Zoo Staking

Smart Contract Audit

Zoo Staking Audit Summary

Project name : Zoo Staking Contract

Project address: None

Code URL : https://etherscan.io/address/0x24F28214de4b594b252041f8505593903CEB2F19#code

Commit : None

Project target : Zoo Staking Contract Audit

Blockchain : Ethereum

Test result : PASSED

Audit Info

Audit NO : 0X202106100016

Audit Team : Armors Labs

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

Zoo Staking Audit

The Zoo Staking team asked us to review and audit their Zoo Staking 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
Zoo Staking Audit	Rock, Sophia, Rushairer, Rico, David, Alice	1.0.0	2021-06-10

Audit results

ZOO Staking pools produce ZOO on an hourly basis, and the user's Staking pledge needs to be greater than or equal to one hour to produce ZOO. The ZOO pool uses a batch mining pool weighting mechanism, with an overall weight of 100% for each phase of the pool, with different weights and different amounts of ZOO output.

Staking output mechanism: Single currency individual pledge / network-wide pledge x weight x (current time - last operation time) x output per hour

If the token of the reward is taken out, the user can only get back the principal.

Note that as of the date of publishing, the above review reflects the current understanding of known security patterns as they relate to the Zoo Staking 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
Zoo.sol	9eaee94431fe102b3cf9fd8683a29bb3
Address.sol	7dff26ff266d4e8e71dd4cf036454f5d
Context.sol	b74d20930fbde7bbe062f0149154391e
Ownable.sol	7b46ccfd1c4d6ce90738954ed0d84c2b
SafeBEP20.sol	f77f5da43fe886b654f7d2dde2c18127
BEP20.sol	87682ec24de45003579b201cecefb9e5
IBEP20.sol	82cec5f9600e2ef3b6c376230ed62f78
SafeMath.sol	4e9fb7b172262977ad8256330baa51f2
ZooToken.sol	7851dd1a256c168b89829d48206a1f73

