xETH - Versus Contest Findings & Analysis Report

2023-08-15

Table of contents

Overview

About C4

Code4rena (C4) is an open organization consisting of security researchers, auditors, developers, and individuals with domain expertise in smart contracts.

A C4 audit is an event in which community participants, referred to as Wardens, review, audit, or analyze smart contract logic in exchange for a bounty provided by sponsoring projects.

During the audit outlined in this document, C4 conducted an analysis of the xETH smart contract system written in Solidity. The audit took place between May 12—May 15 2023.

Following the C4 audit, 3 wardens (bin2chen, d3e4 and ronnyx2017) reviewed the mitigations for all identified issues; the mitigation review report is appended below

the audit report.

Wardens

In Code4rena's Invitational audits, the competition is limited to a small group of wardens; for this audit, 5 wardens contributed reports:

1. adriro
2. bin2chen
3. carlitox477
4. d3e4
5. ronnyx2017

This audit was judged by kirk-baird.

Final report assembled by thebrittfactor.

Summary

The C4 analysis yielded an aggregated total of 10 unique vulnerabilities. Of these vulnerabilities, 0 received a risk rating in the category of HIGH severity and 10 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 5 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 4 reports recommending gas optimizations.

All of the issues presented here are linked back to their original finding.

Scope

The code under review can be found within the C4 xETH repository, and is composed of 4 smart contracts written in the Solidity programming language and includes 636 lines of Solidity code.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities based on three primary risk categories: high, medium, and low/non-critical.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges
- Arithmetic
- Gas use

For more information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on the C4 website, specifically our section on Severity Categorization.

Medium Risk Findings (10)

[M-01] Rebalance amounts should be checked so that updated balances falls within thresholds

Submitted by adriro, also found by d3e4

https://github.com/code-423n4/2023-05-xeth/blob/main/src/AMO2.sol#L239
https://github.com/code-423n4/2023-05-xeth/blob/main/src/AMO2.sol#L286

Rebalance operations are allowed when the current percentage of xETH in the Curve pool is outside the defined thresholds. However, there is no check to ensure that the amount of xETH added or burned from the pool will update the percentage to be within the limits.

Rebalance actions in the AMO contract are first validated to ensure that a rebalance is indeed needed in the first place. This is implemented by calculating the percentage of xETH in the pool and comparing the result against two limits that define an acceptable range.

Rebalance up is allowed when the percentage of xETH in the Curve pool is above the `REBALANCE_UP_THRESHOLD`, in which case xETH is withdrawn from the pool and burned to lower the percentage. Similarly, rebalance down is allowed when the percentage falls below the `REBALANCE_DOWN_THRESHOLD` value, in which case xETH is minted and added to the pool, increasing the percentage.

While care is taken to ensure that these operations are indeed only allowed when the percentages are outside the defined bounds, there is no check to ensure that the amount of xETH burned or minted will update the percentage **to be within the bounds.**

Proof of Concept

During a rebalance operation, a malicious or improperly operating defender can exploit the opportunity to modify the pool balance in their favor, potentially exceeding the desired boundaries.

For example, if `xEthPct > REBALANCE_UP_THRESHOLD` is true, the `preRebalanceCheck()` function would allow a `rebalanceUp()` operation. In such a scenario, the defender could deliberately lower the percentage to below the desired `REBALANCE_DOWN_THRESHOLD`, potentially gaining an advantage if the account is compromised or if there is a malfunction in the bot operating off-chain.

Recommendation

Add a check to ensure that updated balances comply with the thresholds boundaries. When minting or burning xETH from the pool, the updated balance of xETH should produce a percentage that is within the accepted range.

Assessed type

Invalid Validation

vaporkane (xETH) acknowledged

[M-02] Inconsistent check for LP balance in AMO

Submitted by adriro*, also found by* ronnyx2017 *and* bin2chen

https://github.com/code-423n4/2023-05-xeth/blob/main/src/AMO2.sol#L256-L259
https://github.com/code-423n4/2023-05-xeth/blob/main/src/AMO2.sol#L600-L604

https://github.com/code-423n4/2023-05-xeth/blob/main/src/AMO2.sol#L632-L636

While pulling LP tokens from the CVXStaker contract, the AMO queries the current available balance using the staked balance, which is inconsistent with the implementation of the withdraw function.

Curve LP tokens owned by the AMO contract are staked in a Convex pool that is handled using the CVXStaker contract. When liquidity needs to be removed from the Curve pool, the AMO contract needs to first withdraw the LP tokens from the CVXStaker contract.

The `rebalanceUp()`, `removeLiquidity()` and `removeLiquidityOnlyStETH()` functions present in the AMO contract deal with removing liquidity from the Curve pool. In their implementations, all of them query the available LP balance using the `stakedBalance()` function of CVXStaker. Taking the `rebalanceUp()` function as an example (other cases are similar), we can see the following:

https://github.com/code-423n4/2023-05-xeth/blob/main/src/AMO2.sol#L256-L261

```
...
256:         uint256 amoLpBal = cvxStaker.stakedBalance();
257:
258:         // if (amoLpBal == 0 || quote.lpBurn > amoLpBal) re
259:         if (quote.lpBurn > amoLpBal) revert LpBalanceTooLow
260:
261:         cvxStaker.withdrawAndUnwrap(quote.lpBurn, false, ac
...
```

The implementation of `stakedBalance()` basically delegates the call to fetch the staked balance in the Convex reward pool contract:

https://github.com/code-423n4/2023-05-xeth/blob/main/src/CVXStaker.sol#L204-L206

```
204:     function stakedBalance() public view returns (uint256 b
205:         balance = IBaseRewardPool(cvxPoolInfo.rewards).bala
```

```
206:     }
```

However, this check is not correct. As we can see in the implementation of `withdrawAndUnwrap()`, the CVXStaker contract consider not only staked tokens, **but also available balance held in the contract itself:**

https://github.com/code-423n4/2023-05-xeth/blob/main/src/CVXStaker.sol#L142-L161

```
142:     function withdrawAndUnwrap(
143:         uint256 amount,
144:         bool claim,
145:         address to
146:     ) external onlyOperatorOrOwner {
147:         // Optimistically use CLP balance in this contract,
148:         uint256 clpBalance = clpToken.balanceOf(address(thi
149:         uint256 toUnstake = (amount < clpBalance) ? 0 : amo
150:         if (toUnstake > 0) {
151:             IBaseRewardPool(cvxPoolInfo.rewards).withdrawAr
152:                 toUnstake,
153:                 claim
154:             );
155:         }
156: 
157:         if (to != address(0)) {
158:             // unwrapped amount is 1 to 1
159:             clpToken.safeTransfer(to, amount);
160:         }
161:     }
```

Line 148 considers potentially available LP tokens in the contract and withdraws the remaining amount from Convex.

This means that checking against `stakedBalance()` is too restrictive and incorrect, and can potentially lead to situations in which the required LP tokens are enough, but the check in line 259 of the AMO contract will revert the operation.

Proof of Concept

As an example, let's take a call to the `rebalanceUp()` function and assume that the quoted `lpBurn` amount is 4. The CVXStaker contract has 3 LP tokens staked in Convex and 2 tokens held as balance in the contract.

In this situation, the condition `quote.lpBurn > amoLpBal` will be true, as `amoLpBal = cvxStaker.stakedBalance() = 3`, which evaluates the condition to `5 > 3`, causing a revert in the transaction.

However, the operation would succeed if the check weren't there, as `withdrawAndUnwrap()` will first consider the 2 tokens already present in the CVXStaker contract and withdraw the remaining 2 from the Convex pool, successfully fulfilling the requested amount.

Recommendation

The validation to ensure available LP tokens in `rebalanceUp()`, `removeLiquidity()` and `removeLiquidityOnlyStETH()` should not only consider `stakedBalance()`, but also available LP tokens present in the CVXStaker contract. Alternatively, the check can be removed as the call to `withdrawAndUnwrap()` will eventually fail if the available tokens are not enough.

Assessed type

Invalid Validation

vaporkane (xETH) confirmed

xETH mitigated:

> This mitigation adds a `getTotalBalance` function.

> PR: https://github.com/code-423n4/2023-05-xeth/commit/60b9e1e2562d585831e3d8e2a7e345294d9dacbd

Status: Mitigation confirmed. Full details in reports from d3e4, bin2chen and ronnyx2017.

