Asymmetry Finance Report

AfEth

Conducted by: adriro (@adrianromero)

Date: Jan 22 to 29, 2024

Asymmetry Finance 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 Asymmetry Finance

Asymmetry Finance is a protocol designed to bring a solution to the centralization of the staked Ether market. The protocol will incentivize users with a sustainable yield generation model that results in market leading returns. At the same time, risk is diffused through diversification that is characteristic of the product, the Asymmetry Ethereum Products (afETH & safETH). AfETH and safETH are Liquid Staked Token (LST) Ethereum Index products designed to more equally distribute Total Value Locked (TVL) to LST providers.

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 AfEth codebase at revision `8eb665e5434f7433c211e3ca4e2650ebffc3ecd6` and is limited to the following files:

```
src
├── AfEth.sol
├── AfEthRelayer.sol
├── interfaces
│   ├── IChainlinkFeed.sol
│   ├── IClaimZap.sol
│   ├── ISwapRouter.sol
│   ├── IWETH.sol
│   ├── afeth
│   │   ├── IAfEth.sol
│   │   └── IVotiumStrategy.sol
│   ├── curve-convex
│   │   ├── Constants.sol
│   │   ├── ICvxEthPool.sol
│   │   ├── ILockedCvx.sol
│   │   ├── ISnapshotDelegationRegistry.sol
│   │   └── IVotiumMerkleStash.sol
│   ├── frax
│   │   ├── IFraxEthMinter.sol
│   │   ├── IFrxEthPool.sol
│   │   ├── ISfrxETH.sol
│   │   └── frxETH.sol
│   └── safeth
│       └── ISafEth.sol
├── strategies
│   ├── SfrxEthStrategy.sol
```

```
│   └── VotiumStrategy.sol
└── utils
    ├── CvxEthOracleLib.sol
    ├── HashLib.sol
    └── TrackedAllowances.sol
```

Summary

Identifier	Title	Severity	Status
C-1	ETH from instant withdrawals in the Votium strategy are not forwarded to AfEth	Critical	Fixed
H-1	Incorrect share calculation while depositing in AfEth	High	Fixed
H-2	Incorrect slippage check when rewards are compounded in the sfrxETH strategy	High	Fixed
H-3	Output calculation is incorrect in AfEth quick actions	High	Fixed
H-4	Potential overflow while depositing in the Votium strategy	High	Fixed
H-5	Expired locks in vlCVX can be kicked	High	Mitigated
M-1	Withdrawals can be blocked if vlCVX is shutdown	Medium	Fixed
L-1	`AfEth::deposit()` can revert if the deposit value in Votium is zero	Low	Ack
L-2	Missing storage gap in TrackedAllowances	Low	Fixed
L-3	Conflicting management for quick actions	Low	Fixed
L-4	Inaccurate strict comparisons in VotiumStrategy	Low	Fixed
L-5	Withdrawals can be bricked when redeeming zero shares from the sfrxETH strategy	Low	Ack
L-6	Queued withdrawals can be overwritten when requesting a zero amount withdrawal	Low	Fixed
L-7	Reward swapping should ensure no CVX held for obligations is traded	Low	Fixed
I-1	Locked withdrawals can still be executed after emergency pause	Informational	Fixed

Identifier	Title	Severity	Status
I-2	Rewards can be externally claimed	Informational	Ack
I-3	Missing events for parameter change	Informational	Fixed
I-4	Payable deposit function takes an amount parameter	Informational	Fixed
I-5	Function not used internally should have `external` visibility	Informational	Fixed
G-1	Access control is checked twice in `VotiumStrategy::deposit()`	Gas	Fixed
G-2	Duplicate check for zero in `VotiumStrategy::deposit()`	Gas	Fixed
G-3	Check before sending ETH is always satisfiable in `VotiumStrategy::withdrawLocked()`	Gas	Ack
G-4	Chainlink response validation can be simplified	Gas	Fixed

Critical Findings

[C-1] ETH from instant withdrawals in the Votium strategy are not forwarded to AfEth

Summary

ETH resulting from instant withdrawals in VotiumStrategy is not forwarded to the AfEth contract to be returned to the user, causing different negative consequences.

