Armors Labs

PETBOX (PBOX) Token

Smart Contract Audit

PETBOX (PBOX) Token Audit Summary

Project name : PETBOX (PBOX) Token Contract

Project address: None

Code URL : https://bscscan.com/address/0xD30bce3B806f8861877c3f7AcE7C0406895154fc#code

Commit : None

Project target : PETBOX (PBOX) Token Contract Audit

Blockchain : Binance Smart Chain（BSC）

Test result : FAILED!

Audit Info

Audit NO : 0X202110020006

Audit Team : Armors Labs

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

PETBOX (PBOX) Token Audit

The PETBOX (PBOX) Token team asked us to review and audit their PETBOX (PBOX) Token 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
PETBOX (PBOX) Token Audit	Rock, Sophia, Rushairer, Rico, David, Alice	1.0.0	2021-10-02

Audit results

Warning:

1. The owner can add and delete administrators, who can mint token.

2. The total amount is inconsistent with the total amount held by users due to division precision problems.

3. burn 3% token by transfer interface.

-

Note that as of the date of publishing, the above review reflects the current understanding of known security patterns as they relate to the PETBOX (PBOX) Token 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
./PboXToken.sol	0598f3ba3dd1dd7e35ac4e404f860eb8

Vulnerability analysis

Vulnerability distribution

vulnerability level	number
Critical severity	0
High severity	0
Medium severity	1
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

