Concur Finance contest Findings & Analysis Report

2022-05-24

Table of contents

- Disclosures

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 contest 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 contest outlined in this document, C4 conducted an analysis of the Concur Finance smart contract system written in Solidity. The audit contest took place between February 3—February 9 2022.

Wardens

58 Wardens contributed reports to the Concur Finance contest:

1. leastwood
2. WatchPug (jtp and ming)
3. cmichel
4. hickuphh3
5. pauliax
6. wuwe1
7. gzeon
8. hyh
9. throttle
10. Czar102
11. csanuragjain
12. kirk-baird
13. IllIllI

14. 0x1f8b
15. 0xw4rd3n
16. CertoraInc (danb, egjlmn1, OriDabush, ItayG, and shakedwinder)
17. cccz
18. 0xliumin
19. Dravee
20. harleythedog
21. reassor
22. kenta
23. danb
24. Ruhum
25. hubble (ksk2345 and shri4net)
26. Jujic
27. defsec
28. bobi
29. Randyyy
30. ShadowyNoobDev
31. bitbopper
32. SolidityScan (cyberboy and zombie)
33. Heartless
34. BouSalman
35. mtz
36. robee
37. rfa
38. Sleepy
39. peritoflores
40. ye0lde
41. Rhynorater
42. samruna

43. cryptphi
44. 0xngndev
45. 0x0x0x
46. 0xNot0rious
47. Tomio
48. 0x510c
49. sabtikw
50. GeekyLumberjack
51. ckksec

This contest was judged by Alex the Entreprenerd. The judge also competed in the contest as a warden, but forfeited their winnings.

Final report assembled by liveactionllama.

Summary

The C4 analysis yielded an aggregated total of 42 unique vulnerabilities. Of these vulnerabilities, 11 received a risk rating in the category of HIGH severity and 31 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 36 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 33 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 Concur Finance contest repository, and is composed of 8 smart contracts written in the Solidity programming language and includes 1,213 lines of Solidity code.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities according to a methodology based on OWASP standards.

Vulnerabilities are divided into 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

Further information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on the C4 website.

High Risk Findings (11)

[H-01] Wrong reward token calculation in MasterChef contract

Submitted by throttle, also found by cccz, cmichel, and leastwood

MasterChef.sol#L86

When adding new token pool for staking in MasterChef contract

```
    function add(address _token, uint _allocationPoints, uint16 _dep
```

All other, already added, pools should be updated but currently they are not.
Instead, only totalPoints is updated. Therefore, old (and not updated) pools will lose it’s share during the next update.
Therefore, user rewards are not computed correctly (will be always smaller).

Proof of Concept

Scenario 1:

1. Owner adds new pool (first pool) for staking with points = 100 (totalPoints=100) and 1 block later Alice stakes 10 tokens in the first pool.
2. 1 week passes
3. Alice withdraws her 10 tokens and claims X amount of reward tokens.
   and 1 block later Bob stakes 10 tokens in the first pool.
4. 1 week passes
5. Owner adds new pool (second pool) for staking with points = 100 (totalPoints=200)
   and 1 block later Bob withdraws his 10 tokens and claims X/2 amount of reward tokens.
   But he should get X amount

Scenario 2:

1. Owner adds new pool (first pool) for staking with points = 100 (totalPoints=100).
2. 1 block later Alice, Bob and Charlie stake 10 tokens there (at the same time).
3. 1 week passes
4. Owner adds new pool (second pool) for staking with points = 400 (totalPoints=500)
5. Right after that, when Alice, Bob or Charlie wants to withdraw tokens and claim rewards they will only be able to claim 20% of what they should be eligible for, because their pool is updated with 20% (100/500) rewards instead of 100% (100/100) rewards for the past week.

Recommended Mitigation Steps

Update all existing pools before adding new pool. Use the massUdpate() function which is already present ... but unused.

ryuheimat (Concur) confirmed

Alex the Entreprenerd (judge) commented:

> The warden has identified a fallacy in how `add` s logic work.

> Ultimately rewards in this contract have to be linearly vested over time, adding a new pool would change the rate at which vesting in all pools will go.

> For that reason, it is necessary to accrue the rewards that each pool generated up to that point, before changing the slope at which rewards will be distributed.

> In this case add should massUpdateFirst.

> Because this vulnerability ultimately breaks the accounting of the protocol, I believe High Severity to be appropriate.

[H-02] Masterchef: Improper handling of deposit fee

Submitted by hickuphh3, also found by leastwood

[MasterChef.sol#L170-L172]()

If a pool's deposit fee is non-zero, it is subtracted from the amount to be credited to the user.

```
if (pool.depositFeeBP > 0) {
  uint depositFee = _amount.mul(pool.depositFeeBP).div(_perMille
  user.amount = SafeCast.toUint128(user.amount + _amount - depos
}
```

However, the deposit fee is not credited to anyone, leading to permanent lockups of deposit fees in the relevant depositor contracts (StakingRewards and ConvexStakingWrapper for now).

Proof of Concept

Example 1: ConvexStakingWrapper

Assume the following

- The curve cDai / cUSDC / cUSDT LP token corresponds to `pid = 1` in the convex booster contract.
- Pool is added in Masterchef with `depositFeeBP = 100 (10%)`.
- Alice deposits 1000 LP tokens via the ConvexStakingWrapper contract. A deposit fee of 100 LP tokens is charged. Note that the `deposits` mapping of the ConvexStakingWrapper contract credits 1000 LP tokens to her.
- However, Alice will only be able to withdraw 900 LP tokens. The 100 LP tokens is not credited to any party, and is therefore locked up permanently (essentially becomes protocol-owned liquidity). While she is able to do `requestWithdraw()` for 1000 LP tokens, attempts to execute `withdraw()` with amount = 1000 will revert because she is only credited 900 LP tokens in the Masterchef contract.

Example 2: StakingRewards

- CRV pool is added in Masterchef with `depositFeeBP = 100 (10%)`.
- Alice deposits 1000 CRV into the StakingRewards contract. A deposit fee of 100 CRV is charged.
- Alice is only able to withdraw 900 CRV tokens, while the 100 CRV is not credited to any party, and is therefore locked up permanently.

These examples are non-exhaustive as more depositors can be added / removed from the Masterchef contract.

Recommended Mitigation Steps

I recommend shifting the deposit fee logic out of the masterchef contract into the depositor contracts themselves, as additional logic would have to be added in the masterchef to update the fee recipient's state (rewardDebt, send pending concur rewards, update amount), which further complicates matters. As the fee recipient is likely to be the treasury, it is also not desirable for it to accrue concur rewards.

```
if (pool.depositFeeBP > 0) {
  uint depositFee = _amount.mul(pool.depositFeeBP).div(_perMille
  user.amount = SafeCast.toUint128(user.amount + _amount - depos
  UserInfo storage feeRecipient = userInfo[_pid][feeRecipient];
  // TODO: update and send feeRecipient pending concur rewards
  feeRecipient.amount = SafeCast.toUint128(feeRecipient.amount +
        // TODO: update fee recipient's rewardDebt
    }
```

ryuheimat (Concur) confirmed

Alex the Entreprenerd (judge) commented:

> The warden has identified a way for funds to be forever lost, because of that reason I believe High Severity to be appropriate.

> Mitigation could be as simple as transferring the fee to a `feeReceiver` or adding a way to pull those fees.

[H-03] Repeated Calls to Shelter.withdraw Can Drain All Funds in Shelter

Submitted by mtz, also found by 0x1f8b, 0xliumin, bitbopper, cccz, cmichel, csanuragjain, Czar102, danb, Alex the Entreprenerd, GeekyLumberjack, gzeon, hickuphh3, hyh, leastwood, Randyyy, Rhynorater, Ruhum, and ShadowyNoobDev

Shelter.sol#L52-L57

tl;dr Anyone who can call `withdraw` to withdraw their own funds can call it repeatedly to withdraw the funds of others. `withdraw` should only succeed if the user hasn't withdrawn the token already.

The shelter can be used for users to withdraw funds in the event of an emergency. The `withdraw` function allows callers to withdraw tokens based on the tokens they have deposited into the shelter client: ConvexStakingWrapper. However, `withdraw` does not check if a user has already withdrawn their tokens. Thus a user that can `withdraw` tokens, can call withdraw repeatedly to steal the tokens of others.

Proof of Concept

tl;dr an attacker that can successfully call `withdraw` once on a shelter, can call it repeatedly to steal the funds of others. Below is a detailed scenario where this situation can be exploited.

1. Mallory deposits 1 `wETH` into `ConvexStakingWrapper` using `deposit`. Let's also assume that other users have deposited 2 `wETH` into the same contract.
2. An emergency happens and the owner of `ConvexStakingWrapper` calls `setShelter(shelter)` and `enterShelter([pidOfWETHToken, ...])`. Now `shelter` has 3 `wETH` and is activated for `wETH`.
3. Mallory calls `shelter.withdraw(wETHAddr, MalloryAddr)`, Mallory will rightfully receive 1 wETH because her share of wETH in the shelter is 1/3.
4. Mallory calls `shelter.withdraw(wETHAddr, MalloryAddr)` again, receiving 1/3*2 = 2/3 wETH. `withdraw` does not check that she has already withdrawn. This time, the wETH does not belong to her, she has stolen the wETH of the other users. She can continue calling `withdraw` to steal the rest of the funds

Recommended Mitigation Steps

To mitigate this, `withdraw` must first check that `msg.sender` has not withdrawn this token before and `withdraw` must also record that `msg.sender` has withdrawn the token. The exact steps for this are below:

1. Add the following line to the beginning of `withdraw` (line 53):

```
require(!claimed[_token][msg.sender], "already claimed")
```

2. Replace line 55 with the following:

```
claimed[_token][msg.sender] = true;
```

This replacement is necessary because we want to record who is withdrawing, not where they are sending the token which isn't really useful info.

ryuheimat (Concur) confirmed

Alex the Entreprenerd (judge) commented:

> The warden has identified a logical fallacy in the `Shelter` contract.

> This would allow a caller to claim their tokens multiple times, as long as they send them to a new address.

> Mitigation is as simple as checking claims against `msg.sender`, however because all funds can be drained, this finding is of High Severity.

[H-04] `ConvexStakingWrapper`, `StakingRewards` Wrong implementation will send `concur` rewards to the wrong receiver

Submitted by WatchPug, also found by bobi, CertoraInc, csanuragjain, danb, hickuphh3, and leastwood

ConvexStakingWrapper.sol#L246
StakingRewards.sol#L99
MasterChef.sol#L159-L167

```
    UserInfo storage user = userInfo[_pid][_msgSender()];
    updatePool(_pid);

    if(user.amount > 0) {
        uint pending = user.amount * pool.accConcurPerShare / _concu
        if (pending > 0) {
            safeConcurTransfer(_recipient, pending);
        }
    }
```

`ConvexStakingWrapper`, `StakingRewards` is using `masterChef.deposit()`, `masterChef.withdraw()`, and these two functions on `masterChef` will take `_msgSender()` as the user address, which is actually the address of `ConvexStakingWrapper` and `StakingRewards`.

As a result, when calling `ConvexStakingWrapper.deposit()`, `ConvexStakingWrapper.withdraw()`, `StakingRewards.stake()`, `StakingRewards.withdraw()`, the `concur` rewards belongs to all the users of ConvexStakingWrapper / StakingRewards will be sent to the caller wrongfully.

Proof of Concept

1. Alice deposits `1,000,000` token to `pid 1`

Actual results on `masterChef`:

- userInfo[1][address(ConvexStakingWrapper)] = `1,000,000`

Expected results:

- userInfo[1][address(Alice)] = `1,000,000`
- 1 day later, Bob deposits `1` token to `pid 1`

Actual results on `masterChef`:

- userInfo[1][address(ConvexStakingWrapper)] = `1,000,001`
- all `pending rewards` sent to Bob

Expected results:

- userInfo[1][address(Alice)] = `1,000,000`
- userInfo[1][address(Bob)] = `1`
- all `pending rewards` should be sent to Alice

Recommended Mitigation Steps

Consider adding two new functions to MasterChef: `depositFor()` and `withdrawFor()`.

`ConvexStakingWrapper`, `StakingRewards` can utilize these two functions and get the accounting right.

```
    function depositFor(address _user, uint _pid, uint _amount) exte
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][_user];
```

ryuheimat (Concur) confirmed

[H-05] `USDMPegRecovery` Risk of fund locked, due to discrepancy between curveLP token value against internal contract math

Submitted by Alex the Entreprenerd, also found by gzeon, IllIIl, and leastwood

USDMPegRecovery.sol#L90
USDMPegRecovery.sol#L110
USDMPegRecovery.sol#L73
USDMPegRecovery.sol#L84

In `USDMPegRecovery` `deposit` and `withdraw` allow for direct deposits of a specific token (3crv or usdm).

The balances are directly changed and tracked in storage.

