 is IERC20 { using SafeMath for uint256;

```
mapping (address => uint256) private _balances;

mapping (address => mapping (address => uint256)) private _allowed;

uint256 private _totalSupply;

/**
* @dev Total number of tokens in existence
*/
function totalSupply() public view returns (uint256) {
    return _totalSupply;
}

/**
* @dev Gets the balance of the specified address.
* @param owner The address to query the balance of.
* @return An uint256 representing the amount owned by the passed address.
*/
function balanceOf(address owner) public view returns (uint256) {
    return _balances[owner];
}

/**
 * @dev Function to check the amount of tokens that an owner allowed to a
 * @param owner address The address which owns the funds.
 * @param spender address The address which will spend the funds.
 * @return A uint256 specifying the amount of tokens still available for t
 */
function allowance(address owner, address spender) public view returns (ui
    return _allowed[owner][spender];
}

/**
* @dev Transfer token for a specified address
* @param to The address to transfer to.
* @param value The amount to be transferred.
*/
```

```
    function transfer(address to, uint256 value) public returns (bool) {
        _transfer(msg.sender, to, value);
        return true;
    }

    /**
     * @dev Approve the passed address to spend the specified amount of tokens
     * Beware that changing an allowance with this method brings the risk that
     * and the new allowance by unfortunate transaction ordering. One possible
     * race condition is to first reduce the spender's allowance to 0 and set
     * https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
     * @param spender The address which will spend the funds.
     * @param value The amount of tokens to be spent.
     */
    function approve(address spender, uint256 value) public returns (bool) {
        require(spender != address(0));

        _allowed[msg.sender][spender] = value;
        emit Approval(msg.sender, spender, value);
        return true;
    }

    /**
     * @dev Transfer tokens from one address to another.
     * Note that while this function emits an Approval event, this is not requ
     * and other compliant implementations may not emit the event.
     * @param from address The address which you want to send tokens from
     * @param to address The address which you want to transfer to
     * @param value uint256 the amount of tokens to be transferred
     */
    function transferFrom(address from, address to, uint256 value) public retu
        _allowed[from][msg.sender] = _allowed[from][msg.sender].sub(value);
        _transfer(from, to, value);
        emit Approval(from, msg.sender, _allowed[from][msg.sender]);
        return true;
    }

    /**
     * @dev Increase the amount of tokens that an owner allowed to a spender.
     * approve should be called when allowed_[_spender] == 0. To increment
     * allowed value is better to use this function to avoid 2 calls (and wait
     * the first transaction is mined)
     * From MonolithDAO Token.sol
     * Emits an Approval event.
     * @param spender The address which will spend the funds.
     * @param addedValue The amount of tokens to increase the allowance by.
     */
    function increaseAllowance(address spender, uint256 addedValue) public ret
        require(spender != address(0));

        _allowed[msg.sender][spender] = _allowed[msg.sender][spender].add(adde
        emit Approval(msg.sender, spender, _allowed[msg.sender][spender]);
        return true;
    }

    /**
     * @dev Decrease the amount of tokens that an owner allowed to a spender.
     * approve should be called when allowed_[_spender] == 0. To decrement
     * allowed value is better to use this function to avoid 2 calls (and wait
     * the first transaction is mined)
     * From MonolithDAO Token.sol
     * Emits an Approval event.
     * @param spender The address which will spend the funds.
     * @param subtractedValue The amount of tokens to decrease the allowance b
     */
    function decreaseAllowance(address spender, uint256 subtractedValue) publi
        require(spender != address(0));

        _allowed[msg.sender][spender] = _allowed[msg.sender][spender].sub(subt
        emit Approval(msg.sender, spender, _allowed[msg.sender][spender]);
        return true;
    }

    /**
    * @dev Transfer token for a specified addresses
    * @param from The address to transfer from.
    * @param to The address to transfer to.
    * @param value The amount to be transferred.
    */
    function _transfer(address from, address to, uint256 value) internal {
        require(to != address(0));

        _balances[from] = _balances[from].sub(value);
        _balances[to] = _balances[to].add(value);
        emit Transfer(from, to, value);
    }

    /**
     * @dev Internal function that mints an amount of the token and assigns it
     * an account. This encapsulates the modification of balances such that th
     * proper events are emitted.
     * @param account The account that will receive the created tokens.
     * @param value The amount that will be created.
     */
    function _mint(address account, uint256 value) internal {
        require(account != address(0));

        _totalSupply = _totalSupply.add(value);
        _balances[account] = _balances[account].add(value);
        emit Transfer(address(0), account, value);
    }

    /**
     * @dev Internal function that burns an amount of the token of a given
```

```
 * account.
 * @param account The account whose tokens will be burnt.
 * @param value The amount that will be burnt.
 */
function _burn(address account, uint256 value) internal {
    require(account != address(0));

    _totalSupply = _totalSupply.sub(value);
    _balances[account] = _balances[account].sub(value);
    emit Transfer(account, address(0), value);
}

/**
 * @dev Internal function that burns an amount of the token of a given
 * account, deducting from the sender's allowance for said account. Uses t
 * internal burn function.
 * Emits an Approval event (reflecting the reduced allowance).
 * @param account The account whose tokens will be burnt.
 * @param value The amount that will be burnt.
 */
function _burnFrom(address account, uint256 value) internal {
    _allowed[account][msg.sender] = _allowed[account][msg.sender].sub(valu
    _burn(account, value);
    emit Approval(account, msg.sender, _allowed[account][msg.sender]);
}

}
```