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

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

Contract file

zoo.sol

```
// SPDX-License-Identifier: MIT

pragma solidity 0.6.12;

//
//
//                    ┌─┐   ┌─┐ + +
//                  ┌─┘ └───┘ └─┐++
//                  │           │
//                  │     ─     │++ + + +
//                  ████████─████████ │+
//                  │           │+
//                  │     ┴─    │
//                  │           │
//                  └─┐       ┌─┘
//                    │       │
//                    │   + + │
//                    │       │
//                    │       └─────────┐
//                    │                 │
//                    │                 ├─┐
//                    │                 └─┐
//                    │                   │
//                    └─┐ ┐ ┌─────┬─┐ ┌─┘   + + + +
//                    │ ┤ ┤     │ ┤ ┤
//                    └─┴─┘     └─┴─┘   + + + +
//                  神兽保佑
//                  代码无BUG!

import "./SafeMath.sol";
import "./IBEP20.sol";
```

```
import "./SafeBEP20.sol";
import "./Ownable.sol";

import "./ZooToken.sol";

// MasterChef is the master of Lyptus. He can make Lyptus and he is a fair guy.
//
// Note that it's ownable and the owner wields tremendous power. The ownership
// will be transferred to a governance smart contract once LYPTUS is sufficiently
// distributed and the community can show to govern itself.
//
// Have fun reading it. Hopefully it's bug-free. God bless.
contract Zoo is Ownable {
    using SafeMath for uint256;
    using SafeBEP20 for IBEP20;

    // Info of each user.
    struct UserInfo {
        uint256 stakeAmount;         // How many LP tokens the user has provided.
        uint256 balance;
        uint256 pledgeTime;
        bool isExist;
    }

    // Info of each pool.
    struct PoolInfo {
        IBEP20 poolToken;           // Address of LP token contract.
        uint256 zooRewardRate;
        uint256 totalStakeAmount;
        uint256 openTime;
        bool isOpen;
    }


    // The ZOO TOKEN!
    ZooToken public zoo;

    // Info of each pool.
    PoolInfo[] public poolInfo;

    mapping (uint256 => mapping (address => UserInfo)) public userInfo;

    event Stake(address indexed user, uint256 indexed pid, uint256 amount);
    event CancelStake(address indexed user, uint256 indexed pid, uint256 amount);
    event Withdraw(address indexed user, uint256 indexed pid, uint256 amount);

    constructor(
        address _zooTokenAddress
    ) public {
        zoo = ZooToken(_zooTokenAddress);
    }

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

    // Add a new lp to the pool. Can only be called by the owner.
    // XXX DO NOT add the same LP token more than once. Rewards will be messed up if you do.
    function addPool(address _poolAddress, uint256 _zooRewardRate, uint256 _openTime, bool _isOpen) p
        IBEP20 _poolToken = IBEP20(_poolAddress);
        poolInfo.push(PoolInfo({
        poolToken: _poolToken,
        zooRewardRate: _zooRewardRate,
        totalStakeAmount: 0,
        openTime: _openTime,
        isOpen: _isOpen
```

```
        }));
    }

    function updatePool(uint256 _pid, uint256 _zooRewardRate, uint256 _openTime, bool _isOpen) public
        poolInfo[_pid].zooRewardRate = _zooRewardRate;
        poolInfo[_pid].openTime = _openTime;
        poolInfo[_pid].isOpen = _isOpen;
    }


    function addUser(uint256 _pid, uint256 _amount) private {
        userInfo[_pid][msg.sender] = UserInfo(
            _amount,
            0,