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

Contract file

```
/**
 *Submitted for verification at BscScan.com on 2021-10-02
*/

/**
 *Submitted for verification at BscScan.com on 2021-09-28
*/

/**
 *Submitted for verification at BscScan.com on 2021-09-15
*/

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

/**
 * @dev Interface of the ERC20 standard as defined in the
 */
interface IERC20 {
    /**
     * @dev Returns the amount of tokens in existence.
     */
    function totalSupply() external view returns (uint256);

    /**
     * @dev Returns the amount of tokens owned by `account`.
```

```
*/
 function balanceOf(address account) external view returns (uint256);

                    /**
* @dev Moves `amount` tokens from the caller's account to `recipient`.
*
* Returns a boolean value indicating whether the opera
*
* Emits a {Transfer} event.
*/
 function transfer(address recipient, uint256 amount) external returns (bool);

                    /**
* @dev Returns the remaining number of tokens that `spender` will
* allowed to spend on behalf of `owner` through {transferFrom}. This is
* zero by default.
*
* This value changes when {approve} or {transferFrom} are called.
*/
 function allowance(address owner, address spender) external view returns (uint256);

                    /**
* @dev Sets `amount` as the allowance of `spender` over the
*
* Returns a boolean value indicating whether the opera
*
* IMPORTANT: Beware that changing an allowance with this method brings
* that someone may use both the old and the new allov
* 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
*
* Emits an {Approval} event.
*/
 function approve(address spender, uint256 amount) external returns (bool);

                    /**
* @dev Moves `amount` tokens from `sender` to `recipient` using the
* allowance mechanism. `amount` is then deducted from the caller's
* allowance.
*
* Returns a boolean value indicating whether the opera
*
* Emits a {Transfer} event.
*/
 function transferFrom(address sender, address recipient, uint256 amount) external returns (bool);

                    /**
* @dev Emitted when `value` tokens are moved from one account (`from`) to
* another (`to`).
*
```

```
   * Note that `value` may be zero.
   */
    event Transfer(address indexed from, address indexed to, uint256 value);

    /**
   * @dev Emitted when the allowance of a `spender` for
   * a call to {approve}. `value` is the new allowance.
   */
    event Approval(address indexed owner, address indexed spender, uint256 value);
}


pragma solidity ^0.8.0;

/*
 * @dev Provides information about the current execution context, including the
 * sender of the transaction and its data. While these are generally available
 * via msg.sender and msg.data, they should not be accessed in such a direct
 * manner, since when dealing with meta-transactions the account sending and
 * paying for execution may not be the actual sender (as far as an application
 * is concerned).
 *
 * This contract is only required for intermediate, library-like contracts.
 */
abstract contract Context {
    function _msgSender() internal view virtual returns (address) {
        return msg.sender;
    }

    function _msgData() internal view virtual returns (bytes calldata) {
        this; // silence state mutability warning without generating bytecode - see https://github.co
        return msg.data;
    }
}

pragma solidity ^0.8.0;

/**
 * @dev Implementation of the {IERC20} interface.
 *
 * This implementation is agnostic to the way tokens are c
 * that a supply mechanism has to be added in a derived c
 * For a generic mechanism see {ERC20PresetMinterPauser}.
 *
 * TIP: For a detailed writeup see our guide
 * https://forum.zeppelin.solutions/t/how-to-implement-erc20-supply-mechanisms/226[How
 * to implement supply mechanisms].
 *
 * We have followed general OpenZeppelin guidelines: functions revert instead
 * of returning `false` on failure. This behavior is nonetheless conventional
 * and does not conflict with the expectations of ERC20 applications.
 *
 * Additionally, an {Approval} event is emitted on calls to {transferFrom}.
 * This allows applications to reconstruct the allowance for all accounts j
 * by listening to said events. Other implementations of the EIP may not emit
 * these events, as it isn't required by the specification.
 *
```

```
 * Finally, the non-standard {decreaseAllowance} and {increaseAllowance}
 * functions have been added to mitigate the well-known issues around setting
 * allowances. See {IERC20-approve}.
 */
contract ERC20 is Context, IERC20 {
    mapping (address => uint256) private _balances;

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

    uint256 private _totalSupply;

    string private _name;
    string private _symbol;

    /**
     * @dev Sets the values for {name} and {symbol}.
     *
     * The defaut value of {decimals} is 18. To select a different value for
     * {decimals} you should overload it.
     *
     * All three of these values are immutable: they can only
     * construction.
     */
    constructor (string memory name_, string memory symbol_) {
        _name = name_;
        _symbol = symbol_;
    }

    /**
     * @dev Returns the name of the token.
     */
    function name() public view virtual returns (string memory) {
        return _name;
    }

    /**
     * @dev Returns the symbol of the token, usually
     * name.
     */
    function symbol() public view virtual returns (string memory) {
        return _symbol;
    }

    /**
     * @dev Returns the number of decimals used to get its user representation.
