Asymmetry Finance Report

afCVX

Conducted by: adriro (@adrianromero)

Date: May 6 to 10, 2024

afCVX Security Review

Disclaimer

The conducted security review represents an evaluation based on the information and code provided by the client. The author, employing a combination of automated tools and manual expertise, has endeavored to identify potential vulnerabilities. It is crucial to understand that this review does not ensure the absolute absence of vulnerabilities or errors within the smart contracts.

Despite exercising due diligence, this assessment may not uncover all potential issues or undiscovered vulnerabilities present in the code. Findings and recommendations are based solely on the information available at the time of the review.

This report is not to be considered as an endorsement or certification of the smart contract's absolute security. Authors cannot assume responsibility for any losses or damages that may arise from the utilization of the smart contracts.

While this assessment aims to identify vulnerabilities, it cannot guarantee absolute security or eliminate all potential risks associated with smart contract usage.

About afCVX

afCVX is a new protocol by Asymmetry Finance built to maximize yield on CVX tokens. The design works as a hybrid CVX wrapper, in which a share of the tokens remain liquid in the Convex staking rewards pool, while the rest is deposited at CLever CVX, a protocol that enables CVX locking with the option to leverage on future yield. Rewards coming from both of these underlying platforms are compounded back into the protocol.

About adriro

adriro is an independent security researcher currently focused on security reviews of smart contracts. He is a top warden at code4rena and serves as a resident auditor at yAudit.

You can follow him on X at @adrianromero or browse his portfolio.

Scope

The scope for the current review targets the afCVX codebase at revision 952b03b18d59136192af8dc70c9a1613ca1a5e48 and includes the following files:

```
src
├── AfCvx.sol
├── PirexMigrator.sol
├── interfaces
│   ├── IProxySource.sol
│   ├── afCvx
│   │   ├── IAfCvx.sol
│   │   └── ICleverCvxStrategy.sol
│   ├── clever
│   │   ├── Constants.sol
│   │   ├── ICLeverCVXLocker.sol
│   │   └── IFurnace.sol
│   ├── convex
│   │   ├── Constants.sol
│   │   ├── ICvxLocker.sol
│   │   └── ICvxRewardsPool.sol
│   └── pirex
│       ├── IPirexCVX.sol
│       └── IPirexLiquidityPool.sol
├── strategies
│   └── CLeverCVXStrategy.sol
└── utils
    ├── SimpleProxy.sol
    ├── SimpleProxyFactory.sol
    ├── TrackedAllowances.sol
    └── Zap.sol
```

Fixes addressing the issues identified in this review have been implemented through PR#5 and PR#6.

Summary

Identifier	Title	Severity	Status
H-1	Convex staking rewards can be lost if anyone claims them on behalf of AfCvx	High	Fixed
H-2	Incorrect rounding could lead to an overflow in the weekly limit	High	Fixed
M-1	Repay amount should be rounded up	Medium	Fixed
M-2	`repay()` incorrectly assumes the current position is at max leverage	Medium	Fixed
M-3	Potential overflow in `maxTotalUnlock()`	Medium	Fixed
M-4	PirexMigrator ignores future rewards issued when a redemption is initiated	Medium	Fixed
M-5	Scheduled redemptions could be bricked if upxCvx is deprecated	Medium	Fixed
L-1	Implement a defensive guard in `requestUnlock()`	Low	Fixed
L-2	Queued withdrawals depend heavily on timely unlocks	Low	Fixed
L-3	afCVX price can be manipulated	Low	Ack
I-1	Pending Convex staking rewards are not included in the vault's assets	Informational	Ack
I-2	ERC4626 functions do not require a pause check	Informational	Fixed
I-3	Weekly withdrawals can be higher than the limit	Informational	Ack
I-4	Weekly withdrawal limit could not be enough to limit immediate withdrawals	Informational	Ack
I-5	Extra rewards from Convex stake are not taken into account	Informational	Ack
I-6	Pausing does not affect operator access	Informational	Fixed
I-7	Max borrow amount calculation ignores pending obligations	Informational	Ack
I-8	Unneeded upxCVX approval to PirexCvx	Informational	Fixed

Identifier	Title	Severity	Status
I-9	Potential denial of service in CRV approval	Informational	Fixed
I-10	Potential spikes in the afCVX share price	Informational	Ack
I-11	CLever rewards are not considered in CleverCvxStrategy	Informational	Ack
G-1	Invert check order in `onlyOperatorOrOwner()`	Gas	Fixed
G-2	Change strict check in `_previewDistribute()`	Gas	Fixed
G-3	Duplicate check in `setProtocolFeeCollector()`	Gas	Fixed
G-4	Cache `unlockObligations` in `totalValue()` and `maxTotalUnlock()`	Gas	Fixed
G-5	`getRequestedUnlocks()` copies the entire unlock history to memory	Gas	Fixed
G-6	Use existing pending obligations to update `unlockObligations`	Gas	Fixed

Critical Findings

None.

