Security Audit Report for Resonate

Date: Aug 18, 2022

Version: 1.0

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	Revest Finance
Target	Resonate

Version History

Version	Date	Description
1.0	Aug 18, 2022	First Release

About BlockSec The BlockSec focuses on the security of the blockchain ecosystem and collaborates with leading DeFi projects to secure their products. BlockSec is founded by top-notch security researchers and experienced experts from both academia and industry. They have published multiple blockchain security papers in prestigious conferences, reported several zero-day attacks of DeFi applications, and successfully protected digital assets that are worth more than 5 million dollars by blocking multiple attacks. They can be reached at Email, Twitter and Medium.

Chapter 1 Introduction

1.1 About Target Contracts

Information	Description
Type	Smart Contract
Language	Solidity
Approach	Semi-automatic and manual verification

The target of this audit is Resonate [1], a project that aims to provide a financial tool with the concept of time-value-of-money. The users of this project can be classified into two categories, i.e., *consumer*s and *provider*s. Specifically, the consumers hold capital for staking into the underlying protocols (e.g., Yearn and AAVE) and they would like to receive cash rather than the future interests. While as the counterparty, the providers would rather pay cash for more profitable future interests. Resonate can match the consumers and the providers to serve their demands on both sides.

The auditing process is iterative. Specifically, we would audit the commits that fix the discovered issues. If there are new issues, we will continue this process. The commit SHA values during the audit are shown in the following table. Our audit report is responsible for the code in the initial version (`Version 1`), as well as new code (in the following versions) to fix issues in the audit report.

Project		Commit SHA
Resonate	`Version 1`	`9177c788cb2f3304b16f1583696794f24e1a0a92`
	`Version 2`	`f08d7dda78de0f0835c55d81b33f36ccca381c01`

Note that, this audit does **NOT** cover all modules in the repository. Specifically, the smart contracts under the **hardhat/contracts/oracle** folder (introduced by `Version 2`) are excluded.

1.2 Disclaimer

This audit report does not constitute investment advice or a personal recommendation. It does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Any entity should not rely on this report in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset.

This audit report is not an endorsement of any particular project or team, and the report does not guarantee the security of any particular project. This audit does not give any warranties on discovering all security issues of the smart contracts, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with independent audits and a public bug bounty program to ensure the security of smart contracts.

The scope of this audit is limited to the code mentioned in Section 1.1. Unless explicitly specified, the security of the language itself (e.g., the solidity language), the underlying compiling toolchain and the

[1] https://github.com/Revest-Finance/Resonate

computing infrastructure are out of the scope.

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

- **Vulnerability Detection** We first scan smart contracts with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.
- **Semantic Analysis** We study the business logic of smart contracts and conduct further investigation on the possible vulnerabilities using an automatic fuzzing tool (developed by our research team). We also manually analyze possible attack scenarios with independent auditors to cross-check the result.
- **Recommendation** We provide some useful advice to developers from the perspective of good programming practice, including gas optimization, code style, and etc.

We show the main concrete checkpoints in the following.

1.3.1 Software Security

* Reentrancy
* DoS
* Access control
* Data handling and data flow
* Exception handling
* Untrusted external call and control flow
* Initialization consistency
* Events operation
* Error-prone randomness
* Improper use of the proxy system

1.3.2 DeFi Security

* Semantic consistency
* Functionality consistency
* Permission management
* Business logic
* Token operation
* Emergency mechanism
* Oracle security
* Whitelist and blacklist
* Economic impact
* Batch transfer

1.3.3 NFT Security

* Duplicated item

- Verification of the token receiver
- Off-chain metadata security

1.3.4 Additional Recommendation

- Gas optimization
- Code quality and style

Note *The previous checkpoints are the main ones. We may use more checkpoints during the auditing process according to the functionality of the project.*

1.4 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [2] and Common Weakness Enumeration [3]. The overall *severity* of the risk is determined by *likelihood* and *impact*. Specifically, likelihood is used to estimate how likely a particular vulnerability can be uncovered and exploited by an attacker, while impact is used to measure the consequences of a successful exploit.

In this report, both likelihood and impact are categorized into two ratings, i.e., *high* and *low* respectively, and their combinations are shown in Table 1.1.

Table 1.1: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Low*
High	High	Medium
Low	Medium	Low

Accordingly, the severity measured in this report are classified into three categories: **High**, **Medium**, **Low**. For the sake of completeness, **Undetermined** is also used to cover circumstances when the risk cannot be well determined.

Furthermore, the status of a discovered item will fall into one of the following four categories:

- **Undetermined** No response yet.
- **Acknowledged** The item has been received by the client, but not confirmed yet.
- **Confirmed** The item has been recognized by the client, but not fixed yet.
- **Fixed** The item has been confirmed and fixed by the client.

[2] https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

[3] https://cwe.mitre.org/

Chapter 2 Findings

In total, we find **ten** potential issues. We have **six** recommendations and **three** notes.

- High Risk: 4
- Medium Risk: 3
- Low Risk: 3
- Recommendation: 6
- Note: 3

ID	Severity	Description	Category	Status
1	Low	Inconsistent rounding check	Software Security	Acknowledged
2	Low	Immutable variable derives from mutable state	Software Security	Acknowledged
3	Medium	Precision losses	Software Security	Fixed
4	Medium	Incorrect parameters for amount conversion	Software Security	Fixed
5	Low	Unhandled corner case	Software Security	Fixed
6	High	Mixed usages of pool asset and vault asset	DeFi Security	Fixed
7	Medium	Infinite claims of interest	DeFi Security	Fixed
8	High	Arbitrary transfer via `proxyCall`	DeFi Security	Fixed
9	High	Price manipulation attack	DeFi Security	Fixed
10	High	Potential DoS attack	NFT Security	Fixed
11	-	Check parameters in constructors and governance functions	Recommendation	Fixed
12	-	Move state variable changes out of event logs	Recommendation	Fixed
13	-	Remove unused struct fields	Recommendation	Fixed
14	-	Refactor clearing mapping fields into a `delete` statement	Recommendation	Fixed
15	-	Remove duplicate calls in the `OutputReceiverProxy` contract	Recommendation	Fixed
16	-	Check the pool in the `MasterChefAdapter` contract	Recommendation	Acknowledged
17	-	Refunding procedure	Note	
18	-	ID continuity assumption of the interest and principal FNFTs	Note	
19	-	Potential vulnerability in the `harvest` function	Note	

The details are provided in the following sections.

