Frax Finance Audit Report

(FPISLocker + MintRedeemer)

Frax Security Cartel

0xleastwood, Riley Holterhus, Zach Obront

October 24, 2024

Contents

1 Introduction **3**

1.1 About Frax Finance . . . 3
1.2 About the Auditors . . . 3
1.3 Disclaimer . . . 3

2 Audit Overview **4**

2.1 Scope of Work . . . 4
2.2 Summary of Findings . . . 4

3 Findings **5**

3.1 High Severity Findings . . . 5
3.1.1 Incorrect bias decrement in `_checkpoint()` . . . 5
3.1.2 `_withdrawActiveLock()` leads to inconsistent `_checkpoint()` accounting . . . 5
3.1.3 Attacker can sandwich oracle updates to extract funds from MintRedeemer . . . 7
3.1.4 Deployment script sets incorrect values for MintRedeemer . . . 10
3.1.5 Supply decay uses incorrect floor, resulting in inflated values . . . 11
3.2 Medium Severity Findings . . . 12
3.2.1 sFRAX oracle should include freshness check . . . 12
3.2.2 Hardcoded FPIS to FXS conversion rate creates problematic market dynamics . . . 12
3.2.3 `maxDeposit()` and `maxMint()` should be independent of user balance . . . 13
3.2.4 Balances will be inflated in the case of emergency unlock . . . 14
3.2.5 veFPIS to veFXS conversion into existing lock always returns index of 0 . . . 15
3.3 Low Severity Findings . . . 16
3.3.1 User error removes zeroth lock index instead of reverting . . . 16
3.3.2 Redeemer implementation can be initialized by anyone . . . 17
3.4 Informational Findings . . . 17
3.4.1 Expected lFPIS calculation can be more precise . . . 17
3.4.2 `bulkConvertToFXSAndLockInVeFXS()` can be refactored . . . 18
3.4.3 Minor Nits . . . 20

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 15, 2024 through May 22, 2024, the Frax Security Cartel conducted an audit on two new components of the Frax codebase:

FPISLocker - A voting escrow contract for FPIS with altered decay variables and an ability to later convert to FXS locks. The scope of this review was in Frax's internal dev-fraxchain-contracts GitHub repository on commit hash cc0cde2b75274cdae09085b08284434d20401909, specifically on the following files:

- contracts/VestedFXS-and-Flox/FPISLocker/FPISLocker.sol
- contracts/VestedFXS-and-Flox/FPISLocker/FPISLockerUtils.sol
- contracts/VestedFXS-and-Flox/FPISLocker/IlFPISEvents.sol
- contracts/VestedFXS-and-Flox/FPISLocker/IlFPISStructs.sol

FraxtalERC4626MintRedeemer - An ERC4626 mint/redeem vault for sFRAX on the Fraxtal L2 chain. The scope of this review was in Frax's internal dev-fraxchain-contracts GitHub repository on commit hash cc0cde2b75274cdae09085b08284434d20401909, specifically on the following file:

- contracts/Miscellany/FraxtalERC4626MintRedeemer.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, **15 findings** were identified. This includes **5 high**, **5 medium**, **2 low**, and **3 informational** severity 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 Incorrect bias decrement in `_checkpoint()`

Description: The `FPISLocker` and `VestedFXS` contracts differ from typical `VotingEscrow` implementations in how they manage balances. Instead of decaying multiplied balances to zero over time, the `FPISLocker` reduces the balance to 33.3% (0.333x) of the underlying amount, while the `VestedFXS` ultimately ends the balance at 100% (1x) of the underlying amount. To remove this base bias during a withdrawal, the following code is used in both contracts:

```
if (_newLocked.amount == 0 && _oldLocked.end <= block.timestamp) {
    lastPoint.bias -= _oldLocked.amount;
}
```

Notice that this code will always remove a 1x base bias. Since the `FPISLocker` has a different base bias of 0.333x, a withdrawal will lead to a global subtraction that is too large. This will permanently affect the contract's internal accounting, and can lead to incorrect results in any downstream code that relies on functions like `totalSupply()` or `supplyAt()`.

Note that in this code, the `_oldLocked.end <= block.timestamp` check exists to prevent the base bias decrement in an early withdrawal. The decrement is not needed in this case, because an earlier part of the code (specifically: `lastPoint.bias += (uNew.bias - uOld.bias)`) would handle the bias difference itself.

Recommendation: Change the `FPISLocker` implementation to remove a 0.333x base bias instead of a 1x base bias, which can be accomplished as follows:

```
if (_newLocked.amount == 0 && _oldLocked.end <= block.timestamp) {
    lastPoint.bias -= (_oldLocked.amount * VOTE_END_POWER_BASIS_POINTS_INT128) /
        MAX_BASIS_POINTS_INT128;
}
```

Frax: Fixed in commit c42d8ff.

Frax Security Cartel: Verified.