`provide` seems to be using the real balances (not the ones store) to provide liquidity.
Because of how curve works, you'll be able (first deposit) to provide exactly matching liquidity.
But after (even just 1 or) multiple swaps, the pool will be slightly imbalanced, adding or removing liquidity at that point will drastically change the balances in the contract from the ones tracked in storage.

Eventually users won't be able to withdraw the exact amounts they deposited.

This will culminate with real balances not matching user deposits, sometimes to user advantage and other times to user disadvantage, ultimately to the protocol dismay.

Proof of Concept

Deposit equal usdm and 3crv
LP
Do one trade on CRV
Withdraw the LP

The real balances are not matching the balances in storage.

User tries to withdraw all their balances, inevitable revert.

Recommended Mitigation Steps

Either find a way to price the user contribution based on the LP tokens (use virtual_price)
Or simply have people deposit the LP token directly (avoiding the IL math which is a massive headache)

leekt (Concur) confirmed

Alex the Entreprenerd (judge) commented:

> I'm forfeitting winnings as I am judging the contest.

> The sponsor confirmed.

> I believe the closest findings are #191 and #94 these both focus on the provide aspect.
> However, this finding shows how the Curve LP Math will cause the internal balances to break after just one LP provision.

> Because this breaks accounting of the protocol and will cause funds to be stuck I believe High Severity to be appropriate.

[H-06] `ConvexStakingWrapper.sol#_calcRewardIntegral` Wrong implementation can disrupt rewards calculation and distribution

Submitted by WatchPug, also found by cmichel, harleythedog, hickuphh3, kirk-baird, and leastwood

ConvexStakingWrapper.sol#L175-L204

```
        uint256 bal = IERC20(reward.token).balanceOf(address(this));
        uint256 d_reward = bal - reward.remaining;
        // send 20 % of cvx / crv reward to treasury
        if (reward.token == cvx || reward.token == crv) {
            IERC20(reward.token).transfer(treasury, d_reward / 5);
            d_reward = (d_reward * 4) / 5;
        }
        IERC20(reward.token).transfer(address(claimContract), d_rewa

        if (_supply > 0 && d_reward > 0) {
            reward.integral =
                reward.integral +
                uint128((d_reward * 1e20) / _supply);
        }

        //update user integrals
        uint256 userI = userReward[_pid][_index][_account].integral;
        if (userI < reward.integral) {
            userReward[_pid][_index][_account].integral = reward.int
            claimContract.pushReward(
                _account,
                reward.token,
                (_balance * (reward.integral - userI)) / 1e20
            );
        }

        //update remaining reward here since balance could have char
        if (bal != reward.remaining) {
            reward.remaining = uint128(bal);
        }
```

The problems in the current implementation:

- `reward.remaining` is not a global state; the `reward.remaining` of other `reward` s with the same rewardToken are not updated;
- `bal` should be refreshed before `reward.remaining = uint128(bal);` ;
- L175 should not use `balanceOf` but take the diff before and after `getReward()` .

Proof of Concept

- convexPool[1] is incentivized with CRV as the reward token, `1000 lpToken` can get `10 CRV` per day;
- convexPool[2] is incentivized with CRV as the reward token, `1000 lpToken` can get `20 CRV` per day.

- Alice deposits `1,000` lpToken to `_pid = 1`
- 1 day later, Alice deposits `500` lpToken to `_pid = 1`
- convexPool `getReward()` sends `10 CRV` as reward to contract
- `d_reward` = 10, `2 CRV` sends to `treasury`, `8 CRV` send to `claimContract`
- `rewards[1][0].remaining` = 10
- 0.5 day later, Alice deposits `500` lpToken to `_pid = 1`, and the tx will fail:
- convexPool `getReward()` sends `7.5 CRV` as reward to contract
- `reward.remaining` = 10
- `bal` = 7.5
- `bal - reward.remaining` will fail due to underflow
- 0.5 day later, Alice deposits `500` lpToken to `_pid = 1`, most of the reward tokens will be left in the contract:
- convexPool `getReward()` sends `15 CRV` as reward to the contract;
- `d_reward = bal - reward.remaining` = 5
- `1 CRV` got sent to `treasury`, `4 CRV` sent to `claimContract`, `10 CRV` left in the contract;
- `rewards[1][0].remaining` = 15

Expected Results:

All the `15 CRV` get distributed: `3 CRV` to the `treasury`, and `12 CRV` to `claimContract`.

Actual Results:

Only `5 CRV` got distributed. The other `10 CRV` got left in the contract which can be frozen in the contract, see below for the details:

5. Bob deposits `1,000` lpToken to `_pid = 2`
6. convexPool `getReward()` sends `0 CRV` as reward to the contract
7. `d_reward = bal - reward.remaining` = 10

8. `2 CRV` sent to `treasury`, `8 CRV` sent to `claimContract` without calling `pushReward()`, so the `8 CRV` are now frozen in `claimContract`;
9. `rewards[2][0].remaining` = 10

Impact

- The two most important methods: `deposit()` and `withdraw()` will frequently fail as the tx will revert at `_calcRewardIntegral()`;
- Rewards distributed to users can often be fewer than expected;
- If there are different pools that use the same token as rewards, part of the rewards can be frozen at `claimContract` and no one can claim them.

Recommended Mitigation Steps

Consider comparing the `balanceOf` reward token before and after `getReward()` to get the actual rewarded amount, and `reward.remaining` should be removed.

[leekt (Concur) confirmed]

[Alex the Entreprenerd (judge) commented]:

> The warden has shown how `_calcRewardIntegral` can be broken in multiple ways.

> While I believe a set of similar findings have been reported, this one is extremely well written so I think this can stand on it's own.

> Because `_calRewardIntegral` is a core functionality of the contract (giving out reward) and the warden has shown how it can be broken, I agree with High Severity.

[H-07] Shelter `claimed` mapping is set with `_to` address and not `msg.sender`

Submitted by 0xliumin, also found by cmichel, leastwood, and pauliax

Any user can withdraw all the funds from the shelter. This is done by calling withdraw repeatedly until all funds are drained. You only need to have a small share.

Even if the `claimed` mapping was checked, there would still be a vulnerability. This is because the `claimed` mapping is updated with the `_to` address, not the `msg.sender` address.

Recommended Mitigation Steps

Remediation is to change the `_to` to `msg.sender`.
[Shelter.sol#L55]

[leekt (Concur) confirmed]

[Alex the Entreprenerd (judge) increased severity to High and commented]:

> Am marking this as a unique finding as this one shows another issue with the Shelter withdraw function.

> Because this also allows for draining of all rewards, am raising to High Severity.

[H-08] `MasterChef.sol` Users won't be able to receive the `concur` rewards

Submitted by WatchPug, also found by hickuphh3 and leastwood

According to:

- [README]
- Implementation of `deposit()`: [/contracts/MasterChef.sol#L157-L180]

MasterChef is only recording the deposited amount in the states, it's not actually holding the `depositToken`.

`depositToken` won't be transferred from `_msgSender()` to the MasterChef contract.

Therefore, in `updatePool()` L140 `lpSupply = pool.depositToken.balanceOf(address(this))` will always be `0`. And the `updatePool()` will be returned at L147.

MasterChef.sol#L135-L154

```
    function updatePool(uint _pid) public {
        PoolInfo storage pool = poolInfo[_pid];
        if (block.number <= pool.lastRewardBlock) {
            return;
        }
        uint lpSupply = pool.depositToken.balanceOf(address(this));
        if (lpSupply == 0 || pool.allocPoint == 0) {
            pool.lastRewardBlock = block.number;
            return;
        }
        if(block.number >= endBlock) {
            pool.lastRewardBlock = block.number;
            return;
        }

        uint multiplier = getMultiplier(pool.lastRewardBlock, block.
        uint concurReward = multiplier.mul(concurPerBlock).mul(pool.
        pool.accConcurPerShare = pool.accConcurPerShare.add(concurRe
        pool.lastRewardBlock = block.number;
    }
```

Impact

- The MasterChef contract fail to implement the most essential function;
- Users won't be able to receive any `Concur` rewards from MasterChef;

Recommended Mitigation Steps

Consider creating a receipt token to represent the invested token and use the receipt tokens in MasterChef.

See: https://github.com/convex-eth/platform/blob/883ffd4ebcaee12e64d18f75bdfe404bcd900616/contracts/contracts/Booster.sol#L272-L277

ryuheimat (Concur) confirmed

Alex the Entreprenerd (judge) commented:

> The warden has identified a logical flaw in the `Masterchef` contract.
>
> The contract is expecting `lpTokens` (deposited in another depositor contract) to be in the `Masterchef` at the time in which `updatePool` is called.
>
> However, due to the fact that the `lpToken` will be somewhere else, a more appropriate check would be to ask the depositor contract for the total supply.
>
> Given this finding, the Masterchef contract will always reward 0 tokens.
>
> This should classify the finding as Medium Severity (loss of Yield).
>
> However, because the finding shows how this can happen reliably, and effectively breaks the purpose of the contract, I believe High Severity to be more appropriate.

[H-09] deposit in `ConvexStakingWrapper` will most certainly revert

Submitted by wuwe1, also found by WatchPug

ConvexStakingWrapper.sol#L94-L99

```
            address mainPool = IRewardStaking(convexBooster)
                .poolInfo(_pid)
                .crvRewards;
            if (rewards[_pid].length == 0) {
                pids[IRewardStaking(convexBooster).poolInfo(_pid).lp
                convexPool[_pid] = mainPool;
```

`convexPool[_pid]` is set to `IRewardStaking(convexBooster).poolInfo(_pid).crvRewards;`

`crvRewards` is a `BaseRewardPool` like this one: https://etherscan.io/address/0x8B55351ea358e5Eda371575B031ee24F462d503e#code.

`BaseRewardPool` does not implement `poolInfo`

ConvexStakingWrapper.sol#L238

```
    IRewardStaking(convexPool[_pid]).poolInfo(_pid).lptoken
```

Above line calls `poolInfo` of `crvRewards` which causes revert.

Recommended Mitigation Steps

According to Booster's code

https://etherscan.io/address/0xF403C135812408BFbE8713b5A23a04b3D48AAE31#code

```
        //deposit lp tokens and stake
        function deposit(uint256 _pid, uint256 _amount, bool _stake)
            require(!isShutdown,"shutdown");
            PoolInfo storage pool = poolInfo[_pid];
            require(pool.shutdown == false, "pool is closed");

            //send to proxy to stake
            address lptoken = pool.lptoken;
            IERC20(lptoken).safeTransferFrom(msg.sender, staker, _am
```

`convexBooster` requires `poolInfo[_pid].lptoken`.

change L238 to

```
    IRewardStaking(convexBooster).poolInfo(_pid).lptoken
```

leekt (Concur) confirmed

Alex the Entreprenerd (judge) commented:

> The warden has shown how an improper assumption about the pool contract can cause reverts.

> While the risk of loss of funds is non-existent because all calls will revert, I believe the core functionality of the code is broken. For that reason, I think High Severity to be the proper severity.

[H-10] `ConvexStakingWrapper.exitShelter()` Will Lock LP Tokens, Preventing Users From Withdrawing

Submitted by leastwood

The shelter mechanism provides emergency functionality in an effort to protect users' funds. The `enterShelter` function will withdraw all LP tokens from the pool, transfer them to the shelter contract and activate the shelter for the target LP token. Conversely, the `exitShelter` function will deactivate the shelter and transfer all LP tokens back to the `ConvexStakingWrapper.sol` contract.

Unfortunately, LP tokens aren't restaked in the pool, causing LP tokens to be stuck within the contract. Users will be unable to withdraw their LP tokens as the `withdraw` function attempts to `withdrawAndUnwrap` LP tokens from the staking pool. As a result, this function will always revert due to insufficient staked balance. If other users decide to deposit their LP tokens, then these tokens can be swiped by users who have had their LP tokens locked in the contract.

This guarantees poor UX for the protocol and will most definitely lead to LP token loss.

Proof of Concept

ConvexStakingWrapper.sol#L121-L130

```
    function exitShelter(uint256[] calldata _pids) external onlyOwne
        for(uint256 i = 0; i<_pids.length; i++){
            IRewardStaking pool = IRewardStaking(convexPool[_pids[i]
            IERC20 lpToken = IERC20(
                pool.poolInfo(_pids[i]).lptoken
            );
            amountInShelter[lpToken] = 0;
            shelter.deactivate(lpToken);
        }
    }
```

[ConvexStakingWrapper.sol#L309-L331]()

```
    function withdraw(uint256 _pid, uint256 _amount)
        external
        nonReentrant
        whenNotInShelter(_pid)
    {
        WithdrawRequest memory request = withdrawRequest[_pid][msg.s
        require(request.epoch < currentEpoch() && deposits[_pid][msg
        require(request.amount >= _amount, "too much");
        _checkpoint(_pid, msg.sender);
        deposits[_pid][msg.sender].amount -= uint192(_amount);
        if (_amount > 0) {
            IRewardStaking(convexPool[_pid]).withdrawAndUnwrap(_amou
            IERC20 lpToken = IERC20(
                IRewardStaking(convexPool[_pid]).poolInfo(_pid).lptc
            );
            lpToken.safeTransfer(msg.sender, _amount);
            uint256 pid = masterChef.pid(address(lpToken));
            masterChef.withdraw(msg.sender, pid, _amount);
        }
        delete withdrawRequest[_pid][msg.sender];
        //events
        emit Withdrawn(msg.sender, _amount);
    }
```

Tools Used

Manual code review.
Confirmation from Taek.

