ADAMoracle (Mining)

Smart Contract Audit

ADAMoracle (Mining) Audit Summary

Project name : ADAMoracle (Mining) Contract

Project address: None

Code URL : https://github.com/lztll/Adam-mining-core

Commit : 258029c1745a6aeb6ba9881753040f5c8774cf53

Project target : ADAMoracle (Mining) Contract Audit

Blockchain : OKExChain

Test result : PASSED

Audit Info

Audit NO : 0X202105110006

Audit Team : Armors Labs

Audit Proofreading: https://armors.io/#project-cases

ADAMoracle (Mining) Audit

The ADAMoracle team asked us to review and audit their ADAMoracle (Mining) contract. We looked at the code and now publish our results.

Here is our assessment and recommendations, in order of importance.

Document information

Name	Auditor	Version	Date
ADAMoracle (Mining) Audit	Rock, Sophia, Rushairer, Rico, David, Alice	1.0.0	2021-05-11

Audit results

Note: It's just a mining contract.

Note that as of the date of publishing, the above review reflects the current understanding of known security patterns as they relate to the ADAMoracle (Mining) contract. The above should not be construed as investment advice.

Based on the widely recognized security status of the current underlying blockchain and smart contract, this audit report is valid for 3 months from the date of output.

(Statement: Armors Labs reports only on facts that have occurred or existed before this report is issued and assumes corresponding responsibilities. Armors Labs is not able to determine the security of its smart contracts and is not responsible for any subsequent or existing facts after this report is issued. The security audit analysis and other content of this report are only based on the documents and information provided by the information provider to Armors Labs at the time of issuance of this report (" information provided " for short). Armors Labs postulates that the

information provided is not missing, tampered, deleted or hidden. If the information provided is missing, tampered, deleted, hidden or reflected in a way that is not consistent with the actual situation, Armors Labs shall not be responsible for the losses and adverse effects caused.)

Audited target file

file	md5
./AdamMasterChef.sol	f1b8a8f1c885f785789354192060ec4d
./AdamRewardChef.sol	51782aba2ddf722c147df5ba58116f68

Vulnerability analysis

Vulnerability distribution

vulnerability level	number
Critical severity	0
High severity	0
Medium severity	0
Low severity	0

Summary of audit results

Vulnerability	status
Re-Entrancy	safe
Arithmetic Over/Under Flows	safe
Unexpected Blockchain Currency	safe
Delegatecall	safe
Default Visibilities	safe
Entropy Illusion	safe
External Contract Referencing	safe
Short Address/Parameter Attack	safe
Unchecked CALL Return Values	safe
Race Conditions / Front Running	safe
Denial Of Service (DOS)	safe
Block Timestamp Manipulation	safe
Constructors with Care	safe
Unintialised Storage Pointers	safe

Vulnerability	status
Floating Points and Numerical Precision	safe
tx.origin Authentication	safe
Permission restrictions	safe

