Smart Contract Security Audit Report

Table Of Contents

1 Executive Summary

2 Audit Methodology

3 Project Overview

3.1 Project Introduction

3.2 Vulnerability Information

4 Code Overview

4.1 Contracts Description

4.2 Visibility Description

4.3 Vulnerability Summary

5 Audit Result

6 Statement

1 Executive Summary

On 2024.04.01, the SlowMist security team received the Particle team's security audit application for Particle Vault Contract, 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

The Particle Protocol is an interest rate swapping protocol that allows LPs to access fixed rate yield until maturity and variable but potentially higher yield compared to the yield source. A single Vault contract houses both liquidity flows. Users can deposit funds into the vault and get yield by these two strategies.

3.2 Vulnerability Information

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

NO	Title	Category	Level	Status

NO	Title	Category	Level	Status
N1	Risk of excessive authority	Authority Control Vulnerability Audit	Medium	Acknowledged
N2	mintFixedRate may fail on the first call and the yield will be 0	Design Logic Audit	Low	Acknowledged
N3	First opposite strategy mint can gain all yield and fee	Design Logic Audit	Low	Acknowledged
N4	Missing the burn logic in the FixedRateNFT	Design Logic Audit	Low	Acknowledged
N5	Receive can lock users' native tokens	Others	Low	Acknowledged
N6	Withdraw can affect the rate and yield	Design Logic Audit	Suggestion	Acknowledged
N7	Missing the approve when calling the burnFrom function	Design Logic Audit	Suggestion	Acknowledged
N8	Redundant code	Others	Suggestion	Acknowledged
N9	Missing the 0 address/value check	Others	Suggestion	Acknowledged
N10	Missing the event records	Others	Suggestion	Fixed
N11	Preemptive Initialization	Race Conditions Vulnerability	Suggestion	Acknowledged
N12	External call reminder	Unsafe External Call Audit	Suggestion	Acknowledged
N13	Successive mint and burn calls to get the yield and fee	Design Logic Audit	Suggestion	Acknowledged

4 Code Overview

4.1 Contracts Description

Audit Version:

https://github.com/Particle-Platforms/vault-contract

commit: 0cc4a4c134bf0566cb55dcf9f47ee40e793fc072

Fixed Version:

https://github.com/Particle-Platforms/vault-contract

commit: c7e529cdfed22033c47bf4b19adf5adc9ed767fb

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:

Vault			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	CoreRef
mintFixedRate	External	Can Modify State	nonReentrant
burnFixedRate	External	Can Modify State	nonReentrant
mintVariableRate	External	Can Modify State	nonReentrant
burnVariableRate	External	Can Modify State	nonReentrant
_getVariableRatePortion	Internal	-	-
estimateYield	External	Can Modify State	nonReentrant
_estimateYield	Internal	Can Modify State	-
getYieldToLock	Public	-	-
getMaxFixedRateAmount	External	-	-

Vault			
getFixedRate	Public	-	-
getBalance	Public	-	-
getTotalYield	Public	-	-
getCurrentVariableRate	External	-	-
updateBlastPointsAdmin	Public	Can Modify State	onlyOwner
configureClaimableGas	External	Can Modify State	onlyOwner
claimMaxGas	External	Can Modify State	onlyOwner nonReentrant
updateYieldManager	External	Can Modify State	onlyGuardianOrGovernor
updateYieldEstimateWindow	External	Can Modify State	onlyGuardianOrGovernor nonReentrant
updateCurve	External	Can Modify State	onlyGuardianOrGovernor nonReentrant
updatePositionFee	External	Can Modify State	onlyGuardianOrGovernor nonReentrant

YieldManager			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	CoreRef
mintYield	External	Can Modify State	nonReentrant onlyVault
burnUnmaturedYield	External	Can Modify State	nonReentrant onlyVault
getIdleYield	Public	-	-
_getMaxIdleYield	Internal	-	-
_updateIdleYield	Internal	Can Modify State	-
depositPrincipal	External	Can Modify State	nonReentrant onlyVault

