Smart Contract Security Audit Report

The SlowMist Security Team received the Hot Cross team's application for smart contract security audit of the Cross Pool on Mar 31, 2021. The following are the details and results of this smart contract security audit:

Token name :

CROSS

File name and HASH(SHA256) :

proj-cross-pool-solidity-main.zip:
6b1a031af42416d3a1f9f1887c2c585c0a0de18f14030e8449b40d245f5af6a9

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 : 0X002104020001

Audit Date : April 02, 2021

Audit Team : SlowMist Security Team

Summary: This is a staking + token contract. The total amount of contract tokens is variable,The owner can mint token via the mint method, but the method has a limit on the number of coins that can be minted. OpenZeppelin's SafeMath security module is used, which is a commendable approach. The contract does not have the Overflow and the Race Conditions issue.

During the audit, we found the following issues:

1. The owner can modify the block reward via the setRewardPerBlock method

The source code:

Libs/Misc.sol

```
// SPDX-License-Identifier: MIT
pragma solidity 0.7.6;

library Misc {
  /**
    * @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) public 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;
    }

    function today() external view returns (uint256) {
        return block.timestamp / 1 days;
    }

    function zeroOrContract(address account, string memory errorMsg) external view {
        require(
            account != address(0) && isContract(account),
            errorMsg
        );
    }
}
```

Mocks/BEP20Mock.sol

```
// SPDX-License-Identifier: MIT
pragma solidity 0.7.6;

import "@openzeppelin/contracts/access/Ownable.sol";
import "@openzeppelin/contracts/token/ERC20/ERC20.sol";
```

```
import "@openzeppelin/contracts/token/ERC20/ERC20Capped.sol";
import "@openzeppelin/contracts/math/SafeMath.sol";
import "../token/IBEP20.sol";

contract BEP20Mock is Ownable, IBEP20, ERC20Capped {
    uint256 constant CAP = 100e24; // 100 million

    constructor() ERC20("TOKEN", "TOK") ERC20Capped(CAP) {}

    /**
     * @notice Allow the owner to mint tokens
     * @param to Address that will receive the minted tokens
     * @param amount Amount of tokens that will be minted
     */
    function mint(address to, uint256 amount) public onlyOwner {
        _mint(to, amount);
    }
}
```

Token/Cross.sol

```
// SPDX-License-Identifier: MIT
pragma solidity 0.7.6;

import "@openzeppelin/contracts/access/Ownable.sol";
import "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import "@openzeppelin/contracts/token/ERC20/ERC20Capped.sol";
import "@openzeppelin/contracts/math/SafeMath.sol";
import "./IBEP20.sol";

contract Cross is Ownable, IBEP20, ERC20Capped {
  uint256 constant CAP = 100e24; // 100 million

  constructor()
    ERC20('Hot Cross', 'CROSS')
    ERC20Capped(CAP){}

  /**
   * @notice Allow the owner to mint tokens
   * @param to Address that will receive the minted tokens
   * @param amount Amount of tokens that will be minted
   */
```

```
  //SlowMist// The owner can mint coins to any user via the mint method, but the method has a limit on the number of coins that can be minted

  function mint(address to, uint256 amount) public onlyOwner {
    _mint(to, amount);
  }
}
```

Token/IBEP20.sol

```
// SPDX-License-Identifier: MIT
pragma solidity 0.7.6;

import "@openzeppelin/contracts/token/ERC20/IERC20.sol";

/**
 * @dev Interface of the BEP20 standard as defined in the EIP.
 */
interface IBEP20 is IERC20 {}
```

RewardVault.sol

```
// SPDX-License-Identifier: MIT
pragma solidity 0.7.6;

import "@openzeppelin/contracts-upgradeable/proxy/Initializable.sol";
import "@openzeppelin/contracts-upgradeable/access/OwnableUpgradeable.sol";
import "@openzeppelin/contracts/math/SafeMath.sol";
import "./token/IBEP20.sol";

/**
 * @title A contract that holds the Cake rewards minted from the CrossPool
 */
contract RewardVault is Initializable, OwnableUpgradeable {
  IBEP20 public rewardToken;

  /**
   * Initializes the contract
   */
  function initialize(IBEP20 _rewardToken) public initializer {
    __Ownable_init();
    rewardToken = _rewardToken;
```

```
  }

  /**
   * @notice Transfers the given amount of reward tokens to the address provided
   * @param to The address that will receive the reward tokens
   * @param amount The amount of reward tokens that will be transfered
   */
  function safeRewardTransfer(
    address to,
    uint256 amount
  ) public onlyOwner {
    uint256 rewardTokenBalance = rewardToken.balanceOf(address(this));
    uint256 transferableAmount = amount > rewardTokenBalance
      ? rewardTokenBalance
      : amount;

    rewardToken.transfer(to, transferableAmount);
  }
}
```