[M-03] Zero token transfer can cause a potential DoS in CVXStaker

Submitted by adriro*, also found by* bin2chen

The CVXStaker contract doesn't check for zero amount while transferring rewards, which can end up blocking the operation.

The CVXStaker contract is in charge of handling interaction with the Convex pool. The `getReward()` function is used to claim rewards and transfer them to the rewards recipient:

https://github.com/code-423n4/2023-05-xeth/blob/main/src/CVXStaker.sol#L185-L198

```
185:     function getReward(bool claimExtras) external {
186:         IBaseRewardPool(cvxPoolInfo.rewards).getReward(
187:             address(this),
188:             claimExtras
189:         );
190:         if (rewardsRecipient != address(0)) {
191:             for (uint i = 0; i < rewardTokens.length; i++)
192:                 uint256 balance = IERC20(rewardTokens[i]).b
193:                     address(this)
194:                 );
195:                 IERC20(rewardTokens[i]).safeTransfer(rewarc
196:             }
197:         }
198:     }
```

As we can see in the previous snippet of code, the implementation will loop all the configured reward tokens and transfer them one by one to the reward recipient.

This is a bit concerning as some ERC20 implementations revert on zero value transfers (see https://github.com/d-xo/weird-erc20#revert-on-zero-value-transfers). If at least one of the reward tokens includes this behavior, then the current implementation may cause a denial of service, as a zero amount transfer in this token will block the whole action and revert the transaction.

Note that the rewards array is not modifiable, it is defined at construction time, a token cannot be removed.

Proof of Concept

We reproduce the issue in the following test. `token1` is a normal ERC20 and `token2` reverts on zero transfer amounts. Rewards from `token1` can't be transferred to the recipient as the zero transfer on `token2` will revert the operation.

Note: the snippet shows only the relevant code for the test. Full test file can be found here.

```
function test_CVXStaker_RevertOnZeroTokenTransfer() public {
    MockErc20 token1 = new MockErc20("Token1", "TOK1", 18);
    MockErc20 token2 = new RevertOnZeroErc20("Token2", "TOK2", 1

    MockCVXRewards rewards = new MockCVXRewards();

    address operator = makeAddr("operator");
    IERC20 clpToken = IERC20(makeAddr("clpToken"));
    ICVXBooster booster = ICVXBooster(makeAddr("booster"));
    address[] memory rewardTokens = new address[](2);
    rewardTokens[0] = address(token1);
    rewardTokens[1] = address(token2);

    CVXStaker staker = new CVXStaker(operator, clpToken, booster
    staker.setCvxPoolInfo(0, address(clpToken), address(rewards)

    address rewardsRecipient = makeAddr("rewardsRecipient");
    staker.setRewardsRecipient(rewardsRecipient);

    // simulate staker has some token1 but zero token2 after cal
    deal(address(token1), address(staker), 1e18);

    // The transaction will fail as the implementation will try
    // tokens for token2, blocking the whole operation.
    vm.expectRevert("cannot transfer zero amount");
    staker.getReward(true);
}
```

Recommendation

Check for zero amount before executing the transfer:

```
    function getReward(bool claimExtras) external {
        IBaseRewardPool(cvxPoolInfo.rewards).getReward(
            address(this),
            claimExtras
        );
        if (rewardsRecipient != address(0)) {
            for (uint i = 0; i < rewardTokens.length; i++) {
                uint256 balance = IERC20(rewardTokens[i]).balanceOf(
                    address(this)
                );

+               if (balance > 0) {
                  IERC20(rewardTokens[i]).safeTransfer(rewardsRecipi
+               }
            }
        }
    }
```

Assessed type

ERC20

vaporkane (xETH) confirmed

xETH mitigated:

> This mitigation adds a balance check.

> PR: https://github.com/code-423n4/2023-05-xeth/commit/1f714868f193cdeb472ec097110901a997d87ec4

Status: Mitigation confirmed. Full details in reports from ronnyx2017, bin2chen and d3e4.

[M-04] Unspent allowance may break functionality in AMO

Submitted by adriro

https://github.com/code-423n4/2023-05-xeth/blob/main/src/AMO2.sol#L545
https://github.com/code-423n4/2023-05-xeth/blob/main/src/AMO2.sol#L546
https://github.com/code-423n4/2023-05-xeth/blob/main/src/AMO2.sol#L571

An unspent allowance may cause a denial of service during the calls to `safeApprove()` in the AMO contract.

The AMO contract uses the `safeApprove()` function to grant the Curve pool permission to spend funds while adding liquidity. When adding liquidity into the Curve pool, the AMO contract needs to approve allowance so the AMM can pull tokens from the caller.

The `safeApprove()` function is a wrapper provided by the SafeERC20 library present in the OpenZeppelin contracts package, its implementation is the following:

https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/utils/SafeERC20.sol#L45-L54

```
    function safeApprove(IERC20 token, address spender, uint256 valu
        // safeApprove should only be called when setting an initial
        // or when resetting it to zero. To increase and decrease it
        // 'safeIncreaseAllowance' and 'safeDecreaseAllowance'
        require(
            (value == 0) || (token.allowance(address(this), spender)
            "SafeERC20: approve from non-zero to non-zero allowance'
        );
        _callOptionalReturn(token, abi.encodeWithSelector(token.appr
    }
```

As the comment warns, this should only be used when setting an initial balance or resetting it to zero. In the AMO contract the use of `safeApprove()` is included in the functions that are in charge of adding liquidity to the Curve pool (`addLiquidity()` and `addLiquidityOnlyStETH()`), implying a repeatedly use whenever the allowance needs to be set so that the pool can pull the funds. As we can see in the implementation, if the current allowance is not zero the function will revert.

This means that any unspent allowance of xETH or stETH (i.e. `allowance(AMO, curvePool) > 0`) will cause a denial of service in the `addLiquidity()` and `addLiquidityOnlyStETH()` functions, potentially bricking the contract.

Proof of Concept

1. Suppose there is an unspent allowance of stETH in the AMO contract, `stETH.allowance(AMO, curvePool) > 0`.
2. Admin calls `addLiquidityOnlyStETH(stETHAmount, minLpOut)` with `stETHAmount > 0` to provide liquidity to the pool.
3. Transaction will be reverted in the call to `safeApprove()` as `(value == 0) || (token.allowance(address(this), spender) == 0)` will be false.

Recommendation

Simply use `approve()`, or first reset the allowance to zero using `safeApprove(spender, 0)`, or use `safeIncreaseAllowance()`.

Note from warden

Even though the deprecated usage of `safeApprove()` is mentioned in the automated findings, this report demonstrates how this function can cause a serious vulnerability that may end up bricking the contract.

Assessed type

DoS

vaporkane (xETH) confirmed

xETH mitigated:

> Change `safeApprove` to approve.

> PR: https://github.com/code-423n4/2023-05-xeth/commit/793dade5217bd5751856f7cf0bccd4936286aeab

Status: Mitigation confirmed. Full details in reports from ronnyx2017, bin2chen and d3e4.

[M-05] Virgin stake can claim all drops

Submitted by d3e4, also found by ronnyx2017

If wxETH drips when nothing is staked, then the first staker can claim every drop.

Proof of Concept

Suppose drip is enabled when `totalSupply() == 0`. At least one block passes and the first staker stakes, just `1` xETH is enough. This mints her `1` wxETH. This also calls `_accrueDrip()` (by the `drip` modifier) which drips some amount of xETH. Note that `_accrueDrip()` is independent of `totalSupply()`, so it doesn't matter how little she stakes. `cashMinusLocked` is now `1` plus the amount dripped. Naturally, since she owns the entire supply she can immediately unstake the entire `cashMinusLocked`. Specifically, the `exchangeRate()` is now `(cashMinusLocked * BASE_UNIT) / totalSupply()` and she gets `(totalSupply() * exchangeRate()) / BASE_UNIT = cashMinusLocked`.

The issue is simply that drip is independent of staked amount; especially since it may drip even when nothing is staked, which enables the above attack.

Recommended Mitigation Steps

Note what happens when `totalSupply() > 0`. Then drops will fall on existing wxETH, and any new staker will trigger a drip before having to pay the new exchange rate which includes the extra drops. So a new staker, in this case, cannot unstake more than they staked; all drops go to previous holders. Therefore, do not drip when `totalSupply == 0`; in `_accrueDrip()`:

```
-if (!dripEnabled) return;
+if (!dripEnabled || totalSupply() == 0) return;
```

Assessed type

MEV

vaporkane (xETH) disagreed with severity

kirk-baird (judge) decreased severity to Medium and commented:

> It is true that the first staker may claim all previous dripped rewards. I consider this a medium severity issue as the likelihood is very low since it only occurs for the first staker and only if the strip was started before any stakers deposited.

vaporkane (xETH) confirmed

xETH mitigated:

> Add a `totalSupply` check.