Contract file

```
pragma solidity 0.6.12;

import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import "@openzeppelin/contracts/token/ERC20/SafeERC20.sol";
import "@openzeppelin/contracts/math/SafeMath.sol";
import "@openzeppelin/contracts/access/Ownable.sol";

contract AdamMasterPool is Ownable {
    using SafeMath for uint256;
    using SafeERC20 for IERC20;

    // 用户信息
    struct UserInfo {
        uint256 amount; // How many LP tokens the user has provided.
        uint256 rewardDebt; // Reward debt. See explanation below.
    }

    // 池子信息
    struct PoolInfo {
        IERC20 lpToken; // Address of LP token contract.
        uint256 lastRewardBlock;  // Last block number that CAKEs distribution occurs.
        uint256 accAdamPerShare; // Accumulated adam per share
        uint256 maxStaking;
        uint256 rewardPerBlock;
        address[] accounts;
    }

    // 池子信息数组
    PoolInfo[] public poolInfo;

    address[] internal totalAccounts;

    mapping (address => bool) internal Wallets;

    // 池子ID=>用户地址=>用户信息 的映射
    mapping(uint256 => mapping(address => UserInfo)) public userInfo;

    // The CAKE TOKEN!
    IERC20 public rewardToken;

    // The block number when ADAM mining starts.
    uint256 public startBlock;

    // The block number when ADAM mining ends.
    uint256 public bonusEndBlock;

    uint256 public decimal;

    event Deposit(address indexed user, uint256 indexed pid, uint256 amount);
    event Harvest(address indexed user, uint256 indexed pid, uint256 amount);
    event Withdraw(address indexed user, uint256 indexed pid, uint256 amount);
    event EmergencyWithdraw(
```

```
        address indexed user,
        uint256 indexed pid,
        uint256 amount
    );

    constructor(IERC20 _rewardToken,
        uint256 _startBlock,
        uint256 _bonusEndBlock) public  {
        startBlock = _startBlock;
        bonusEndBlock = _bonusEndBlock;
        rewardToken = _rewardToken;
        decimal = 10**10;
    }


    //add lpToken for pool
    function add(
        IERC20 _lpToken,
        uint256 _maxStaking
    ) public onlyOwner{
        poolInfo.push(
            PoolInfo({
            lpToken: _lpToken,
            lastRewardBlock: startBlock,
            accAdamPerShare: 0,
            maxStaking:_maxStaking,
            rewardPerBlock:856110000000000000,
            accounts:totalAccounts
            })
        );

    }

    // Return reward multiplier over the given _from to _to block.
    function getMultiplier(uint256 _from, uint256 _to)
    public
    view
    returns (uint256)
    {
        if (_to <= bonusEndBlock) {
            return _to.sub(_from);
        } else if (_from >= bonusEndBlock) {
            return 0;
        } else {
            return bonusEndBlock.sub(_from);
        }
    }

    // Update reward variables of the given pool to be up-to-date.
    function updatePool(uint256 _pid) internal {
        PoolInfo storage pool = poolInfo[_pid];
        if (block.number <= pool.lastRewardBlock) {
            return;
        }
        uint256 lpSupply = pool.lpToken.balanceOf(address(this));
        if (lpSupply == 0) {
            pool.lastRewardBlock = block.number;
            return;
        }
        uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
        uint256 adamReward = multiplier.mul(pool.rewardPerBlock);
        pool.accAdamPerShare = pool.accAdamPerShare.add(
            adamReward.mul(1e12).div(lpSupply)
        );

        pool.lastRewardBlock = block.number;
```

```
    }

    // View function to see pending Reward on frontend.
    function pendingADAM(uint256 _pid, address _user)
    public
    view
    returns (uint256)
    {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][_user];
        uint256 accAdamPerShare = pool.accAdamPerShare;
        uint256 lpSupply = pool.lpToken.balanceOf(address(this));

        if (block.number > pool.lastRewardBlock && lpSupply != 0) {
            uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
            uint256 adamReward = multiplier.mul(pool.rewardPerBlock);
            accAdamPerShare = accAdamPerShare.add(
                adamReward.mul(1e12).div(lpSupply)
            );
        }
        return user.amount.mul(accAdamPerShare).div(1e12).div(decimal).sub(user.rewardDebt);
    }


    // Deposit LP tokens to AdamRewardPool for ADAM allocation.
    function deposit(uint256 _pid, uint256 _amount) public {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][msg.sender];

        updatePool(_pid);
        if (user.amount > 0) {
            uint256 pending =
            user.amount.mul(pool.accAdamPerShare).div(1e12).div(decimal).sub(
                user.rewardDebt
            );
            if (pending > 0) {
