Smart Contract Security Audit Report

Contents

1. Executive Summary

On Nov. 04, 2020, the SlowMist security team received the WePiggy team's security audit application for WePiggy, 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 DeFi 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 DeFi project risk level:

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

Low-risk vulnerabilities	Low-risk vulnerabilities may affect the operation of DeFi 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

WePiggy is an open source, non-custodial crypto asset lending market protocol. In WePiggy's market, users can deposit their crypto assets to earn interest, or borrow others by paying interests.

Project website:

https://wepiggy.com

Audit version code:

https://github.com/WePiggy/wepiggy-contracts/tree/599e854a5cab44de2355a1bcda09b170c20

ccc35/contracts

The documents provided by the WePiggy team are as follows:

https://www.yuque.com/zgryhn/fg3t76/qccll7

Fixed version code:

https://github.com/WePiggy/wepiggy-contracts/tree/38c1e240578e11621b665007592a21d2c8e

611ec

4. Code Overview

4.1 Contracts Description

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

PiggyBreeder			
Function Name	Visibility	Mutability	Modifiers
poolLength	External	-	-
usersLength	External	-	-
setDevAddr	Public	Can modify state	-
setMigrator	Public	Can modify state	onlyOwner
setEnableClaimBlock	Public	Can modify state	onlyOwner
setReduceIntervalBlock	Public	Can modify state	onlyOwner
setAllocPoint	Public	Can modify state	onlyOwner
setReduceRate	Public	Can modify state	onlyOwner
setDevMiningRate	Public	Can modify state	onlyOwner
replaceMigrate	Public	Can modify state	onlyOwner
migrate	Public	Can modify state	onlyOwner
safePiggyTransfer	Internal	Can modify state	-
getPiggyPerBlock	Public	-	-
getMultiplier	Public	-	-
allPendingPiggy	External	-	-
pendingPiggy	External	-	-
_pending	Internal	-	-
massUpdatePools	Public	Can modify state	-
updatePool	Public	Can modify state	-
add	Public	Can modify state	onlyOwner
stake	Public	Can modify state	-

unStake	Public	Can modify state	-
claim	Public	Can modify state	-
emergencyWithdraw	Public	Can modify state	-

FundingHolder			
Function Name	Visibility	Mutability	Modifiers
transfer	Public	Can modify state	onlyOwner

FundingManager			
Function Name	Visibility	Mutability	Modifiers
safePiggyTransfer	Internal	Can modify state	-
addFunding	Public	Can modify state	onlyOwner
setFunding	Public	Can modify state	onlyOwner
getPendingBalance	Public	-	-
claim	Public	Can modify state	-

WePiggyToken			
Function Name	Visibility	Mutability	Modifiers
mint	Public	Can modify state	-
_transfer	Internal	Can modify state	-
delegates	External	-	-
delegate	External	Can modify state	-
delegateBySig	External	Can modify state	-
getCurrentVotes	External	-	-
getPriorVotes	External	-	-
_delegate	Internal	Can modify state	-
_moveDelegates	Internal	Can modify state	-
_writeCheckpoint	Internal	Can modify state	-
safe32	Internal	-	-
getChainId	Internal	-	-

Timelock			
Function Name	Visibility	Mutability	Modifiers
setDelay	Public	Can modify state	-
acceptAdmin	Public	Can modify state	-
setPendingAdmin	Public	Can modify state	-
queueTransaction	Public	Can modify state	-
cancelTransaction	Public	Can modify state	-
executeTransaction	Public	Payable	-
getBlockTimestamp	Internal	-	-
receive()	External	Payable	-

4.2 Contract Information

Contract Name	Contract Address
WePiggyToken	0x4C78015679FabE22F6e02Ce8102AFbF7d93794eA
FundingHolder[InsurancePayment]	0xb205d0AeF84C666FBBe441C61DC04fEb844444E6
FundingHolder[ResourceExpansion]	0x7a603D06007fc09f896Fb75644365AB091A7b91a
FundingHolder[TeamVote]	0x8C14A40c488E16b055bc4e250563B5480047f01E
FundingHolder[TeamSpending]	0x209EE6f924a39BdDC9A57c0E263dd5E29CEAc78A
FundingHolder[CommunityRewards]	0x1fe0F15546858d9A1E84CE2E7908b160608267c5
FundingManager	0xf18D727C034f47AE2C0FE221C1cf4A15f0557b5F
PiggyBreeder	0x451032C55F813338b6e73c1c4B24217614165454
TimeLock	0x311aEA58Ca127B955890647413846E351df32554