Details

When a new withdrawal is requested in VotiumStrategy, the implementation of `requestWithdraw()` checks if the available unlocked balance is enough to cover the withdrawal and executes an instant withdrawal instead of queueing it. When this happens, the CVX is sold for ETH:

```
172:         if (unlockedCvx > totalUnlockObligations) {
173:             ethOutNow = unsafeSellCvx(cvxAmount);
174:             unchecked {
175:                 _lock(unlockedCvx - totalUnlockObligations);
176:             }
177:         } else {
```

However, the resulting ETH is not forwarded to back to AfEth and will simply remain in VotiumStrategy. In AfEth, `requestWithdraw()` is expecting this ETH to send it back to the user:

```
167:         (locked, totalEthOut, cumulativeUnlockThreshold) = VOTIUM.requestWithdraw
168:         totalEthOut += sfrxEthOut;
169:         uint256 minOut = locked ? minOutOnlySfrx : minOutAll;
170:
171:         if (totalEthOut < minOut) revert BelowMinOut();
172:         if (totalEthOut > 0) msg.sender.safeTransferETH(totalEthOut);
```

Impact

Critical. Different outcomes may happen. If AfEth lacks the required ETH, it will revert the transaction, blocking the process. If AfEth has the required balance, then the ETH used will be balance from locked rewards, or owner funds (premint or fees). ETH left in VotiumStrategy will be then used as part of rewards as the implementation of `swapRewards()` will use all ETH present in the contract to be compounded in the strategies.

Recommendation

Forward the ETH to the AfEth contract.

```
     if (unlockedCvx > totalUnlockObligations) {
         ethOutNow = unsafeSellCvx(cvxAmount);
         unchecked {
             _lock(unlockedCvx - totalUnlockObligations);
         }
+        msg.sender.safeTransferETH(ethOutNow);
     } else {
```

High Findings

[H-1] Incorrect share calculation while depositing in AfEth

Summary

Share calculation in AfEth includes the amounts from the deposit, leading to an incorrect number of shares minted.

Details

The `deposit()` function in AfEth first makes the deposit into each strategy and then queries for the total ETH value of each strategy. This will cause an issue when shares are calculated, as the total (`totalValue`) already factors the user's deposit.

```
144:         amount = supply == 0 ? depositValue : depositValue * supply / totalValue;
```

Impact

High. The amount of minted shares of AfEth is incorrect.

Recommendation

Inverse the order of the operations, first fetch the totals of each strategy and then make the deposits.

[H-2] Incorrect slippage check when rewards are compounded in the sfrxETH strategy

Summary

The implementation of `depositRewardsAndRebalance()` uses an incorrect expression to calculate the slippage when depositing in the sfrxETH strategy.

Details

When rewards are compounded using `depositRewardsAndRebalance()`, the caller specifies a `sfrxPerEthMin` parameter to check slippage when ETH is deposited in the sfrxETH strategy. This check is given by the following expression:

```
228:             uint256 sfrxOut = SfrxEthStrategy.deposit(sfrxDepositAmountEth);
229:             if (sfrxDepositAmountEth.divWad(sfrxOut) < params.sfrxPerEthMin) reve
```

The ETH amount is divided by the resulting sfrxETH, and checked against `sfrxPerEthMin`. This is incorrect as `sfrxPerEthMin` indicates the amount of sfrxETH per 1 unit of ETH, not the opposite.

Impact

High. Since sfrxETH is more expensive than ETH, the slippage calculation could result in a sandwich attack.

Recommendation

The proper check should be to divide `sfrxOut` by `sfrxDepositAmountEth`.

```
-   if (sfrxDepositAmountEth.divWad(sfrxOut) < params.sfrxPerEthMin) revert BelowMinOu
+   if (sfrxOut.divWad(sfrxDepositAmountEth) < params.sfrxPerEthMin) revert BelowMinOu
```

[H-3] Output calculation is incorrect in AfEth quick actions

Summary

Both `quickDeposit()` and `quickWithdraw()` present errors in the calculation of their output amounts.

Details

In `quickDeposit()`, the resulting AfEth is given by:

```
281:         afEthOut = mulBps(minOut.divWad(price()), quickDepositFeeBps);
```

We can see that first it uses `minOut` instead of `msg.value`. Second, it multiplies that result by the quick deposit fee, resulting in the fee that should be paid, and not the final amount returned to the user.

