Smart Contract Security Audit Report

Table Of Contents

1 Executive Summary

On 2025.01.09, the SlowMist security team received the Puffer Finance team's security audit application for Puffer Point Token Wrapper, 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 Puffer Point Token Wrapper contract built by Merkl, a partner protocol of Puffer Finance. It allows users to send specified underlying tokens to the Token Wrapper contract to receive wrapper tokens and stake them in the distributor contract. Users can withdraw their underlying tokens after waiting for a specified cliff period. Additionally, the Governor role can withdraw all tokens from the Token Wrapper contract.

The audit also includes the review of the CARROT token, which is a standard ERC20 token with Permit functionality.

3.2 Vulnerability Information

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

NO	Title	Category	Level	Status
N1	Redundant contract import	Others	Suggestion	Fixed
N2	distributionCreator address is not initialized	Design Logic Audit	Low	Fixed
N3	The wrapper token cannot be transferred from distributor to feeRecipient	Design Logic Audit	Suggestion	Acknowledged
N4	Potential denial of service risk for `claim`	Denial of Service Vulnerability	Low	Fixed
N5	Risks of excessive privilege	Authority Control Vulnerability Audit	Medium	Acknowledged
N6	Missing event records	Others	Suggestion	Fixed
N7	No checks were performed when updating sensitive addresses	Design Logic Audit	Suggestion	Fixed
N8	Cannot update distributor and feeRecipient addresses at the same time	Design Logic Audit	Low	Fixed

4 Code Overview

4.1 Contracts Description

Audit Version:

https://github.com/PufferFinance/puffer-contracts/tree/master/partners/merkle-contracts/src

commit: 0b9f95c4790d2c95eec3c3dd96145eade22c6a61

https://etherscan.io/token/0x282a69142bac47855c3fbe1693fcc4ba3b4d5ed6

Fixed Version:

https://etherscan.io/address/0xd0e14173cf140c84b1baf7da4dde1eb1a04b7fe6

The main network address of the contract is as follows:

Contract Name	Contract Address	Chain
CARROT	0x282A69142bac47855C3fbE1693FcC4bA3B4d5Ed6	Ethereum
PufferPointTokenWrapper Proxy	0x8A5A5DE9db5770123Ff2145F59e9F20047f0A8EC	Ethereum
PufferPointTokenWrapper Impl	0xD0E14173cf140c84B1baf7DA4DDe1eb1A04b7fe6	Ethereum

4.2 Visibility Description

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

UUPSHelper			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	initializer

BaseMerklTokenWrapper			
Function Name	Visibility	Mutability	Modifiers
token	Public	-	-
isTokenWrapper	External	-	-
initialize	Public	Can Modify State	initializer onlyProxy
recoverERC20	External	Can Modify State	onlyGovernor
_authorizeUpgrade	Internal	-	onlyGovernorUpgrader

PufferPointTokenWrapper			
Function Name	Visibility	Mutability	Modifiers

PufferPointTokenWrapper			
initialize	Public	Can Modify State	initializer
isTokenWrapper	External	-	-
token	Public	-	-
_beforeTokenTransfer	Internal	Can Modify State	-
_afterTokenTransfer	Internal	Can Modify State	-
claim	External	Can Modify State	-
claimable	External	-	-
getUserVestings	External	-	-
_claimable	Internal	-	-
_authorizeUpgrade	Internal	-	onlyGovernorUpgrader
recoverERC20	External	Can Modify State	onlyGovernor
setDistributor	External	Can Modify State	onlyGovernor
setCliffDuration	External	Can Modify State	onlyGuardian
setFeeRecipient	External	Can Modify State	-

4.3 Vulnerability Summary

[N1] [Suggestion] Redundant contract import

Category: Others

Content

PufferPointTokenWrapper imports both BaseMerklTokenWrapper and IAccessControlManager from `BaseTokenWrapper.sol`, but the imported BaseMerklTokenWrapper contract is not actually used, making it redundant.

Code location: src/PufferPointTokenWrapper.sol#L9

```
import {BaseMerklTokenWrapper, IAccessControlManager} from "./BaseTokenWrapper.sol";
```

Solution

If this is not an intended design, it is recommended to remove the BaseMerklTokenWrapper import to reduce deployment costs.

Status

Fixed

[N2] [Low] distributionCreator address is not initialized

Category: Design Logic Audit

Content

In the PufferPointTokenWrapper contract, the initialize function is used to initialize necessary contract parameters. The `_distributionCreator` parameter it accepts is used to initialize the distributor and feeRecipient addresses, but it is not assigned to the distributionCreator variable. This leaves the distributionCreator address as the zero address after contract deployment, which will render the setFeeRecipient function unusable.