3.1.2 `_withdrawActiveLock()` leads to inconsistent `_checkpoint()` accounting

Description: In order to withdraw from a lock before it has expired, the `_withdrawActiveLock()` function overwrites the lock's `end` timestamp to `block.timestamp` before calling `_withdraw()`:

```
function _withdrawActiveLock(address _addr, address _recipient, uint128 _lockIndex) internal {
    // Get the lock ID
    uint256 lockId = indicesToIds[msg.sender][_lockIndex];

    // Set the lock end time to now
    locked[msg.sender][lockId].end = uint128(block.timestamp); // Downcasting uint256 to uint128
        should be safe in this case, as the lock end times are always in uint128

    // Withdraw the lock
    _withdraw(_addr, _recipient, _lockIndex);
}
```

While this does succesfully bypass the `LockDidNotExpire()` error in the `_withdraw()` function, this can also lead to incorrect behavior in the internal `_checkpoint()` accounting. This is because the `_checkpoint()` function will consider `_oldLocked.end == block.timestamp`, which means the accounting will be updated as if the lock is actually expired. This leads to the following outcomes that are not fully consistent:

- The lock's current effect on the global bias/slope would remain in the `pointHistory` logic, with the exception of the base 0.333x bias, which would be instantly removed (since `_newLocked.amount == 0 && _oldLocked.end <= block.timestamp`).
- The lock's future effect on the global slope would remain queued in the `slopeChanges` mapping.
- The lock's `amount` would be instantly removed from `lastPoint.fpisAmt`.

Recommendation: Consider changing the logic so that `_withdrawActiveLock()` has the same effect as an early withdrawal due to an `emergencyUnlockActive` scenario. This could be accomplished by adding a new boolean to `_withdraw()` that allows bypassing the expiration check:

```
  function _withdraw(
      address _staker,
      address _recipient,
+     bool skipExpirationCheck,
      uint128 _lockIndex
  ) internal nonReentrant returns (uint256 _value) {
      // Revert if it would be an array out-of-bounds
      if (_lockIndex >= numLocks[_staker]) revert InvalidLockIndex();

      // Revert if paused
      if (isPaused) revert OperationIsPaused();

      // Get old lock information
      uint256 lockId = indicesToIds[_staker][_lockIndex];
      LockedBalance memory oldLocked = locked[_staker][lockId];

      // Revert if the lock is not expired yet, unless you are in an emergency unlock
-     if ((uint128(block.timestamp) < oldLocked.end) && !emergencyUnlockActive) {
+     if ((uint128(block.timestamp) < oldLocked.end) && !emergencyUnlockActive &&
    !skipExpirationCheck) {
          revert LockDidNotExpire();
      }
```

```
        // ...
    }
```

With this change, all the existing calls to `_withdraw()` could set `skipExpirationCheck == false`, and all calls to `_withdrawActiveLock()` could be replaced with a call to `_withdraw()` with `skipExpirationCheck == true`.

Frax: Fixed in commit 03fac0f.

Frax Security Cartel: Verified.

3.1.3 Attacker can sandwich oracle updates to extract funds from MintRedeemer

Description: The MintRedeemer uses the FraxtalERC4626TransportOracle to determine the relative value of `FRAX` vs `sFRAX`. This oracle's prices are derived from it's internal accounting of `totalAssets` vs `totalSupply`, which are updated by the following function:

```
function updatesFRAXData(
    uint96 _l1BlockNumber,
    uint256 _totalSupply,
    uint256 _totalAssets,
    uint256 _lastRewardsDistribution,
    RewardsCycleData memory _data
) external {
    if (msg.sender != priceSource) revert OnlyPriceSource();
    if (_l1BlockNumber < lastL1Block) revert StalePush();
    rewardsCycleData = _data;
    lastRewardsDistribution = _lastRewardsDistribution;
    totalSupply = _totalSupply;
    storedTotalAssets = _totalAssets;
    emit VaultDataUpdated(totalSupply, storedTotalAssets, lastRewardsDistribution, rewardsCycleData
        );
}
```

If we look at the price source contract that performs this update, we can see that it calls `updatesFRAXData()` when any user permissionlessly calls `addRoundDataSfrax()`, proving valid values against the L1 state.

```
function addRoundDataSfrax(
    IERC4626Receiver _sFraxOracle,
    uint96 _blockNumber,
    PoofPackedsFrax memory proof
) external {
    uint96 lastBlockProofed = oracleLookup[address(_sFraxOracle)].lastBlockProofed;
    address _proofAddress;
    {
        if (lastBlockProofed != 0) {
            if (_blockNumber < lastBlockProofed) revert StalePush();
        }
        // Address of the L1 oracle
```

```
            _proofAddress = oracleLookup[address(_sFraxOracle)].layer1Oracle;
            if (_proofAddress == address(0)) revert WrongOracleAddress();
        }
        (
            uint256 totalSupply,
            uint256 totalStoredAssets,
            uint256 lastRewardsDistribution,
            IERC4626Receiver.RewardsCycleData memory data
        ) = _fetchAndProofSfrax(_proofAddress, _blockNumber, proof);
        _sFraxOracle.updatesFRAXData(_blockNumber, totalSupply, totalStoredAssets,
            lastRewardsDistribution, data);
        oracleLookup[address(_sFraxOracle)].lastBlockProofed = _blockNumber;
    }
```

While this appears to accurately confirm the increased value of sFRAX on L1, it does allow any user to permissionlessly determine when the price is updated.