                rewardToken.safeTransfer(address(msg.sender), pending);
            }
        }

        if (_amount > 0) {
            pool.lpToken.safeTransferFrom(
                address(msg.sender),
                address(this),
                _amount
            );
            user.amount = user.amount.add(_amount);
        }
        user.rewardDebt = user.amount.mul(pool.accAdamPerShare).div(1e12).div(decimal);

        if (contains(msg.sender) == false){
            pool.accounts.push(msg.sender);
            totalAccounts.push(msg.sender);
            setWallet(msg.sender);
        }

        emit Deposit(msg.sender, _pid, _amount);
    }


    // harvest ADAM tokens from AdamRewardPool.
    function harvest(uint256 _pid) public {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][msg.sender];
        require(user.amount >= 0, "harvest: not good");
        updatePool(_pid);
```

```
        uint256 pending =
        user.amount.mul(pool.accAdamPerShare).div(1e12).div(decimal).sub(
            user.rewardDebt
        );
        if (pending > 0) {
            rewardToken.safeTransfer(address(msg.sender), pending);
        }

        user.rewardDebt = user.amount.mul(pool.accAdamPerShare).div(1e12).div(decimal);

        emit Harvest(msg.sender, _pid, pending);
    }


    // Withdraw LP tokens from AdamRewardPool.
    function exit(uint256 _pid) public {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][msg.sender];
        require(user.amount >= 0, "withdraw: not good");
        updatePool(_pid);
        uint256 pending =
        user.amount.mul(pool.accAdamPerShare).div(1e12).div(decimal).sub(
            user.rewardDebt
        );
        if (pending > 0) {
            rewardToken.safeTransfer(address(msg.sender), pending);
        }

        uint256 _amount = user.amount;
        if (_amount > 0) {
            user.amount = 0;
            pool.lpToken.safeTransfer(address(msg.sender), _amount);
        }
        user.rewardDebt = user.amount.mul(pool.accAdamPerShare).div(1e12).div(decimal);

        emit Withdraw(msg.sender, _pid, _amount);
    }

    // Withdraw without caring about rewards. EMERGENCY ONLY.
    function emergencyWithdraw(uint256 _pid) public {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][msg.sender];
        uint256 _amount = user.amount;
        pool.lpToken.safeTransfer(address(msg.sender), user.amount);
        user.amount = 0;
        user.rewardDebt = 0;
        emit EmergencyWithdraw(msg.sender, _pid, _amount);
    }


    // Withdraw reward. EMERGENCY ONLY.
    function emergencyRewardWithdraw(uint256 _amount) public onlyOwner {
        require(
            _amount < rewardToken.balanceOf(address(this)),
            "not enough token"
        );

        rewardToken.safeTransfer(address(msg.sender), _amount);
    }

    function poolLength() external view returns (uint256) {
        return poolInfo.length;
    }

    function setMaxStaking(
        uint256 _pid,
```

```
        uint256 _maxStaking
    ) public onlyOwner {
        poolInfo[_pid].maxStaking = _maxStaking;
    }


    function stopReward() public onlyOwner {
        bonusEndBlock = block.number;
    }

    //set bounus end bolck
    function setBonusEndBlock(uint256 _bonusEndBlock) public onlyOwner {
        bonusEndBlock = _bonusEndBlock;
    }

    //set reward per bolck
    function setRewardPerBlock(uint256 _pid, uint256 _rewardPerBlock) public onlyOwner {
        PoolInfo storage pool = poolInfo[_pid];
        pool.rewardPerBlock = _rewardPerBlock;
    }

    //get lp token supply
    function getLpSupply(uint256 _pid)  external view returns (uint256) {
        PoolInfo storage pool = poolInfo[_pid];
        uint256 lpSupply = pool.lpToken.balanceOf(address(this));
        return lpSupply;
    }

    function getBlockNum()  public view onlyOwner returns (uint256) {
        uint256 blNum = block.number;
        return blNum;
    }

    function getAccounts()  public view onlyOwner returns (address[] memory) {
        return totalAccounts;
    }

    function getAccountsLength()  public view onlyOwner returns (uint256) {
        return totalAccounts.length;
    }

    function getPoolAccounts(uint256 _pid)  public view onlyOwner returns (address[] memory) {
        PoolInfo storage pool = poolInfo[_pid];
        return pool.accounts;
    }

    function getPoolAccountsLength(uint256 _pid)  public view onlyOwner returns (uint256) {
        PoolInfo storage pool = poolInfo[_pid];
        return pool.accounts.length;
    }

    function setWallet(address _wallet) internal{
        Wallets[_wallet] = true;
    }

    function contains(address _wallet) internal view returns (bool){
        return Wallets[_wallet];
    }

}