CrossPool.sol

```
// SPDX-License-Identifier: MIT
pragma solidity 0.7.6;

import "@openzeppelin/contracts-upgradeable/proxy/Initializable.sol";
import "@openzeppelin/contracts-upgradeable/access/OwnableUpgradeable.sol";
import "@openzeppelin/contracts/math/SafeMath.sol";
import "@openzeppelin/contracts/math/Math.sol";
import "./token/IBEP20.sol";
import "./RewardVault.sol";
import "./libs/Misc.sol";

contract CrossPool is Initializable, OwnableUpgradeable {
  using SafeMath for uint256;
  using Math for uint256;

  struct UserInfo {
    uint256 amount;
    uint256 rewardDebt;
    uint256 accClaim;
  }
```

```
struct PoolInfo {
  // Address of LP token contract.
  IBEP20 stakingToken;
  // How many allocation points assigned to this pool. rewards to distribute per block.
  uint256 allocPoint;
  // Last block number that rewards distribution occurs.
  uint256 lastRewardBlock;
  // Accumulated rewards per share, times 1e12.
  uint256 accRewardPerShare;
}

uint256 public rewardPerBlock;
uint256 public startBlock;
uint256 public endBlock;
uint256 public timeLock;

// Total allocation points. Must be the sum of all allocation points in all pools.
uint256 public totalAllocPoint;

RewardVault public rewardVault;
IBEP20 public rewardToken;

PoolInfo[] public poolInfo;
mapping (uint256 => mapping (address => UserInfo)) public userInfo;

event PoolAdded(uint256 indexed pid);
event AllocPointsChanged(uint256 indexed pid, uint256 poolAllocPoint, uint256 totalAllocPoint);
event Deposit(address indexed user, uint256 indexed pid, uint256 amount);
event Withdraw(address indexed user, uint256 indexed pid, uint256 amount);
event EmergencyWithdraw(address indexed user, uint256 amount);
event RewardPerBlock(uint256 rewardPerBlock);

/**
 * Initializes the contract
 * @param rewardToken_ The reward token address
 * @param rewardVault_ The contract that will hold the unclaimed rewards
 * @param rewardPerBlock_ The reward to be rewarded on each block
 * @param startBlock_ The block number when reward mining starts.
 */
```

```
function initialize(
    IBEP20 rewardToken_,
    RewardVault rewardVault_,
    uint256 rewardPerBlock_,
    uint256 startBlock_,
    uint256 endBlock_,
    uint256 timeLock_
) public initializer {
    __Ownable_init();
    Misc.zeroOrContract(address(rewardToken_), "Invalid rewardToken address");
    Misc.zeroOrContract(address(rewardVault_), "Invalid reward vault address");

    __Ownable_init();

    rewardToken = rewardToken_;
    rewardVault = rewardVault_;
    rewardPerBlock = rewardPerBlock_;
    startBlock = startBlock_;
    endBlock = endBlock_;
    timeLock = timeLock_;
    totalAllocPoint = 0;
}

/**
 * @notice Returns the total amount of pool, active and inactive
 */
function getPoolCount() public view returns(uint256){
    return poolInfo.length;
}

/**
 * @notice Add a new pool for the given token
 * @param allocPoint The alllocation point that will determine the portion of the block reward this pool
 * is entitled to
 * @param stakingToken The staking token for this pool
 * @param withUpdate If true will update all the pools
 */
function add(
    uint256 allocPoint,
    IBEP20 stakingToken,
```

```
    bool withUpdate
) external onlyOwner {
    Misc.zeroOrContract(address(stakingToken), "Invalid staking token address");

    if (withUpdate) {
        massUpdatePools();
    }

    uint256 lastRewardBlock = block.number > startBlock ? block.number : startBlock;
    totalAllocPoint = totalAllocPoint.add(allocPoint);

    poolInfo.push(PoolInfo({
        stakingToken: stakingToken,
        allocPoint: allocPoint,
        lastRewardBlock: lastRewardBlock,
        accRewardPerShare: 0
    }));

    emit PoolAdded(poolInfo.length - 1);
}

/**
 * @notice Updates every pool in the contract
 */
function massUpdatePools() public {
    uint256 length = poolInfo.length;
    for (uint256 pid = 0; pid < length; ++pid) {
