Frax Finance Audit Report

(Curve AMO)

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 May 22, 2024 through May 31, 2024, the Frax Security Cartel conducted an audit on the Curve AMO component of the frxETH V2 codebase. The scope of this review was in Frax's frxETH_V2 GitHub repository on commit hash ecb8ab64d5cf71fa4706e8f100f92bdf679fb7b9, specifically in the `contracts/curve-amo/` directory.

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, **31 findings** were identified. This includes **2 high**, **5 medium**, **13 low**, and **8 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 High Severity Findings

3.1.1 Ineffective slippage when withdrawing liquidity from the pool in one coin

Description: In the Curve AMO, unbacked `frxETH` is minted and paired 1:1 with the other pool token to deepen the peg liquidity. The operator can redeem LP tokens by directly withdrawing or requesting ether from the AMO. In either of these cases, `_withdrawOneCoin()` will calculate `_absMinCoinOut` as `(_lpIn * 1e18) / _lpPerCoinsBalancedE18[_coinIndex]` where the `_lpPerCoinsBalancedE18` array is populated from `CurveLsdAmoHelper.calcMiscBalancedInfoWithParams()`:

```
_lpPerCoinsBalancedE18[0] = (_lpTotalSupply * 1e18) / pool.balances(0);
_lpPerCoinsBalancedE18[1] = (_lpTotalSupply * 1e18) / pool.balances(1);
```

This assumes a balanced LP token redemption which would understate the `_min_amount` parameter by ~50% when calling `pool.remove_liquidity_one_coin()`, making it susceptible to some considerable slippage.

Recommendation: Calculate the expected amount for a one coin withdrawal and apply the tolerated slippage to this amount.

Frax: Fixed in commit 716497c.

Frax Security Cartel: Verified. `_withdrawOneCoin()` now makes use of the Curve pool's `calc_withdraw_one_coin()` function which handles the amount expected when redeeming LP tokens for one coin. Slippage settings are applied to `_absMinCoinOut` depending on the caller and overridden if the user-supplied value is less than this.

Note: it's important that user-supplied slippage overwrites `_absMinCoinOut` because `calc_withdraw_one_coin()` can return a manipulatable value for `_absMinCoinOut`.

3.1.2 Deposits can be sandwiched to steal a portion of the deposited ETH

Description: ETH is deposited into the AMO when `sweepEther()` is called on the Router.

When this transaction is seen in the mempool, an attacker can create a Flashbots bundle to sandwich it with swaps in and out of the Curve pool on either side of the deposit. Because the added liquidity increases the pool's resistance to price movement, the attacker can profit by getting less slippage on the trade back than the original trade.

Because of the settings of a number of variables (namely, the max pool allocation of 10_000 and the slippage being calculated incorrectly in `calcMiscBalancedInfoWithParams()`), this has limited potential for abuse. However, it can be used to steal just over 0.01% of any deposit, as demonstrated in the proof of concept below.

Proof of Concept: The following test can be dropped into `EtherRouterTest.t.sol` to demonstrate the issue. See comments for an explanation of what's happening.

```
function testZach_CurveManipulation() public {
    // set up router and add eth
    etherRouterSetUp();
    vm.deal(address(etherRouter), 10000 ether);

    // set up callable curve pool
    CurvePool curvePool = CurvePool(0x9c3B46C0Ceb5B9e304FCd6D88Fc50f7DD24B31Bc);

    // assert we are starting with no weth and no frxETH
    assert(WETH.balanceOf(address(this)) == 0);
    assert(frxETH.balanceOf(address(this)) == 0);

    // "flashloan" 1200 frxETH
    deal(address(frxETH), address(this), 1200 ether);

    // attacker swaps frxeth to eth to throw off balance
    frxETH.approve(address(curvePool), 1200 ether);
    curvePool.exchange(1, 0, 1200 ether, 0);

    // sandwiched deposit transaction, adding liquidity reduces slippage
    hoax(ConstantsSBTS.Mainnet.TIMELOCK_ADDRESS);
    etherRouter.sweepEther(1_000 ether, true);

    // attacker swaps back and profits
    uint wethBal = WETH.balanceOf(address(this));
    WETH.approve(address(curvePool), wethBal);
    curvePool.exchange(0, 1, wethBal, 0);

    // "return the flashloan" by burning frxeth we dealt
    frxETH.transfer(address(1), 1200 ether);

    // resulting profit
    assertEq(frxETH.balanceOf(address(this)), 120455520335216745);
}
```

The result is a profit of 0.12 ETH, which is just over 0.01% of what was being deposited.

Note that the `CurvePool` interface will need to be added to the file for it to run:

```
interface CurvePool {
    function exchange(int128 i, int128 j, uint256 dx, uint256 min_dy) external;
    function get_virtual_price() external view returns (uint256);
    function balances(uint i) external view returns (uint256);
}
```

Also note that I updated the block for the fork to `19992241`, and based on a separate issue, have updated `CONVEX_FXS_BOOSTER` to `0xD8Bd5Cdd145ed2197CB16ddB172DF954e3F28659`.

Recommendation: The easiest solution is to always use Flashbots or some other private RPC for `sweepEther()` calls.

The alternative would be to specify a maximum ratio between the two deposited tokens that is acceptable. Because only the amount of ETH is inputted and the amount of frxETH is derived from that, this check would avoid an

attacker being able to trick the contract into providing more frxETH than is acceptable.

Frax: Confirmed that private transactions will be used for `sweepEther()` calls (and noted this in function comments). Additionally, added min/max in commit d4f9a6a.

Frax Security Cartel: Our testing shows that the min/max fix does not fully protect against this issue. However, since `sweepEther()` is permissioned and only private transactions will be used, this fix is verified.

3.2 Medium Severity Findings

3.2.1 AMO LP allocations are manipulatable

Description: The Curve AMO helper contract has a `showAllocationsWithParams()` function to read allocations of `frxETH` and `ETH/LSD/WETH` which is almost entirely used solely by `LendingPoolCore._getUtilizationPostCore()` in interest calculations.