High Findings

[H-1] Convex staking rewards can be lost if anyone claims them on behalf of AfCvx

Summary

Claiming the Convex staking rewards on behalf of AfCvx will result in a loss of value as these are not considered when rewards are harvested.

Details

When AfCvx rewards are harvested, the implementation first fetches the Convex staking rewards using the `earned()` function. If the returned value is not zero, it then claims those rewards using `getReward()`. Only after this step does it zap the tokens into CVX and compound the resulting amount back into the protocol.

```
450:     function harvest(uint256 minAmountOut) external onlyOperatorOrOwner returns (
451:         uint256 convexStakedRewards = CVX_REWARDS_POOL.earned(address(this));
```

```
452:         if (convexStakedRewards != 0) {
453:             CVX_REWARDS_POOL.getReward(address(this), false, false);
454:             convexStakedRewards = Zap.swapCvxCrvToCvx(convexStakedRewards, minAmo
455:         }
```

However, it is important to note that Convex staking rewards can be claimed permissionlessly on behalf of any account. `getReward()` takes an `account` parameter and processes their associated rewards.

```
249:     function getReward(address _account, bool _claimExtras, bool _stake) public u
250:         uint256 reward = earnedReward(_account);
251:         if (reward > 0) {
252:             rewards[_account] = 0;
253:             rewardToken.safeApprove(crvDeposits,0);
254:             rewardToken.safeApprove(crvDeposits,reward);
255:             ICrvDeposit(crvDeposits).deposit(reward,false);
256: 
257:             uint256 cvxCrvBalance = cvxCrvToken.balanceOf(address(this));
258:             if(_stake){
259:                 IERC20(cvxCrvToken).safeApprove(cvxCrvRewards,0);
260:                 IERC20(cvxCrvToken).safeApprove(cvxCrvRewards,cvxCrvBalance);
261:                 IRewards(cvxCrvRewards).stakeFor(_account,cvxCrvBalance);
262:             }else{
263:                 cvxCrvToken.safeTransfer(_account, cvxCrvBalance);
264:             }
265:             emit RewardPaid(_account, cvxCrvBalance);
266:         }
267: 
268:         //also get rewards from linked rewards
269:         if(_claimExtras){
270:             uint256 length = extraRewards.length;
271:             for(uint i=0; i < length; i++){
272:                 IRewards(extraRewards[i]).getReward(_account);
273:             }
274:         }
275:     }
```

This means that if anyone other than the intended caller claims the associated Convex staking rewards for the AfCvx contract, those rewards will be sent to the contract but left unconsidered when rewards are harvested and compounded.

Impact

Rewards from Convex stake could be lost and left unprocessed in the contract if anyone claims them on behalf of AfCvx.

Recommendation

Fetch the rewards (the zero check is already performed in the implementation of `getReward()`) and then use the balance of cvxCRV held by the contract.

```
CVX_REWARDS_POOL.getReward(address(this), false, false);
uint256 convexStakedRewards = CVXCRV.balanceOf(address(this));
if (convexStakedRewards != 0) {
    convexStakedRewards = Zap.swapCvxCrvToCvx(convexStakedRewards, minAmountOut);
}
```

[H-2] Incorrect rounding could lead to an overflow in the weekly limit

Summary

Assets are incorrectly rounded up in the implementation of `maxRedeem()`, causing a potential overflow of the weekly withdrawal limit.

Details

To determine the maximum amount of shares that can be redeemed, the implementation considers the owner balance, the weekly withdrawal limit, and the amount of available funds for immediate withdrawal.

```
240:     function maxRedeem(address owner)
241:         public
242:         view
243:         virtual
244:         override(ERC4626Upgradeable, IERC4626)
245:         returns (uint256 maxShares)
246:     {
247:         if (paused) return 0;
248:         uint256 availableCvx = CVX.balanceOf(address(this)) + CVX_REWARDS_POOL.ba
249:         return balanceOf(owner).min(previewWithdraw(weeklyWithdrawalLimit)).min(p
250:     }
```

As we can see in the previous snippet, assets are converted back to shares using `previewWithdraw()`, which rounds up. Rounding up here will mean that more assets than the intended limits will be allowed when redeeming shares.