4.3 Code Audit

4.3.1 High-risk vulnerabilities

4.3.1.1 Delegation Double Spending Bug

There has a delegation double spending bug, The governance contract allows token holders to give voting power to a delegatee. but there has a bug, voting power stays with the delegatee even when the token holder transfers the tokens from the wallet. In this case, the voting delegation should disappear. Instead, the voting power can be inflated with delegate & transfer transactions.

- wepiggy-contracts/contracts/token/WePiggyToken.sol

```
// It is suggested to fix the delegation double spending bug by using the following code:
function _transfer(address sender, address recipient, uint256 amount) internal override virtual {
        super._transfer(sender, recipient, amount);
        _moveDelegates(_delegates[sender], _delegates[recipient], amount);
    }
```

Fix Status: The issues has been fixed in this commit:

da7b30e5098721119962eb5678a7d7b0f37434a4

4.3.1.2 Business logic error

The migration did not save the user's "rewardDebt", and set the user's rewardDebt to 0. This is a business logic error that can affect the user's historical "rewardDebt". It is suggested that the user's reward be retained during the migration.

- contracts/farm/PiggyBreeder.sol

```
function migrate(uint256 _pid, uint256 _targetPid) public onlyOwner {

        PoolInfo storage pool = poolInfo[_pid];
        IMigrator migrator = pool.migrator;
        require(address(migrator) != address(0), "migrate: no migrator");

        IERC20 lpToken = pool.lpToken;
        uint256 bal = lpToken.balanceOf(address(this));
        lpToken.safeApprove(address(migrator), bal);
        (IERC20 newLpToken, uint mintBal) = migrator.migrate(lpToken);

        PoolInfo storage targetPool = poolInfo[_targetPid];
        IERC20 targetToken = targetPool.lpToken;
        require(address(targetToken) == address(newLpToken), "migrate: bad");

        uint rate = mintBal.mul(1e12).div(bal);
        for (uint i = 0; i < userAddresses[_pid].length; i++) {

            updatePool(_targetPid);

            address addr = userAddresses[_pid][i];
            UserInfo storage user = userInfo[_pid][addr];
            UserInfo storage tUser = userInfo[_targetPid][addr];

            uint tmp = user.amount.mul(rate).div(1e12);

            tUser.amount = tUser.amount.add(tmp);
            targetPool.totalDeposit = targetPool.totalDeposit.add(tmp);
            pool.totalDeposit = pool.totalDeposit.sub(user.amount);
            user.rewardDebt = 0;
            user.amount = 0;
        }

    }
```

Fix Status: The issues has been fixed in this commit:

38c1e240578e11621b665007592a21d2c8e611ec

4.3.2 Medium-risk vulnerabilities

4.3.2.1 DoS Issue

There is no limit on the number of user addresses. When the number of addresses is too large, The for loop will be out of gas and unable to execute normally. It is suggested to avoid out of gas by batching or using algorithms.

- contracts/farm/PiggyBreeder.sol   Line: 196

```
for (uint i = 0; i < userAddresses[_pid].length; i++) {
```

```
            updatePool(_targetPid);

            address addr = userAddresses[_pid][i];
            UserInfo storage user = userInfo[_pid][addr];
            UserInfo storage tUser = userInfo[_targetPid][addr];

            uint tmp = user.amount.mul(rate).div(1e12);

            tUser.amount = tUser.amount.add(tmp);
            targetPool.totalDeposit = targetPool.totalDeposit.add(tmp);
            pool.totalDeposit = pool.totalDeposit.sub(user.amount);
            user.rewardDebt = 0;
            user.amount = 0;
        }

    }
```

Fix Status: The issues has been Incomplete fixed in this commit:

38c1e240578e11621b665007592a21d2c8e611ec

4.3.3 Low-risk vulnerabilities

4.3.3.1 Excessive authority auditing

The owner in PiggyBreeder contract can add a new lpToken through the add function, but if there is a black swan event, such as the addition of a malicious lpToken, there will be useless lpToken to recharge to get rewards. It is suggested that the owner can be handed over to the governance contract or time lock contract for management.