/* ** @title Roles * @dev Library for managing addresses assigned to a Role.* /
library Roles { struct Role { mapping (address => bool) bearer; }

```
/**
 * @dev give an account access to this role
 */
function add(Role storage role, address account) internal {
    require(account != address(0));
    require(!has(role, account));

    role.bearer[account] = true;
}

/**
 * @dev remove an account's access to this role
 */
function remove(Role storage role, address account) internal {
    require(account != address(0));
    require(has(role, account));

    role.bearer[account] = false;
}

/**
```

```
 * @dev check if an account has this role
 * @return bool
 */
function has(Role storage role, address account) internal view returns (bo
    require(account != address(0));
    return role.bearer[account];
}

}
```

contract MinterRole { using Roles for Roles.Role;

```
event MinterAdded(address indexed account);
event MinterRemoved(address indexed account);

Roles.Role private _minters;

constructor () internal {
    _addMinter(msg.sender);
}

modifier onlyMinter() {
    require(isMinter(msg.sender));
    _;
}

function isMinter(address account) public view returns (bool) {
    return _minters.has(account);
}

function addMinter(address account) public onlyMinter {
    _addMinter(account);
}

function renounceMinter() public {
    _removeMinter(msg.sender);
}

function _addMinter(address account) internal {
    _minters.add(account);
    emit MinterAdded(account);
}

function _removeMinter(address account) internal {
    _minters.remove(account);
    emit MinterRemoved(account);
}

}
```

/ * @title ERC20Mintable * @dev ERC20 minting logic */ contract ERC20Mintable is ERC20, MinterRole { / *** @dev Function to mint tokens * @param to The address that will receive the minted tokens. * @param value The amount of tokens to mint. * @return A boolean that indicates if the operation was successful. / *function mint(address to, uint256 value) public onlyMinter returns (bool) { _mint(to, value); return true; } } / * @title Crowdsale * @dev Crowdsale is a base contract for managing a token crowdsale. * Crowdsales have a start and end block, where investors can make * token purchases and the crowdsale will assign them tokens based * on a token per ETH rate. Funds collected are forwarded to a wallet * as they arrive.* / contract Crowdsale { using SafeMath for uint256;

```
// The token being sold
ERC20Mintable public token;

// start and end block where investments are allowed (both inclusive)
uint256 public startBlock;
uint256 public endBlock;

// address where funds are collected
address public wallet;

// how many token units a buyer gets per wei
uint256 public rate;

// amount of raised money in wei
uint256 public weiRaised;

/**
* event for token purchase logging
* @param purchaser who paid for the tokens
* @param beneficiary who got the tokens
* @param value weis paid for purchase
* @param amount amount of tokens purchased
*/
event TokenPurchase(address indexed purchaser, address indexed beneficiary

function Crowdsale(uint256 _startBlock, uint256 _endBlock, uint256 _rate,
    require(_startBlock >= block.number);
    require(_endBlock >= _startBlock);
    require(_rate > 0);
    require(_wallet != 0x0);

    token = createTokenContract();
    startBlock = _startBlock;
    endBlock = _endBlock;
    rate = _rate;
    wallet = _wallet;
}

// creates the token to be sold.
// override this method to have crowdsale of a specific mintable token.
function createTokenContract() internal returns (ERC20Mintable) {
    return new ERC20Mintable();
}


// fallback function can be used to buy tokens
function () payable {
    buyTokens(msg.sender);
}

// low level token purchase function
function buyTokens(address beneficiary) payable {
    require(beneficiary != 0x0);
    require(validPurchase());

    uint256 weiAmount = msg.value;

    // calculate token amount to be created
    uint256 tokens = weiAmount.mul(rate);

    // update state
    weiRaised = weiRaised.add(weiAmount);

    token.mint(beneficiary, tokens);
    TokenPurchase(msg.sender, beneficiary, weiAmount, tokens);

    forwardFunds();
}

// send ether to the fund collection wallet
// override to create custom fund forwarding mechanisms
function forwardFunds() internal {
    wallet.transfer(msg.value);
}

// @return true if the transaction can buy tokens
function validPurchase() internal constant returns (bool) {
    uint256 current = block.number;
    bool withinPeriod = current >= startBlock && current <= endBlock;
    bool nonZeroPurchase = msg.value != 0;
    return withinPeriod && nonZeroPurchase;
}

// @return true if crowdsale event has ended
function hasEnded() public constant returns (bool) {
    return block.number > endBlock;
}

}
```

/* ** @title CappedCrowdsale * @dev Extension of Crowsdale with a max amount of funds raised* / contract CappedCrowdsale is Crowdsale { using SafeMath for uint256; uint256 public cap;

```
function CappedCrowdsale(uint256 _cap) {
    require(_cap > 0);
    cap = _cap;
}

// overriding Crowdsale#validPurchase to add extra cap logic
// @return true if investors can buy at the moment
function validPurchase() internal constant returns (bool) {
    bool withinCap = weiRaised.add(msg.value) <= cap;
    return super.validPurchase() && withinCap;
}

// overriding Crowdsale#hasEnded to add cap logic
// @return true if crowdsale event has ended
function hasEnded() public constant returns (bool) {
    bool capReached = weiRaised >= cap;
    return super.hasEnded() || capReached;
}

}
```