pragma solidity 0.6.12;

import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import "@openzeppelin/contracts/token/ERC20/SafeERC20.sol";
```

```
import "@openzeppelin/contracts/math/SafeMath.sol";
import "@openzeppelin/contracts/access/Ownable.sol";

contract AdamRewardPool is Ownable {
    using SafeMath for uint256;
    using SafeERC20 for IERC20;

    // 用户信息
    struct UserInfo {
        uint256 amount; // How many LP tokens the user has provided.
        uint256 rewardDebt; // Reward debt. See explanation below.
    }

    // 池子信息
    struct PoolInfo {
        IERC20 lpToken; // Address of LP token contract.
        uint256 lastRewardBlock;  // Last block number that CAKEs distribution occurs.
        uint256 accAdamPerShare; // Accumulated adam per share
        uint256 maxStaking;
        uint256 lpSupply;
        uint256 rewardPerBlock;
        address[] accounts;
    }

    // 池子信息数组
    PoolInfo[] public poolInfo;

    address[] internal totalAccounts;

    mapping (address => bool) internal Wallets;

    // 池子ID=>用户地址=>用户信息 的映射
    mapping(uint256 => mapping(address => UserInfo)) public userInfo;

    // The CAKE TOKEN!
    IERC20 public rewardToken;

    // The block number when ADAM mining starts.
    uint256 public startBlock;

    // The block number when ADAM mining ends.
    uint256 public bonusEndBlock;

    uint256 public decimal;

    event Deposit(address indexed user, uint256 indexed pid, uint256 amount);
    event Harvest(address indexed user, uint256 indexed pid, uint256 amount);
    event Withdraw(address indexed user, uint256 indexed pid, uint256 amount);
    event EmergencyWithdraw(
        address indexed user,
        uint256 indexed pid,
        uint256 amount
    );

    constructor(IERC20 _rewardToken,
        uint256 _startBlock,
        uint256 _bonusEndBlock) public  {
        startBlock = _startBlock;
        bonusEndBlock = _bonusEndBlock;
        rewardToken = _rewardToken;
        decimal = 10**10;
    }


    //add lpToken for pool
    function add(
```

```
        IERC20 _lpToken,
        uint256 _maxStaking
    ) public onlyOwner{
        poolInfo.push(
            PoolInfo({
            lpToken: _lpToken,
            lastRewardBlock: startBlock,
            accAdamPerShare: 0,
            lpSupply: 0,
            maxStaking:_maxStaking,
            rewardPerBlock:85611000000000000,
            accounts:totalAccounts
            })
        );

    }

    // Return reward multiplier over the given _from to _to block.
    function getMultiplier(uint256 _from, uint256 _to)
    public
    view
    returns (uint256)
    {
        if (_to <= bonusEndBlock) {
            return _to.sub(_from);
        } else if (_from >= bonusEndBlock) {
            return 0;
        } else {
            return bonusEndBlock.sub(_from);
        }
    }

    // Update reward variables of the given pool to be up-to-date.
    function updatePool(uint256 _pid) internal {
        PoolInfo storage pool = poolInfo[_pid];
        if (block.number <= pool.lastRewardBlock) {
            return;
        }
        // uint256 lpSupply = pool.lpToken.balanceOf(address(this));
        if (pool.lpSupply == 0) {
            pool.lastRewardBlock = block.number;
            return;
        }
        uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
        uint256 adamReward = multiplier.mul(pool.rewardPerBlock);

        pool.accAdamPerShare = pool.accAdamPerShare.add(
            adamReward.mul(1e12).div(pool.lpSupply)
        );


        pool.lastRewardBlock = block.number;
    }

    // View function to see pending Reward on frontend.
    function pendingADAM(uint256 _pid, address _user)
    public
    view
    returns (uint256)
    {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][_user];
        uint256 accAdamPerShare = pool.accAdamPerShare;
        uint256 adamReward = 0;
        if (block.number > pool.lastRewardBlock && pool.lpSupply != 0) {
            uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
```

```
            adamReward = multiplier.mul(pool.rewardPerBlock);
            accAdamPerShare = accAdamPerShare.add(
                adamReward.mul(1e12).div(pool.lpSupply)
            );
        }
        return user.amount.mul(accAdamPerShare).div(1e12).div(decimal).sub(user.rewardDebt);
    }


    // Deposit LP tokens to AdamRewardPool for ADAM allocation.
    function deposit(uint256 _pid, uint256 _amount) public {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][msg.sender];

        updatePool(_pid);
        if (user.amount > 0) {
            uint256 pending =
            user.amount.mul(pool.accAdamPerShare).div(1e12).div(decimal).sub(
                user.rewardDebt
            );