More specifically, `_cachedBals.frxEthInLpBalanced` represents the `frxETH` amount available when a balanced withdrawal is performed on the AMO's LP token balance. Balanced withdrawals will redeem a proportional amount of all pool tokens dependent on the LP token holder's share of all LP tokens. Hence, large swaps in the pool can impact this as Curve stable pools are effective in reducing slippage due to their design.

Recommendation: Ensure large swaps in the Curve pools are unable to manipulate interest rate calculations in any meaningful way.

Frax: Utilization logic has been simplified by calculating it as a function of total borrowed ETH in commit e1278b2. The redemption queue shortage/surplus is included as part of the calculation in commit 3a14474 as well as a delay to the utilization rate used in interest rate calculations. Every interest calculation will use the previously stored utilization rate, meaning only the future rate can be manipulated which allows borrowers to react to direct manipulation attacks.

Frax Security Team: Verified fixes.

3.2.2 Curve pool read-only reentrancy can affect `getConsolidatedEthFrxEthBalancePacked()`

Description: In some Curve pools, there is a known issue where control flow can be granted to the user when the pool's state is inconsistent. For example, notice how the `raw_call` below can give a user control flow when one or more balances have decreased, but the LP token total supply has not been updated:

```
@external
@nonreentrant('lock')
def remove_liquidity(_amount: uint256, _min_amounts: uint256[N_COINS]) -> uint256[N_COINS]:
    lp_token: address = self.lp_token
    total_supply: uint256 = CurveToken(lp_token).totalSupply()
    amounts: uint256[N_COINS] = empty(uint256[N_COINS])

    for i in range(N_COINS):
        old_balance: uint256 = self.balances[i]
```

```
        value: uint256 = old_balance * _amount / total_supply
        assert value >= _min_amounts[i], "Withdrawal resulted in fewer coins than expected"
        self.balances[i] = old_balance - value
        amounts[i] = value
        coin: address = self.coins[i]
        if coin == 0xEeeeeEeeeEeEeeEeEeEeeEEEeeeeEeeeeeeeEEeE:
            raw_call(msg.sender, b"", value=value)
        else:
            assert ERC20(coin).transfer(msg.sender, value, default_return_value=True)

    CurveToken(lp_token).burnFrom(msg.sender, _amount)  # dev: insufficient funds

    log RemoveLiquidity(msg.sender, amounts, empty(uint256[N_COINS]), total_supply - _amount)

    return amounts
```

Typically, this issue doesn't affect integrating protocols because they call other functions in the Curve pool that have reentrancy guards. However, this is not the case with the `CurveLsdAmoHelper`, since `getConsolidatedEthFrxEthBalancePacked()` reads the Curve pool's state but doesn't trigger any reentrancy guards.

Specifically, the downstream `calcTknsForLPBalancedWithParams()` function is affected by the pool's inconsistent state, but none of its external calls (`totalSupply()` and `balances()`) trigger reentrancy guards in the Curve pool:

```
function calcTknsForLPBalancedWithParams(
    address _poolAddress,
    CurveLsdAmo.PoolInfo memory _poolInfo,
    uint256 _lpAmount
) public view returns (uint256[2] memory _withdrawables) {
    // Get the total LP supply
    ERC20 _lpToken = ERC20(_poolInfo.lpTokenAddress);
    uint256 _lpTotalSupply = _lpToken.totalSupply();

    IMinCurvePool pool = IMinCurvePool(_poolAddress);
    for (uint256 i = 0; i < 2; ) {
        _withdrawables[i] = (pool.balances(i) * _lpAmount) / _lpTotalSupply;
        unchecked {
            ++i;
        }
    }
}
```

As a result, the `_withdrawables` return value can be based on an inconsistent state and can therefore be incorrect. This has downstream consequences in the frxETHV2 `EtherRouter` and `LendingPool` contracts, which eventually use the `_withdrawables` values to calculate a utilization rate.

Recommendation: Add a call to a function with a reentrancy guard somewhere in the `getConsolidatedEthFrxEthBalancePacked()` code. Since this issue currently only seems relevant to the frxETH/ETH Curve pool, and since the frxETH/ETH pool has reentrancy guards on `get_virtual_price()`, the following change is sufficient to trigger the reentrancy guard:

```
function calcTknsForLPBalancedWithParams(
    address _poolAddress,
    CurveLsdAmo.PoolInfo memory _poolInfo,
    uint256 _lpAmount
) public view returns (uint256[2] memory _withdrawables) {
    // Get the total LP supply
    ERC20 _lpToken = ERC20(_poolInfo.lpTokenAddress);
    uint256 _lpTotalSupply = _lpToken.totalSupply();

    IMinCurvePool pool = IMinCurvePool(_poolAddress);
+   pool.get_virtual_price();
    for (uint256 i = 0; i < 2; ) {
        _withdrawables[i] = (pool.balances(i) * _lpAmount) / _lpTotalSupply;
        unchecked {
            ++i;
        }
    }
}
```

To prevent issues arising from different pool implementations, consider adding a call to `get_virtual_price()` in the `CurveLsdAmo` constructor to ensure this function exists in the pool. It's also suggested to include some comments/notes about this behavior.

Frax: Added the `calcTknsForLPBalancedWithParams()` fix in commit 8557f7f. Added the constructor call and warning comments in commit 20f2d68.

Frax Security Cartel: Verified.

3.2.3 Insufficient validation for oracle calls in AMO Helper

Description: In `CurveLsdAmoHelper.sol`, we use a Chainlink oracle to pull the price of ETH in USD.

```
    function getEthPriceE18() public view returns (uint256) {
        (uint80 _roundId, int256 _price, , uint256 _updatedAt, uint80 _answeredInRound) =
            priceFeedEthUsd
            .latestRoundData();
        if (!(_price >= 0 && _updatedAt != 0 && _answeredInRound >= _roundId)) revert
            InvalidChainlinkPrice();

        return (uint256(_price) * 1e18) / (10 ** chainlinkEthUsdDecimals);
    }
```

The validations after this oracle call check that: 1) the returned price isn't 0 2) the last updated time isn't 0 3) the round answered is greater than or equal to roundId

Importantly, these checks do not include a check for the freshness of the oracle, which can result in stale data.