Attack Analysis: This creates an opportunity for any user to steal a percentage of the FRAX tokens held in the MintRedeemer proportionate to the percentage increase in the value of sFRAX during a given update.

In most cases, these updates will be small. Rewards are streamed in a way that mirrors L1 streaming, and should minimize discrete jumps in the relative price of sFRAX. However, there are certain situations where a discrete jump could occur:

1) If Fraxtal (or the monitoring bot) were to temporarily go down over the border between cycles, it would leave a period where a new cycle has started on L1 but is not being reflected on L2. The first update when Fraxtal came back online would increase the `pricePerShare()` by the full amount of rewards in the cycle so far.
2) More commonly (but on a smaller scale), the first update in a new cycle will always cause some discrete jump in value, because it will accrue the rewards for the full period of the cycle so far.
3) Without time on this audit to get deeply into the details of L1 sFRAX, I'm not sure if there are other possibilities where L1 behavior could cause a discrete jump on L2.

With these situations in mind, the attack works as follows: - A user waits for one of these situations to occur. - The user flashloans and purchases a large amount of FRAX. - They deposit all their FRAX into the vault, withdrawing the vault's full holdings of sFRAX at the old exchange rate. - They permissionlessly perform the update to increase the price per share. - After the price is updated, they sell all the sFRAX back into the vault and pocket the profit.

Proof of Concept: First, add the following interfaces to the`Unit_Test_sFRAXRedeemer.t.sol` file:

```
interface IERC20 {
    function balanceOf(address account) external view returns (uint256);
    function approve(address spender, uint256 amount) external returns (bool);
    function transfer(address recipient, uint256 amount) external returns (bool);
}

interface FraxtalERC4626TransportOracle {
    struct RewardsCycleData {
        uint40 cycleEnd; // Timestamp of the end of the current rewards cycle
```

```
        uint40 lastSync; // Timestamp of the last time the rewards cycle was synced
        uint216 rewardCycleAmount; // Amount of rewards to be distributed in the current cycle
    }

    function updatesFRAXData(
        uint96 _l1BlockNumber,
        uint256 _totalSupply,
        uint256 _totalAssets,
        uint256 _lastRewardsDistribution,
        RewardsCycleData memory _data
    ) external;

    function priceSource() external view returns (address);
    function totalAssets() external view returns (uint256);
    function totalSupply() external view returns (uint256);
}
```

Then add this test to demonstrate the issue:

```
function testZach_ExploitMintRedeemer() public {
    vm.createSelectFork("https://rpc.frax.com/", 4578153);
    FraxtalERC4626TransportOracle oracle = FraxtalERC4626TransportOracle(0
        xF750636E1df115e3B334eD06E5b45c375107FC60);
    address FRAX = 0xFc0000000000000000000000000000000000001;
    address sFRAX = 0xfc0000000000000000000000000000000000008;

    // deploy and initialize a new MintRedeemer, and stock it with 10mm FRAX and sFRAX
    FraxtalERC4626MintRedeemer redeemer = new FraxtalERC4626MintRedeemer();
    redeemer.initialize({
        _owner: address(this),
        _underlyingTkn: FRAX,
        _vaultTkn: sFRAX,
        _underlyingOracle: address(0),
        _vaultOracle: address(oracle)
    });
    deal(FRAX, address(redeemer), 10_000_000e18);
    deal(sFRAX, address(redeemer), 10_000_000e18);

    // buy max sFRAX at lower rate before oracle update
    uint amountToDeposit = redeemer.convertToAssets(IERC20(sFRAX).balanceOf(address(redeemer)));
    deal(FRAX, address(this), amountToDeposit); // simulate flash loan
    IERC20(FRAX).approve(address(redeemer), amountToDeposit);
    redeemer.deposit(amountToDeposit, address(this));

    // update the oracle, simulating with an increase of 1%
    // this is possible because oracle updates are permissionless
    vm.startPrank(oracle.priceSource());
    FraxtalERC4626TransportOracle.RewardsCycleData memory cycleData = FraxtalERC4626TransportOracle
        .RewardsCycleData(1, 0, 0);
    oracle.updatesFRAXData({
        _l1BlockNumber: 0,
        _totalSupply: oracle.totalSupply(),
        _totalAssets: oracle.totalAssets() * 101 / 100,
        _lastRewardsDistribution: 0,
        _data: cycleData
```

```
        });
        vm.stopPrank();

        // we can now trade all our sFRAX back for FRAX and return the flash loan
        uint sFRAXBal = IERC20(sFRAX).balanceOf(address(this));
        IERC20(sFRAX).approve(address(redeemer), sFRAXBal);
        uint amountOut = redeemer.redeem(sFRAXBal, address(this), address(this));

        // simulate returning the flashloan
        IERC20(FRAX).transfer(address(redeemer), amountToDeposit);

        // profit of ~$105k
        assertGt(IERC20(FRAX).balanceOf(address(this)), 100_000e18);
    }
```