YieldManager			
withdrawPrincipal	External	Can Modify State	nonReentrant onlyVault
updateRate	External	Can Modify State	onlyGuardianOrGovernor
enableControlled	External	Can Modify State	onlyGuardianOrGovernor
disableControlled	External	Can Modify State	onlyGuardianOrGovernor
updateBlastPointsAdmin	Public	Can Modify State	onlyOwner

CoreRef			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	-
setCore	External	Can Modify State	onlyGovernor
emergency	External	-	-
startEmergency	External	Can Modify State	onlyGuardianOrGovernor
stopEmergency	External	Can Modify State	onlyGuardianOrGovernor
core	Public	-	-

FixedRate			
Function Name	Visibility	Mutability	Modifiers
initialize	External	Can Modify State	nonReentrant
mint	External	Can Modify State	nonReentrant onlyVault
burn	External	Can Modify State	nonReentrant onlyOwner
withdraw	External	Can Modify State	nonReentrant onlyOwner onlyEmergency

VariableRate			
Function Name	Visibility	Mutability	Modifiers

VariableRate			
initialize	External	Can Modify State	nonReentrant
mint	External	Can Modify State	nonReentrant onlyVault
burn	External	Can Modify State	nonReentrant onlyOwner
withdraw	External	Can Modify State	nonReentrant onlyOwner onlyEmergency

WrapMint			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	-
_swap	Internal	Can Modify State	-
mintFixedRate	External	Can Modify State	nonReentrant
mintVariableRate	External	Can Modify State	nonReentrant
addExchange	External	Can Modify State	onlyOwner
removeExchange	External	Can Modify State	onlyOwner
setFixedRateNft	External	Can Modify State	onlyOwner
onERC721Received	External	-	-

FixedRateNFT			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	ERC721
mint	External	Can Modify State	nonReentrant
burn	External	Can Modify State	nonReentrant onlyOwner
withdraw	External	Can Modify State	nonReentrant onlyOwner

Permissions			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	-
createRole	External	Can Modify State	onlyGovernor
grantGovernor	External	Can Modify State	onlyGovernor
grantGuardian	External	Can Modify State	onlyGovernor
revokeGovernor	External	Can Modify State	onlyGovernor
revokeGuardian	External	Can Modify State	onlyGovernor
revokeOverride	External	Can Modify State	onlyGuardian
isGovernor	Public	-	-
isGuardian	Public	-	-
_setupGovernor	Internal	Can Modify State	-

Core			
Function Name	Visibility	Mutability	Modifiers
init	External	Can Modify State	initializer

4.3 Vulnerability Summary

[N1] [Medium] Risk of excessive authority

Category: Authority Control Vulnerability Audit

Content

1.In the Vault contract, the Guardian Or the Governor roles can modify the `yieldManager` address, the `rateMapping`, and the `positionFeeRate` parameters. These parameters can affect the calculation of the yield rate and the recording of principal and yield.

Code location:

Vault.sol#360-383

```
    function updateYieldManager(address newYieldManager) external override
onlyGuardianOrGovernor {
        yieldManager = newYieldManager;
        emit UpdateYieldManager(newYieldManager);
    }

    function updateCurve(
        uint256 s1,
        uint256 s2,
        uint256 s3,
        uint256 r1,
        uint256 r2
    ) external override onlyGuardianOrGovernor nonReentrant {
        require(r1 <= r2, "Vault: invalid curve");
        rateMapping = RateMapping(s1, s2, s3, r1, r2);
        emit UpdateCurve(s1, s2, s3, r1, r2);
    }

    function updatePositionFeeRate(uint256 fee) external override
onlyGuardianOrGovernor nonReentrant {
        require(fee < BASIS, "Vault: invalid position fee");
        positionFeeRate = fee;
        emit UpdatePositionFeeRate(fee);
    }
```

2.In the YieldManager contract, the Guardian Or the Governor roles can modify the `vault` address and the `positionFeeRate` parameters. These parameters can affect the calculation of the yield rate and the calculation of principal and yield.

Code location:

YieldManager.sol#95-106

```
    function updateVault(address newVault) external onlyGuardianOrGovernor {
        vault = newVault;
    }

    function updateRate(uint256 rate) external onlyGuardianOrGovernor {
        _updateIdleYield();
        yieldRate = rate;
    }
```

3.In the WrapMint contract, the owner role can add or remove the whitelistedExchanges and modify the fixedRateNft address.

