veToken Finance contest Findings & Analysis Report

2022-10-25

Table of contents

- [M-06] Incorrectly set `_maxTime` to be in line with the locking `maxTime` of each veToken could render the `deposit` of this contract to be unfunctional or even freeze assets inside the contract
- [M-07] `VE3DRewardPool` and `VE3DLocker` adds to an unbounded array which may potentially lock all rewards in the contract
- [M-08] Not updating `totalWeight` when operator is removed in `VeTokenMinter`
- [M-09] in notifyRewardAmount() of VE3DRewardPool and BaseRewardPool some tokes will be locked and not distributed becasue of rounding error
- [M-10] Unable To Get Rewards If Admin Withdraws $VE3D tokens From `VeTokenMinter` Contract
- [M-11] Misconfiguration of Fees Incentive Might Cause Tokens To Be Stuck In `Booster` Contract
- [M-12] Malicious operator can rug pull
- [M-13] Unused rewards(because of totalSupply()==0 for some period) will be locked forever in VE3DRewardPool and BaseRewardPool
- [M-14] Deposited staking tokens can be lost if rewards token info added by mistake in addReward() in VE3DRewardPool and there is no checking to ensure this would not happen (ve3Token for one reward was equal to stacking token)
- [M-15] Owner should be allowed to change feeManager
- [M-16] Admin Privilege in minting to arbitrary address allows operator to dilute tokens
- [M-17] Missing sane bounds on asset weights
- [M-18] Governance can arbitrarily burn VeToken from any address
- [M-19] `VE3DRewardPool` claim in loop depend on pausable token
- [M-20] Contracts should be robust to upgrades of underlying gauges and eventually changes of the underlying tokens
- [M-21] `VoterProxy` incorrectly assumes a 1-1 mapping between the gauge and the LP tokens.

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 veToken Finance smart contract system written in Solidity. The audit contest took place between May 26—June 2 2022.

Wardens

76 Wardens contributed reports to the veToken Finance contest:

1. csanuragjain
2. xiaoming90
3. unforgiven
4. Picodes
5. IllIllI
6. kirk-baird
7. hyh
8. shenwilly
9. VAD37
10. oyc_109
11. Ruhum
12. sorrynotsorry
13. 0x52
14. Dravee
15. SmartSek (0xDjango and hake)
16. cryptphi
17. pauliax
18. sseefried
19. jonatascm
20. 0x1f8b

21. ch13fd357r0y3r
22. SecureZeroX
23. WatchPug (jtp and ming)
24. Kumpa
25. gzeon
26. TerrierLover
27. reassor
28. MiloTruck
29. Funen
30. minhquanym
31. 0xNazgul
32. sashik_eth
33. FSchmoede
34. _Adam
35. berndartmueller
36. cccz
37. robee
38. cogitoergosumsw
39. horsefacts
40. catchup
41. 0x29A (0x4non and rotcivegaf)
42. Hawkeye (0xwags and 0xmint)
43. hansfriese
44. simon135
45. 0xf15ers (remora and twojoy)
46. ellahi
47. c3phas
48. delfin454000
49. asutorufos

50. ElKu
51. z3s
52. dipp
53. Deivitto
54. BouSalman
55. GimelSec (rayn and sces60107)
56. 0xDjango
57. Chom
58. Tomio
59. Koustre
60. fatherOfBlocks
61. 0xKitsune
62. Kaiziron
63. TomJ
64. 0xkatana
65. Cityscape
66. Randyyy
67. RoiEvenHaim
68. sach1r0
69. saian
70. Waze

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 itsmetechjay.

Summary

The C4 analysis yielded an aggregated total of 31 unique vulnerabilities. Of these vulnerabilities, 1 received a risk rating in the category of HIGH severity and 30 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 51 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 48 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 veToken Finance contest repository](), and is composed of 6 smart contracts written in the Solidity programming language and includes 1,602 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 (1)

[H-01] Gauge Rewards Stuck In `VoterProxy` Contract When `ExtraRewardStashV3` Is Used Within Angle Deployment

Submitted by xiaoming90

> Note: This report aims to discuss the issue encountered when `ExtraRewardStashV3` is used within Angle Deployment. There is also another issue when `ExtraRewardStashV2` is used within Angle Deployment, but I will raise it in a separate report since `ExtraRewardStashV2` and `ExtraRewardStashV3` operate differently, and the proof-of-concept and mitigation are different too.