> PR: https://github.com/code-423n4/2023-05-xeth/commit/aebc3244cbb0deb67f3cdef160b390da27888de7

Status: Not fully mitigated. Full details in reports from ronnyx2017, bin2chen and d3e4, and also included in the Mitigation Review section below.

[M-06] Inflation attack by token transfer

Submitted by d3e4*, also found by* adriro *and* bin2chen

The first staker can inflate the exchange rate by transferring tokens directly to the contract such that subsequent stakers get minted zero wxETH. Their stake can then be unstaked by the first staker, together with their own first stake and inflation investment. Effectively, the first staker can steal the second stake. The attack exploits the rounding error in tokens minted, caused by the inflation. This vulnerability has a more general impact than just described, in that stakes might be partially stolen and that it also affects further stakers down the line, but the below example demonstrates the basic case.

Proof of Concept

Alice is the first staker, so `totalSupply() == 0`. She stakes `1` xETH by calling `stake(1)`.

```
    function stake(uint256 xETHAmount) external drip returns (uint25
        /// @dev calculate the amount of wxETH to mint
        uint256 mintAmount = previewStake(xETHAmount);

        /// @dev transfer xETH from the user to the contract
        xETH.safeTransferFrom(msg.sender, address(this), xETHAmount)

        /// @dev emit event
        emit Stake(msg.sender, xETHAmount, mintAmount);

        /// @dev mint the wxETH to the user
        _mint(msg.sender, mintAmount);

        return mintAmount;
    }
```

and gets minted `previewStake(1)`

```
    function previewStake(uint256 xETHAmount) public view returns (u
        /// @dev if xETHAmount is 0, revert.
        if (xETHAmount == 0) revert AmountZeroProvided();

        /// @dev calculate the amount of wxETH to mint before transf
        return (xETHAmount * BASE_UNIT) / exchangeRate();
    }
```

which thus is `1 * BASE_UNIT / exchangeRate()`, where

```
    function exchangeRate() public view returns (uint256) {
        /// @dev if there are no tokens minted, return the initial e
        uint256 _totalSupply = totalSupply();
        if (_totalSupply == 0) {
            return INITIAL_EXCHANGE_RATE;
        }

        /// @dev calculate the cash on hand by removing locked funds
        /// @notice this balanceOf call will include any lockedFunds
```

```
        /// @notice as the locked funds are also in xETH
        uint256 cashMinusLocked = xETH.balanceOf(address(this)) - lo

        /// @dev return the exchange rate by dividing the cash on ha
        return (cashMinusLocked * BASE_UNIT) / _totalSupply;
    }
```

so this works out to `1 * BASE_UNIT / INITIAL_EXCHANGE_RATE`. This is what Alice is minted and also the new `totalSupply()`.

Bob is about to stake `b` xETH (which Alice can frontrun).

Before Bob's stake, Alice transfers `a` xETH directly to the wxETH contract. The xETH balance of wxETH is now `1 + a + lockedFunds`; `1` from Alice's stake, `a` from her token transfer and whatever `lockedFunds` may have been added.

Now Bob stakes his `b` xETH by calling `stake(b)`. By reference to the above code, this mints him `previewStake(b)`, which is `n * BASE_UNIT / exchangeRate()`. This time `totalSupply() > 0` so `exchangeRate()` this time is `(1 + a) * BASE_UNIT / (1 * BASE_UNIT / INITIAL_EXCHANGE_RATE)`, which simplifies to `(1 + a) * INITIAL_EXCHANGE_RATE`. So Bob gets minted `b * BASE_UNIT / ((1 + a) * INITIAL_EXCHANGE_RATE)`. This may clearly be `< 1` which is therefore rounded down to `0` in Solidity. `BASE_UNIT` and `INITIAL_EXCHANGE_RATE` are both set to `1e18`, so Bob is minted `b/(1 + a)` tokens. In that case, we simply have that if `a >= b` then Bob is minted `0` wxETH.

Now note in `stake()` above that whenever a non-zero amount is staked, those funds are transferred to the contract (even if nothing is minted).

Bob cannot unstake anything with `0` wxETH, so he has lost his `b` xETH.

`totalSupply()` is now `1` and the xETH balance of wxETH is `1 + a + b + lockedFunds`. `exchangeRate()` is therefore `(1 + a + b) * BASE_UNIT / 1`.

Alice owns the `1` wxETH ever minted so if she unstakes it

```
    function unstake(uint256 wxETHAmount) external drip returns (uir
        /// @dev calculate the amount of xETH to return
        uint256 returnAmount = previewUnstake(wxETHAmount);

        /// @dev emit event
        emit Unstake(msg.sender, wxETHAmount, returnAmount);

        /// @dev burn the wxETH from the user
        _burn(msg.sender, wxETHAmount);

        /// @dev return the xETH back to user
        xETH.safeTransfer(msg.sender, returnAmount);

        /// @dev return the amount of xETH sent to user
        return returnAmount;
    }
```

she gets `previewUnstake(1)`

```
    function previewUnstake(uint256 wxETHAmount) public view returns
        /// @dev if wxETHAmount is 0, revert.
        if (wxETHAmount == 0) revert AmountZeroProvided();

        /// @dev calculate the amount of xETH to return
        return (wxETHAmount * exchangeRate()) / BASE_UNIT;
    }
```

which thus is `1 * (1 + a + b) * BASE_UNIT / BASE_UNIT == (1 + a + b)`.

That is, Alice gets both hers and Bob's stakes.

Recommended Mitigation Steps

Do not use `xETH.balanceOf(address(this))` when calculating the funds staked. Account only for funds transferred through `stake()` by keeping an internal accounting of the balance. Consider implementing a sweep function to access any unaccounted funds, or use them as locked funds if free (but unlikely) funds would be accepted as such.

Assessed type

ERC4626

vaporkane (xETH) disagreed with severity

kirk-baird (judge) decreased severity to Medium

vaporkane (xETH) confirmed

[M-07] Incorrect slippage check in the AMO2.rebalanceUp can be attacked by MEV

Submitted by ronnyx2017

https://github.com/code-423n4/2023-05-xeth/blob/main/src/AMO2.sol#L323-L325
https://github.com/code-423n4/2023-05-xeth/blob/main/src/AMO2.sol#L335-L351

The `AMO2.rebalanceUp` uses `AMO2.bestRebalanceUpQuote` function to avoid MEV attack when removing liquidity with only one coin. But the `bestRebalanceUpQuote` does not calculate the slippage correctly in this case, which is vulnerable to be attacked by MEV sandwich.

Proof of Concept

`AMO2.rebalanceUp` can be executed successfully only if the `preRebalanceCheck` returns true.

```
bool isRebalanceUp = preRebalanceCheck();
if (!isRebalanceUp) revert RebalanceUpNotAllowed();
```

So as the logic in the `preRebalanceCheck`, if `isRebalanceUp = ture`, the token balances in the curve should satisfy the following condition

```
xETHBal / (stETHBal + xETHBal) > REBALANCE_UP_THRESHOLD
```

The default value of `REBALANCE_UP_THRESHOLD` is 0.75 :

```
uint256 public REBALANCE_UP_THRESHOLD = 0.75E18;
```

It's always greater than 0.5. So when removing liquidity for only xETH, which is the `rebalanceUp` function actually doing, the slippage will be positive instead of negative. You will receive more than 1 uint of xETH when you burn the lp token of 1 virtual price.

But the slippage calculation is incorrect in the `bestRebalanceUpQuote` :

```
// min received caculate in bestRebalanceUpQuote
    bestQuote.min_xETHReceived = applySlippage(
        (vp * defenderQuote.lpBurn) / BASE_UNIT
    );

// applySlippage function
function applySlippage(uint256 amount) internal view returns
    return (amount * (BASE_UNIT - maxSlippageBPS)) / BASE_UN
}
```

It uses the target amount subtracted slippage as the min token amount received. It is equivalent to expanding the slippage in the opposite direction. It makes the `contractQuote` can't protect the `defenderQuote` , increasing the risk of a large sandwich.

Recommended Mitigation Steps

`rebalanceUp` and `rebalanceDown` should use different slippage calculation methods in the `applySlippage` .

