Smart Contract Security Audit Report

The SlowMist Security Team received the CAMP team's application for smart contract security audit of the CAMP Token on Sep 17, 2020. The following are the details and results of this smart contract security audit:

Token name :

CAMP

The Contract file and file Hash (SHA256):

Camp2ndDelivery.sol

bb262ff58d04deae5453e31ee802ab471521f0ae076762a8076e4df5f849f934

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 : 0X002009280002

Audit Date : Sep 28, 2020

Audit Team : SlowMist Security Team

Summary: This is a token contract that contains the tokenVault section. Owner can add any account to _pausers, _pausers can freeze any account or lock any account token. SafeMath security module is used, which is a commendable approach. The contract does not have the Overflow and the Race Conditions issue. The comprehensive evaluation contract is no risk.

The source code:

```
/**
 *Submitted for verification at Etherscan.io on 2019-12-12
*/

pragma solidity ^0.5.0;

/**
 * @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) {
        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;
    }
}

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);
    }
}
```

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

```
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 stopped state
     */

    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 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));

    _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 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 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
        require(from != address(0));

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

    /**
     * @dev Destroys `amount` tokens from the caller.
     *
     * See {ERC20-_burn}.
     */
    function burn(uint256 value) public returns (bool) {
        _burn(msg.sender, value);
        return true;
    }

    /**
```

```
     * @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// this code redundancy

```
    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]);
    }

    /**
     * @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);
    }
}

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 Paused 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 ERC20Flag is ERC20Detailed, ERC20Pausable {

    struct LockInfo {
        uint256 _releaseTime;
        uint256 _amount;
    }

    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("CAMP Coin", "CAMP", 18) public  {

    _mint(msg.sender, 10000000000 * (10 ** 18));
}

function timelockListLength(address owner) public view returns (uint256) {
    return timelockList[owner].length;
}

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]._amount);
        }
    }

    return totalBalance;
}

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

    uint256 totalLocked = 0;
    if( timelockList[owner].length >0 ){
        for(uint i=0; i<timelockList[owner].length;i++){
            totalLocked = totalLocked.add(timelockList[owner][i]._amount);
        }
    }

    return totalLocked;
}

function balanceOfAvailable(address owner) public view returns (uint256) {
```

```
    uint256 totalBalance = super.balanceOf(owner);
    return totalBalance;
}

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);
}

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 owner can freeze any account

```
function freezeAccount(address holder) public onlyPauser returns (bool) {
    require(!frozenAccount[holder]);
    require(timelockList[holder].length == 0);
    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 owner can freeze any account

```
function lockByQuantity(address holder, uint256 value, uint256 releaseTime) public onlyPauser returns (bool) {
    require(!frozenAccount[holder]);
    _lock(holder,value,releaseTime);
    return true;
}
```

//SlowMist// The owner can lock any account token

```
function unlockByQuantity(address holder, uint256 value, uint256 releaseTime) public onlyPauser returns (bool) {
    //1
    require(!frozenAccount[holder]);
    //2
    require(timelockList[holder].length >0);

    //3
    uint256 totalLocked;
    for(uint idx = 0; idx < timelockList[holder].length ; idx++ ){
        totalLocked = totalLocked.add(timelockList[holder][idx]._amount);
    }
    require(totalLocked >value);

    //4
    for(uint idx = 0; idx < timelockList[holder].length ; idx++ ) {
        if( _unlock(holder, idx) ) {
            idx -=1;
        }
    }

    //5
    _lock(holder,totalLocked.sub(value),releaseTime);

    return true;
}

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

function unlock(address holder, uint256 idx) public onlyPauser returns (bool) {
    require( timelockList[holder].length > idx, "AhnLog_There is not lock info.");
    _unlock(holder,idx);
    return true;
}
```

```
    function _lock(address holder, uint256 value, uint256 releaseTime) internal returns (bool) {
        _balances[holder] = _balances[holder].sub(value);
        timelockList[holder].push( LockInfo(releaseTime, value) );

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

    function _unlock(address holder, uint256 idx) internal returns(bool) {
        LockInfo storage lockinfo = timelockList[holder][idx];
        uint256 releaseAmount = lockinfo._amount;

        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;
    }

    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( _unlock(holder, idx) ) {
                    idx -=1;
                }
            }
        }
        return true;
    }

}
```

Official Website
www.slowmist.com

E-mail
team@slowmist.com

Twitter
@SlowMist_Team

Github
https://github.com/slowmist