Smart Contract Security Audit Report

Table Of Contents

1 Executive Summary

On 2024.01.26, the SlowMist security team received the Ring Protocol team's security audit application for Ring Protocol Contracts, 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.

The test method information:

Test method	Description
Black box testing	Conduct security tests from an attacker's perspective externally.
Grey box testing	Conduct security testing on code modules 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.

The vulnerability severity level information:

Level	Description
Critical	Critical severity vulnerabilities will have a significant impact on the security of the DeFi project, and it is strongly recommended to fix the critical vulnerabilities.
High	High severity vulnerabilities will affect the normal operation of the DeFi project. It is strongly recommended to fix high-risk vulnerabilities.
Medium	Medium severity vulnerability will affect the operation of the DeFi project. It is recommended to fix medium-risk vulnerabilities.
Low	Low severity vulnerabilities may affect the operation of the DeFi project in certain scenarios. It is suggested that the project team should evaluate and consider whether these vulnerabilities need to be fixed.
Weakness	There are safety risks theoretically, but it is extremely difficult to reproduce in engineering.
Suggestion	There are better practices for coding or architecture.

2 Audit Methodology

The security audit process of SlowMist security team for smart contract includes two steps:

- Smart contract codes are scanned/tested for commonly known and more specific vulnerabilities using 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:

Serial Number	Audit Class	Audit Subclass
1	Overflow Audit	-
2	Reentrancy Attack Audit	-
3	Replay Attack Audit	-
4	Flashloan Attack Audit	-
5	Race Conditions Audit	Reordering Attack Audit
6	Permission Vulnerability Audit	Access Control Audit
		Excessive Authority Audit
7	Security Design Audit	External Module Safe Use Audit
		Compiler Version Security Audit
		Hard-coded Address Security Audit
		Fallback Function Safe Use Audit
		Show Coding Security Audit
		Function Return Value Security Audit
		External Call Function Security Audit

Serial Number	Audit Class	Audit Subclass
7	Security Design Audit	Block data Dependence Security Audit
		tx.origin Authentication Security Audit
8	Denial of Service Audit	-
9	Gas Optimization Audit	-
10	Design Logic Audit	-
11	Variable Coverage Vulnerability Audit	-
12	"False Top-up" Vulnerability Audit	-
13	Scoping and Declarations Audit	-
14	Malicious Event Log Audit	-
15	Arithmetic Accuracy Deviation Audit	-
16	Uninitialized Storage Pointer Audit	-

3 Project Overview

3.1 Project Introduction

This is the Ring Protocol which includes the Factory, WrappedToken, and Staking Rewards parts. The Factory part can help users to create the WrappedToken contract. The WrappedToken contract is wrapping ERC20 tokens into a new format, enabling interaction within a specific decentralized ecosystem. The Staking Rewards contract is designed for staking and reward distribution in decentralized finance (DeFi) applications. Utilizing the ERC20 token standard, facilitates users to stake specific tokens and earn rewards over time.

3.2 Vulnerability Information

The following is the status of the vulnerabilities found in this audit:

NO	Title	Category	Level	Status
N1	Risk of excessive authority	Authority Control Vulnerability Audit	High	Acknowledged
N2	Token compatibility reminder	Others	Suggestion	Acknowledged
N3	Potential token decimal compatibility reminder	Others	Suggestion	Acknowledged
N4	Missing the event records	Others	Suggestion	Fixed
N5	Missing the 0 address check	Others	Suggestion	Fixed
N6	Malleable attack risk	Replay Vulnerability	Suggestion	Acknowledged
N7	Risk of replay attack	Replay Vulnerability	Suggestion	Acknowledged

4 Code Overview

4.1 Contracts Description

Audit Version:

https://github.com/Few-Protocol/few-core-contracts

commit: ce67bd361a9c6c2b044511176304db53dccc3c45

Fixed Version:

https://github.com/Few-Protocol/few-core-contracts

commit: f03679727dc3d86856c1a3dabede0ce890e3ba28

The main network address of the contract is as follows:

The code was not deployed to the mainnet.