2.1 Software Security

2.1.1 Inconsistent rounding check

Severity Low

Status Acknowledged

Introduced by `Version 1`

Description In the `Resonate` contract, pools are created for different underlying protocols and fee rates. These pools are used to provide a place for the consumers and producers to match their orders. For each pool, user deposits are divided into packets with a fixed packet size specified by a parameter named `packetSize`. However, there exist three different types of rounding checks for calculations related to `packetSize`, as follows:

- A check with less-than in the `submitProducer` function (Line 312).

```
308  if (vaultAsset == pool.asset) {
309    sharesPerPacket = shouldFarm ? 1 : 0;
310    producerPacket = pool.packetSize * pool.rate / PRECISION;
311    // return; //39533
312    require(amount % producerPacket < 5, 'ER006'); //This should fail, but it's not because
           amount is already 0
313  }
```

Listing 2.1: Resonate.sol

- A check with less-than-or-equal-to in the `submitConsumer` function (Line 184). Notice that the error code is also not the same.

```
180  function submitConsumer(bytes32 poolId, uint amount) external nonReentrant {
181      // Common code
182      PoolConfig memory pool = pools[poolId];
183      require(amount > 0, 'ER003');
184      require(amount % pool.packetSize <= 5, 'ER005'); //be within 10 gwei to handle round-
             offs
185      ...
186  }
```

Listing 2.2: Resonate.sol

- Rounding by division without any check in the `submitProducer` function (Line 317).

```
314  shouldFarm = false;
315  sharesPerPacket = IOracleDispatch(oracleDispatch[vaultAsset][pool.asset]).
         getValueOfAsset(vaultAsset, pool.asset, true);
316  producerPacket = _getAmountPaymentAsset(pool.rate * pool.packetSize/PRECISION,
         sharesPerPacket, vaultAsset, vaultAsset);
317  amount = amount / producerPacket * producerPacket; // Is this a rounding operation?,
         would be safer to use modulo subtraction
318  require(amount > 0, "ER003");
```

Listing 2.3: Resonate.sol

Impact Inconsistent rounding check may results in unexpected behaviors.

Suggestion Make the rounding checks consistent.

Feedback from the Project This is not an issue, it is a design decision. Lidos StETH contains a corner-case where some amount of wei of StETH may be rounded off and remain with the user after a transfer based on the packetSize. As a result to ensure the full amount is transferred to the vaults/consumers, allowing a slight confidence-interval on deposit ensures the proper amount is transferred and not rounded-off.

2.1.2 Immutable variable derives from mutable state

Severity Low

Status Acknowledged

Introduced by Version 1

Description In the OutputReceiverProxy contract, the FNFT_HANDLER address is derived from the addressRegistry state variable in the constructor. However, the FNFT_HANDLER is an immutable state variable, while addressRegistry can be modified in setAddressRegistry function.

```
IFNFTHandler private immutable FNFT_HANDLER;

constructor(address _addressRegistry) {
    addressRegistry = _addressRegistry;
    TOKEN_VAULT = IAddressRegistry(_addressRegistry).getTokenVault();
    FNFT_HANDLER = IFNFTHandler(IAddressRegistry(_addressRegistry).getRevestFNFT());
}
```

Listing 2.4: OutputReceiverProxy.sol

```
function setAddressRegistry(address _addressRegistry) external onlyOwner {
  addressRegistry = _addressRegistry;
}
```

Listing 2.5: OutputReceiverProxy.sol

Impact The immutable variable cannot be updated when the variable it derives from changes.

Suggestion N/A

Feedback from the Project This is not an issue, it is a design decision. The entry point may need to change, but the FNFT handler should never be mutable.

2.1.3 Precision losses

Severity Medium

Status Fixed in Version 2

Introduced by Version 1

Description There are two precision loss problems in the project.

The first problem is in the _activateCapital function of Resonate contract. Specifically, the producers and consumers submit their orders to the Resonate contract and the contract matches the orders with their counterparties. If there is no counterparty, the orders will be pushed into the corresponding queues and the assets are deposited into the underlying adapter vaults. The shares minted are kept in the pool wallet on behalf of the users.

Whenever a new consumer order (or producer order) is matched with a counterparty order in the queue, the _activateCapital function is called to invoke the corresponding function of the pool wallet to get the number of total shares deposited into the underlying adapter. These shares would be divided by the number of packets (Line 713 and Line 738). This procedure incurs a precision loss. Due to the precision loss of the integer division, there would be some residual shares left in the pool wallet, which could bring financial loss to the user.

```
672 function _activateCapital(
673   ParamPacker memory packer
```

```
) private returns (uint principalId) {
    // Double check in the future on the vaultAdapters
    IERC4626 vault = IERC4626(packer.adapter);
    address vaultAsset = vault.asset(); // The native asset
    // Fetch curPrice if necessary
    // State where it would be zero is when producer order is being submitted for non-farming position
    // Needs to come before FNFT creation, since curPrice is saved within that storage

    // Need to withdraw from the vault for this operation if value was previously stored in it
    // Utilize this opportunity to charge fee on interest that has accumulated during dwell time
    uint amountFromConsumer = packer.quantityPackets * packer.pool.packetSize;
    uint amountToConsumer = packer.isCrossAsset ? (
        _getAmountPaymentAsset(
            amountFromConsumer * packer.pool.rate / PRECISION,
            packer.currentExchangeRate,
            packer.pool.asset,
            vaultAsset)
        ) : amountFromConsumer * packer.pool.rate / PRECISION; //upfront?

    if(packer.isProducerNew) {
        {
            address consumerOwner = packer.consumerOrder.owner.toAddress();
            // The producer position is the new one, take value from them and transfer to consumer
            IERC20(packer.pool.asset).safeTransferFrom(msg.sender, consumerOwner, amountToConsumer);

            // Prepare the desired FNFTs
            principalId = _createFNFTs(packer.quantityPackets, packer.poolId, consumerOwner, packer.producerOrder.owner.toAddress());
        }
        {
            // Claim interest on the farming of the consumer's capital
            (uint shares, uint interest) = IPoolWallet(_getAddressForPool(packer.poolId)).activateExistingConsumerPosition(
                amountFromConsumer,
                packer.quantityPackets * packer.consumerOrder.depositedShares,
                _getAddressForFNFT(packer.poolId),
                DEV_ADDRESS,
                packer.pool.vault,
                packer.adapter
            );

            shares /= packer.quantityPackets;

            Active storage active = activated[principalId];
            active.sharesPerPacket = shares;
            if(packer.pool.addInterestRate != 0) {
                active.startingSharesPerPacket = shares;
            }

            emit FeeCollection(packer.poolId, interest);
        }
```

```


    } else {
        // The consumer position is the new one, take stored producer value and transfer to them
        // If the producer was farming, we can detect this and charge our fee on interest

        address producerOwner = packer.producerOrder.owner.toAddress();

        // Need to deposit to vault from consumer and store in FNFT
        IERC20(vaultAsset).safeTransferFrom(msg.sender, address(this), amountFromConsumer);

        // Prepare the desired FNFTs
        principalId = _createFNFTs(packer.quantityPackets, packer.poolId, packer.consumerOrder.
            owner.toAddress(), producerOwner);
        {
            Active storage active = activated[principalId];
            uint shares = vault.deposit(amountFromConsumer, _getAddressForFNFT(packer.poolId)) /
                packer.quantityPackets;
            active.sharesPerPacket = shares;
            if(packer.pool.addInterestRate != 0) {
                active.startingSharesPerPacket = shares;
            }

        }


        // Need to then pay out to consumer from producer position
        if(packer.producerOrder.depositedShares > 0 && !packer.isCrossAsset) {
            uint interest = IPoolWallet(_getAddressForPool(packer.poolId)).
                activateExistingProducerPosition(
                amountToConsumer,
                packer.quantityPackets * packer.producerOrder.depositedShares,
                msg.sender,
                DEV_ADDRESS,
                packer.pool.vault,
                packer.adapter
            );
            emit FeeCollection(packer.poolId, interest);

        } else {
            IPoolWallet(_getAddressForPool(packer.poolId)).withdraw(amountToConsumer, packer.pool.
                asset, msg.sender);
        }
    }
    emit CapitalActivated(packer.poolId, packer.quantityPackets, principalId, packer.
        sharesPerPacket);
}
```