/* * *@title WhitelistedCrowdsale * @dev Extension of Crowsdale with a whitelist of investors that * can buy before the start block* / contract WhitelistedCrowdsale is Crowdsale { using SafeMath for uint256;

```
mapping (address => bool) public whitelist;

function addToWhitelist(address addr) {
    require(msg.sender != address(this));
    whitelist[addr] = true;
}

// overriding Crowdsale#validPurchase to add extra whitelit logic
// @return true if investors can buy at the moment
function validPurchase() internal constant returns (bool) {
    return super.validPurchase() || (whitelist[msg.sender] && !hasEnded())
}

}
```

contract MDTCrowdsale is CappedCrowdsale, WhitelistedCrowdsale {

```
function MDTCrowdsale()
CappedCrowdsale(50000000000000000000000)
Crowdsale(block.number, block.number + 100000, 1, msg.sender) { // Wallet
    addToWhitelist(msg.sender);
    addToWhitelist(0x0d5bda9db5dd36278c6a40683960ba58cac0149b);
    addToWhitelist(0x1b6ddc637c24305b354d7c337f9126f68aad4886);
}

}
```

\`\`\`

==
File: SWC-126.md
==

Title

Insufficient Gas Griefing

Relationships

- CWE-691: Insufficient Control Flow Management
- EEA EthTrust Security Levels:
- **[Q] Manage Gas Usage Increases**
- **[Q] Protect Gas Usage**

Description

Insufficient gas griefing attacks can be performed on contracts which accept data and use it in a sub-call on another contract. If the sub-call fails, either the whole transaction is reverted, or execution is continued. In the case of a relayer contract, the user who executes the transaction, the 'forwarder', can effectively censor transactions by using just enough gas to execute the transaction, but not enough for the sub-call to succeed.

Remediation

There are two options to prevent insufficient gas griefing:

- Only allow trusted users to relay transactions.
- Require that the forwarder provides enough gas.

References

- Consensys Smart Contract Best Practices
- What does griefing mean?
- Griefing Attacks: Are they profitable for the attacker?

Samples

relayer.sol

```solidity / * *@source: https://consensys.github.io/smart-contract-best-practices/known_attacks/#insufficient-gas-griefing * @author: ConsenSys Diligence * Modified by Kaden Zipfel /*

pragma solidity ^0.5.0;
```

contract Relayer { uint transactionId;

```
struct Tx {
    bytes data;
    bool executed;
}

mapping (uint => Tx) transactions;

function relay(Target target, bytes memory _data) public returns(bool) {
    // replay protection; do not call the same transaction twice
    require(transactions[transactionId].executed == false, 'same transacti
    transactions[transactionId].data = _data;
    transactions[transactionId].executed = true;
    transactionId += 1;

    (bool success, ) = address(target).call(abi.encodeWithSignature("execu
    return success;
}
```

}

// Contract called by Relayer contract Target { function execute(bytes memory _data) public { // Execute contract code } }

~~~

relayer_fixed.sol

```solidity / * *@source: https://consensys.github.io/smart-contract-best-practices/known_attacks/#insufficient-gas-griefing * @author: ConsenSys Diligence * Modified by Kaden Zipfel* /

pragma solidity ^0.5.0;

contract Relayer { uint transactionId;

~~~
struct Tx {
    bytes data;
    bool executed;
}

mapping (uint => Tx) transactions;

function relay(Target target, bytes memory _data, uint _gasLimit) public {
    // replay protection; do not call the same transaction twice
    require(transactions[transactionId].executed == false, 'same transacti
    transactions[transactionId].data = _data;
    transactions[transactionId].executed = true;
    transactionId += 1;

    address(target).call(abi.encodeWithSignature("execute(bytes)", _data,
}
```
```

}

// Contract called by Relayer contract Target { function execute(bytes memory _data, uint _gasLimit) public { require(gasleft() >= _gasLimit, 'not enough gas'); // Execute contract code } }

```

================================================================
File: SWC-127.md
================================================================

# Title

Arbitrary Jump with Function Type Variable

## Relationships

- CWE-695: Use of Low-Level Functionality
- EEA EthTrust Security Levels:
- **[S] No `assembly {}`**
- **[M] Avoid Common `assembly {}` Attack Vectors**

## Description

Solidity supports function types. That is, a variable of function type can be assigned with a reference to a function with a matching signature. The function saved to such variable can be called just like a regular function.

The problem arises when a user has the ability to arbitrarily change the function type variable and thus execute random code instructions. As Solidity doesn't support pointer arithmetics, it's impossible to change such variable to an arbitrary value. However, if the developer uses assembly instructions, such as `mstore` or assign operator, in the worst case scenario an attacker is able to point a function type variable to any code instruction, violating required validations and required state changes.

## Remediation

The use of assembly should be minimal. A developer should not allow a user to assign arbitrary values to function type variables.

## References

- Solidity CTF
- Solidity docs - Solidity Assembly
- Solidity docs - Function Types
```

Samples

FunctionTypes.sol

```solidity / * *@source: https://gist.github.com/wadeAlexC/ 7a18de852693b3f890560ab6a211a2b8 * @author: Alexander Wade /*

pragma solidity ^0.4.25;

contract FunctionTypes {

```
constructor() public payable { require(msg.value != 0); }

function withdraw() private {
    require(msg.value == 0, 'dont send funds!');
    address(msg.sender).transfer(address(this).balance);
}

function frwd() internal
    { withdraw(); }

struct Func { function () internal f; }

function breakIt() public payable {
    require(msg.value != 0, 'send funds!');
    Func memory func;
    func.f = frwd;
    assembly { mstore(func, add(mload(func), callvalue)) }
    func.f();
}
```