4.2 Visibility Description

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

FewFactory			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	CoreRef
allWrappedTokensLength	External	-	-
paused	Public	-	-
createToken	External	Can Modify State	-
claimMaxGas	External	Can Modify State	onlyGovernor

CoreRef			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	-
setCore	External	Can Modify State	onlyGovernor
pause	Public	Can Modify State	onlyGuardianOrGovernor
unpause	Public	Can Modify State	onlyGuardianOrGovernor
core	Public	-	-

FewWrappedToken			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	-
_mint	Internal	Can Modify State	-
_burn	Internal	Can Modify State	-
_approve	Internal	Can Modify State	-
_transfer	Private	Can Modify State	-

FewWrappedToken			
approve	External	Can Modify State	-
transfer	External	Can Modify State	-
transferFrom	External	Can Modify State	-
permit	External	Can Modify State	-
mint	External	Can Modify State	onlyMinter whenNotPaused
burn	Public	Can Modify State	-
burnFrom	Public	Can Modify State	onlyBurner whenNotPaused
wrapTo	Public	Can Modify State	-
wrap	External	Can Modify State	-
unwrapTo	Public	Can Modify State	-
unwrap	External	Can Modify State	-
claimMaxGas	External	Can Modify State	onlyGovernor

FixedStakingRewards			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	-
rewardPerTokenPerSecond	External	-	-
totalSupply	External	-	-
periodFinish	Public	-	-
lastTimeRewardApplicable	Public	-	-
lastUpdateTimeOf	External	-	-
balanceOf	External	-	-
rewardOf	External	-	-

FixedStakingRewards			
earned	Public	-	-
stakeWithPermit	External	Can Modify State	nonReentrant updateReward
stake	External	Can Modify State	nonReentrant updateReward
withdraw	Public	Can Modify State	nonReentrant updateReward
getReward	Public	Can Modify State	nonReentrant updateReward
exit	External	Can Modify State	-
deploy	External	Can Modify State	-
setRewardPerTokenPerSecond	External	Can Modify State	-
setPeriodFinish	External	Can Modify State	-
setRewardSetter	External	Can Modify State	-
claimMaxGas	External	Can Modify State	-

4.3 Vulnerability Summary

[N1] [High] Risk of excessive authority

Category: Authority Control Vulnerability Audit

Content

1.In the FewWrappedToken contract, the burner role can burn any users' Wrapped tokens through the burnFrom function without users' approval. All role settings are completed in the core contract, which is not within the scope of this audit.

Code location:

FewWrappedToken.sol#160-168

```
    function burnFrom(address account, uint256 amount)
        public
        override
        onlyBurner
        whenNotPaused
    {
        _burn(account, amount);
        emit Burn(msg.sender, amount, account);
    }
```

2.In the FewWrappedToken contract, the minter role can mint tokens arbitrarily through the mint function and there is no upper limit for the mint amount of tokens.

Code location:

FewWrappedToken.sol#140-148

```
    function mint(address account, uint256 amount)
        external
        override
        onlyMinter
        whenNotPaused
    {
        _mint(account, amount);
        emit Mint(msg.sender, amount, account);
    }
```

3.In the FixedStakingRewards contract, the rewardSetter can arbitrarily modify every `rewardPerTokenPerSecond`, `periodFinish`, and `rewardSetter` parameters in each StakingInfo.

Code location:

FixedStakingRewards.sol#165-182

```
    function setRewardPerTokenPerSecond(uint256 index, uint256
 _rewardPerTokenPerSecond) external override {
        require(msg.sender == rewardSetter, 'Ring: FORBIDDEN');
        StakingInfo storage info = stakingInfos[index];
        require(info.stakingToken != address(0), 'FixedStakingRewards::set: not
 deployed yet');
        info.rewardPerTokenPerSecond = _rewardPerTokenPerSecond;
    }

    function setPeriodFinish(uint256 index, uint256 _periodFinish) external override
 {
        require(msg.sender == rewardSetter, 'Ring: FORBIDDEN');
        StakingInfo storage info = stakingInfos[index];
        require(info.stakingToken != address(0), 'FixedStakingRewards::set: not
 deployed yet');
        info.periodFinish = _periodFinish;
    }
    function setRewardSetter(address _rewardSetter) external override {
        require(msg.sender == rewardSetter, 'Ring: FORBIDDEN');
        rewardSetter = _rewardSetter;
    }
```

Solution

1. It’s recommended to add the allowance part from the approval of the burnFrom function or only allow the users to burn their own Wrapped tokens through the burn function.