Listing 2.6: Resonate.sol

The second problem is in the `_enqueue` function of the `Resonate` contract. Specifically, when the submitted order is to be put into the queue, the `_enqueue` function is invoked. If the asset left in this order should be deposited into the underlying adapter vault, the shares per packet will be calculated by first invoking the `deposit` function of the vault adapter to get the total shares, and then dividing it by the number of packets. This process also incurs precision loss and further potential financial loss.

```
function _enqueue(
  bytes32 poolId,
  bool isProvider,
  bool shouldFarm,
  Order memory order,
  uint amount,
  address asset,
  address vaultAdapter
) private {

  if(shouldFarm) {
      // Store in pool smart wallet as vault deposit
      IERC20(asset).safeTransferFrom(msg.sender, address(this), amount);

      // Decision to deposit costs 62,080 gas
      order.depositedShares = IERC4626(vaultAdapter).deposit(amount, _getAddressForPool(poolId))
          / order.packetsRemaining;
  } else {
      // Leaving rateAtDeposit as zero signifies non-farming nature of order
      // Similarly stores value in pool smart wallet


      IERC20(asset).safeTransferFrom(msg.sender, _getAddressForPool(poolId), amount);
  }

  PoolQueue storage qm = queueMarkers[poolId]; //cold sload
  // Allow overflow to reuse indices
  unchecked {
      if(isProvider) {
          providerQueue[poolId][qm.providerTail] = order; //cold? sstore
          emit EnqueueProvider(poolId, msg.sender, qm.providerTail++, shouldFarm, order);
      } else {
          consumerQueue[poolId][qm.consumerTail] = order;
          emit EnqueueConsumer(poolId, msg.sender, qm.consumerTail++, order);
      }
  }
  }
```

Listing 2.7: Resonate.sol

Impact Precision losses in the share calculating process would leave share dust in the project, which may result in financial losses for the users.

Suggestion Refactor the calculation logic to prevent precision losses.

2.1.4 Incorrect parameters for amount conversion

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `Resonate` contract, the producers offer payment tokens of the pool (i.e., the `pool.asset` token) to purchase future interests and the consumers offer vault tokens (i.e., the `vaultAsset` token) that are deposited into the underlying protocols. These two token types can be different and there is a `_getAmountPaymentAsset` function for converting the amount of one token to that of another one based on current price. The last two parameters of this function suggests that those two addresses should be `pool.asset` and `vaultAsset`. However, at line 316 of the `submitProducer` function (Listing 2.3), the addresses passed in are both `vaultAsset`.

```
function _getAmountPaymentAsset(uint amountNativeAsset, uint currentExchangeRate, address
    poolAsset, address vaultAsset) private view returns (uint amount) {
  //Amount of payout Token to consumer immediately
  uint divisor;
  uint8 poolDecimals;
  uint8 vaultDecimals;
  try IERC20Detailed(poolAsset).decimals() returns (uint8 dec) {
      poolDecimals = dec;
  } catch {
      poolDecimals = 18;
  }
  try IERC20Detailed(vaultAsset).decimals() returns (uint8 dec) {
      vaultDecimals = dec;
  } catch {
      vaultDecimals = 18;
  }
  if(poolDecimals == vaultDecimals) {
      // 1E36 or 1E12
      // REALLY unusual edge-case handling
      divisor = poolDecimals;
  } else {
      // 1E24
      //          1E18          1E6
      divisor = vaultDecimals > poolDecimals ? vaultDecimals : Math.min(poolDecimals -
          vaultDecimals, vaultDecimals);
  }
    //1E6          1E24
  amount = amountNativeAsset * currentExchangeRate / (10 ** divisor);
}
```

Listing 2.8: Resonate.sol

Impact Incorrect parameters may lead to incorrect calculation result, which might cause financial losses.

Suggestion Check the usage of the function parameters.

2.1.5 Unhandled corner case

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The `valueRewardTokens` function in the `MasterChefAdapter` contract is used to estimate the current value of the reward tokens in the adapter. It is done by simulating a swap from the reward token to tokens in the pair, and a liquidity provision using the swapped tokens. To simulate the swap, the function decides the swap path (`tokenRoute`) according to whether the `rewardToken` is `lpToken0` or `lpToken1` (`token0` or `token1` in the underlying pair). However, there's a corner case which is not properly handled. Specifically, if the `rewardToken` is neither `lpToken0` nor `lpToken1`, the `tokenRoute` will be set to the default one, which is incorrect and can cause miscalculation of the value of the reward tokens.

```
function valueRewardTokens() public view virtual returns (uint256 lpTokens) {
  if (IERC20(rewardToken).balanceOf(address(this)) > 1) {
      uint256 rewardTokenHalf = IERC20(rewardToken).balanceOf(address(this)).div(2);
      // ("Balance reward token: %s", IERC20(rewardToken).balanceOf(address(this)));

      // ("reward token half: %s", rewardTokenHalf);

      (uint reserveA, uint reserveB,) = IUniswapV2Pair(lpPair).getReserves();

      uint256 reserveTokens = reserveA;
      address[] memory tokenRoute = rewardTokenToLp0Route;

      if (lpToken0 == rewardToken) {
          reserveTokens = reserveB;
          tokenRoute = rewardTokenToLp1Route;
      }

      uint256 amountTokenOut = IUniswapV2Router02(uniRouter).getAmountsOut(rewardTokenHalf,
          tokenRoute)[tokenRoute.length.sub(1)];

      uint256 totalSupply = asset.totalSupply();
      uint256 _kLast = IUniswapV2Pair(lpPair).kLast();
      uint256 newSupply;

      if (_kLast != 0) {
          uint rootK = FixedPointMathLib.sqrt(uint(reserveA).mul(reserveB));
          uint rootKLast = FixedPointMathLib.sqrt(_kLast);

          if (rootK > rootKLast) {
              uint numerator = totalSupply.mul(rootK.sub(rootKLast));
              uint denominator = rootK.mul(5).add(rootKLast);
              uint liquidity = numerator / denominator;
              if (liquidity > 0) newSupply = totalSupply.add(liquidity);
          }
      }
      lpTokens = amountTokenOut.mulDivDown(newSupply, reserveTokens);
  }

  else return lpTokens = 0;
}
```

Listing 2.9: MasterChefAdapter.sol

Impact Unhandled corner case may lead to unexpected behaviours.

