Smart Contract Security Audit Report

Table Of Contents

1 Executive Summary

On 2024.12.13, the SlowMist security team received the Prosper team's security audit application for Prosper Upgraded Token, 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 an audit of the Upgraded Token contract for the Prosper protocol, consisting of two main parts: Token and TrancheManager. The Token contract inherits from OFT token, allowing users to purchase tokens through the TrancheManager contract, and use these tokens for cross-chain transfers and staking operations.

3.2 Vulnerability Information

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

NO	Title	Category	Level	Status
N1	Multiple different roles have mint permissions	Authority Control Vulnerability Audit	Suggestion	Acknowledged
N2	Risks of excessive privilege	Authority Control Vulnerability Audit	Medium	Acknowledged
N3	`requestBurn` only processes the latest request	Others	Information	Acknowledged
N4	Potential DoS risk of batch burning	Denial of Service Vulnerability	Medium	Fixed
N5	Suggestions for code optimization	Others	Suggestion	Fixed
N6	Not following the Checks-Effects-Interactions specification	Reentrancy Vulnerability	Low	Acknowledged
N7	Privileged roles can update tranche information before the tranche deadline	Others	Information	Acknowledged
N8	Redundant approval operations	Design Logic Audit	Suggestion	Fixed
N9	The matured status check for tranche is flawed	Design Logic Audit	Critical	Fixed

4 Code Overview

4.1 Contracts Description

Audit Version:

https://github.com/hyplabs/prosper-hash-token

commit: 1cd92b15f220e0f93e78ad1bcc373b1e73523669

Fixed Version:

https://github.com/hyplabs/prosper-hash-token commit: f199133d335b0b397d0a26d856a71c2e2ab94db6

The main network address of the contract is as follows:

Contract Name	Contract Address	Chain
Token Proxy	0x915424Ac489433130d92B04096F3b96c82e92a9D	BNB Smart Chain
Token Impl	0xc062db6a83FaC421232E5931542D10B4950AD00D	BNB Smart Chain
Token Proxy	0x915424Ac489433130d92B04096F3b96c82e92a9D	Ethereum
Token Impl	0xfC7d4Bc36Cbf73c854f401077d57fa15b8eb8Fe3	Ethereum

4.2 Visibility Description

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

Token			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	OFTUpgradeable
_authorizeUpgrade	Internal	Can Modify State	onlyRole
__Token_init	External	Can Modify State	initializer
batchMint	External	Can Modify State	onlyOwner
batchExecuteBurn	External	Can Modify State	onlyRole
executeBurn	External	Can Modify State	onlyRole
mint	External	Can Modify State	onlyRole whenNotPaused
pause	External	Can Modify State	onlyOwner
requestBurn	External	Can Modify State	-
send	External	Payable	whenNotPaused
swapToken	External	Can Modify State	whenNotPaused

Token			
unpause	External	Can Modify State	onlyOwner
withdrawSwappables	External	Can Modify State	onlyOwner
getBurnRequest	External	-	-
getPendingBurnRequests	External	-	-
getPendingBurnRequestsCount	External	-	-

TrancheManager			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	-
__TrancheManager_init	External	Can Modify State	initializer
_authorizeUpgrade	Internal	Can Modify State	onlyOwner
claim	External	Can Modify State	whenNotPaused
claimAndBridge	External	Payable	whenNotPaused
claimAndStake	External	Can Modify State	whenNotPaused
collectProceeds	External	Can Modify State	onlyOwner
initializeTranche	External	Can Modify State	onlyOwner
pause	External	Can Modify State	whenNotPaused onlyOwner
purchaseTokens	External	Can Modify State	whenNotPaused
refundCommitment	External	Can Modify State	whenNotPaused
setNewMinerFund	External	Can Modify State	onlyOwner
setSaleToken	External	Can Modify State	onlyOwner
setStakingPool	External	Can Modify State	onlyOwner
terminateTranche	External	Can Modify State	onlyOwner

