



Smart Contract Security Audit Report



The SlowMist Security Team received the Ludena Protocol team's application for smart contract security audit of the LDN on July 12, 2020. The following are the details and results of this smart contract security audit:

Token name :

LDN

The Contract address :

0xb29663Aa4E2e81e425294193616c1B102B70a158

Link address :

<https://etherscan.io/address/0xb29663Aa4E2e81e425294193616c1B102B70a158>

The audit items and results :

(Other unknown security vulnerabilities are not included in the audit responsibility scope)

No.	Audit Items	Audit Subclass	Audit Subclass Result
1	Overflow Audit	-	Passed
2	Race Conditions Audit	-	Passed
3	Authority Control Audit	Permission vulnerability audit	Passed
		Excessive auditing authority	Passed
4	Safety Design Audit	Zeppelin module safe use	Passed
		Compiler version security	Passed
		Hard-coded address security	Passed
		Fallback function safe use	Passed
		Show coding security	Passed
		Function return value security	Passed
		Call function security	Passed
5	Denial of Service Audit	-	Passed
6	Gas Optimization Audit	-	Passed
7	Design Logic Audit	-	Passed
8	"False Deposit" vulnerability Audit	-	Passed

9	Malicious Event Log Audit	-	Passed
10	Scoping and Declarations Audit	-	Passed
11	Replay Attack Audit	ECDSA's Signature Replay Audit	Passed
12	Uninitialized Storage Pointers Audit	-	Passed
13	Arithmetic Accuracy Deviation Audit	-	Passed

Audit Result : Passed

Audit Number : 0X002007170001

Audit Date : July 17, 2020

Audit Team : SlowMist Security Team

(Statement : SlowMist only issues this report based on the fact that has occurred or existed before the report is issued, and bears the corresponding responsibility in this regard. For the facts occur or exist later after the report, SlowMist cannot judge the security status of its smart contract. SlowMist is not responsible for it. The security audit analysis and other contents of this report are based on the documents and materials provided by the information provider to SlowMist as of the date of this report (referred to as "the provided information"). SlowMist assumes that: there has been no information missing, tampered, deleted, or concealed. If the information provided has been missed, modified, deleted, concealed or reflected and is inconsistent with the actual situation, SlowMist will not bear any responsibility for the resulting loss and adverse effects. SlowMist will not bear any responsibility for the background or other circumstances of the project.)

Summary: This is a token contract that contains the tokenVault section. The total amount of contract tokens remains unchanged. OpenZeppelin ' s SafeMath security module is used, which is a commendable approach. The contract does not have the Overflow and the Race Conditions issue. This is an upgradable contract. The owner can upgrade the contract at any time.

During the audit, we found the following issues:

- 1. In the transfer function, the frozen state of [to] was not checked.**
- 2. In the transferFrom function, the frozen state of [msg.sender] and [to] was not checked.**
- 3. The pauser role can freeze any user account through the freezeAccount function.**
- 4. The Pauser role can lock any user amount of balance through the lock function**

The source code:

```
/**
 *Submitted for verification at Etherscan.io on 2020-05-20
 */

//SlowMist// The contract does not have the Overflow and the Race Conditions issue

pragma solidity ^0.5.17;

//SlowMist// OpenZeppelin' s SafeMath security Module is used, which is a recommend approach

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 zero.
     */
    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 minuend).
     */
    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;
}
}

library Roles {
    struct Role {
        mapping (address => bool) bearer;
    }

    /**
     * @dev give an account access to this role
     */
    function add(Role storage role, address account) internal {
        require(account != address(0));
        require(!has(role, account));

        role.bearer[account] = true;
    }

    /**
```

```
* @dev remove an account's access to this role
*/

function remove(Role storage role, address account) internal {
    require(account != address(0));
    require(has(role, account));

    role.bearer[account] = false;
}

/**
* @dev check if an account has this role
* @return bool
*/

function has(Role storage role, address account) internal view returns (bool) {
    require(account != address(0));
    return role.bearer[account];
}
}

contract Ownable {
    address public owner;
    address public newOwner;

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

    constructor() public {
        owner = msg.sender;
        newOwner = address(0);
    }

    modifier onlyOwner() {
        require(msg.sender == owner);
        _;
    }

    modifier onlyNewOwner() {
        require(msg.sender != address(0));
        require(msg.sender == newOwner);
        _;
    }

    function isOwner(address account) public view returns (bool) {
        if( account == owner ){
```

```
        return true;
    }
    else {
        return false;
    }
}
```

```
function transferOwnership(address _newOwner) public onlyOwner {
```

```
    require(_newOwner != address(0)); //SlowMist// This check is quite good in avoiding losing control of
```