Suggestion Make sure the `rewardToken` is either `lpToken0` or `lpToken1`.

2.2 DeFi Security

2.2.1 Mixed usages of pool asset and vault asset

Severity High

Status Fixed in Version 2

Introduced by Version 1

Description There are two kinds of assets in the Resonate contract, i.e., pool asset and vault asset. The pool asset is used by the producers to pay to the consumers for the interests, while the vault asset is used by the consumers to deposit to the underlying adapter vault to make interests for the producers. However, there are two mixed usages of these two assets.

First, in the submitConsumer function, if the consumer order is not fully matched with the producer orders, the _enqueue function (see Listing 2.7) will deposit the vault assets transferred from the consumer to the underlying adapter vault. That means the consumer *deposits vault assets* and *gets shares that can be redeemed to vault assets*.

```
if(!hasCounterparty && consumerOrder.packetsRemaining > 0) {
  // No currently available trade, add this order to consumer queue
  _enqueue(poolId, false, true, consumerOrder, amount, vaultAsset, adapter);
}
```

Listing 2.10: Resonate.sol

However, there is a function for users to modify or cancel their orders. At line 429, for the consumer orders that have been pushed into the queue (with a depositedShares that is larger than 0) when *the pool asset is not the same as the vault asset*, this function can withdraw pool assets to the consumer (line 441). Therefore, *the consumer actually "swapped" the vault assets to the pool assets at a 1:1 ratio*.

```
387 function modifyExistingOrder(bytes32 poolId, uint112 amount, uint64 position, bool isProvider)
        external nonReentrant {
388     // This function can withdraw tokens from an existing queued order and remove that order
            entirely if needed
389     // amount = number of packets for order
390     // if amount == packets remaining then just go and null out the rest of the order
391     // delete sets the owner address to zero which is an edge case handled elsewhere
392 
393     Order memory order = isProvider ? providerQueue[poolId][position] : consumerQueue[poolId][
            position];
394     require(msg.sender == order.owner.toAddress(), "ER007");
395 
396     //State changes
397     if (order.packetsRemaining == amount) {
398         PoolQueue storage qm = queueMarkers[poolId];
399         emit OrderWithdrawal(poolId, amount, true, msg.sender);
400 
401         if (isProvider) {
402             if (position == qm.providerHead) {
403                 qm.providerHead++;
404             }
405             else if (position == qm.providerTail) {
```

```
                qm.providerTail--;
            }
            delete providerQueue[poolId][position];
        } else {
            if (position == qm.consumerHead) {
                qm.consumerHead++;
            } else if (position == qm.consumerTail) {
                qm.consumerTail--;
            }
            delete consumerQueue[poolId][position];
        }
    } else {
        if (isProvider) {
            providerQueue[poolId][position].packetsRemaining -= amount;
        } else {
            consumerQueue[poolId][position].packetsRemaining -= amount;
        }
        emit OrderWithdrawal(poolId, amount, false, msg.sender);
    }

    PoolConfig memory pool = pools[poolId];
    uint amountTokens = isProvider ? amount * pool.packetSize * pool.rate / PRECISION : amount
         * pool.packetSize;
    //Token Transfers
    if (order.depositedShares > 0 && IERC4626(vaultAdapters[pool.vault]).asset() == pool.asset)
         { // > 0 signifies it was farming
        address asset = IERC4626(vaultAdapters[pool.vault]).asset();
        uint tokensReceived = _getWalletForPool(poolId).withdrawFromVault(order.depositedShares
             * amount, address(this), vaultAdapters[pool.vault]);
        uint fee;
        if(tokensReceived > amountTokens) {
            fee = tokensReceived - amountTokens;
            IERC20(asset).safeTransfer(DEV_ADDRESS, fee);
        }
        IERC20(asset).safeTransfer(msg.sender, tokensReceived - fee);

    } else {
        // Withdraw from non-farming pool
        _getWalletForPool(poolId).withdraw(amountTokens, pool.asset, msg.sender);
    }
}
```

Listing 2.11: Resonate.sol

Second, the `claimInterest` and `batchClaimInterest` functions are designed for the producers to claim the interests they've purchased in cash. Since the consumer assets are deposited to the underlying adapter vault, the interests should also be returned in *vault assets*. However, at line 540 of the `claimInterest` function and line 506 of the `batchClaimInterest` function, the interests are actually transferred to the producer in *pool assets*, which is another mixed usage of the two assets.

```
517 function claimInterest(uint fnftId, address recipient) public override nonReentrant {
518   require(msg.sender == PROXY_OUTPUT_RECEIVER || FNFT_HANDLER.getBalance(msg.sender, fnftId) >
        0, 'ER010');
```

```
    Active memory active = activated[fnftIdToIndex[fnftId]];
    require(fnftId == active.principalId + 1, 'ER009');
    uint prinPackets = FNFT_HANDLER.getSupply(active.principalId);
    require(prinPackets > 0, 'ER016');
    PoolConfig memory pool = pools[active.poolId];
    // Withdraw to this contract
    // NB: Potential violation of checks-effects-interaction. Likely acceptable within context of ERC-20 transfer to this vault
    // NB: This is the kind of question to pose to the auditors
    (uint interest, uint claimPerPacket) = _getWalletForFNFT(active.poolId).calculateAndClaimInterest(pool.vault, vaultAdapters[pool.vault], address(this), prinPackets * pool.packetSize, active.sharesPerPacket * prinPackets);
    claimPerPacket /= prinPackets;
    if(claimPerPacket <= active.sharesPerPacket) {
        activated[fnftIdToIndex[fnftId]].sharesPerPacket -= claimPerPacket;
    } else {
        activated[fnftIdToIndex[fnftId]].sharesPerPacket = 0;
    }


    // Claim fee on interest
    uint fee = interest * FEE / DENOM; // round the feed
    IERC20(pool.asset).transfer(DEV_ADDRESS, fee);
    // Forward to recipient
    IERC20(pool.asset).transfer(recipient, interest-fee);

    emit FeeCollection(active.poolId, fee);
    // TODO: Why are we formatting this event this way?
    emit InterestClaimed(active.poolId, fnftId, recipient, interest);
    }
```