TrancheManager			
unpause	External	Can Modify State	whenPaused onlyOwner
updateTranche	External	Can Modify State	onlyOwner
getActiveTrancheIds	External	-	-
getActiveTranches	External	-	-
getBoughtIds	External	-	-
getCommittedPayments	External	-	-
getCurrentTranchePrice	External	-	-
getCurrentTrancheSupply	External	-	-
getNewMinerFund	External	-	-
getSaleToken	External	-	-
getStakingPool	External	-	-
getTranche	External	-	-
getUserTrancheStatus	External	-	-
isTrancheRefundable	External	-	-
_bridge	Internal	Can Modify State	-
_killTranche	Internal	Can Modify State	-
_updateTrancheDeadline	Internal	Can Modify State	-
_updateTrancheDiscountPrice	Internal	Can Modify State	-
_updateTrancheDiscountThreshold	Internal	Can Modify State	-
_updateTrancheGatedStatus	Internal	Can Modify State	-
_updateTranchePrice	Internal	Can Modify State	-
_updateTrancheSupply	Internal	Can Modify State	-

TrancheManager			
_exactBridgeAmount	Internal	-	-
_currentPrice	Internal	-	-
_currentSupply	Internal	-	-
_validateTrancheInitialization	Internal	-	-
_processMatureTranches	Internal	Can Modify State	-
_isMature	Internal	-	-

4.3 Vulnerability Summary

[N1] [Suggestion] Multiple different roles have mint permissions

Category: Authority Control Vulnerability Audit

Content

In the Token contract, the owner can batch mint tokens to multiple addresses using the batchMint function, while accounts with MINTER_ROLE can mint tokens to specific addresses through the mint function. Granting minting privileges to multiple roles may lead to confusion in permission management.

Code location: contracts/Token.sol#L74,L127

```
function batchMint(
    address[] calldata accounts,
    uint256[] calldata amounts
) external onlyOwner {
    ...
}

function mint(
    address account,
    uint256 amount
) external onlyRole(MINTER_ROLE) whenNotPaused {
    ...
}
```

Solution

It is recommended to delegate identical permissions to a single role for management, while the owner can manage various roles through AccessControl.

Status

Acknowledged; After communicating with the project team, the project team stated that this was the expected design.

[N2] [Medium] Risks of excessive privilege

Category: Authority Control Vulnerability Audit

Content

In the Token contract, the owner role can arbitrarily mint OFT tokens through the batchMint function, and accounts with MINTER_ROLE can also perform arbitrary minting via the mint function. This poses a risk of excessive privileges.

Code location: contracts/Token.sol#L74,L127

```
function batchMint(
    address[] calldata accounts,
    uint256[] calldata amounts
) external onlyOwner {
    ...
}

function mint(
    address account,
    uint256 amount
) external onlyRole(MINTER_ROLE) whenNotPaused {
    ...
}
```

Solution

For cross-chain tokens, unlimited minting authority may be necessary. This risk can be effectively mitigated by transferring the management of this privilege to DAO governance. During the project's early stages, delegating the authority to a timelock contract and implementing execution through multi-signature can help mitigate this risk.

Status

Acknowledged; After communication with the project team, they indicated that once the protocol is launched, both the protocol's owner and `DEFAULT_ADMIN` will be controlled by a multisig wallet. However, this does not mitigate the risk of excessive privileges.

[N3] [Information] `requestBurn` only processes the latest request

Category: Others

Content

In the Token contract, users can request token burns through the requestBurn function, with burn requests being recorded in burnRequests. Accounts with BURNER_ROLE can burn users' tokens by reading requests from burnRequests. Notably, when a user submits a new burn request while their previous request is still pending, the new request overwrites the earlier one - meaning burnRequests only maintains the user's most recent burn request. Users cannot submit multiple burn requests until their pending request has been processed.

Code location: contracts/Token.sol#L147

```
    function requestBurn(uint256 amount) external {
        Storage.Layout storage $ = Storage.layout();

        if (amount == 0) {
            revert RequestBurn__AmountIsZero();
        }

        uint256 balance = balanceOf(_msgSender());
        uint256 burnAmount = amount > balance ? balance : amount;

        $.burnRequests.set(_msgSender(), burnAmount);

        emit BurnRequested(_msgSender(), burnAmount);
    }
```

Solution

N/A

Status

Acknowledged

[N4] [Medium] Potential DoS risk of batch burning

Category: Denial of Service Vulnerability

Content

In the Token contract, users can request token burns through the requestBurn function with a user-specified amount, and these burn requests are recorded in burnRequests. Importantly, when users submit burn requests, the contract neither locks nor transfers the to-be-burned tokens into the current contract. This allows users to continue transferring tokens that are pending burn. When a user's balance falls below the requested burn amount, accounts with BURNER_ROLE will be unable to execute the burn operation for these users. If malicious users exploit this by submitting numerous burn requests, they could render getPendingBurnRequests unusable.