Similarly in `quickWithdraw()`, the resulting ETH amount presents the same issues:

```
299:         ethOut = mulBps(minOut.mulWad(price()), quickWithdrawFeeBps);
```

Impact

High. Incorrect calculation might lead to severe loss of funds, however this should be caught by the slippage check.

Recommendation

For `quickDeposit()`:

```
afEthOut = mulBps(msg.value.divWad(price()), ONE_BPS - quickDepositFeeBps);
```

For `quickWithdraw()`:

```
ethOut = mulBps(amount.mulWad(price()), ONE_BPS - quickWithdrawFeeBps);
```

[H-4] Potential overflow while depositing in the Votium strategy

Summary

Optimistic CVX relocking when a new deposit is made may cause an accidental denial of service in VotiumStrategy.

Details

The implementation of `deposit()` relocks assets in case these are not routed towards pending obligations.

```
135:         if (cvxAmount > totalUnlockObligations) {
136:             _processExpiredLocks(true);
137:             unchecked {
138:                 uint256 netExtraLock = cvxAmount - totalUnlockObligations;
139:                 if (netExtraLock > 0) _lock(netExtraLock);
140:             }
141:         } else {
142:             uint256 unlocked = _unlockAvailable();
143:             _lock(unlocked - totalUnlockObligations);
144:         }
```

In this scenario (lines 141-144), the function calculates the amount of assets to be relocked as the difference between the available unlocked balance and the pending obligations, causing an overflow if the latter is greater than the former.

Impact

High. Deposits would be blocked if pending obligations exceed available unlocked balance.

Recommendation

Check if `unlocked` is greater than `totalUnlockObligations` before relocking assets.

```
    uint256 unlocked = _unlockAvailable();
+   if (unlocked > totalUnlockObligations) {
+     unchecked {
        _lock(unlocked - totalUnlockObligations);
+     }
+   }
```

[H-5] Expired locks in vlCVX can be kicked

Summary

Idle CVX locks can eventually be kicked out by a third party, causing the CVX to be sent back to the Votium strategy carrying a potential penalty.

Details

Locks in the CVXLocker contract can expire if they sit for too long. The contract includes a `kickExpiredLocks()` function that allows any account to kick expired locks if these exceed a certain threshold (4 epochs).

```
function kickExpiredLocks(address _account) external nonReentrant {
    //allow kick after grace period of 'kickRewardEpochDelay'
    _processExpiredLocks(_account, false, 0, _account, msg.sender, rewardsDuration.mul
}
```

When this happens, CVX is sent back to the owner, and a penalty is taken that is paid as an incentive to the kicker.

Impact

High. CVX will be sent back to the VotiumStrategy contract and will need to be relocked. Any applied penalty may break the assumption that CVX balance owned by the strategy can never decrease, potentially causing issues with queued withdrawals, as these rely on specific amounts of CVX to be executed.

Recommendation

- Provide a way in VotiumStrategy to relock balances outside of the deposit/withdraw workflows.
- Monitor for expired locks and relock if necessary before these can be kicked out.
- Ensure the strategy can eventually take losses.

Medium Findings

[M-1] Withdrawals can be blocked if vlCVX is shutdown

Summary

Eager relocking as part of withdrawals in the Votium strategy can cause a denial of service if the CvxLocker contract is shutdown.

Details

When withdrawals are requested in VotiumStrategy, the implementation of `requestWithdraw()` will check if the available unlocked balance is enough for instant withdrawal. Doing so will also relock any excess of CVX:

```
172:         if (unlockedCvx > totalUnlockObligations) {
173:             ethOutNow = unsafeSellCvx(cvxAmount);
174:             unchecked {
175:                 _lock(unlockedCvx - totalUnlockObligations);
176:             }
```

If the CVXLocker contract is shutdown, relocking will cause a revert, leading to a denial of service in the withdrawal.

Regression of code-423n4/2023-09-asymmetry-findings#50

Impact

Medium. Withdrawals can be blocked in the unlikely case that the Convex vault is shutdown.

Recommendation

Check if vlCVX is shutdown before relocking assets.

Low Findings

[L-1] `AfEth::deposit()` can revert if the deposit value in Votium is zero

In the `deposit()` function, if the calculated Votium share results in a zero amount, the action is still executed.

The implementation of `deposit()` in VotiumStrategy will try to execute a swap using a zero amount, which will revert in the Curve pool, causing a denial of service.