            block.timestamp,
            true
        );
    }

    function stake(uint256 _pid, uint256 _amount) public {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][msg.sender];
        pool.poolToken.transferFrom(address(msg.sender), address(this), _amount);

        if(!user.isExist){
            addUser(_pid, _amount);
        }else{
            user.stakeAmount = user.stakeAmount.add(_amount);
            uint256 profit = getUserProfit(_pid, false);

            if (profit > 0) {
                user.balance = user.balance.add(profit);
            }

            user.pledgeTime = block.timestamp;
        }

        pool.totalStakeAmount = pool.totalStakeAmount.add(_amount);

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

    function cancelStake(uint256 _pid) public {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][msg.sender];

        require( user.isExist && user.stakeAmount > 0,"user not exist or no profit")  ;


        uint256 stakeAmount = user.stakeAmount;
        uint256 profitAmount = getUserProfit(_pid, true);

        user.stakeAmount = 0;
        user.balance = 0;
        pool.totalStakeAmount = pool.totalStakeAmount.sub(stakeAmount);

        pool.poolToken.safeTransfer(address(msg.sender), stakeAmount);

        if (profitAmount > 0) {
            safeZooTransfer(address(msg.sender), profitAmount);
        }

        emit CancelStake(msg.sender, _pid, stakeAmount);
    }

    function withdraw(uint256 _pid) public {
```

```
        uint256 profitAmount = getUserProfit(_pid, true);
        require(profitAmount > 0,"profit must gt 0");
        UserInfo storage user = userInfo[_pid][msg.sender];
        user.pledgeTime = block.timestamp;
        user.balance = 0;
        safeZooTransfer(address(msg.sender), profitAmount);
        emit Withdraw(msg.sender, _pid, profitAmount);
    }


    function getUserProfit(uint256 _pid, bool _withBalance) private view returns (uint256) {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][msg.sender];

        uint256 profit = 0;

        if (user.stakeAmount > 0) {
            uint256 totalStakeAmount = pool.totalStakeAmount;
            if (totalStakeAmount > 0) {
                uint256 time = block.timestamp;
                uint256 hour = time.sub(user.pledgeTime).div(3600);

                if (hour >= 1) {
                    uint256 rate = user.stakeAmount.mul(1e18).div(totalStakeAmount);
                    uint256 profitAmount = rate.mul(pool.zooRewardRate).mul(hour).div(1e18);
                    if (profitAmount > 0) {
                        profit = profit.add(profitAmount);
                    }
                }
            }
        }

        if (_withBalance) {
            profit = profit.add(user.balance);
        }

        return profit;
    }

    function getProfit(uint256 _pid) public view returns (uint256) {
        uint256 profit = getUserProfit(_pid, true);
        return profit;
    }

    function getPoolStake(uint256 _pid) public view returns (uint256) {
        PoolInfo memory pool = poolInfo[_pid];
        return pool.totalStakeAmount;
    }

    function getUserStake(uint256 _pid) public view returns (uint256){
        UserInfo storage user = userInfo[_pid][msg.sender];
        return user.stakeAmount;
    }

    function safeZooTransfer(address _to, uint256 _amount) internal {
        uint256 zooBalance = zoo.balanceOf(address(this));
        if (_amount > zooBalance) {
            zoo.transfer(_to, zooBalance);
        } else {
            zoo.transfer(_to, _amount);
        }
    }
}
```