}

```

==
File: SWC-128.md
==

Title

DoS With Block Gas Limit

Relationships

- CWE-400: Uncontrolled Resource Consumption
- EEA EthTrust Security Levels:
- **[Q] Manage Gas Usage Increases**

Description

When smart contracts are deployed or functions inside them are called, the execution of these actions always requires a certain amount of gas, based of how much computation is needed to complete them. The Ethereum network specifies a block gas limit and the sum of all transactions included in a block can not exceed the threshold.

Programming patterns that are harmless in centralized applications can lead to Denial of Service conditions in smart contracts when the cost of executing a function exceeds the block gas limit. Modifying an array of unknown size, that increases in size over time, can lead to such a Denial of Service condition.

Remediation

Caution is advised when you expect to have large arrays that grow over time. Actions that require looping across the entire data structure should be avoided.

If you absolutely must loop over an array of unknown size, then you should plan for it to potentially take multiple blocks, and therefore require multiple transactions.

References

- Ethereum Design Rationale
- Ethereum Yellow Paper
- Clear Large Array Without Blowing Gas Limit
- GovernMental jackpot payout DoS Gas

Samples

dos_address.sol

```` ```solidity pragma solidity ^0.4.25; ````

contract DosGas {

```
address[] creditorAddresses;
bool win = false;

function emptyCreditors() public {
    if(creditorAddresses.length>1500) {
        creditorAddresses = new address[](0);
        win = true;
    }
}

function addCreditors() public returns (bool) {
```
````

~~~
    for(uint i=0;i<350;i++) {
      creditorAddresses.push(msg.sender);
    }
    return true;
}

function iWin() public view returns (bool) {
    return win;
}

function numberCreditors() public view returns (uint) {
    return creditorAddresses.length;
}

}
~~~

```

## dos_number.sol

```solidity pragma solidity ^0.4.25;

contract DosNumber {

~~~
uint numElements = 0;
uint[] array;

function insertNnumbers(uint value,uint numbers) public {

    // Gas DOS if number > 382 more or less, it depends on actual gas limi
    for(uint i=0;i<numbers;i++) {
        if(numElements == array.length) {
            array.length += 1;
        }
        array[numElements++] = value;
    }
}

function clear() public {
    require(numElements>1500);
    numElements = 0;
}

// Gas DOS clear
function clearDOS() public {

    // number depends on actual gas limit
    require(numElements>1500);
    array = new uint[](0);
    numElements = 0;
}
~~~

~~~
function getLengthArray() public view returns(uint) {
    return numElements;
}

function getRealLengthArray() public view returns(uint) {
    return array.length;
}

}
~~~

```

dos_simple.sol

```solidity pragma solidity ^0.4.25;

contract DosOneFunc {

~~~
address[] listAddresses;

function ifillArray() public returns (bool){
    if(listAddresses.length<1500) {

        for(uint i=0;i<350;i++) {
            listAddresses.push(msg.sender);
        }
        return true;

    } else {
        listAddresses = new address[](0);
        return false;
    }
}

}
~~~

```

==

File: SWC-129.md

==

Title

Typographical Error

Relationships

- CWE-480: Use of Incorrect Operator
- EEA EthTrust Security Levels:
- **[Q] Implement as Documented**
```

Description

A typographical error can occur for example when the intent of a defined operation is to sum a number to a variable (+=) but it has accidentally been defined in a wrong way (=+), introducing a typo which happens to be a valid operator. Instead of calculating the sum it initializes the variable again.

The unary + operator is deprecated in new solidity compiler versions.

Remediation

The weakness can be avoided by performing pre-condition checks on any math operation or using a vetted library for arithmetic calculations such as SafeMath developed by OpenZeppelin.

References

- HackerGold Bug Analysis
- SafeMath by OpenZeppelin
- Disallow Unary plus

Samples

typo_one_command.sol

```solidity pragma solidity ^0.4.25;

contract TypoOneCommand { uint numberOne = 1;

~~~
function alwaysOne() public {
    numberOne =+ 1;
}
~~~

}

```

typo_safe_math.sol

```solidity pragma solidity ^0.4.25;

_/* Taken from the OpenZeppelin github * @title SafeMath * @dev Math operations with safety checks that revert on error_ / library SafeMath {

_/* * @dev Multiplies two numbers, reverts on overflow._ / function mul(uint256 a, uint256 b) internal pure returns (uint256) { // Gas optimization: this is cheaper than requiring 'a' not being zero, but the // benefit is lost if 'b' is also tested. // See: https://github.com/OpenZeppelin/openzeppelin-solidity/pull/522 if (a == 0) { return 0; }
```

```
uint256 c = a * b;
require(c / a == b);

return c;

}
```

/* * *@dev Integer division of two numbers truncating the quotient, reverts on division by zero.* / function div(uint256 a, uint256 b) internal pure returns (uint256) { require(b > 0); // Solidity only automatically asserts when dividing by 0 uint256 c = a / b; // assert(a == b * c + a % b); // There is no case in which this doesn't hold

```
return c;

}
```

/* * *@dev Subtracts two numbers, reverts on overflow (i.e. if subtrahend is greater than minuend).* / function sub(uint256 a, uint256 b) internal pure returns (uint256) { require(b <= a); uint256 c = a - b;

```
return c;

}
```

/* * *@dev Adds two numbers, reverts on overflow.* / function add(uint256 a, uint256 b) internal pure returns (uint256) { uint256 c = a + b; require(c >= a);

```
return c;

}
```

/* * *@dev Divides two numbers and returns the remainder (unsigned integer modulo), * reverts when dividing by zero.* / function mod(uint256 a, uint256 b) internal pure returns (uint256) { require(b != 0); return a % b; } }

contract TypoSafeMath {