            if (pending > 0) {
                rewardToken.safeTransfer(address(msg.sender), pending);
            }
        }

        if (_amount > 0) {
            pool.lpToken.safeTransferFrom(
                address(msg.sender),
                address(this),
                _amount
            );
            user.amount = user.amount.add(_amount);
            pool.lpSupply = pool.lpSupply.add(_amount);

        }
        user.rewardDebt = user.amount.mul(pool.accAdamPerShare).div(1e12).div(decimal);

        if (contains(msg.sender) == false){
            pool.accounts.push(msg.sender);
            totalAccounts.push(msg.sender);
            setWallet(msg.sender);
        }

        emit Deposit(msg.sender, _pid, _amount);
    }


    // harvest ADAM tokens from AdamRewardPool.
    function harvest(uint256 _pid) public {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][msg.sender];
        require(user.amount >= 0, "harvest: not good");
        updatePool(_pid);
        uint256 pending =
        user.amount.mul(pool.accAdamPerShare).div(1e12).div(decimal).sub(
            user.rewardDebt
        );
        if (pending > 0) {
            rewardToken.safeTransfer(address(msg.sender), pending);
        }

        user.rewardDebt = user.amount.mul(pool.accAdamPerShare).div(1e12).div(decimal);

        emit Harvest(msg.sender, _pid, pending);
    }
```

```
// Withdraw LP tokens from AdamRewardPool.
function exit(uint256 _pid) public {
    PoolInfo storage pool = poolInfo[_pid];
    UserInfo storage user = userInfo[_pid][msg.sender];
    require(user.amount >= 0, "withdraw: not good");
    updatePool(_pid);
    uint256 pending =
    user.amount.mul(pool.accAdamPerShare).div(1e12).div(decimal).sub(
        user.rewardDebt
    );
    if (pending > 0) {
        rewardToken.safeTransfer(address(msg.sender), pending);
    }

    uint256 _amount = user.amount;
    if (_amount > 0) {
        user.amount = 0;
        pool.lpToken.safeTransfer(address(msg.sender), _amount);
        pool.lpSupply = pool.lpSupply.sub(_amount);
    }
    user.rewardDebt = user.amount.mul(pool.accAdamPerShare).div(1e12).div(decimal);

    emit Withdraw(msg.sender, _pid, _amount);
}

// Withdraw without caring about rewards. EMERGENCY ONLY.
function emergencyWithdraw(uint256 _pid) public {
    PoolInfo storage pool = poolInfo[_pid];
    UserInfo storage user = userInfo[_pid][msg.sender];
    uint256 _amount = user.amount;
    pool.lpToken.safeTransfer(address(msg.sender), user.amount);
    user.amount = 0;
    user.rewardDebt = 0;
    emit EmergencyWithdraw(msg.sender, _pid, _amount);
}


// Withdraw reward. EMERGENCY ONLY.
function emergencyRewardWithdraw(uint256 _amount) public onlyOwner {
    require(
        _amount < rewardToken.balanceOf(address(this)),
        "not enough token"
    );

    rewardToken.safeTransfer(address(msg.sender), _amount);
}

function poolLength() external view returns (uint256) {
    return poolInfo.length;
}

function setMaxStaking(
    uint256 _pid,
    uint256 _maxStaking
) public onlyOwner {
    poolInfo[_pid].maxStaking = _maxStaking;
}


function stopReward() public onlyOwner {
    bonusEndBlock = block.number;
}

//set bounus end bolck
function setBonusEndBlock(uint256 _bonusEndBlock) public onlyOwner {
```

```
        bonusEndBlock = _bonusEndBlock;
    }

    //set reward per bolck
    function setRewardPerBlock(uint256 _pid, uint256 _rewardPerBlock) public onlyOwner {
        PoolInfo storage pool = poolInfo[_pid];
        pool.rewardPerBlock = _rewardPerBlock;
    }

    //get lp token supply
    function getLpSupply(uint256 _pid)  external view returns (uint256) {
        PoolInfo storage pool = poolInfo[_pid];
        uint256 lpSupply = pool.lpToken.balanceOf(address(this));
        return lpSupply;
    }


    function getBlockNum()  public view onlyOwner returns (uint256) {
        uint256 blNum = block.number;
        return blNum;
    }

    function getAccounts()  public view onlyOwner returns (address[] memory) {
        return totalAccounts;
    }

    function getAccountsLength()  public view onlyOwner returns (uint256) {
        return totalAccounts.length;
    }

    function getPoolAccounts(uint256 _pid)  public view onlyOwner returns (address[] memory) {
        PoolInfo storage pool = poolInfo[_pid];
        return pool.accounts;
    }

    function getPoolAccountsLength(uint256 _pid)  public view onlyOwner returns (uint256) {
        PoolInfo storage pool = poolInfo[_pid];
        return pool.accounts.length;
    }

    function setWallet(address _wallet) internal{
        Wallets[_wallet] = true;
    }

    function contains(address _wallet) internal view returns (bool){
        return Wallets[_wallet];
    }

}
```