Suppose the price per share is 1.1, and that `weeklyWithdrawalLimit` is set to 10. In this scenario, `previewWithdraw(weeklyWithdrawalLimit)` will output 10 due to rounding up. However, when calling `redeem(10)`, those 10 shares will be converted back to 11 assets, effectively allowing the redemption of a higher amount of assets than the intended withdrawal limit of 10.

Impact

Rounding up could not only allow a higher withdrawal limit than expected, but the use of unchecked math when updating the `weeklyWithdrawalLimit` variable could potentially lead to an integer overflow, completely bypassing the weekly withdrawal limits.

```
288:     function _withdraw(address caller, address receiver, address owner, uint256 a
289:         internal
290:         virtual
291:         override
292:     {
293:         unchecked {
294:             weeklyWithdrawalLimit -= uint128(assets);
295:         }
```

Recommendation

Use `_convertToShares(assets, Math.Rounding.Floor)` instead of `previewWithdraw(assets)`.

Medium Findings

[M-1] Repay amount should be rounded up

Summary

The repay amount is incorrectly rounded down while winding down the debt in the CLever CVX strategy.

Details

Before CVX can be withdrawn from CLever, the current debt needs to be repaid in order to make margin for the unlock. The core logic is present in the `repay()` function, and the repay amount is calculated in `_calculateRepayAmount()`.

```
245:     function repay() external onlyOperatorOrOwner {
246:         unlockInProgress = true;
247:         uint256 amount = unlockObligations;
248:         if (amount != 0) {
249:             (uint256 repayAmount, uint256 repayFee) = _calculateRepayAmount(amoun
```

```
316:     function _calculateRepayAmount(uint256 _lockedCVX) private view returns (uint
317:         uint256 reserveRate = CLEVER_CVX_LOCKER.reserveRate();
318:         uint256 repayRate = CLEVER_CVX_LOCKER.repayFeePercentage();
319:         repayAmount = _lockedCVX.mulDiv(reserveRate, CLEVER_FEE_PRECISION);
320:         repayFee = repayAmount.mulDiv(repayRate, CLEVER_FEE_PRECISION);
321:     }
```

As we can see in the previous snippets of code, the implementation grabs the pending obligations (`unlockObligations`) and calculates the repayment using the reserve rate (currently at 50%), rounding down in the division by the fee precision.

At max leverage, rounding down in this calculation could mean not reaching the needed repayment amount to later keep a healthy position when the unlock is performed.

Suppose we have deposited 100 tokens and borrowed 50, and that pending obligations are 25. To simplify things, let's consider a repayment fee of 0%. The implementation will calculate `repayAmount` as 50% of the amount to be unlocked, which will be 12 since it rounds down, leaving a debt of `50 - 12 = 38` . When trying to perform the unlock, deposits will be offset by `100 - 25 = 75` , invalidating the account health invariant in `_checkAccountHealth()` .

Impact

In certain scenarios, the repaid amount may be insufficient to perform the requested withdrawals, effectively bricking the unlocking process.

Recommendation

The calculation of `repayAmount` in `_calculateRepayAmount()` should round up.

[M-2] `repay()` incorrectly assumes the current position is at max leverage

Summary

Repayments are always calculated assuming the current debt taken in CLever is at its maximum, overestimating the repay amount and potentially causing asset losses if this amount exceeds the current debt.