```
using SafeMath for uint256;
uint256 public numberOne = 1;
bool public win = false;

function addOne() public {
    numberOne =+ 1;
}

function addOneCorrect() public {
    numberOne += 1;
}

function addOneSafeMath() public  {
    numberOne = numberOne.add(1);
}
```

```
function iWin() public {
    if(!win && numberOne>3) {
        win = true;
    }
}

}
```

\`\`\`

typo_simple.sol

\`\`\`solidity pragma solidity ^0.4.25;

contract TypoSimple {

```
uint onlyOne = 1;
bool win = false;

function addOne() public {
    onlyOne =+ 1;
    if(onlyOne>1) {
        win = true;
    }
}

function iWin() view public returns (bool) {
    return win;
}

}
```

\`\`\`

```
=================================================================
```
File: SWC-130.md
```
=================================================================
```

Right-To-Left-Override control character (U+202E)

Relationships

- CWE-451: User Interface (UI) Misrepresentation of Critical Information
- EEA EthTrust Security Levels specification:
- **[S] No Unicode Direction Control Characters**
- **[M] No Unnecessary Unicode Controls**
- **[M] No Homoglyph-style Attack**

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.

Remediation

There are very few legitimate uses of the U+202E character. It should not appear in the source code of a smart contract.

References

- Outsmarting Smart Contracts

Samples

guess_the_number.sol

```` ``` ````solidity / * *@source: https://youtu.be/P_Mtd5Fc_3E* * *@author: Shahar Zini* / pragma solidity ^0.5.0;

contract GuessTheNumber { uint _secretNumber; address payable _owner; event success(string); event wrongNumber(string);

```
constructor(uint secretNumber) payable public
{
    require(secretNumber <= 10);
    _secretNumber = secretNumber;
    _owner = msg.sender;
}

function getValue() view public returns (uint)
{
    return address(this).balance;
}

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

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

function checkAndTransferPrize(uint p, uint n, address payable guesser) in
{
    if(n == _secretNumber)
```
````

````
    {
        guesser.transfer(p);
        emit success("You guessed the correct number!");
    }
    else
    {
        emit wrongNumber("You've made an incorrect guess!");
    }
}

function kill() public
{
    require(msg.sender == _owner);
    selfdestruct(_owner);
}

} ```
````

Comments

The line

```
checkAndTransferPrize(/*The prize*/p , n/*guessed number*/
```

inside the function `guess(uint n)` uses invisible direction control characters, so what is present on the screen misrepresents the order of the parameters - the function is called with parameters `n, p, address`, which is the **logical order** of characters, but some are displayed from right to left, so that the segment 'n, p' appears in reverse order to a reader, because invisible direction control characters are included in the code.

Selecting the text character by character will usually show this - the selection suddenly jumps to the end of the right-to-left text, and starts to extend from the right hand side leftward. It is also possible to check for the unicode characters explicitly in the content.

==
File: SWC-131.md
==

Presence of unused variables

Relationships

- [CWE-1164: Irrelevant Code]
- EthTrust Security Levels:
- **[Q] Code Linting**

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

Remediation

Remove all unused variables from the code base.

References

- Unused local variables warnings discussion
- Shadowing of inherited state variables discussion

Samples

unused_state_variables.sol

```solidity 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();

}
~~~

} ```

### unused_state_variables_fixed.sol

```solidity pragma solidity >=0.5.0; pragma experimental ABIEncoderV2;

import "./base_fixed.sol";

contract DerivedA is Base {

```
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();

}
```

} ```

unused_variables.sol

```solidity pragma solidity ^0.5.0;

contract UnusedVariables { int a = 1;

```
// y is not used
function unusedArg(int x, int y) public view returns (int z) {
    z = x + a;
}

// n is not reported it is part of another SWC category
function unusedReturn(int x, int y) public pure returns (int m, int n, int
    m = y - x;
    o = m/2;
}

// x is not accessed
function neverAccessed(int test) public pure returns (int) {
    int z = 10;

    if (test > z) {
        // x is not used
        int x = test - z;

        return test - z;
    }

    return z;
}
```
```

~~~
function tupleAssignment(int p) public returns (int q, int r){
    (q, , r) = unusedReturn(p,2);

}

}
~~~

```

## unused_variables_fixed.sol

```solidity pragma solidity ^0.5.0;

contract UnusedVariables { int a = 1;

~~~
function unusedArg(int x) public view returns (int z) {
    z = x + a;
}

// n is not reported it is part of another SWC category
function unusedReturn(int x, int y) public pure returns (int m, int n,int
    m = y - x;
    o = m/2;
}

// x is not accessed
function neverAccessed(int test) public pure returns (int) {
    int z = 10;

    if (test > z) {
        return test - z;
    }

    return z;
}

function tupleAssignment(int p) public returns (int q, int r){
    (q, , r) = unusedReturn(p,2);

}

}
~~~

```

~~~
==================================================================
~~~
File: SWC-132.md
~~~
==================================================================
~~~

Unexpected Ether balance
```

Relationships

- CWE-667: Improper Locking
- EEA EthTrust Security Levels:
- **[S] No Exact Balance Check**
- **[M] Verify Exact Balance Checks**

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.

Remediation

Avoid strict equality checks for the Ether balance in a contract.