Listing 2.12: Resonate.sol

```
function batchClaimInterest(uint[][] calldata fnftIds, address recipient) external {
    // Outer array is an array of all FNFTs segregated by pool
    // Inner array is array of FNFTs to claim interest on
    uint numberPools = fnftIds.length;
    require(numberPools > 0, 'ER003');

    // for each pool
    for(uint i; i < numberPools; ++i) {
        // save the list of ids for the pool
        uint[] calldata fnftsByPool = fnftIds[i];
        require(fnftsByPool.length > 0, 'ER003');

        // get the first order, we commit one SLOAD here
        bytes32 poolId = activated[fnftIdToIndex[fnftsByPool[0]]].poolId;
        PoolConfig memory pool = pools[poolId];
        IERC4626 vault = IERC4626(vaultAdapters[pool.vault]);
        uint shareNormalization = vault.totalSupply() * PRECISION / vault.totalAssets(); // shares per asset
        // set up global to track total shares
        uint totalSharesToRedeem;
```

```
        // Precision loss from this is negligible
        // for each id, should be for loop
        for(uint j; j < fnftsByPool.length; ++j) {
            {
                Active memory active = activated[fnftIdToIndex[fnftsByPool[j]]];
                require(active.poolId == poolId, 'ER026');
                // save the individual id
                uint fnftId = fnftsByPool[j];
                require(msg.sender == PROXY_OUTPUT_RECEIVER || FNFT_HANDLER.getBalance(msg.sender,
                    fnftId) > 0, 'ER010');
                require(fnftId == active.principalId + 1, 'ER009');
                // 1
                uint prinPackets = FNFT_HANDLER.getSupply(active.principalId);
                require(prinPackets > 0, 'ER016');
                {
                    // 1000e6         = 1000e6      *     1
                    uint amountUnderlying = pool.packetSize * prinPackets;
                    // huh?
                    uint totalSharesUnderlying = shareNormalization * amountUnderlying / PRECISION;
                    // huh?
                    uint sharesRedeemed = active.sharesPerPacket * prinPackets -
                        totalSharesUnderlying;
                    // add to cumulative total
                    totalSharesToRedeem += sharesRedeemed;
                    // huh? presumably this is to save off the value
                    sharesRedeemed /= prinPackets;
                    if(sharesRedeemed <= active.sharesPerPacket) {
                        activated[fnftIdToIndex[fnftId]].sharesPerPacket -= sharesRedeemed;
                    } else {
                        activated[fnftIdToIndex[fnftId]].sharesPerPacket = 0;
                    }
                }
            }
        }
        uint interest = _getWalletForFNFT(poolId).redeemShares(pool.vault, vaultAdapters[pool.vault
            ], address(this), totalSharesToRedeem);
        uint fee = interest * FEE / DENOM;
        IERC20(pool.asset).transfer(DEV_ADDRESS, fee);
        // Forward to recipient
        IERC20(pool.asset).transfer(recipient, interest-fee);
        emit FeeCollection(poolId, fee);
    }
    }
```

Listing 2.13: Resonate.sol

Impact Mixed usages of pool asset and vault asset would lead to logical errors and cause financial losses to the users.

Suggestion Refactor the misusages.

2.2.2 Infinite claims of interest

Severity Medium

Status Fixed in Version 2

Introduced by Version 1

Description In the claimInterest function of the Resonate contract (see Listing 2.12), the claimPerPacket returned by the calculateAndClaimInterest function is divided by the number of packets (i.e., prinPackets). If the number of packets is very large, claimPerPacket can be zero due to the precision loss, but the interest (which can be non-zero) has been transferred to the claimer. In such a case, the interest FNFT holder can infinitely claim the interests because the sharesPerPacket would not decrease because of the precision loss.

Impact Infinite claims can happen due to the precision loss.

Suggestion Revise the code accordingly.

2.2.3 Arbitrary transfer via proxyCall

Severity High

Status Fixed in Version 2

Introduced by Version 1

Description The FNFT smart wallet (i.e., the ResonateSmartWallet contract) has a generic proxyCall interface that can do arbitrary calls by a special sandwich bot account. There is a check by design to ensure that the specific token balance is not decreased after the calls. However, the token address is specified in the parameter (i.e., targets), hence the check could easily be bypassed by providing an irrelevant token. As there are multiple types of tokens stored in the FNFT smart wallet, current checks are insufficient to ensure that these funds would not be transferred out. Besides, this function is only callable by a privileged account, which inevitably leads to a centrality problem.

```
function proxyCall(address vault, address[] memory targets, uint256[] memory values, bytes[]
    memory calldatas) external override onlyMaster nonReentrant {
  uint preBalVaultToken = IERC20(vault).balanceOf(address(this));

  for (uint256 i = 0; i < targets.length; i++) {
     (bool success, ) = targets[i].call{value: values[i]}(calldatas[i]);
     require(success, "ER022");
  }

  require(IERC20(vault).balanceOf(address(this)) >= preBalVaultToken, "ER019");
}
```

Listing 2.14: SmartWallet.sol

Impact The privileged account has the ability to transfer all funds out.

Suggestion Add sanity checks to verify the parameter.

2.2.4 Price manipulation attack

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The `valueRewardTokens` function of the `MasterChefAdapter` contract suffers from price manipulation attacks. This function simulates the process of adding liquidity to the token pair to calculate the number of LP tokens for the current reward tokens in the adapter. However, this process has a vulnerable step that can be exploited by the attacker. Specifically, at line 103, the `getAmountsOut` function is invoked to swap from the reward token to either `lpToken0` or `lpToken1` (i.e., `token0` or `token1` in the underlying pair). By manipulating the price, the variable named `amountTokenOut` can be enlarged and eventually affects the number of `lpTokens` being calculated.

```
86  function valueRewardTokens() public view virtual returns (uint256 lpTokens) {
87    if (IERC20(rewardToken).balanceOf(address(this)) > 1) {
88        uint256 rewardTokenHalf = IERC20(rewardToken).balanceOf(address(this)).div(2);
89        // ("Balance reward token: %s", IERC20(rewardToken).balanceOf(address(this)));
90
91        // ("reward token half: %s", rewardTokenHalf);
92
93        (uint reserveA, uint reserveB,) = IUniswapV2Pair(lpPair).getReserves();
94
95        uint256 reserveTokens = reserveA;
96        address[] memory tokenRoute = rewardTokenToLp0Route;
97
98        if (lpToken0 == rewardToken) {
99            reserveTokens = reserveB;
100           tokenRoute = rewardTokenToLp1Route;
101       }
102
103       uint256 amountTokenOut = IUniswapV2Router02(uniRouter).getAmountsOut(rewardTokenHalf,
              tokenRoute)[tokenRoute.length.sub(1)];
104
105       uint256 totalSupply = asset.totalSupply();
106       uint256 _kLast = IUniswapV2Pair(lpPair).kLast();
107       uint256 newSupply;
108
109       if (_kLast != 0) {
110           uint rootK = FixedPointMathLib.sqrt(uint(reserveA).mul(reserveB));
111           uint rootKLast = FixedPointMathLib.sqrt(_kLast);
112
113           if (rootK > rootKLast) {
114               uint numerator = totalSupply.mul(rootK.sub(rootKLast));
115               uint denominator = rootK.mul(5).add(rootKLast);
116               uint liquidity = numerator / denominator;
117               if (liquidity > 0) newSupply = totalSupply.add(liquidity);
118           }
119       }
120       lpTokens = amountTokenOut.mulDivDown(newSupply, reserveTokens);
121   }
122
```

```
    else return lpTokens = 0;
}
```