Details

To perform the unlocking of CVX needed to fulfill pending obligations, the CLever CVX strategy needs to consider first the repayment of the current debt held at CLever in order to maintain a healthy position when locked assets decreased.

```
245:     function repay() external onlyOperatorOrOwner {
246:         unlockInProgress = true;
247:         uint256 amount = unlockObligations;
248:         if (amount != 0) {
249:             (uint256 repayAmount, uint256 repayFee) = _calculateRepayAmount(amoun
250:             (uint256 clevCvxAvailable,) = FURNACE.getUserInfo(address(this));
251:             uint256 clevCvxRequired = repayAmount + repayFee;
252:
253:             if (clevCvxRequired > clevCvxAvailable) revert InsufficientFurnaceBal
254:
255:             FURNACE.withdraw(address(this), clevCvxRequired);
256:             CLEVER_CVX_LOCKER.repay(0, repayAmount);
```

```
257:         }
258:     }
```

```
316:     function _calculateRepayAmount(uint256 _lockedCVX) private view returns (uint
317:         uint256 reserveRate = CLEVER_CVX_LOCKER.reserveRate();
318:         uint256 repayRate = CLEVER_CVX_LOCKER.repayFeePercentage();
319:         repayAmount = _lockedCVX.mulDiv(reserveRate, CLEVER_FEE_PRECISION);
320:         repayFee = repayAmount.mulDiv(repayRate, CLEVER_FEE_PRECISION);
321:     }
```

We can see that `repayAmount` is calculated in `_calculateRepayAmount()` by applying the `reserveRate` factor to the amount that needs to be unlocked (coming from `unlockObligations`). Here, the implementation is assuming the current position is at max leverage. This will work fine if, for example, the locked assets are 100 and the current debt is 50; if we want to withdraw 20 tokens we need to wind down the debt by 10. However, in any other case, it will overestimate the amount of clevCVX that needs to be repaid.

More importantly, if the calculated repay amount is bigger than the current debt, the implementation of `CLEVER_CVX_LOCKER.repay()` will ignore any excess:

```
461:     // 3. check repay with cvx and take fee
462:     if (_cvxAmount > 0 && _totalDebt > 0) {
463:       if (_cvxAmount > _totalDebt) _cvxAmount = _totalDebt;
```

This means that clevCVX tokens will be withdrawn from the Furnace without actually being used to repay any debt.

Impact

Repay amount is overestimated, leading to unnecessary repayments that have associated fees. Asset losses could occur if the repay amount exceeds the current debt, as those tokens will be withdrawn from the Furnace and sit in the strategy contract without being used.

Recommendation

The implementation of `_calculateRepayAmount()` should take into account the current debt held at CLever to calculate `repayAmount`, and repay the minumum amount such that the health check succeeds while later unlocking the assets (refer to the implementation of `_checkAccountHealth()` in CLever).

[M-3] Potential overflow in `maxTotalUnlock()`

Summary

The implementation of `maxTotalUnlock()` assumes that the available unrealised deposits in the Furnace should not exceed the amount borrowed at the CLever locker.

Details

To determine the maximum amount of assets that can be unlocked, `maxTotalUnlock()` needs to calculate how much of the current debt can be repaid using the unrealised assets available at the Furnace.

```
260:     function maxTotalUnlock() external view returns (uint256 maxUnlock) {
261:         // get available clevCVX from Furnace
262:         (uint256 clevCvxAvailable,) = FURNACE.getUserInfo(address(this));
263:
264:         // subtract repay fee
265:         uint256 reserveRate = CLEVER_CVX_LOCKER.reserveRate();
266:         uint256 repayRate = CLEVER_CVX_LOCKER.repayFeePercentage();
267:         uint256 repayAmount = clevCvxAvailable.mulDiv(CLEVER_FEE_PRECISION, CLEVE
268:
269:         (uint256 totalDeposited,,, uint256 totalBorrowed,) = CLEVER_CVX_LOCKER.ge
270:         // decrease borrowed amount
271:         totalBorrowed -= repayAmount;
```

An overflow could occur if `repayAmount` is bigger than `totalBorrowed` (line 271), due to the native checked math added by the compiler.

It is unlikely to expect this could happen since borrows are deposited into the Furnace. However, deposits can be made directly to the Furnace, which could derive in the conditions required to trigger the overflow.

Impact

An overflow in `maxTotalUnlock()` implies blocking the unlocking process, since `requestUnlock()` needs to query this limit.

Recommendation

Check if `totalBorrowed` is greater than `repayAmount`, and cap `totalBorrowed` to zero if not.

```
if (totalBorrowed > repayAmount) {
    unchecked {
        totalBorrowed -= repayAmount;
    }
} else {
    totalBorrowed = 0;
}
```

[M-4] PirexMigrator ignores future rewards issued when a redemption is initiated

Summary

During a pxCvx redemption, PirexCvx mints tokens representing potential future rewards that could accrue during the unlocking process. However, PirexMigrator does not account for these tokens, causing migrators to lose access to the associated rewards.