Address.sol

```
// SPDX-License-Identifier: MIT

pragma solidity >=0.6.2 <0.8.0;

/**
 * @dev Collection of functions related to the address type
 */
library Address {
    /**
     * @dev Returns true if `account` is a contract.
     *
     * [IMPORTANT]
     * ====
     * It is unsafe to assume that an address for which this function returns
     * false is an externally-owned account (EOA) and not a contract.
     *
     * Among others, `isContract` will return false for the following
     * types of addresses:
     *
     *  - an externally-owned account
     *  - a contract in construction
     *  - an address where a contract will be created
     *  - an address where a contract lived, but was destroyed
     * ====
     */
    function isContract(address account) internal view returns (bool) {
        // This method relies on extcodesize, which returns 0 for contracts in
        // construction, since the code is only stored at the end of the
        // constructor execution.

        uint256 size;
        // solhint-disable-next-line no-inline-assembly
        assembly { size := extcodesize(account) }
        return size > 0;
    }

    /**
     * @dev Replacement for Solidity's `transfer`: sends `amount` wei to
     * `recipient`, forwarding all available gas and reverting on errors.
     *
     * https://eips.ethereum.org/EIPS/eip-1884[EIP1884] increases the gas cost
     * of certain opcodes, possibly making contracts go over the 2300 gas limit
     * imposed by `transfer`, making them unable to receive funds via
     * `transfer`. {sendValue} removes this limitation.
     *
     * https://diligence.consensys.net/posts/2019/09/stop-using-soliditys-transfer-now/[Learn more].
     *
     * IMPORTANT: because control is transferred to `recipient`, care must be
     * taken to not create reentrancy vulnerabilities. Consider using
     * {ReentrancyGuard} or the
     * https://solidity.readthedocs.io/en/v0.5.11/security-considerations.html#use-the-checks-effects
     */
    function sendValue(address payable recipient, uint256 amount) internal {
        require(address(this).balance >= amount, "Address: insufficient balance");

        // solhint-disable-next-line avoid-low-level-calls, avoid-call-value
        (bool success, ) = recipient.call{ value: amount }("");
        require(success, "Address: unable to send value, recipient may have reverted");
    }

    /**
     * @dev Performs a Solidity function call using a low level `call`. A
     * plain`call` is an unsafe replacement for a function call: use this
```

```
     * function instead.
     *
     * If `target` reverts with a revert reason, it is bubbled up by this
     * function (like regular Solidity function calls).
     *
     * Returns the raw returned data. To convert to the expected return value,
     * use https://solidity.readthedocs.io/en/latest/units-and-global-variables.html?highlight=abi.de
     *
     * Requirements:
     *
     * - `target` must be a contract.
     * - calling `target` with `data` must not revert.
     *
     * _Available since v3.1._
     */
    function functionCall(address target, bytes memory data) internal returns (bytes memory) {
      return functionCall(target, data, "Address: low-level call failed");
    }

    /**
     * @dev Same as {xref-Address-functionCall-address-bytes-}[`functionCall`], but with
     * `errorMessage` as a fallback revert reason when `target` reverts.
     *
     * _Available since v3.1._
     */
    function functionCall(address target, bytes memory data, string memory errorMessage) internal ret
        return functionCallWithValue(target, data, 0, errorMessage);
    }

    /**
     * @dev Same as {xref-Address-functionCall-address-bytes-}[`functionCall`],
     * but also transferring `value` wei to `target`.
     *
     * Requirements:
     *
     * - the calling contract must have an ETH balance of at least `value`.
     * - the called Solidity function must be `payable`.
     *
     * _Available since v3.1._
     */
    function functionCallWithValue(address target, bytes memory data, uint256 value) internal returns
        return functionCallWithValue(target, data, value, "Address: low-level call with value failed"
    }

    /**
     * @dev Same as {xref-Address-functionCallWithValue-address-bytes-uint256-}[`functionCallWithValu
     * with `errorMessage` as a fallback revert reason when `target` reverts.
     *
     * _Available since v3.1._
     */
    function functionCallWithValue(address target, bytes memory data, uint256 value, string memory er
        require(address(this).balance >= value, "Address: insufficient balance for call");
        require(isContract(target), "Address: call to non-contract");

        // solhint-disable-next-line avoid-low-level-calls
        (bool success, bytes memory returndata) = target.call{ value: value }(data);
        return _verifyCallResult(success, returndata, errorMessage);
    }

    /**
     * @dev Same as {xref-Address-functionCall-address-bytes-}[`functionCall`],
     * but performing a static call.
     *
     * _Available since v3.3._
     */
    function functionStaticCall(address target, bytes memory data) internal view returns (bytes memor
```

```
        return functionStaticCall(target, data, "Address: low-level static call failed");
    }

    /**
     * @dev Same as {xref-Address-functionCall-address-bytes-string-}[`functionCall`],
     * but performing a static call.
     *
     * _Available since v3.3._
     */
    function functionStaticCall(address target, bytes memory data, string memory errorMessage) intern
        require(isContract(target), "Address: static call to non-contract");

        // solhint-disable-next-line avoid-low-level-calls
        (bool success, bytes memory returndata) = target.staticcall(data);
        return _verifyCallResult(success, returndata, errorMessage);
    }

    function _verifyCallResult(bool success, bytes memory returndata, string memory errorMessage) pri
        if (success) {
            return returndata;
        } else {
            // Look for revert reason and bubble it up if present
            if (returndata.length > 0) {
                // The easiest way to bubble the revert reason is using memory via assembly

                // solhint-disable-next-line no-inline-assembly
                assembly {
                    let returndata_size := mload(returndata)
                    revert(add(32, returndata), returndata_size)
                }
            } else {
                revert(errorMessage);
            }
        }
    }
}
```