Recommendation: It is recommended to include a check that `block.timestamp - updatedAt > THRESHOLD`, where `THRESHOLD` is the acceptable age of an oracle update to be included in the system.

It is also not necessary to check that `_updatedAt != 0 && _answeredInRound >= _roundId`.

Frax: Fixed in commit 75b8d0c. The code currently uses a tolerance of 1 year for testing, but 6 hours will be the value used in deployment.

Frax Security Cartel: Verified.

3.2.4 `_depositToCurveLP()` doesn't use existing stETH balance

Description: In the `_depositToCurveLP()` function, the code may convert raw ETH to the appropriate token for the Curve pool, based on the value specified by `poolInfo.ethType`:

```
// If WETH or stETH is part of the pool, instead of ETH
if (poolInfo.ethType == EthType.WETH) {
    // See how much WETH you currently have
    uint256 _currWeth = WETH.balanceOf(address(this));

    // Use existing WETH first
    if (_currWeth >= _ethIn) {
        // Do nothing and use existing WETH
    } else {
        // Convert ETH to WETH as needed

        // ETH -> WETH
        WETH.deposit{ value: _ethIn - _currWeth }();
    }
} else if (poolInfo.ethType == EthType.STETH) {
    // ETH -> stETH
    convertEthToStEth(_ethIn);
}
```

In the case where `poolInfo.ethType == EthType.STETH`, notice that this code will always attempt to convert `_ethIn` amount of ETH into stETH, regardless of if the contract already has stETH. It's possible that the contract holds stETH but does not hold ETH, which will lead to this code reverting. In order to successfully call `_depositToCurveLP()` in this scenario, an otherwise unnecessary swap of stETH to ETH will be required beforehand.

Recommendation: Change the `poolInfo.ethType == EthType.STETH` scenario to match the behavior of the `poolInfo.ethType == EthType.WETH` case. This code only does a conversion if the existing balance of the token is not sufficient for the deposit.

Frax: Fixed in commit d32c5a0.

Frax Security Cartel: Verified.

3.2.5 Accounting issue causes reverts on partially filled withdrawals

Description: When ETH is withdrawn from the AMO to the router, we withdraw in the following order: (1) withdraw native ETH, (2) scrounge WETH / stETH, (3) withdraw from CVX vault and convert to ETH.

For the final step, we check the balance of the Convex rewards contract:

```
uint256 _lpAvailable = IConvexBaseRewardPool(poolInfo.rewardsContractAddress).balanceOf(address(
    this));
```

Based on that balance, we perform the withdrawal as follows:

```
if (_lpAvailable > 0) {
    // Do the unwrap (vaulted cvxLP -> coins). Skip reward claim to save gas
    _withdrawAndUnwrapVaultedCvxLP(_lpNeeded, false);

    // Do the LP withdrawal (LP -> coins)
    if (_useOneCoin) {
        // Do the withdrawal (oneCoin)
        // Min out of 0 here will get corrected to the slippage-allowable amount downstream
        _coinsOutActual = _withdrawOneCoin(_caller, _lpNeeded, _ethIndex, 0);
    } else {
        // Do the withdrawal (balanced)
        // Min outs of [0, 0] here will get corrected to the slippage-allowable amount downstream
        _coinsOutActual = _withdrawBalanced(_caller, _lpNeeded, _coinsOutMinToUse);
    }

    // Note the amount of ETH withdrawn
    _withdrawnEth = _coinsOutActual[_ethIndex];
} else {
    // Zero out the remaining ETH
    _remainingEth = 0;
}
```

In the event that the `_lpAvailable` value is `0`, we will skip the withdrawal logic and set `_remainingEth = 0`. This can happen even if there was remaining ETH waiting to be withdrawn.

At the end of the function, there is a sanity check to make sure the `_remainingEth` matches up with the amount of the request that was actually not sent out:

```
// Sanity check
if (_remainingEth != (_ethRequested - _ethOut)) {
    revert RequestEtherSanityCheck(_remainingEth, (_ethRequested - _ethOut));
}
```

In the case that this value is set to 0, this will always fail and revert the transaction.

This could happen in the following situations: 1) There just aren't any funds left in the Convex vault. Normally, the router would then move on to the next AMO to continue its withdrawal, but it will revert instead. 2) The funds are in the stkCvx vault, which requires a manual withdrawal by the operator. In this case, `_lpAvailable` will equal zero (because it only measures funds directly in Convex) and the above issue will occur.

Proof of Concept: The following test can be dropped into `CurveAMOFunctionsTest.t.sol` to demonstrate:

```
function testZach_RemainingMisCalc() public {
    amoFunctionsTestSetup();
    uint ethBal = address(curveLsdAmo).balance;
    uint wethBal = WETH.balanceOf(address(curveLsdAmo));
    curveLsdAmo.requestEtherByOperator(
        payable(ConstantsSBTS.Mainnet.CURVEAMO_OPERATOR_ADDRESS),
        ethBal + wethBal + 1e18,
        true
    );
}
```

```
[FAIL. Reason: RequestEtherSanityCheck(0, 1000000000000000000 [1e18])] testZach_RemainingMisCalc()
    (gas: 47379705)
```

Recommendation: `_remainingEth` should not be altered in the event that the withdrawal does not occur.

Frax: Fixed in commit a00275a.

Frax Security Cartel: Verified.

3.3 Low Severity Findings

3.3.1 `tokenDeposited` counts are inaccurate

Description: The Curve AMO tracks a `tokenDeposited` count for each of the underlying tokens. When tokens are deposited, this value is incremented for each.