Details

When a redemption is initiated in PirexCvx, the `initiateRedemptions()` function mints tokens representing future rewards that will accrue during the epochs between the redemption initiation and the unlock completion.

```
357:     function _mintFutures(
358:         uint256 rounds,
359:         Futures f,
360:         uint256 assets,
361:         address receiver
362:     ) internal {
363:         emit MintFutures(rounds, f, assets, receiver);
364: 
365:         ERC1155PresetMinterSupply token = f == Futures.Vote ? vpxCvx : rpxCvx;
366:         uint256 startingEpoch = getCurrentEpoch() + EPOCH_DURATION;
367:         uint256[] memory tokenIds = new uint256[](rounds);
368:         uint256[] memory amounts = new uint256[](rounds);
369: 
370:         for (uint256 i; i < rounds; ++i) {
371:             tokenIds[i] = startingEpoch + i * EPOCH_DURATION;
372:             amounts[i] = assets;
373:         }
374: 
375:         token.mintBatch(receiver, tokenIds, amounts, UNUSED_1155_DATA);
376:     }
```

These tokens, in the form of vpxCvx or rpxCvx, can be eventually exchanged for rewards using `redeemFuturesRewards()`.

In the PirexMigrator side, `_initiateRedemption()` records the upxCvx balance, but ignores the minted rpxCvx tokens. These tokens are neither transferred to the caller nor handled during redemption completions in `redeem()`. Consequently, the rpxCvx tokens will eventually become locked within the PirexMigrator contract.

```
161:     function _initiateRedemption(uint256 _amount, uint256 _lockIndex, address _re
162:         (,,,ICVXLocker.LockedBalance[] memory _lockData) = CVX_LOCKER.lockedBalan
163:         uint256 _unlockTime = _lockData[_lockIndex].unlockTime;
164:         uint256 _balanceBefore = UPX_CVX.balanceOf(address(this), _unlockTime);
```

```
165:
166:         {
167:             uint256[] memory _assets = new uint256[](1);
168:             _assets[0] = _amount;
169:             uint256[] memory _lockIndexes = new uint256[](1);
170:             _lockIndexes[0] = _lockIndex;
171:             PIREX_CVX.initiateRedemptions(_lockIndexes, IPirexCVX.Futures.Reward,
172:         }
173:
174:         _amount = UPX_CVX.balanceOf(address(this), _unlockTime) - _balanceBefore;
175:         balances[_receiver][_unlockTime] += _amount;
176:
177:         emit InitiatedRedemption(_amount, _unlockTime, _receiver);
178:
179:         return _amount;
180:     }
```

Impact

pxCvx holders using PirexMigrator will be unable to access the future rewards normally issued when initiating redemptions through the Pirex protocol.

Recommendation

Tokens representing future rewards could be forwarded to the caller that initiates the redemption. This would need to be handled by the PirexMigrator contract itself, as the call to `initiateRedemptions()` receives a single recipient that gets both upxCvx and the futures.

[M-5] Scheduled redemptions could be bricked if upxCvx is deprecated

Summary

Redemptions in PirexMigrator could be affected by a denial of service if upxCvx is deprecated.

Details

The PirexCvx contract has a boolean variable that indicates whether upxCvx has been deprecated. According to the following comment, this behavior could be enabled in case of a mass unlock of CVX tokens held by the protocol.

```
109:     // In the case of a mass unlock by Convex, the current upxCVX would be deprec
110:     // and should allow holders to immediately redeem their CVX by burning upxCVX
111:     bool public upxCvxDeprecated;
```

If this variable gets enabled, `PirexCvx.redeem()` would revert, creating a denial of service in `PirexMigrator.redeem()`.

```
646:     function redeem(
647:         uint256[] calldata unlockTimes,
648:         uint256[] calldata assets,
649:         address receiver
650:     ) external whenNotPaused nonReentrant {
651:         if (upxCvxDeprecated) revert RedeemClosed();
652:
653:         _redeem(unlockTimes, assets, receiver, false);
654:     }
```

Impact

Initiated redemptions in PirexMigrator will be bricked in the unlikely scenario that upxCvx gets deprecated in PirexCvx.

Recommendation

Add a boolean flag to `redeem()` to allow switching the underlying call to PirexCvx between `redeem()` and `redeemLegacy()`.