BEP20.sol

```
// SPDX-License-Identifier: MIT

pragma solidity >=0.4.0;

import './Ownable.sol';
import './Context.sol';
import './IBEP20.sol';
import './SafeMath.sol';

/**
 * @dev Implementation of the {IBEP20} interface.
 *
 * This implementation is agnostic to the way tokens are created. This means
 * that a supply mechanism has to be added in a derived contract using {_mint}.
 * For a generic mechanism see {BEP20PresetMinterPauser}.
 *
 * TIP: For a detailed writeup see our guide
 * https://forum.zeppelin.solutions/t/how-to-implement-BEP20-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 BEP20 applications.
 *
```

```
 * Additionally, an {Approval} event is emitted on calls to {transferFrom}.
 * This allows applications to reconstruct the allowance for all accounts just
 * 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 {IBEP20-approve}.
 */
contract BEP20 is Context, IBEP20, Ownable {
    using SafeMath for uint256;

    mapping(address => uint256) private _balances;

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

    uint256 private _totalSupply;

    string private _name;
    string private _symbol;
    uint8 private _decimals;

    /**
     * @dev Sets the values for {name} and {symbol}, initializes {decimals} with
     * a default value of 18.
     *
     * To select a different value for {decimals}, use {_setupDecimals}.
     *
     * All three of these values are immutable: they can only be set once during
     * construction.
     */
    constructor(string memory name, string memory symbol, uint256 totalSupply) public {
        _name = name;
        _symbol = symbol;
        _totalSupply = totalSupply;
        _decimals = 18;
         _balances[msg.sender] = _totalSupply;
        emit Transfer(address(0), msg.sender, _totalSupply);
    }

    /**
     * @dev Returns the bep token owner.
     */
    function getOwner() external override view returns (address) {
        return owner();
    }

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

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

    /**
    * @dev Returns the number of decimals used to get its user representation.
    */
    function decimals() public override view returns (uint8) {
```

```
        return _decimals;
    }

    /**
     * @dev See {BEP20-totalSupply}.
     */
    function totalSupply() public override view returns (uint256) {
        return _totalSupply;
    }

    /**
     * @dev See {BEP20-balanceOf}.
     */
    function balanceOf(address account) public override view returns (uint256) {
        return _balances[account];
    }

    /**
     * @dev See {BEP20-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 override returns (bool) {
        _transfer(_msgSender(), recipient, amount);
        return true;
    }

    /**
     * @dev See {BEP20-allowance}.
     */
    function allowance(address owner, address spender) public override view returns (uint256) {
        return _allowances[owner][spender];
    }

    /**
     * @dev See {BEP20-approve}.
     *
     * Requirements:
     *
     * - `spender` cannot be the zero address.
     */
    function approve(address spender, uint256 amount) public override returns (bool) {
        _approve(_msgSender(), spender, amount);
        return true;
    }

    /**
     * @dev See {BEP20-transferFrom}.
     *
     * Emits an {Approval} event indicating the updated allowance. This is not
     * required by the EIP. See the note at the beginning of {BEP20};
     *
     * 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 override returns
        _transfer(sender, recipient, amount);
        _approve(
            sender,
            _msgSender(),
```

```
            _allowances[sender][_msgSender()].sub(amount)
        );
        return true;
    }

    /**
     * @dev Atomically increases the allowance granted to `spender` by the caller.
     *
     * This is an alternative to {approve} that can be used as a mitigation for
     * problems described in {BEP20-approve}.
     *
     * Emits an {Approval} event indicating the updated allowance.
     *
     * Requirements:
     *
     * - `spender` cannot be the zero address.
     */
    function increaseAllowance(address spender, uint256 addedValue) public returns (bool) {
        _approve(_msgSender(), spender, _allowances[_msgSender()][spender].add(addedValue));
        return true;
    }

    /**
     * @dev Atomically decreases the allowance granted to `spender` by the caller.
     *
     * This is an alternative to {approve} that can be used as a mitigation for
     * problems described in {BEP20-approve}.
     *
     * Emits an {Approval} event indicating the updated allowance.
     *
     * 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 returns (bool) {
        _approve(_msgSender(), spender, _allowances[_msgSender()][spender].sub(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 {
        require(sender != address(0), 'BEP20: transfer from the zero address');
        require(recipient != address(0), 'BEP20: transfer to the zero address');

        _balances[sender] = _balances[sender].sub(amount);
        _balances[recipient] = _balances[recipient].add(amount);
        emit Transfer(sender, recipient, amount);
    }
```

```
    /**
     * @dev Destroys `amount` tokens from `account`, reducing the
     * total supply.
     *
     * Emits a {Transfer} event with `to` set to the zero address.
     *
     * Requirements
     *
     * - `account` cannot be the zero address.
     * - `account` must have at least `amount` tokens.
     */
    function _burn(address account, uint256 amount) internal {
        require(account != address(0), 'BEP20: burn from the zero address');

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

    /**
     * @dev Sets `amount` as the allowance of `spender` over the `owner`s tokens.
     *
     * This is 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 {
        require(owner != address(0), 'BEP20: approve from the zero address');
        require(spender != address(0), 'BEP20: approve to the zero address');

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

    /**
     * @dev Destroys `amount` tokens from `account`.`amount` is then deducted
     * from the caller's allowance.
     *
     * See {_burn} and {_approve}.
     */
    function _burnFrom(address account, uint256 amount) internal {
        _burn(account, amount);
        _approve(account, _msgSender(), _allowances[account][_msgSender()].sub(amount));
    }
}
```