The result of a 1% increase in the value of sFRAX is that the user was able to steal ~1% of the holdings of the vault.

Recommendation: There are a number of possible solutions that completely solve this issue: - Make oracle upgrades permissioned. Since there is no mempool on the L2, the attacker will be unlikely to be able to sandwich the update. - Impose a minimum waiting time between minting and redeeming to avoid the ability to use flash loans.

There are also simpler solutions that might minimize the risk without completely eliminating it: - Upgrade the bot providing these proofs to check if `l1 cycle end != l2 cycle end` and always push an upgrade in these cases. - Run multiple bots to provide redundancy in case one of them temporarily goes down.

Frax: Initial changes were made in commit e9ce1e3, commit 61078ef, and commit 69a6669, including a 5-minute ramp-up period. After further discussion, commit d5b9188 was made. This removed the 5-minute ramp-up period, added a fee to the conversion rates, and added a cached `vaultTknPrice` that is only updated once per timestamp.

Also, a new version of `priceFeedVault` has been deployed at `0x1B680F4385f24420D264D78cab7C58365ED3F1FF`. With this new price feed, the oracle price will be considered stale when the last known cycle ends, or when the timestamp of the last known L1 state is too old (determined by `oracleTimeTolerance`, which is initially 6 hours).

Frax Security Cartel: Verified. The fee and staleness checks will make short-term sandwiching impossible or unprofitable, and the other measures taken will help eliminate large discrete jumps from happening in the first place.

3.1.4 Deployment script sets incorrect values for MintRedeemer

Description: In `DeploySfraxMintRedeemer.s.sol`, the wrong values are used for production deployment.

```
if (false) {
    // Prod deploy
    frax = Constants.FraxtalStandardProxies.FPIS_PROXY;
    sfrax = Constants.FraxtalStandardProxies.FXS_PROXY;
    fraxOracle = address(0);
    sfraxOracle = 0xa41107f9259bB835275eaCaAd8048307B80D7c00;
    eventualAdmin = 0xC4EB45d80DC1F079045E75D5d55de8eD1c1090E6;
```

```
    ...
}
```

There are a few issues with this:

1) `frax` is set to the address of `FPIS`, which sets `FPIS` as the underlying token.
2) `sfrax` is set to the address of `FXS`, which sets `FXS` as the vault token.
3) sfraxOracle is set to `0xa41107f9259bB835275eaCaAd8048307B80D7c00`, while the intended oracle for the price of sfrax is `0xf750636e1df115e3b334ed06e5b45c375107fc60`.

Recommendation: Update the deployment script to use the correct values.

Frax: Fixed in commit a437ff5.

Frax Security Cartel: Verified.

3.1.5 Supply decay uses incorrect floor, resulting in inflated values

Description: The FPIS Locker is a fork of `VestedFXS`, with the key difference being that the voting power decays from 1.33x to 0.33x rather than 4x to 1x.

All functions that calculate individual user balances handle this difference properly, by setting the minimum value of a balance to `upoint.fpisAmt * VOTE_END_POWER_BASIS_POINTS_UINT256) / MAX_BASIS_POINTS_UINT256`, which is equivalent to `upoint.fpisAmt * 0.33`.

However, when the total FPIS supply is calculated, this adjustment is not performed. As a result, the functions continue to implement a floor of 1x rather than 0.33x.

This issue appears in a number of places:

1) In the `supplyAt()` function, the final value is explicitly set to a minimum value of `lastPoint.fpisAmt`:

```
uint256 weightedSupply = uint256(uint128(lastPoint.bias));
if (weightedSupply < lastPoint.fpisAmt) {
    weightedSupply = lastPoint.fpisAmt;
}
```

2) In `totalSupply()`, in the case that `emergencyUnlockActive`, we return `totalFPISSupply()`, which is just the total balance of `FPIS`. This function is intended to return the total voting power, so it should be adjusted in this situation to return `totalFPISSupply() * 0.33` because all voting power will be reduced to 0.33x in this situation.
3) In `supplyAt()`, in the case that `emergencyUnlockActive`, the same as true as above.

Recommendation: Ensure all supply functions that will be used in voting return the correct total voting power by using the correct floor of `fpis balance * 0.33`.