- wepiggy-contracts/contracts/farm/PiggyBreeder.sol   Line: 358-379

```
function add(uint256 _allocPoint, IERC20 _lpToken, IMigrator _migrator, bool _withUpdate) public onlyOwner {

        if (_withUpdate) {
            massUpdatePools();
        }
        uint256 lastRewardBlock = block.number > startBlock ? block.number : startBlock;

        //update totalAllocPoint
        totalAllocPoint = totalAllocPoint.add(_allocPoint);

        // add poolInfo
        poolInfo.push(PoolInfo({
        lpToken : _lpToken,
        allocPoint : _allocPoint,
        lastRewardBlock : lastRewardBlock,
        accPiggyPerShare : 0,
        totalDeposit : 0,
        migrator : _migrator
        }));
    }
```

Owner can set devaddr, set migrator, set enable claimBlock etc. It is suggested that the owner can be handed over to the governance contract or time lock contract for management.

- wepiggy-contracts/contracts/farm/PiggyBreeder.sol

```
    // Update dev address by the previous dev.
    function setDevAddr(address _devAddr) public onlyOwner {
        devAddr = _devAddr;
    }

    // Set the migrator contract. Can only be called by the owner.
    function setMigrator(uint256 _pid, IMigrator _migrator) public onlyOwner {
        poolInfo[_pid].migrator = _migrator;
    }

    // set the enable claim block
    function setEnableClaimBlock(uint256 _enableClaimBlock) public onlyOwner {
        enableClaimBlock = _enableClaimBlock;
    }

    // update reduceIntervalBlock
    function setReduceIntervalBlock(uint256 _reduceIntervalBlock, bool _withUpdate) public onlyOwner {
        if (_withUpdate) {
            massUpdatePools();
        }
        reduceIntervalBlock = _reduceIntervalBlock;
    }

    // Update the given pool's PIGGY allocation point. Can only be called by the owner.
    function setAllocPoint(uint256 _pid, uint256 _allocPoint, bool _withUpdate) public onlyOwner {
        if (_withUpdate) {
            massUpdatePools();
        }
        //update totalAllocPoint
        totalAllocPoint = totalAllocPoint.sub(poolInfo[_pid].allocPoint).add(_allocPoint);

        //update poolInfo
        poolInfo[_pid].allocPoint = _allocPoint;
    }

    // update reduce rate
    function setReduceRate(uint256 _reduceRate, bool _withUpdate) public onlyOwner {
        if (_withUpdate) {
            massUpdatePools();
        }
        reduceRate = _reduceRate;
    }

    // update dev mining rate
    function setDevMiningRate(uint256 _devMiningRate) public onlyOwner {
        devMiningRate = _devMiningRate;
    }

    // Migrate lp token to another lp contract.
    function replaceMigrate(uint256 _pid) public onlyOwner {
        PoolInfo storage pool = poolInfo[_pid];
        IMigrator migrator = pool.migrator;
        require(address(migrator) != address(0), "migrate: no migrator");

        IERC20 lpToken = pool.lpToken;
        uint256 bal = lpToken.balanceOf(address(this));
        lpToken.safeApprove(address(migrator), bal);
        (IERC20 newLpToken, uint mintBal) = migrator.replaceMigrate(lpToken);

        require(bal == newLpToken.balanceOf(address(this)), "migrate: bad");
        pool.lpToken = newLpToken;
        emit ReplaceMigrate(address(migrator), _pid, bal);
    }

    // Move lp token data to another lp contract.
    function migrate(uint256 _pid, uint256 _targetPid, uint256 begin) public onlyOwner {

        require(begin < userAddresses[_pid].length, "migrate: begin error");

        PoolInfo storage pool = poolInfo[_pid];
        IMigrator migrator = pool.migrator;
        require(address(migrator) != address(0), "migrate: no migrator");

        IERC20 lpToken = pool.lpToken;
        uint256 bal = lpToken.balanceOf(address(this));
        lpToken.safeApprove(address(migrator), bal);
        (IERC20 newLpToken, uint mintBal) = migrator.migrate(lpToken);

        PoolInfo storage targetPool = poolInfo[_targetPid];
        require(address(targetPool.lpToken) == address(newLpToken), "migrate: bad");

        uint rate = mintBal.mul(1e12).div(bal);
        for (uint i = begin; i < begin.add(20); i++) {

            if (i < userAddresses[_pid].length) {
                updatePool(_targetPid);

                address addr = userAddresses[_pid][i];
                UserInfo storage user = userInfo[_pid][addr];
                UserInfo storage tUser = userInfo[_targetPid][addr];

                if (user.amount <= 0) {
                    continue;
                }

                uint tmp = user.amount.mul(rate).div(1e12);

                tUser.amount = tUser.amount.add(tmp);
                tUser.rewardDebt = tUser.rewardDebt.add(user.rewardDebt.mul(rate).div(1e12));
                targetPool.totalDeposit = targetPool.totalDeposit.add(tmp);
                pool.totalDeposit = pool.totalDeposit.sub(user.amount);
                user.rewardDebt = 0;
                user.amount = 0;
            } else {
                break;
            }

        }

        emit Migrate(address(migrator), _pid, _targetPid, bal);

    }
```