In the short term, transferring owner ownership to multisig contracts is an effective solution to avoid single-point risk. But in the long run, it is a more reasonable solution to implement a privilege separation strategy and set up multiple privileged roles to manage each privileged function separately. The authority involving user funds should be managed by the community, and the authority involving emergency contract suspension can be managed by the EOA address. This ensures both a quick response to threats and the safety of user funds.

Status

Acknowledged; After communicating with the project team, they expressed that the minter and burner will be timelock that is governed by DAO. The DAO will need to mint token to some specific address and also burn token from some specific address when necessary.

[N2] [Suggestion] Token compatibility reminder

Category: Others

Content

In the FixedStakingRewards contract, users can stake, stakeWithPermit, and withdraw the stakeingTokens by safetransferFrom and safetransfer functions to the staking contract and the amount will be directly recorded in the totalSupply. If the stakingTokens are deflationary tokens, the actual amount of tokens received by the FixedStakingRewards contract will be less than the amount recorded by the amount parameter.

Code location:

FixedStakingRewards.sol#90-122

```
    function stakeWithPermit(uint256 index, uint256 amount, uint deadline, uint8 v,
  bytes32 r, bytes32 s) external override nonReentrant updateReward(index, msg.sender)
```

```
{
        …
        info.totalSupply = info.totalSupply.add(amount);
        info.balances[msg.sender] = info.balances[msg.sender].add(amount);
        …
        emit Staked(index, info.stakingToken, msg.sender, amount);
    }

    function stake(uint256 index, uint256 amount) external override nonReentrant
updateReward(index, msg.sender) {
        …
        info.totalSupply = info.totalSupply.add(amount);
        info.balances[msg.sender] = info.balances[msg.sender].add(amount);
        IERC20(info.stakingToken).safeTransferFrom(msg.sender, address(this), amount);
        emit Staked(index, info.stakingToken, msg.sender, amount);
    }

    function withdraw(uint256 index, uint256 amount) public override nonReentrant
updateReward(index, msg.sender) {
        …
        info.totalSupply = info.totalSupply.sub(amount);
        info.balances[msg.sender] = info.balances[msg.sender].sub(amount);
        IERC20(info.stakingToken).safeTransfer(msg.sender, amount);
        emit Withdrawn(index, info.stakingToken, msg.sender, amount);
    }
```

Solution

It is recommended to record the difference before and after the user's transfer as the actual amount of the user's staking or add the token whitelist for the stakingTokens.

Status

Acknowledged; After communicating with the project team, they expressed that they will not want any deflationary token to be used in this contract.

[N3] [Suggestion] Potential token decimal compatibility reminder

Category: Others

Content

In the FixedStakingRewards contract, users can stake the tokens through the stake and stakeWithPermit functions. It will update each totalSupply and balances parameters according to the amount of user deposits. These parameters will not distinguish different stakingTokens, if the stakingTokens deposit with different decimals will may lead to errors in the calculation of rewards in the protocol.

Solution

It's recommended to use the same decimal for all the stakingTokens.

Status

Acknowledged

[N4] [Suggestion] Missing the event records

Category: Others

Content

In the FixedStakingRewards contract, the rewardSetter can arbitrarily modify every `rewardPerTokenPerSecond`, `periodFinish`, and `rewardSetter` parameters in each StakingInfo, but there are no event logs.

