Smart Contract Security Audit Report

[2021]

The SlowMist Security Team received the team's application for smart contract security audit of the Coin98VaultV2 on 2021.12.22. The following are the details and results of this smart contract security audit:

Project Name :

Coin98VaultV2

File name and hash (SHA256) :

Coin98VaultV2.sol: 989529f5a14d025ee232c33dda932705ef52081f1adb85850ceccaaf8fb77978

The audit items and results :

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

NO.	Audit Items	Result
1	Replay Vulnerability	Passed
2	Denial of Service Vulnerability	Passed
3	Race Conditions Vulnerability	Passed
4	Authority Control Vulnerability	Passed
5	Integer Overflow and Underflow Vulnerability	Passed
6	Gas Optimization Audit	Passed
7	Design Logic Audit	Passed
8	Uninitialized Storage Pointers Vulnerability	Passed
9	Arithmetic Accuracy Deviation Vulnerability	Passed
10	"False top-up" Vulnerability	Passed
11	Malicious Event Log Audit	Passed
12	Scoping and Declarations Audit	Passed

NO.	Audit Items	Result
13	Safety Design Audit	Passed

Audit Result : Passed

Audit Number : 0X002112240002

Audit Date : 2021.12.22 - 2021.12.24

Audit Team : SlowMist Security Team

Summary conclusion : This is a tokenVault contract. The vault contract limits the consumption of gas when transferring tokens to target users through the call function. The owner role can modify the status of the admin role through the setAdmins function. The admin role can transfer any Nfts and tokens of the vault to the specified user. The contract does not have the Overflow and the Race Conditions issue.

The source code:

Coin98VaultV2.sol

```
// SPDX-License-Identifier: Apache-2.0
//SlowMist// The contract does not have the Overflow and the Race Conditions issue
pragma solidity ^0.8.0;

/**
 * @dev Interface of the ERC20 standard as defined in the EIP.
 */
interface IERC20 {

  /**
    * @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 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 Required interface of an ERC721 compliant contract.
 */
interface IERC721 {

  /**
    * @dev Transfers `tokenId` token from `from` to `to`.
    *
    * WARNING: Usage of this method is discouraged, use {safeTransferFrom} whenever
possible.
    *
    * Requirements:
    *
    * - `from` cannot be the zero address.
    * - `to` cannot be the zero address.
    * - `tokenId` token must be owned by `from`.
    * - If the caller is not `from`, it must be approved to move this token by either
{approve} or {setApprovalForAll}.
    *
    * Emits a {Transfer} event.
    */
  function transferFrom(
    address from,
    address to,
    uint256 tokenId
  ) external;
}
```

```
interface IVaultConfig {

  function fee() external view returns (uint256);
  function gasLimit() external view returns (uint256);
  function ownerReward() external view returns (uint256);
}

/*
 * @dev Provides information about the current execution context, including the
 * sender of the transaction and its data. While these are generally available
 * via msg.sender and msg.data, they should not be accessed in such a direct
 * manner, since when dealing with GSN meta-transactions the account sending and
 * paying for execution may not be the actual sender (as far as an application
 * is concerned).
 *
 * This contract is only required for intermediate, library-like contracts.
 */
abstract contract Context {
  function _msgSender() internal view returns (address) {
    return msg.sender;
  }

  function _msgData() internal view returns (bytes memory) {
    this; // silence state mutability warning without generating bytecode - see
https://github.com/ethereum/solidity/issues/2691
    return msg.data;
  }

  function _msgValue() internal view returns (uint256) {
    return msg.value;
  }
}

/**
 * @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;
  address private _newOwner;

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

  /**
   * @dev Initializes the contract setting the deployer as the initial owner.
   */
  constructor (address owner_) {
    _owner = owner_;
    emit OwnershipTransferred(address(0), owner_);
  }

  /**
   * @dev Returns the address of the current owner.
   */
  function owner() public view returns (address) {
    return _owner;
  }

  /**
   * @dev Throws if called by any account other than the owner.
   */
  modifier onlyOwner() {
    require(owner() == _msgSender(), "Ownable: caller is not the owner");
    _;
  }

  /**
   * @dev Accept the ownership transfer. This is to make sure that the contract is
   * transferred to a working address
   *
   * Can only be called by the newly transfered owner.
   */
  function acceptOwnership() public {
    require(_msgSender() == _newOwner, "Ownable: only new owner can accept
ownership");
    address oldOwner = _owner;
    _owner = _newOwner;
    _newOwner = address(0);
```

```
    emit OwnershipTransferred(oldOwner, _owner);
  }

  /**
   * @dev Transfers ownership of the contract to a new account (`newOwner`).
   *
   * Can only be called by the current owner.
   */
  function transferOwnership(address newOwner) public onlyOwner {
    //SlowMist// This check is quite good in avoiding losing control of the contract caused by user mistakes
    require(newOwner != address(0), "Ownable: new owner is the zero address");
    _newOwner = newOwner;
  }
}

/**
 * @dev Enable contract to receive gas token
 */
abstract contract Payable {

  event Deposited(address indexed sender, uint256 value);

  fallback() external payable {
    if(msg.value > 0) {
      emit Deposited(msg.sender, msg.value);
    }
  }

  /// @dev enable wallet to receive ETH
  receive() external payable {
    if(msg.value > 0) {
      emit Deposited(msg.sender, msg.value);
    }
  }
}

/**
 * @dev These functions deal with verification of Merkle trees (hash trees),
 */
library MerkleProof {
  /**
    * @dev Returns true if a `leaf` can be proved to be a part of a Merkle tree
    * defined by `root`. For this, a `proof` must be provided, containing
```

```
    * sibling hashes on the branch from the leaf to the root of the tree. Each
    * pair of leaves and each pair of pre-images are assumed to be sorted.
    */
  function verify(bytes32[] memory proof, bytes32 root, bytes32 leaf) internal pure
returns (bool) {
    bytes32 computedHash = leaf;

    for (uint256 i = 0; i < proof.length; i++) {
      bytes32 proofElement = proof[i];

      if (computedHash <= proofElement) {
        // Hash(current computed hash + current element of the proof)
        computedHash = keccak256(abi.encodePacked(computedHash, proofElement));
      } else {