Assessed type

MEV

d3e4 (warden) commented:

> Whether you will receive more or less xETH per LP token is accounted for in the price `vp`. `(vp * defenderQuote.lpBurn) / BASE_UNIT` is the amount of xETH received when burning `lpBurn` LP tokens. We want to receive as much xETH as possible, but at least `applySlippage` of that, e.g. 99%. It doesn't matter what direction we are exchanging; we always want to receive as much as possible for whatever fixed amount we provide. So the slippage should always be calculated as less than 100% of the ideal amount received.

vaporkane (xETH) disputed

kirk-baird (judge) commented:

> I agree with the comments that slippage is calculated as a percentage of the received token. This is the desirable behaviour and therefore this issue is invalid.

ronnyx2017 (warden) commented:

> @d3e4 - In fact, the price `vp` only accounts the token amounts when the peg pool is balanced, which means stETH:xEth is 1:1 in the pool. The case in the comment is from the point of a defi user instead of a rebalance mechanism. The `REBALANCE_UP_THRESHOLD` = 0.75 , and `REBALANCE_DOWN_THRESHOLD` = 0.68 . So when calling `rebalanceUp` , there must be more than 75% xETH in the pool and after the rebalance, there should be more than 68% xETH in the pool. Otherwise, the pool is unbalanced again. So use `vp` minus slippage as the contractQuote, the rebalance up can reduce the xETH proportion to lower than 50%, which makes the pool fall into the rebalance down process.

ronnyx2017 (warden) commented:

> Aha, I find a similar issue submitted by another warden https://github.com/code-423n4/2023-05-xeth-findings/issues/35. This issue is the root cause and solution for the https://github.com/code-423n4/2023-05-xeth-findings/issues/35.

> Actually, `rebalanceDown` can't leap the boundary about `REBALANCE_UP_THRESHOLD` if slippage is appropriate. Because the

> `rebalanceDown` mints more xETH and adds it to the pool. If it adds too much xETH, the lp minted will lower than the slippage allows.

> But for the `rebalanceUp`, the slippage protects nothing because of the incorrect calculation method.

kirk-baird (judge) commented:

> @vaporkane you're input would be valuable if that's okay.

> Okay so the first point I am taking out of this is:

> In fact, the price vp only accounts the token amounts when the peg pool is balance, which means stETH:xEth is 1:1 in the pool.

> Does `vp * defenderQuote.lpBurn / BASE_UNIT` always give us the correct value of `xETH` that would be received when burning `lpBurn` amount of LP tokens?

adriro (warden) commented:

> @ronnyx2017 - I can't see how this is linked to #35.

> When the pool is unbalanced you won't get **exactly** the virtual price amount when you redeem it as one coin. This would imply that there's always 1:1 relation which is not what curve does. But I think it is fair to use it as a slippage check, which is the sponsor's intention here.

> What the warden is trying to argue is that, let's say that virtual price is 1.05 so 1 LP should be redeemed as 1.05. Due to imbalance (rebalance up should require 75% of xETH using the default settings), if you redeem it as one coin (xETH) you won't actually get 1.05, you will get a bit more because you are removing the coin which has more liquidity. Note that we are talking about curve's stable pool model, where balance differences are eased.

> Note also that this is a safeguard measure against a defender supplied parameter. The calculation is only used if the argument provided by the off chain process ends up being lower than the quote from calculation, which means that the calculation is just a lower bound safety measure.

kirk-baird (judge) commented:

> Thanks for the feedback @adriro. To summarise this issue there is a miscalculation in the slippage protection which could allow for a small sandwich attack. The conditions are that a defender must send a transaction with the minimum xETH received to be significantly lower than the real price in which case the `applySlippage()` lower bound will be taken rather than the provided minimum. Since there is a bug in the `applySlippage()` calculations this may result in a sandwich attack.

> I'm going to reinstate this issue as medium severity as there is a small bug in the slippage protection which under niche circumstances could result in a sandwich attack.

> Noting, although related to #35 I do not see this as a direct duplicate since it is a slippage attack rather than over rebalancing.

adriro (warden) commented:

> @kirk-baird - I'm not sure I would call this a bug, as the intention is not to prevent MEV (which is already mitigated by the min amount) but to prevent a rogue or malfunctioning defender. Just making sure my previous comment intention is fully understood, I'll leave the final decision (severity) to you.

xETH mitigated:

> Partial: add admin controlled slippage values.

> PR: https://github.com/code-423n4/2023-05-xeth/commit/630114ea6a3782f2f539b5936c524f8da62d0179

Status: Partially mitigated. Full details in reports from ronnyx2017 (here and here) and d3e4, and also included in the Mitigation Review section below.

[M-08] CVXStaker.sol Unable to process newly add rewardTokens

Submitted by [bin2chen]

The lack of a mechanism to modify `rewardTokens[]`

If `convex` adds new `extraRewards`

`CVXStaker.sol` cannot transfer the added token

Proof of Concept

`CVXStaker.sol` will pass in `rewardTokens[]` in `constructor` and in `getReward()`, loop this array to transfer `rewardTokens`

```
    function getReward(bool claimExtras) external {
        IBaseRewardPool(cvxPoolInfo.rewards).getReward(
            address(this),
            claimExtras
        );
        if (rewardsRecipient != address(0)) {
            for (uint i = 0; i < rewardTokens.length; i++) { //<
                uint256 balance = IERC20(rewardTokens[i]).balanc
                    address(this)
                );
                IERC20(rewardTokens[i]).safeTransfer(rewardsReci
            }
        }
    }
```

The main problem is that this `rewardTokens[]` does not provide a way to modify it but it is possible to add a new `rewardsToken` in `convex`.

The following code is from `BaseRewardPool.sol` of `convex`

https://github.com/convex-eth/platform/blob/main/contracts/contracts/BaseRewardPool.sol#L238

```
    function addExtraReward(address _reward) external returns(bo
        require(msg.sender == rewardManager, "!authorized");
        require(_reward != address(0),"!reward setting");

        extraRewards.push(_reward);
```

```
            return true;
        }
```

This will result in a situation : if new `extraRewards` are added to `IBaseRewardPool` later on But since the `rewardTokens` of `CVXStaker` cannot be modified (e.g. added), then the new `extraRewards` cannot be transferred out of `CVXStaker`. After `IBaseRewardPool(cvxPoolInfo.rewards).getReward()`, the newly added token can only stay in the `CVXStaker` contract.

Recommended Mitigation Steps

Add a new method to modify `CVXStaker.rewardTokens[]`

Assessed type

Context

vaporkane (xETH) acknowledged

carlitox477 (warden) commented:

> Issue in QA #7 titled "CVXStaker::getReward: Possible DOS if rewardTokens is enough large" is a dup of this one

kirk-baird (judge) commented:

> @carlitox477 - This is not true. The issue in QA#7 is related to a DoS if there are too many rewards in claim rewards. This issue says if new rewards are added to convex they will not be paid out by the contract since the locally stored list is not updated.

vaporkane (xETH) confirmed

xETH mitigated:

> Add a `setRewardTokens` function.

> PR: https://github.com/code-423n4/2023-05-xeth/commit/1f714868f193cdeb472ec097110901a997d87ec4

Status: Mitigation confirmed with comments. Full details in reports from bin2chen, d3e4 and ronnyx2017.

[M-09] withdrawAllAndUnwrap() the clpToken transfer to AMO.sol may be locked in the contract

Submitted by bin2chen, also found by ronnyx2017

in `withdrawAllAndUnwrap()` the clpToken transfer to AMO.sol may be locked in the contract

Proof of Concept

`withdrawAllAndUnwrap()` You can specify `sendToOperator==true` to transfer the `clpToken` to `operator`

The code is as follows:

```
    function withdrawAllAndUnwrap(
        bool claim,
        bool sendToOperator
    ) external onlyOwner {
        IBaseRewardPool(cvxPoolInfo.rewards).withdrawAllAndUnwra
        if (sendToOperator) {
            uint256 totalBalance = clpToken.balanceOf(address(th
            clpToken.safeTransfer(operator, totalBalance); //<--
        }
    }
```

current protocols, `operator` is currently set to `AMO.sol` as normal

But `AMO.sol` doesn't have any way to use the transferred `clpToken` . The reason is that in AMO.sol, the method that transfers the `clpToken` , the number of transfers is from the newly generated `clpToken` from `curvePool`

It doesn't include `clpToken` that already exists in `AMO.sol` contract, for example (rebalanceDown/addLiquidity/addLiquidityOnlyStETH)

example `rebalanceDown` :

```
        function rebalanceDown(
            RebalanceDownQuote memory quote
        )
    ...

            lpAmountOut = curvePool.add_liquidity(amounts, quote.mir

            IERC20(address(curvePool)).safeTransfer(
                address(cvxStaker),
                lpAmountOut //<---------@audit this clpToken from cu
            );
            cvxStaker.depositAndStake(lpAmountOut);
```

So the `clpToken` transferred to 'AMO.sol' by `withdrawAllAndUnwrap()` will stay in the AMO contract and it is locked.