Frax: Fixed in commit 03fac0f.

Frax Security Cartel: Verified.

3.2 Medium Severity Findings

3.2.1 sFRAX oracle should include freshness check

Description: When the MintRedeemer gets the latest price from the `priceFeedVault`, it only validates the response with the following check:

```
require(price >= 0 && updatedAt != 0 && answeredInRound >= roundID, "Invalid oracle price");
```

Since `updatedAt = block.timestamp` and `answeredInRound = roundID = 0`, this really only checks that `price >= 0`.

While in most cases the L2 contract will stream rewards at the same pace as the L1 contract, there are exceptions some situations, for example if Fraxtal was down over the border between cycles, or if the bot handling the oracle proofs was temporarily down.

It would increase safety to perform a check that the oracle has been updated within some reasonable threshold of time.

Recommendation: On the MintRedeem contract, add a check that `updatedAt` returns a value within a reasonable time threshold (say, 1 hour).

Additionally, the `transportSFraxFraxOracle` that is being queried needs to be updated so that `updatedAt` returns a useful value. This can be accomplished by adding the `block.timestamp` of the proof to the oracle, or using some other data to approximate freshness. It would also be useful if the `transportSFraxFraxOracle` was updated so the `roundID` and `answeredInRound` logic returned non-zero values depending on the oracle update.

Frax: Fixed in commit e9ce1e3 and commit 61078ef. Also, a new `priceFeedVault` was deployed at address `0x1B680F4385f24420D264D78cab7C58365ED3F1FF`. With these changes, the oracle price will be considered stale when the last known cycle ends, or when the timestamp of the last known L1 state is too old (determined by `oracleTimeTolerance`, which is initially 6 hours).

Frax Security Cartel: Verified.

3.2.2 Hardcoded FPIS to FXS conversion rate creates problematic market dynamics

Description: The `FPISLocker` contract allows anyone with locked FPIS to convert it to locked FXS in 4 years. The conversion rate between the two tokens is hardcoded to 0.4 FXS per FPIS burned:

```
function convertFpisToFxs(uint256 _fpisAmount) internal pure returns (uint256 _fxsAmount) {
    return (_fpisAmount * MAX_BASIS_POINTS_UINT256) / FPS_TO_FXS_CONVERSION_CONSTANT_BASIS_POINTS;
}
```

While this exchange rate was approximately right at the time the contracts were written, there's no reason for the exchange rate between these two tokens to stay consistent.

The simplest issue this causes is that the exchange rate is likely to be different in 2028 when conversions are enabled. At this point, if the exchange rate is greater than 2.5:1, users are incentivized to buy and lock as much FPIS as possible in order to swap it for FXS at a preferable rate.

There are also more complex market dynamics that this could create. Having `FXS Price / 2.5` as a floor value for FPIS opens the door to the possibility of a more complex attack that could involve either using FPIS governance or pushing for an FPI depeg in order to issue more FPIS to ramp up the arbitrage. It is unclear whether this is possible, but is a situation that would be better to protect against.

Recommendation: Allow the admin to set the conversion rate so that it is set appropriately closer to the conversion start time.

Frax: This is by design and governance vote.

Frax Security Cartel: Acknowledged.

3.2.3 `maxDeposit()` and `maxMint()` should be independent of user balance

Description: In the MintRedeemer's `maxDeposit()` and `maxMint()` functions, we return the minimum value of either:

1) The amount that could be deposited/minted to claim the entire `sFRAX`/`FRAX` balance of the vault.
2) The user's balance of `sFRAX`/`FRAX` they have available to deposit/mint.

ERC4626 specifies that the `maxDeposit()` and `maxMint()` functions should not take the user's balance into account. From the spec:

> This assumes that the user has infinite assets, i.e. MUST NOT rely on `balanceOf` of `asset`.

While it is understood that this contract is not exactly conforming to the ERC4626 spec, it would be more helpful for the `maxDeposit()` and `maxMint()` functions to return the maximum amount that is allowed to be deposited/minted, not just the user's current balance, in case users want to use it to calculate the amount of assets/shares they can use.

Recommendation: Remove the `balanceOf()` logic that is present in `maxDeposit()` and `maxMint()`. For example:

```
function maxDeposit(address _addr) public view returns (uint256 _maxAssetsIn) {
    // See how much underlyingTkn you would need to exchange for 100% of the vaultTkn in the
    contract
    // TODO: Check rounding direction
```

```
-   uint256 _assetsNeededForAllShares = _convertToAssets(vaultTkn.balanceOf(address(this)),
    Math.Rounding.Down);
+   return _convertToAssets(vaultTkn.balanceOf(address(this)), Math.Rounding.Down);

-   // See how much assets the user has
-   uint256 _assetBalanceUser = underlyingTkn.balanceOf(address(_addr));

-   // Return the lesser of the two
-   _maxAssetsIn = (
-       (_assetsNeededForAllShares > _assetBalanceUser) ? _assetBalanceUser :
    _assetsNeededForAllShares
-   );
}
```

Frax: Fixed in commit aab4b58.

