SLOWMIST

Smart Contract Security Audit Report

Contents

1. Executive Summary

On Jan. 6th, 2021, the SlowMist security team received the Umbrella team's security audit application for overture, developed the audit plan according to the agreement of both parties and the characteristics of the project, and finally issued the security audit report.

The SlowMist security team adopts the strategy of "white box lead, black, grey box assists" to conduct a complete security test on the project in the way closest to the real attack.

SlowMist Smart Contract project test method:

Black box testing	Conduct security tests from an attacker's perspective externally.
Grey box testing	Conduct security testing on code module through the scripting tool, observing the internal running status, mining weaknesses.
White box testing	Based on the open source code, non-open source code, to detect whether there are vulnerabilities in programs such as nodes, SDK, etc.

SlowMist Smart Contract project risk level:

Critical vulnerabilities	Critical vulnerabilities will have a significant impact on the security of the project, and it is strongly recommended to fix the critical vulnerabilities.
High-risk vulnerabilities	High-risk vulnerabilities will affect the normal operation of project. It is strongly recommended to fix high-risk vulnerabilities.
Medium-risk vulnerabilities	Medium vulnerability will affect the operation of project. It is recommended to fix medium-risk vulnerabilities.

Low-risk vulnerabilities	Low-risk vulnerabilities may affect the operation of project in certain scenarios. It is suggested that the project party should evaluate and consider whether these vulnerabilities need to be fixed.
Weaknesses	There are safety risks theoretically, but it is extremely difficult to reproduce in engineering.
Enhancement Suggestions	There are better practices for coding or architecture.

2. Audit Methodology

Our security audit process for smart contract includes two steps:

- Smart contract codes are scanned/tested for commonly known and more specific vulnerabilities using public and in-house automated analysis tools.
- Manual audit of the codes for security issues. The contracts are manually analyzed to look for any potential problems.

Following is the list of commonly known vulnerabilities that was considered during the audit of the smart contract:

- Reentrancy attack and other Race Conditions
- Replay attack
- Reordering attack
- Short address attack
- Denial of service attack
- Transaction Ordering Dependence attack
- Conditional Completion attack
- Authority Control attack
- Integer Overflow and Underflow attack

- TimeStamp Dependence attack
- Gas Usage, Gas Limit and Loops
- Redundant fallback function
- Unsafe type Inference
- Explicit visibility of functions state variables
- Logic Flaws
- Uninitialized Storage Pointers
- Floating Points and Numerical Precision
- tx.origin Authentication
- "False top-up" Vulnerability
- Scoping and Declarations

3. Project Background

3.1 Project Introduction

This repo includes token smart contract (main UMB and rewards rUMB)

as well as distribution strategies for:

- auction

- DeFi farming

- linear vesting.

Audit version file information

Initial audit files:

https://github.com/umbrella-network/overture

commit: 4bca11a73a3996e65aabf0fc04733410dba9923d (develop branch)

Final audit files:

https://github.com/umbrella-network/overture commit: 23778f165bc00a0906ae0aa81d119849982d4c56 (develop branch)

3.2 Project Structure

```
├── Auction.sol
├── Rewards.sol
├── StakingRewards.sol
├── UMB.sol
├── UmbMultiSig.sol
├── auction
│   └── AuctionDummyToken.sol
├── interfaces
│   ├── IBurnableToken.sol
│   ├── IStakingRewards.sol
│   ├── ISwapReceiver.sol
│   ├── MintableToken.sol
│   ├── Owned.sol
│   ├── Pausable.sol
│   ├── PowerMultiSig.sol
│   ├── RewardsDistributionRecipient.sol
│   ├── SelfDestructible.sol
│   └── SwappableToken.sol
├── lib
│   └── Strings.sol
└── rUMB.sol
```

4. Code Overview

4.1 Main Contract address

The contract has not yet been deployed on the mainnet.

4.2 Contracts Description

The SlowMist Security team analyzed the visibility of major contracts during the audit, the result as follows:

Auction			
Function Name	Visibility	Mutability	Modifiers
_max	Internal	-	-
_sendEth	Private	Can modify state	-
tokenPrice	Public	-	-
centsToWeiPrice	Public	-	-
umbFor	Public	-	-
wasAuctionSuccessful	Public	-	-
isAuctionOver	Public	-	-
unsoldUMB	Public	-	-
	External	Payable	-
withdraw	External	Can modify state	whenAuctionInProgress
claim	External	Can modify state	whenAuctionOver
setup	External	Can modify state	onlyOwner onlyBeforeAuction
stop	External	Can modify state	-
start	External	Can modify state	onlyOwner
withdrawETH	External	Can modify state	onlyOwner whenAuctionOver
withdrawUMB	External	Can modify state	onlyOwner whenAuctionOver
burnUnsoldUMB	External	Can modify state	onlyOwner whenAuctionOver