        // Hash(current element of the proof + current computed hash)
        computedHash = keccak256(abi.encodePacked(proofElement, computedHash));
      }
    }

    // Check if the computed hash (root) is equal to the provided root
    return computedHash == root;
  }
}

/**
 * @dev Coin98Vault contract to enable vesting funds to investors
 */
contract Coin98Vault is Ownable, Payable {

  address private _factory;
  address[] private _admins;
  mapping(address => bool) private _adminStatuses;
  mapping(uint256 => EventData) private _eventDatas;
  mapping(uint256 => mapping(uint256 => bool)) private _eventRedemptions;

  /// @dev Initialize a new vault
  /// @param factory_ Back reference to the factory initialized this vault for global
configuration
  /// @param owner_ Owner of this vault
  constructor(address factory_, address owner_) Ownable(owner_) {
    _factory = factory_;
  }

  struct EventData {
```

```
    uint256 timestamp;
    bytes32 merkleRoot;
    address receivingToken;
    address sendingToken;
    uint8 isActive;
  }

  event AdminAdded(address indexed admin);
  event AdminRemoved(address indexed admin);
  event EventCreated(uint256 eventId, EventData eventData);
  event EventUpdated(uint256 eventId, uint8 isActive);
  event Redeemed(uint256 eventId, uint256 index, address indexed recipient, address
indexed receivingToken, uint256 receivingTokenAmount, address indexed sendingToken,
uint256 sendingTokenAmount);
  event Withdrawn(address indexed owner, address indexed recipient, address indexed
token, uint256 value);

  function _setRedemption(uint256 eventId_, uint256 index_) private {
    _eventRedemptions[eventId_][index_] = true;
  }

  /// @dev Access Control, only owner and admins are able to access the specified
function
  modifier onlyAdmin() {
    require(owner() == _msgSender() || _adminStatuses[_msgSender()], "Ownable: caller
is not an admin");
    _;
  }

  /// @dev returns current admins who can manage the vault
  function admins() public view returns (address[] memory) {
    return _admins;
  }

  /// @dev returns info of an event
  /// @param eventId_ ID of the event
  function eventInfo(uint256 eventId_) public view returns (EventData memory) {
    return _eventDatas[eventId_];
  }

  /// @dev address of the factory
  function factory() public view returns (address) {
    return _factory;
  }
```

```
  /// @dev check an index whether it's redeemed
  /// @param eventId_ event ID
  /// @param index_ index of redemption pre-assigned to user
  function isRedeemed(uint256 eventId_, uint256 index_) public view returns (bool) {
    return _eventRedemptions[eventId_][index_];
  }

  /// @dev claim the token which user is eligible from schedule
  /// @param eventId_ event ID
  /// @param index_ index of redemption pre-assigned to user
  /// @param recipient_ index of redemption pre-assigned to user
  /// @param receivingAmount_ amount of *receivingToken* user is eligible to redeem
  /// @param sendingAmount_ amount of *sendingToken* user must send the contract to
get *receivingToken*
  /// @param proofs additional data to validate that the inputted information is
valid
  function redeem(uint256 eventId_, uint256 index_, address recipient_, uint256
receivingAmount_, uint256 sendingAmount_, bytes32[] calldata proofs) public payable {
    uint256 fee = IVaultConfig(_factory).fee();
    uint256 gasLimit = IVaultConfig(_factory).gasLimit();
    if(fee > 0) {
      require(_msgValue() == fee, "C98Vault: Invalid fee");
    }

    EventData storage eventData = _eventDatas[eventId_];
    require(eventData.isActive > 0, "C98Vault: Invalid event");
    require(eventData.timestamp <= block.timestamp, "C98Vault: Schedule locked");
    require(recipient_ != address(0), "C98Vault: Invalid schedule");

    bytes32 node = keccak256(abi.encodePacked(index_, recipient_, receivingAmount_,
sendingAmount_));
    require(MerkleProof.verify(proofs, eventData.merkleRoot, node), "C98Vault:
Invalid proof");
    require(!isRedeemed(eventId_, index_), "C98Vault: Redeemed");

    uint256 availableAmount;
    if(eventData.receivingToken == address(0)) {
      availableAmount = address(this).balance;
    } else {
      availableAmount = IERC20(eventData.receivingToken).balanceOf(address(this));
    }

    require(receivingAmount_ <= availableAmount, "C98Vault: Insufficient token");
```

```
    _setRedemption(eventId_, index_);
    if(fee > 0) {
      uint256 reward = IVaultConfig(_factory).ownerReward();
      uint256 finalFee = fee - reward;
      (bool success, bytes memory data) = _factory.call{value:finalFee, gas:gasLimit}
("");
      require(success, "C98Vault: Unable to charge fee");
    }
    if(sendingAmount_ > 0) {
      IERC20(eventData.sendingToken).transferFrom(_msgSender(), address(this),
sendingAmount_);
    }
    if(eventData.receivingToken == address(0)) {
      recipient_.call{value:receivingAmount_, gas:gasLimit}("");
    } else {
      IERC20(eventData.receivingToken).transfer(recipient_, receivingAmount_);
    }