References

- Consensys Best Practices: Force Feeding
- Sigmaprime: Unexpected Ether
- Gridlock (a smart contract bug)

Samples

Lockdrop.sol

```solidity /* * *@source: https://github.com/hicommonwealth/edgeware-lockdrop/blob/93ecb524c9c88d25bab36278541f190fa9e910c2/contracts/Lockdrop.sol* /

pragma solidity ^0.5.0;

contract Lock { // address owner; slot #0 // address unlockTime; slot #1 constructor (address owner, uint256 unlockTime) public payable { assembly { sstore(0x00, owner) sstore(0x01, unlockTime) } }

```
/**
 * @dev   Withdraw function once timestamp has passed unlock time
 */
function () external payable { // payable so solidity doesn't add unnecess
    assembly {
        switch gt(timestamp, sload(0x01))
        case 0 { revert(0, 0) }
        case 1 {
            switch call(gas, sload(0x00), balance(address), 0, 0, 0, 0)
            case 0 { revert(0, 0) }
```
```

```
        }
    }
}

}
```

contract Lockdrop { enum Term { ThreeMo, SixMo, TwelveMo } // Time constants uint256 constant public LOCK_DROP_PERIOD = 1 days * 92; // 3 months uint256 public LOCK_START_TIME; uint256 public LOCK_END_TIME; // ETH locking events event Locked(address indexed owner, uint256 eth, Lock lockAddr, Term term, bytes edgewareAddr, bool isValidator, uint time); event Signaled(address indexed contractAddr, bytes edgewareAddr, uint time);

```
constructor(uint startTime) public {
    LOCK_START_TIME = startTime;
    LOCK_END_TIME = startTime + LOCK_DROP_PERIOD;
}

/**
 * @dev        Locks up the value sent to contract in a new Lock
 * @param      term         The length of the lock up
 * @param      edgewareAddr The bytes representation of the target edgewar
 * @param      isValidator  Indicates if sender wishes to be a validator
 */
function lock(Term term, bytes calldata edgewareAddr, bool isValidator)
    external
    payable
    didStart
    didNotEnd
{
    uint256 eth = msg.value;
    address owner = msg.sender;
    uint256 unlockTime = unlockTimeForTerm(term);
    // Create ETH lock contract
    Lock lockAddr = (new Lock).value(eth)(owner, unlockTime);
    // ensure lock contract has all ETH, or fail
    assert(address(lockAddr).balance == msg.value);
    emit Locked(owner, eth, lockAddr, term, edgewareAddr, isValidator, now
}

/**
 * @dev        Signals a contract's (or address's) balance decided after l
 * @param      contractAddr  The contract address from which to signal the
 * @param      nonce         The transaction nonce of the creator of the c
 * @param      edgewareAddr   The bytes representation of the target edgew
 */
function signal(address contractAddr, uint32 nonce, bytes calldata edgewar
    external
    didStart
    didNotEnd
    didCreate(contractAddr, msg.sender, nonce)
{
    emit Signaled(contractAddr, edgewareAddr, now);
}

function unlockTimeForTerm(Term term) internal view returns (uint256) {
    if (term == Term.ThreeMo) return now + 92 days;
    if (term == Term.SixMo) return now + 183 days;
    if (term == Term.TwelveMo) return now + 365 days;

    revert();
}

/**
 * @dev        Ensures the lockdrop has started
 */
modifier didStart() {
    require(now >= LOCK_START_TIME);
    _;
}

/**
 * @dev        Ensures the lockdrop has not ended
 */
modifier didNotEnd() {
    require(now <= LOCK_END_TIME);
    _;
}

/**
 * @dev        Rebuilds the contract address from a normal address and tra
 * @param      _origin  The non-contract address derived from a user's pub
 * @param      _nonce   The transaction nonce from which to generate a con
 */
function addressFrom(address _origin, uint32 _nonce) public pure returns (
    if(_nonce == 0x00)     return address(uint160(uint256(keccak256(abi.en
    if(_nonce <= 0x7f)     return address(uint160(uint256(keccak256(abi.en
    if(_nonce <= 0xff)     return address(uint160(uint256(keccak256(abi.en
    if(_nonce <= 0xffff)   return address(uint160(uint256(keccak256(abi.en
    if(_nonce <= 0xffffff) return address(uint160(uint256(keccak256(abi.en
    return address(uint160(uint256(keccak256(abi.encodePacked(byte(0xda),
}

/**
 * @dev        Ensures the target address was created by a parent at some
 * @param      target  The target contract address (or trivially the paren
 * @param      parent  The creator of the alleged contract address
 * @param      nonce   The creator's tx nonce at the time of the contract
 */
modifier didCreate(address target, address parent, uint32 nonce) {
    // Trivially let senders "create" themselves
    if (target == parent) {
        _;
    } else {
        require(target == addressFrom(parent, nonce));
        _;
    }
}
```

} ```

==
File: SWC-133.md
==

Hash Collisions With Multiple Variable Length Arguments

Relationships

CWE-294: Authentication Bypass by Capture-replay

Description

Using `abi.encodePacked()` with multiple variable length arguments can, in certain situations, lead to a hash collision. Since `abi.encodePacked()` packs all elements in order regardless of whether they're part of an array, you can move elements between arrays and, so long as all elements are in the same order, it will return the same encoding. In a signature verification situation, an attacker could exploit this by modifying the position of elements in a previous function call to effectively bypass authorization.

Remediation

When using `abi.encodePacked()`, it's crucial to ensure that a matching signature cannot be achieved using different parameters. To do so, either do not allow users access to parameters used in `abi.encodePacked()`, or use fixed length arrays. Alternatively, you can simply use `abi.encode()` instead.

It is also recommended that you use replay protection (see SWC-121), although an attacker can still bypass this by front-running.