Context.sol

```
// SPDX-License-Identifier: MIT

pragma solidity >=0.6.0 <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 GSN 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 payable) {
        return msg.sender;
    }

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

IBEP20.sol

```
// SPDX-License-Identifier: MIT

pragma solidity >=0.6.4;

interface IBEP20 {
    /**
     * @dev Returns the amount of tokens in existence.
     */
    function totalSupply() external view returns (uint256);

    /**
     * @dev Returns the token decimals.
     */
    function decimals() external view returns (uint8);

    /**
     * @dev Returns the token symbol.
     */
    function symbol() external view returns (string memory);

    /**
     * @dev Returns the token name.
     */
    function name() external view returns (string memory);

    /**
     * @dev Returns the bep token owner.
     */
    function getOwner() external view returns (address);

    /**
     * @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 operation succeeded.
     *
     * Emits a {Transfer} event.
     */
    function transfer(address recipient, uint256 amount) external returns (bool);

    /**
```

```
     * @dev Returns the remaining number of tokens that `spender` will be
     * 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 caller's tokens.
     *
     * Returns a boolean value indicating whether the operation succeeded.
     *
     * IMPORTANT: Beware that changing an allowance with this method brings the risk
     * that someone may use both the old and the new allowance by unfortunate
     * transaction ordering. One possible solution to mitigate this race
     * condition is to first reduce the spender's allowance to 0 and set the
     * desired value afterwards:
     * https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
     *
     * 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 operation succeeded.
     *
     * 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 an `owner` is set by
     * a call to {approve}. `value` is the new allowance.
     */
    event Approval(address indexed owner, address indexed spender, uint256 value);
}
```

Ownable.sol

```
// SPDX-License-Identifier: MIT

pragma solidity >=0.6.0 <0.8.0;

import "./Context.sol";
/**
 * @dev Contract module which provides a basic access control mechanism, where
 * there is an account (an owner) that can be granted exclusive access to
 * specific functions.
 *
 * By default, the owner account will be the one that deploys the contract. This
```

```
 * can later be changed with {transferOwnership}.
 *
 * This module is used through inheritance. It will make available the modifier
 * `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 the initial owner.
     */
    constructor () internal {
        address msgSender = _msgSender();
        _owner = msgSender;
        emit OwnershipTransferred(address(0), msgSender);
    }

    /**
     * @dev Returns the address of the current owner.
     */
    function owner() public view 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 be possible to call
     * `onlyOwner` functions anymore. Can only be called by the current owner.
     *
     * NOTE: Renouncing ownership will leave the contract without an owner,
     * 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 account (`newOwner`).
     * 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;
    }
}
```

SafeBEP20.sol

```
// SPDX-License-Identifier: MIT

pragma solidity >=0.6.0 <0.8.0;
```

```
import "./IBEP20.sol";
import "./SafeMath.sol";
import "./Address.sol";

/**
 * @title SafeBEP20
 * @dev Wrappers around BEP20 operations that throw on failure (when the token
 * contract returns false). Tokens that return no value (and instead revert or
 * throw on failure) are also supported, non-reverting calls are assumed to be
 * successful.
 * To use this library you can add a `using SafeBEP20 for IBEP20;` statement to your contract,
 * which allows you to call the safe operations as `token.safeTransfer(...)`, etc.
 */
library SafeBEP20 {
    using SafeMath for uint256;
    using Address for address;

    function safeTransfer(IBEP20 token, address to, uint256 value) internal {
        _callOptionalReturn(token, abi.encodeWithSelector(token.transfer.selector, to, value));
    }

    function safeTransferFrom(IBEP20 token, address from, address to, uint256 value) internal {
        _callOptionalReturn(token, abi.encodeWithSelector(token.transferFrom.selector, from, to, valu
    }