Code location: src/PufferPointTokenWrapper.sol#L63

```
    function initialize(
        address _underlying,
        uint32 _cliffDuration,
        IAccessControlManager _core,
        address _distributionCreator
    ) public initializer {
        __ERC20_init(
            string.concat("Merkl Token Wrapper - ",
IERC20Metadata(_underlying).name()),
            string.concat("mtw", IERC20Metadata(_underlying).symbol())
        );
        __UUPSUpgradeable_init();
        if (address(_core) == address(0)) revert ZeroAddress();
        underlying = _underlying;
        core = _core;
        cliffDuration = _cliffDuration;
        distributor = IDistributionCreator(_distributionCreator).distributor();
        feeRecipient = IDistributionCreator(_distributionCreator).feeRecipient();
    }
```

Solution

If this is not an intended design, it is recommended to initialize the distributionCreator address during contract deployment.

Status

Fixed

[N3] [Suggestion] The wrapper token cannot be transferred from distributor to feeRecipient

Category: Design Logic Audit

Content

In the PufferPointTokenWrapper contract, the `_beforeTokenTransfer` and `_afterTokenTransfer` functions are overridden to implement the minting/burning of wrapped tokens. It's important to note that during token transfers, when `from` is the distributor address and `to` is the feeRecipient address, `_beforeTokenTransfer` will trigger a `_mint(from, amount)` operation, causing the distributor to receive unexpected additional wrapped tokens. Meanwhile, in the `_afterTokenTransfer` function, `_burn(to, amount)` will be triggered twice, which may result in the feeRecipient address not having enough wrapped tokens to burn. This will prevent tokens from being transferred from the distributor to the feeRecipient. Of course, if this business scenario doesn't exist, then this risk won't be present.

Code location:

src/PufferPointTokenWrapper.sol#L94-L97

src/PufferPointTokenWrapper.sol#L101-L106

```
    function _beforeTokenTransfer(address from, address to, uint256 amount) internal
override {
        ...

        // Will be burnt right after, to avoid having any token aside from on the
distributor
        if (to == feeRecipient) {
            IERC20(underlying).safeTransferFrom(from, feeRecipient, amount);
            _mint(from, amount); // These are then transferred to the fee manager
        }
    }

    function _afterTokenTransfer(address from, address to, uint256 amount) internal
```

```
override {
        if (to == feeRecipient) {
            _burn(to, amount); // To avoid having any token aside from on the
distributor
        }

        if (from == distributor) {
            _burn(to, amount);

            ...
        }
    }
```

Solution

If there's a need to support the above business scenario in the future, it is recommended to modify the functions as follows:

In `_beforeTokenTransfer`, only perform the mint operation when the `to` address is feeRecipient and the `from` address is not distributor.

In `_afterTokenTransfer`, only perform the burn operation when the `to` address is feeRecipient and the `from` address is not distributor.

Status

Acknowledged

[N4] [Low] Potential denial of service risk for `claim`

Category: Denial of Service Vulnerability

Content

In the PufferPointTokenWrapper contract, users can claim matured underlying tokens through the `claim` function, which uses the `_claimable` function to calculate the amount of tokens that can be claimed. In the `_claimable` function, a while loop is used to check if the user's vestings have matured. If a user's allVestings list becomes too large, this could lead to a potential DoS (Denial of Service) risk.

Code location: src/PufferPointTokenWrapper.sol#L148

```
    function _claimable(address user) internal view returns (uint256 amountClaimable,
uint256 nextClaimIndex) {
        VestingData storage userVestingData = vestingData[user];
```

```
        VestingID[] storage userAllVestings = userVestingData.allVestings;
        uint256 i = userVestingData.nextClaimIndex;
        uint256 length = userAllVestings.length;
        while (i < length) {
            VestingID storage userCurrentVesting = userAllVestings[i];
            if (block.timestamp > userCurrentVesting.unlockTimestamp) {
                amountClaimable += userCurrentVesting.amount;
                nextClaimIndex = ++i;
            } else {
                break;
            }
        }
    }
```

Solution

It is recommended to either:

Allow users to provide the specific vesting index they want to claim, enabling them to claim unlocked underlying tokens one at a time, or

Allow users to provide a range of vesting indices they wish to claim, to avoid the aforementioned risk.