Analysis of audit results

Re-Entrancy

- **Description:**
  One of the features of smart contracts is the ability to call and utilise code of other external contracts. Contracts also typically handle Blockchain Currency, and as such often send Blockchain Currency to various external user addresses. The operation of calling external contracts, or sending Blockchain Currency to an address, requires the contract to submit an external call. These external calls can be hijacked by attackers whereby they force the

contract to execute further code (i.e. through a fallback function) , including calls back into itself. Thus the code execution "re-enters" the contract. Attacks of this kind were used in the infamous DAO hack.

- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Arithmetic Over/Under Flows

- **Description:**
  The Virtual Machine (EVM) specifies fixed-size data types for integers. This means that an integer variable, only has a certain range of numbers it can represent. A uint8 for example, can only store numbers in the range [0,255]. Trying to store 256 into a uint8 will result in 0. If care is not taken, variables in Solidity can be exploited if user input is unchecked and calculations are performed which result in numbers that lie outside the range of the data type that stores them.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Unexpected Blockchain Currency

- **Description:**
  Typically when Blockchain Currency is sent to a contract, it must execute either the fallback function, or another function described in the contract. There are two exceptions to this, where Blockchain Currency can exist in a contract without having executed any code. Contracts which rely on code execution for every Blockchain Currency sent to the contract can be vulnerable to attacks where Blockchain Currency is forcibly sent to a contract.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:** no.

Delegatecall

- **Description:**
  The CALL and DELEGATECALL opcodes are useful in allowing developers to modularise their code. Standard external message calls to contracts are handled by the CALL opcode whereby code is run in the context of the external contract/function. The DELEGATECALL opcode is identical to the standard message call, except that the code executed at the targeted address is run in the context of the calling contract along with the fact that msg.sender and msg.value remain unchanged. This feature enables the implementation of libraries whereby developers can create reusable code for future contracts.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:** no.