Recommended Mitigation Steps

modify `withdrawAllAndUnwrap()` , directly transfer to `msg.sender` .

Assessed type

Context

vaporkane (xETH) confirmed

xETH mitigated:

> Change `sendToOperator` to `sendToOwner` .

> PR: https://github.com/code-423n4/2023-05-xeth/commit/a840dc0b8a1de59a3ea06e0814ea3ce26707bdae

Status: Mitigation confirmed. Full details in reports from ronnyx2017, bin2chen and d3e4.

[M-10] First 1 wei deposit can produce loss of user xETH funds in `wxETH`

Submitted by carlitox477, also found by d3e4

The present implementation of the `wxETH::stake` functions permits the sending of tokens to the contract, even if the quantity of wxETH is zero. This can result in users losing funds, particularly when the initial deposit is only 1 wei, and the extent to which xETH is dripped (alongside its dripping period) is taken into consideration.

The issue arises when `xETHAmount * BASE_UNIT) < exchangeRate()` during the second deposit.

There is a potential risk of losing funds in certain feasible scenarios due to the incorrect handling of round-down situations.

Proof of Concept

Here a coded POC that shows what would happen if:

1. The protocol wants to drip 70 xETH in its first week as a way of promotion
2. Alice realized this and decided to make the first deposit (by frontrunning every other deposits)
3. The second deposit of $10416666666666601; wei$ happens 15 minutes after Alice's deposit

Here's the output of the coded POC:

```
Logs:
  Min amount to deposit to get a single share: 1041666666666666(
  Bob xETH before staking: 10416666666666600
  Bob wxETH after staking balance 0
```

Other scenarios

Here are the numbers of other scenario:

	1 eth in 1 year		10 eth in 1 year		100 eth in 1 year		1000 eth in 1 year	
Time until second deposit	Min amount to deposit (ETH)	Value(USD)	Min amount to deposit	Value(USD)	Min amount to deposit	Value(USD)	Min amount to deposit	Value (USD)
1 minute	0,000001 902587519	0,003 42465 7534	0,00001 902587 519	0,034 24657 534	0,00019 0258751 9	0,342 4657 534	0,00190 2587519	3,424 6575 34
15 minutes	0,000028 53881279	0,0513 69863 01	0,0002 8538812 79	0,5136 9863 01	0,00285 3881279	5,136 9863 01	0,02853 881279	51,36 9863 01
1 hour	0,0001141 552511	0,205 47945 21	0,001141 552511	2,054 79452 1	0,011415 52511	20,54 7945 21	0,114155 2511	205, 4794 521
24 hours	0,002739 726027	4,9315 06849	0,02739 726027	49,315 0684 9	0,27397 26027	493,1 5068 49	2,73972 6027	4931, 5068 49
1 week	0,0191780 8219	34,520 54795	0,191780 8219	345,2 05479 5	1,917808 219	3452, 0547 95	19,17808 219	3452 0,547 95

	70 eth in 1 week	
Time until second deposit	Min amount to deposit (ETH)	Value(USD)
1 minute	0,006944444444	12,5
15 minutes	0,1041666667	187,5
1 hour	0,4166666667	750

Mitigation steps

Check `mintAmount > 0` in function `wxETH::stake`. This means:

```
    function stake(uint256 xETHAmount) external drip returns (ui
        /// @dev calculate the amount of wxETH to mint
        uint256 mintAmount = previewStake(xETHAmount);
```

```
+       if (mintAmount == 0){
+           revert ERROR_MINTING_0_SHARES();
+       }

        /// @dev transfer xETH from the user to the contract
        xETH.safeTransferFrom(msg.sender, address(this), xETHAmc

        /// @dev emit event
        emit Stake(msg.sender, xETHAmount, mintAmount);

        /// @dev mint the wxETH to the user
        _mint(msg.sender, mintAmount);

        return mintAmount;
    }
```

d3e4 (warden) commented:

> I don't think this and it's duplicates are all one issue, but two different issues. One issue is the standard inflation attack by transferring tokens directly to the contract. This is described by #4, #21 and #22. The other issue is the one described in #3 and #24. There the inflation is instead caused by the contract itself, by dripping, which is another loophole that may be exploited even if the vulnerability causing the inflation attack is patched.

vaporkane (xETH) acknowledged, but disagreed with severity

kirk-baird (judge) decreased severity to Medium and commented:

> @d3e4 - I agree with this comment that these attacks are different enough to warrant separate issues.

> Further, since these attacks can only be executed by the very first staker and there is only a single exchange rate the likelihood of attack is very low. I'm therefore downgrading the severity to medium.

> Additionally, the chance of the drip attack is extremely low since the user would need to be the only staker for an extended period of time. This borders between a low and medium severity but I will generously consider it medium in this case.

xETH mitigated:

> Don't allow minting of 0 shares.

> PR: https://github.com/code-423n4/2023-05-xeth/commit/fbb29727ce21857f60c1c94fff43c73b4124a4b4

Status: Not fully mitigated. Full details in reports from ronnyx2017 and d3e4, and also included in the Mitigation Review section below.

Low Risk and Non-Critical Issues

For this audit, 5 reports were submitted by wardens detailing low risk and non-critical issues. The report highlighted below by **adriro** received the top score from the judge.

The following wardens also submitted reports: d3e4, ronnyx2017, bin2chen *and* carlitox477.

Low Issue Summary

	Issue
[L-01]	Wrong function declaration in IBaseRewardPool interface
[L-02]	Incorrect function declaration in ICurveFactory interface
[L-03]	`grantRole` operation in `setAMO` function may incorrectly check role access
[L-04]	Potential accidental inclusion of ERC20Burnable in xETH token
[L-05]	`addLockedFunds` should check `dripRatePerBlock` is not zero
[L-06]	Use Ownable2Step instead of Ownable for access control
[L-07]	`recoverToken` function should restrict which tokens are allowed to be recovered
[L-08]	`setOperator` and `setRewardsRecipient` don't check for `address(0)`
[L-09]	Initialize `rebalanceUpCap` and `rebalanceDownCap` in AMO constructor
[L-10]	Validate argument in `setRebalanceUpThreshold` and `setRebalanceDownThreshold`

	Issue
[L-11]	`rebalanceUpCap` and `rebalanceDownCap` operate on different types of value

[L-01] Wrong function declaration in IBaseRewardPool interface

The `withdraw` function declaration is missing the `bool` return value.

https://github.com/code-423n4/2023-05-xeth/blob/main/src/interfaces/IBaseRewardPool.sol#L14

```
    function withdraw(uint256 amount, bool claim) external;
```

[L-02] Incorrect function declaration in ICurveFactory interface

The `deploy_pool` function doesn't exist in the Curve Factory contract (see https://github.com/curvefi/curve-factory/blob/master/contracts/Factory.vy).

https://github.com/code-423n4/2023-05-xeth/blob/main/src/interfaces/ICurveFactory.sol#L5-L18

```
    function deploy_pool(
        string memory name,
        string memory symbol,
        address[] memory coins,
        uint256 A,
        uint256 mid_fee,
        uint256 out_fee,
        uint256 allowed_extra_profit,
        uint256 fee_gamma,
        uint256 adjustment_step,
        uint256 admin_fee,
        uint256 ma_half_time,
        uint256 initial_price
    ) external returns (address);
```

[L-03] `grantRole` operation in `setAMO` function may incorrectly check role access

https://github.com/code-423n4/2023-05-xeth/blob/main/src/xETH.sol#L54

Instead of using the internal `_grantRole` function, the `setAMO` function present in the xETH is using the `grantRole` function of the OpenZeppelin AccessControl library, which includes a validation that the caller has the admin role for the role being granted (`MINTER_ROLE` in this case):

https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/access/AccessControl.sol#L145-L147

```
    function grantRole(bytes32 role, address account) public virtual
        _grantRole(role, account);
    }
```

This should be ok as long as the admin role for `MINTER_ROLE` is not defined. Currently, the `setAMO` function will be called by the `DEFAULT_ADMIN_ROLE` which is the default admin role for the `MINTER_ROLE` role. But if the role admin for the `MINTER_ROLE` is defined, then the default admin role will not be the admin role for the `MINTER_ROLE` and the operation will be reverted.

