



Armors Labs

Lovelace(LACE) Token

Smart Contract Audit

- Lovelace(LACE) Token Audit Summary
- Lovelace(LACE) Token Audit
 - Document information
 - Audit results
 - Audited target file
 - Vulnerability analysis
 - Vulnerability distribution
 - Summary of audit results
 - Contract file
 - Analysis of audit results
 - Re-Entrancy
 - Arithmetic Over/Under Flows
 - Unexpected Blockchain Currency
 - Delegatecall
 - Default Visibilities
 - Entropy Illusion
 - External Contract Referencing
 - Unsolved TODO comments
 - Short Address/Parameter Attack
 - Unchecked CALL Return Values
 - Race Conditions / Front Running
 - Denial Of Service (DOS)
 - Block Timestamp Manipulation
 - Constructors with Care
 - Unintialised Storage Pointers
 - Floating Points and Numerical Precision
 - tx.origin Authentication
 - Permission restrictions

Lovelace(LACE) Token Audit Summary

Project name : Lovelace(LACE) Token Contract

Project address: None

Code URL : <https://bscscan.com/address/0xA3499dd7dBBBD93CB0f8303f8a8AcE8D02508E73#code>

Commit : None

Project target : Lovelace(LACE) Token Contract Audit

Blockchain : Binance Smart Chain (BSC)

Test result : PASSED

Audit Info

Audit NO : 0X202111230016

Audit Team : Armors Labs

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

Lovelace(LACE) Token Audit

The Lovelace(LACE) Token team asked us to review and audit their Lovelace(LACE) 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
Lovelace(LACE) Token Audit	Rock, Sophia, Rushairer, Rico, David, Alice	1.0.0	2021-11-23

Audit results

Notic: There is a white list function in the contract. When the specified address is correctly set, the specified (_lgePairAddress) address transfer will trigger the function in the white list, and other users will not be affected

Note that as of the date of publishing, the above review reflects the current understanding of known security patterns as they relate to the Lovelace(LACE) 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
Lovelace(LACE) Token.sol	a49a6382e8d65d1f472cff9ad0663a40

Vulnerability analysis

Vulnerability distribution

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

Summary of audit results

Vulnerability	status
Re-Entrancy	safe
Arithmetic Over/Under Flows	safe
Unexpected Blockchain Currency	safe
Delegatecall	safe
Default Visibilities	safe
Entropy Illusion	safe
External Contract Referencing	safe
Short Address/Parameter Attack	safe
Unchecked CALL Return Values	safe
Race Conditions / Front Running	safe

Vulnerability	status
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

Lovelace(LACE) Token.sol

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

pragma solidity 0.8.9;

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);
}

/**
 * @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;
}
}

// File: @openzeppelin/contracts/access/Ownable.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 () {
        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 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;
    }
}

/**
 * @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 {
    /**
     * @dev Returns the addition of two unsigned integers, reverting on
     * overflow.
     *
     * Counterpart to Solidity's `+` operator.
     *
     * Requirements:
     * - Addition cannot overflow.
     */
    function add(uint256 a, uint256 b) internal pure returns (uint256) {
        uint256 c = a + b;
        require(c >= a, "SafeMath: addition overflow");

        return c;
    }

    /**
     * @dev Returns the subtraction of two unsigned integers, reverting on
     * overflow (when the result is negative).
     *
     * Counterpart to Solidity's `-` operator.
     *
     * Requirements:
     * - Subtraction cannot overflow.
     */
    function sub(uint256 a, uint256 b) internal pure returns (uint256) {
        return sub(a, b, "SafeMath: subtraction overflow");
    }

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

```



```

/**
 * @dev Returns the multiplication of two unsigned integers, reverting on
 * overflow.
 *
 * Counterpart to Solidity's `` operator.
 *
 * Requirements:
 * - Multiplication cannot 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-contracts/pull/522
    if (a == 0) {
        return 0;
    }

    uint256 c = a * b;
    require(c / a == b, "SafeMath: multiplication overflow");

    return c;
}

/**
 * @dev Returns the integer division of two unsigned integers. Reverts on
 * division by zero. The result is rounded towards zero.
 *
 * Counterpart to Solidity's `` operator. Note: this function uses a
 * `revert` opcode (which leaves remaining gas untouched) while Solidity
 * uses an invalid opcode to revert (consuming all remaining gas).
 *
 * Requirements:
 * - The divisor cannot be zero.
 */
function div(uint256 a, uint256 b) internal pure returns (uint256) {
    return div(a, b, "SafeMath: division by zero");
}