Proof of Concept

In this example, assume the following Angle's gauge setup

> Name = Angle sanDAI_EUR Gauge

> Symbol = SsanDAI_EUR

> reward_count = 2

> reward_tokens(0) = ANGLE

> reward_tokens(1) = DAI

> Gauge Contract: [LiquidityGaugeV4.vy]

> Stash Contract: [ExtraRewardStashV3]

To collect the gauge rewards, users would trigger the `Booster._earmarkRewards` function to claim veAsset and extra rewards from a gauge.

Per the code logic, the function will attempt to execute the following two key operations:

1. First Operation - Claim the veAsset by calling `VoterProxy.claimVeAsset`. Call Flow as follow: `VoterProxy.claimVeAsset() > IGauge(_gauge).claim_rewards()`.
2. Second Operation - Claim extra rewards by calling `ExtraRewardStashV3.claimRewards`. Call flow as follows:

`ExtraRewardStashV3.claimRewards > Booster.claimRewards > VoterProxy.claimRewards > IGauge(_gauge).claim_rewards()` .

Note that `IGauge(_gauge).claim_rewards()` will claim all available reward tokens from the Angle's gauge.

https://github.com/code-423n4/2022-05-vetoken/blob/2d7cd1f6780a9bcc8387dea8fecfbd758462c152/contracts/Booster.sol#L495

```
//claim veAsset and extra rewards and disperse to reward contrac
function _earmarkRewards(uint256 _pid) internal {
    PoolInfo storage pool = poolInfo[_pid];
    require(pool.shutdown == false, "pool is closed");

    address gauge = pool.gauge;

    //claim veAsset
    IStaker(staker).claimVeAsset(gauge);

    //check if there are extra rewards
    address stash = pool.stash;
    if (stash != address(0)) {
        //claim extra rewards
        IStash(stash).claimRewards();
        //process extra rewards
        IStash(stash).processStash();
    }
        ..SNIP..
}
```

First Operation - Claim the veAsset

Since this is a Angle Deployment, when the `VoterProxy.claimVeAsset` is triggered, it will go through the if-else logic (`escrowModle == IVoteEscrow.EscrowModle.ANGLE`) and execute `IGauge(_gauge).claim_rewards()` , and all rewards tokens will be sent to `VoterProxy` contract. Assume that `100 ANGLE` and `100 DAI` were received.

Note that in this example, we have two reward tokens (ANGLE and DAI). Additionally, gauge redirection was not configured on the gauge at this point, thus the gauge rewards will be sent to the caller, which is the `VoterProxy` contract.

Subsequently, the code `IERC20(veAsset).safeTransfer(operator, _balance);` will be executed, and veAsset (`100 ANGLE`) reward tokens will be transferred to the `Booster` contract for distribution. However, the `100 DAI` reward tokens will remain stuck in the `VoterProxy` contract. As such, users will not be able to get any reward tokens (e.g. DAI, WETH) except veAsset (ANGLE) tokens from the gauges.

https://github.com/code-423n4/2022-05-vetoken/blob/2d7cd1f6780a9bcc8387dea8fecfbd758462c152/contracts/VoterProxy.sol#L224

```
function claimVeAsset(address _gauge) external returns (uint256)
    require(msg.sender == operator, "!auth");

    uint256 _balance = 0;

    if (escrowModle == IVoteEscrow.EscrowModle.PICKLE) {
        try IGauge(_gauge).getReward() {} catch {
            return _balance;
        }
    } else if (
        escrowModle == IVoteEscrow.EscrowModle.CURVE ||
        escrowModle == IVoteEscrow.EscrowModle.RIBBON
    ) {
        try ITokenMinter(minter).mint(_gauge) {} catch {
            return _balance;
        }
    } else if (escrowModle == IVoteEscrow.EscrowModle.IDLE) {
        try ITokenMinter(minter).distribute(_gauge) {} catch {
            return _balance;
        }
    } else if (escrowModle == IVoteEscrow.EscrowModle.ANGLE) {
        try IGauge(_gauge).claim_rewards() {} catch {
            return _balance;
        }
    }

    _balance = IERC20(veAsset).balanceOf(address(this));
    IERC20(veAsset).safeTransfer(operator, _balance);
```

```
        return _balance;
    }
```