Frax Security Cartel: Verified.

3.2.4 Balances will be inflated in the case of emergency unlock

Description: When `emergencyUnlockActive` is set to true, all users are able to withdraw their locks, even in the event that they have not expired.

The accounting of `VestedFXS` is such that, in this situation, all user's voting power is dropped to the minimum (1x).

On the `FPISLocker`, this minimum should be represented as 0.33x, but it remains at 1x. As a result, many users will see their voting power increase when emergency unlock is activated. This gives locked FPIS a relative advantage in voting in an emergency situation, which could be abused by a large FPIS holder to unfairly sway governance.

This issue occurs in both `balanceOfOneLockAtTime()` and `balanceOfOneLockAtBlock()`, so should be fixed in both. Additionally, note that the same calculation exists in the utils contract in `getCrudeExpectedLFPISUser()` and `getCrudeExpectedLFPISMultiLock()`, so should be corrected there as well.

Proof of Concept: The following test (which can be dropped into `Unit_Test_FPISLocker.t.sol`) shows a user with a balance of ~33e18. After activating the emergency lock, the balance jumps up to 100e18.

```
function testZach_EmergencyInflateBalance() public {
    lockedFPISSetup();
    FPISLockerUtils lockerUtils = new FPISLockerUtils(address(lockedFPIS));

    vm.startPrank(bob);
    token.approve(address(lockedFPIS), 100e18);
    uint128 unlockTimestamp = uint128(block.timestamp + 2 weeks);
    lockedFPIS.createLock(bob, 100e18, unlockTimestamp);
    vm.stopPrank();

    assertEq(lockedFPIS.balanceOf(bob), 33870719970826908826);
```

```
        vm.prank(lockedFPIS.admin());
        lockedFPIS.activateEmergencyUnlock();

        assertEq(lockedFPIS.balanceOf(bob), 10000000000000000000);
    }
```

Recommendation: Change these calculations which occur in the situation when `emergencyUnlockActive` to calculate the minimum voting power (0.33x) vs the amount (1x).

Frax: Fixed in commit 03fac0f.

Frax Security Cartel: Verified.

3.2.5 veFPIS to veFXS conversion into existing lock always returns index of 0

Description: When locked FPIS is converted into veFXS using the `convertToFXSAndLockInVeFXS()`, the user can specify whether to use it to create a new lock or to merge it with an existing lock.

At the end of the function, the `_lockIndex` is returned, which represents the index of the lock in the veFXS contract that the funds are now a part of (whether new or existing). As described in the natspec:

> @return _lockIndex Index of the user's veFXS lock that received the migrated assets

However, in the case that the funds are merged into an existing lock, the `_lockIndex` is never set, so always returns `0`.

Proof of Concept: The following test can be added to `Unit_Test_FPISLocker.t.sol` to demonstrate:

```
function testZach_IncorrectLockIndex() public {
    lockedFPISSetup();
    fxs.mint(address(lockedFPIS), 100e18);
    fxs.mint(address(alice), 100e18);

    vm.startPrank(alice);

    uint128 unlockTime = uint128(lockedFPIS.FXS_CONVERSION_START_TIMESTAMP() + 100 days);
    vm.warp(unlockTime - 4 * 52 weeks);

    // first, create two veFXS locks
    fxs.approve(address(vestedFXS), 100e18);
    (uint128 vefxsIndex0,) = vestedFXS.createLock(alice, 50e18, unlockTime);
    (uint128 vefxsIndex1,) = vestedFXS.createLock(alice, 50e18, unlockTime);

    // then, create one FPIS lock
    token.approve(address(lockedFPIS), 100e18);
    (uint128 fpisIndex0,) = lockedFPIS.createLock(alice, 10e18, unlockTime);

    // skip until the time when conversion is allowed
    vm.warp(lockedFPIS.FXS_CONVERSION_START_TIMESTAMP());

    // move the fpis lock into the index 1 veFXS lock
```

```
        (uint128 vefxsLockIndex,) = lockedFPIS.convertToFXSAndLockInVeFXS(false, fpisIndex0,
            vefxsIndex1);

        // it returns an index of 0
        assertEq(vefxsLockIndex, 0);

        // but it's actually the correct index 1 lock that's incremented
        (int128 amount,) = vestedFXS.locked(alice, vestedFXS.indicesToIds(alice, vefxsIndex1));
        assertEq(uint128(amount), 50e18 + 10e18 * 4 / 10);
    }
```

Recommendation: In the `else` codepath, `_lockIndex` should be set to `_veFxsLockIndex`.

Frax: Fixed in commit 6c110a0.

Frax Security Cartel: Verified.