References

- Solidity Non-standard Packed Mode
- Hash Collision Attack

Samples

access_control.sol

```solidity / * *@author: Steve Marx* /
```

pragma solidity ^0.5.0;

import "./ECDSA.sol";

contract AccessControl { using ECDSA for bytes32; mapping(address => bool) isAdmin; mapping(address => bool) isRegularUser; // Add admins and regular users. function addUsers(address[] calldata admins, address[] calldata regularUsers, bytes calldata signature) external { if (!isAdmin[msg.sender]) { // Allow calls to be relayed with an admin's signature. bytes32 hash = keccak256(abi.encodePacked(admins, regularUsers)); address signer = hash.toEthSignedMessageHash().recover(signature); require(isAdmin[signer], "Only admins can add users."); } for (uint256 i = 0; i < admins.length; i++) { isAdmin[admins[i]] = true; } for (uint256 i = 0; i < regularUsers.length; i++) { isRegularUser[regularUsers[i]] = true; } } } ```

access_control_fixed_1.sol

```` ```solidity / * *@author: Steve Marx * Modified by Kaden Zipfel* / ````

pragma solidity ^0.5.0;

import "./ECDSA.sol";

contract AccessControl { using ECDSA for bytes32; mapping(address => bool) isAdmin; mapping(address => bool) isRegularUser; // Add a single user, either an admin or regular user. function addUser(address user, bool admin, bytes calldata signature) external { if (!isAdmin[msg.sender]) { // Allow calls to be relayed with an admin's signature. bytes32 hash = keccak256(abi.encodePacked(user)); address signer = hash.toEthSignedMessageHash().recover(signature); require(isAdmin[signer], "Only admins can add users."); } if (admin) { isAdmin[user] = true; } else { isRegularUser[user] = true; } } } ```

access_control_fixed_2.sol

```` ```solidity / * *@author: Steve Marx * Modified by Kaden Zipfel* / ````

pragma solidity ^0.5.0;

import "./ECDSA.sol";

contract AccessControl { using ECDSA for bytes32; mapping(address => bool) isAdmin; mapping(address => bool) isRegularUser; // Add admins and regular users. function addUsers(// Use fixed length arrays. address[3] calldata admins, address[3] calldata regularUsers, bytes calldata signature) external { if (!isAdmin[msg.sender]) { // Allow calls to be relayed with an admin's signature. bytes32 hash = keccak256(abi.encodePacked(admins, regularUsers)); address signer = hash.toEthSignedMessageHash().recover(signature); require(isAdmin[signer], "Only admins can add users."); } for (uint256 i = 0;

i < admins.length; i++) { isAdmin[admins[i]] = true; } for (uint256 i = 0; i < regularUsers.length; i++) { isRegularUser[regularUsers[i]] = true; } } } ```

==
File: SWC-134.md
==

Message call with hardcoded gas amount

Relationships

CWE-655: Improper Initialization

Description

The `transfer()` and `send()` functions forward a fixed amount of 2300 gas. Historically, it has often been recommended to use these functions for value transfers to guard against reentrancy attacks. However, the gas cost of EVM instructions may change significantly during hard forks which may break already deployed contract systems that make fixed assumptions about gas costs. For example. EIP 1884 broke several existing smart contracts due to a cost increase of the SLOAD instruction.

Remediation

Avoid the use of `transfer()` and `send()` and do not otherwise specify a fixed amount of gas when performing calls. Use `.call.value(...)("")` instead. Use the checks-effects-interactions pattern and/or reentrancy locks to prevent reentrancy attacks.

References

- ChainSecurity - Ethereum Istanbul Hardfork: The Security Perspective
- Steve Marx - Stop Using Solidity's transfer() Now
- EIP 1884

Samples

hardcoded_gas_limits.sol

```solidity / * *@author: Bernhard Mueller (ConsenSys / MythX)* /

pragma solidity 0.6.4;

interface ICallable { function callMe() external; }

contract HardcodedNotGood {
```

```
address payable _callable = 0xaAaAaAaaAaAaAaaAaAAAAAAAAaaaAaAaAaaAaaAa;
ICallable callable = ICallable(_callable);

constructor() public payable {
}

function doTransfer(uint256 amount) public {
    _callable.transfer(amount);
}

function doSend(uint256 amount) public {
    _callable.send(amount);
}

 function callLowLevel() public {
     _callable.call.value(0).gas(10000)("");
 }

 function callWithArgs() public {
     callable.callMe{gas: 10000}();
 }

}
```

==

File: SWC-135.md

==

Code With No Effects

Relationships

- CWE-710: Improper Adherence to Coding Standards
- EthTrust Security Levels:
- **[Q] Code Linting**

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 perform an intended action.

For example, it's easy to miss the trailing parentheses in `msg.sender.call.value(address(this).balance)("");`, which could lead to a function proceeding without transferring funds to `msg.sender`. Although, this should be avoided by checking the return value of the call.

Remediation

It's important to carefully ensure that your contract works as intended. Write unit tests to verify correct behaviour of the code.