Listing 2.15: MasterChefAdapter.sol

Note that `MasterChefAdapter` is an ERC-4626 vault. When depositing to the vault, the corresponding shares are calculated through the `convertToShares` function. The `convertToShares` function invokes the `totalAssets` function which eventually invokes the vulnerable `valueRewardTokens` function. As a result, if the attacker successfully manipulates the return value of the `totalAssets` function, the shares he gets back would be much larger than they should be.

```
function convertToShares(uint256 assets) public view returns (uint256) {
  uint256 supply = totalSupply; // Saves an extra SLOAD if totalSupply is non-zero.

  return supply == 0 ? assets : assets.mulDivDown(supply, totalAssets());
}
```

Listing 2.16: ERC4626.sol

```
function totalAssets() public view virtual override returns (uint256) {
  (uint256 amount, ) = IMasterChef(masterChef).userInfo(poolId, address(this));
  return amount + valueRewardTokens();
}
```

Listing 2.17: MasterChefAdapter.sol

Besides, another adapter for `MasterChefV2` (i.e., the `MasterChefV2Adapter` contract) has the same problem.

Impact May lead to price manipulation attacks.

Suggestion Revise the code accordingly.

2.3 NFT Security

2.3.1 Potential DoS attack

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the procedure of matching orders, new FNFTs (which are ERC-1155 NFTs) are minted to order owners of both sides in the `_createFNFTs` function. According to the ERC-1155 specification, if the NFT token receivers are contracts, the `onERC1155Received` callback must be called. Therefore, a malicious user can submit a producer or consumer order using a contract and make the contract revert in its `onERC1155Received` callback function. Once this malicious order is put in the queue, the `Resonate` contract cannot function properly again, because the order matching is done in a FIFO manner and all incoming order matchings would fail in the callback.

```
function _createFNFTs(
  uint quantityPackets,
  bytes32 poolId,
```

```
address consumerOwner,
address producerOwner
) private returns (uint principalId) {

PoolConfig memory pool = pools[poolId];

// We should know current deposit mul from previous work
// Should have already deposited value by this point in workflow

// Initialize base FNFT config
IRevest.FNFTConfig memory fnftConfig;
// Common method, both will reference this contract
fnftConfig.pipeToContract = PROXY_OUTPUT_RECEIVER;
// Further common components
address[] memory recipients = new address[](1);
uint[] memory quantities = new uint[](1);

// Begin minting principal FNFTs

// How many principal FNFTs are we creating?
quantities[0] = quantityPackets;
// Who should get the principal FNFTs?
recipients[0] = consumerOwner;

if (pool.addInterestRate != 0) {
    // Mint Type 1
    principalId = _getRevest().mintAddressLock(PROXY_ADDRESS_LOCK, "", recipients, quantities,
        fnftConfig);
} else {
    // Mint Type 0
    principalId = _getRevest().mintTimeLock(block.timestamp + pool.lockupPeriod, recipients,
        quantities, fnftConfig);
}

// Begin minting interest FNFT

// Interest FNFTs will always be singular
// NB: Interest ID will always be +1 of principal ID
quantities[0] = 1;
recipients[0] = producerOwner;
uint interestId;

if (pool.addInterestRate != 0) {
    // Mint Type 1
    interestId = _getRevest().mintAddressLock(PROXY_ADDRESS_LOCK, "", recipients, quantities,
        fnftConfig);
}  else {
    // Mint Type 0
    interestId = _getRevest().mintTimeLock(block.timestamp + pool.lockupPeriod, recipients,
        quantities, fnftConfig);
}

{
```

```

        // GAS: Four SSTORE operations // Uses currPricePerShare twice for current and starting
            value
        activated[principalId] = Active(principalId, 1, 0, poolId);

        // GAS: Two SSTORE operations
        fnftIdToIndex[principalId] = principalId;
        fnftIdToIndex[interestId] = principalId;
    }

    emit FNFTCreation(poolId, true, principalId, quantityPackets);
    emit FNFTCreation(poolId, false, interestId, 1);
}
```

Listing 2.18: Resonate.sol

Impact A malicious order can block the order matching of the whole contract.

Suggestion Verify the owners of the orders.

2.4 Additional Recommendation

2.4.1 Check parameters in constructors and governance functions

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the constructors and governance functions, no check is specified to verify the validity of some important parameters (e.g., zero addresses or not).

```
constructor(address _router, address _proxyOutputReceiver, address _proxyAddressLock, address
    _resonateHelper) {
    REGISTRY_ADDRESS = _router;

    PROXY_OUTPUT_RECEIVER = _proxyOutputReceiver;
    PROXY_ADDRESS_LOCK = _proxyAddressLock;
    RESONATE_HELPER = _resonateHelper;
    FNFT_HANDLER = IFNFTHandler(IAddressRegistry(_router).getRevestFNFT());

    owner = msg.sender;
    emit OwnershipTransferred(address(0), msg.sender);
}
```

Listing 2.19: Resonate.sol

Impact N/A

Suggestion Add proper sanity checks.

2.4.2 Move state variable changes out of event logs

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `_enqueue` and `_dequeue` function of `Resonate` contract, there are state variable modifications in the event emissions. It is a good practice to move the state variable updates out of the emissions.

```
unchecked {
    if(isProvider) {
        providerQueue[poolId][qm.providerTail] = order; //cold? sstore
        emit EnqueueProvider(poolId, msg.sender, qm.providerTail++, shouldFarm, order);
    } else {
        consumerQueue[poolId][qm.consumerTail] = order;
        emit EnqueueConsumer(poolId, msg.sender, qm.consumerTail++, order);
    }
}
```

Listing 2.20: Resonate.sol

Impact N/A

Suggestion Revise the code accordingly.

2.4.3 Remove unused struct fields

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The `sharesPerPacket` field in the `ParamPacker` struct for the `_activateCapital` function of the `Resonate` contract is not used.

```
struct ParamPacker {
    Order consumerOrder;
    Order producerOrder;
    bool isProducerNew;
    bool isCrossAsset;
    uint quantityPackets;
    uint sharesPerPacket;
    uint currentExchangeRate;
    PoolConfig pool;
    address adapter;
    bytes32 poolId;
}
```

Listing 2.21: IResonate.sol

Impact N/A

Suggestion Remove unused struct fields.