Following is Angle's Gauge Contract for reference:

https://github.com/AngleProtocol/angle-core/blob/4d854e0d74be703a3707898f26ea2dd4166bc9b6/contracts/staking/LiquidityGaugeV4.vy#L344

(Mainnet Deployed Address: https://etherscan.io/address/0x8E2c0CbDa6bA7B65dbcA333798A3949B07638026)

> Note: Angle Protocol is observed to use LiquidityGaugeV4 contract for all of their gauges. Thus, ExtraRewardStashV3 is utilised during pool creation.

```
    @external
    @nonreentrant('lock')
    def claim_rewards(_addr: address = msg.sender, _receiver: addres
        """
        @notice Claim available reward tokens for `_addr`
        @param _addr Address to claim for
        @param _receiver Address to transfer rewards to - if set to
                         ZERO_ADDRESS, uses the default reward recei
                         for the caller
        """
        if _receiver != ZERO_ADDRESS:
            assert _addr == msg.sender  \# dev: cannot redirect wher
        self._checkpoint_rewards(_addr, self.totalSupply, True, _rec
```

Second Operation - Claim extra rewards

After the `IStaker(staker).claimVeAsset(gauge);` code within the `Booster._earmarkRewards` function is executed, `IStash(stash).claimRewards();` and `IStash(stash).processStash();` functions will be executed next. `stash == ExtraRewardStashV3`.

The `ExtraRewardStashV3.claimRewards` will call the `Booster.setGaugeRedirect` first so that all the gauge rewards will be redirected to `ExtraRewardStashV3` stash contract. Subsequently, `ExtraRewardStashV3.claimRewards` will trigger `Booster.claimRewards` to claim the gauge rewards from the Angle's gauge.

Note that this is the second time the contract attempts to claim gauge rewards from the gauge. Thus, no gauge rewards will be received since we already claimed them earlier. Next, `ExtraRewardStashV3` will attempt to process all the tokens stored in its contract and send them to the respective reward contracts for distribution to the users. However, the contract does not have any tokens stored in it because the earlier attempt to claim gauge rewards return nothing.

As we can see, the DAI reward tokens are still stuck in the `VoterProxy` contract at this point.

https://github.com/AngleProtocol/angle-core/blob/4d854e0d74be703a3707898f26ea2dd4166bc9b6/contracts/staking/LiquidityGaugeV4.vy#L332

```
def set_rewards_receiver(_receiver: address):
    """
    @notice Set the default reward receiver for the caller.
    @dev When set to ZERO_ADDRESS, rewards are sent to the calle
    @param _receiver Receiver address for any rewards claimed vi
    """
    self.rewards_receiver[msg.sender] = _receiver
```

https://github.com/code-423n4/2022-05-vetoken/blob/2d7cd1f6780a9bcc8387dea8fecfbd758462c152/contracts/ExtraRewardStashV3.sol#L61

```
//try claiming if there are reward tokens registered
function claimRewards() external returns (bool) {
    require(msg.sender == operator, "!authorized");

    //this is updateable from v2 gauges now so must check each t
    checkForNewRewardTokens();
```

```
        //make sure we're redirected
        if (!hasRedirected) {
            IDeposit(operator).setGaugeRedirect(pid);
            hasRedirected = true;
        }

        uint256 length = tokenCount;
        if (length > 0) {
            //claim rewards on gauge for staker
            //using reward_receiver so all rewards will be moved to
            IDeposit(operator).claimRewards(pid, gauge);
        }
        return true;
    }
```

Impact

User's gauge rewards are frozen/stuck in `VoterProxy` contract. Additionally, there is no method to sweep/collect the reward tokens stuck in the `VoterProxy` contract.