        updatePool(pid);
    }
}

/**
 * @notice Update the allocation points; that is the portion of the reward per block that
 * the given pool is entitled to receive
 * @param pid The pool id
 * @param allocPoint The new allocation points
 * @param withUpdate If true will update all the pools
 */
function setAllocationPoints(
    uint256 pid,
```

```
    uint256 allocPoint,
    bool withUpdate
  ) external onlyOwner {
    if (withUpdate) {
      massUpdatePools();
    }

    uint256 prevAllocPoint = poolInfo[pid].allocPoint;
    poolInfo[pid].allocPoint = allocPoint;

    if (prevAllocPoint != allocPoint) {
      totalAllocPoint = totalAllocPoint
        .sub(prevAllocPoint)
        .add(allocPoint);
    }

    emit AllocPointsChanged(pid, allocPoint, totalAllocPoint);
  }

  /**
   * @notice Sets the new rewardToken reward per block
   * @param rewardPerBlock_ The new rewardToken reward per block
   */
  //SlowMist// The owner can modify the block reward via the setRewardPerBlock method
  function setRewardPerBlock(uint256 rewardPerBlock_) external onlyOwner {
    rewardPerBlock = rewardPerBlock_;

    emit RewardPerBlock(rewardPerBlock);
  }

  /**
   * @notice Useful function to withdraw the remaining rewards which have not been rewarded to the end users
   * This is an emergency hook which will not be needed in normal conditions.
   * @param recipient The address that will receive the reward tokens
   */
  function withdrawRemainingRewards(address recipient) external onlyOwner {
    require(block.number < startBlock, "cannot withdraw remaining rewards after the start block");
    rewardVault.safeRewardTransfer(recipient, rewardToken.balanceOf(address(rewardVault)));
  }
```

```
/**
 * @dev Returns the blocks that rewards have not been calculated for yer
 * @param lastRewardBlock The last block we calculated reward for
 */
function getPendingBlocks(uint256 lastRewardBlock) private view returns (uint256) {
  return block.number
    .min(endBlock)
    .sub(lastRewardBlock);
}

/**
 * @notice Returns the current rewardToken rewards for the given pool at the current block height
 * @param pool The information about the pool
 */
function getReward(PoolInfo memory pool) private view returns (uint256) {
  uint256 pendingBlocks = getPendingBlocks(pool.lastRewardBlock);

  return pendingBlocks
    .mul(rewardPerBlock)
      .mul(pool.allocPoint)
      .div(totalAllocPoint);
}

/**
 * @notice Returns the new accumulated rewardToken per share
 * @param currentAccRewardPerShare The current accumulated rewardToken per share
 * @param tokenReward The rewardToken reward an the current block height
 * @param stakingTokenSupply The current staking token supply
 */
function getAccRewardPerShare(
  uint256 currentAccRewardPerShare,
  uint256 tokenReward,
  uint256 stakingTokenSupply
) private pure returns (uint256) {
  return currentAccRewardPerShare.add(
    tokenReward
      .mul(1e12)
      .div(stakingTokenSupply)
  );
}
```

```
/**
 * @notice Allows external services to understand the amount of rewards that the
 * given user can claim
 * @param pid The pool id
 * @param account The account that we query the data for
 * @return Documents the return variables of a contract's function state variable
 */
function pendingRewards(uint256 pid, address account) external view returns (uint256) {
  PoolInfo memory pool = poolInfo[pid];
  UserInfo memory user = userInfo[pid][account];

  uint256 stakingTokenSupply = pool.stakingToken.balanceOf(address(this));

  if (block.number > pool.lastRewardBlock && stakingTokenSupply > 0) {
    uint256 tokenReward = getReward(pool);
    pool.accRewardPerShare = getAccRewardPerShare(
      pool.accRewardPerShare,
      tokenReward,
      stakingTokenSupply
    );
  }

  uint256 pending = user.amount
    .mul(pool.accRewardPerShare)
    .div(1e12)
    .sub(user.rewardDebt);