Code location:

periphery/WrapMint.sol#307-325

```
function addExchange(address exchange) external onlyOwner {
    whitelistedExchanges[exchange] = true;
    emit UpdateExchange(exchange, true);
}

function removeExchange(address exchange) external onlyOwner {
    whitelistedExchanges[exchange] = false;
    emit UpdateExchange(exchange, false);
}

function setFixedRateNft(address nft) external onlyOwner {
    fixedRateNft = nft;
    emit UpdateFixedRateNft(nft);
}

function setVariableRateNft(address nft) external onlyOwner {
    variableRateNft = nft;
    emit UpdateVariableRateNft(nft);
}
```

4.In the CoreRef contract, the Governor role can set the core contract, and the Guardian Or the Governor roles can modify the EMERGENCY to true or false. And in the Permissions contract, the Governor role can create roles as the Governor role or the Guardian and also can grant or revoke these roles.

Code location:

core/CoreRef.sol#36-39, 45-51

core/Permissions.sol#35-71

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

function startEmergency() external override onlyGuardianOrGovernor {
    EMERGENCY = true;
}
```

```
function stopEmergency() external override onlyGuardianOrGovernor {
    EMERGENCY = false;
}
```

Solution

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 after deployment, they will transfer governor from the deployer EOA to multisig co-signed by Particle and Ring.

[N2] [Low] mintFixedRate may fail on the first call and the yield will be 0

Category: Design Logic Audit

Content

In the Vault contract, the user can choose between a fixed rate strategy and a variable rate strategy for yield by selecting two different mint functions. However, the first time a deposit call is made to mintFixedRate in a Vault contract, the contract is called and updated through a series of fieldManager contract deposits before any other principal and yield support is available. At this point, since the first `_updateIdleYield` update operation is performed and the getIdleYield function is called to update the controlledIdleYield as well as the principal. but the actual value of these two parameters is 0. This is followed by an update to the `prevTimestamp` parameter which is the current `block.timestamp`. In the call path of the mintFixedRate function, the `lockedYield` is obtained by calling the getYieldToLock function, which in turn calls the getIdleYield function of the yieldManager contract. The lockedYield obtained will then have a `lockedYield` of 0 due to the previously updated prevTimestamp value, and if the first user sets the minLockedYield value to greater than 0 on the first call, then the call will fail.

Code location:

Vault.sol#62-115, 312-343

YieldManager.sol#54-56, 66-69, 76-79

```
        lockedYield = getYieldToLock(amount);

    function getIdleYield() public view override returns (uint256) {
        return controlledIdleYield + ((principal * (block.timestamp - prevTimestamp) *
yieldRate) / BASIS);
    }
```

Solution

The first user to make a mintFixedRate function call needs to pay attention to the minLockedYield input and no yield of this user.

Status

Acknowledged; After communicating with the project team, they expressed that their frontend will surface the fixed term yield that this user will get. It will be zero to start, should they take the action.

[N3] [Low] First opposite strategy mint can gain all yield and fee

Category: Design Logic Audit

Content

There are two strategies in the project, one is VariableRate and the other is FixedRate, which is a fixed rate to get the returns. When the mint is initially done through the FixedRate strategy, the earnings are updated through the `_updateIdleYield` function and the `getIdleYield` function. At this point, when a user makes a deposit via VariableRate, there is no record of the update of the `variableRatePrincipal.principalTimespan` and `variableRatePrincipal.principal` parameters. The user can then call `burnVariableRate` immediately after making a deposit to get the yield and principal. At this time, the `amountTimespan` and the `variableRateTotalPrincipalTimespan` calculated by the `_getVariableRatePortion` function are the same, and the yield is the `getIdle` parameter of `getIdlePortion`. This means the yield is the total amount calculated by `getIdleYield` and the total fee is obtained by IERC20(TOKEN).balanceOf(address(this)). Similarly, the reverse situation is also similar, when all the deposits are made by VariableRate in the previous period, the first time the mint deposit is made by FixedRate, getFixedRate is used. The first time a mint deposit is made via FixedRate, the fixed

rate value obtained by `getFixedRate` will be equal to `BASIS`, and `yieldToLock` will be the result of the deposited `principal * MATURITY`. However, mintFixedRate has a yield limitation, `require(lockedYield <= IYieldManager(yieldManager).getIdleYield(), "Vault: overspend");`, and if there is insufficient existing yield to calculate the resulting lockedYield then the deposit is not supported.

Vault.sol#215-263, 275-293