Default Visibilities

- **Description:**
  Functions in Solidity have visibility specifiers which dictate how functions are allowed to be called. The visibility determines whBlockchain Currency a function can be called externally by users, by other derived contracts, only internally or only externally. There are four visibility specifiers, which are described in detail in the Solidity Docs. Functions default to public allowing users to call them externally. Incorrect use of visibility specifiers can lead to some devestating vulernabilities in smart contracts as will be discussed in this section.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Entropy Illusion

- **Description:**
  All transactions on the blockchain are deterministic state transition operations. Meaning that every transaction modifies the global state of the ecosystem and it does so in a calculable way with no uncertainty. This ultimately means that inside the blockchain ecosystem there is no source of entropy or randomness. There is no rand() function in Solidity. Achieving decentralised entropy (randomness) is a well established problem and many ideas have been proposed to address this (see for example, RandDAO or using a chain of Hashes as described by Vitalik in this post).
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

External Contract Referencing

- **Description:**
  One of the benefits of the global computer is the ability to re-use code and interact with contracts already deployed on the network. As a result, a large number of contracts reference external contracts and in general operation use external message calls to interact with these contracts. These external message calls can mask malicious actors intentions in some non-obvious ways, which we will discuss.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Unsolved TODO comments

- **Description:**
  Check for Unsolved TODO comments
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Short Address/Parameter Attack

- **Description:**
  This attack is not specifically performed on Solidity contracts themselves but on third party applications that may interact with them. I add this attack for completeness and to be aware of how parameters can be manipulated in contracts.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Unchecked CALL Return Values

- **Description:**
  There a number of ways of performing external calls in solidity. Sending Blockchain Currency to external accounts is commonly performed via the transfer() method. However, the send() function can also be used and, for more versatile external calls, the CALL opcode can be directly employed in solidity. The call() and send() functions return a boolean indicating if the call succeeded or failed. Thus these functions have a simple caveat, in that the transaction that executes these functions will not revert if the external call (intialised by call() or send()) fails, rather the call() or send() will simply return false. A common pitfall arises when the return value is not checked, rather the developer expects a revert to occur.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Race Conditions / Front Running

- **Description:**
  The combination of external calls to other contracts and the multi-user nature of the underlying blockchain gives rise to a variety of potential Solidity pitfalls whereby users race code execution to obtain unexpected states. Re-Entrancy is one example of such a race condition. In this section we will talk more generally about different kinds of race conditions that can occur on the blockchain. There is a variety of good posts on this subject, a few are: Wiki - Safety, DASP - Front-Running and the Consensus - Smart Contract Best Practices.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Denial Of Service (DOS)

- **Description:**
  This category is very broad, but fundamentally consists of attacks where users can leave the contract inoperable for a small period of time, or in some cases, permanently. This can trap Blockchain Currency in these contracts forever, as was the case with the Second Parity MultiSig hack
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Block Timestamp Manipulation

- **Description:**
  Block timestamps have historically been used for a variety of applications, such as entropy for random numbers (see the Entropy Illusion section for further details), locking funds for periods of time and various state-changing conditional statements that are time-dependent. Miner's have the ability to adjust timestamps slightly which can prove to be quite dangerous if block timestamps are used incorrectly in smart contracts.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Constructors with Care

- **Description:**
  Constructors are special functions which often perform critical, privileged tasks when initialising contracts. Before solidity v0.4.22 constructors were defined as functions that had the same name as the contract that contained them. Thus, when a contract name gets changed in development, if the constructor name isn't changed, it becomes a normal, callable function. As you can imagine, this can (and has) lead to some interesting contract hacks.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Unintialised Storage Pointers

- **Description:**
  The EVM stores data either as storage or as memory. Understanding exactly how this is done and the default

types for local variables of functions is highly recommended when developing contracts. This is because it is possible to produce vulnerable contracts by inappropriately intialising variables.

- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Floating Points and Numerical Precision

- **Description:**
  As of this writing (Solidity v0.4.24), fixed point or floating point numbers are not supported. This means that floating point representations must be made with the integer types in Solidity. This can lead to errors/vulnerabilities if not implemented correctly.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

tx.origin Authentication

- **Description:**
  Solidity has a global variable, tx.origin which traverses the entire call stack and returns the address of the account that originally sent the call (or transaction). Using this variable for authentication in smart contracts leaves the contract vulnerable to a phishing-like attack.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

Permission restrictions

- **Description:**
  Contract managers who can control liquidity or pledge pools, etc., or impose unreasonable restrictions on other users.
- **Detection results:**

```
PASSED!
```

- **Security suggestion:**
  no.

armors.io

contact@armors.io