    emit Redeemed(eventId_, index_, recipient_, eventData.receivingToken,
receivingAmount_, eventData.sendingToken, sendingAmount_);
  }

  /// @dev withdraw the token in the vault, no limit
  /// @param token_ address of the token, use address(0) to withdraw gas token
  /// @param destination_ recipient address to receive the fund
  /// @param amount_ amount of fund to withdaw
    //SlowMist// The admin role can call the withdraw function to transfer any tokens
of the vault to the specified user.
  function withdraw(address token_, address destination_, uint256 amount_) public
onlyAdmin {
    //SlowMist// This kind of check is very good, avoiding user mistake leading to
the loss of token during transfer
    require(destination_ != address(0), "C98Vault: Destination is zero address");

    uint256 availableAmount;
    if(token_ == address(0)) {
      availableAmount = address(this).balance;
    } else {
      availableAmount = IERC20(token_).balanceOf(address(this));
    }

    require(amount_ <= availableAmount, "C98Vault: Not enough balance");
```

```
    uint256 gasLimit = IVaultConfig(_factory).gasLimit();
    if(token_ == address(0)) {
      destination_.call{value:amount_, gas:gasLimit}("");
    } else {
      IERC20(token_).transfer(destination_, amount_);
    }

    emit Withdrawn(_msgSender(), destination_, token_, amount_);
  }

  //SlowMist// The admin role can call the withdrawNft function to transfer any Nfts of the vault to the specified user.
  /// @dev withdraw NFT from contract
  /// @param token_ address of the token, use address(0) to withdraw gas token
  /// @param destination_ recipient address to receive the fund
  /// @param tokenId_ ID of NFT to withdraw
  function withdrawNft(address token_, address destination_, uint256 tokenId_) public onlyAdmin {
    //SlowMist// This kind of check is very good, avoiding user mistake leading to the loss of token during transfer
    require(destination_ != address(0), "C98Vault: destination is zero address");

    IERC721(token_).transferFrom(address(this), destination_, tokenId_);

    emit Withdrawn(_msgSender(), destination_, token_, 1);
  }

  /// @dev create an event to specify how user can claim their token
  /// @param eventId_ event ID
  /// @param timestamp_ when the token will be available for redemption
  /// @param receivingToken_ token user will be receiving, mandatory
  /// @param sendingToken_ token user need to send in order to receive *receivingToken_*
  function createEvent(uint256 eventId_, uint256 timestamp_, bytes32 merkleRoot_, address receivingToken_, address sendingToken_) public onlyAdmin {
    require(_eventDatas[eventId_].timestamp == 0, "C98Vault: Event existed");
    require(timestamp_ != 0, "C98Vault: Invalid timestamp");
    _eventDatas[eventId_].timestamp = timestamp_;
    _eventDatas[eventId_].merkleRoot = merkleRoot_;
    _eventDatas[eventId_].receivingToken = receivingToken_;
    _eventDatas[eventId_].sendingToken = sendingToken_;
    _eventDatas[eventId_].isActive = 1;

    emit EventCreated(eventId_, _eventDatas[eventId_]);
```

```
  }

  /// @dev enable/disable a particular event
  /// @param eventId_ event ID
  /// @param isActive_ zero to inactive, any number to active
  function setEventStatus(uint256 eventId_, uint8 isActive_) public onlyAdmin {
    require(_eventDatas[eventId_].timestamp != 0, "C98Vault: Invalid event");
    _eventDatas[eventId_].isActive = isActive_;

    emit EventUpdated(eventId_, isActive_);
  }

  /// @dev add/remove admin of the vault.
  /// @param nAdmins_ list to address to update
  /// @param nStatuses_ address with same index will be added if true, or remove if
false
  /// admins will have access to all tokens in the vault, and can define vesting
schedule
  function setAdmins(address[] memory nAdmins_, bool[] memory nStatuses_) public
onlyOwner {
    require(nAdmins_.length != 0, "C98Vault: Empty arguments");
    require(nStatuses_.length != 0, "C98Vault: Empty arguments");
    require(nAdmins_.length == nStatuses_.length, "C98Vault: Invalid arguments");

    uint256 i;
    for(i = 0; i < nAdmins_.length; i++) {
      address nAdmin = nAdmins_[i];
      if(nStatuses_[i]) {
        if(!_adminStatuses[nAdmin]) {
          _admins.push(nAdmin);
          _adminStatuses[nAdmin] = nStatuses_[i];
          emit AdminAdded(nAdmin);
        }
      } else {
        uint256 j;
        for(j = 0; j < _admins.length; j++) {
          if(_admins[j] == nAdmin) {
            _admins[j] = _admins[_admins.length - 1];
            _admins.pop();
            delete _adminStatuses[nAdmin];
            emit AdminRemoved(nAdmin);
            break;
          }
        }
```

```
      }
    }
  }
}

contract Coin98VaultFactory is Ownable, Payable, IVaultConfig {

  uint256 private _fee;
  uint256 private _gasLimit;
  uint256 private _ownerReward;
  address[] private _vaults;

  constructor () Ownable(_msgSender()) {
    _gasLimit = 9000;
  }

  /// @dev Emit `FeeUpdated` when a new vault is created