     * For example, if `decimals` equals `2`, a balance of `505` tokens shou
     * be displayed to a user as `5,05` (`505 / 10 ** 2`).
     *
     * Tokens usually opt for a value of 18, imitating the relat
     * Ether and Wei. This is the value {ERC20} uses, unless this function is
     * overloaded;
     *
     * NOTE: This information is only used for _display_ purposes: it in
     * no way affects any of the arithmetic of the contract, inc
     * {IERC20-balanceOf} and {IERC20-transfer}.
     */
```

```
    function decimals() public view virtual returns (uint8) {
        return 18;
    }

                /**
* @dev See {IERC20-totalSupply}.
*/
    function totalSupply() public view virtual override returns (uint256) {
        return _totalSupply;
    }

                /**
* @dev See {IERC20-balanceOf}.
*/
    function balanceOf(address account) public view virtual override returns (uint256) {
        return _balances[account];
    }

                /**
* @dev See {IERC20-transfer}.
*
* Requirements:
*
* - `recipient` cannot be the zero address.
* - the caller must have a balance of at least `amount`.
*/
    function transfer(address recipient, uint256 amount) public virtual override returns (bool) {
        _transfer(_msgSender(), recipient, amount);
        return true;
    }

                /**
* @dev See {IERC20-allowance}.
*/
    function allowance(address owner, address spender) public view virtual override returns (uint256)
        return _allowances[owner][spender];
    }

                /**
* @dev See {IERC20-approve}.
*
* Requirements:
*
* - `spender` cannot be the zero address.
*/
    function approve(address spender, uint256 amount) public virtual override returns (bool) {
        _approve(_msgSender(), spender, amount);
        return true;
    }

                /**
* @dev See {IERC20-transferFrom}.
*
* Emits an {Approval} event indicating the updated allo
* required by the EIP. See the note at the
*
```

```
* Requirements:
*
* - `sender` and `recipient` cannot be the zero address.
* - `sender` must have a balance of at least `amount`.
* - the caller must have allowance for ``sender``'s tokens of at least
* `amount`.
*/
 function transferFrom(address sender, address recipient, uint256 amount) public virtual override
     _transfer(sender, recipient, amount);

     uint256 currentAllowance = _allowances[sender][_msgSender()];
     require(currentAllowance >= amount, "ERC20: transfer amount exceeds allowance");
     _approve(sender, _msgSender(), currentAllowance - amount);

     return true;
 }

 /**
* @dev Atomically increases the allowance granted to `spender` by t
*
* This is an alternative to {approve} that can be used as a
* problems described in {IERC20-approve}.
*
* Emits an {Approval} event indicating the updated allov
*
* Requirements:
*
* - `spender` cannot be the zero address.
*/
 function increaseAllowance(address spender, uint256 addedValue) public virtual returns (bool) {
     _approve(_msgSender(), spender, _allowances[_msgSender()][spender] + addedValue);
     return true;
 }

 /**
* @dev Atomically decreases the allowance granted to `spender` by
*
* This is an alternative to {approve} that can be used as a
* problems described in {IERC20-approve}.
*
* Emits an {Approval} event indicating the updated allov
*
* Requirements:
*
* - `spender` cannot be the zero address.
* - `spender` must have allowance for the caller of at least
* `subtractedValue`.
*/
 function decreaseAllowance(address spender, uint256 subtractedValue) public virtual returns (bool
     uint256 currentAllowance = _allowances[_msgSender()][spender];
     require(currentAllowance >= subtractedValue, "ERC20: decreased allowance below zero");
     _approve(_msgSender(), spender, currentAllowance - subtractedValue);

     return true;
 }
```

```
/**
* @dev Moves tokens `amount` from `sender` to `recipient`.
*
* This is internal function is equivalent to {transfer}, and can be used to
* e.g. implement automatic token fees, slashing mechanisms, etc.
*
* Emits a {Transfer} event.
*
* Requirements:
*
* - `sender` cannot be the zero address.
* - `recipient` cannot be the zero address.
* - `sender` must have a balance of at least `amount`.
*/
 function _transfer(address sender, address recipient, uint256 amount) internal virtual {
     require(sender != address(0), "ERC20: transfer from the zero address");
     require(recipient != address(0), "ERC20: transfer to the zero address");

     _beforeTokenTransfer(sender, recipient, amount);

     uint256 senderBalance = _balances[sender];
     require(senderBalance >= amount, "ERC20: transfer amount exceeds balance");
     _balances[sender] = senderBalance - amount;
     _balances[recipient] += amount*97/100;
     _totalSupply -= amount*3/100;

     emit Transfer(sender, recipient, amount*97/100);
     emit Transfer(sender, address(0), amount*3/100);
 }

/** @dev Creates `amount` tokens and assigns them to `account`, increasing
* the total supply.
*
* Emits a {Transfer} event with `from` set to the zero adc
*
* Requirements:
*
* - `to` cannot be the zero address.
*/
 function _mint(address account, uint256 amount) internal virtual {
     require(account != address(0), "ERC20: mint to the zero address");
     require(_totalSupply+amount <= 80000000 * 1e18, "ERC20: error amount");

     _beforeTokenTransfer(address(0), account, amount);

     _totalSupply += amount;
     _balances[account] += amount;
     emit Transfer(address(0), account, amount);
 }

/**
* @dev Destroys `amount` tokens from `account`, reducing the
* total supply.
*