/**
 * @dev Returns the integer division of two unsigned integers. Reverts with custom message on
 * division by zero. The result is rounded towards zero.
 *
 * Counterpart to Solidity's `` operator. Note: this function uses a
 * `revert` opcode (which leaves remaining gas untouched) while Solidity
 * uses an invalid opcode to revert (consuming all remaining gas).
 *
 * Requirements:
 * - The divisor cannot be zero.
 */
function div(uint256 a, uint256 b, string memory errorMessage) internal pure returns (uint256) {
    // Solidity only automatically asserts when dividing by 0
    require(b > 0, errorMessage);
    uint256 c = a / b;
    // assert(a == b * c + a % b); // There is no case in which this doesn't hold

    return c;
}

/**
 * @dev Returns the remainder of dividing two unsigned integers. (unsigned integer modulo),
 * Reverts when dividing by zero.
 *
 * Counterpart to Solidity's `` operator. This function uses a `revert`
 * opcode (which leaves remaining gas untouched) while Solidity uses an
 * invalid opcode to revert (consuming all remaining gas).
 *
 */

```

```

* Requirements:
* - The divisor cannot be zero.
*/
function mod(uint256 a, uint256 b) internal pure returns (uint256) {
    return mod(a, b, "SafeMath: modulo by zero");
}

/**
 * @dev Returns the remainder of dividing two unsigned integers. (unsigned integer modulo),
 * Reverts with custom message when dividing by zero.
 *
 * Counterpart to Solidity's `%` operator. This function uses a `revert`
 * opcode (which leaves remaining gas untouched) while Solidity uses an
 * invalid opcode to revert (consuming all remaining gas).
 *
 * Requirements:
 * - The divisor cannot be zero.
 */
function mod(uint256 a, uint256 b, string memory errorMessage) internal pure returns (uint256) {
    require(b != 0, errorMessage);
    return a % b;
}
}

contract LGEWhitelisted is Context {
    struct WhitelistRound {
        uint256 duration;
        uint256 amountMax;
        mapping(address => bool) addresses;
        mapping(address => uint256) purchased;
    }

    WhitelistRound[] public _lgeWhitelistRounds;

    uint256 public _lgeTimestamp;
    address public _lgePairAddress;

    address public _whitelister;

    event WhitelisterTransferred(address indexed previousWhitelister, address indexed newWhitelister)

    constructor() {
        _whitelister = _msgSender();
    }

    modifier onlyWhitelister() {
        require(_whitelister == _msgSender(), "Caller is not the whitelister");
        _;
    }

    function renounceWhitelister() external onlyWhitelister {
        emit WhitelisterTransferred(_whitelister, address(0));
        _whitelister = address(0);
    }

    function transferWhitelister(address newWhitelister) external onlyWhitelister {
        _transferWhitelister(newWhitelister);
    }

    function _transferWhitelister(address newWhitelister) internal {
        require(newWhitelister != address(0), "New whitelister is the zero address");
        emit WhitelisterTransferred(_whitelister, newWhitelister);
        _whitelister = newWhitelister;
    }
}

```

```

/*
 * createLGEWhitelist - Call this after initial Token Generation Event (TGE)
 *
 * pairAddress - address generated from createPair() event on DEX
 * durations - array of durations (seconds) for each whitelist rounds
 * amountsMax - array of max amounts (TOKEN decimals) for each whitelist round
 *
 */

function createLGEWhitelist(
    address pairAddress,
    uint256[] calldata durations,
    uint256[] calldata amountsMax
) external onlyWhitelister() {
    require(durations.length == amountsMax.length, "Invalid whitelist(s)");

    _lgePairAddress = pairAddress;

    if (durations.length > 0) {
        delete _lgeWhitelistRounds;

        for (uint256 i = 0; i < durations.length; i++) {
            WhitelistRound storage whitelistRound = _lgeWhitelistRounds.push();
            whitelistRound.duration = durations[i];
            whitelistRound.amountMax = amountsMax[i];
        }
    }
}

/*
 * modifyLGEWhitelistAddresses - Define what addresses are included/excluded from a whitelist round
 *
 * index - 0-based index of round to modify whitelist
 * duration - period in seconds from LGE event or previous whitelist round
 * amountMax - max amount (TOKEN decimals) for each whitelist round
 *
 */
function modifyLGEWhitelist(
    uint256 index,
    uint256 duration,
    uint256 amountMax,
    address[] calldata addresses,
    bool enabled
) external onlyWhitelister() {
    require(index < _lgeWhitelistRounds.length, "Invalid index");
    require(amountMax > 0, "Invalid amountMax");

    if (duration != _lgeWhitelistRounds[index].duration) _lgeWhitelistRounds[index].duration = duration;

    if (amountMax != _lgeWhitelistRounds[index].amountMax) _lgeWhitelistRounds[index].amountMax = amountMax;

    for (uint256 i = 0; i < addresses.length; i++) {
        _lgeWhitelistRounds[index].addresses[addresses[i]] = enabled;
    }
}

/*
 * getLGEWhitelistRound
 *
 * returns:
 *
 * 1. whitelist round number ( 0 = no active round now )
 * 2. duration, in seconds, current whitelist round is active for
 * 3. timestamp current whitelist round closes at
 * 4. maximum amount a whitelister can purchase in this round
 * 5. is caller whitelisted

```