Status

Fixed

[N5] [Medium] Risks of excessive privilege

Category: Authority Control Vulnerability Audit

Content

In the PufferPointTokenWrapper contract, the Governor role can recover any ERC20 tokens from the contract through the recoverERC20 function. It's important to note that users' underlying tokens are also stored in this contract, which means the Governor role can withdraw user funds at will, creating a privilege escalation risk.

Code location: src/PufferPointTokenWrapper.sol#L181

```
    function recoverERC20(address tokenAddress, address to, uint256 amountToRecover)
external onlyGovernor {
        IERC20(tokenAddress).safeTransfer(to, amountToRecover);
        emit Recovered(tokenAddress, to, amountToRecover);
    }
```

Solution

There are two possible solutions:

1.Exclude the underlying tokens from the scope of tokens that can be recovered through the recoverERC20 function. This would resolve the privilege escalation risk, but would also mean that underlying tokens cannot be rescued in emergency situations.

2.Transfer the Governor's privileges to DAO governance to mitigate this risk.

Status

Acknowledged

[N6] [Suggestion] Missing event records

Category: Others

Content

In the PufferPointTokenWrapper contract, the setDistributor, setCliffDuration, and setFeeRecipient functions are used to set the distributor, distributionCreator, cliffDuration, and feeRecipient variables respectively. However, the contract does not emit events for these changes.

Code location: src/PufferPointTokenWrapper.sol#L185-L197

```
    function setDistributor(address _distributionCreator) external onlyGovernor {
        distributor = IDistributionCreator(_distributionCreator).distributor();
        distributionCreator = _distributionCreator;
    }

    function setCliffDuration(uint32 _newCliffDuration) external onlyGuardian {
        if (_newCliffDuration < cliffDuration && _newCliffDuration != 0) revert
InvalidParam();
        cliffDuration = _newCliffDuration;
    }

    function setFeeRecipient() external {
        feeRecipient = IDistributionCreator(distributionCreator).feeRecipient();
    }
```

Solution

It is recommended to emit events when these sensitive parameters are modified, to facilitate future self-review or community auditing.

Status

Fixed

[N7] [Suggestion] No checks were performed when updating sensitive addresses

Category: Design Logic Audit

Content

In the PufferPointTokenWrapper contract, distributor and feeRecipient addresses are essential for achieving business objectives. However, there are no zero-address checks during the initialization of distributor and feeRecipient addresses. Similarly, when these addresses are modified through the setDistributor and setFeeRecipient functions, there are no checks to prevent setting them to zero addresses. If these addresses are set to zero addresses, it would render the protocol's functionality unusable.

Code location: src/PufferPointTokenWrapper.sol#L74-L75, L186, L196

```
function initialize(
    ...
) public initializer {
    ...
    distributor = IDistributionCreator(_distributionCreator).distributor();
    feeRecipient = IDistributionCreator(_distributionCreator).feeRecipient();
}

function setDistributor(address _distributionCreator) external onlyGovernor {
    distributor = IDistributionCreator(_distributionCreator).distributor();
    distributionCreator = _distributionCreator;
}

function setFeeRecipient() external {
    feeRecipient = IDistributionCreator(distributionCreator).feeRecipient();
}
```

Solution

It is recommended to implement zero-address validation both during initialization and modification of distributor and feeRecipient addresses.

Status

Fixed

[N8] [Low] Cannot update distributor and feeRecipient addresses at the same time

Category: Design Logic Audit

Content

In the PufferPointTokenWrapper contract, the Governor role can update the distributionCreator address through the setDistributor function. When this happens, the contract retrieves the distributor address from the new distributionCreator to update its own distributor address. However, it does not simultaneously update the feeRecipient address. If the new distributionCreator has a different feeRecipient, this could impact the business process.

Code location: src/PufferPointTokenWrapper.sol#L185-L188

```
function setDistributor(address _distributionCreator) external onlyGovernor {
    distributor = IDistributionCreator(_distributionCreator).distributor();
    distributionCreator = _distributionCreator;
}
```

Solution

If this is not the intended design, it is recommended to update both the distributor and feeRecipient addresses simultaneously when the distributionCreator is updated.

Status

Fixed

5 Audit Result

Audit Number	Audit Team	Audit Date	Audit Result
0X002501100002	SlowMist Security Team	2025.01.09 - 2025.01.10	Medium Risk

Summary conclusion: The SlowMist security team uses a manual and SlowMist team's analysis tool to audit the project, during the audit work we found 1 medium risk, 3 low risk, and 4 suggestions. All the findings were acknowledged or fixed. Since the risk of excessive privileges has not yet been addressed, the final audit conclusion remains at medium 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