Low Findings

[L-1] Implement a defensive guard in `requestUnlock()`

The implementation of `requestUnlock()` loops through the CVX locks to find the earliest epoch in which the request amount could be fit.

```
167:         uint256 locksLength = locks.length;
168:         for (uint256 i; i < locksLength; i++) {
169:             // amount that can be unlocked at the unlock epoch
170:             uint256 locked = locks[i].pendingUnlock;
171:             uint64 epoch = locks[i].unlockEpoch;
172:
173:             if (existingUnlockObligations != 0) {
174:                 // subtract previous unlock requests from the available amount
175:                 if (existingUnlockObligations < locked) {
176:                     unchecked {
177:                         locked = locked - existingUnlockObligations;
178:                     }
179:                     existingUnlockObligations = 0;
180:                 } else {
181:                     unchecked {
182:                         existingUnlockObligations = existingUnlockObligations - l
183:                     }
184:                     // move to the next epoch as all available amount was already
185:                     continue;
186:                 }
187:             }
```

```
188:
189:             if (amount > locked) {
190:                 unlocks.push(UnlockRequest({ unlockAmount: uint192(locked), unloc
191:                 unchecked {
192:                     amount = amount - locked;
193:                 }
194:             } else {
195:                 unlocks.push(UnlockRequest({ unlockAmount: uint192(amount), unloc
196:                 unlockEpoch = epoch;
197:                 break;
198:             }
199:         }
200:     }
```

The loop ranges from line 167 until line 199, after which the enclosing function finishes. The implementation assumes that there will always be enough space to fit the requested amount, which is correct if everything goes well. However, if this invariant is somehow broken, the function will silently finish without accommodating the requested amount.

Consider adding an extra check to revert if the loop ends without assigning the entirety of the requested amount.

```
            if (amount > locked) {
                unlocks.push(UnlockRequest({ unlockAmount: uint192(locked), unlockEpoc
                unchecked {
                    amount = amount - locked;
                }
            } else {
                unlocks.push(UnlockRequest({ unlockAmount: uint192(amount), unlockEpoc
-               unlockEpoch = epoch;
-               break;
+               return epoch;
            }
        }

+       revert InvalidState();
    }
```

[L-2] Queued withdrawals depend heavily on timely unlocks

When a user requests a withdrawal, the implemention in `requestUnlock()` will fetch the current state of Convex locks through the CLever locker, and schedule the operation to the earliest epoch having enough free liquidity to accommodate the requested amount.

Note that this process just enqueues the operation in the `unlocks` array corresponding to the account. At this point there is no request made to CLever, state is only modified at the strategy level.

The process continues with the operator calling `repay()` and `unlock()` to complete the unlocking of the CVX required to fulfill the previously scheduled operations. It is at this point that unlocks are requested in the CLever protocol.

The nature of this two-step process requires that both actions take place during the same epoch. In the first phase, `requestUnlock()` uses the state of locks at the current epoch. If `unlock()` is later called at a different epoch, then unlocks at CLever will be executed using a schedule that diverges from the expected scheduled operations in the strategy.

From the contracts it remains unclear how and when these operations will be triggered off-chain, but is it critical that no pending obligation crosses the epoch boundary without being processed. This implies that no requests should be made after the last call to `unlock()` (for the current epoch) and until the beginning of the next epoch, which could be implemented as a "maintenance window" that blocks new requests and gives enough time to the operator to complete the unlocking process.

[L-3] afCVX price can be manipulated

The afCVX vault bases its price (or share to asset relation) by quantifying the number of assets in `totalAssets()`. These assets come from the CVX balance held by the contract (`unlocked`), the value held by the CLever CVX strategy (`lockedInClever`) and the tokens staked in the Convex reward pool (`staked`).

All of these sources can be manipulated by donations. CVX can be transferred to the contract, staking can be made on behalf of the contract, and CLever deposits can also be made on behalf of another account.

Price manipulation attacks imply a huge risk if the afCVX token is used as collateral, or if its oracle (`convertToShares()` and `convertToAssets()`) is being accessed to query for the price.

Out of these options, the most interesting case are donations to the CLever Furnace. At the time of writing, clevCVX can be obtained at a ~20% discount over CVX, while the afCVX implementation assumes a 1:1 relation. Attackers could purchase clevCVX at a discount and then donate these to the furnace to be considered as CVX in the vault's TVL.

Informational Findings

[I-1] Pending Convex staking rewards are not included in the vault's assets

The implementation of `totalAssets()` considers unlocked balance present in the AfCvx contract, locked balance in CLever, rewards from CLever (realised CVX from the Furnace) and staked tokens in the Convex rewards pool. However, unclaimed rewards from the latter are not factored into this total amount.

