Armors Labs

Balincer Token (BLCR)

Smart Contract Audit

Armors Labs

Balincer Token (BLCR) Audit Summary

Project name : Balincer Token (BLCR) Contract

Project address: None

Code URL : https://www.oklink.com/okexchain/address/0x2Ac78e8D02FE62b020da7a1C4980203faffbe0F5

Commit : None

Project target : Balincer Token (BLCR) Contract Audit

Blockchain : OKExChain

Test result : PASSED

Audit Info

Audit NO : 0X202111030006

Audit Team : Armors Labs

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

Armors Labs

Balincer Token (BLCR) Audit

The Balincer Token (BLCR) team asked us to review and audit their Balincer Token (BLCR) 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
Balincer Token (BLCR) Audit	Rock, Sophia, Rushairer, Rico, David, Alice	1.0.0	2021-11-03

Audit results

Note that as of the date of publishing, the above review reflects the current understanding of known security patterns as they relate to the Balincer Token (BLCR) 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.

Disclaimer

Armors Labs Reports is not and should not be regarded as an "approval" or "disapproval" of any particular project or team. These reports are not and should not be regarded as indicators of the economy or value of any "product" or "asset" created by any team. Armors do not cover testing or auditing the integration with external contract or services (such as Unicrypt, Uniswap, PancakeSwap etc'…)

Armors Labs Reports represent an extensive auditing process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology. Armors does not guarantee the safety or functionality of the technology agreed to be analyzed.

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. Armors Labs Audits should not be used in any way to make decisions around investment or involvement with any particular project. These reports in no way provide investment advice, nor should be leveraged as investment advice of any sort.

Audited target file

file	md5
./BalincerToken.sol	ced31f8c96d54a835857e90eba1c4347

Vulnerability analysis

Vulnerability distribution

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

Armors Labs

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

Vulnerability	status
Block Timestamp Manipulation	safe
Constructors with Care	safe
Unintialised Storage Pointers	safe
Floating Points and Numerical Precision	safe
tx.origin Authentication	safe
Permission restrictions	safe

Contract file

```
pragma solidity ^0.5.17;
// pragma experimental ABIEncoderV2;
/**
 * @dev Wrappers over Solidity's arithmetic operations with added overflow
 * checks.
 *
 * Arithmetic operations in Solidity wrap on overflow. This can easily result
 * in bugs, because programmers usually assume that an overflow raises an
 * error, which is the standard behavior in high level programming languages.
 * SafeMath restores this intuition by reverting the transaction when an
 * operation overflows.
 *
 * Using this library instead of the unchecked operations eliminates an entire
 * class of bugs, so it's recommended to use it always.
 */

library SafeMath {
    function trx(uint256 a) internal pure returns (uint256) {
        assert(a > 0);
        uint256 c = a * 1000000;
        assert(a == 0 || c / a == 1000000);
        return c;
    }

    function mul(uint256 a, uint256 b) internal pure returns (uint256) {
        uint256 c = a * b;
        assert(a == 0 || c / a == b);
        return c;
    }

    function div(uint256 a, uint256 b) internal pure returns (uint256) {
        // assert(b > 0); // Solidity automatically throws 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;
    }

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

    /**
     * @dev Returns the subtraction of two unsigned integers, reverting with custom message on
     * overflow (when the result is negative).
     *
```

Armors Labs

```
       * Counterpart to Solidity's - operator.
       *
       * Requirements:
       *
       * - Subtraction cannot overflow.
       */
    function sub(uint256 a, uint256 b, string memory errorMessage) internal pure returns (uint256) {
        require(b <= a, errorMessage);
        uint256 c = a - b;

        return c;
    }

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

    function ceil(uint256 a, uint256 m) internal pure returns (uint256) {
        return ((a + m - 1) / m) * m;
    }

    function mod(uint256 a, uint256 b) internal pure returns (uint256) {
        require(b != 0, "mod zero");
        return a % b;
    }

    function min(uint256 a, uint256 b) internal pure returns (uint256) {
        return a > b ? b : a;
    }

    function sq(uint256 x) internal pure returns (uint256)
    {
        return (mul(x, x));
    }

    function sqrt(uint256 x) internal pure returns (uint256 y)
    {
        uint256 z = ((add(x, 1)) / 2);
        y = x;
        while (z < y)
        {
            y = z;
            z = ((add((x / z), z)) / 2);
        }
    }

    function pwr(uint256 x, uint256 y) internal pure returns (uint256)
    {
        if (x == 0) return (0);
        else if (y == 0) return (1);
        else {
            uint256 z = x;
            for (uint256 i = 1; i < y; i++) z = mul(z, x);
            return (z);
        }
    }
}

contract BalincerToken {
    /// @notice EIP-20 token name for this token
    string public constant name = "Balincer Token";
    /// @notice EIP-20 token symbol for this token
    string public constant symbol = "BLCR";
    /// @notice EIP-20 token decimals for this token
```