This value is used in the `onBudget()` modifier to ensure that the total deposited tokens do not exceed the cap.

```
modifier onBudget() {
    _;
    for (uint256 i = 0; i < 2; ) {
        if (poolInfo.tokenDeposited[i] > poolInfo.tokenMaxAllocation[i]) {
            revert OverTokenBudget();
        }
        unchecked {
            ++i;
        }
    }
}
```

However, there is no accounting mechanism to ensure these values remain accurate. In a general sense, when Token A is deposited, Token B can be withdrawn, meaning these values do not necessarily represent the amount of tokens that are currently deposited in the pool.

This is handled by simply zeroing out the count if a token's balance is withdrawn below zero:

```
if (_amountReceived < poolInfo.tokenDeposited[_coinIndex]) {
    poolInfo.tokenDeposited[_coinIndex] -= _amountReceived;
} else {
    poolInfo.tokenDeposited[_coinIndex] = 0;
}
```

Each time this "zeroing out" occurs, tokens are withdrawn that aren't accounted for. Eventually, it can lead to `tokenDeposited` values that are far in excess of reality.

As an example, imagine 100 Token A are deposited, then 100 Token B are withdrawn. Each time we repeat this cycle, `poolInfo.tokenDeposited[TokenA]` is incremented by 100, while the value for Token B stays the same.

Recommendation: Replace the separate `tokenDeposited` accounting with something simpler, such as tracking the total LP Token holdings.

Frax: Fixed by tracking LP tokens in commit fec7ad7a. Small additional fixes made in commit c20d4e2 and commit c2c5496.

Frax Security Cartel: Verified.

3.3.2 `_withdrawBalanced()` hardcoded `_minAmountsToUse` considerations

Description: The `_withdrawBalanced()` function currently ignores the `_minAmountsFromUser` parameter and uses hardcoded zeros for its `_minAmountsToUse` values. According to the function's comments, this approach is deemed safe for LP withdrawals under any pool condition. The rationale is that a withdrawal from a manipulated pool will always result in a greater total value compared to a withdrawal from a pool with balances that reflect the real prices of the underlying assets.

While this rationale does seem correct, it should be noted the actual composition of the withdrawn assets depends on the state of the pool. This consideration may be important when specific assets are needed from the withdrawal, for example with the `_requestEther()` function, which uses `_withdrawBalanced()` to withdraw ETH specifically. In a manipulated pool, the function could instead return a large amount of the other token, which would not be as useful.

Recommendation: Consider whether manipulated balanced withdrawals could pose issues in cases where a specific asset is needed from the withdrawal. If not, consider removing the `_minAmountsFromUser` parameter altogether. If so, consider allowing the `_minAmountsFromUser` parameter to override the `_minAmountsToUse` values passed to `remove_liquidity()`.

Frax: Addressed in commit 9a2de6c.

Frax Security Cartel: Verified.

3.3.3 Unnecessary LP token approval

Description: At the start of both the `_withdrawBalanced()` and `_withdrawOneCoin()` functions, LP token approval is granted to the underlying Curve pool. This approval is unnecessary, since the Curve pool implementations do not

require an LP token approval to withdraw the user's liquidity. For example, notice that the `stETHfrxETH_Pool` does not consume an allowance in `remove_liquidity()`, and instead decrements an internal balance with this code:

```
self.balanceOf[msg.sender] -= _burn_amount
```

For another example, notice that the `frxETHETH_Pool` calls `burnFrom()` with this code:

```
CurveToken(lp_token).burnFrom(msg.sender, _amount)
```

In the LP token code, it can be seen that `burnFrom()` is a privileged call from the Curve pool, and does not require a prior approval to be granted:

```
@external
def burnFrom(_to: address, _value: uint256) -> bool:
    """
    @dev Burn an amount of the token from a given account.
    @param _to The account whose tokens will be burned.
    @param _value The amount that will be burned.
    """
    assert msg.sender == self.minter

    self.totalSupply -= _value
    self.balanceOf[_to] -= _value

    log Transfer(_to, ZERO_ADDRESS, _value)
    return True
```

Recommendation: Remove the unnecessary LP token approval from both `_withdrawBalanced()` and `_withdrawOneCoin()`.

Frax: Fixed in commit 5f89cf7.

Frax Security Cartel: Verified.

3.3.4 `vaultKekIds` array iteration can be avoided

Description: The `lpInVaultsWithParams()` function returns information about LP deposits in the `rewardsContractAddress` and the `fxsPersonalVaultAddress`. This is implemented as follows:

```
function lpInVaultsWithParams(
    CurveLsdAmo _curveAmo,
    CurveLsdAmo.PoolInfo memory _poolInfo
) public view returns (uint256 inCvxRewPool, uint256 inStkCvxFarm, uint256 totalVaultLP) {
    // cvxLP
    if (_poolInfo.hasCvxVault) {
```

```
            IConvexBaseRewardPool _convexBaseRewardPool = IConvexBaseRewardPool(_poolInfo.
                rewardsContractAddress);
            inCvxRewPool = _convexBaseRewardPool.balanceOf(address(_curveAmo));
        }

        // stkcvxLP
        if (_poolInfo.hasStkCvxFxsVault) {
            bytes32[] memory _theseKeks = _curveAmo.getVaultKekIds();
            for (uint256 i = 0; i < _theseKeks.length; ) {
                inStkCvxFarm += _curveAmo.kekIdTotalDeposit(_theseKeks[i]);
                unchecked {
                    ++i;
                }
            }
        }
        totalVaultLP = inCvxRewPool + inStkCvxFarm;
    }
```

Notice that the `fxsPersonalVaultAddress` deposits are tracked with individual entries, which are iterated through to calculate a total deposit sum. If the AMO's `vaultKekIds` array becomes large over time, this operation can be very gas-expensive.