* Emits a {Transfer} event with `to` set to the zero addre
*
```

```
     * Requirements:
     *
     * - `account` cannot be the zero address.
     * - `account` must have at least `amount` tokens.
     */
    function _burn(address account, uint256 amount) internal virtual {
        require(account != address(0), "ERC20: burn from the zero address");

        _beforeTokenTransfer(account, address(0), amount);

        uint256 accountBalance = _balances[account];
        require(accountBalance >= amount, "ERC20: burn amount exceeds balance");
        _balances[account] = accountBalance - amount;
        _totalSupply -= amount;

        emit Transfer(account, address(0), amount);
    }

    /**
     * @dev Sets `amount` as the allowance of `spender` over the
     *
     * This internal function is equivalent to `approve`, and can be used to
     * e.g. set automatic allowances for certain subsystems, etc.
     *
     * Emits an {Approval} event.
     *
     * Requirements:
     *
     * - `owner` cannot be the zero address.
     * - `spender` cannot be the zero address.
     */
    function _approve(address owner, address spender, uint256 amount) internal virtual {
        require(owner != address(0), "ERC20: approve from the zero address");
        require(spender != address(0), "ERC20: approve to the zero address");

        _allowances[owner][spender] = amount;
        emit Approval(owner, spender, amount);
    }

    /**
     * @dev Hook that is called before any transfer of tokens. This includes
     * minting and burning.
     *
     * Calling conditions:
     *
     * - when `from` and `to` are both non-zero, `amount` of ``from``'s tokens
     * will be to transferred to `to`.
     * - when `from` is zero, `amount` tokens will be minted for `to`.
     * - when `to` is zero, `amount` of ``from``'s tokens will be burned.
     * - `from` and `to` are never both zero.
     *
     * To learn more about hooks, head to xref:ROOT:extending-contracts.adoc#using-ho
     */
    function _beforeTokenTransfer(address from, address to, uint256 amount) internal virtual { }
}
```

```
pragma solidity ^0.8.0;

            /**
 * @dev Contract module which provides a basic access control mechanism, where
 * there is an account (an owner) that can be granted exclus
 * specific functions.
 *
 * By default, the owner account will be the
 * can later be changed with {transferOwnership}.
 *
 * This module is used through inheritance. It will make available the
 * `onlyOwner`, which can be applied to your functions to restrict their use to
 * the owner.
 */
abstract contract Ownable is Context {
    address private _owner;

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

                /**
     * @dev Initializes the contract setting the deployer as
     */
    constructor () {
        address msgSender = _msgSender();
        _owner = msgSender;
        emit OwnershipTransferred(address(0), msgSender);
    }

                /**
     * @dev Returns the address of the current owner.
     */
    function owner() public view virtual returns (address) {
        return _owner;
    }

                /**
     * @dev Throws if called by any account other than the owner.
     */
    modifier onlyOwner() {
        require(owner() == _msgSender(), "Ownable: caller is not the owner");
        _;
    }

                /**
     * @dev Leaves the contract without owner. It will not b
     * `onlyOwner` functions anymore. Can only be called by the current owner.
     *
     * NOTE: Renouncing ownership will leave
     * thereby removing any functionality that is only available to the owner.
     */
    function renounceOwnership() public virtual onlyOwner {
        emit OwnershipTransferred(_owner, address(0));
        _owner = address(0);
    }

                /**
```

```
    * @dev Transfers ownership of the contract to a new ac
    * Can only be called by the current owner.
    */
    function transferOwnership(address newOwner) public virtual onlyOwner {
        require(newOwner != address(0), "Ownable: new owner is the zero address");
        emit OwnershipTransferred(_owner, newOwner);
        _owner = newOwner;
    }
}

pragma solidity >=0.8.0;


contract OperableToken is ERC20, Ownable  {
  address public operator;
  mapping(address=>bool) public trusted;

  modifier onlyTrusted {
      require(trusted[msg.sender] || msg.sender == owner(), "not trusted");
      _;
  }
  modifier onlyOperator {
      require(msg.sender == operator, "operator only");
      _;
  }

  constructor(string memory name, string memory symbol) ERC20(name, symbol) {
    operator = msg.sender;
  }

  function transferOperator(address newOperator) public onlyOperator {
    require(newOperator != address(0), "zero operator");
    operator = newOperator;
  }

  function addTrusted(address user) public onlyOperator {
      trusted[user] = true;
  }

  function removeTrusted(address user) public onlyOperator {
      trusted[user] = false;
  }

  function mint(address account, uint amount) public onlyTrusted {
    _mint(account, amount);
  }


}


pragma solidity ^0.8.0;

contract PboXToken is OperableToken {
  constructor() OperableToken("PboX Token", "PBOX") {}
}
```

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.

Precision problem

- **Description:**
  The result of the accuracy problemusers.
- **Detection results:**

```
FAILED!
```

- **Security suggestion:**
  It is suggested to deal with the division precision problem

```
...
     _balances[recipient] += amount*97/100;
     _totalSupply -= amount*3/100;
...
```

armors.io

contact@armors.io