Code location: contracts/Token.sol#L137-L150

```
function requestBurn(uint256 amount) external {
    Storage.Layout storage $ = Storage.layout();

    if (amount == 0) {
        revert RequestBurn__AmountIsZero();
    }

    uint256 balance = balanceOf(_msgSender());
    uint256 burnAmount = amount > balance ? balance : amount;

    $.burnRequests.set(_msgSender(), burnAmount);

    emit BurnRequested(_msgSender(), burnAmount);
}
```

Solution

It is recommended to transfer users' tokens to the current contract when they submit burn requests, and have the BURNER_ROLE burn tokens from the contract's balance when executing the burn operation.

Status

Fixed

[N5] [Suggestion] Suggestions for code optimization

Category: Others

Content

1.In the TrancheManager contract, users can claim tokens and perform cross-chain operations through the claimAndBridge function. The claimable token amount is calculated via the `_processMatureTranches` function. However, the contract proceeds with subsequent operations without verifying if the claimable amount is greater than 0, which may result in unnecessary gas consumption.

2.In the TrancheManager contract, the `_updateTranchePrice` function is used to update `tranche.price`. Notably, it updates `tranche.price` regardless of whether the new price is different from the old price. If the new price is identical to the old price, the update is unnecessary.

Code location:

contracts/TrancheManager.sol#L91,L130

```
function claimAndBridge(
    uint256 percentageBP,
    uint16 dstChainId,
    MessagingFee memory fee
)
    external
    payable
    whenNotPaused
    returns (uint256 claimed, uint256 bridged)
{
    ...
    claimed = _processMatureTranches(_msgSender());
    ...
}

function claimAndStake(
    uint256 percentageBP,
    bytes32[] calldata proof
) external whenNotPaused returns (uint256 claimed, uint256 staked) {
    ...
    claimed = _processMatureTranches(_msgSender());
    staked = claimed;
    ...
}
```

contracts/TrancheManager.sol#L648

```
function _updateTranchePrice(
    Tranche storage tranche,
    uint256 newPrice
) internal {
```

```
        if (tranche.price != newPrice) {
            ...
        }

        tranche.price = newPrice;
    }
```

Solution

1.Add a check in claimAndBridge to verify that claimed is greater than 0 before proceeding with subsequent operations.

2.Only update the price when the new price differs from the old price.

Status

Fixed

[N6] [Low] Not following the Checks-Effects-Interactions specification

Category: Reentrancy Vulnerability

Content

In the TrancheManager contract, the owner role can withdraw paymentTokens belonging to matured tranches through the collectProceeds function. It first transfers the paymentToken to newMinerFund and then clears the tranche's totalCommittedPayments. This violates the Checks-Effects-Interactions (CEI) pattern. If the paymentToken has hook functionality, this could lead to reentrancy risks. Even though this is a privileged function, measures should still be taken to mitigate this risk.

Code location: contracts/TrancheManager.sol#L167-L173

```
    function collectProceeds() external onlyOwner {
        ...
        for (uint256 i; i < activeTranchesLength; ++i) {
            tranche = $.tranches[activeTrancheIds[i]];

            if (
                tranche.supply <= _currentSupply(tranche) &&
                tranche.deadline < block.timestamp
            ) {
                proceeds = tranche.totalCommittedPayments;

                IERC20(tranche.paymentToken).safeTransfer(
                    $.newMinerFund,
```

```
                    proceeds
                );
                emit ProceedsCollected(activeTrancheIds[i], proceeds);

                delete tranche.totalCommittedPayments;
            }
        }
    }
```

Solution

It is recommended to first set the tranche's totalCommittedPayments to 0, and then perform the token transfer operation.