Fix Status: The owner authority has been transferred to the timelock contract.

Reference:

https://cn.etherscan.com/tx/0x902e0eaf5251249b03eb0ad354b7d058dbb8a4ee44407b640babd

28d045a0ca2

4.3.4 Enhancement Suggestions

4.3.4.1 Enhancement Point of DelegateBySig Function

The nonce in the delegateBySig function is input by the user. When the user input a larger nonce, the current transaction cannot be success but the relevant signature data will still remain on the chain, causing this signature to be available for some time in the future. It is recommended to fix it according to EIP-2612.

Reference: https://github.com/ethereum/EIPs/blob/master/EIPS/eip-2612.md#implementation.

- wepiggy-contracts/contracts/token/WePiggyToken.sol   Line: 105-146

```
function delegateBySig(
        address delegatee,
        uint nonce,
        uint expiry,
        uint8 v,
        bytes32 r,
        bytes32 s
    )
    external
    {
        bytes32 domainSeparator = keccak256(
            abi.encode(
                DOMAIN_TYPEHASH,
                keccak256(bytes(name())),
                getChainId(),
                address(this)
            )
        );

        bytes32 structHash = keccak256(
            abi.encode(
                DELEGATION_TYPEHASH,
                delegatee,
                nonce,
                expiry
            )
        );

        bytes32 digest = keccak256(
            abi.encodePacked(
                "\x19\x01",
                domainSeparator,
                structHash
            )
        );

        address signatory = ecrecover(digest, v, r, s);
        require(signatory != address(0), "WePiggyToken::delegateBySig: invalid signature");
        require(nonce == nonces[signatory]++, "WePiggyToken::delegateBySig: invalid nonce");
        require(now <= expiry, "WePiggyToken::delegateBySig: signature expired");
        return _delegate(signatory, delegatee);
```

Fix Status: This issues has been ignore.

5. Audit Result

5.1 Conclusion

Audit Result : Passed

Audit Number : 0X002011110001

Audit Date : Nov. 11, 2020

Audit Team : SlowMist Security Team

Summary conclusion: The SlowMist security team use a manual and SlowMist team's analysis tool to audit the project, 2 high-risk, 1 medium-risk, 1 low-risk vulnerabilities were found during the audit, the high-risk, medium-risk and low-risk vulnerabilities identified have been fixed, the owner authority has been transferred to the timelock contract. The delay time is 48 hours. There is an enhancement suggestion has been ignore.

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.