It is recommended to add a check to skip the deposit if the given amount is zero.

[L-2] Missing storage gap in TrackedAllowances

The TrackedAllowances contract is used as a base contract of an upgradeable contract (VotiumStrategy) and lacks any storage gap to eventually deal with future needs if upgraded.

[L-3] Conflicting management for quick actions

In `depositForQuickActions()`, the implementation will default to the caller's balance when `afEthAmount` is zero. Since the intention is to also allow ETH deposits (as it is a payable function), this will create a conflict if the owner wants to only deposit ETH funds. Setting `afEthAmount = 0` will also pull any AfEth in their account.

Similarly, in `withdrawOwnerFunds()`, the owner cannot withdraw just ETH, since setting `afEthAmount == 0` will also transfer all AfEth present in the contract, and cannot withdraw only AfEth, as passing `ethAmount == 0` will send all ETH owed to the owner that is present in the contract.

[L-4] Inaccurate strict comparisons in VotiumStrategy

The implementation of `deposit()` is segmented in two paths according to the condition if the CVX deposit is enough to cover the pending obligations.

```
135:         if (cvxAmount > totalUnlockObligations) {
136:             _processExpiredLocks(true);
```

```
137:             unchecked {
138:                 uint256 netExtraLock = cvxAmount - totalUnlockObligations;
139:                 if (netExtraLock > 0) _lock(netExtraLock);
140:             }
141:         } else {
```

The condition in line 135 should be using a greater than or equal operator, to cover the case where the amounts match.

Similarly, in `requestWithdraw()`, instant withdrawals can be executed when the available unlocked balance is greater than or equal to the pending obligations (line 172).

```
172:         if (unlockedCvx > totalUnlockObligations) {
173:             ethOutNow = unsafeSellCvx(cvxAmount);
174:             unchecked {
175:                 _lock(unlockedCvx - totalUnlockObligations);
176:             }
177:         } else {
```

[L-5] Withdrawals can be bricked when redeeming zero shares from the sfrxETH strategy

The sfrxETH vault follows solmate's ERC4626 implementation, which guards against zero amount withdrawals and reverts the call.

```
095:     function redeem(
096:         uint256 shares,
097:         address receiver,
098:         address owner
099:     ) public virtual returns (uint256 assets) {
100:         if (msg.sender != owner) {
101:             uint256 allowed = allowance[owner][msg.sender]; // Saves gas for limi
102: 
103:             if (allowed != type(uint256).max) allowance[owner][msg.sender] = allo
104:         }
105: 
106:         // Check for rounding error since we round down in previewRedeem.
107:         require((assets = previewRedeem(shares)) != 0, "ZERO_ASSETS");
```

This could cause an accidental denial of service in AfEth if zero shares withdrawals are executed in the sfrxETH strategy. Note also that shares could be rounded down in the calculation of `previewRedeem()`.

```
    function withdraw(uint256 withdrawShare) internal returns (uint256 ethOut) {
        uint256 sfrxEthAmount = availableBalance().mulWad(withdrawShare);
+       if (SFRX_ETH.previewRedeem(sfrxEthAmount) > 0) {
```

```
            uint256 frxEthAmount = SFRX_ETH.redeem(sfrxEthAmount, address(this), addre
            ethOut = _unsafeSellFrxEth(frxEthAmount);
+       }
    }
```

[L-6] Queued withdrawals can be overwritten when requesting a zero amount withdrawal

Queued withdrawals in VotiumStrategy follow a clever design in which the cumulative CVX unlock obligations is both used as a key and the threshold to determine when the withdrawal can be executed. This is done in `requestWithdraw()` by storing this information in the `withdrawableAfterUnlocked` mapping:

```
178:             locked = true;
179:             cumulativeUnlockThreshold = uint128(cumCvxUnlockObligations) + cvxAmo
180:             // Don't have to worry about withdrawable being overwritten as
181:             // `cumulativeCvxUnlockObligations` is stritcly increasing. Only edge
182:             // withdrawals with a `cvxAmount` of 0 in which case you'd be overwri
183:             withdrawableAfterUnlocked[to][cumulativeUnlockThreshold] = cvxAmount;
184:             cumulativeCvxUnlockObligations = uint128(cumulativeUnlockThreshold);
```

This logic contains an edge case when a user requests a second withdrawal of zero CVX after queuing a first request of a positive amount. The first request will write `withdrawableAfterUnlocked[user][cumulativeUnlockThreshold] = initialCvxAmount`, but the second will overwrite it since the keys will be the same, `cumulativeUnlockThreshold` remains the same as the previous value will be incremented by zero (line 179).