Status

Acknowledged

[N7] [Information] Privileged roles can update tranche information before the tranche deadline

Category: Others

Content

In the TrancheManager contract, the owner role can update certain key information of a Tranche through the updateTranche function before the Tranche deadline, and the updated data must be beneficial to users. (For example: the new price must be lower than the old price)

Code location: contracts/TrancheManager.sol#L360-L380

```
    function updateTranche(
        uint256 trancheId,
        uint256 newSupply,
        uint256 newPrice,
        uint256 newDiscountPrice,
        uint256 newDiscountThreshold,
        uint256 newDeadline,
        bytes32 newGateRoot,
        bool isGatedStatus
    ) external onlyOwner {
        Tranche storage tranche = Storage.layout().tranches[trancheId];

        _updateTrancheSupply(tranche, newSupply);
        _updateTranchePrice(tranche, newPrice);
        _updateTrancheDiscountPrice(tranche, newDiscountPrice);
        _updateTrancheDiscountThreshold(tranche, newDiscountThreshold);
```

```
        _updateTrancheDeadline(tranche, newDeadline);
        _updateTrancheGatedStatus(tranche, isGatedStatus, newGateRoot);

        emit TrancheUpdated(trancheId, tranche);
    }
```

Solution

N/A

Status

Acknowledged; After communicating with the project team, they indicated that this is the intended design, and the parameter updates will be beneficial to users.

[N8] [Suggestion] Redundant approval operations

Category: Design Logic Audit

Content

In the TrancheManager contract, the `_bridge` function is used for cross-chain operations of users' OFT tokens. It first approves tokens to the saleToken contract before performing the send operation. However, it's important to note that in the OFT's send operation, the `_debit` function directly burns user tokens through the `_burn` function without requiring user allowance. Therefore, the prior approval to the saleToken contract is unnecessary.

Code location:

oft-evm-upgradeable/contracts/oft/OFTUpgradeable.sol#L77

```
    function _debit(
        address _from,
        uint256 _amountLD,
        uint256 _minAmountLD,
        uint32 _dstEid
    ) internal virtual override returns (uint256 amountSentLD, uint256
amountReceivedLD) {
        ...
        _burn(_from, amountSentLD);
    }
```

contracts/TrancheManager.sol#L517

```
function _bridge(
    IToken saleToken,
    SendParam memory sendParams,
    MessagingFee memory fee
) internal {
    saleToken.mint(address(this), sendParams.amountLD);
    saleToken.approve(address(saleToken), sendParams.amountLD);
    saleToken.send{ value: fee.nativeFee }(sendParams, fee, _msgSender());
}
```

Solution

It is recommended to remove this redundant approval operation.

Status

Fixed

[N9] [Critical] The matured status check for tranche is flawed

Category: Design Logic Audit

Content

In the TrancheManager contract, the `_isMature` function checks if a tranche is in a matured state. Specifically, it verifies that the current time must be greater than `tranche.deadline + tranche.lockupPeriod`, and the current supply must be greater than `tranche.supply`. The `_processMatureTranches` function uses this to calculate the amount of sale tokens users can claim.

Unfortunately, the check for refund eligibility has the condition `tranche.supply > _currentSupply(tranche)`. This means that when `_currentSupply(tranche)` equals `tranche.supply`, the tranche will neither qualify for a refund nor meet the matured condition. This results in the tranche being locked, preventing users from retrieving their funds or claiming sale tokens.

Code location: contracts/TrancheManager.sol#L802,L291

```
function _isMature(
    Tranche memory tranche
) internal view returns (bool isMature) {
    isMature =
        tranche.deadline + tranche.lockupPeriod < block.timestamp &&
        tranche.supply < _currentSupply(tranche);
}
```

```
function refundCommitment(
    uint256 trancheId
) external whenNotPaused returns (uint256 refund) {
    Storage.Layout storage $ = Storage.layout();
    Tranche storage tranche = $.tranches[trancheId];

    bool isRefundable = tranche.deadline < block.timestamp &&
        tranche.supply > _currentSupply(tranche);

    if (!isRefundable) {
        revert Refund__TrancheNotRefundable();
    }

    ...
}
```

Solution

It is recommended to modify the check in the `_isMature` function to `tranche.supply <= _currentSupply(tranche)` to avoid this risk.

Status

Fixed

5 Audit Result

Audit Number	Audit Team	Audit Date	Audit Result
0X002412170003	SlowMist Security Team	2024.12.13 - 2024.12.17	Medium Risk

Summary conclusion: The SlowMist security team uses a manual and the SlowMist team's analysis tool to audit the project. During the audit work, we found 1 critical risk, 2 medium risks, 1 low risk, 3 suggestions, and 2 information. All the findings were fixed or acknowledged. The audit finding remains at medium risk since the project team has not yet addressed the excessive privilege risk.

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.

Official Website
www.slowmist.com

E-mail
team@slowmist.com

Twitter
@SlowMist_Team

Github
https://github.com/slowmist