2.4.4 Refactor clearing mapping fields into a `delete` statement

Status Fixed in `Version 2`

Introduced by `Version 1`

Description At the end of the `receiveRevestOutput` function of the `Resonate` contract, if there is no packet left for the principal FNFT ID and all interest FNFTs are claimed, the `activated` mapping field would be cleared. It is suggested that these statements should be refactored into a single `delete` statement.

```
if(prinPackets == 0 && FNFT_HANDLER.getSupply(active.principalId + 1) == 0) {
    activated[index].principalId = 0;
    activated[index].sharesPerPacket = 0;
    activated[index].startingSharesPerPacket = 0;
    activated[index].poolId = 0;
}
```

Listing 2.22: Resonate.sol

Impact N/A

Suggestion Refactor the corresponding code.

2.4.5 Remove duplicate calls in the `OutputReceiverProxy` contract

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the constructor of the `OutputReceiverProxy` contract, the `TOKEN_VAULT` address is retrieved and set as a state variable. However, in the `receiveRevestOutput` function, the vault address (i.e., the `vault` variable) is retrieved again.

```
constructor(address _addressRegistry) {
    addressRegistry = _addressRegistry;
    TOKEN_VAULT = IAddressRegistry(_addressRegistry).getTokenVault();
    FNFT_HANDLER = IFNFTHandler(IAddressRegistry(_addressRegistry).getRevestFNFT());
}
```

Listing 2.23: OutputReceiverProxy.sol

```
function receiveRevestOutput(
    uint fnftId,
    address asset,
    address payable owner,
    uint quantity
) external override {
    address vault = IAddressRegistry(addressRegistry).getTokenVault();
    require(msg.sender == vault, 'ER012');

    IResonate(resonate).receiveRevestOutput(fnftId, asset, owner, quantity);
}
```

Listing 2.24: OutputReceiverProxy.sol

Impact N/A

Suggestion Remove the duplicate calls and use the state variable.

2.4.6 Check the pool in the `MasterChefAdapter` contract

Status Acknowledged

Introduced by `Version 1`

Description In the constructor of the `MasterChefAdapter` contract, the contract should check the validity of the LP pair and the `MasterChef` pool ID. This check can be done through the `MasterChef` interface.

```
constructor(
    ERC20 _asset,
    uint256 _poolId,
    address[] memory _rewardTokenToLp0Route,
    address[] memory _rewardTokenToLp1Route,
    address _uniRouter,
    address _masterChef,
    address _rewardToken
) ERC4626(_asset, "MasterChefAdapter", "MFA") { //TODO - Change those things

    //TODO - Ownership locks on contract

    lpPair = address(_asset);
    poolId = _poolId;

    lpToken0 = IUniswapV2Pair(lpPair).token0();
    lpToken1 = IUniswapV2Pair(lpPair).token1();

    rewardTokenToLp0Route = _rewardTokenToLp0Route;
    rewardTokenToLp1Route = _rewardTokenToLp1Route;

    uniRouter = _uniRouter;
    masterChef = _masterChef;

    rewardToken = _rewardToken;
    giveAllowances();

}
```

Listing 2.25: MasterChefAdapter.sol

Impact N/A

Suggestion Add sanity checks accordingly.

Feedback from the Project This isn't something we feel is a threat to the system, as proper control of the LP pair and poolID is entirely delegated to the team. It would be possible to add some checks, but as failure here would simply result in a broken contract rather than an exploit, we feel the effort isn't worthwhile.

2.5 Note

2.5.1 Refunding procedure

Introduced by `Version 1`

Description In the `submitConsumer` function of `Resonate` contract, there is a refunding procedure. If the pool asset is different from the vault asset and the price between the two assets has changed since the matching producer order was put in the queue, the function would refund to the `DEV_ADDRESS`. According to the auditors' understanding, since the producer packet size would change as the price fluctuates, the assets retrieved from the producer cannot exactly fulfill the number of packets calculated previously. As a result, the refunding procedure is to cut off the part that is not divisible by the current producer packet size.

However, the auditors cannot derive the original meanings of the elements used in the calculation, as there does not exist any detailed illustration (noted by the developers: "the result of setting up a series of long equations and canceling out their terms"). For example, the `amountToRefund` is divided by the `packetsRemaining` when `currentExchange > previousExchange`, while the division does not occur on the opposite. After discussion, the developers confirmed that the code logic is correct because they had already tested it and would leave it as is.

```
while(hasCounterparty && consumerOrder.packetsRemaining > 0) {
  // Pull object for counterparty at head of queue
  Order storage producerOrder = _peek(poolId, true); // Not sure if I can make this memory because of Reentrancy concerns
  if(pool.asset != vaultAsset) {
      uint previousExchange = producerOrder.depositedShares;
      if(currentExchange != previousExchange) { // This will almost always be true
          uint maxPacketNumber = producerOrder.packetsRemaining * previousExchange / currentExchange; // 5
          uint amountToRefund;
          if(currentExchange > previousExchange) {
              // Position is partially or fully insolvent
              amountToRefund = _getAmountPaymentAsset(
                  (pool.rate * pool.packetSize / PRECISION) * ((producerOrder.packetsRemaining * currentExchange) -
                  (maxPacketNumber * previousExchange)),
                  1,
                  pool.asset,
                  vaultAsset
              );
              amountToRefund /= consumerOrder.packetsRemaining;

          } else {
              // There will be a surplus in the position
              amountToRefund = _getAmountPaymentAsset(
                  (pool.rate * pool.packetSize / PRECISION) * ((maxPacketNumber * previousExchange) -
                  (producerOrder.packetsRemaining * currentExchange)),
                  1,
                  pool.asset,
                  vaultAsset
              );
          }

          if(maxPacketNumber == 0) {
              // Need to cancel the order because it is totally insolvent
              // No storage update
              _dequeue(poolId, true);
              wallet.withdraw(amountToRefund, pool.asset, producerOrder.owner.toAddress());
              hasCounterparty = !_isQueueEmpty(poolId, true);
              continue;
          }
```

```
        // Storage update
        producerOrder.depositedShares = currentExchange;
        producerOrder.packetsRemaining = maxPacketNumber;


        wallet.withdraw(amountToRefund, pool.asset, DEV_ADDRESS);
    }
}
if (producerOrder.owner.toAddress() == address(0)) {
    // Order has previously been cancelled
    // Dequeue and move on to next iteration
    // No storage update
    _dequeue(poolId, true);
} else {
    uint digestAmt;
    {
        uint consumerAmt = consumerOrder.packetsRemaining;
        uint producerAmt = producerOrder.packetsRemaining;
        digestAmt = producerAmt >= consumerAmt ? consumerAmt : producerAmt;
    }
    _activateCapital(ParamPacker(consumerOrder, producerOrder, false, pool.asset != vaultAsset,
         digestAmt, 0, currentExchange, pool, adapter, poolId));

    consumerOrder.packetsRemaining -= digestAmt;
    producerOrder.packetsRemaining -= digestAmt; // NB: Consider modification not via multiple
        storage methods, gas optimziation

    amount -= (digestAmt * pool.packetSize);

    // Handle _dequeue as needed
    if (producerOrder.packetsRemaining == 0) {
        _dequeue(poolId, true);
    }
}
// Check if queue is empty
hasCounterparty = !_isQueueEmpty(poolId, true);
}
```