Recommendation: To prevent future inefficiencies caused by iterating through the `vaultKekIds` array, consider maintaining a real-time total of `vaultKekIds` values using a new storage variable. This can be accomplished by adding the following to the end of `depositCurveLPToVaultedStkCvxLP()`:

```
kekIdGlobalSum += _poolLpIn;
```

In addition to adding the following to the start of `withdrawAndUnwrapFromFxsVault()`:

```
kekIdGlobalSum -= kekIdTotalDeposit[_kekId];
```

Alternatively, consider utilizing the `fxsPersonalVaultAddress` itself to calculate the AMO's total deposits. This can be accomplished with code similar to the following:

```
IFxsPersonalVault fxsVault = IFxsPersonalVault(poolInfo.fxsPersonalVaultAddress);
FraxUnifiedFarmTemplate farm = FraxUnifiedFarmTemplate(fxsVault.stakingAddress());
inStkCvxFarm = farm.lockedLiquidityOf(poolInfo.fxsPersonalVaultAddress);
```

Frax: Fixed in commit 658701b.

Frax Security Cartel: Verified.

3.3.5 `_requestEther()` doesn't withdraw all types of deposits

Description: The `CurveLsdAmo` contract stores its Curve LP tokens in three potential locations:

1. "Free" LP tokens in the AMO's balance.
2. Deposits in the `rewardsContractAddress` contract.
3. Deposits in the `fxsPersonalVaultAddress` contract.

Currently, the `_requestEther()` function is only capable of withdrawing funds from the LP tokens in location (2). On the other hand, the `showAllocationsWithParams()` helper function tracks the ETH/frxETH from LP tokens across all three locations. As a result, in certain situations (e.g. with `fullRedeemNft()` in the `FraxEtherRedemptionQueueV2`) `_requestEther()` might be called in an attempt to withdraw ETH that the function is not capable of accessing. This can lead to reverts in code downstream from the `requestEther()` call.

Recommendation: Consider adding the ability for `_requestEther()` to withdraw from LP tokens in location (1). If possible, implementing programmatic withdrawals from location (3) would also be desirable, although this would require managing lock timestamps/ids and would be more complicated.

Frax: Fixed in commit b624a1b. The `_requestEther()` function can now withdraw from locations (1) and (2), with location (1) being used first if there are excess LP tokens.

Frax Security Cartel: Verified.

3.3.6 `lpAmount` can be miscalculated in out of balance pools

Description: We use the helper contract's `lpAmount` calculation both to (a) determine our slippage tolerance when performing deposits and (b) determine how much LP to withdraw to get the amount of ETH we would like.

This calculation will be slightly off when pools are very out of balance. Let's look at an example.

1) To determine the LP amount, the helper performs the following:

```
_lpAmount = ((_desiredCoinAmt + _undesiredCoinAmt) * 1e18) / _lp_virtual_price;
```

This incorrectly assumes that `_lp_virtual_price` is based on an average of the two token amounts.

2) On the Curve contract, on the other hand, the amount to actually withdraw is calculated as follows.

```
old_balance: uint256 = self.balances[i]
value: uint256 = old_balance * _burn_amount / total_supply
```

This doesn't use the virtual price of the LP token, and instead takes the percentage of the LP token supply that's being burned, and provides that share of each token's balance.

Proof of Concept: The following test can be dropped into `EtherRouterTest.t.sol`, which calls out to the view functions to show the gap.

```
function testZach_CurveBalances() public {
    // set up contract and callable curve pool
    etherRouterSetUp();
    CurvePool curvePool = CurvePool(0x9c3B46C0Ceb5B9e304FCd6D88Fc50f7DD24B31Bc);

    // swap 10_000 frxETH to create imbalance
    deal(address(frxETH), address(this), 10_000 ether);
    frxETH.approve(address(curvePool), 10_000 ether);
    curvePool.exchange(1, 0, 10_000 ether, 0);

    // now let's look at some data
    uint amountOfEthToWithdraw = 100 ether;
    (uint lpAmount,,,,) = amoHelper.calcMiscBalancedInfo(curveLsdAmoAddress, 0,
        amountOfEthToWithdraw);
    uint amountOfEthWithdrawn = curvePool.balances(0) * lpAmount / curvePool.totalSupply();

    assertEq(amountOfEthWithdrawn, 13439284671757509106l);
}
```

This shows that the LP amount is miscalculated in this imbalance situation such that it results in 34% more tokens being withdrawn than were requested.

Recommendation: The amount of LP tokens that can be withdrawn to generate a given number of underlying tokens can be calculated as:

```
uint lpAmount = _desiredCoinAmt * pool.totalSupply() / pool.balances(_desiredCoinIdx)
```

Note that this fix increases the risk to #15, so let's discuss to make sure there is a solution to that problem that doesn't rely on this miscalculation.

Frax: Fixed as recommended in commit 6cdfc53.

Frax Security Cartel: Verified.

3.3.7 stETH rounding issues

Description: In order to implement rebasing balances, the stETH contract calculates balances using an underlying "shares" accounting system. As a result of this implementation, a known issue can arise where small rounding errors happen in ETH to stETH conversions and stETH transfers. The following two resources describe this issue in detail:

- Lido integration guide
- Lido GitHub issue

In the `CurveLsdAmo`, this issue can lead to unintended reverts.

For example, whenever stETH is transferred into the AMO, the `stETH.balanceOf()` increase may not equal the expected transfer amount. This is handled correctly in `convertEthToStEth()`. However this is not handled correctly

in the downstream `_depositToCurveLP()` code, since it proceeds with the `add_liquidity()` call assuming the conversion was 1:1. This can lead to an insufficient balance revert during the `transferFrom()` call.

Also, whenever stETH is transferred out of the AMO, the receiving contract may change its behavior based on its own `stETH.balanceOf()` logic. For example, if either of the pools defined in `convertStEthToEth()` did their own `stETH.balanceOf()` tracking, the `exchange()` call may return slightly fewer tokens than the initial estimate returned from `get_dy()`. This can lead to a revert given the `get_dy()` estimate is used as the `_min_dy` parameter. However, both Curve pools in the `convertStEthToEth()` function do not seem to have logic depending on `balanceOf()` tracking, which appears to make the `convertStEthToEth()` safe from this issue.