Armors Labs

```
    uint8 public constant decimals = 18;
    /// @notice Total number of tokens in circulation
    uint256 public totalSupply = 100_000_000e18;
    /// @notice Allowance amounts on behalf of others
    mapping(address => mapping(address => uint96)) internal allowances;
    /// @notice Official record of token balances for each account
    mapping(address => uint96) internal balances;
    /// @notice A record of each accounts delegate
    mapping(address => address) public delegates;
    /// @notice A checkpoint for marking number of votes from a given block
    struct Checkpoint {
        uint32 fromBlock;
        uint96 votes;
    }
    /// @notice A record of votes checkpoints for each account, by index
    mapping(address => mapping(uint32 => Checkpoint)) public checkpoints;
    /// @notice The number of checkpoints for each account
    mapping(address => uint32) public numCheckpoints;
    /// @notice The EIP-712 typehash for the contract's domain
    bytes32 public constant DOMAIN_TYPEHASH = keccak256("EIP712Domain(string name,uint256 chainId,add
    /// @notice The EIP-712 typehash for the delegation struct used by the contract
    bytes32 public constant DELEGATION_TYPEHASH = keccak256("Delegation(address delegatee,uint256 non
    /// @notice The EIP-712 typehash for the permit struct used by the contract
    bytes32 public constant PERMIT_TYPEHASH = keccak256("Permit(address owner,address spender,uint256
    /// @notice A record of states for signing / validating signatures
    mapping(address => uint256) public nonces;
    /// @notice An event thats emitted when an account changes its delegate
    event DelegateChanged(address indexed delegator, address indexed fromDelegate, address indexed to
    /// @notice An event thats emitted when a delegate account's vote balance changes
    event DelegateVotesChanged(address indexed delegate, uint256 previousBalance, uint256 newBalance)
    /// @notice The standard EIP-20 transfer event
    event Transfer(address indexed from, address indexed to, uint256 amount);
    /// @notice The standard EIP-20 approval event
    event Approval(address indexed owner, address indexed spender, uint256 amount);
    /**
     * @notice Construct a new Ba token
     * @param account The initial account to grant all the tokens
     */
    constructor(address account) public {
        balances[account] = uint96(totalSupply);
        emit Transfer(address(0), account, totalSupply);
    }
    /**
     * @notice Get the number of tokens spender is approved to spend on behalf of account
     * @param account The address of the account holding the funds
     * @param spender The address of the account spending the funds
     * @return The number of tokens approved
     */
    function allowance(address account, address spender) external view returns (uint256) {
        return allowances[account][spender];
    }

    /**
     * @notice Approve spender to transfer up to amount from src
     * @dev This will overwrite the approval amount for spender
     *  and is subject to issues noted [here](https://eips.ethereum.org/EIPS/eip-20#approve)
     * @param spender The address of the account which may transfer tokens
     * @param rawAmount The number of tokens that are approved (2^256-1 means infinite)
     * @return Whether or not the approval succeeded
     */
    function approve(address spender, uint256 rawAmount) external returns (bool) {
        uint96 amount;
        if (rawAmount == uint256(-1)) {
            amount = uint96(-1);
        } else {
            amount = safe96(rawAmount, "Ba::approve: amount exceeds 96 bits");
```