```
    function burnVariableRate(
        uint256 amount,
        uint256 minYield
    ) external override nonReentrant returns (uint256 yield, uint256 fee) {
        ...
        require(amount <= principal, "Vault: overspend");
        variableRatePrincipal.principalTimespan +=
            variableRatePrincipal.principal * // update principal timespan until now
first
            (block.timestamp - variableRatePrincipal.checkpointTimestamp);
        variableRatePrincipal.checkpointTimestamp = block.timestamp;

        uint256 amountTimespan;
        (amountTimespan, yield, fee) = _getVariableRatePortion(msg.sender, amount);
        ...
        variableRatePrincipal.principalTimespan -= amountTimespan;
        variableRatePrincipal.principal -= amount;
        ...
        IYieldManager(yieldManager).mintYield(msg.sender, yield);
        IERC20(TOKEN).safeTransfer(msg.sender, fee);
        emit BurnVariableRate(msg.sender, amount, yield, fee);
    }

    function _getVariableRatePortion(address owner, uint256 amount) internal view
returns (uint256 amountTimespan, uint256 yield, uint256 fee) {
        amountTimespan = amount * (block.timestamp -
variableRateTerms[owner].mintTimestamp);
        uint256 idleYield = IYieldManager(yieldManager).getIdleYield();

        uint256 variableRateTotalPrincipalTimespan =
variableRatePrincipal.principalTimespan;

        if (variableRateTotalPrincipalTimespan > 0) {
            yield = (idleYield * amountTimespan) / variableRateTotalPrincipalTimespan;
            fee = (IERC20(TOKEN).balanceOf(address(this)) * amountTimespan) /
variableRateTotalPrincipalTimespan;
        }
    }
```

Solution

For the yield calculation part of the pool a certain depth of funds for both strategies is needed initially to avoid this issue.

Status

Acknowledged; After communicating with the project team, they expressed that only variable rate deposits are allowed in the first four days after the mainnet is deployed, and we will also deposit variable rate liquidity in advance to attract fixed rate. At the beginning, the variable rate will sell the yield that has been obtained. The purpose is to capture the yield difference (yield difference) given up by those who come in after the fixed rate, which is reflected in the position fee and curve.

[N4] [Low] Missing the burn logic in the FixedRateNFT

Category: Design Logic Audit

Content

In the FixedRateNFT contract, users can call the mint function to deposit to the Vault. It can get an ERC721 NFT and traded on the secondary market. However, the contract has no NFT burn operation in the burn and withdraw functions. That means if the user to withdraw all his principal and yields through the burn and withdraw function, the user can still trade the NFT. And the buyer cannot get the principal and yield from the NFT anymore.

Code location:

periphery/FixedRateNFT.sol#66-86

```
    function burn(
        uint256 tokenId,
        uint256 amount
    ) external nonReentrant onlyOwner(tokenId) returns (uint256 yieldToUnlock,
uint256 yieldToRelease) {
        uint256 balanceBefore = IERC20(TOKEN).balanceOf(address(this));
        (yieldToUnlock, yieldToRelease) =
FixedRate(fixedRateContracts[tokenId]).burn(amount);
        require(
            IERC20(TOKEN).balanceOf(address(this)) == balanceBefore + amount +
yieldToRelease,
            "FixedRateNFT: insufficient fund"
        );
        IERC20(TOKEN).safeTransfer(msg.sender, amount + yieldToRelease);
        emit Burn(msg.sender, tokenId, amount, yieldToUnlock, yieldToRelease);
```