  return pending.add(user.accClaim);
}

/**
 * @dev Performs all the necessary tasks when core user transactions happen i.e. deposit, withdraw
 * @param pid The pool id
 */
function updatePool(uint256 pid) private {
  PoolInfo storage pool = poolInfo[pid];

  if (block.number <= pool.lastRewardBlock) {
```

```
        return;
    }

    uint256 stakingTokenSupply = pool.stakingToken.balanceOf(address(this));

    if (stakingTokenSupply == 0) {
        pool.lastRewardBlock = block.number;
        return;
    }

    uint256 tokenReward = getReward(pool);
    pool.accRewardPerShare = getAccRewardPerShare(
      pool.accRewardPerShare,
      tokenReward,
      stakingTokenSupply
    );

    pool.lastRewardBlock = block.number.min(endBlock);
}

/**
 * @notice Releases any pending rewards for the given user
 * @param accRewardPerShare The acc rewardToken per share
 */
function releasePending(uint256 accRewardPerShare, uint256 pid) private {
    UserInfo storage user = userInfo[pid][msg.sender];

    if(user.amount > 0) {
      uint256 pending = user.amount
        .mul(accRewardPerShare)
        .div(1e12)
        .sub(user.rewardDebt);
      uint256 totalPending = pending.add(user.accClaim);

      if(block.number >= timeLock && totalPending > 0) {
        user.accClaim = 0;
        rewardVault.safeRewardTransfer(msg.sender, totalPending);
      }
      else if(pending > 0) {
        user.accClaim = totalPending;
```

```
    }
  }
  else if(block.number >= timeLock && user.accClaim > 0) {
    rewardVault.safeRewardTransfer(msg.sender, user.accClaim);
    user.accClaim = 0;
  }
}

/**
 * @notice Accepts contribution of staking tokens to the given pool id
 * @param pid The pool id
 * @param amount The amount of the staking tokens to be deposited
 */
function deposit(uint256 pid, uint256 amount) external {
  UserInfo storage user = userInfo[pid][msg.sender];
  PoolInfo storage pool = poolInfo[pid];

  updatePool(pid);
  releasePending(pool.accRewardPerShare, pid);

  if (amount > 0) {
    pool.stakingToken.transferFrom(
      address(msg.sender),
      address(this),
      amount
    );
    user.amount = user.amount.add(amount);
  }

  user.rewardDebt = user.amount
    .mul(pool.accRewardPerShare)
    .div(1e12);

  emit Deposit(msg.sender, pid, amount);
}

/**
 * @notice Allows user to withdraw the staking token locked in the given pool
 * @param pid The pool id
 * @param amount The amount of the staking tokens to be withdrawn
```

```
    */
    function withdraw(uint256 pid, uint256 amount) external {
        PoolInfo storage pool = poolInfo[pid];
        UserInfo storage user = userInfo[pid][msg.sender];
        require(user.amount >= amount, "wrong withdraw amount");

        updatePool(pid);
        releasePending(pool.accRewardPerShare, pid);

        if(amount > 0) {
            user.amount = user.amount.sub(amount);
            pool.stakingToken.transfer(address(msg.sender), amount);
        }
        user.rewardDebt = user.amount
            .mul(pool.accRewardPerShare)
            .div(1e12);

        emit Withdraw(msg.sender, pid, amount);
    }
    /**
     * @notice Allows the user to withdraw the staking token without caring about the rewards
     * @param pid The pool id
     */
    function emergencyWithdraw(uint256 pid) public {
        PoolInfo storage pool = poolInfo[pid];
        UserInfo storage user = userInfo[pid][msg.sender];
        pool.stakingToken.transfer(address(msg.sender), user.amount);

        emit EmergencyWithdraw(msg.sender, user.amount);
        //SlowMist// It is recommended to set user.amount and user.rewardDebt to 0 first, and then
perform the transfer operation
        user.amount = 0;
        user.rewardDebt = 0;
        user.accClaim = 0;
    }
}
```

Official Website
www.slowmist.com

E-mail
team@slowmist.com

Twitter
@SlowMist_Team

Github
https://github.com/slowmist