Recommended Mitigation Steps

Consider re-depositing LP tokens upon calling `exitShelter`. This should ensure the same tokens can be reclaimed by users wishing to exit the `ConvexStakingWrapper.sol` contract.

leekt (Concur) confirmed

Alex the Entreprenerd (judge) commented:

> The warden has identified how through a combination of using the shelter and sending funds back, the funds would actually end up being stuck and non-withdrawable by depositors.

> I believe that generally speaking this would be a Medium Severity finding as the funds would be stuck if the sponsor were to activate the shelter and then send the tokens back (conditionality).

> However, the warden has shown that the system of Contract + Shelter is effectively broken, and for this reason I believe the finding is of High Severity.

[H-11] `ConvexStakingWrapper._calcRewardIntegral()` Can Be Manipulated To Steal Tokens From Other Pools

Submitted by leastwood, also found by cmichel and kirk-baird

The `ConvexStakingWrapper.sol` implementation makes several modifications to the original design. One of the key changes is the ability to add multiple pools into the wrapper contract, where each pool is represented by a unique `_pid`. By doing this, we are able to aggregate pools and their LP tokens to simplify the token distribution process.

However, the interdependence between pools introduces new problems. Because the original implementation uses the contract's reward token balance to track newly claimed tokens, it is possible for a malicious user to abuse the unguarded `getReward` function to maximise the profit they are able to generate. By calling `getReward` on multiple pools with the same reward token (i.e. `cvx`), users are able to siphon rewards from other pools. This inevitably leads to certain loss of rewards for users who have deposited LP tokens into these victim pools. As `crv` and `cvx` are reward tokens by default, it is very likely that someone will want to exploit this issue.

Proof of Concept

Let's consider the following scenario:

- There are two convex pools with `_pid` 0 and 1.
- Both pools currently only distribute `cvx` tokens.
- Alice deposits LP tokens into the pool with `_pid` 0.
- Both pools earn 100 `cvx` tokens which are to be distributed to the holders of the two pools.
- While Alice is a sole staker of the pool with `_pid` 0, the pool with `_pid` 1 has several stakers.
- Alice decides she wants to maximise her potential rewards, so she directly calls the unguarded `IRewardStaking(convexPool[_pid]).getReward` function on both pools, resulting in 200 `cvx` tokens being sent to the contract.
- She then decides to deposit the 0 amount to execute the `_calcRewardIntegral` function on the pool with `_pid` 0. However, this function will calculate `d_reward` as `bal - reward.remaining` which is effectively the change in contract balance. As we have directly claimed `cvx` tokens over the two pools, this `d_reward` will be equal to 200.
- Alice is then entitled to the entire 200 tokens as she is the sole staker of her pool. So instead of receiving 100 tokens, she is able to siphon rewards from other pools.

Altogether, this will lead to the loss of rewards for other stakers as they are unable to then claim their rewards.

[ConvexStakingWrapper.sol#L216-L259]

```
    function _calcRewardIntegral(
        uint256 _pid,
        uint256 _index,
        address _account,
        uint256 _balance,
        uint256 _supply
    ) internal {
        RewardType memory reward = rewards[_pid][_index];

        //get difference in balance and remaining rewards
        //getReward is unguarded so we use remaining to keep track o
        uint256 bal = IERC20(reward.token).balanceOf(address(this));
        uint256 d_reward = bal - reward.remaining;
        // send 20 % of cvx / crv reward to treasury
        if (reward.token == cvx || reward.token == crv) {
            IERC20(reward.token).transfer(treasury, d_reward / 5);
            d_reward = (d_reward * 4) / 5;
        }
        IERC20(reward.token).transfer(address(claimContract), d_rewa

        if (_supply > 0 && d_reward > 0) {
            reward.integral =
                reward.integral +
                uint128((d_reward * 1e20) / _supply);
        }

        //update user integrals
        uint256 userI = userReward[_pid][_index][_account].integral;
        if (userI < reward.integral) {
            userReward[_pid][_index][_account].integral = reward.int
            claimContract.pushReward(
                _account,
                reward.token,
                (_balance * (reward.integral - userI)) / 1e20
            );
        }

        //update remaining reward here since balance could have char
        if (bal != reward.remaining) {
            reward.remaining = uint128(bal);
        }

        rewards[_pid][_index] = reward;
    }
```

Tools Used

Manual code review.
Confirmation from Taek.

Recommended Mitigation Steps

Consider redesigning this mechanism such that all pools have their `getReward` function called in `_checkpoint`. The `_calcRewardIntegral` function can then ensure that each pool is allocated only a fraction of the total rewards instead of the change in contract balance. Other implementations might be more ideal, so it is important that careful consideration is taken when making these changes.

leekt (Concur) confirmed

Alex the Entreprenerd (judge) commented:

> The warden has shown how, by having the same token as rewards for multiple pools, the math for claiming can be broken, allowing the depositor of one pool to claim a portion of the token reward earned by all pools.

> Normally this would be contingent on implementation or overlap of the tokens, however, because we're dealing with CVX we already know for certain that CVX and cvxCRV is going to be a reward for the majority of the pools.

> This finding ultimately shows how to break the accounting of the reward contract while stealing yield from all other pools, and for that reason, I believe High Severity to be valid.

Medium Risk Findings (31)

[M-01] Deposits after the grace period should not be allowed

Submitted by pauliax

Shelter.sol#L34
Shelter.sol#L54

Function donate in Shelter shouldn't allow new deposits after the grace period ends, when the claim period begins.
Otherwise, it will be possible to increase savedTokens[_token], and thus new user

claim amounts will increase after some users might already have withdrawn their shares.

Recommended Mitigation Steps

Based on my understanding, it should contain this check:

```
    require(activated[_token] + GRACE_PERIOD > block.timestamp, "t
```

[leekt (Concur) confirmed]

[Alex the Entreprenerd (judge) commented]:

> Separating between a rescuing period and a redemption period does make sense and avoids losing on future rewards by withdrawing early.

> The sponsor confirmed, I believe medium severity to be appropriate.

[M-02] Unconstrained fee

Submitted by Czar102, also found by defsec, Dravee, harleythedog, hickuphh3, and throttle

[MasterChef.sol#L86-L101]

Token fee in `MasterChef` can be set to more than 100%, (for example, by accident) causing all `deposit` calls to fail due to underflow on subtraction when reward is lowered by the fee, thus breaking essential mechanics. Note that after the fee has been set to any value, it cannot be undone. A token cannot be removed, added, or added the second time. Thus, mistakenly (or deliberately, maliciously) added fee that is larger than 100% will make the contract impossible to recover from not being able to use the token.

Recommended Mitigation Steps

On setting fee ensure that it is below a set maximum, which is set to no more than 100%.

ryuheimat (Concur) confirmed

Alex the Entreprenerd (judge) commented:

> The warden has identified admin privilege that would enable them to set the deposit fee to 100%.
> The value can also be increased above 100% to cause a denial of service to the user.

> Mitigation would require offering a more appropriate upper limit to the fee.

[M-03] `USDMPegRecovery.sol#withdraw()` withdraw may often fail

Submitted by WatchPug

Per the doc:

> USDM deposits are locked based on the KPI's from carrot.eth.

> 3Crv deposits are not locked.

USDMPegRecovery.sol#L110-L128

```
    function withdraw(Liquidity calldata _withdrawal) external {
            Liquidity memory total = totalLiquidity;
            Liquidity memory user = userLiquidity[msg.sender];
            if(_withdrawal.usdm > 0) {
                require(unlockable, "!unlock usdm");
                usdm.safeTransfer(msg.sender, uint256(_withdrawal.us
                total.usdm -= _withdrawal.usdm;
                user.usdm -= _withdrawal.usdm;
            }

            if(_withdrawal.pool3 > 0) {
                pool3.safeTransfer(msg.sender, uint256(_withdrawal.p
                total.pool3 -= _withdrawal.pool3;
                user.pool3 -= _withdrawal.pool3;
            }
            totalLiquidity = total;
            userLiquidity[msg.sender] = user;
            emit Withdraw(msg.sender, _withdrawal);
        }
```

However, because the `withdraw()` function takes funds from the balance of the contract, once the majority of the funds are added to the curve pool via `provide()`. The `withdraw()` may often fail due to insufficient funds in the balance.

Proof of Concept

1. Alice deposits `4M` USDM and `4M` pool3 tokens;
2. Guardian calls `provide()` and all the `usdm` and `pool3` to `usdm3crv`;
3. Alice calls `withdraw()`, the tx will fail, due to insufficient balance.

Recommended Mitigation Steps

Consider calling `usdm3crv.remove_liquidity_one_coin()` when the balance is insufficient for the user's withdrawal.

leekt (Concur) confirmed

Alex the Entreprenerd (judge) commented:

> The warden has identified a specific scenario in which user funds would not be withdrawable

> Because the code uses internal storage for accounting rather than "value" this scenario can happen fairly reliably.

> I believe mitigation requires further thought than just withdrawing and ideally it would be best to setup a system similar to Vault Shares so that a withdrawal could be triggered either by available liquidity or via a withdrawal from the pool.

> I think Medium severity is appropriate.

[M-04] `USDMPegRecovery.sol#provide()` Improper design/implementation make it often unable to add liquidity to the `usdm3crv` pool

Submitted by WatchPug

USDMPegRecovery.sol#L73-L82

```
    function provide(uint256 _minimumLP) external onlyGuardian {
        require(usdm.balanceOf(address(this)) >= totalLiquidity.usdm
        // truncate amounts under step
        uint256 addingLiquidity = (usdm.balanceOf(address(this)) / s
        // match usdm : pool3 = 1 : 1
        uint256[2] memory amounts = [addingLiquidity, addingLiquidit
        usdm.approve(address(usdm3crv), addingLiquidity);
        pool3.approve(address(usdm3crv), addingLiquidity);
        usdm3crv.add_liquidity(amounts, _minimumLP);
    }
```

In the current implementation of `USDMPegRecovery.sol#provide()`, `addingLiquidity` is calculated solely based on `usdm` balance (truncate at a step of 250k), and it always uses the same amount of 3pool tokens to add_liquidity with.

Based on other functions of the contract, the balance of `usdm` can usually be more than the `pool3` balance, in that case, `usdm3crv.add_liquidity()` will fail.

Impact

When the balance of `pool3` is less than `usdm` (which is can be a common scenario), funds cannot be added to the curve pool.

For example:

When the contract got 5M of USDM and 4.2M of `pool3` tokens, it won't be possible to call `provide()` and add liquidity to the `usdm3crv` pool, as there are not enough pool3 tokens to match the 5M of USDM yet.

We expect it to add liquidity with 4M of USDM and 4M of pool3 tokens in that case.

Recommended Mitigation Steps

Change to:

```
    function provide(uint256 _minimumLP) external onlyGuardian {
        require(usdm.balanceOf(address(this)) >= totalLiquidity.usdm
        uint256 tokenBalance = Math.min(usdm.balanceOf(address(this)
        // truncate amounts under step
        uint256 addingLiquidity = (tokenBalance / step) * step;
        // match usdm : pool3 = 1 : 1
        uint256[2] memory amounts = [addingLiquidity, addingLiquidit
        usdm.approve(address(usdm3crv), addingLiquidity);
        pool3.approve(address(usdm3crv), addingLiquidity);
        usdm3crv.add_liquidity(amounts, _minimumLP);
    }
```

ryuheimat (Concur) confirmed

Alex the Entreprenerd (judge) commented:

> I agree with the finding, ultimately liquidity addition won't be at a 1:1 rate and the code won't adapt to that situation causing reverts.

[M-05] USDM locked unless guardian remove liquidity

Submitted by gzeon

In README.me:

> USDM deposits are locked based on the KPI's from carrot.eth

However, USDM deposits are also locked until guardian remove liquidity because there are no mechanism to remove deposited USDM in `withdraw`.

USDMPegRecovery.sol#L90

leekt (Concur) confirmed

Alex the Entreprenerd (judge) commented:

> The warden has identified Admin Privilege in that the deposit contract is controlled by the admin and liquidity LPd into the Curve Pool cannot be withdrawn by user (although code for redemption is present).

> Ultimately a refactoring that transforms this contract in something similar to a Yield Bearing Vault would solve for accounting while allowing an easier time adding and removing liquidity.

> Personally I'd recommend the sponsor to denominate the deposit token in the CRV_LP token to avoid issues with Single Sided Exposure, which other findings in this contest already discuss.