```
    }

    function withdraw(uint256 tokenId, uint256 amount) external nonReentrant
onlyOwner(tokenId) {
        uint256 balanceBefore = IERC20(TOKEN).balanceOf(address(this));
        FixedRate(fixedRateContracts[tokenId]).withdraw(amount);
        require(IERC20(TOKEN).balanceOf(address(this)) == balanceBefore + amount,
"FixedRateNFT: insufficient fund");
        IERC20(TOKEN).safeTransfer(msg.sender, amount);
        emit Withdraw(msg.sender, tokenId, amount);
    }
```

Solution

It's recommended to add the NFT burn function in the burn and withdraw function, and the partial withdrawal check will also need to be considered.

Status

Acknowledged; After communicating with the project team, they expressed that the NFT buyer needs to check carefully the intrinsic value in this NFT. Their solution is very similar to UNISWAP V3 LP NFT. It is tradable, but buyer's duty to make sure he/she is buying the right one with right value.

[N5] [Low] Receive can lock users' native tokens

Category: Others

Content

There is a receive function in the WrapMint contract so that the contracts can receive native tokens from the WETH contract. However, the receive function can lock users' native tokens when users transfer the native token in these contracts by mistake.

Code location:

periphery/WrapMint.sol#318

```
    receive() external payable {}
```

Solution

It's recommended to add addresses that can send the native tokens in the contract in the require check logic in the receive() function.

Status

Acknowledged; After communicating with the project team, they expressed that they prefer to prevent this from front end. They are not encouraging users to interact with this smart contracts directly.

[N6] [Suggestion] Withdraw can affect the rate and yield

Category: Design Logic Audit

Content

Throughout the project, the principal is recorded and calculated in 3 parts: VariableRate and FixedRate contracts record the principal deposited by a single user, which is input and output by the Vault contract, and calculate the transformation of the deposits and withdrawals. The Vault contract records and calculates the accumulated deposits and withdrawals of all users in the YieldManager contract. When the Vault contract makes mint and burn rates calls, it updates and recalculates the principal and yield totals stored in the contract. The owner of the VariableRate and FixedRate contracts, i.e., the individual user who made the deposit, can then withdraw the principal directly through the withdraw function in the Emergency situation, and the individual's principal data will not be transferred to the YieldManager contract for updating and calculation through the vault contract. contract for update calculation. The total amount of principal in the program changes after withdrawal, but the total amount of principal recorded in the YieldManager contract remains unchanged, and the calculation of earnings is still based on the total amount of principal withdrawn. Then, in the subsequent accesses by the user, the calculation of the rates and the calculation of the earnings are based on the amount before the withdrawal. Thus, even after the user withdraws the principal, the calculation of the return is still based on the previous total, and the principal in the contract is still the same as before. This is inconsistent with the expected total return and the calculation of rates.

Code location:

FixedRate.sol#62-66

VariableRate.sol#47-51

periphery/FixedRateNFT.sol#80-86

```
    function withdraw(uint256 amount) external nonReentrant onlyOwner onlyEmergency {
        require(amount <= principal, "FixedRate: overspend");
        principal -= amount;
        IERC20(TOKEN).safeTransfer(OWNER, amount);
    }
```

```
    function withdraw(uint256 amount) external nonReentrant onlyOwner onlyEmergency {
        require(amount <= principal, "VariableRate: overspend");
        principal -= amount;
        IERC20(TOKEN).safeTransfer(OWNER, amount);
    }

    function withdraw(uint256 tokenId, uint256 amount) external nonReentrant
onlyOwner(tokenId) {
        uint256 balanceBefore = IERC20(TOKEN).balanceOf(address(this));
        FixedRate(fixedRateContracts[tokenId]).withdraw(amount);
        require(IERC20(TOKEN).balanceOf(address(this)) == balanceBefore + amount,
"FixedRateNFT: insufficient fund");
        IERC20(TOKEN).safeTransfer(msg.sender, amount);
        emit Withdraw(msg.sender, tokenId, amount);
    }
```

Solution

It is recommended that the design of the operation and the calculation of totals and benefits be clarified and that the real-time status be updated promptly.

Status

Acknowledged; After communicating with the project team, they expressed that the emergency steps of operation should be (1) turn off YieldManager's minter/burner access, then (2) turn on the emergency flag. By design, there are minter and burner roles for the YieldManager that are the YieldStrippedToken contract. Once the emergency happens, the minter and burner roles will be revoked, and the call of the mintYield and burnUnmaturedYield will revert by the mint and burnFrom functions.

[N7] [Suggestion] Missing the approve when calling the burnFrom function

Category: Design Logic Audit

Content

When calling the burnFixedRate function to withdraw the principal and yield of the users. At last, it will call the burnUnmaturedYield function of the yieldManager contract to burn the `yieldToBurn` yield tokens. It uses the burnFrom function, but there is no approval from the FixedRate contract to the YieldManager contract.

Code location:

Vault.sol#167

YieldManager.sol#48-51