AuctionDummyToken			
Function Name	Visibility	Mutability	Modifiers
burn	External	Can modify state	-
mint	External	Can modify state	-

StakingRewards			
Function Name	Visibility	Mutability	Modifiers
totalSupply	Public	-	-
balanceOf	External	-	-
lastTimeRewardApplicable	Public	-	-

rewardPerToken	Public	-	-
earned	Public	-	-
getRewardForDuration	External	-	-
stake	External	Can modify state	nonReentrant notPaused updateReward
withdraw	Public	Can modify state	nonReentrant updateReward
getReward	Public	Can modify state	nonReentrant updateReward
exit	External	Can modify state	NO
notifyRewardAmount	External	Can modify state	onlyRewardsDistribution updateReward
setRewardsDuration	External	Can modify state	onlyOwner

UMB			
Function Name	Visibility	Mutability	Modifiers
setRewardTokens	External	Can modify state	onlyOwner
swapMint	External	Can modify state	assertMaxSupply

UmbMultiSig			
Function Name	Visibility	Mutability	Modifiers
createFunctionSignature	Public	-	-
submitTokenMintTx	Public	Can modify state	-
submitSetRewardTokensTx	Public	Can modify state	-
submitStartSwapNowTx	Public	Can modify state	-
submitSetRewardsDistributionTx	Public	Can modify state	-
submitSetRewardsDistributionTx	Public	Can modify state	-
submitSetRewardsDurationTx	Public	Can modify state	-
submitNotifyRewardAmountTx	Public	Can modify state	-

Rewards			
Function Name	Visibility	Mutability	Modifiers
balanceOf	Public	-	-
claim	External	Can modify state	-

startDistribution	External	Can modify state	onlyOwner

4.3 Code Audit

4.3.1 High-risk vulnerabilities

4.3.1.1 Arbitrary Minting Token Issue

Any user can mint ADT tokens through the mint function. This leads to the risk of missing permissions for the mint function.

Fix suggestions: It is suggested to add permission control for the mint function.

Code location: auction/AuctionDummyToken.sol

```
function mint(address _holder, uint256 _amount) external {
    require(_amount > 0, "_amount is empty");
    _mint(_holder, _amount);
}
```

Fix status: Project party reply: lets ignore this, as name says its just dummy token we will not use it anywhere-only for tesing.

4.3.2 Medium-risk vulnerabilities

4.3.2.1 totalCurrentPower is not counted correctly

When the wallet calls the addOwner function to increase the owner role, totalCurrentPower does not increase correspondingly. When the owner role is removed, totalCurrentPower is reduced.

Fix suggestions: It is recommended that when the wallet calls the addOwner function to increase the owner role, totalCurrentPower should also be increased accordingly:

totalCurrentPower = totalCurrentPower.add(_power);

Code location: interfaces/PowerMutiSig.sol

```
function addOwner(address _owner, uint _power)
public
onlyWallet
whenOwnerDoesNotExist(_owner)
notNull(_owner)
validRequirement(owners.length + 1, totalCurrentPower + _power, requiredPower)
{
  require(_power != 0, "_power is empty");

  ownersPowers[_owner] = _power;
  owners.push(_owner);
  emit LogOwnerAddition(_owner, _power);
}

function removeOwner(address _owner) public onlyWallet whenOwnerExists(_owner)
{
  uint ownerPower = ownersPowers[_owner];
  require(
    totalCurrentPower - ownerPower >= requiredPower,
    "can't remove owner, because there will be not enough power left"
  );

  ownersPowers[_owner] = 0;
  totalCurrentPower = totalCurrentPower.sub(ownerPower);

  for (uint i=0; i<owners.length - 1; i++) {
    if (owners[i] == _owner) {
      owners[i] = owners[owners.length - 1];
      break;
    }
  }

  owners.pop();

  // if (requiredPower > owners.length) {
  //     changeRequiredPower(requiredPower - ownerPower);
  // }
```

```
    emit LogOwnerRemoval(_owner);
  }
```

Fix status: Fixed.

4.3.2.2 Wrong periodFinish check

In the StakingRewards contract, when the user calls the stake function to perform a stake operation, it checks whether the value of periodFinish is 0. When the RewardsDistribution role sets reward through the notifyRewardAmount function, periodFinish will not be equal to 0, and the user will no longer be able to perform stake operations.

Fix suggestions: The user should check whether periodFinish is greater than 0 when performing a stake operation.