  event Created(address indexed vault);
  /// @dev Emit `FeeUpdated` when fee of the protocol is updated
  event FeeUpdated(uint256 fee);
  /// @dev Emit `OwnerRewardUpdated` when reward for vault owner is updated
  event OwnerRewardUpdated(uint256 fee);
  /// @dev Emit `Withdrawn` when owner withdraw fund from the factory
  event Withdrawn(address indexed owner, address indexed recipient, address indexed
token, uint256 value);

  /// @dev get current protocol fee in gas token
  function fee() override external view returns (uint256) {
    return _fee;
  }

  /// @dev limit gas to send native token
  function gasLimit() override external view returns (uint256) {
    return _gasLimit;
  }

  /// @dev get current owner reward in gas token
  function ownerReward() override external view returns (uint256) {
    return _ownerReward;
  }

  /// @dev get list of vaults initialized through this factory
  function vaults() external view returns (address[] memory) {
    return _vaults;
```

```
  }

  /// @dev create a new vault
  /// @param owner_ Owner of newly created vault
  function createVault(address owner_) external returns (Coin98Vault vault) {
    vault = new Coin98Vault(address(this), owner_);
    _vaults.push(address(vault));
    emit Created(address(vault));
  }

  function setGasLimit(uint256 limit_) public onlyOwner {
    _gasLimit = limit_;
  }

  /// @dev change protocol fee
  /// @param fee_ amount of gas token to charge for every redeem. can be ZERO to
disable protocol fee
  /// @param reward_ amount of gas token to incentive vault owner. this reward will
be deduce from protocol fee
  function setFee(uint256 fee_, uint256 reward_) public onlyOwner {
    require(fee_ >= reward_, "C98Vault: Invalid reward amount");

    _fee = fee_;
    _ownerReward = reward_;

    emit FeeUpdated(fee_);
    emit OwnerRewardUpdated(reward_);
  }

  //SlowMist// The admin role can call the withdraw function to transfer any tokens
in the vault to the specified user.
  /// @dev withdraw fee collected for protocol
  /// @param token_ address of the token, use address(0) to withdraw gas token
  /// @param destination_ recipient address to receive the fund
  /// @param amount_ amount of fund to withdaw
  function withdraw(address token_, address destination_, uint256 amount_) public
onlyOwner {
    //SlowMist// This kind of check is very good, avoiding user mistake leading to
the loss of token during transfer
    require(destination_ != address(0), "C98Vault: Destination is zero address");

    uint256 availableAmount;
    if(token_ == address(0)) {
      availableAmount = address(this).balance;
```

```
    } else {
      availableAmount = IERC20(token_).balanceOf(address(this));
    }

    require(amount_ <= availableAmount, "C98Vault: Not enough balance");

    if(token_ == address(0)) {
      destination_.call{value:amount_, gas:_gasLimit}("");
    } else {
      IERC20(token_).transfer(destination_, amount_);
    }

    emit Withdrawn(_msgSender(), destination_, token_, amount_);
  }

  //SlowMist// The admin role can call the withdrawNft function to transfer any Nfts of the vault to the specified user.
  /// @dev withdraw NFT from contract
  /// @param token_ address of the token, use address(0) to withdraw gas token
  /// @param destination_ recipient address to receive the fund
  /// @param tokenId_ ID of NFT to withdraw
  function withdrawNft(address token_, address destination_, uint256 tokenId_) public onlyOwner {
    //SlowMist// This kind of check is very good, avoiding user mistake leading to the loss of token during transfer
    require(destination_ != address(0), "C98Vault: destination is zero address");

    IERC721(token_).transferFrom(address(this), destination_, tokenId_);

    emit Withdrawn(_msgSender(), destination_, token_, 1);
  }
}
```

Statement

SlowMist issues this report with reference to the facts that have occurred or existed before the issuance of this report, and only assumes corresponding responsibility based on these.

For the facts that occurred or existed after the issuance, SlowMist is not able to judge the security status of this project, and is not responsible for them. The security audit analysis and other contents of this report are based on the documents and materials provided to SlowMist by the information provider till the date of the insurance report (referred to as "provided information"). SlowMist assumes: The information provided is not missing, tampered with, deleted or concealed. If the information provided is missing, tampered with, deleted, concealed, or inconsistent with the actual situation, the SlowMist shall not be liable for any loss or adverse effect resulting therefrom. SlowMist only conducts the agreed security audit on the security situation of the project and issues this report. SlowMist is not responsible for the background and other conditions of the project.

Official Website

www.slowmist.com

E-mail

team@slowmist.com

Twitter

@SlowMist_Team

Github

https://github.com/slowmist