```
    function burnFixedRate(
        uint256 id,
        uint256 amount
    ) external override nonReentrant returns (uint256 yieldToUnlock, uint256
yieldToRelease) {
        ...
        if (yieldToUnlock > yieldToRelease) {
            ...
            IYieldManager(yieldManager).burnUnmaturedYield(msg.sender, yieldToBurn);
//SlowMist//
        }
        ...
    }


    function burnUnmaturedYield(address fixedRate, uint256 amount) external override
nonReentrant onlyVault {
        IYieldStrippedToken(TOKEN).burnFrom(fixedRate, amount);
        controlledIdleYield += amount;
    }
```

Solution

It's recommended to use the burn function or add the approval logic.

Status

Acknowledged; After communicating with the project team, they expressed that the YieldManager will be granted as a 'burner' of IYieldStrippedToken (IFewWrappedToken). The 'burnFrom' is a 'onlyBurner' function, no approve is required.

[N8] [Suggestion] Redundant code

Category: Others

Content

1.In the Vault, deposit users can call the burnFixedRate function to get their principal and yield by their FixdRate contract. In the burnFixedRate function, the input data amount is already checked in the upper-level contract to check if it is less than the user's principal. Therefore, the input `amount` is always less than or equal to the user's principal. and `yieldToUnlock` is calculated from `(fixedRateTerm.yield * amount)/fixedRateTerm.principal`. This means that `yieldToUnlock` will always be less than or equal to `fixedRateTerm.yield`. and the value of `FixedRate(msg.sender).yield()` is initially based on

`FixedRateTerm.yield`, so this means that `yieldToBurn` will be constant less than the value of `yieldToUnlock`. this makes `require(yieldToBurn <= FixedRate(msg.sender).yield(), "Vault: overburn yield");` judgment redundant.

Code location:

Vault.sol#166

```
    function burnFixedRate(
        uint256 id,
        uint256 amount
    ) external override nonReentrant returns (uint256 yieldToUnlock, uint256
yieldToRelease) {
        require(amount > 0, "Vault: invalid value");
        IYieldManager(yieldManager).withdrawPrincipal(amount);
        FixedRateTerm memory fixedRateTerm = fixedRateTerms[id];
        require(fixedRateTerm.owner == msg.sender, "Vault: unauthorized");
        require(amount <= fixedRateTerm.principal, "Vault: overspend");
        ...
        yieldToUnlock = (fixedRateTerm.yield * amount) / fixedRateTerm.principal;
        yieldToRelease = (yieldToUnlock * (MATURITY - timeLeft)) / MATURITY;
        fixedRateLockedYield -= yieldToUnlock;
        if (fixedRateTerm.principal == amount) {
            delete fixedRateTerms[id];
        } else {
            fixedRateTerms[id].principal = fixedRateTerm.principal - amount;
            fixedRateTerms[id].yield = fixedRateTerm.yield - yieldToUnlock;
        }
        if (yieldToUnlock > yieldToRelease) {
            uint256 yieldToBurn = yieldToUnlock - yieldToRelease;
            require(yieldToBurn <= FixedRate(msg.sender).yield(), "Vault: overburn
yield");
            ...
        }
        ...
    }
```

2.When the user selects a variable interest rate for the deposit, the mintVariableRate function is called to deposit the principal and create the VariableRate contract for the user to record the principal and yield deposited, and then the mint function that creates the VariableRate contract is called to record it. Then, in the mint function, it will first judge whether the balance of IERC20(TOKEN) in the VariableRate contract is greater than or equal to principal + amount, however, the principal in the judgment is 0 at this time because of the initial creation of the contract, and each time

the mint function is called, it is only through the mintVariableRate function that the principal and yield of the contract are recorded. function to call, each time the user enters will create a new VariableRate contract. This makes the principal in the judgment a redundant parameter.

Code location:

VariableRate.sol#31-34