[M-06] `StakingRewards.sol` `recoverERC20()` can be used as a backdoor by the `owner` to retrieve `rewardsToken`

Submitted by WatchPug, also found by cmichel

StakingRewards.sol#L166-L176

```
    function recoverERC20(address tokenAddress, uint256 tokenAmo
        external
        onlyOwner
    {
        require(
            tokenAddress != address(stakingToken),
            "Cannot withdraw the staking token"
        );
        IERC20(tokenAddress).safeTransfer(owner(), tokenAmount);
        emit Recovered(tokenAddress, tokenAmount);
    }
```

Impact

Users can lose all the rewards to the malicious/compromised `owner`.

Recommended Mitigation Steps

Change to:

```
     function recoverERC20(
        address tokenAddress,
        address to,
        uint256 amount
    ) external onlyOwner {
        require(tokenAddress != address(stakingToken) && tokenAddres

        IERC20(tokenAddress).safeTransfer(to, amount);
        emit Recovered(tokenAddress, to, amount);
    }
```

[ryuheimat (Concur) confirmed]()

[Alex the Entreprenerd (judge) decreased severity to Medium and commented]():

> Agree with the finding, ultimately a simple check would provide stronger security guarantees.

> Because this is contingent on a malicious owner, I believe Medium Severity to be more appropriate.

[M-07] Fee-on-transfer token donations in `Shelter` break withdrawals

Submitted by cmichel, also found by Dravee, IllIllI, and Ruhum

[Shelter.sol#L34]()

The `Sheler.donate` function `transferFrom`s `_amount` and adds the entire `_amount` to `savedTokens[_token]`.
But the actual received token amount from the transfer can be less for fee-on-transfer tokens.

The last person to withdraw will not be able to as `withdraw` uses a share computation for the entire `savedTokens[_token]` amount.
The calculated `amount` will then be higher than the actual contract balance.

```
    function donate(IERC20 _token, uint256 _amount) external {
        require(activated[_token] != 0, "!activated");
        savedTokens[_token] += _amount;
        // @audit fee-on-transfer. then fails for last person in `wi
        _token.safeTransferFrom(msg.sender, address(this), _amount);
    }

    function withdraw(IERC20 _token, address _to) external override
        // @audit percentage on storage var, not on actual balance
        uint256 amount = savedTokens[_token] * client.shareOf(_toker
        // @audit amount might not be in contract anymore as savedTc
        _token.safeTransfer(_to, amount);
    }
```

Recommended Mitigation Steps

In `donate`, add only the actual transferred amounts (computed by `post-transfer balance - pre-transfer balance`) to `savedTokens[_token]`.

leekt (Concur) acknowledged

Alex the Entreprenerd (judge) commented:

> The warden has identified a specific interaction between a `feeOnTransfer` token and the Shelter Contract.

> Because the Shelter Contract can receive any token, and anyone could claim them based on percentage, and because some people will lose the ability to claim due to the internal accounting being incorrect, I believe that in this instance the finding is valid, and of medium severity.

[M-08] Donated Tokens Cannot Be Recovered If A Shelter Is Deactivated

Submitted by leastwood, also found by reassor

The shelter mechanism can be activated and deactivated on a target LP token. The owner of the `ConvexStakingWrapper.sol` contract can initiate the shelter whereby LP tokens are sent to the `Shelter.sol` contract. However, if the owner decides to deactivate the shelter before the grace period has passed, all LP tokens are transferred back to the `ConvexStakingWrapper.sol` contract. Donated tokens are also sent back to the contract. As a result, these tokens do not actually belong to any user and will effectively be lost in the contract.

Proof of Concept

Shelter.sol#L32-L36

```
    function donate(IERC20 _token, uint256 _amount) external {
        require(activated[_token] != 0, "!activated");
        savedTokens[_token] += _amount;
        _token.safeTransferFrom(msg.sender, address(this), _amount);
    }
```

ConvexStakingWrapper.sol#L107-L130

```
    function enterShelter(uint256[] calldata _pids) external onlyOwr
        for(uint256 i = 0; i<_pids.length; i++){
            IRewardStaking pool = IRewardStaking(convexPool[_pids[i]
            uint256 amount = pool.balanceOf(address(this));
            pool.withdrawAndUnwrap(amount, false);
            IERC20 lpToken = IERC20(
                pool.poolInfo(_pids[i]).lptoken
            );
            amountInShelter[lpToken] = amount;
            lpToken.safeTransfer(address(shelter), amount);
            shelter.activate(lpToken);
        }
    }

    function exitShelter(uint256[] calldata _pids) external onlyOwne
        for(uint256 i = 0; i<_pids.length; i++){
            IRewardStaking pool = IRewardStaking(convexPool[_pids[i]
            IERC20 lpToken = IERC20(
                pool.poolInfo(_pids[i]).lptoken
            );
            amountInShelter[lpToken] = 0;
            shelter.deactivate(lpToken);
        }
    }
```

Recommended Mitigation Steps

Consider allocating donated LP tokens to the contract owner when a shelter is deactivated. This can be done by checking for an excess of LP tokens. Anything greater than `amountInShelter` can be considered as donated.

leekt (Concur) acknowledged

Alex the Entreprenerd (judge) commented:

> The warden has indentified a way to potentially lose tokens, ultimately the contract could be rewritten to constantly allow redemption.

> For those reasons, and because the sponsor acknowledged, I believe the finding to be valid and of medium severity.

[M-09] `StakingRewards.sol#notifyRewardAmount()` Improper reward balance checks can make some users unable to withdraw their rewards

Submitted by WatchPug

StakingRewards.sol#L154-L158

```
        uint256 balance = rewardsToken.balanceOf(address(this));
        require(
            rewardRate <= balance / rewardsDuration,
            "Provided reward too high"
        );
```

In the current implementation, the contract only checks if balanceOf `rewardsToken` is greater than or equal to the future rewards.

However, under normal circumstances, since users can not withdraw all their rewards in time, the balance in the contract contains rewards that belong to the users but have not been withdrawn yet. This means the current checks can not be sufficient enough to make sure the contract has enough amount of rewardsToken.

As a result, if the `rewardsDistribution` mistakenly `notifyRewardAmount` with a larger amount, the contract may end up in a wrong state that makes some users unable to claim their rewards.

Proof of Concept

Given:

- rewardsDuration = 7 days;
- Alice stakes `1,000` stakingToken;
- `rewardsDistribution` sends `100` rewardsToken to the contract;
- `rewardsDistribution` calls `notifyRewardAmount()` with `amount = 100`;
- 7 days later, Alice calls `earned()` and it returns `100` rewardsToken, but Alice choose not to `getReward()` for now;
- `rewardsDistribution` calls `notifyRewardAmount()` with `amount = 100` without send any fund to contract, the tx will succees;
- 7 days later, Alice calls `earned()` `200` rewardsToken, when Alice tries to call `getReward()`, the transaction will fail due to insufficient balance of rewardsToken.

Expected Results:

The tx in step 5 should revert.

Recommended Mitigation Steps

Consider changing the function `notifyRewardAmount` to `addRward` and use `transferFrom` to transfer rewardsToken into the contract:

```
    function addRward(uint256 reward)
        external
        updateReward(address(0))
    {
        require(
            msg.sender == rewardsDistribution,
            "Caller is not RewardsDistribution contract"
        );

        if (block.timestamp >= periodFinish) {
            rewardRate = reward / rewardsDuration;
        } else {
            uint256 remaining = periodFinish - block.timestamp;
            uint256 leftover = remaining * rewardRate;
            rewardRate = (reward + leftover) / rewardsDuration;
        }

        rewardsToken.safeTransferFrom(msg.sender, address(this), rev

        lastUpdateTime = block.timestamp;
        periodFinish = block.timestamp + rewardsDuration;
        emit RewardAdded(reward);
    }
```

ryuheimat (Concur) confirmed

Alex the Entreprenerd (judge) decreased severity to Medium and commented:

> Given the code available, the warden has shown a possible scenario where certain depositors cannot receive reward tokens.

> Because this is contingent on an improper configuration and because this relates to loss of Yield, I believe Medium Severity to be more appropriate.

[M-10] Users Will Lose Rewards If The Shelter Mechanism Is Enacted Before A Recent Checkpoint

Submitted by leastwood

The shelter mechanism aims to protect the protocol's users by draining funds into a separate contract in the event of an emergency. However, while users are able to reclaim their funds through the `Shelter.sol` contract, they will still have a deposited balance from the perspective of `ConvexStakingWrapper.sol`.

Because users will only receive their rewards upon depositing/withdrawing their funds due to how the checkpointing mechanism works, it is likely that by draining funds to the `Shelter.sol` contract, users will lose out on any rewards they had accrued up and until that point. These rewards are unrecoverable and can potentially be locked within the contract if the reward token is unique and only belongs to the sheltered `_pid`.

Proof of Concept

[ConvexStakingWrapper.sol](#)

Recommended Mitigation Steps

Consider allowing users to call a public facing `_checkpoint` function once their funds have been drained to the `Shelter.sol` contract. This should ensure they receive their fair share of rewards. Careful consideration needs to be made when designing this mechanism, as by giving users full control of the `_checkpoint` function may allow them to continue receiving rewards after they have withdrawn their LP tokens.

[leekt (Concur) confirmed](#)

[Alex the Entreprenerd (judge) commented](#):

> The warden has shown how using the Shelter can cause depositors to lose yield they had accrued.

> Specifically the loss of yield will be for the time of new rewards accrued since the last `_checkpoint`.

> Moving to a global index (to compare user accrual against global rewards), or modifying the shelter to account for yield could be a potential way to mitigate.

> The sponsor confirmed and I believe medium severity to be appropriate because this is a Owner Privilege + Yield Loss finding.

[M-11] `ConvexStakingWrapper.enterShelter()` May Erroneously Overwrite `amountInShelter` Leading To Locked Tokens

Submitted by leastwood

The shelter mechanism provides emergency functionality in an effort to protect users' funds. The `enterShelter` function will withdraw all LP tokens from the pool, transfer them to the shelter contract and activate the shelter for the target LP token. If this function is called again on the same LP token, the `amountInShelter` value is overwritten, potentially by the zero amount. As a result its possible that the shelter is put in a state where no users can withdraw from it or only a select few users with a finite number of shares are able to. Once the shelter has passed its grace period, these tokens may forever be locked in the shelter contract.

Proof of Concept

ConvexStakingWrapper.sol#L107-L119

```
    function enterShelter(uint256[] calldata _pids) external onlyOwr
        for(uint256 i = 0; i<_pids.length; i++){
            IRewardStaking pool = IRewardStaking(convexPool[_pids[i]
            uint256 amount = pool.balanceOf(address(this));
            pool.withdrawAndUnwrap(amount, false);
            IERC20 lpToken = IERC20(
                pool.poolInfo(_pids[i]).lptoken
            );
            amountInShelter[lpToken] = amount;
            lpToken.safeTransfer(address(shelter), amount);
            shelter.activate(lpToken);
        }
    }
```

ConvexStakingWrapper.sol#L132-L135

```
    function totalShare(IERC20 _token) external view override returr
        // this will be zero if shelter is not activated
        return amountInShelter[_token];
    }
```

Recommended Mitigation Steps

Consider adding to the `amountInShelter[lpToken]` mapping instead of overwriting it altogether. This will allow `enterShelter` to be called multiple times with no loss of funds for the protocol's users.

[ryuheimat (Concur) confirmed]

[Alex the Entreprenerd (judge) commented]:

> The finding is valid, the owner can overwrite the `amountInShelter` storage variable causing the math in `Shelter.withdraw`. This will cause issues with distributing previously sheltered tokens.

> I believe using += instead of overwriting the value may be a sufficient remediation.

> Because this is contingent on a "distracted / malicious" admin, I believe Medium Severity to be appropriate.

[M-12] `USDMPegRecovery.provide()` Will Fail If There Is An Excess Of `usdm` Tokens

Submitted by leastwood

The `provide` function does not take a `_steps` argument and will instead calculate `addingLiquidity` by truncating amounts under `step`. As a result, if there is an excess of `usdm` such that the truncated amount exceeds the contract's `pool3` truncated balance, then the function will revert due to insufficient `pool3` collateral.

This will prevent guardians from effectively providing liquidity whenever tokens are available. Consider the following example:

- The contract has `500000e18` `usdm` tokens and `250000e18` `pool3` tokens.
- `addingLiquidity` will be calculated as `500000e18 / 250000e18 * 250000e18`.
- The function will attempt to add `500000e18` `usdm` and `pool3` tokens in which there are insufficient `pool3` tokens in the contract. As a result, it will revert even though there is an abundance of tokens that satisfy the `step` amount.

Proof of Concept

USDMPegRecovery.sol#L73-L82

```
    function provide(uint256 _minimumLP) external onlyGuardian {
        require(usdm.balanceOf(address(this)) >= totalLiquidity.usdm
        // truncate amounts under step
        uint256 addingLiquidity = (usdm.balanceOf(address(this)) / s
        // match usdm : pool3 = 1 : 1
        uint256[2] memory amounts = [addingLiquidity, addingLiquidit
        usdm.approve(address(usdm3crv), addingLiquidity);
        pool3.approve(address(usdm3crv), addingLiquidity);
        usdm3crv.add_liquidity(amounts, _minimumLP);
    }
```

Tools Used

Manual code review.
Discussions with Taek.