Recommended Mitigation Steps

> Note: I do not see `Booster.setGaugeRedirect` being called in the deployment and testing scripts. Thus, it is fair to assume that the team is not aware of the need to trigger `Booster.setGaugeRedirect` during deployment. If the gauge redirection has been set to the stash contract `ExtraRewardStashV3` right from the start before anyone triggered the `earmarkRewards` function, this issue should not occur.

Consider triggering `Booster.setGaugeRedirect` during the deployment to set gauge redirection to stash contract (`ExtraRewardStashV3`) so that the Angle's gauge rewards will not be redirected to `VoterProxy` contract and get stuck there.

Alternatively, update the `Booster._earmarkRewards` to as follows:

```
    //claim veAsset and extra rewards and disperse to reward contrac
    function _earmarkRewards(uint256 _pid) internal {
            PoolInfo storage pool = poolInfo[_pid];
            require(pool.shutdown == false, "pool is closed");
```

```
        address stash = pool.stash;
        if (escrowModle == IVoteEscrow.EscrowModle.ANGLE) {
                //claims gauges rewards
                IStash(stash).claimRewards();
                //process gauges rewards
                IStash(stash).processStash();
        } else {
                //claim veAsset
        IStaker(staker).claimVeAsset(gauge);

        //check if there are extra rewards
        address stash = pool.stash;
        if (stash != address(0)) {
            //claim extra rewards
            IStash(stash).claimRewards();
            //process extra rewards
            IStash(stash).processStash();
        }
        }

        //veAsset balance
    uint256 veAssetBal = IERC20(veAsset).balanceOf(address(this)
        ..SNIP..
}
```

There is no need to specifically call `VoterProxy.claimVeAsset` to fetch ANGLE for Angle Protocol because calling `IStash(stash).claimRewards()` will fetch both ANGLE and other reward tokens from the gauge anyway. When the stash contract receives the ANGLE tokens, it will automatically transfer all of them back to `Booster` contract when `IStash(stash).processStash()` is executed. The `IStash(stash).claimRewards()` function also performs a sanity check to ensure that the gauge redirection is pointing to itself before claiming the gauge rewards, and automatically configure them if it is not, so it will not cause the reward tokens to get stuck in `VoterProxy` contract.

- Curve uses an older version of LiquidityGauge contract. Thus, two calls are needed (`Minter.mint` to claim CRV and `LiquidityGauge.claim_rewards` to claim other rewards).
- Angle uses newer version of LiquidityGauge (V4) contract that just need one function call (`LiquidityGauge.claim_rewards`) to fetch both veAsset and

other rewards.

- IDLE uses LiquidityGauge (V3) contract. veAsset (IDLE) is minted by calling `DistributorProxy.distribute` and gauge rewards are claimed by calling `LiquidityGauge.claim_rewards`.

Due to the discrepancies between different protocols in the reward claiming process, additional care must be taken to ensure that the flow of veAsset and gauge rewards are transferred to the appropriate contracts during integration. Otherwise, rewards will be stuck.

Lastly, I only see test cases written for claiming veAsset from the gauge. For completeness, it is recommended to also write test cases for claiming extra rewards from the gauge apart from veAsset.

solvetony (veToken Finance) confirmed and commented:

> Good catch, this issue is because Angle uses the same function for claim veAsset and extra rewards.

Alex the Entreprenerd (judge) commented:

> The warden has shown how Angle protocol will break certain invariants as the code assumes that claiming of `veAsset` to always be separate from claiming of `additionalRewards`.

> Due to this any additional reward emitted by the Angle Gauge will be stuck in the claiming contract.

> While impact is limited to loss of yield (loss of additional tokens), because the finding has broken the assumptions of the contract, meaning that Angle Protocol should not be integrated without a fix, I believe High Severity to be appropriate.