```
    function mint(uint256 amount) external nonReentrant onlyVault {
        require(IERC20(TOKEN).balanceOf(address(this)) >= principal + amount,
"VariableRate: insufficient fund");
        principal += amount;
    }
```

3.When the user calls mintFixedRate to make a Fixed Rate deposit, it will first call getIdleYield for the second level of the fieldManager function's depositPrincipal call and getYieldToLock call to get and update the total controlledIdleYield in the fieldManager contract. To get lockedYield, you assign the field value of FixedRate to the user and determine whether the reward to be locked is greater than getIdleYield, and then when you re-enter the YieldManager contract, you call the mintYield function is called to mintYield for the user. Then in the mintYield function, it will again check if the incoming yield, which is the lockedYield parameter of the upper level, is less than or equal to getIdleYield. since the upper level has already judged it once, this makes the judgment in the YieldManager contract redundant.

Code location:

YieldManager.sol#42

```
    function mintFixedRate(
        uint256 amount,
        uint256 minLockedYield,
        address recipient
    ) external override nonReentrant returns (address owner, uint256 lockedYield,
uint256 maturityTimestamp) {
        require(amount > 0, "Vault: invalid value");
        IYieldManager(yieldManager).depositPrincipal(amount);
        ...
        lockedYield = getYieldToLock(amount);
        ...
        require(lockedYield <= IYieldManager(yieldManager).getIdleYield(), "Vault:
overspend");
        ...
```

```
        uint256 id = _nextId++;
        fixedRateTerms[id] = FixedRateTerm({
            owner: owner,
            principal: amount,
            yield: lockedYield,
            maturityTimestamp: maturityTimestamp = block.timestamp + MATURITY
        });

        IYieldManager(yieldManager).mintYield(owner, lockedYield);
        ...
    }


    function mintYield(address recipient, uint256 yield) external override
nonReentrant onlyVault {
        require(yield <= getIdleYield(), "Vault: overmint yield");
        IYieldStrippedToken(TOKEN).mint(recipient, yield);
        controlledIdleYield = controlledIdleYield > yield ? controlledIdleYield -
yield : 0;
    }
```

Solution

It is recommended to clarify the business logic implementation, and if it is redundant code, it is recommended to remove it from the contract.

Status

Acknowledged; After communicating with the project team, they expressed that the redundancy here is by design. The reason is that it clearly shows how much is allowed from the viewpoint of `FixedRate` (from user's viewpoint) and `YieldManger` (from protocol's viewpoint).

[N9] [Suggestion] Missing the 0 address/value check

Category: Others

Content

All the addresses and parameters are missing the 0 address and 0 value check in the constructor, and these parameters can not be modified after the parameters' initialization.

Code location:

YieldManager.sol#32-34

Vault.sol#44-55

periphery/WrapMint.sol#39-42

periphery/FixedRateNFT.sol#29-32

Solution

It is recommended to add the 0 address or 0 value check.

Status

Acknowledged

[N10] [Suggestion] Missing the event records

Category: Others

Content

Guardian, governor, and owner roles can arbitrarily modify `EMERGENCY`, `vault`, `yieldRate`, `fixedRateNft`, and `whitelistedExchanges` parameters, but there are no event logs in these functions.

Code location:

core/CoreRef.sol#45-52

YieldManager.sol#95-106

periphery/WrapMint.sol#298-308

```
function startEmergency() external override onlyGuardianOrGovernor {
    EMERGENCY = true;
}

function stopEmergency() external override onlyGuardianOrGovernor {
    EMERGENCY = false;
}

function updateVault(address newVault) external onlyGuardianOrGovernor {
    vault = newVault;
}

function updateRate(uint256 rate) external onlyGuardianOrGovernor {
    _updateIdleYield();
    yieldRate = rate;
}

function addExchange(address exchange) external onlyOwner {
    whitelistedExchanges[exchange] = true;
}
```