The recommendation is to change `grantRole` for the internal variant `_grantRole`.

[L-04] Potential accidental inclusion of ERC20Burnable in xETH token

https://github.com/code-423n4/2023-05-xeth/blob/main/src/xETH.sol#L9

The xETH token inherits from ERC20Burnable, which defines public methods to allow burning of tokens.

Presumably, this was added accidentally as a mistake in order to implement the `burnShares()` function in xETH. The internal function `_burn` is already provided in the base implementation of ERC20 (OpenZepplin library), and doesn't need any extra implementation, which may increase the attack surface of the protocol.

The recommendation is to remove ERC20Burnable from the inheritance list of xETH.

[L-05] `addLockedFunds` should check `dripRatePerBlock` is not zero

https://github.com/code-423n4/2023-05-xeth/blob/main/src/wxETH.sol#L146-L148

The `addLockedFunds` function present in the wxETH contract should check that `dripRatePerBlock > 0`.

[L-06] Use Ownable2Step instead of Ownable for access control

- https://github.com/code-423n4/2023-05-xeth/blob/main/src/wxETH.sol#L9
- https://github.com/code-423n4/2023-05-xeth/blob/main/src/CVXStaker.sol#L11

Use the Ownable2Step variant of the Ownable contract to better safeguard against accidental transfers of access control.

[L-07] `recoverToken` function should restrict which tokens are allowed to be recovered

https://github.com/code-423n4/2023-05-xeth/blob/main/src/CVXStaker.sol#L101-L109

The CVXStaker contract manages different tokens as part of its natural intended behavior. Notably, these are the Curve LP tokens, Convex tokens, and the different reward tokens associated with the Convex staking pool.

The `recoverToken` implementation can arbitrarily transfer any ERC20 token, which may also be used to transfer these tokens.

The recommendation is to restrict these tokens, which are an important part of the process around the contract and should not be “recovered” as part of an accidental side effect.

[L-08] `setOperator` and `setRewardsRecipient` don't check for `address(0)`

- https://github.com/code-423n4/2023-05-xeth/blob/main/src/CVXStaker.sol#L78
- https://github.com/code-423n4/2023-05-xeth/blob/main/src/CVXStaker.sol#L89

Check that the arguments are not `address(0)`.

[L-09] Initialize `rebalanceUpCap` and `rebalanceDownCap` in AMO constructor

https://github.com/code-423n4/2023-05-xeth/blob/main/src/AMO2.sol#L175

Initialize these two variables to comply with the invariant of these values being different from zero, which is checked in their respective setters.

[L-10] Validate argument in `setRebalanceUpThreshold` and `setRebalanceDownThreshold`

- https://github.com/code-423n4/2023-05-xeth/blob/main/src/AMO2.sol#L495
- https://github.com/code-423n4/2023-05-xeth/blob/main/src/AMO2.sol#L512

Validate that the thresholds are within the 0-1e18 range in `setRebalanceUpThreshold` and `setRebalanceDownThreshold` function of the AMO contract.

[L-11] `rebalanceUpCap` and `rebalanceDownCap` operate on different types of value

Caps operate on different types of value. `rebalanceUpCap` is checked against LP tokens:

https://github.com/code-423n4/2023-05-xeth/blob/main/src/AMO2.sol#L252

```
if (quote.lpBurn > rebalanceUpCap) revert RebalanceUpCapExceeded
```

While `rebalanceDownCap` is checked against xETH tokens:

https://github.com/code-423n4/2023-05-xeth/blob/main/src/AMO2.sol#L299

```
if (quote.xETHAmount > rebalanceDownCap)
    revert RebalanceDownCapExceeded();
```

Informational Issue Summary

	Issue	
[INFO-01]	Type of Curve pool is an important and sensible parameter	
[INFO-02]	Several centralization risks introduce multiple points of failure	

[INFO-01] Type of Curve pool is an important and sensible parameter

Protocol owners will deploy a Curve pool as an essential piece of the protocol. The current intention is to use a Curve for the pair xETH/stETH.

The protocol team has asserted that they will be using a plain pool, “balances” type, in which none of the coins are native ETH.

There are two issues that shouldn’t be a problem, as long as these conditions are maintained:

1. stETH being a rebasing token needs the Curve pool to be of “balances” type. Metapools are not supported either.
2. As the protocol uses Curve’s `get_virtual_price()` (in the AMO contract) it can potentially be vulnerable to the read-only reentrancy attack described in the following article. This attack is not possible as long as none of the coins is ETH, which is the main driver to trigger the reentrancy.

If conditions are changed, and the type of Curve pool is changed, these two potential issues could be manifested as vulnerabilities in the protocol.

[INFO-02] Several centralization risks introduce multiple points of failure

Although much care has been taken to contain and bound the accessible functionality of the defender role, there are multiple centralization risks around the protocol that imply a lot of trust in owners or controllers of the protocol.

Here is a non-exhaustive list of the different places that may introduce potential failures due to centralization:

- xETH can be paused, freezing all operations.
- AMO can be changed in xETH, which grants arbitrary minting capabilities to the new account.
- Minting and burning is also accessible to the `DEFAULT_ADMIN_ROLE`.
- Configuration for wxETH rewards can be arbitrarily changed.
- Rewards for wxETH are handled manually via off-chain processes (claim revenue and call to add locked funds).
- Rebalance operations have cooldown, but cooldown can be changed to zero.
- Rebalance can be neutralized by setting zero caps.
- Rebalance thresholds can be arbitrarily changed to any amount.
- Remove liquidity functions in AMO contract can be used to empty the Curve pool.
- CXVStaker can be changed in AMO contract (sensible because this contract controls Curve LPs).
- LP tokens in CVXStaker can be withdrawn and sent to an arbitrary account (`withdrawAndUnwrap`).

Non Critical Issue Summary

	Issue	Instances
[N-01]	Import declarations should import specific symbols	20
[N-02]	Use `uint256` instead of the `uint` alias	2
[N-03]	The `accrueDrip()` function can use `_accrueDrip()` to avoid the empty block implementation	-

[N-01] Import declarations should import specific symbols

Prefer import declarations that specify the symbol(s) using the form `import {SYMBOL} from "SomeContract.sol"` rather than importing the whole file

Instances (20):

```
File: src/AMO2.sol

4: import "@openzeppelin-contracts/token/ERC20/IERC20.sol";

5: import "@openzeppelin-contracts/token/ERC20/utils/SafeERC20.s

6: import "@openzeppelin-contracts/access/AccessControl.sol";

7: import "./interfaces/ICurvePool.sol";


File: src/CVXStaker.sol

4: import "@openzeppelin-contracts/token/ERC20/IERC20.sol";

5: import "@openzeppelin-contracts/token/ERC20/utils/SafeERC20.s

6: import "@openzeppelin-contracts/access/Ownable.sol";

7: import "./interfaces/ICurvePool.sol";

8: import "./interfaces/ICVXBooster.sol";

9: import "./interfaces/IBaseRewardPool.sol";
```

```
File: src/interfaces/IXETH.sol

4: import "@openzeppelin-contracts/token/ERC20/IERC20.sol";


File: src/wxETH.sol

3: import "@openzeppelin-contracts/token/ERC20/ERC20.sol";

4: import "@openzeppelin-contracts/token/ERC20/IERC20.sol";

5: import "@openzeppelin-contracts/token/ERC20/utils/SafeERC20.s

6: import "@openzeppelin-contracts/access/Ownable.sol";

7: import "solmate/utils/FixedPointMathLib.sol";


File: src/xETH.sol

4: import "@openzeppelin-contracts/token/ERC20/ERC20.sol";

5: import "@openzeppelin-contracts/token/ERC20/extensions/ERC20F

6: import "@openzeppelin-contracts/security/Pausable.sol";

7: import "@openzeppelin-contracts/access/AccessControl.sol";
```

[N-02] Use `uint256` instead of the `uint` alias

Prefer using the `uint256` type definition over its `uint` alias.