Recommended Mitigation Steps

Consider modifying the `provide` function such that a `_steps` argument can be supplied. This will allow guardians to maximise the amount of liquidity provided to the Curve pool.

[leekt (Concur) confirmed]()

[Alex the Entreprenerd (judge) commented]():

> The warden identified a logical fallacy that would prevent the code from providing liquidity.

> This is because the code is only accounting for one token, ignoring the other token's amount.

> Given the information I have, I agree with validity and severity of the finding. Mitigation could be achieved by following the warden's advice or by also using the balance of the `pool3` token to calculate the LP amounts.

[M-13] `StakingRewards.recoverERC20` allows owner to rug the `rewardsToken`

Submitted by Alex the Entreprenerd, also found by pauliax

[StakingRewards.sol#L166]()

`StakingRewards.recoverERC20` rightfully checks against the `stakingToken` being sweeped away.
However, there's no check against the `rewardsToken` which over time will sit in this contract.

This is the case of an admin privilege, which allows the owner to sweep the rewards tokens, perhaps as a way to rug depositors.

Proof of Concept

Calling `StakingRewards.recoverERC20(rewardsToken, rewardsToken.balanceOf(this))` enables the `owner` to sweep the token.

Recommended Mitigation Steps

Add an additional check

```
            require(
                tokenAddress != address(rewardsToken),
                "Cannot withdraw the rewards token"
            );
```

[leekt (Concur) confirmed]()

Alex the Entreprenerd (judge) commented:

> Because I'm judging the contest, I am forfeiting any warden winnings.

> The sponsor confirms and I believe this to be medium severity as it is contingent on a malicious owner.
> Adding the extra check removes the rug vector.

[M-14] Owner can steal Concur rewards

Submitted by Czar102

MasterChef.sol#L78-L80
MasterChef.sol#L157-L180

Owner can steal Concur rewards by adding a depositor and inflating other depositors' assigned balance of the token within the contract. Thus, the owner-managed depositor can get most (all but one wei) of the created tokens.

Recommended Mitigation Steps

Do not allow the owner to add depositors after the depositors have been configured.

ryuheimat (Concur) disputed and commented:

> Owner is a multisig & timelock. New depositors can be added later as well.

Alex the Entreprenerd (judge) decreased severity to Medium and commented:

> I think the warden could have done a better job at writing a POC.

> That said the finding is valid. The sponsor could set a `depositor` to be any EOA and because there's no transfer of tokens the balances could be inflated. Setting an immutable depositor would bring stronger security guarantees instead of allowing any contract to become a depositor.

> Because this is contingent on admin privilege, I believe medium severity to be more appropriate.

[M-15] Owner can lock tokens in `MasterChef`

Submitted by Czar102, also found by csanuragjain and Jujic

MasterChef.sol#L82-L84

Owner can remove a depositor. Since only depositors can deposit and withdraw, the owner may add a contract to the whitelist, let users deposit in the contract and remove the depositor from the whitelist. Depositor's reward cannot be withdrawn then. And takes a share of Concur tokens that will not be distributed.

Recommended Mitigation Steps

Remove `onlyDepositor` modifier from the `withdraw` function.

leekt (Concur) disputed

Alex the Entreprenerd (judge) decreased severity to Medium and commented:

> The finding is valid in that the sponsor / owner can remove all depositors.

> I believe having an immutable depositor that can't be changed would give stronger security guarantees.

> While the finding is valid, because it is contingent on a malicious owner, I believe Medium Severity to be more appropriate.

[M-16] Rewards get diluted because `totalAllocPoint` can only increase.

Submitted by throttle

MasterChef.sol

There is no functionality for removing pools/setting pool's allocPoints. Therefore `totalAllocPoint` only increases and rewards for pool decreases.

Proof of Concept

Scenario:

1. Owner adds new pool (first pool) for staking with points = 900 (totalAllocPoint=900).
2. 1 week passes.
3. First pool staking period ends (or for other reasons that pool is not meaningfully anymore).
4. Owner adds new pool (second pool) for staking with points = 100 (totalAllocPoint=1000).
5. 1 block later Alice stake 10 tokens there (at the same time).
6. 1 week passes.
7. After some time Alice claims rewards. But she is eligible only for 10% of the rewards. 90% goes to unused pool.

Recommended Mitigation Steps

Add functionality for removing pool or functionality for setting pool's `totalAllocPoint` param.

ryuheimat (Concur) confirmed

Alex the Entreprenerd (judge) decreased severity to Medium and commented:

> While the problem can seem trivial, the warden has proven that the contract can over time end up leaking excess value as any additional pool will dilute the `totalAllocPoint` and old pools cannot be retired.

> The sponsor also confirms.

> I believe the finding to be valid, but because the leak is contingent on settings, I believe Medium Severity to be more appropriate.

[M-17] Deactivate function can be bypassed

Submitted by csanuragjain, also found by gzeon

onlyClient can deactivate a token even after deadline is passed and transfer all token balance to itself.

Proof of Concept

1. Navigate to contract Shelter.sol
2. Observe that token can only be deactivated if activated[_token] + GRACE_PERIOD > block.timestamp. We will bypass this
3. onlyClient activates a token X using the activate function
4. Assume Grace period is crossed such that activated[_token] + GRACE_PERIOD < block.timestamp
5. Now if onlyClient calls deactivate function, it fails with "too late"
6. But onlyClient can bypass this by calling activate function again on token X which will reset the timestamp to latest in activated[_token] and hence onlyClient can now call deactivate function to disable the token and retrieve all funds present in the contract to his own address

Recommended Mitigation Steps

Add below condition to activate function:

```
function activate(IERC20 _token) external override onlyClient {
require(activated[_token]==0, "Already activated");
        activated[_token] = block.timestamp;
        savedTokens[_token] = _token.balanceOf(address(this));
        emit ShelterActivated(_token);
    }
```

leekt (Concur) confirmed

Alex the Entreprenerd (judge) decreased severity to Medium and commented:

> The warden has identified a way for the client to trick the shelter into sending all tokens to the client, effectively rugging all other users.

> Because this is contingent on a malicious client, I believe Medium Severity to be more appropriate.

[M-18] Users Will Lose Concur Rewards If The Shelter Mechanism Is Enacted On A Pool

Submitted by leastwood

The shelter mechanism aims to protect the protocol's users by draining funds into a separate contract in the event of an emergency. However, while users are able to reclaim their funds through the `Shelter.sol` contract, they will still have a deposited balance from the perspective of `ConvexStakingWrapper.sol`.

However, if the shelter mechanism is enacted before users are able to claim their Concur rewards, any accrued tokens will be lost and the `MasterChef.sol` contract will continue to allocate tokens to the sheltered pool which will be forever locked within this contract.

There is currently no way to remove sheltered pools from the `MasterChef.sol` contract, hence any balance lost in the contract cannot be recovered due to a lack of a sweep mechanism which can be called by the contract owner.

Proof of Concept

ConvexStakingWrapper.sol

MasterChef.sol

Recommended Mitigation Steps

Consider removing sheltered pools from the `MasterChef.sol` Concur token distribution. It is important to ensure `massUpdatePools` is called before making any changes to the list of pools. Additionally, removing pools from this list may also create issues with how `_pid` is produced on each new pool. Therefore, it may be worthwhile to rethink this mechanism such that `_pid` tracks some counter variable and not `poolInfo.length - 1`.

leekt (Concur) confirmed

Alex the Entreprenerd (judge) decreased severity to Medium and commented:

> The warden has shown how the Shelter Mechanism can cause depositors to lose the unharvested field they were entitled to.

> This is contingent on the Shelter being used by the admin.

> For that reason (and because the finding is for loss of yield), I believe Medium Severity to be more appropriate.

[M-19] Rogue pool in Shelter

Submitted by 0x1f8b

Shelter.sol#L38-L42

Shelter contract can steal user tokens.

Proof of Concept

Shelter `client` can call `activate` on an already activated token, this will reset its start time, so if the client activate a token when it `GRACE_PERIOD` is almost finished, it will reset this time.
This will prevent the user to call `withdraw` because the condition `activated[_token] + GRACE_PERIOD < block.timestamp` but will allow the client to call `deactivate` and receive all funds from the users because it will satisfy the condition `activated[_token] + GRACE_PERIOD > block.timestamp`.

Steps:

- client `activate` tokenA.
- Users deposit tokenA using `donate`.
- client `activate` tokenA again until they has enough tokens.
- More users use `donate`.
- client deactivate tokenA and receive all tokens.

Recommended Mitigation Steps

- Avoid `activate` twice for the same token
- `donate` only after the `GRACE_PERIOD`

[leekt (Concur) disputed]

[Alex the Entreprenerd (judge) decreased severity to Medium and commented]:

> I believe the finding to be valid. The warden has shown how the Shelter design allows the client to repeatedly call `activate` to prevent anyone from withdrawing the tokens.

> Because this is contingent on a malicious admin, I believe Medium Severity to be more appropraite.

[M-20] `MasterChef.updatePool()` Fails To Update Reward Variables If `block.number >= endBlock`

Submitted by leastwood, also found by CertoraInc, csanuragjain, Czar102, hickuphh3, kirk-baird, and WatchPug

The `updatePool` function intends to calculate the accumulated Concur rewards by tracking the number of blocks passed since the last update to correctly determine how many Concur tokens to distribute to each share. The reward distribution has a start and end block which dictates the timeframe by which rewards will be distributed to the underlying pool.

If a pool has not recently updated itself and has reached the `block.number >= endBlock` statement in `updatePool`, then any rewards that it would normally be entitled to prior to reaching `endBlock` will not be attributed to the pool. Therefore, once rewards are no longer being distributed, pools who had not recently called `updatePool` before reaching `endBlock` are at a disadvantage as compared to more active pools.

Proof of Concept

MasterChef.sol#L135-L154

```
    // Update reward variables of the given pool to be up-to-date.
    function updatePool(uint _pid) public {
        PoolInfo storage pool = poolInfo[_pid];
        if (block.number <= pool.lastRewardBlock) {
            return;
        }
        uint lpSupply = pool.depositToken.balanceOf(address(this));
        if (lpSupply == 0 || pool.allocPoint == 0) {
            pool.lastRewardBlock = block.number;
            return;
        }
        if(block.number >= endBlock) {
            pool.lastRewardBlock = block.number;
            return;
        }

        uint multiplier = getMultiplier(pool.lastRewardBlock, block.
        uint concurReward = multiplier.mul(concurPerBlock).mul(pool.
        pool.accConcurPerShare = pool.accConcurPerShare.add(concurRe
        pool.lastRewardBlock = block.number;
    }
```

Recommended Mitigation Steps

Ensure that once the `block.number >= endBlock` statement has been reached, the `pool.accConcurPerShare` is updated to reflect the number of blocks that have passed up until `endBlock`. The number of blocks should be equal to `endBlock - pool.lastRewardBlock`. This will ensure stale pools are not negatively impacted once `endBlock` has been reached by the contract.

ryuheimat (Concur) confirmed

Alex the Entreprenerd (judge) commented:

> The warden has identified a way in which the contract will not release rewards that are due for a depositor.

> Because the check doesn't accrue until the last eligible block, the reward loss can be quantified as:
> LastTimeAccrueBeforeEndBlock - endBlock

> The finding is valid, but because it pertains to loss of yield, and because the loss can be quantified and reduced by simply calling at the last available block, I believe Medium Severity to be more appropriate.

[M-21] [ConcurRewardPool] Possible reentrancy when claiming rewards

Submitted by ShadowyNoobDev, also found by 0xw4rd3n, CertoraInc, ckksec, Czar102, defsec, Alex the Entreprenerd, Heartless, IllIllI, Jujic, kirk-baird, leastwood, pauliax, peritoflores, Randyyy, reassor, Rhynorater, Sleepy, SolidityScan, and wuwe1

ConcurRewardPool.sol#L34

Since the reward tokens are transferred before the balances are set to 0, it is possible to perform a reentrancy attack if the reward token has some kind of call back functionality e.g. ERC777. pBTC is an ERC777 token that is currently available on Convex. A similar attack occurred with imBTC on uniswap v1.

Proof of Concept

- Preparation

  1. Assume that pBTC is used as extra rewards for this victim convex pool.
  2. A malicious user interacts with Concur through a smart contract. He follows the standard flow and has some rewards to be claimed.
  3. The malicious user interacts with this smart contract to register a bad `tokensToSend()` callback function through the ERC-1820 contract.
  4. In this `tokensToSend()` function, he calls `ConcurRewardPool.claimRewards()` n-1 more times to drain contract.