Recommendation: For the `_depositToCurveLP()` issue, consider updating the code to not assume the `convertStEthToEth()` conversion is 1:1:

```
  // ETH -> stETH
- convertEthToStEth(_ethIn);
+ _coinsInToUse[_ethIndex] = convertEthToStEth(_ethIn);
```

Also, keep this behavior in mind for potential issues with transferring stETH out of the AMO. If any external contracts actually inspect the `stETH.balanceOf()` from the transfer, the resulting behavior may be unexpected.

Frax: Fixed in commit ad0a9ae.

Frax Security Cartel: Verified.

3.3.8 Wrong `CONVEX_FXS_BOOSTER` used, bricking user vault creation

Description: As a part of the AMO deployment process, we call `CONVEX_FXS_BOOSTER.createVault(63);`.

This address is set to the following hardcoded constant:

```
// in CurveLsdAmo.sol
IConvexFxsBooster private convexFXSBooster = IConvexFxsBooster(0
    x2B8b301B90Eb8801f1eEFe73285Eec117D2fFC95);
```

```
// in Constants.sol
address internal constant CONVEX_FXS_BOOSTER = 0x2B8b301B90Eb8801f1eEFe73285Eec117D2fFC95;
```

This was the Convex Booster address until April 27th. At that point, it was moved over to the address: `0xD8Bd5Cdd145ed2197CB16ddB172DF954e3F28659`.

At that time, the `PoolRegistry` updated its operator to the new Booster. As a result, if the old Booster is called with `createVault()`, it calls out to `poolRegistry.addUserVault()`, which reverts because it is no longer the current operator address.

Recommendation: Update both constants to the correct current address.

Frax: Fixed in commit 3a746c0.

Frax Security Cartel: Verified.

3.3.9 `whitelistedExecute` transactions can fail without reverting

Description: The `whitelistExecute()` function is intended to be called by the Timelock to execute arbitrary code on behalf of the AMO. For this to happen, Frax governance would have to vote for the proposal, the vote would pass, and then `executeTransaction()` would be called on the Timelock.

Typically, in this situation, if the transaction were to revert, `executeTransaction()` could be called again to retry. For example, if a transaction was approved by governance to spend certain tokens, but they hadn't been sent to the Timelock yet, attempting to perform this transfer would fail and could be tried again when they'd been received.

However, in this case of the `whitelistExecute()` function, calls do not revert, but instead return `bool, bytes`, with the `bool` representing the success of the transaction.

On the Timelock, the following logic is implemented:

```
// Execute the call
(bool success, bytes memory returnData) = target.call{ value: value }(callData);
require(success, "Timelock::executeTransaction: Transaction execution reverted.");
```

It appears that the returned `bool, bytes` matches the return values, but this isn't the case. The `bool success` value is injected by the EVM to represent whether the call itself succeeded. Both of the return values are encoded together and become `returnData`.

The result is that a failed transaction will not revert, and will instead simply return `abi.encode(false, returnData)`.

At the moment, the Frax Timelock is permissioned so that `executeTransaction()` can only be called by admins. However, if this ever became unpermissioned, it would be possible to use up passed proposal with a failed transaction and require governance to vote again.

Recommendation:

```
  function whitelistedExecute(
      address _to,
      uint256 _value,
      bytes calldata _data
- ) external returns (bool, bytes memory) {
+ ) external returns (bytes memory) {
      _requireSenderIsTimelock();

-     return _whitelistedExecute(_to, _value, _data);
+     (bool success, bytes returnData) =  _whitelistedExecute(_to, _value, _data);
+     require(success, "whitelisted transaction failed");

+     return returnData
```

```
}
```

Frax: Fixed in commit dd92a59.

Frax Security Cartel: Verified.

3.3.10 AMOs holding `stETH` will attempt to perform unnecessary LP withdrawals, which could fail

Description: When ETH is withdrawn from the AMO to the router, we withdraw in the following order: (1) withdraw native ETH, (2) scrounge WETH / stETH, (3) withdraw from CVX vault and convert to ETH.

In the case that the WETH / stETH is sufficient to cover the request, we get to skip the (gas heavy) step of withdrawing from the vault.

However, in the case that it is stETH being scrounged, the amount we get out can be slightly less than the amount requested. This is accounted for properly, but leaves a dust amount of `_remainingEth` to claim, which triggers the withdrawal logic.

- In the case that the additional assets are held in the Convex vault, this will waste gas but work as intended.
- If the assets are held in the stkCvx vault (which requires manual withdrawal), this will lead to insufficient funds being available. If this is the only AMO, it will cause the request to fail.

This leads to a situation where a vault that has (a) X stETH and (b) the rest of its funds in stkCvx vault will not be able to fulfill requests, even if they are for substantially less than X.

Recommendation: Scrounging stETH for ETH should increase the amount requested using `swapSlippage` to ensure that the amount of ETH that is claimed is at least the amount requested.

Frax: Fixed in commit d7c8c89, commit e0a9ecb and commit 8440928.

Frax Security Cartel: Verified.

3.3.11 Slippage protection on `TWOLSDSTABLE` or `LSDWETH` swap is ineffective

Description: When `poolSwap()` is called on a pool with `TWOLSDSTABLE` or `LSDWETH`, slippage protection is implemented as follows:

```
_minOutToUse = _pool.get_dy(_inIndex128, _outIndex128, _inAmount);
_minOutToUse = (_minOutToUse * (1e6 - swapSlippageE6)) / 1e6;
```

Using the `get_dy()` function gets the current exchange rate from the contract, which would already be manipulated in the case of a manipulation, so this does not provide any protection.

Fortunately, this value is later overridden with the `_minOutFromUser`, so as long as the user is careful in their settings, the harm is minimal.

Recommendation: Either make the same 1:1 assumption as is done in `LSDETH` pools, or ignore this check altogether and only use the user's passed value.

Frax: Fixed in commit a4e24b8.

Frax Security Cartel: Verified.