```

* 6. how much caller has purchased in current whitelist round
*
*/

function getLGEWhitelistRound()
    public
    view
    returns (
        uint256,
        uint256,
        uint256,
        uint256,
        bool,
        uint256
    )
{
    if (_lgeTimestamp > 0) {
        uint256 wlCloseTimestampLast = _lgeTimestamp;

        for (uint256 i = 0; i < _lgeWhitelistRounds.length; i++) {
            WhitelistRound storage wlRound = _lgeWhitelistRounds[i];

            wlCloseTimestampLast = wlCloseTimestampLast + wlRound.duration;
            if (block.timestamp <= wlCloseTimestampLast)
                return (
                    i + 1,
                    wlRound.duration,
                    wlCloseTimestampLast,
                    wlRound.amountMax,
                    wlRound.addresses[_msgSender()],
                    wlRound.purchased[_msgSender()]
                );
        }

        return (0, 0, 0, 0, false, 0);
    }
}

/*
* _applyLGEWhitelist - internal function to be called initially before any transfers
*
*/

function _applyLGEWhitelist(
    address sender,
    address recipient,
    uint256 amount
) internal {
    if (_lgePairAddress == address(0) || _lgeWhitelistRounds.length == 0) return;

    if (_lgeTimestamp == 0 && sender != _lgePairAddress && recipient == _lgePairAddress && amount > 0)
        _lgeTimestamp = block.timestamp;

    if (sender == _lgePairAddress && recipient != _lgePairAddress) {
        //buying

        (uint256 wlRoundNumber, , , , ) = getLGEWhitelistRound();

        if (wlRoundNumber > 0) {
            WhitelistRound storage wlRound = _lgeWhitelistRounds[wlRoundNumber - 1];

            require(wlRound.addresses[recipient], "LGE - Buyer is not whitelisted");

            uint256 amountRemaining = 0;

            if (wlRound.purchased[recipient] < wlRound.amountMax)

```

```

        amountRemaining = wlRound.amountMax - wlRound.purchased[recipient];

        require(amount <= amountRemaining, "LGE - Amount exceeds whitelist maximum");
        wlRound.purchased[recipient] = wlRound.purchased[recipient] + amount;
    }
}
}

contract Lovelace is Context, IBEP20, Ownable, LGEWhitelisted {

    using SafeMath for uint256;

    mapping (address => uint256) private _balances;

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

    uint256 private _totalSupply;
    uint8 private _decimals;
    string private _symbol;
    string private _name;

    constructor() {
        _name = "Lovelace";
        _symbol = "LACE";
        _decimals = 18;
        _totalSupply = 250_000_000 * 10 ** 18;
        _balances[_msgSender()] = _totalSupply;

        emit Transfer(address(0), _msgSender(), _totalSupply);
    }

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

    /**
     * @dev Returns the token decimals.
     */
    function decimals() external view override returns (uint8) {
        return _decimals;
    }

    /**
     * @dev Returns the token symbol.
     */
    function symbol() external view override returns (string memory) {
        return _symbol;
    }

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

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

```

```

/**
 * @dev See {BEP20-balanceOf}.
 */
function balanceOf(address account) external view override 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) external override returns (bool) {
    _transfer(_msgSender(), recipient, amount);
    return true;
}

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

/**
 * @dev See {BEP20-approve}.
 *
 * Requirements:
 *
 * - `spender` cannot be the zero address.
 */
function approve(address spender, uint256 amount) external 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) external override return
    _transfer(sender, recipient, amount);
    _approve(sender, _msgSender(), _allowances[sender][_msgSender()].sub(amount, "BEP20: transfer
    return true;
}

/**
 * @dev Destroys `amount` tokens from the caller.
 *
 * See {ERC20-burn}.
 */
function burn(uint256 amount) public virtual {
    _burn(_msgSender(), amount);
}

```

```

/**
 * @dev Destroys `amount` tokens from `account`, deducting from the caller's
 * allowance.
 *
 * See {ERC20-burn} and {ERC20-allowance}.
 *
 * Requirements:
 *
 * - the caller must have allowance for ``accounts``'s tokens of at least
 * `amount`.
 */
function burnFrom(address account, uint256 amount) public virtual {
    uint256 decreasedAllowance = allowance(account, _msgSender()).sub(amount, "ERC20: burn amount");

    _approve(account, _msgSender(), decreasedAllowance);
    _burn(account, amount);
}

/**
 * @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) external 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) external returns (bool) {
    _approve(_msgSender(), spender, _allowances[_msgSender()][spender].sub(subtractedValue, "BEP20: decrease allowance"));
    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");

    _applyLGEWhitelist(sender, recipient, amount);

    _balances[sender] = _balances[sender].sub(amount, "BEP20: transfer amount exceeds balance");
    _balances[recipient] = _balances[recipient].add(amount);
    emit Transfer(sender, recipient, 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`, 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 virtual {
    require(account != address(0), "ERC20: burn from the zero address");

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

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

```

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 whether 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 devastating vulnerabilities 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 initialising 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