Instances (2):

```
File: src/CVXStaker.sol

195:             for (uint i = 0; i < rewardTokens.length; i++)


File: src/wxETH.sol
```

```
16:     event UpdateDripRate(uint oldDripRatePerBlock, uint256 r
```

[N-03] The `accrueDrip()` function can use `_accrueDrip()` to avoid the empty block implementation

The implementation of the `accrueDrip()` function can be changed to use `_accrueDrip()` instead of the `drip` modifier to avoid the empty block implementation.

```
function accrueDrip() external {
  _accrueDrip();
}
```

vaporkane (xETH) confirmed

kirk-baird (judge) commented:

> This is a high quality report, where all issues have the correct security rating and are valid.

Gas Optimizations

For this audit, 4 reports were submitted by wardens detailing gas optimizations. The report highlighted below by **adriro** received the top score from the judge.

The following wardens also submitted reports: d3e4, ronnyx2017 *and* carlitox477.

xETH.sol contract

- `grantRole` call in `setAMO` function can be changed to use the internal variant `_grantRole` to avoid re-checking for access control, saving gas.
  https://github.com/code-423n4/2023-05-xeth/blob/main/src/xETH.sol#L54
- The `curveAMO` storage variable is read 3 times from storage in `setAMO`. Consider using a local variable to save gas.

https://github.com/code-423n4/2023-05-xeth/blob/main/src/xETH.sol#L44
https://github.com/code-423n4/2023-05-xeth/blob/main/src/xETH.sol#L45
https://github.com/code-423n4/2023-05-xeth/blob/main/src/xETH.sol#L54

- `mintShares` function visibility can be changed to external as it isn't used internally in the contract.
https://github.com/code-423n4/2023-05-xeth/blob/main/src/xETH.sol#L63
- `burnShares` function visibility can be changed to external as it isn't used internally in the contract.
https://github.com/code-423n4/2023-05-xeth/blob/main/src/xETH.sol#L75
- `whenNotPaused` modifier can be removed from the `mintShares` function as this check is already provided by the internal `_beforeTokenTransfer` callback.
https://github.com/code-423n4/2023-05-xeth/blob/main/src/xETH.sol#L63
- `whenNotPaused` modifier can be removed from the `burnShares` function as this check is already provided by the internal `_beforeTokenTransfer` callback.
https://github.com/code-423n4/2023-05-xeth/blob/main/src/xETH.sol#L75
- `whenNotPaused` modifier can be removed from the `pause` function as this check is already included in the internal `_pause` function.
https://github.com/code-423n4/2023-05-xeth/blob/main/src/xETH.sol#L86
- `whenNotPaused` modifier can be removed from the `unpause` function as this check is already included in the internal `_unpause` function.
https://github.com/code-423n4/2023-05-xeth/blob/main/src/xETH.sol#L93

wxETH.sol contract

- The SafeERC20 library can be safely removed as the contract will only deal with the xETH token, which is a known implementation that adheres correctly to the ERC20 standard.
https://github.com/code-423n4/2023-05-xeth/blob/main/src/wxETH.sol#L10
- `lastReport` and `dripEnabled` storage variables can be combined to use a single slot and save gas by decreasing the precision of `lastReport` to `uint248`, which should still be plenty enough as this represents a block number.
https://github.com/code-423n4/2023-05-xeth/blob/main/src/wxETH.sol#L60-L62

- `lastReport` variable initialization can be safely skipped in the constructor, as it will be eventually initialized by the call to `startDrip()` (dripping is disabled at start).
https://github.com/code-423n4/2023-05-xeth/blob/main/src/wxETH.sol#L73
- `lockedFunds` storage variable is re-read from storage while emitting the event in the `addLockedFunds()` function. Consider using a local variable cache.
https://github.com/code-423n4/2023-05-xeth/blob/main/src/wxETH.sol#L156
- `drip` is unneeded in the function `startDrip()` as it is presumed that currently dripping is disabled, making `_accrueDrip` have no effect.
https://github.com/code-423n4/2023-05-xeth/blob/main/src/wxETH.sol#L170
- Updating `lastReport` in the `stopDrip()` is not needed as it is already updated during the call to the `drip` modifier.
https://github.com/code-423n4/2023-05-xeth/blob/main/src/wxETH.sol#L191
- `lockedFunds` storage variable is read from storage 5 times during the course of the `_accrueDrip()` function. Consider using local variable variants.
https://github.com/code-423n4/2023-05-xeth/blob/main/src/wxETH.sol#L236
https://github.com/code-423n4/2023-05-xeth/blob/main/src/wxETH.sol#L241
https://github.com/code-423n4/2023-05-xeth/blob/main/src/wxETH.sol#L248
https://github.com/code-423n4/2023-05-xeth/blob/main/src/wxETH.sol#L254
- Calculation to update `lockedFunds` can be done using unchecked math since `dripAmount` is guaranteed to be not greater than `lockedFunds`.
https://github.com/code-423n4/2023-05-xeth/blob/main/src/wxETH.sol#L241

CVXStaker.sol contract

- `rewardsRecipient` is re-read multiple times from storage (once per each iteration of the loop) in the `getRewards()` function. Consider reading it once and using a local variable after.

https://github.com/code-423n4/2023-05-xeth/blob/main/src/CVXStaker.sol#L190
https://github.com/code-423n4/2023-05-xeth/blob/main/src/CVXStaker.sol#L195

- `rewardTokens[i]` is read twice in each iteration of the loop in the `getRewards()` function. Consider reading it once and using a local variable after.
https://github.com/code-423n4/2023-05-xeth/blob/main/src/CVXStaker.sol#L192 https://github.com/code-423n4/2023-05-xeth/blob/main/src/CVXStaker.sol#L195

AMO2.sol contract

- The SafeERC20 library can be safely removed as the contract will only deal with xETH and stETH tokens, which are known implementations that adhere correctly to the ERC20 standard.
https://github.com/code-423n4/2023-05-xeth/blob/main/src/AMO2.sol#L12
- Infinite token approval can be granted to the curve pool for the xETH and stETH tokens at contract construction time. This will save the approval calls present in each of the functions that deal with adding liquidity to the pool.
https://github.com/code-423n4/2023-05-xeth/blob/main/src/AMO2.sol#L309
https://github.com/code-423n4/2023-05-xeth/blob/main/src/AMO2.sol#L545
https://github.com/code-423n4/2023-05-xeth/blob/main/src/AMO2.sol#L546
https://github.com/code-423n4/2023-05-xeth/blob/main/src/AMO2.sol#L571
- `rebalanceUp` function can take struct parameter from `calldata` instead of `memory`, saving a copy operation.
https://github.com/code-423n4/2023-05-xeth/blob/main/src/AMO2.sol#L240
- `rebalanceDown` function can take struct parameter from `calldata` instead of `memory`, saving a copy operation.
https://github.com/code-423n4/2023-05-xeth/blob/main/src/AMO2.sol#L287

- `bestRebalanceUpQuote` function can take struct parameter from `calldata` instead of `memory`, saving a copy operation. https://github.com/code-423n4/2023-05-xeth/blob/main/src/AMO2.sol#L336
- `bestRebalanceDownQuote` function can take struct parameter from `calldata` instead of `memory`, saving a copy operation. https://github.com/code-423n4/2023-05-xeth/blob/main/src/AMO2.sol#L361
- `rebalanceUp` function reads the `cvxStaker` twice from storage. Consider reading it once and using a local variable after. https://github.com/code-423n4/2023-05-xeth/blob/main/src/AMO2.sol#L256 https://github.com/code-423n4/2023-05-xeth/blob/main/src/AMO2.sol#L261
- `rebalanceDown` function reads the `cvxStaker` twice from storage. Consider reading it once and using a local variable after. https://github.com/code-423n4/2023-05-xeth/blob/main/src/AMO2.sol#L314 https://github.com/code-423n4/2023-05-xeth/blob/main/src/AMO2.sol#L317
- `bestRebalanceUpQuote` and `bestRebalanceDownQuote` can return a single result (`min_xETHReceived` and `minLpReceived` respectively) instead of returning a whole new struct in memory. https://github.com/code-423n4/2023-05-xeth/blob/main/src/AMO2.sol#L337 https://github.com/code-423n4/2023-05-xeth/blob/main/src/AMO2.sol#L362
- `defender` storage variable is read 3 times from storage in the implementation of `setRebalanceDefender`. Consider using a local variable variant. https://github.com/code-423n4/2023-05-xeth/blob/main/src/AMO2.sol#L391 https://github.com/code-423n4/2023-05-xeth/blob/main/src/AMO2.sol#L392 https://github.com/code-423n4/2023-05-xeth/blob/main/src/AMO2.sol#L395
- `minAmounts` array is not needed in the `removeLiquidityOnlyStETH()` function and can be safely removed.

https://github.com/code-423n4/2023-05-xeth/blob/main/src/AMO2.sol#L640-L642

vaporkane (xETH) confirmed

Mitigation Review

Introduction

Following the C4 audit, 3 wardens (bin2chen, d3e4 and ronnyx2017) reviewed the mitigations for all identified issues. Additional details can be found within the C4 xETH Mitigation Review repository.