the contract caused by user mistakes

```
        newOwner = _newOwner;
    }
}
```

```
function acceptOwnership() public onlyNewOwner returns(bool) {
```

```
    emit OwnershipTransferred(owner, newOwner);
```

```
    owner = newOwner;
```

```
    newOwner = address(0);
```

```
}
```

```
}
```

```
contract PauserRole is Ownable{
```

```
    using Roles for Roles.Role;
```

```
    event PauserAdded(address indexed account);
```

```
    event PauserRemoved(address indexed account);
```

```
    Roles.Role private _pausers;
```

```
    constructor () internal {
```

```
        _addPauser(msg.sender);
```

```
}
```

```
    modifier onlyPauser() {
```

```
        require(isPauser(msg.sender) || isOwner(msg.sender));
```

```
        _;
```

```
}
```

```
function isPauser(address account) public view returns (bool) {
```

```
    return _pausers.has(account);
```

```
}
```

```
function addPauser(address account) public onlyPauser {
    _addPauser(account);
}

function removePauser(address account) public onlyOwner {
    _removePauser(account);
}

function renouncePauser() public {
    _removePauser(msg.sender);
}

function _addPauser(address account) internal {
    _pausers.add(account);
    emit PauserAdded(account);
}

function _removePauser(address account) internal {
    _pausers.remove(account);
    emit PauserRemoved(account);
}
}

contract Pausable is PauserRole {
    event Paused(address account);
    event Unpaused(address account);

    bool private _paused;

    constructor () internal {
        _paused = false;
    }

    /**
     * @return true if the contract is paused, false otherwise.
     */
    function paused() public view returns (bool) {
        return _paused;
    }

    /**
```



```
* @dev Modifier to make a function callable only when the contract is not paused.
*/
modifier whenNotPaused() {
    require(!_paused);
    _;
}

/**
* @dev Modifier to make a function callable only when the contract is paused.
*/
modifier whenPaused() {
    require(_paused);
    _;
}

/**
* @dev called by the owner to pause, triggers stoppea state
*/

//SlowMist// Suspending all transactions upon major abnormalities is a recommended approach

function pause() public onlyPauser whenNotPaused {
    _paused = true;
    emit Paused(msg.sender);
}

/**
* @dev called by the owner to unpause, returns to normal state
*/
function unpause() public onlyPauser whenPaused {
    _paused = false;
    emit Unpaused(msg.sender);
}
}

interface IERC20 {
    function transfer(address to, uint256 value) external returns (bool);

    function approve(address spender, uint256 value) external returns (bool);

    function transferFrom(address from, address to, uint256 value) external returns (bool);

    function totalSupply() external view returns (uint256);
}
```

```
function balanceOf(address who) external view returns (uint256);

function allowance(address owner, address spender) external view returns (uint256);

event Transfer(address indexed from, address indexed to, uint256 value);

event Approval(address indexed owner, address indexed spender, uint256 value);
}

contract ERC20 is IERC20 {
    using SafeMath for uint256;

    mapping (address => uint256) internal _balances;

    mapping (address => mapping (address => uint256)) internal _allowed;

    uint256 private _totalSupply;

    /**
     * @dev Total number of tokens in existence
     */
    function totalSupply() public view returns (uint256) {
        return _totalSupply;
    }

    /**
     * @dev Gets the balance of the specified address.
     * @param owner The address to query the balance of.
     * @return An uint256 representing the amount owned by the passed address.
     */
    function balanceOf(address owner) public view returns (uint256) {
        return _balances[owner];
    }

    /**
     * @dev Function to check the amount of tokens that an owner allowed to a spender.
     * @param owner address The address which owns the funds.
     * @param spender address The address which will spend the funds.
     * @return A uint256 specifying the amount of tokens still available for the spender.
     */
    function allowance(address owner, address spender) public view returns (uint256) {
```

```
    return _allowed[owner][spender];
}

/**
 * @dev Transfer token for a specified address
 * @param to The address to transfer to.
 * @param value The amount to be transferred.
 */
function transfer(address to, uint256 value) public returns (bool) {
    _transfer(msg.sender, to, value);

    return true; //SlowMist// The return value conforms to the EIP20 specification
}

/**
 * @dev Approve the passed address to spend the specified amount of tokens on behalf of msg.sender.
 * 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
 * @param spender The address which will spend the funds.
 * @param value The amount of tokens to be spent.
 */
function approve(address spender, uint256 value) public returns (bool) {
    require(spender != address(0)); //SlowMist// This kind of check is very good, avoiding user mistake
```