Code location: StakingRewards.sol

```
  function stake(uint256 amount) override external nonReentrant notPaused updateReward(msg.sender) {
    require(periodFinish == 0, "Stake period not started yet");

    require(amount > 0, "Cannot stake 0");
    _totalSupply = _totalSupply.add(amount);
    _balances[msg.sender] = _balances[msg.sender].add(amount);
    stakingToken.safeTransferFrom(msg.sender, address(this), amount);

    emit Staked(msg.sender, amount);
  }
```

Fix status: Fixed.

4.3.3 Low-risk vulnerabilities

4.3.3.1 Design flaws

Users can burn their own tokens through the burn function. The burn function allows the user to pass in the `_amount` parameter and checks the `_amount` parameter, but when the _burn function is finally called to perform the burn operation, all the user's balance (balance) is passed in. This will mislead the user and make the user think that the user's specified amount (_amount) of tokens is burned, but in fact all the user's balance is burned.

Fix suggestions: It is suggested that only the amount specified by the user be burned.

Code location: auction/MintableToken.sol

```
function burn(uint256 _amount) override external {
    uint balance = balanceOf(msg.sender);
    require(_amount <= balance, "not enough tokens to burn");

    _burn(msg.sender, balance);
    maxAllowedTotalSupply = maxAllowedTotalSupply - balance;
}
```

Fix status: Fixed.

4.3.3.2 Auction logic flaws

During the auction, it was not checked whether totalEthLocked was greater than maximumLockedEth.

Fix suggestions: It is suggested that the purchase of UMB is prohibited when totalEthLocked is greater than or equal to maximumLockedEth.

Code location: Auction.sol

```
receive() external payable {
    require(auctionEndsAt > 0, "The auction has not started yet");
    require(auctionEndsAt >= block.timestamp, "The auction has already ended");

    totalEthLocked = totalEthLocked.add(msg.value);
    balances[msg.sender] = balances[msg.sender].add(msg.value);

    emit LogReceive(msg.sender, msg.value);
}
```

Fix status: Fixed.

4.3.4 Enhancement Suggestions

4.3.4.1 PowerMultiSig can set super owner

In this multisig contract, if you want to execute a transaction, you need to confirm that the total power of all Owners who provide signatures is greater than the required Power.

But the power of the owner can be set when the contract is initialized and by calling the addOwner function. If the power value set for a certain owner role exceeds the requiredPower, the owner role will have super authority, and the transaction can be executed only with the owner's own signature.

Fix suggestions: When setting the owner's power value, you should check whether it is greater than requiredPower.

Code location: interfaces/PowerMutiSig.sol

```
function isConfirmed(uint _transactionId) public view returns (bool) {
    uint power = 0;

    for (uint i=0; i<owners.length; i++) {
        if (confirmations[_transactionId][owners[i]]) {
            power += ownersPowers[owners[i]];
        }

        if (power >= requiredPower) {
            return true;
        }
    }

constructor(address[] memory _owners, uint256[] memory _powers, uint256 _requiredPower)
validRequirement(_owners.length, sum(_powers), _requiredPower)
{
    uint sumOfPowers = 0;

    for (uint i=0; i<_owners.length; i++) {
        require(ownersPowers[_owners[i]] == 0, "owner already exists");
        require(_owners[i] != address(0), "owner is empty");
        require(_powers[i] != 0, "power is empty");

        ownersPowers[_owners[i]] = _powers[i];
        sumOfPowers = sumOfPowers.add(_powers[i]);
    }

    owners = _owners;
    requiredPower = _requiredPower;
    totalCurrentPower = sumOfPowers;
}

function addOwner(address _owner, uint _power)
public
onlyWallet
whenOwnerDoesNotExist(_owner)
notNull(_owner)
validRequirement(owners.length + 1, totalCurrentPower + _power, requiredPower)
{
    require(_power != 0, "_power is empty");

    ownersPowers[_owner] = _power;
    owners.push(_owner);
    emit LogOwnerAddition(_owner, _power);
}
```

Fix status: Project party reply: this is requirement, it will stay that way. we want to have "super owner".

5. Audit Result

5.1 Conclusion

Audit Result : Passed

Audit Number : 0X002101130003

Audit Date : Jan. 13, 2021

Audit Team : SlowMist Security Team

Summary conclusion: The SlowMist security team use a manual and SlowMist Team analysis tool audit of the codes for security issues. There are six security issues found during the audit. There are one high-risk vulnerabilities, two medium-risk vulnerabilities and two low-risk vulnerabilities. We also provide one enhancement suggestions. After communication with the project party, all issues have been fixed or the risks are within the acceptable range.

6. 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 base 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 this 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.

SLOWMIST

Official Website
www.slowmist.com

E-mail
team@slowmist.com

Twitter
@SlowMist_Team

Github
https://github.com/slowmist