3.3.12 `convertEthToStEth()` does not set its return value

Description: The `convertEthToStEth()` function defines a named `uint256 _stEthOutActual` return value, but the code never assigns this variable. Fortunately, this return value is not used anywhere explicitly, but this could lead to problems in downstream code or problems in future versions of the `CurveLsdAmo` contract itself.

Recommendation: Update the `convertEthToStEth()` implementation to return the actual amount of stETH that resulted from the conversion. This can be accomplished with the following changes:

```
/// @notice Converts ETH to stETH
/// @param _ethIn Amount of ETH in
/// @return _stEthOutActual Actual amount of output stETH
function convertEthToStEth(uint256 _ethIn) public payable returns (uint256 _stEthOutActual) {
    _requireIsTimelockOperatorOrEthRouter();

    // ETH -> stETH
    // ====================================================
    uint256 _stETHBalBefore = stETH.balanceOf(address(this));
    {
        stETH.submit{ value: _ethIn }(address(this));
    }
    uint256 _stETHBalAfter = stETH.balanceOf(address(this));

    // Check slippage, should be 1:1 but sometimes it can be off by a few wei
    uint256 _absoluteMinOut = ((_ethIn * (1e6 - swapSlippageE6)) / 1e6);
+   _stEthOutActual = _stETHBalAfter - _stETHBalBefore;
+   if (_stEthOutActual < _absoluteMinOut) {
-   if ((_stETHBalAfter - _stETHBalBefore) < _absoluteMinOut) {
        revert EthLsdConversionSlippage(_stETHBalAfter - _stETHBalBefore, _absoluteMinOut);
    }
}
```

Frax: Fixed in commit ed1de0b.

Frax Security Cartel: Verified.

3.3.13 `hasCvxVault` and `hasStkCvxFxsVault` will always be set to `true`

Description: The Curve AMO logic is set up so that deposits into the various pools are only allowed if the pools are set up.

```
function _requireHasCVXVault() internal view {
    if (!poolInfo.hasCvxVault) revert PoolNoCVXVault();
}
```

However, both `hasCvxVault` and `hasStkCvxFxsVault` are always set to `true` in the constructor, and can never be reset to `false`.

Recommendation: If the contract should support pools without these vaults, update the constructor logic so that, in the event that `address(0)` is passed, they are not set to `true`.

Otherwise, all this logic can be removed.

Frax: Fixed in commit 0358bf2. There will only be a handful of attached AMOs, and this updated logic should be sufficient for these.

Frax Security Cartel: Verified.

3.4 Informational Findings

3.4.1 `depositEther` checks sender twice

Description: The `depositEther()` function is only callable by the ether router contract and this is checked both in the external and internal facing functions.

Recommendation: Be consistent on checking sender in the most external facing functions. All duplicate instances where `msg.sender` is checked unnecessarily can also be removed as there are likely other instances of this behavior.

Frax: Fixed in commit a458e1e.

Frax Security Cartel: Verified.

3.4.2 Inaccurate naming of `requestEtherByOperator`

Description: The `requestEtherByOperator()` function allows the caller to pull out ETH, redeeming LP tokens if necessary to fulfil the request. This function is callable by both the AMO operator and the timelock address, however, the naming seems to indicate only the operator can call this.

Recommendation: Update the function name to `requestEtherByTimelockOrOperator()` instead.

Frax: Fixed in commit 8f87ba8.

Frax Security Cartel: Verified.

3.4.3 AMO operator is a trusted role

Description: The operator of the AMO is able to perform the same set of actions that the timelock is able to except for a small set of governance functions which are used to maintain a list of executable targets, set new operator addresses, modify Curve pool accounting and set default slippages.

However, it is important to note that in no way is the operator limited from leaking funds in a few different ways. Even with slippage tightly bound, the operator can perform a large number of swaps to drain the protocol by taking advantage of `swapSlippageE6`. Additionally, they can withdraw LP tokens to themselves or lock Curve LP tokens to a `_kekId` that they control.

Recommendation: Ensure it is well-documented that this operator is entirely trusted at the center of the `frxETH` protocol. It may be worth thinking of ways to restrict the rate of value extraction in case this is ever compromised.

3.4.4 No mechanism to change `requestEther()` withdrawal order

Description: The `sweepEther()` function in the `EtherRouter` contract sends ETH to the AMO specified by `depositToAmoAddr`. On the other hand, the `requestEther()` function withdraws ETH by iterating through the `amosArray` in order until sufficient funds are collected.

As a result of this behavior, if `depositToAmoAddr` is not near the start of the `amosArray`, AMOs earlier in the array will gradually lose TVL while the `depositToAmoAddr` will gradually gain TVL. While the AMO operators can manually correct this, it may be desirable to implement a system that supports a different automatic withdrawal order.

Recommendation: For more flexibility in how the `EtherRouter` automatically withdraws from AMOs, consider implementing a mechanism that can change the order of AMOs used in `requestEther()`.

Frax: Fixed in commit 344a0ba. There is now a `primaryWithdrawFromAmoAddr` that will always be used first in withdrawals.

Frax Security Cartel: Verified.

3.4.5 `stETH` pricing inaccuracy in pool allocations

Description: The `showAllocationsWithParams()` function should accurately reflect the correct amount of free ETH. While converting `WETH -> ETH` is 1:1, the conversion of `stETH -> ETH` is not as it relies on the the best route available in `CurveLsdAmo.convertStEthToEth()`.

Recommendation: A function similar to `CurveLsdAmo.convertStEthToEth()` could be used to read the expected converted amount, however, this might be prone to some manipulation and the extent of this issue is only some minor inaccuracy in the utilization ratio which has other problems.

Frax: I see what you are saying, but it might be best to just leave this alone.

Frax Security Team: Acknowledged as a won't-fix.