- Attack

  1. When he calls `ConcurRewardPool.claimRewards()` for the first time, the pBTC reward tokens are transferred.

2. You can see from the pBTC contract on line 871 that `_callTokensToSend(from, from, recipient, amount, "", "");` is called inside the `transfer()` function.
3. If you trace to the `_callTokensToSend` function definition to line 1147, you will notice that it calls `IERC777Sender(implementer).tokensToSend(operator, from, to, amount, userData, operatorData);` on line 1159.
4. Since the malicious user already registered a bad `tokensToSend()` function, this function will be called thus draining majority of the pBTC rewards available on the `ConcurRewardPool` contract.

You can also find a walkthrough replicating a similar attack here.

Recommended Mitigation Steps

- Use a nonReentrant modifier
- set balances to 0 first before disbursing the rewards

ryuheimat (Concur) confirmed

Alex the Entreprenerd (judge) commented:

> The warden has shown how using a specific reward token can lead to reentrancy for the function `claimRewards`.

> Because the finding is contingent on a specific token that enables the exploit, I believe Medium Severity to be appropriate.

[M-22] If The Staking Token Exists In Both `StakingRewards.sol` And `ConvexStakingWrapper.sol` Then It Will Be Possible To Continue Claiming Concur Rewards After The Shelter Has Been Activated

Submitted by leastwood

Staking tokens are used to deposit into the `StakingRewards.sol` and `ConvexStakingWrapper.sol` contracts. Once deposited, the user is entitled to Concur rewards in proportion to their staked balance and the underlying pool's `allocPoint` in the `MasterChef.sol` contract.

The `Shelter.sol` mechanism allows the owner of the `ConvexStakingWrapper.sol` to react to emergency events and protect depositor's assets. The staking tokens can be withdrawn after the grace period has passed. However, these staking tokens can be deposited into the `StakingRewards.sol` contract to continue receiving Concur rewards not only for `StakingRewards.sol` but also for their `ConvexStakingWrapper.sol` deposited balance which has not been wiped. As a result, users are able to effectively claim double the amount of Concur rewards they should be receiving.

Proof of Concept

MasterChef.sol

StakingRewards.sol

ConvexStakingWrapper.sol

Recommended Mitigation Steps

Ensure that staking tokens cannot be deposited in both the `StakingRewards.sol` and `ConvexStakingWrapper.sol` contracts. If this is intended behaviour, it may be worthwhile to ensure that the sheltered users have their deposited balance wiped from the `MasterChef.sol` contract upon being sheltered.

leekt (Concur) confirmed

Alex the Entreprenerd (judge) decreased severity to Medium and commented:

> The warden has shown that because the shelter mechanism doesn't wipe the balance in the contract, those same tokens can be used to further break the accounting of the contract, with the goal of extracting further rewards.

> While I believe the wardens work is commendable and have considered High Severity because the accounting of the protocol has been broken, I believe Medium Severity to be more appropriate because the finding:

- Is contingent on the Shelter being used
- There must be more rewards in the StakingRewardsContract
- The impact is limited to the additional rewards and nothing else

[M-23] Transfer to treasury can register as succeeded when failing in `_calcRewardIntegral`

Submitted by 0xw4rd3n

[StakingRewards.sol#L126]()

If the transfer of the reward token fails to the treasury (due to insufficient funds for example), the function `_calcRewardIntegral` will still update accounting and cause devastating accounting discrepancies in the contract.

Proof of Concept

Provide direct links to all referenced code in GitHub. Add screenshots, logs, or any other relevant proof that illustrates the concept.

Recommended Mitigation Steps

`require(IERC20(reward.token).transfer(treasury, d_reward / 5), "ERROR_MESSAGE");`

[ryuheimat (Concur) confirmed]()

[Alex the Entreprenerd (judge) decreased severity to Medium and commented]():

> All in all this report is the classic “No safeApprove”. But with an actual idea of a POC.
> Ultimately the risk is contingent on the specific reward.token being a nonRevertingOnError.

> For that reason, I believe Medium Severity to be more appropriate.

[M-24] Rewards distribution can be disrupted by a early user

Submitted by WatchPug

[ConvexStakingWrapper.sol#L184-L188]

```
    if (_supply > 0 && d_reward > 0) {
        reward.integral =
            reward.integral +
            uint128((d_reward * 1e20) / _supply);
    }
```

`reward.integral` is `uint128`, if an early user deposits with just `1` Wei of `lpToken`, and make `_supply == 1`, and then transferring `5e18` of `reward_token` to the contract.

As a result, `reward.integral` can exceed `type(uint128).max` and overflow, causing the rewards distribution to be disrupted.

Recommended Mitigation Steps

Consider `wrap` a certain amount of initial totalSupply at deployment, e.g. `1e8`, and never burn it. And consider using uint256 instead of uint128 for `reward.integral`. Also, consider lower `1e20` down to `1e12`.

[ryuheimat (Concur) confirmed]

[Alex the Entreprenerd (judge) decreased severity to Medium and commented]:

> The warden has shown a way to break the uint128 accounting system in place.

> This is contingent on frontrunning the pool and depositing a small amount to cause the division to fail.
> Additionally, this will cause a DOS that prevents other people from depositing.

> I believe that this could be unstuck by continuously (via loop) depositing 1 wei as to slowly increase the totalSupply again.

> Mitigation can be attained by either refactoring or by ensuring that the first deposit is big enough (18 decimals) to keep numbers to rational values.

> Because the finding is contingent on a setup and because tokens will be rescuable, I believe Medium Severity to be more appropriate.

[M-25] `ConvexStakingWrapper#deposit()` depositors may lose their funds when the `_amount` is huge

Submitted by WatchPug, also found by danb, gzeon, Heartless, and pauliax

When the value of `_amount` is larger than `type(uint192).max`, due to unsafe type casting, the recorded deposited amount can be much smaller than their invested amount.

[ConvexStakingWrapper.sol#L228-L250]

```
    function deposit(uint256 _pid, uint256 _amount)
        external
        whenNotPaused
        nonReentrant
    {
        _checkpoint(_pid, msg.sender);
        deposits[_pid][msg.sender].epoch = currentEpoch();
        deposits[_pid][msg.sender].amount += uint192(_amount);
        if (_amount > 0) {
            IERC20 lpToken = IERC20(
                IRewardStaking(convexPool[_pid]).poolInfo(_pid).lptc
            );

            lpToken.safeTransferFrom(msg.sender, address(this), _amc
            lpToken.safeApprove(convexBooster, _amount);
            IConvexDeposits(convexBooster).deposit(_pid, _amount, tı
            lpToken.safeApprove(convexBooster, 0);
            uint256 pid = masterChef.pid(address(lpToken));
            masterChef.deposit(msg.sender, pid, _amount);
        }
        emit Deposited(msg.sender, _amount);
    }
```

Proof of Concept

When `_amount = uint256(type(uint192).max) + 1`:

- At L235, `uint192(_amount) = 0`, `deposits[_pid][msg.sender].amount = 0`;
- At L241, `uint256(type(uint192).max) + 1` will be transferFrom `msg.sender`.

Expected results:

```
deposits[_pid][msg.sender].amount == uint256(type(uint192).max) + 1;
```

Actual results:

```
deposits[_pid][msg.sender].amount = 0.
```

The depositor loses all their invested funds.

Recommended Mitigation Steps

Consider adding a upper limit for the `_amount` parameter:

```
    require(_amount <= type(uint192).max, "...");
```

ryuheimat (Concur) confirmed

Alex the Entreprenerd (judge) decreased severity to Medium and commented:

> The warden has shown how casting without safe checks can cause the accounting to break and cause end users to lose deposited tokens.

> While the finding has merit, I believe that because this applies to niche situations, and is conditional on specific inputs, that Medium Severity is more appropriate.

[M-26] `StakingRewards.setRewardsDuration` allows setting near zero or enormous `rewardsDuration`, which breaks reward logic

Submitted by hyh

notifyRewardAmount will be inoperable if rewardsDuration is set to zero. If will cease to produce meaningful results if rewardsDuration is too small or too big.

Proof of Concept

The setter does not control the value, allowing zero/near zero/enormous duration:

[StakingRewards.sol#L178-185]()

Division by the duration is used in notifyRewardAmount:

[StakingRewards.sol#L143-156]()

Recommended Mitigation Steps

Check for min and max range in the rewardsDuration setter, as too small or too big rewardsDuration breaks the logic.

[ryuheimat (Concur) confirmed]()

[Alex the Entreprenerd (judge) commented]():

> Finding is valid. Ultimately contingent on admin privilege so I believe Medium Severity to be appropriate.

[M-27] `MasterChef.sol` A `depositor` can deposit an arbitrary amount without no cost

Submitted by WatchPug, also found by cmichel

The owner of `MasterChef.sol` can add a `depositor` with `addDepositor()`.

MasterChef.sol#L78-L80

```
    function addDepositor(address _depositor) external onlyOwner {
        isDepositor[_depositor] = true;
    }
```

A `depositor` can deposit with an arbitrary amount, without any cost.

MasterChef.sol#L157-L180

```
    function deposit(address _recipient, uint _pid, uint _amount) ex
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][_msgSender()];
        updatePool(_pid);

        if(user.amount > 0) {
            uint pending = user.amount * pool.accConcurPerShare / _c
            if (pending > 0) {
                safeConcurTransfer(_recipient, pending);
            }
        }

        if (_amount > 0) {
            if (pool.depositFeeBP > 0) {
                uint depositFee = _amount.mul(pool.depositFeeBP).div
                user.amount = SafeCast.toUint128(user.amount + _amou
            } else {
                user.amount = SafeCast.toUint128(user.amount + _amou
            }
        }

        user.rewardDebt = SafeCast.toUint128(user.amount * pool.accC
        emit Deposit(_recipient, _pid, _amount);
    }
```

This allows a malicious/compromised depositor to take the majority share (nearly 100%) of all pools simply by calling `deposit()` with extremely large amounts, and take all the rewards.

Recommended Mitigation Steps

See the `Recommendation` section on [issue #200]() and remove the `depositor` role.

[ryuheimat (Concur) disputed and commented]():

> This function will be called by whitelisted depositor contract and the actual token transfer will be done there.

[Alex the Entreprenerd (judge) decreased severity to Medium and commented]():

> Because the're a mapping that allows any contract to be set as depositor, I believe the warden has shown a potential admin privilege that can cause issues with the fair distribution of rewards.

> Because this is contingent on a malicious admin, I believe Medium Severity to be more appropriate.

> Mitigation would be as simple as having one depositor, alternative a more complicated architecture may need to be used.

[M-28] During stake or deposit, users would not be rewarded the correct Concur token, when MasterChef has under-supply of it

Submitted by hubble, also found by CertoraInc and Czar102

During stake or deposit, users would not be transferred the correct Concur token, when MasterChef has under-supply of it.

There is an assumption that MasterChef contract would own enough Concur tokens so as to distribute to users as reward, during deposit or withdraw. But say, due to excess user activity, MasterChef runs out of Concur tokens. All deposits & withdraws that happen after that, would have zero transfer of Concur token to the user. This will continue until the MasterChef contract is replenished again.

Proof of Concept

[MasterChef.sol#L205-L206]()

Makeshift unit test Note: Temporarily modify the private function MasterChef.safeConcurTransfer to public function, for unit test validation

```
    //Unit Test starts
      it("MasterChef - Zero Concur balance", async function() {
        await concurToken.mint(masterChef.address, 100);
        console.log(await concurToken.balanceOf(masterChef.address),
        await masterChef.safeConcurTransfer(user1.address, 60); // u
        console.log(await concurToken.balanceOf(masterChef.address),
        await masterChef.safeConcurTransfer(user1.address, 60); // u
        console.log(await concurToken.balanceOf(masterChef.address),
        await masterChef.safeConcurTransfer(user1.address, 60); // u
        console.log(await concurToken.balanceOf(masterChef.address),
      });
    //Unit Test ends
```

Tools Used

Manual review, & makeshift Unit test

Recommended Mitigation Steps

Minimal recommended fix:

To `MasterChef.safeConcurTransfer` function, add the following require statement. This will at least ensure that, when there is zero balance in MasterChef contract, the safeConcurTransfer function will not succeed.

```
        function safeConcurTransfer(address _to, uint _amount) priva
            uint concurBalance = concur.balanceOf(address(this));
            require(concurBalance>0, "safeConcurTransfer: balance is
```

leekt (Concur) acknowledged

Alex the Entreprenerd (judge) commented:

> The finding is valid as in that depositors could not receive token incentives, am not fully convinced this should be of medium severity as ultimately the contract will eventually run out of tokens and as such this is a situation that has to be handled.

> If anything, reverting may cause the tokens to be stuck.

Alex the Entreprenerd (judge) commented:

> While I feel like this is a situational finding, it ultimately is a loss of yield finding with a very clear example.
> For that reason I believe Medium Severity to be appropriate.

[M-29] `ConvexStakingWrapper` deposits and withdraws will frequently be disabled if a token that doesn't allow zero value transfers will be added as a reward one

Submitted by hyh

If deposits and withdraws are done frequently enough, the reward update operation they invoke will deal mostly with the case when there is nothing to add yet, i.e. `reward.remaining` match the reward token balance.