3.3 Low Severity Findings

3.3.1 User error removes zeroth lock index instead of reverting

Description: In the `bulkWithdrawLockAsFxs()` and `bulkConvertToFXSAndLockInVeFXS()` functions, users specify the lock indices they intend to interact with. The code converts these indices into a list of lock ids, which is useful because indices may be shuffled during the function execution. For example, consider the following code snippet:

```
function bulkWithdrawLockAsFxs(uint128[] memory _lockIndices) external {

    // ...

    uint256[] memory lockIds = new uint256[](_lockIndices.length);

    // Loop and get the lock IDs
    for (uint256 i; i < _lockIndices.length; ) {
        lockIds[i] = indicesToIds[msg.sender][_lockIndices[i]];

        unchecked {
            ++i;
        }
    }

    // Loop through and withdraw the locks
    for (uint256 i; i < _lockIndices.length; ) {
        withdrawLockAsFxs(idsToIndices[msg.sender][lockIds[i]].index);

        unchecked {
            ++i;
        }
    }
}
```

Note that in this code, a user error may result in `lockIds[i]` being an unused lock id (e.g. lock id 0). This can happen if the provided `_lockIndices` contains invalid or duplicate indices.

In this scenario, the null `index` value will be interpreted as a desire to withdraw/convert the zeroth lock index, which may succeed unexpectedly. It'd likely be better to revert in this scenario instead.

Recommendation: Consider adding checks to ensure that `lockIds[i]` is not an unused value. This can be accomplished by using the `getLockIndexById()` helper function vs `idsToIndices[msg.sender][lockIds[i]].index` directly.

Frax: Fixed in commit 1fb2155.

Frax Security Cartel: Verified.

3.3.2 Redeemer implementation can be initialized by anyone

Description: The `FraxtalERC4626MintRedeemer` is deployed as an implementation, with a proxy that is upgraded to point at it and initialized by calling `initialize()` on it. However, the implementation itself does not have initializing disabled or have `initialize()` called on it.

This allows an attacker to set themselves as the `owner` of the implementation contract, as well as setting any values they would like for the various tokens and oracles.

I don't believe there is any risk to this in the current contract, and it looks like this is handled properly by initializing it in the deployment script, but it would be prudent to disable it at the contract level.

Recommendation: Update the constructor to call `initialize()` to set `_underlyingTkn`, which locks the function.

```
constructor() {
+   underlyingTkn = address(1);
}
```

Frax: Fixed in commit b18af69 and commit c12dc92.

Frax Security Cartel: Verified.

3.4 Informational Findings

3.4.1 Expected lFPIS calculation can be more precise

Description: In the `FPISLockerUtils` contract, the `getCrudeExpectedLFPISOneLock()` function calculates the expected lFPIS amount as follows:

```
// Calculate the expected lFPIS
_expectedLFPIS = uint256(
```

```
    uint128(
        ((_fpisAmount * lFPIS.VOTE_END_POWER_BASIS_POINTS_INT128()) / lFPIS.MAX_BASIS_POINTS_INT128
            ()) +
            ((_fpisAmount * _lockSecsI128) / lFPIS.MAXTIME_INT128())
    )
);
```

In the second term (which is: `((_fpisAmount * _lockSecsI128) / lFPIS.MAXTIME_INT128())`), there is technically an intermediate step missing that multiplies by `VOTE_WEIGHT_MULTIPLIER_INT128` and divides by `MAX_BASIS_POINTS_INT128`. Since both of these values are equal to `10_000` it does not change the result to exclude them, but it may be preferred to explicitly add the intermediate step in case these values ever change.

Recommendation: Consider incorporating a multiplication of `VOTE_WEIGHT_MULTIPLIER_INT128` and a division of `MAX_BASIS_POINTS_INT128` into the expected lFPIS calculation:

```
// Calculate the expected lFPIS
_expectedLFPIS = uint256(
    uint128(
        ((_fpisAmount * lFPIS.VOTE_END_POWER_BASIS_POINTS_INT128()) / lFPIS.MAX_BASIS_POINTS_INT128
            ()) +
            ((_fpisAmount * _lockSecsI128 * lFPIS.VOTE_WEIGHT_MULTIPLIER_INT128()) / lFPIS.
                MAXTIME_INT128() / lFPIS.MAX_BASIS_POINTS_INT128())
    )
);
```

This would be a closer match to the logic in the `FPISLocker` itself.

Frax: Fixed in commit 64da8d9.

Frax Security Cartel: Verified.

3.4.2 `bulkConvertToFXSAndLockInVeFXS()` can be refactored

Description: The `bulkConvertToFXSAndLockInVeFXS()` function contains a number of unnecessary operations and can be refactored to save gas and increase user convenience.