References

- Issue on Solidity's Github - raise an error when a statement can never have side-effects
- Issue on Solidity's Github - msg.sender.call.value(address(this).balance); should produce a warning

Samples

deposit_box.sol

```solidity 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;
}
~~~

} ```

deposit_box_fixed.sol

```solidity 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;
}
~~~

} ```

### wallet.sol

```solidity / * *@author: Kaden Zipfel* /

pragma solidity ^0.5.0;
~~~

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

```
// Deposit funds in contract
function deposit(uint amount) public payable {
    require(msg.value == amount, 'msg.value must be equal to amount');
    balance[msg.sender] = amount;
}

// Withdraw funds from contract
function withdraw(uint amount) public {
    require(amount <= balance[msg.sender], 'amount must be less than balan

    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);
}
```

} ```

wallet_fixed.sol

```solidity / * *@author: Kaden Zipfel* /

pragma solidity ^0.5.0;

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

```
// Deposit funds in contract
function deposit(uint amount) public payable {
    require(msg.value == amount, 'msg.value must be equal to amount');
    balance[msg.sender] = amount;
}

// Withdraw funds from contract
function withdraw(uint amount) public {
    require(amount <= balance[msg.sender], 'amount must be less than balan

    uint previousBalance = balance[msg.sender];
    balance[msg.sender] = previousBalance - amount;

    // Attempt to send amount from the contract to msg.sender
    (bool success,) = msg.sender.call.value(amount)("");
    require(success, 'transfer failed');
}
```

} ```

================================================
File: SWC-136.md
================================================
```

Title

Unencrypted Private Data On-Chain

Relationships

- CWE-767: Access to Critical Private Variable via Public Method
- EthTrust Security Levels:
- **[Q] No Private Data**
- **[Q] Enforce Least Privilege**

Description

It is a common misconception that `private` type variables cannot be read. Even if your contract is not published, attackers can look at contract transactions to determine values stored in the state of the contract. For this reason, it's important that unencrypted private data is not stored in the contract code or state.

Remediation

Any private data should either be stored off-chain, or carefully encrypted.

References

- Keeping secrets on Ethereum
- A Survey of Attacks on Ethereum Smart Contracts (SoK)
- Unencrypted Secrets
- Stack Overflow - Decrypt message on-chain

Samples

odd_even.sol

```solidity / *@source: https://gist.github.com/manojpramesh/336882804402bee8d6b99bea453caadd#file-odd-even-sol * @author: https://github.com/manojpramesh * Modified by Kaden Zipfel /*

pragma solidity ^0.5.0;

contract OddEven { struct Player { address addr; uint number; } 

```
Player[2] private players;
uint count = 0;

function play(uint number) public payable {
        require(msg.value == 1 ether, 'msg.value must be 1 eth');
        players[count] = Player(msg.sender, number);
```
```

```
        count++;
        if (count == 2) selectWinner();
}

function selectWinner() private {
        uint n = players[0].number + players[1].number;
        (bool success, ) = players[n%2].addr.call.value(address(this).bala
        require(success, 'transfer failed');
        delete players;
        count = 0;
}
```

} ```

Comments

The vulnerable version above requires the players to send the number they are using as part of the transaction. This means the first player's number will be visible, allowing the second player to select a number that they know will make them a winner. (This assumption is simplistic to illustrate - there are also possibilities to front-run players, among other potential issues).

In the fixed version below, the players instead submit a commitment that obfuscates their number, and only subsequently reveal that they know the secret to set in train the process of a payout.

odd_even_fixed.sol

```solidity / * *@source: https://github.com/yahgwai/rps * @author: Chris Buckland * Modified by Kaden Zipfel * Modified by Kacper Żuk* /

pragma solidity ^0.5.0;

contract OddEven { enum Stage { FirstCommit, SecondCommit, FirstReveal, SecondReveal, Distribution }

```
struct Player {
    address addr;
    bytes32 commitment;
    bool revealed;
    uint number;
}

Player[2] private players;
Stage public stage = Stage.FirstCommit;

function play(bytes32 commitment) public payable {
    // Only run during commit stages
    uint playerIndex;
    if(stage == Stage.FirstCommit) playerIndex = 0;
    else if(stage == Stage.SecondCommit) playerIndex = 1;
    else revert("only two players allowed");
```
```

```
        // Require proper amount deposited
        // 1 ETH as a bet + 1 ETH as a bond
        require(msg.value == 2 ether, 'msg.value must be 2 eth');

        // Store the commitment
        players[playerIndex] = Player(msg.sender, commitment, false, 0);

        // Move to next stage
        if(stage == Stage.FirstCommit) stage = Stage.SecondCommit;
        else stage = Stage.FirstReveal;
    }

    function reveal(uint number, bytes32 blindingFactor) public {
        // Only run during reveal stages
        require(stage == Stage.FirstReveal || stage == Stage.SecondReveal, "wr

        // Find the player index
        uint playerIndex;
        if(players[0].addr == msg.sender) playerIndex = 0;
        else if(players[1].addr == msg.sender) playerIndex = 1;
        else revert("unknown player");

        // Protect against double-reveal, which would trigger move to Stage.Di
        require(!players[playerIndex].revealed, "already revealed");

        // Check the hash to prove the player's honesty
        require(keccak256(abi.encodePacked(msg.sender, number, blindingFactor)

        // Update player number if correct
        players[playerIndex].number = number;

        // Protect against double-reveal
        players[playerIndex].revealed = true;

        // Move to next stage
        if(stage == Stage.FirstReveal) stage = Stage.SecondReveal;
        else stage = Stage.Distribution;
    }

    function distribute() public {
        // Only run during distribution stage
        require(stage == Stage.Distribution, "wrong stage");

        // Find winner
        uint n = players[0].number + players[1].number;

        // Payout winners winnings and bond
        players[n%2].addr.call.value(3 ether)("");

        // Payback losers bond
        players[(n+1)%2].addr.call.value(1 ether)("");
```

~~~
    // Reset the state
    delete players;
    stage = Stage.FirstCommit;
}

}

```
~~~