Armors Labs

```
        }
        allowances[msg.sender][spender] = amount;
        emit Approval(msg.sender, spender, amount);
        return true;
    }

    /**
     * @notice Triggers an approval from owner to spends
     * @param owner The address to approve from
     * @param spender The address to be approved
     * @param rawAmount The number of tokens that are approved (2^256-1 means infinite)
     * @param deadline The time at which to expire the signature
     * @param v The recovery byte of the signature
     * @param r Half of the ECDSA signature pair
     * @param s Half of the ECDSA signature pair
     */
    function permit(address owner, address spender, uint256 rawAmount, uint256 deadline, uint8 v, byt
        uint96 amount;
        if (rawAmount == uint256(-1)) {
            amount = uint96(-1);
        } else {
            amount = safe96(rawAmount, "Ba::permit: amount exceeds 96 bits");
        }
        bytes32 domainSeparator = keccak256(abi.encode(DOMAIN_TYPEHASH, keccak256(bytes(name)), getCh
        bytes32 structHash = keccak256(abi.encode(PERMIT_TYPEHASH, owner, spender, rawAmount, nonces[
        bytes32 digest = keccak256(abi.encodePacked("\x19\x01", domainSeparator, structHash));
        address signatory = ecrecover(digest, v, r, s);
        require(signatory != address(0), "Ba::permit: invalid signature");
        require(signatory == owner, "Ba::permit: unauthorized");
        require(now <= deadline, "Ba::permit: signature expired");
        allowances[owner][spender] = amount;
        emit Approval(owner, spender, amount);
    }

    /**
     * @notice Get the number of tokens held by the account
     * @param account The address of the account to get the balance of
     * @return The number of tokens held
     */
    function balanceOf(address account) external view returns (uint256) {
        return balances[account];
    }

    /**
     * @notice Transfer amount tokens from msg.sender to dst
     * @param dst The address of the destination account
     * @param rawAmount The number of tokens to transfer
     * @return Whether or not the transfer succeeded
     */
    function transfer(address dst, uint256 rawAmount) external returns (bool) {
        uint96 amount = safe96(rawAmount, "Ba::transfer: amount exceeds 96 bits");
        _transferTokens(msg.sender, dst, amount);
        return true;
    }

    /**
     * @notice Transfer amount tokens from src to dst
     * @param src The address of the source account
     * @param dst The address of the destination account
     * @param rawAmount The number of tokens to transfer
     * @return Whether or not the transfer succeeded
     */
    function transferFrom(address src, address dst, uint256 rawAmount) external returns (bool) {
        address spender = msg.sender;
        uint96 spenderAllowance = allowances[src][spender];
        uint96 amount = safe96(rawAmount, "Ba::approve: amount exceeds 96 bits");
```

Armors Labs

```
        if (spender != src && spenderAllowance != uint96(-1)) {
            uint96 newAllowance = sub96(spenderAllowance, amount, "Ba::transferFrom: transfer amount
            allowances[src][spender] = newAllowance;
            emit Approval(src, spender, newAllowance);
        }
        _transferTokens(src, dst, amount);
        return true;
    }

    /**
     * @notice Delegate votes from msg.sender to delegatee
     * @param delegatee The address to delegate votes to
     */
    function delegate(address delegatee) public {
        return _delegate(msg.sender, delegatee);
    }

    /**
     * @notice Delegates votes from signatory to delegatee
     * @param delegatee The address to delegate votes to
     * @param nonce The contract state required to match the signature
     * @param expiry The time at which to expire the signature
     * @param v The recovery byte of the signature
     * @param r Half of the ECDSA signature pair
     * @param s Half of the ECDSA signature pair
     */
    function delegateBySig(address delegatee, uint256 nonce, uint256 expiry, uint8 v, bytes32 r, byte
        bytes32 domainSeparator = keccak256(abi.encode(DOMAIN_TYPEHASH, keccak256(bytes(name)), getCh
        bytes32 structHash = keccak256(abi.encode(DELEGATION_TYPEHASH, delegatee, nonce, expiry));
        bytes32 digest = keccak256(abi.encodePacked("\x19\x01", domainSeparator, structHash));
        address signatory = ecrecover(digest, v, r, s);
        require(signatory != address(0), "Ba::delegateBySig: invalid signature");
        require(nonce == nonces[signatory]++, "Ba::delegateBySig: invalid nonce");
        require(now <= expiry, "Ba::delegateBySig: signature expired");
        return _delegate(signatory, delegatee);
    }

    /**
     * @notice Gets the current votes balance for account
     * @param account The address to get votes balance
     * @return The number of current votes for account
     */
    function getCurrentVotes(address account) external view returns (uint96) {
        uint32 nCheckpoints = numCheckpoints[account];
        return nCheckpoints > 0 ? checkpoints[account][nCheckpoints - 1].votes : 0;
    }

    /**
     * @notice Determine the prior number of votes for an account as of a block number
     * @dev Block number must be a finalized block or else this function will revert to prevent misin
     * @param account The address of the account to check
     * @param blockNumber The block number to get the vote balance at
     * @return The number of votes the account had as of the given block
     */
    function getPriorVotes(address account, uint256 blockNumber) public view returns (uint96) {
        require(blockNumber < block.number, "Ba::getPriorVotes: not yet determined");
        uint32 nCheckpoints = numCheckpoints[account];
        if (nCheckpoints == 0) {return 0;}
        // First check most recent balance
        if (checkpoints[account][nCheckpoints - 1].fromBlock <= blockNumber) { return checkpoints[acc
        // Next check implicit zero balance
        if (checkpoints[account][0].fromBlock > blockNumber) { return 0;}
        uint32 lower = 0;
        uint32 upper = nCheckpoints - 1;
        while (upper > lower) {
            uint32 center = upper - (upper - lower) / 2; // ceil, avoiding overflow
```