The issue is mitigated mainly by the zero share check at the start of the function, however it is also recommended to add a check to guard against the unlikely case of `cvxAmount` being zero.

[L-7] Reward swapping should ensure no CVX held for obligations is traded

The VotiumStrategy contract may hold CVX tokens owed to obligations of queued withdrawals. These tokens will sit at the contract until claimed, and won't be relocked in Convex.

Rewards coming from Votium or Convex are handled by a rewarder entity that is allowed to execute a shallow swap operation. It is important to ensure that CVX held for obligations is not swapped here, else the invariant could be broken.

Note that Votium rewards can be issued as CVX tokens, which means that the contract can potentially hold two different balances of CVX. Special care must be taken to differentiate the CVX dedicated to pending obligations, from the one coming from eventual Votium rewards.

Informational Findings

[I-1] Locked withdrawals can still be executed after emergency pause

Once the emergency pause is triggered, all functionality in AfEth and VotiumStrategy is paused and becomes inaccessible, except for `withdrawLocked()`.

Since this function is used to execute withdrawals already queued, it is unclear if this a deliberate action or a potential miss.

[I-2] Rewards can be externally claimed

Both Votium and Convex rewards can be claimed by anyone on behalf of another account. This means that the effects of `claimRewards()`, which are accessible only to the rewarder role, can actually be executed by directly interfacing with Convex or Votium.

[I-3] Missing events for parameter change

The initializer of AfEth should emit events to notify the initial rewarder and the strategy share.

[I-4] Payable deposit function takes an amount parameter

In VotiumStrategy, `deposit(uint256,uint256)` is a `payable` function, but handles the transferred value using an `amount` parameter. Consider removing this parameter and using `msg.value`.

[I-5] Function not used internally should have `external` visibility

Occurrences:

- `depositRewardsAndRebalance()`
- `quickDeposit(uint256,uint256)`
- `quickWithdraw(uint256,uint256,uint256)`

Gas Findings

[G-1] Access control is checked twice in `VotiumStrategy::deposit()`

The `deposit()` function delegates its implementation to `deposit(uint256,uint256)`. Both of these functions have the `onlyManager` modifier, causing the check to be executed twice.

Consider removing the modifier from `deposit()`, or extracting shared logic into an internal function.

[G-2] Duplicate check for zero in `VotiumStrategy::deposit()`

The implementation of `deposit()` checks if `netExtraLock` is greater than zero before calling `_lock()`. However, the implementation of `_lock()` does the same check internally.

[G-3] Check before sending ETH is always satisfiable in `VotiumStrategy::withdrawLocked()`

The `withdrawLocked()` function first checks that `ethReceived` is greater than zero before executing the transfer.

```
223:         if (minOut == 0) {
224:             CVX.safeTransfer(msg.sender, cvxAmount);
225:         } else {
226:             ethReceived = unsafeSellCvx(cvxAmount);
227:             if (ethReceived < minOut) revert ExchangeOutputBelowMin();
228:
229:             if (ethReceived > 0) msg.sender.safeTransferETH(ethReceived);
230:         }
```

However, this condition should always be true, since `minOut != 0` (line 223) and `ethReceived > minOut` (line 227).

[G-4] Chainlink response validation can be simplified

The implementation of `ethCvxPrice()` does a series of checks to validate the sanity of the Chainlink feed response.

```
27:     function ethCvxPrice() internal view returns (uint256) {
28:         (uint80 roundId, int256 answer, /* startedAt */, uint256 updatedAt, /* ans
29:             CVX_ETH_ORACLE.latestRoundData();
30:
31:         if (roundId == 0 || answer < 0 || updatedAt == 0) revert InvalidOracleData
32:
33:         if (block.timestamp - updatedAt > ORACLE_STALENESS_WINDOW) revert OracleDa
34:
35:         return uint256(answer);
36:     }
```

The `updatedAt == 0` check part of the validations in line 31 can be skipped as it will be guaranteed by the staleness check in line 33.