If reward token doesn't allow for zero value transfers, the reward update function will fail on an empty incremental reward transfer, which is now done unconditionally, reverting the caller deposit/withdrawal functionality

Proof of Concept

When ConvexStakingWrapper isn't paused, every deposit and withdraw update current rewards via `_checkpoint` function before proceeding:

ConvexStakingWrapper.sol#L233

ConvexStakingWrapper.sol#L260

`_checkpoint` calls `_calcRewardIntegral` for each of the reward tokens of the pid:

ConvexStakingWrapper.sol#L220

`_calcRewardIntegral` updates the incremental reward for the token, running the logic even if reward is zero, which is frequently the case:

ConvexStakingWrapper.sol#L182

If the reward token doesn't allow zero value transfers, this transfer will fail, reverting the corresponding deposit or withdraw.

Recommended Mitigation Steps

Consider checking the reward before doing transfer (and the related computations as an efficiency measure):

Now:

```
    IERC20(reward.token).transfer(address(claimContract), d_reward);
```

To be:

```
    if (d_reward > 0)
            IERC20(reward.token).transfer(address(claimContract), d_
```

ryuheimat (Concur) confirmed

Alex the Entreprenerd (judge) commented:

> The warden has shown how, due to a pattern that always transfers the reward token to the claim contract, in the case of a 0 transfer, certain transfers could fail, causing reverts.

> While there can be an argument that this finding may not happen in reality, I believe that ultimately the system has been shown to be flawed in it's conception, perhaps adding a storage variable for the amount to claim would be more appropriate instead of dripping the rewards each time.

> For that reason, and because the finding is contingent on a reward token that does revert on 0 transfer, I believe Medium Severity to be appropriate.

[M-30] `StakingRewards` reward rate can be dragged out and diluted

Submitted by cmichel

[StakingRewards.sol#L161]()

The `StakingRewards.notifyRewardAmount` function receives a `reward` amount and extends the current reward end time to `now + rewardsDuration`.
It rebases the currently remaining rewards + the new rewards (`reward + leftover`) over this new `rewardsDuration` period.

```
    function withdraw(IERC20 _token, address _to) external override
        require(activated[_token] != 0 && activated[_token] + GRACE_
        // @audit uses `msg.sender`'s share but sets `claimed` for _
        uint256 amount = savedTokens[_token] * client.shareOf(_toker
        claimed[_token][_to] = true;
        emit ExitShelter(_token, msg.sender, _to, amount);
        _token.safeTransfer(_to, amount);
    }
```

This can lead to a dilution of the reward rate and rewards being dragged out forever by malicious new reward deposits.

Proof of Concept

Imagine the current rewardRate is `1000 rewards / rewardsDuration`.
20% of the `rewardsDuration` passed, i.e., `now = lastUpdateTime + 20% * rewardsDuration`.
A malicious actor notifies the contract with a reward of `0`: `notifyRewardAmount(0)`.
Then the new `rewardRate = (reward + leftover) / rewardsDuration = (0 + 800) / rewardsDuration = 800 / rewardsDuration`.
The `rewardRate` just dropped by 20%.
This can be repeated infinitely.
After another 20% of reward time passed, they trigger `notifyRewardAmount(0)` to reduce it by another 20% again:
`rewardRate = (0 + 640) / rewardsDuration = 640 / rewardsDuration`.

Recommended Mitigation Steps

Imo, the `rewardRate` should never decrease by a `notifyRewardAmount` call. Consider not extending the reward payouts by `rewardsDuration` on every call. `periodFinish` probably shouldn't change at all, the `rewardRate` should just increase by `rewardRate += reward / (periodFinish - block.timestamp)`.

Alternatively, consider keeping the `rewardRate` constant but extend `periodFinish` time by `+= reward / rewardRate`.

ryuheimat (Concur) disputed and commented:

> notifyRewardAmount check msg.sender's permission.

Alex the Entreprenerd (judge) commented:

> The warden is pointing out an admin privilege that would allow the admin to dilute current rewards.

> While the sponsor claims this won't happen, I can only judge based on the code that is available to me.
> And at this point there seems to be no code for the `rewardsDistribution` contract that would be calling `notifyRewardAmount`

> Given this, I believe the finding to be valid as the POC works out to demonstrate how a malicious owner could dilute the rewardRate.

> This would cause loss of yield for all depositors, which makes the finding of Medium Severity.

[M-31] execute in VoteProxy should be payable

Submitted by wuwe1

`execute` will revert when `msg.value > 0`

Proof of Concept

Lacking `payable` mutability specifier.

VoteProxy.sol#L28-L35

```
    function execute(
        address _to,
        uint256 _value,
        bytes calldata _data
    ) external onlyOwner returns (bool, bytes memory) {
        (bool success, bytes memory result) = _to.call{value: _v
        return (success, result);
    }
```

Recommended Mitigation Steps

Add `payable` mutability specifier.

leekt (Concur) confirmed

Alex the Entreprenerd (judge) commented:

> @leekt Can you tell me what you'd need `execute` to be used for?

> Do you really need it to be payable?

Alex the Entreprenerd (judge) commented:

> After some thinking, I do believe that it would be wise to allow for payable calls.

> Will mark as valid and because this is contingent on a specific usage, I think Medium Severity to be appropriate.

Low Risk and Non-Critical Issues

For this contest, 36 reports were submitted by wardens detailing low risk and non-critical issues. The report highlighted below by **hickuphh3** received the top score from the judge.

The following wardens also submitted reports: wuwe1, pauliax, kenta, IllIllI, WatchPug, hyh, gzeon, csanuragjain, 0x1f8b, SolidityScan, CertoraInc, samruna, cccz, defsec, Dravee, Randyyy, Ruhum, robee, Czar102, BouSalman, ShadowyNoobDev, throttle, ye0lde, Sleepy, cryptphi, harleythedog, kirk-baird, leastwood, peritoflores, rfa, Rhynorater, bitbopper, mtz, 0xw4rd3n, *and* hubble.

Codebase Impressions & Summary

Overall, code quality was fair. A number of contracts were taken from various sources, such as StakingRewards, Masterchef and the ConvexStakingWrapper. Modifications were made to include custom features like taking a 20% fee on CVX and CRV rewards for the treasury, and to not require stake token transfers for deposits / withdrawals into the Masterchef contract.

I found `10` high severity issues, majority of which are found in the Masterchef contract. They were simple logic bugs that would have been discovered with unit tests.

In addition, I made `2` medium severity, `7` low severity, and `1` non-critical findings.

Note that during the contest, an example shelter client was added and pushed to a new branch for wardens to understand how the shelter would operate. The integration of the ConvexStakingWrapper with the Shelter in that branch has a few bugs, but I assume it is outside the current contest scope to report them.

Due to the number of issues raised, I strongly recommend the team to write unit tests for their contracts, and to consider running a mitigation contest.

[L-01]: Masterchef: pendingConcur() shows increasing reward amounts after mining period ends

Line References

MasterChef.sol#L113-L124

Description

Even though rewards distribution cease after `endBlock`, `pendingConcur()` will calculate as if reward distribution has not.

Distribution of rewards will cease after `endBlock`, but `pendingConcur()` will show increasing pending rewards because it does not account for `endBlock`.

Recommended Mitigation Steps

```
function pendingConcur(uint _pid, address _user) external view r
        ...
        // take the minimum of endBlock and currentBlock
        uint endRewardBlock = endBlock >= block.number ? block.r
        if (endRewardBlock > pool.lastRewardBlock && lpSupply !=
                uint multiplier = getMultiplier(pool.lastRewardE
                ...
        }
        ...
}
```

[L-02]: Masterchef: `safeConcurTransfer()` potentially reverts for zero amount

Line References

[MasterChef.sol#L205-L210]

Description

If the contract has zero concur tokens, the following may revert because of zero amount. This is of course dependent on the concur token implementation.

```
// couple of lines omitted
transferSuccess = concur.transfer(_to, concurBalance);
require(transferSuccess, "safeConcurTransfer: transfer failed");
```

[L-03]: ConvexStakingWrapper: Small rounding error in `_calcRewardIntegral()`

Line References

[ConvexStakingWrapper.sol#L179-L180]()

Description

The treasury takes a 20% fee of rewards. The calculation will possibly leave 1 wei unaccounted for.

```
    IERC20(reward.token).transfer(treasury, d_reward / 5);
    d_reward = (d_reward * 4) / 5;
```

For instance, assume `d_reward = 21`. The treasury receives `4` wei while the user receives `16` wei, leaving 1 wei unaccounted for.

Recommended Mitigation Steps

```
    uint256 rewardFee = d_reward / 5;
    IERC20(reward.token).transfer(treasury, rewardFee);
    d_reward -= rewardFee;
```

[L-04]: USDMPegRecovery: 40M or 4M threshold?

Line References

[USDMPegRecovery.sol#L100]()

Description

The README says *"Once 40m USDM is deposited, 3Crv side of the contract starts accepting deposits."* However, the code accepts 3CRV deposits after 4M USDM is deposited instead.

Recommended Mitigation Steps

Specify the threshold as an internal constant, and use underscores for readability. I also recommend double-checking the values of declared variables in all contracts, such as `step` and `concurPerBlock`.

```
uint256 internal constant MIN_USDM_AMOUNT = 40_000_000e18;
require(totalLiquidity.usdm > MIN_USDM_AMOUNT, "usdm low");

// or
require(totalLiquidity.usdm > 40_000_000e18, "usdm low");
```

[N-01]: Masterchef: Incorrect comment on endBlock

Line References

[MasterChef.sol#L52-L53]

Description

`uint public endBlock; // The block number when mining starts.` is incorrect, as it should be the end of the mining period, not the start. Its comment applies to `startBlock`.

Note that `uint public startBlock` does not have a comment. Consider adding it.

Recommended Mitigation Steps

```
uint public startBlock; // The block number when mining starts.
uint public endBlock; // The block number when mining ends.
```

[N-02]: StakingRewards: Incorrect revert statement in `setRewardsDistribution()`

Line References

[StakingRewards.sol#L191-L194]

Description

`setRewardsDistribution()` has the following check:

```
    require(
      block.timestamp > periodFinish,
      "Previous rewards period must be complete before changing the
    );
```

The statement is incorrect because it's `rewardsDistribution` that is being changed, not the rewards duration.

Recommended Mitigation Steps

Actually, the check is redundant, because there is no harm changing `rewardsDistribution` while distribution is ongoing. I suggest removing the check entirely. Otherwise, change the comment to

`"Previous rewards period must be complete before changing rewardsDistribution"`

[N-03]: Masterchef: RADSs → Concurs

[MasterChef.sol#L25]

Rename `RADSs` to `Concurs`

[Alex the Entreprenerd (judge) commented]:

> [L-01]: Masterchef: pendingConcur() shows increasing reward amounts after mining period ends
> Valid finding

> [L-02]: Masterchef: safeConcurTransfer() potentially reverts for zero amount
> I don't believe it will cause issues, but think 0 check is low per industry standard.

> [L-03]: ConvexStakingWrapper: Small rounding error in _calcRewardIntegral()
> After further consideration, I agree.

> [L-04]: USDMPegRecovery: 40M or 4M threshold?
> I feel like this is the only case where I'd give low vs non-critical as the comment and the code have a meaningful, and significant difference for the end users.

> [N-01]: Masterchef: Incorrect comment on endBlock
> Non-critical IMO

> [N-02]: StakingRewards: Incorrect revert statement in setRewardsDistribution()
> Disagree with [low] severity.

> [N-03]: Masterchef: RADSs → Concurs
> Valid finding.

Report has plenty of content, formatting is good, I think most findings are over-emphasized though and under further scrutiny this is basically equivalent to 4 findings.

Alex the Entreprenerd (judge) commented:

> Adding #137 does make the report more well rounded and adding #136 makes this the most interesting report thus far, 6.5 findings at this time

> 6++ with very good formatting

Alex the Entreprenerd (judge) commented:

> [L-01]: Masterchef: pendingConcur() shows increasing reward amounts after mining period ends
> Low

> [L-02]: Masterchef: safeConcurTransfer() potentially reverts for zero amount
> Low

> [L-03]: ConvexStakingWrapper: Small rounding error in _calcRewardIntegral()
> Low

> [L-04]: USDMPegRecovery: 40M or 4M threshold?
> Low

> [N-01]: Masterchef: Incorrect comment on endBlock
> Non-Critical

> [N-02]: StakingRewards: Incorrect revert statement in setRewardsDistribution()
> Non Critical

> [N-03]: Masterchef: RADSs -> Concurs
> Non-Critical

> #137 -> Non-Critical

> #136 -> Low Severity

Gas Optimizations

For this contest, 33 reports were submitted by wardens detailing gas optimizations. The report highlighted below by **WatchPug** received the top score from the judge.

The following wardens also submitted reports: throttle, csanuragjain, Dravee, pauliax, 0x1f8b, Jujic, BouSalman, defsec, Ruhum, hickuphh3, rfa, robee, 0xngndev, kenta, ye0lde, gzeon, bitbopper, SolidityScan, wuwe1, 0x0x0x, 0xNotOrious, Tomio, 0x510c, Heartless, mtz, Randyyy, Sleepy, IllIllI, sabtikw, ShadowyNoobDev, peritoflores, *and* CertoraInc.