1) In all cases, we check that the two inputted arrays are the same length. This is only necessary in the case that `!_createNewLock`. Otherwise, the `_veFxsLockIndices` array is not used. If anything, it would be preferable to check that this list is empty in this case.
2) In the event that `!_createNewLock`, we set `_createdVeFxsLockIndices = _veFxsLockIndices`. This isn't necessary, as each value in the array is overwritten below with the return value from `convertToFXSAndLock-InVeFXS()`. Instead, we can simply only set `_createdVeFxsLockIndices` in the case that `_createNewLock = true`.
3) We don't need the ternary operator in `_lengthToUse` because we always want to use the length of `_lFpis-LockIndices`. If we are creating new locks, `_veFxsLockIndices` isn't used. If we are using existing locks, we've confirmed the lengths are the same.

Recommendation: Putting this all together, I would recommend the following changes:

```
function bulkConvertToFXSAndLockInVeFXS(
    bool _createNewLock,
    uint128[] calldata _lFpisLockIndices,
    uint128[] calldata _veFxsLockIndices
) external returns (uint128[] memory _createdVeFxsLockIndices, uint256[] memory _fxsGenerated) {
    // Revert if operations are paused
    if (isPaused) revert OperationIsPaused();

    // Make sure conversions are active
    _isFxsConversionActive();

-   // Make sure the supplied params match length
-   if (_lFpisLockIndices.length != _veFxsLockIndices.length) revert ArrayLengthMismatch();

+   // make sure _veFxsLockIndices is the correct length
+   if (_createNewLock) {
+       if (_veFxsLockIndices.length > 0) revert VeFXSListShouldBeEmpty();
+   } else {
+       if (_lFpisLockIndices.length != _veFxsLockIndices.length) revert ArrayLengthMismatch();
+   }

    // Get the lock information
    uint256[] memory lockIds = new uint256[](_lFpisLockIndices.length);
    for (uint256 i; i < _lFpisLockIndices.length; ) {
        lockIds[i] = indicesToIds[msg.sender][_lFpisLockIndices[i]];

        unchecked {
            ++i;
        }
    }

    // Prep the lock index return array
-   _createdVeFxsLockIndices = _createNewLock ? new uint128[](_lFpisLockIndices.length) :
    _veFxsLockIndices;
+   _createdVeFxsLockIndices = new uint128[](_lFpisLockIndices.length);

    // Prep the _fxsGenerated return array
-   uint256 _lengthToUse = _createNewLock ? _lFpisLockIndices.length : _veFxsLockIndices.length;
+   uint256 _lengthToUse = _lFpisLockIndices.length;
    _fxsGenerated = new uint256[](_lengthToUse);

    // Loop through and process the locks
    for (uint256 i; i < _lengthToUse; ) {
        (_createdVeFxsLockIndices[i], _fxsGenerated[i]) = convertToFXSAndLockInVeFXS(
            _createNewLock,
            idsToIndices[msg.sender][lockIds[i]].index,
            _veFxsLockIndices[i]
        );

        unchecked {
            ++i;
        }
    }
}
```

Frax: Fixed in commit c6e1958.

Frax Security Cartel: Verified.

3.4.3 Minor Nits

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

- FPISLocker's `initialize()` function should include all the params in the NatSpec. Missing `_fpisAggregator`, `_fxs`, `_veFXS`.
- NatSpec for `totalFPISSupplyAt()` claims it calculates total voting power at the given block, but it actually returns `fpisAmt`. Travis says code is right behavior, so should just update NatSpec.
- `balanceOfLockedFxs()` function should be renamed to `balanceOfLockedFpis()`.
- When MintRedeemer is initialized, we should check that `_underlyingTkn != address(0)` so that we have a strong invariant that it can only be initialized once.
- The `priceFeedUnderlying` is currently redundant, because the intended `priceFeedVault` implies a constant underlying price of `1e18`. However this may change if the `priceFeedVault` uses a different denomination (e.g. USD) in the future, so it makes sense to keep the current logic.
- In the `FraxtalERC4626MintRedeemer` contract, there are TODO comments regarding the rounding directions of `maxDeposit()`, `maxMint()`, `maxWithdraw()`, `maxRedeem()`, and `mdwrComboView()`. All functions round down, which appears correct and is consistent with ERC4626. This is because the ERC4626 spec disallows overestimation and suggests underestimation if necessary. A smaller max value may lead to leftover dust in the `FraxtalERC4626MintRedeemer`, however this can always be fixed through the `recoverERC20()` function.

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

Frax: The issues have been addressed as follows:

- The NatSpec for FPISLocker's `initialize()` function was updated in commit 7f9d378.
- The NatSpec for `totalFPISSupplyAt()` was updated in commit 7f9d378.
- `balanceOfLockedFxs()` was renamed to `balanceOfLockedFpis()` in commit 7f9d378.
- A stronger initialization invariant was added in commit b18af69.
- As noted above, the `priceFeedUnderlying` logic isn't used currently but may be used in the future.
- The `FraxtalERC4626MintRedeemer` TODO comments were removed in commit aab4b58 and commit b18af69.

Frax Security Cartel: Verified.