```
function removeExchange(address exchange) external onlyOwner {
    whitelistedExchanges[exchange] = false;
}

function setFixedRateNft(address nft) external onlyOwner {
    fixedRateNft = nft;
}
```

Solution

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

Status

Fixed

[N11] [Suggestion] Preemptive Initialization

Category: Race Conditions Vulnerability

Content

By calling the initialize and deploy functions to initialize the contracts, there is a potential issue that malicious attackers preemptively call the initialize function to initialize.

Code location:

core/Core.sol#10-12

```
function init() external override initializer {
    _setupGovernor(msg.sender);
}
```

Solution

It is suggested that the initialization operation can be called in the same transaction immediately after the contract is created to avoid being maliciously called by the attacker.

Status

Acknowledged

[N12] [Suggestion] External call reminder

Category: Unsafe External Call Audit

Content

In the WrapMint contract, there is an external call to the low-level call of the token swap, and the calldata for the call in it is constructed and passed in by an arbitrary user without any checking. Although the target contract of the call, exchange, requires the owner to be the owner of the whitelisted exchanges to be added and removed, there is still a need to pay attention to the security of the external call and the external contract.

Code location:

periphery/WrapMint.sol#71

```
    function _swap(
        address exchange,
        address token,
        uint256 amountIn,
        uint256 amountOutMin,
        bytes calldata data
    ) internal returns (uint256 amountOut) {
        require(whitelistedExchanges[exchange], "WrapMint: unauthorized exchange");
        ...
        (bool success, ) = exchange.call(data);
        require(success, "WrapMint: swap failed");
        ...
    }
```

Solution

It is recommended to ensure the security and legitimacy of the external contract exchange that is called.

Status

Acknowledged

[N13] [Suggestion] Successive mint and burn calls to get the yield and fee

Category: Design Logic Audit

Content

In the VariableRate strategy, users can add or remove principal and rewards at any time via mintVariableRate and burnVariableRate. The VariableRate earnings calculation is to take the user's principal * deposit time than on the VariableRate strategy accumulated principalTimespan integrated ratio multiplied by the total getIdleYield and fee.

That means as long as the user's funds in and out of the rewards can be obtained, there is no charge for any fees.

Code location:

Vault.sol#215-263, 275-293

```
    function burnVariableRate(
        uint256 amount,
        uint256 minYield
    ) external override nonReentrant returns (uint256 yield, uint256 fee) {
        ...
        require(amount <= principal, "Vault: overspend");
        variableRatePrincipal.principalTimespan +=
            variableRatePrincipal.principal * // update principal timespan until now
first
            (block.timestamp - variableRatePrincipal.checkpointTimestamp);
        variableRatePrincipal.checkpointTimestamp = block.timestamp;

        uint256 amountTimespan;
        (amountTimespan, yield, fee) = _getVariableRatePortion(msg.sender, amount);
        ...
        variableRatePrincipal.principalTimespan -= amountTimespan;
        variableRatePrincipal.principal -= amount;
        ...
        IYieldManager(yieldManager).mintYield(msg.sender, yield);
        IERC20(TOKEN).safeTransfer(msg.sender, fee);
        emit BurnVariableRate(msg.sender, amount, yield, fee);
    }

    function _getVariableRatePortion(address owner, uint256 amount) internal view
returns (uint256 amountTimespan, uint256 yield, uint256 fee) {
        amountTimespan = amount * (block.timestamp -
variableRateTerms[owner].mintTimestamp);
        uint256 idleYield = IYieldManager(yieldManager).getIdleYield();

        uint256 variableRateTotalPrincipalTimespan =
variableRatePrincipal.principalTimespan;

        if (variableRateTotalPrincipalTimespan > 0) {
            yield = (idleYield * amountTimespan) / variableRateTotalPrincipalTimespan;
            fee = (IERC20(TOKEN).balanceOf(address(this)) * amountTimespan) /
variableRateTotalPrincipalTimespan;
        }
    }
```

Solution

It is recommended to confirm whether the above strategy is the expected design.

Status

Acknowledged; After communicating with the project team, they expressed that this strategy is expected.

5 Audit Result

Audit Number	Audit Team	Audit Date	Audit Result
0X002404070001	SlowMist Security Team	2024.04.01 - 2024.04.07	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, 4 low risks, and 8 suggestions. 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.

Official Website

www.slowmist.com

E-mail

team@slowmist.com

Twitter

@SlowMist_Team

Github

https://github.com/slowmist