Overview of Changes

Summary from the Sponsor:

- Many issues assume a rogue defender as a starting point of attack, so we have chosen to add more granularity of parameters and checks to reduce risks there.
- Most mitigations would either be checks, or smaller changes of function parameters and calls
- In production we have planned to use MEV Protection services such as flashbots rpc

Mitigation Review Scope

URL	Mitigation of	Purpose
https://github.com/code-423n4/2023-05-xeth/commit/60b9e1e2562d585831e3d8e2a7e345294d9dacbd	M-02	This mitigation adds a getTotalBalance function
https://github.com/code-423n4/2023-05-xeth/commit/1f714868f193cdeb472ec097110901a997d87ec4	M-03	This mitigation adds a balance check
https://github.com/code-423n4/2023-05-xeth/commit/793dade5217bd5751856f7cf0bccd4936286aeab	M-04	change safeApprove to approve

URL	Mitigation of	Purpose
https://github.com/code-423n4/2023-05-xeth/commit/aebc3244cbb0deb67f3cdef160b390da27888de7	M-05	add a totalSupply check
https://github.com/code-423n4/2023-05-xeth/commit/630114ea6a3782f2f539b5936c524f8da62d0179	M-07	Partial: add admin controlled slippage values
https://github.com/code-423n4/2023-05-xeth/commit/1f714868f193cdeb472ec097110901a997d87ec4	M-08	Add a setRewardTokens function
https://github.com/code-423n4/2023-05-xeth/commit/a840dc0b8a1de59a3ea06e0814ea3ce26707bdae	M-09	change sendToOperator to sendToOwner
https://github.com/code-423n4/2023-05-xeth/commit/fbb29727ce21857f60c1c94fff43c73b4124a4b4	M-10	Don't allow minting of 0 shares.

Mitigation Review Summary

Original Issue	Status	Full Details
M-02	Mitigation confirmed	Reports from d3e4, bin2chen and ronnyx2017
M-03	Mitigation confirmed	Reports from ronnyx2017, bin2chen and d3e4
M-04	Mitigation confirmed	Reports from ronnyx2017, bin2chen and d3e4
M-05	Not fully mitigated	Reports from ronnyx2017, bin2chen and d3e4, and also shared below
M-07	Partially mitigated	Reports from ronnyx2017 (here and here) and d3e4, and also shared below
M-08	Mitigation confirmed with comments	Reports from bin2chen, d3e4 and ronnyx2017
M-09	Mitigation confirmed	Reports from ronnyx2017, bin2chen and d3e4
M-10	Not fully mitigated	Reports from ronnyx2017 and d3e4, and also shared below

See below for details regarding the three issues that were not fully mitigated.

Mitigation of M-05: Issue not fully mitigated

Submitted by d3e4, also found by bin2chen and ronnyx2017

Original issue

M-05: Virgin stake can claim all drops

Fix: https://github.com/code-423n4/2023-05-xeth/commit/aebc3244cbb0deb67f3cdef160b390da27888de7

The issue is, if dripping is enabled when `totalSupply() == 0`, the entire amount dripped will immediately accrue to the first stake.

Mitigation review - dripping is still implicit when `totalSupply() == 0`

The drip accrual is now skipped, i.e. `_accrueDrip()` simply returns when `totalSupply() == 0`. However, the drip is implicitly accrued at a constant rate per time by actually adding the dripped amount only at discrete points in time. The last point in time of which is `lastReport`, which is what happens in `_accrueDrip()`. This means, simply skipping this drip accrual when `totalSupply() == 0` only defers explicit drip accrual to the next time, but will still drip the same amount because it is calculated from the same `lastReport`. That is, the drip is not truly suspended until the first stake. The attack will still work because the drip will just accrue on the unstake instead, even when it is unstaked immediately after the first stake.

Example:

1. Start drip at t = 0. `lastReport = 0` and `totalSupply() == 0`.
2. First stake at t = T. `totalSupply() == 0` so `_accrueDrip()` immediately returns, which means that nothing is dripped and that `lastReport` remains `0`.
3. Unstake at t = T. `totalSupply() > 0` so `T * dripRatePerBlock` is dripped, which goes to the unstaker.

The issue is in step 2, the `lastReport` remains `0`. In order to truly not drip, `lastReport` must be reset without adding any dripped amount. Then the drip is neither implicitly nor explicitly happening unless something has been staked, i.e. `totalSupply() > 0`.

Suggested mitigation

This can be achieved by, in `_accrueDrip()`:

```
if (!dripEnabled) return;
if (totalSupply() == 0) {
    lastReport = block.number;
    return;
}
```

Now, `startDrip()` enables drip, but the drip only truly starts after the first stake. It also automatically stops when everything is unstaked, i.e. the drip only kicks in when something is staked.

Mitigation of M-10: Issue not fully mitigated

Submitted by ronnyx2017, also found by **d3e4**

Original issue

M-10: First 1 wei deposit can produce lose of user xETH funds in `wxETH`

Comments

The issue is a typical inflation attack, that the first staker adds the huge amount of the underlayer tokens after minting shares, and the following stakers can only mint zero share because of division precision error.

The mitigation uses a regular schedule that check if the minting shares is zero to protect the following staker from losing all their staking tokens. But it doesn't work on edge condition:

1. The initial process is as same as the original issue. The first staker mints only 1 wei share and drip accrued.
2. After the mitigation, the second staker can't deposit any amount less than `dripAmount + 1`, which will revert with CantMintZeroShares error. But if the second staker deposits `2 * (dripAmount + 1) - 1` xETH, the staker will only

receive 1 wei wxETH because of round-down error. After the second deposit, `1 wei wxETH = (3 * (dripAmount + 1) - 1) / 2` . And the second staker losses `0.5 * (dripAmount + 1) - 0.5` xETH.

Suggestion

First I need to clarify that I think it's a low/NC risk issue because it only can loss some dusts.

If the team really wants to fully fix it as inflation attack, please refer to https://ethereum-magicians.org/t/address-eip-4626-inflation-attacks-with-virtual-shares-and-assets/12677 .

But I strongly do not recommend such mitigation for this issue because it adds too many entities only for insignificant loss.

Lack of Initialization and scope check for slippage

Submitted by ronnyx2017, also found by d3e4 and ronnyx2017

Severity: QA

Note: related to mitigation for M-07.

Lines of code

https://github.com/code-423n4/2023-05-xeth/blob/add-xeth/src/AMO2.sol#L145 https://github.com/code-423n4/2023-05-xeth/blob/add-xeth/src/AMO2.sol#L430-L442

Vulnerability details

1. The `upSlippage` doesn't have a default value. So if rebalance is called before the admin `setSlippage` , the `upSlippage` is always zero.
2. The `AMO2.setSlippage` function does not implement the scope check in the comments:

```
    @notice The new maximum slippage must be between 0.06% and 15% (
```

It only limits the slippages are not > 100%:

```
    if (newUpSlippage >= BASE_UNIT || newDownSlippage >= BASE_UNIT)
            revert InvalidSetSlippage();
```

Assessed type

Invalid Validation

kirk-baird (judge) commented via issue #17:

> The development team have scoped M-07 as a partial fix, leaving part of the issue as won't-fix. This is decided based on the low likelihood of attack and small gains by an attacker exploiting this issue.

> This issue, as well as **#23** and **#18**, are tagged as unmitigated. This is not strictly accurate as the initial issue is only intended to be partially mitigated. However, there are some worthwhile comments from the wardens explaining the partial fix and some additional details. Thus, I'm going to leave the tag as `unmitigated` rather than `mitigated`.

Disclosures

C4 is an open organization governed by participants in the community.

C4 audits incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Audit submissions are judged by a knowledgeable security researcher and solidity developer and disclosed to sponsoring developers. C4 does not conduct formal verification regarding the provided code but instead provides final verification.

C4 does not provide any guarantee or warranty regarding the security of this project. All smart contract software should be used at the sole risk and responsibility

of users.

Top