[G-01] Cache external call result in the stack can save gas

Note: Suggested optimation, save a decent amount of gas without compromising readability.

Every call to an external contract costs a decent amount of gas. For optimization of gas usage, external call results should be cached if they are being used for more than one time.

For example:

ConvexStakingWrapper.sol#L93-L140

```
    function addRewards(uint256 _pid) public {
        address mainPool = IRewardStaking(convexBooster)
            .poolInfo(_pid)
            .crvRewards;
        if (rewards[_pid].length == 0) {
            pids[IRewardStaking(convexBooster).poolInfo(_pid).lptoke
            // ...
        }
        // ...
    }
```

`IRewardStaking(convexBooster).poolInfo(_pid)` can be cached to avoid an extra external call.

[G-02] Cache external call result in storage can save gas

Note: Suggested optimation, save a decent amount of gas without compromising readability.

For the unchanged results of an external call that will be reused multiple times, cache and read from storage rather than initiate a fresh external call can save gas.

Instances include:

[ConvexStakingWrapper.sol#L237-L239]()

```
    IERC20 lpToken = IERC20(
        IRewardStaking(convexPool[_pid]).poolInfo(_pid).lptoken
    );
```

[ConvexStakingWrapper.sol#L264-L266]()>br

```
    IERC20 lpToken = IERC20(
        IRewardStaking(convexPool[_pid]).poolInfo(_pid).lptoken
    );
```

`lpToken` of `_pid` can be cached when `addRewards()` to avoid extra external calls.

[G-03] SafeMath is no longer needed

Note: Suggested optimation, save a decent amount of gas without compromising readability.

`SafeMath` is no longer needed starting with Solidity 0.8. The compiler now has built in overflow checking.

Removing `SafeMath` can save some gas.

Instances include:

[MasterChef.sol#L89-L89]

```
    totalAllocPoint = totalAllocPoint.add(_allocationPoints);
```

[MasterChef.sol#L109-L109]

```
    return _to.sub(_from);
```

[MasterChef.sol#L120-L121]

```
    uint concurReward = multiplier.mul(concurPerBlock).mul(pool.allo
    accConcurPerShare = accConcurPerShare.add(concurReward.mul(_conc
```

[G-04] Change unnecessary storage variables to constants can save gas

Note: Suggested optimation, save a decent amount of gas without compromising readability.

[MasterChef.sol#L56-L57]

```
    uint private _concurShareMultiplier = 1e18;
    uint private _perMille = 1000; // 100%
```

Some storage variables include `_concurShareMultiplier`, `_perMille` will never be changed and they should not be.

Changing them to `constant` can save gas.

[G-05] Setting `bool` variables to `false` is redundant

Note: Minor optimation, the amount of gas saved is minor, change when you see fit.

[MasterChef.sol#L204-L204]()

```
    bool transferSuccess = false;
```

Setting `bool` variables to `false` is redundant as they default to `false`.

See https://docs.soliditylang.org/en/v0.8.11/control-structures.html#default-value

[G-06] Using immutable variable can save gas

Note: Suggested optimation, save a decent amount of gas without compromising readability.

[MasterChef.sol#L52-L71]()

```
    uint public startBlock;
    uint public endBlock; // The block number when mining starts.
    IERC20 public concur;

    uint private _concurShareMultiplier = 1e18;
    uint private _perMille = 1000; // 100%

    constructor(IERC20 _concur, uint _startBlock, uint _endBlock) Ov
        startBlock = _startBlock;
        endBlock = _endBlock;
        concur = _concur;
        // ...
    }
```

Considering that `startBlock` , `endBlock` and `concur` will never change, changing them to immutable variables instead of storages variable can save gas.

StakingRewards.sol#L19-L20

```
    IERC20 public rewardsToken;
    IERC20 public stakingToken;
```

StakingRewards.sol#L37-L47

```
    constructor(
        address _rewardsDistribution,
        address _rewardsToken,
        address _stakingToken,
        MasterChef _masterChef
    ) {
        rewardsToken = IERC20(_rewardsToken);
        stakingToken = IERC20(_stakingToken);
        rewardsDistribution = _rewardsDistribution;
        masterChef = _masterChef;
    }
```

Considering that `rewardsToken` and `stakingToken` will never change, changing them to immutable variables instead of storages variable can save gas.

[G-07] Use short reason strings can save gas

Note: Minor optimation, the amount of gas saved is minor, change when you see fit.

Every reason string takes at least 32 bytes.

Use short reason strings that fits in 32 bytes or it will become more expensive.

Instances include:

StakingRewards.sol#L179-L182

```
    require(
        block.timestamp > periodFinish,
        "Previous rewards period must be complete before changing tł
    );
```

StakingRewards.sol#L191-L194

```
    require(
        block.timestamp > periodFinish,
        "Previous rewards period must be complete before changing tł
    );
```

StakingRewards.sol#L137-L140

```
    require(
        msg.sender == rewardsDistribution,
        "Caller is not RewardsDistribution contract"
    );
```

StakingRewards.sol#L170-L173

```
    require(
        tokenAddress != address(stakingToken),
        "Cannot withdraw the staking token"
    );
```

MasterChef.sol#L210-L210

```
    require(transferSuccess, "safeConcurTransfer: transfer failed");
```

[G-08] Setting `uint256` variables to `0` is redundant

Note: Minor optimation, the amount of gas saved is minor, change when you see fit.

Setting `uint256` variables to `0` is redundant as they default to `0`.

[StakingRewards.sol#L21-L22]

```
        uint256 public periodFinish = 0;
        uint256 public rewardRate = 0;
```

Recommended Mitigation Steps

Change to `uint256 public periodFinish; uint256 public rewardRate;` can make the code simpler and save some gas.

[G-09] Adding unchecked directive can save gas

Note: Minor optimation, the amount of gas saved is minor, change when you see fit.

For the arithmetic operations that will never over/underflow, using the unchecked directive (Solidity v0.8 has default overflow/underflow checks) can save some gas from the unnecessary internal over/underflow checks.

For example:

1. [StakingRewards.sol#L88-L101]

```
    function stake(uint256 amount)
        external
        nonReentrant
        whenNotPaused
        updateReward(msg.sender)
    {
        require(amount > 0, "Cannot stake 0");
        _totalSupply += amount;
        _balances[msg.sender] += amount;
        stakingToken.safeTransferFrom(msg.sender, address(this), amo
        uint256 pid = masterChef.pid(address(stakingToken));
        masterChef.deposit(msg.sender, pid, amount);
        emit Staked(msg.sender, amount);
    }
```

`_balances[msg.sender] += amount` will never overflow if `_totalSupply += amount;` does not revert.

2. [StakingRewards.sol#L103-L115]()

```
    function withdraw(uint256 amount)
        public
        nonReentrant
        updateReward(msg.sender)
    {
        require(amount > 0, "Cannot withdraw 0");
        _totalSupply -= amount;
        _balances[msg.sender] -= amount;
        stakingToken.safeTransfer(msg.sender, amount);
        uint256 pid = masterChef.pid(address(stakingToken));
        masterChef.withdraw(msg.sender, pid, amount);
        emit Withdrawn(msg.sender, amount);
    }
```

`_totalSupply -= amount` will never underflow if `_balances[msg.sender] -= amount;` does not underflow revert.

Therefore it can be changed to for gas saving:

```
        _balances[msg.sender] -= amount;
        unchecked {
            _totalSupply -= amount;
        }
```

[G-10] ”> 0” is less efficient than “!= 0” for unsigned integers

Note: Minor optimation, the amount of gas saved is minor, change when you see fit.

It is cheaper to use `!= 0` than `> 0` for uint256.

[StakingRewards.sol#L94-L94]()

```
    require(amount > 0, "Cannot stake 0");
```

[StakingRewards.sol#L108-L108]

```
    require(amount > 0, "Cannot withdraw 0");
```

[StakingRewards.sol#L119-L119]

```
    if (reward > 0) {
```

[G-11] `++i` is more efficient than `i++`

Note: Minor optimation, the amount of gas saved is minor, change when you see fit.

Using `++i` is more gas efficient than `i++`, especially in a loop.

For example:

[ConvexStakingWrapper.sol#L121-L121]

```
    for (uint256 i = 0; i < extraCount; i++) {
```

[ConvexStakingWrapper.sol#L219-L219]

```
    for (uint256 i = 0; i < rewardCount; i++) {
```

[ConcurRewardPool.sol#L35-L35]

```
    for (uint256 i = 0; i < _tokens.length; i++) {
```

[G-12] Reuse existing external call’s cache can save gas

Note: Suggested optimation, save a decent amount of gas without compromising readability.

ConvexStakingWrapper.sol#L120-L139

```
    uint256 extraCount = IRewardStaking(mainPool).extraRewardsLengtł
    for (uint256 i = 0; i < extraCount; i++) {
        address extraPool = IRewardStaking(mainPool).extraRewards(i)
        address extraToken = IRewardStaking(extraPool).rewardToken()
        if (extraToken == cvx) {
            //no-op for cvx, crv rewards
            rewards[_pid][CVX_INDEX].pool = extraPool;
        } else if (registeredRewards[_pid][extraToken] == 0) {
            //add new token to list
            rewards[_pid].push(
                RewardType({
                    token: IRewardStaking(extraPool).rewardToken(),
                    pool: extraPool,
                    integral: 0,
                    remaining: 0
                })
            );
            registeredRewards[_pid][extraToken] = rewards[_pid].lenç
        }
    }
```

`IRewardStaking(extraPool).rewardToken()` at L131 is already cached in the local variable `extraToken` at L123.

Reusing the cached local variable in the stack instead of initiating an external call again can save gas.

[G-13] Unnecessary checked arithmetic in for loops

Note: Non-preferred, the amount of gas saved is at cost of readability, only apply when gas saving is a top priority.

There is no risk of overflow caused by increamenting the iteration index in for loops (the `i++` in for `for (uint256 i = 0; i < rewardCount; i++)`).

Increments perform overflow checks that are not necessary in this case.

Recommended Mitigation Steps

Surround the increment expressions with an `unchecked { ... }` block to avoid the default overflow checks. For example, change the for loop:

ConvexStakingWrapper.sol#L219-L221

```
    for (uint256 i = 0; i < rewardCount; i++) {
        _calcRewardIntegral(_pid, i, _account, depositedBalance, sup
    }
```

to:

```
    for (uint256 i = 0; i < rewardCount;) {
        _calcRewardIntegral(_pid, i, _account, depositedBalance, sup
        unchecked { ++i; }
    }
```

[G-14] Cache array length in for loops can save gas

Note: Minor optimation, the amount of gas saved is minor, change when you see fit.

Reading array length at each iteration of the loop takes 6 gas (3 for mload and 3 to place memory_offset) in the stack.

Caching the array length in the stack saves around 3 gas per iteration.

Instances include:

- `ConcurRewardPool.sol#claimRewards()`

  ConcurRewardPool.sol#L35-L39

Alex the Entreprenerd (judge) commented:

> **[G-01] Cache external call result in the stack can save gas**
> Would save 100 gas for STATICALL + the cost of reading from Storage again, let's say another 100, at the cost of 6 for MSTORE + MLOAD = 194 gas saved

> **[G-02] Cache external call result in storage can save gas**
> Would save 91 gas (First call costs 6 more, second costs 97 less)

> **[G-03] S̀afeMath is no longer needed**
> In contrast to other findings the warden has listed all instances that are unnecessary, for thus reason I will add the gas savings to this report
> 8 instances, 20 gas per instance = 160 gas saved

> **[G-04] Change unnecessary storage variables to constants can save gas**
> Because the warden didn't list the other savings, I'll give one COLD SLOAD per variable
> 2100 * 2 = 4200

> **[G-05] Setting bool variables to false is redundant**
> 3 gas

> **[G-06] Using immutable variable can save gas**
> Similar to S4
> 5 * 2100 = 10500

> **[G-07] Use short reason strings can save gas**
> 5 * 2500 per discussion on other reports
> 12500

> **[G-08] Setting uint256 variables to 0 is redundant**
> 200

> **[G-09] Adding unchecked directive can save gas**
> 2 * 20 = 40

> **[G-10] ”> 0” is less efficient than “!= 0” for unsigned integers**
> Only for require
> 2 * 6 = 12

> **[G-11] ++i is more efficient than i++**
> 3 * 3 = 9

> **[G-12] Reuse existing external call's cache can save gas**
> 91 gas saved (97 - 6)

> **[G-13] Unnecessary checked arithmetic in for loops**
> 20 gas

> **[G-14] Cache array length in for loops can save gas**
> 3 gas

> Overall the report is short and sweet, the formatting is excellent and there's a little more detail than in other reports.
> Would have liked to see the storage to constant gas savings explicitly shown as that would have made the report as close to complete as it could have been.

> Total Gas Saved:
> 28023

Disclosures