Note that compounding assets from yield is mentioned in the standard as a recommendation.

[I-2] ERC4626 functions do not require a pause check

All of the main ERC4626 interactions, `deposit()`, `mint()`, `withdraw()` and `redeem()`, have the `whenNotPaused` check but also factor the pause in their respective maximums. Since these functions include the check in their implementations, the effective maximum will be 0 when the contract is paused.

[I-3] Weekly withdrawals can be higher than the limit

The AfCvx contract has a weekly limit that controls immediate withdrawals. The core logic is included in `_updateWeeklyWithdrawalLimit()`.

```
388:     function _updateWeeklyWithdrawalLimit() private {
389:         if (block.timestamp < withdrawalLimitNextUpdate) return;
390:
391:         uint256 tvl = totalAssets();
392:         uint128 withdrawalLimit = uint128(_mulBps(tvl, weeklyWithdrawalShareBps))
393:         uint64 nextUpdate = uint64(block.timestamp + 7 days);
394:         weeklyWithdrawalLimit = withdrawalLimit;
395:         withdrawalLimitNextUpdate = nextUpdate;
396:
397:         emit WeeklyWithdrawLimitUpdated(withdrawalLimit, nextUpdate);
398:     }
```

Since the update is not done while withdrawing, a user could take advantage of the remainder of a previous period, then update the limit, and withdraw again up to the new updated limit.

[I-4] Weekly withdrawal limit could not be enough to limit immediate withdrawals

Since the weekly withdrawal limit is represented as a share of the total value locked, it could still be possible to drain the Convex staking side through multiple periods.

For example, if the CLever distribution is 80% and the weekly share is 5%, then in the first week it could be possible to withdraw a 25% of the Convex stake. After the limit is updated, and considering the original 80% in CLever has not been modified, the following week it would be possible to withdraw a ~31.6% of the Convex stake.

[I-5] Extra rewards from Convex stake are not taken into account

The Convex staking contract can be setup to issue extra rewards using tokens different from cvxCRV. These potential extra rewards are not considered in the process of harvesting and compounding the protocol's rewards in the AfCvx contract.

Currently these extra rewards are not being used and it is unclear if they will be added in the future. In any case, this can be handled eventually by a protocol upgrade.

[I-6] Pausing does not affect operator access

After the emergency shutdown is triggered, most of the contracts public facing functionality will be inaccessible. However, the operator is still able to perform operations such as `distribute()` or `harvest()` in AfCvx, or `borrow()`, `repay()` or `unlock()` in CleverCvxStrategy.

It should be noted that operator and owner are two different roles. After the owner kills the contracts, the operator will still be able to partially access its functionality.

[I-7] Max borrow amount calculation ignores pending obligations

The implementation of `_calculateMaxBorrowAmount()` does not take into account potential pending obligations (`unlockObligations`) that would eventually need to be performed when unlocking assets from CLever.

If the borrowed amount is leveraged to the maximum while ignoring the pending obligations, it could lead to an unnecessary cycle of borrowing and repaying, which carry associated fees, as the funds would need to be repaid afterwards.

It is recommended to ensure that `borrow()` is called after settling pending unlocks, or, alternatively, to factor the pending obligations in `_calculateMaxBorrowAmount()`.

[I-8] Unneeded upxCVX approval to PirexCvx

PirexMigrator sets the PirexCvx contract as the approved operator for upxCVX, but this is not needed as the implementation of PirexCvx burns the upxCVX tokens directly from the caller. No transfers are required.

[I-9] Potential denial of service in CRV approval

The CRV `approve()` function requires the current allowance to be zero before configuring a new value.

This can cause a denial of service while handling the token's allowance in the implementation of `Zap.swapCvxCrvToCvx()` if the granted allowance is not fully consumed.

Currently, the call to Curve's `exchange_underlying()` should fully consume the granted allowance, resetting its value to zero before the next approval happens. However, it is recommended to change the `safeApprove()` call with `forceApprove()` to prevent any accidental denial of service.

[I-10] Potential spikes in the afCVX share price

There are different situations across the protocol that could create a *spike* in the afCVX share price.

- Since rewards coming from the Convex staking pool are not factored in `totalAssets()`, harvesting these in `harvest()` will create a sudden increase in the assets held by the vault.