    /**
     * @dev Deprecated. This function has issues similar to the ones found in
     * {IBEP20-approve}, and its usage is discouraged.
     *
     * Whenever possible, use {safeIncreaseAllowance} and
     * {safeDecreaseAllowance} instead.
     */
    function safeApprove(IBEP20 token, address spender, uint256 value) internal {
        // safeApprove should only be called when setting an initial allowance,
        // or when resetting it to zero. To increase and decrease it, use
        // 'safeIncreaseAllowance' and 'safeDecreaseAllowance'
        // solhint-disable-next-line max-line-length
        require((value == 0) || (token.allowance(address(this), spender) == 0),
            "SafeBEP20: approve from non-zero to non-zero allowance"
        );
        _callOptionalReturn(token, abi.encodeWithSelector(token.approve.selector, spender, value));
    }

    function safeIncreaseAllowance(IBEP20 token, address spender, uint256 value) internal {
        uint256 newAllowance = token.allowance(address(this), spender).add(value);
        _callOptionalReturn(token, abi.encodeWithSelector(token.approve.selector, spender, newAllowan
    }

    function safeDecreaseAllowance(IBEP20 token, address spender, uint256 value) internal {
        uint256 newAllowance = token.allowance(address(this), spender).sub(value);
        _callOptionalReturn(token, abi.encodeWithSelector(token.approve.selector, spender, newAllowan
    }

    /**
     * @dev Imitates a Solidity high-level call (i.e. a regular function call to a contract), relaxin
     * on the return value: the return value is optional (but if data is returned, it must not be fal
     * @param token The token targeted by the call.
     * @param data The call data (encoded using abi.encode or one of its variants).
     */
    function _callOptionalReturn(IBEP20 token, bytes memory data) private {
        // We need to perform a low level call here, to bypass Solidity's return data size checking m
        // we're implementing it ourselves. We use {Address.functionCall} to perform this call, which
        // the target address contains contract code and also asserts for success in the low-level ca

        bytes memory returndata = address(token).functionCall(data, "SafeBEP20: low-level call failed
        if (returndata.length > 0) { // Return data is optional
```

```
            // solhint-disable-next-line max-line-length
            require(abi.decode(returndata, (bool)), "SafeBEP20: BEP20 operation did not succeed");
        }
    }
}
```

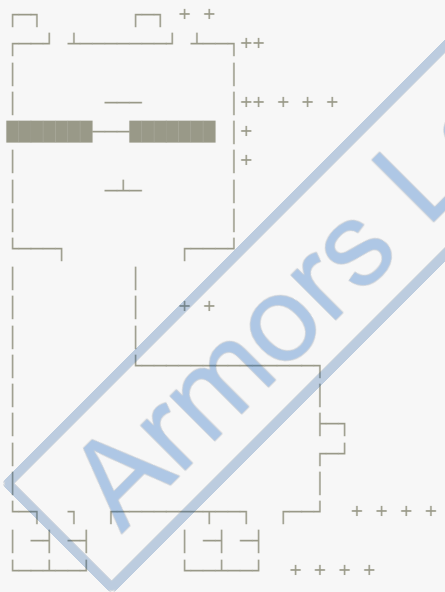
SafeMath.sol

```
// SPDX-License-Identifier: MIT

pragma solidity 0.6.12;

/**
 * @title SafeMath
 * @dev Unsigned math operations with safety checks that revert on error
 */

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

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

        return c;
    }

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

        return c;
    }

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

        return c;
    }

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

```
        return c;
    }

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

ZooToken.sol

```
// SPDX-License-Identifier: MIT

pragma solidity 0.6.12;

//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
//
```




```
import "./BEP20.sol";

contract ZooToken is BEP20('Zoo', 'ZOO', 100000000000000000000000000000) {


function getChainId() internal pure returns (uint) {
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.

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