Code location:

FixedStakingRewards.sol#165-182

```
    function setRewardPerTokenPerSecond(uint256 index, uint256
_rewardPerTokenPerSecond) external override {
        require(msg.sender == rewardSetter, 'Ring: FORBIDDEN');
        StakingInfo storage info = stakingInfos[index];
        require(info.stakingToken != address(0), 'FixedStakingRewards::set: not
deployed yet');
        info.rewardPerTokenPerSecond = _rewardPerTokenPerSecond;
    }

    function setPeriodFinish(uint256 index, uint256 _periodFinish) external override
{
        require(msg.sender == rewardSetter, 'Ring: FORBIDDEN');
        StakingInfo storage info = stakingInfos[index];
        require(info.stakingToken != address(0), 'FixedStakingRewards::set: not
deployed yet');
        info.periodFinish = _periodFinish;
    }

    function setRewardSetter(address _rewardSetter) external override {
        require(msg.sender == rewardSetter, 'Ring: FORBIDDEN');
        rewardSetter = _rewardSetter;
    }
```

Solution

It is recommended to record events when sensitive parameters are modified for self-inspection or community review.

Status

Fixed

[N5] [Suggestion] Missing the 0 address check

Category: Others

Content

In the FixedStakingRewards and CoreRef contract, the Governor role can modify the `_core` address and the rewardSetter can modify the `rewardSetter` address, but there are no 0 address checks.

Code location:

refs/CoreRef.sol#47-50

FixedStakingRewards.sol#179-182

```
    function setCore(address coreAddress) external override onlyGovernor {
        _core = ICore(coreAddress);
        emit CoreUpdate(coreAddress);
    }

    function setRewardSetter(address _rewardSetter) external override {
        require(msg.sender == rewardSetter, 'Ring: FORBIDDEN');
        rewardSetter = _rewardSetter;
    }
```

Solution

It is recommended to add the 0 address check when sensitive parameters are modified.

Status

Fixed

[N6] [Suggestion] Malleable attack risk

Category: Replay Vulnerability

Content

In the permit function of the FewWrappedToken contract, it restores the address of the signer through the ecrecover function, but does not check the value of v and s. Since EIP2 still allows the malleability for ecrecover, this will lead to the risk of transaction malleability attacks.

Code location:

FewWrappedToken.sol#123-135

```
    function permit(address owner, address spender, uint value, uint deadline, uint8
v, bytes32 r, bytes32 s) external override {
        require(deadline >= block.timestamp, 'Few: EXPIRED');
        bytes32 digest = keccak256(
            abi.encodePacked(
                '\x19\x01',
                DOMAIN_SEPARATOR,
                keccak256(abi.encode(PERMIT_TYPEHASH, owner, spender, value,
nonces[owner]++, deadline))
            )
        );
        address recoveredAddress = ecrecover(digest, v, r, s);
        require(recoveredAddress != address(0) && recoveredAddress == owner, 'Few:
INVALID_SIGNATURE');
        _approve(owner, spender, value);
    }
```

Solution

It is recommended to use the ECDSA library of openzeppelin to check the signature.

https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/utils/cryptography/ECDSA.sol

Status

Acknowledged

[N7] [Suggestion] Risk of replay attack

Category: Replay Vulnerability

Content

DOMAIN_SEPARATOR is defined when the contract is initialized, but it is not reimplemented when DOMAIN_SEPARATOR is used in the permit function. So the DOMAIN_SEPARATOR contains the chainId and is defined at contract deployment instead of reconstructed for every signature, there is a risk of possible replay attacks between chains in the event of a future chain split.

Code location:

FewWrappedToken.sol#57-70

```
    constructor() public {
        uint chainId;
        assembly {
            chainId := chainid()
        }
        DOMAIN_SEPARATOR = keccak256(
            abi.encode(
                keccak256('EIP712Domain(string name,string version,uint256 
chainId,address verifyingContract)'),
                keccak256(bytes(name)),
                keccak256(bytes('1')),
                chainId,
                address(this)
            )
        );
        ...
    }
```

Solution

It is recommended to redefine when using DOMAIN_SEPARATOR.

Status

Acknowledged

5 Audit Result

Audit Number	Audit Team	Audit Date	Audit Result
0X002401300001	SlowMist Security Team	2024.01.26 - 2024.01.30	High Risk

Summary conclusion: The SlowMist security team uses manual and SlowMist team's analysis tools to audit the project, during the audit work we found 1 high risk and 6 suggestions. And 1 High risk, 4 suggestions were acknowledged and 2 suggestions were fixed. The code was not deployed to the mainnet.

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 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.