Listing 2.26: Resonate.sol

2.5.2 ID continuity assumption of the interest and principal FNFTs

Introduced by `Version 1`

Description The token economics of the `Resonate` project are based on the FNFT of the `Revest` project. Once two orders are matched, the `Resonate` contract would call the contracts of the `Revest` project for minting two kinds of FNFT, i.e., *interest FNFT* and *principal FNFT*, respectively. In the current implementation of the `Revest` contract of the `Revest` project, the FNFT minting procedure is protected by `nonReentrant` guard so that the ID of the interest FNFT is always the ID of the principal FNFTs plus one for each order. All financial actions are performed based on this assumption.

Although the current logic and dependency seem to be sound, there does not exist actual checks in the `Resonate` contract to ensure the assumption. Considering that the address of `Revest` contract is retrieved from a `REGISTRY_ADDRESS`, there is a possibility that the two FNFTs do not necessarily satisfy the assumption in the future versions of the projects.

```
function _createFNFTs(
    uint quantityPackets,
    bytes32 poolId,
    address consumerOwner,
    address producerOwner
) private returns (uint principalId) {

    PoolConfig memory pool = pools[poolId];

    // We should know current deposit mul from previous work
    // Should have already deposited value by this point in workflow

    // Initialize base FNFT config
    IRevest.FNFTConfig memory fnftConfig;
    // Common method, both will reference this contract
    fnftConfig.pipeToContract = PROXY_OUTPUT_RECEIVER;
    // Further common components
    address[] memory recipients = new address[](1);
    uint[] memory quantities = new uint[](1);

    // Begin minting principal FNFTs

    // How many principal FNFTs are we creating?
    quantities[0] = quantityPackets;
    // Who should get the principal FNFTs?
    recipients[0] = consumerOwner;

    if (pool.addInterestRate != 0) {
        // Mint Type 1
        principalId = _getRevest().mintAddressLock(PROXY_ADDRESS_LOCK, "", recipients,
            quantities, fnftConfig);
    } else {
        // Mint Type 0
        principalId = _getRevest().mintTimeLock(block.timestamp + pool.lockupPeriod, recipients
            , quantities, fnftConfig);
    }

    // Begin minting interest FNFT

    // Interest FNFTs will always be singular
    // NB: Interest ID will always be +1 of principal ID
    quantities[0] = 1;
    recipients[0] = producerOwner;
    uint interestId;

    if (pool.addInterestRate != 0) {
        // Mint Type 1
        interestId = _getRevest().mintAddressLock(PROXY_ADDRESS_LOCK, "", recipients,
            quantities, fnftConfig);
```

```
    } else {
        // Mint Type 0
        interestId = _getRevest().mintTimeLock(block.timestamp + pool.lockupPeriod, recipients, quantities, fnftConfig);
    }

    {

        // GAS: Four SSTORE operations // Uses currPricePerShare twice for current and starting value
        activated[principalId] = Active(principalId, 1, 0, poolId);

        // GAS: Two SSTORE operations
        fnftIdToIndex[principalId] = principalId;
        fnftIdToIndex[interestId] = principalId;
    }

    emit FNFTCreation(poolId, true, principalId, quantityPackets);
    emit FNFTCreation(poolId, false, interestId, 1);
}
```

Listing 2.27: Resonate.sol

```
function _getRevest() private view returns (IRevest) {
    return IRevest(IAddressRegistry(REGISTRY_ADDRESS).getRevest());
}
```

Listing 2.28: Resonate.sol

2.5.3 Potential vulnerability in the `harvest` function

Introduced by `Version 1`

Description In the ERC-4626 adapter for the `MasterChef` contract (i.e., the `MasterChefAdapter` contract), there is a public function called `harvest` which does not have access control. This function harvests rewards from the underlying `MasterChef` contract, swaps in the Uniswap router, and then adds liquidity into the Uniswap/SushiSwap pair. However, no price slippage check is performed in the swapping process.

Since this function is public that can be invoked by anyone (and any contract), a malicious attacker can first manipulate the price of the underlying pool, then call the `harvest` function to swap tokens and provide liquidity in an unbalanced pool, finally swap back to make a profit. This attack is profitable if there are enough tokens in the adapter contract.

As stated by the developers, it is not a critical issue for the difficulty of gaining the profit. However, the producers mainly profit from the underlying interests. When the attack is performed, the producers would always suffer from losses.

```
function harvest() public {
  // require(!Address.isContract(msg.sender), "ER029");
  IMasterChef(masterChef).deposit(poolId, 0);
  addLiquidity();
  deposit();
}
```

Listing 2.29: MasterChefAdapter.sol

```
function addLiquidity() internal {
  uint256 rewardTokenHalf = IERC20(rewardToken).balanceOf(address(this)).div(2);

  if (lpToken0 != rewardToken) {
      IUniswapV2Router02(uniRouter).swapExactTokensForTokensSupportingFeeOnTransferTokens(
          rewardTokenHalf, 0, rewardTokenToLp0Route, address(this), block.timestamp.add(100));
  }

  if (lpToken1 != rewardToken) {
      IUniswapV2Router02(uniRouter).swapExactTokensForTokensSupportingFeeOnTransferTokens(
          rewardTokenHalf, 0, rewardTokenToLp1Route, address(this), block.timestamp.add(600));
  }

  uint256 lp0Bal = IERC20(lpToken0).balanceOf(address(this));
  uint256 lp1Bal = IERC20(lpToken1).balanceOf(address(this));

  IUniswapV2Router02(uniRouter).addLiquidity(lpToken0, lpToken1, lp0Bal, lp1Bal, 1, 1, address(
      this), block.timestamp.add(600));

}
```

Listing 2.30: MasterChefAdapter.sol

Feedback from the Project We've spoken with some other security researchers and yield farmers about the `harvest` question. We think, essentially, it is a one-transaction sandwich attack against DEX to gain profit from slippage. However, it should be not a very critical issue. It is very difficult to gain profit, since it requires the amount of locked reward Token is very huge. Besides, bunch of projects in the wild adopt this style of harvest. At least, it would not worth a bug bounty. I think it's something we'll likely have two versions of our contracts for and work with protocols on a case-by-case basis to figure out what's right for them.