leading to approve errors

```
    _allowed[msg.sender][spender] = value;
    emit Approval(msg.sender, spender, value);

    return true; //SlowMist// The return value conforms to the EIP20 specification
}

/**
 * @dev Transfer tokens from one address to another.
 * Note that while this function emits an Approval event, this is not required as per the specification,
 * and other compliant implementations may not emit the event.
 * @param from address The address which you want to send tokens from
 * @param to address The address which you want to transfer to
 * @param value uint256 the amount of tokens to be transferred
 */
```

```
*/  
  
function transferFrom(address from, address to, uint256 value) public returns (bool) {  
    _allowed[from][msg.sender] = _allowed[from][msg.sender].sub(value);  
    _transfer(from, to, value);  
    emit Approval(from, msg.sender, _allowed[from][msg.sender]);  
  
    return true; //SlowMist// The return value conforms to the EIP20 specification  
}  
  
/**  
 * @dev Increase the amount of tokens that an owner allowed to a spender.  
 * approve should be called when allowed[_spender] == 0. To increment  
 * allowed value is better to use this function to avoid 2 calls (and wait until  
 * the first transaction is mined)  
 * From MonolithDAO Token.sol  
 * Emits an Approval event.  
 * @param spender The address which will spend the funds.  
 * @param addedValue The amount of tokens to increase the allowance by.  
 */  
  
function increaseAllowance(address spender, uint256 addedValue) public returns (bool) {  
    require(spender != address(0));  
  
    _allowed[msg.sender][spender] = _allowed[msg.sender][spender].add(addedValue);  
    emit Approval(msg.sender, spender, _allowed[msg.sender][spender]);  
    return true;  
}  
  
/**  
 * @dev Decrease the amount of tokens that an owner allowed to a spender.  
 * approve should be called when allowed[_spender] == 0. To decrement  
 * allowed value is better to use this function to avoid 2 calls (and wait until  
 * the first transaction is mined)  
 * From MonolithDAO Token.sol  
 * Emits an Approval event.  
 * @param spender The address which will spend the funds.  
 * @param subtractedValue The amount of tokens to decrease the allowance by.  
 */  
  
function decreaseAllowance(address spender, uint256 subtractedValue) public returns (bool) {  
    require(spender != address(0));  
  
    _allowed[msg.sender][spender] = _allowed[msg.sender][spender].sub(subtractedValue);  
    emit Approval(msg.sender, spender, _allowed[msg.sender][spender]);  
}
```

```
    return true;
}

/**
 * @dev Transfer token for a specified addresses
 * @param from The address to transfer from.
 * @param to The address to transfer to.
 * @param value The amount to be transferred.
 */
function _transfer(address from, address to, uint256 value) internal {
    require(to != address(0)); //SlowMist// This kind of check is very good, avoiding user mistake
```

leading to the loss of token during transfer

```
    _balances[from] = _balances[from].sub(value);
    _balances[to] = _balances[to].add(value);
    emit Transfer(from, to, value);
}

/**
 * @dev Internal function that mints an amount of the token and assigns it to
 * an account. This encapsulates the modification of balances such that the
 * proper events are emitted.
 * @param account The account that will receive the created tokens.
 * @param value The amount that will be created.
 */
function _mint(address account, uint256 value) internal {
    require(account != address(0));

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

/**
 * @dev Internal function that burns an amount of the token of a given
 * account.
 * @param account The account whose tokens will be burnt.
 * @param value The amount that will be burnt.
 */
function _burn(address account, uint256 value) internal {
```

```
require(account != address(0));

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

/**
 * @dev Internal function that burns an amount of the token of a given
 * account, deducting from the sender's allowance for said account. Uses the
 * internal burn function.
 * Emits an Approval event (reflecting the reduced allowance).
 * @param account The account whose tokens will be burnt.
 * @param value The amount that will be burnt.
 */
```