Armors Labs

```
            Checkpoint memory cp = checkpoints[account][center];
            if (cp.fromBlock == blockNumber) {
                return cp.votes;
            } else if (cp.fromBlock < blockNumber) {
                lower = center;
            } else {
                upper = center - 1;
            }
        }
        return checkpoints[account][lower].votes;
    }

    function _delegate(address delegator, address delegatee) internal {
        address currentDelegate = delegates[delegator];
        uint96 delegatorBalance = balances[delegator];
        delegates[delegator] = delegatee;
        emit DelegateChanged(delegator, currentDelegate, delegatee);
        _moveDelegates(currentDelegate, delegatee, delegatorBalance);
    }

    function _transferTokens(address src, address dst, uint96 amount) internal {
        require(src != address(0), "Ba::_transferTokens: cannot transfer from the zero address");
        require(dst != address(0), "Ba::_transferTokens: cannot transfer to the zero address");
        balances[src] = sub96(balances[src], amount, "Ba::_transferTokens: transfer amount exceeds ba
        balances[dst] = add96(balances[dst], amount, "Ba::_transferTokens: transfer amount overflows"
        emit Transfer(src, dst, amount);
        _moveDelegates(delegates[src], delegates[dst], amount);
    }

    function _moveDelegates(address srcRep, address dstRep, uint96 amount) internal {
        if (srcRep != dstRep && amount > 0) {
            if (srcRep != address(0)) {
                uint32 srcRepNum = numCheckpoints[srcRep];
                uint96 srcRepOld = srcRepNum > 0 ? checkpoints[srcRep][srcRepNum - 1].votes : 0;
                uint96 srcRepNew = sub96(srcRepOld, amount, "Ba::_moveVotes: vote amount underflows"
                _writeCheckpoint(srcRep, srcRepNum, srcRepOld, srcRepNew);
            }

            if (dstRep != address(0)) {
                uint32 dstRepNum = numCheckpoints[dstRep];
                uint96 dstRepOld = dstRepNum > 0 ? checkpoints[dstRep][dstRepNum - 1].votes : 0;
                uint96 dstRepNew = add96(dstRepOld, amount, "Ba::_moveVotes: vote amount overflows");
                _writeCheckpoint(dstRep, dstRepNum, dstRepOld, dstRepNew);
            }
        }
    }

    function _writeCheckpoint(address delegatee, uint32 nCheckpoints, uint96 oldVotes, uint96 newVote
        uint32 blockNumber = safe32(block.number, "Ba::_writeCheckpoint: block number exceeds 32 bits
        if (nCheckpoints > 0 && checkpoints[delegatee][nCheckpoints - 1].fromBlock == blockNumber) {
            checkpoints[delegatee][nCheckpoints - 1].votes = newVotes;
        } else {
            checkpoints[delegatee][nCheckpoints] = Checkpoint(blockNumber, newVotes);
            numCheckpoints[delegatee] = nCheckpoints + 1;
        }
        emit DelegateVotesChanged(delegatee, oldVotes, newVotes);
    }

    function safe32(uint256 n, string memory errorMessage) internal pure returns (uint32) {
        require(n < 2**32, errorMessage);
        return uint32(n);
    }

    function safe96(uint256 n, string memory errorMessage) internal pure returns (uint96) {
        require(n < 2**96, errorMessage);
        return uint96(n);
```

Armors Labs

```
    }

    function add96( uint96 a, uint96 b, string memory errorMessage) internal pure returns (uint96) {
        uint96 c = a + b;
        require(c >= a, errorMessage);
        return c;
    }

    function sub96(uint96 a, uint96 b, string memory errorMessage) internal pure returns (uint96) {
        require(b <= a, errorMessage);
        return a - b;
    }

    function getChainId() internal pure returns (uint256) {
        uint256 chainId;
        assembly {
            chainId := chainid()
        }
        return chainId;
    }
}
```

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.

Armors Labs

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.

Armors Labs

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.

Armors Labs

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.

Armors Labs

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.

Armors Labs

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 Labs

armors.io

contact@armors.io