- Fees taken in immediate withdrawals also create an increase in the number of assets. Both `withdraw()` and `redeem()` take a fee defined by `withdrawalFeeBps` that will shift the asset/share relation.
- Borrowing used to leverage in the CLever protocol decreases the share price by the repayment fee charged by CLever. Whenever the operator increases the debt in the strategy using `borrow()`, the reported value by the strategy will take a loss of the fee needed to repay the increase in debt (currently 1%), causing the share price to drop.

These sharp price movements can potentially be exploited through sandwich attacks by MEV operators. However, it is important to note that such scenarios are naturally limited by small percentages, requiring substantial amounts to be financially significant. Furthermore, exiting afCVX would eventually require either paying the immediate withdrawal fee or having exposure to CVX during the unlocking process.

[I-11] CLever rewards are not considered in CleverCvxStrategy

The CLever protocol works using self-repaying loans in which the issued debt is being automatically repaid using the yield generated by the collateral. When no debt is taken, this yield is accumulated as *rewards* that keep track of the credit the account has. This can be seen in the implementation of `CLeverCVXLocker.getUserInfo()`.

```
225:     // update total reward and total Borrowed
226:     totalBorrowed = _info.totalDebt;
227:     totalReward = uint256(_info.rewards).add(
228:       accRewardPerShare.sub(_info.rewardPerSharePaid).mul(totalDeposited) / PRECI
229:     );
230:     if (totalBorrowed > 0) {
231:       if (totalReward >= totalBorrowed) {
232:         totalReward -= totalBorrowed;
233:         totalBorrowed = 0;
234:       } else {
235:         totalBorrowed -= totalReward;
236:         totalReward = 0;
237:       }
238:     }
```

When rewards exceed the current debt, `totalBorrowed` will be zero and `totalReward` will be the difference, which can be interpreted as the credit of the position.

The CleverCvxStrategy contract ignores these rewards as they are not considered when fetching the current position using `getUserInfo()`, which impacts the functions `totalValue()` and `_calculateMaxBorrowAmount()`.

Normally the protocol should operate with leverage at all times, meaning rewards should never accumulate. However, take into account that during the early stage of the protocol (until debt is first taken) or whenever debt is fully cleared, yield will be accumulated as rewards, which can be thought as "free to claim" clevCVX.

Gas Findings

[G-1] Invert check order in `onlyOperatorOrOwner()`

Given it is more likely that the operator will access the functions under this modifier, consider switching the order of the checks so that the operator check happens first.

Instances:

- AfCvx.sol#L43
- CLeverCVXStrategy.sol#L39

[G-2] Change strict check in `_previewDistribute()`

In the implementation of `_previewDistribute()`, the greater than or equal check present in line 410 could be turned into a strictly greater check, since `delta` will be 0 if both variables are equal.

```
410:         if (targetLockedInClever >= lockedInClever) {
411:             uint256 delta;
412:             unchecked {
413:                 delta = targetLockedInClever - lockedInClever;
414:             }
415:             cleverDepositAmount = delta > unlocked ? unlocked : delta;
416:         }
```

[G-3] Duplicate check in `setProtocolFeeCollector()`

The implementation of `setProtocolFeeCollector()` validates the `newProtocolFeeCollector` argument twice.

It first uses the `validAddress` modifier, and then does an explicit check again at line 519.

[G-4] Cache `unlockObligations` in `totalValue()` and `maxTotalUnlock()`

The implementation of `totalValue()` first reads `unlockObligations` in the guard of the if at line 81, then uses its value again in each of the branches.

Similarly, in `maxTotalUnlock()` it is first read in line 274, and again in line 276 to perform the subtraction.

Consider caching this value in a local variable.

[G-5] `getRequestedUnlocks()` copies the entire unlock history to memory

To output the pending unlocks, `getRequestedUnlocks()` copies the entire `requestedUnlocks[account].unlocks` array from storage to memory.

This can be a gas intensive operation if the history of queued withdrawals grows in time. Consider just copying the needed portion which goes from `nextUnlockIndex` to the length of the array.

[G-6] Use existing pending obligations to update `unlockObligations`

In `requestUnlock()`, the update to `unlockObligations` could benefit from the current value present in `existingUnlockObligations`.

```
     // total unlock amount already requested
     uint256 existingUnlockObligations = unlockObligations;

-    unlockObligations += amount;
+    unlockObligations = existingUnlockObligations + amount;
```