//SlowMist// It's redundant code

```
function _burnFrom(address account, uint256 value) internal {
    _allowed[account][msg.sender] = _allowed[account][msg.sender].sub(value);
    _burn(account, value);
    emit Approval(account, msg.sender, _allowed[account][msg.sender]);
}

contract ERC20Pausable is ERC20, Pausable {
    function transfer(address to, uint256 value) public whenNotPaused returns (bool) {
        return super.transfer(to, value);
    }

    function transferFrom(address from, address to, uint256 value) public whenNotPaused returns (bool) {
        return super.transferFrom(from, to, value);
    }

    /**
     * approve/increaseApprove/decreaseApprove can be set when Pause state
     */

    /**
     * function approve(address spender, uint256 value) public whenNotPaused returns (bool) {
     *     return super.approve(spender, value);
     * }
     */
}
```

```
    * function increaseAllowance(address spender, uint addedValue) public whenNotPaused returns (bool success) {
    *     return super.increaseAllowance(spender, addedValue);
    * }
    *
    * function decreaseAllowance(address spender, uint subtractedValue) public whenNotPaused returns (bool success) {
    *     return super.decreaseAllowance(spender, subtractedValue);
    * }
    */
}

contract ERC20Detailed is IERC20 {
    string private _name;
    string private _symbol;
    uint8 private _decimals;

    constructor (string memory name, string memory symbol, uint8 decimals) public {
        _name = name;
        _symbol = symbol;
        _decimals = decimals;
    }

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

    /**
     * @return the symbol of the token.
     */
    function symbol() public view returns (string memory) {
        return _symbol;
    }

    /**
     * @return the number of decimals of the token.
     */
    function decimals() public view returns (uint8) {
        return _decimals;
    }
}
```

```
contract LDNToken is ERC20Detailed, ERC20Pausable {

    struct LockInfo {
        uint256 _releaseTime;
        uint256 _amount;
        uint256 _remainingAmount;
        uint256 _termOfRound;
        uint256 _unlockAmountPerRound;
    }

    address public implementation;

    mapping (address => LockInfo[]) public timelockList;
    mapping (address => bool) public frozenAccount;

    event Freeze(address indexed holder);
    event Unfreeze(address indexed holder);
    event Lock(address indexed holder, uint256 value, uint256 releaseTime);
    event Unlock(address indexed holder, uint256 value);

    modifier notFrozen(address _holder) {
        require(!frozenAccount[_holder]);
        _;
    }

    constructor() ERC20Detailed("Ludena Protocol", "LDN", 18) public {

        _mint(msg.sender, 1200000000 * (10 ** 18));
    }

    function balanceOf(address owner) public view returns (uint256) {

        uint256 totalBalance = super.balanceOf(owner);
        if( timelockList[owner].length > 0 ){
            for(uint i=0; i<timelockList[owner].length;i++){
                totalBalance = totalBalance.add( timelockList[owner][i]._remainingAmount );
            }
        }
    }
}
```



```
    return totalBalance;
}
```

//SlowMist// The frozen state of [to] was not checked

```
function transfer(address to, uint256 value) public notFrozen(msg.sender) returns (bool) {
    if (timelockList[msg.sender].length > 0) {
        _autoUnlock(msg.sender);
    }
    return super.transfer(to, value);
}
```

//SlowMist// The frozen state of [msg.sender] and [to] was not checked

```
function transferFrom(address from, address to, uint256 value) public notFrozen(from) returns (bool) {
    if (timelockList[from].length > 0) {
        _autoUnlock(from);
    }
    return super.transferFrom(from, to, value);
}
```

//SlowMist// The pauser role can freeze any user account through the freezeAccount function

```
function freezeAccount(address holder) public onlyPauser returns (bool) {
    require(!frozenAccount[holder]);
    frozenAccount[holder] = true;
    emit Freeze(holder);
    return true;
}
```

```
function unfreezeAccount(address holder) public onlyPauser returns (bool) {
    require(frozenAccount[holder]);
    frozenAccount[holder] = false;
    emit Unfreeze(holder);
    return true;
}
```