3.4.6 `setPoolAllocation()` and `setPoolManualLPTrans()` can check `onBudget()` modifier

Description: The AMO's `onBudget()` modifier is currently only used to check that `poolInfo.tokenDeposited[i] <= poolInfo.tokenMaxAllocation[i]` after each call to `_depositToCurveLP()`. Since `setPoolAllocation()` and `setPoolManualLPTrans()` are whitelisted functions that can manually change the `poolInfo.tokenDeposited` and `poolInfo.tokenMaxAllocation` values, it would be useful if these functions used the `onBudget()` modifier as well.

Recommendation: Consider adding the `onBudget()` modifier to both `setPoolAllocation()` and `setPoolManualLPTrans()`.

Frax: Addressed in commit fec7ad7.

Frax Security Cartel: Verified. There has been a refactor and `tokenDeposited` no longer exists.

3.4.7 Usage of `ethType` and `frxEthType` can be more explicit

Description: Each `CurveLsdAmo` has a specific `ethType` and `frxEthType` defined in its `poolInfo` storage variable. The possible values for these types are:

```
enum FrxSfrxType {
    NONE, // neither frxETH or sfrxETH
    FRXETH, // frxETH
    SFRXETH // sfrxETH
}

enum EthType {
    NONE, // ankrETH/frxETH
    RAWETH, // frxETH/ETH
    STETH, // frxETH/stETH
    WETH // frxETH/WETH
}
```

These values determine the types of conversions that are required in deposits/withdrawals, and they are also utilized in the `showAllocationsWithParams()` accounting.

In the current code, there are several ways for the timelock/operator to convert ETH in ways that do not match these values. Specifically, the `wrapEthToWeth()`, `unwrapWethToEth()`, `convertEthToStEth()`, `convertStEthToEth()`, and `exchangeFrxEthSfrxEth()` functions can be called on any pool, regardless of their `ethType` and `frxEthType` values. Since this would lead to unexpected behavior in the downstream `showAllocationsWithParams()` accounting, this is a potential footgun that can be prevented.

Recommendation: Consider making the following restrictions:

- Require that `poolInfo.ethType == EthType.WETH` in `wrapEthToWeth()` and `unwrapWethToEth()`
- Require that `poolInfo.ethType == EthType.STETH` in `convertEthToStEth()` and `convertStEthToEth()`
- Require that `poolInfo.frxEthType == FrxSfrxType.SFRXETH` in `exchangeFrxEthSfrxEth()`

Frax: Fixed in commit a5142ac.

Frax Security Cartel: Verified.

3.4.8 Nits

Description: The following minor issues have been noted within the codebase:

- The `stETH/ETH` and `stETH/ETH NG` pools can be saved as constants instead of hardcoded into the code.
- The `priceFeedfrxEthUsd` is temporarily set to the ETH chainlink feed, this will require an update in the future.

Recommendation: Consider addressing/documenting each of these small issues as described above.

Frax: The issues have been addressed as follows:

- The `stETH/ETH` and `stETH/ETH NG` pools have remained as is, this may change in the future if/when there are multiple Curve AMO variants.
- The `priceFeedfrxETHUsd` was changed to the `SfrxEthUsdDualOracle` in commit 8f87ba8.

Frax Security Cartel: Verified.

3.5 Gas Optimizations

3.5.1 Pool address storage variables can be simplified

Description: The `CurveLsdAmo` currently stores the underlying Curve pool's address in three different storage values: `pool`, `poolAddress`, and `poolInfo.poolAddress`.

Recommendation: Consider simplifying this storage layout, so that the AMO only uses one storage slot for the underlying Curve pool's address.

Frax: Fixed in commit 7718ddc. The `pool` and `poolAddress` variables will remain for code legibility, and `poolInfo.poolAddress` has been removed.

Frax Security Cartel: Verified.

3.5.2 Incorrect check in `_requestEther` leads to unnecessary computation

Description: In `_requestEther()`, if there isn't sufficient ETH or WETH/stETH to cover the request, we withdraw from the vault.

First, we calculate `_lpNeeded` based on the pool's state and our request, and fetch `_lpAvailable` as our balance in the Convex rewards contract available for withdrawal.

Next, we set `_lpNeeded` to the lower of these two values.

```
if (_lpAvailable < _lpNeeded) _lpNeeded = _lpAvailable;
```

Finally, we perform the withdrawal if `_lpAvailable > 0`:

```
if (_lpAvailable > 0) {
    // Do the unwrap (vaulted cvxLP -> coins). Skip reward claim to save gas
    _withdrawAndUnwrapVaultedCvxLP(_lpNeeded, false);

    ...
}
```

This check should use `_lpNeeded`, not `_lpAvailable`.

Since `_lpNeeded = min(_lpNeeded, _lpAvailable)`, it cannot be higher than it, and therefore cannot cause problems with requesting more than is available.

However, in the case where `_lpNeeded == 0`, but `_lpAvailable > 0`, we will run the withdrawal process with `0` as the value and waste gas.

Recommendation: Replace `_lpAvailable` with `_lpNeeded` in this check.

Frax: Fixed in commit af3793a.

Frax Security Cartel: Verified.

3.5.3 `poolInfo` can be cached in `onBudget` modifier to save gas

Description: In the `onBudget` modifier, we iterate over the tokens and check `tokenDeposited` and `tokenMaxAllocation` for each of them. This requires multiple SLOADs to access the struct.

```
modifier onBudget() {
    _;
    for (uint256 i = 0; i < 2; ) {
        if (poolInfo.tokenDeposited[i] > poolInfo.tokenMaxAllocation[i]) {
            revert OverTokenBudget();
        }
        unchecked {
            ++i;
        }
    }
}
```

This can be optimized by caching the `poolInfo` struct and checking the values directly from memory, which saves 2268 gas.

Recommendation:

```
PoolInfo memory p = poolInfo;
if (p.tokenDeposited[0] > p.tokenMaxAllocation[0] || p.tokenDeposited[1] > p.tokenMaxAllocation[1])
     {
    revert OverTokenBudget();
}
```

Frax: Initially fixed in commit b0ab36d. Later changed with commit fec7ad7 to fix a separate issue.

Frax Security Cartel: Verified, recommendation is no longer applicable.