//SlowMist// The Pauser role can lock any user amount of balance through the lock function

```
function lock(address holder, uint256 value, uint256 releaseStart, uint256 termOfRound, uint256 unlockAmountPerRound)
public onlyPauser returns (bool) {

    _lock(holder, value, releaseStart, termOfRound, unlockAmountPerRound);

    return true;
}
```

```
function transferWithLock(address holder, uint256 value, uint256 releaseTime) public onlyPauser returns (bool) {
    _transfer(msg.sender, holder, value);
    _lock(holder, value, releaseTime, 1, value);
    return true;
}

function transferWithSliceLock(address holder, uint256 value, uint256 releaseStart, uint256 termOfRound, uint256
unlockAmountPerRound) public onlyPauser returns (bool) {
    _transfer(msg.sender, holder, value);
    _lock(holder, value, releaseStart, termOfRound, unlockAmountPerRound);
    return true;
}

/**
 * @dev Upgrades the implementation address
 * @param _newImplementation address of the new implementation
 */

//SlowMist// This is an upgradeable contract, and a security audit of the upgradedAddress
contract is required before the contract is upgraded

function upgradeTo(address _newImplementation) public onlyOwner {
    require(implementation != _newImplementation);
    _setImplementation(_newImplementation);
}

function _lock(address holder, uint256 value, uint256 releaseTime, uint256 termOfRound, uint256
unlockAmountPerRound) internal returns(bool) {
    require(_balances[holder] >= value, "There is not enough balances of holder.");
    _balances[holder] = _balances[holder].sub(value);
    timelockList[holder].push( LockInfo(releaseTime, value, value, termOfRound, unlockAmountPerRound) );

    emit Lock(holder, value, releaseTime);
    return true;
}

function _releaseTimeLock(address _holder, uint256 _idx) internal returns(bool) {
    require(_idx < timelockList[_holder].length);

    // If lock status of holder is finished, delete lockup info.
    LockInfo storage info = timelockList[_holder][_idx];
```

```
uint256 releaseAmount = info._unlockAmountPerRound;
uint256 sinceFrom = now.sub(info._releaseTime);
uint256 sinceRound = sinceFrom.div(info._termOfRound);
releaseAmount = releaseAmount.add( sinceRound.mul(info._unlockAmountPerRound) );

if(releaseAmount >= info._remainingAmount) {
    releaseAmount = info._remainingAmount;

    delete timelockList[_holder][_idx];
    timelockList[_holder][_idx] = timelockList[_holder][timelockList[_holder].length.sub(1)];
    timelockList[_holder].length -=1;

    emit Unlock(_holder, releaseAmount);
    _balances[_holder] = _balances[_holder].add(releaseAmount);

    return true;
} else {
    timelockList[_holder][_idx]._releaseTime =
timelockList[_holder][_idx]._releaseTime.add( sinceRound.add(1).mul(info._termOfRound) );
    timelockList[_holder][_idx]._remainingAmount =
timelockList[_holder][_idx]._remainingAmount.sub(releaseAmount);

    emit Unlock(_holder, releaseAmount);
    _balances[_holder] = _balances[_holder].add(releaseAmount);

    return false;
}
}

function _autoUnlock(address holder) internal returns(bool) {
    for(uint256 idx =0; idx < timelockList[holder].length ; idx++ ) {
        if (timelockList[holder][idx]._releaseTime <= now) {
            // If lockupinfo was deleted, loop restart at same position.
            if( _releaseTimeLock(holder, idx) ) {
                idx -=1;
            }
        }
    }
    return true;
}

/**
```

```
* @dev Sets the address of the current implementation  
* @param _newImp address of the new implementation  
*/
```

```
function _setImplementation(address _newImp) internal {  
    implementation = _newImp;  
}
```

```
/**  
* @dev Fallback function allowing to perform a delegatecall  
* to the given implementation. This function will return  
* whatever the implementation call returns  
*/
```

```
function () payable external {  
    address impl = implementation;  
    require(impl != address(0));  
    assembly {  
        let ptr := mload(0x40)  
        calldatacopy(ptr, 0, calldatasize)  
  
        let result := delegatecall(gas, impl, ptr, calldatasize, 0, 0) //SlowMist// The specific details of the
```

external contract invoked by Delegatecall are not reflected in this contract

```
        let size := returndatasize  
        returndatacopy(ptr, 0, size)
```

```
        switch result  
        case 0 { revert(ptr, size) }  
        default { return(ptr, size) }
```

```
    }
```

```
}
```

```
}
```



Official Website

www.slowmist.com

E-mail

team@slowmist.com

Twitter

[@SlowMist_Team](https://twitter.com/SlowMist_Team)

WeChat Official Account