Alex the Entreprenerd (judge) commented:

> Upon further review, we may raise the concern of the contract being out of scope.

> However, given that:

- The sponsor Confirmed
- The vulnerability would be present in a normal configuration

> I believe the finding is of High Severity.

Medium Risk Findings (30)

[M-01] compromised `owner` can drain funds from `VeTokenMinter.sol`

Submitted by SmartSek, also found by ch13fd357r0y3r

Compromised `owner` can `withdraw()` entire balance of `VeTokenMinter.sol` to any other account.

Proof of Concept

https://github.com/code-423n4/2022-05-vetoken/blob/2d7cd1f6780a9bcc8387dea8fecfbd758462c152/contracts/VeTokenMinter.sol#L77-L81

```
function withdraw(address _destination, uint256 _amount) externa
    veToken.safeTransfer(_destination, _amount);

    emit Withdraw(_destination, _amount);
}
```

The `owner` can choose any `_destination` and `_amount` to send funds to with no delay or limit. These funds could be used to call `Booster.deposit()` and then `Booster.withdraw()` (withdraw) the equivalent in `lptoken`.

Recommended Mitigation Steps

Consider implementing a timelock on `VeTokenMinter.withdraw()` and changing the `destination` to an address that `owner` has no control over.

Example of similar issues illustrating the severity of the finding can be found here (H-09).

solvetony (veToken Finance) acknowledged and commented:

> Requires compromised owner.

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

> Finding is valid, but contingent on a Malicious or Compromised Admin, Medium Severity is more appropriate.

[M-02] VE3DRewardPool.sol is incompatible with Bal/veBal

Submitted by 0x52

`getReward` will become completely unusable if bal is added as an support asset.

Proof of Concept

veBal is not staked bal, it is staked 80-20 bal/eth LP. Rewards from gauges are paid in bal NOT 80-20 bal/eth LP. All rewards to this address will be received as bal. If the contract tries to deposit bal directly it will fail, causing getReward to always revert if bal is a supported asset (as all documentation conveys that it will be).

Recommended Mitigation Steps

Exclude veBal/Bal as a supported asset or create a special wrapper for Bal that adds the Bal as one sided liquidity then stakes the LP.

solvetony (veToken Finance) disputed and commented:

> Out of the scope, Balancer. But we need to consider this, once we start working on these.

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

> The finding shows how the code cannot support veBAL.

> In the specific case of veBAL, the deposit token is 80/20 BAL but the reward is BAL, meaning the function will eventually revert.

> This is conditional on the token being the BAL token, which means the finding cannot be of High Severity (conditional on configuration).

> The sponsor claims that Balancer will not be used, and that may be the case, however the [README for the contest]() does include BAL and for that reason I think the finding is Valid and of Medium Severity.

[M-03] No check for existing extraRewards during push

Submitted by cryptphi, also found by csanuragjain

https://github.com/code-423n4/2022-05-vetoken/blob/main/contracts/VE3DRewardPool.sol#L138

https://github.com/code-423n4/2022-05-vetoken/blob/main/contracts/VE3DLocker.sol#L156

Impact

Similar to a report I submitted for BaseRewardPool.sol (https://github.com/code-423n4/2022-05-vetoken/blob/main/contracts/BaseRewardPool.sol#L126)

When adding `extraRewards` to the extra reward pool in https://github.com/code-423n4/2022-05-vetoken/blob/main/contracts/VE3DRewardPool.sol#L138 , there's no check for already existing address.

Assume a particular address takes up 2 slots out of 3, and a user withdraws staked extra rewards, the user will receive double the amount requested in https://github.com/code-423n4/2022-05-vetoken/blob/main/contracts/VE3DRewardPool.sol#L257-L258

Proof of Concept

1. Assume `rewardManager` had mistakenly added the same address twice in `addExtraReward()`

2. A user calls `stake()` , linked rewards is staked twice to the same address (unexpected behaviour I guess but not severe issue)
3. Now, user calls `withdraw()` to withdraw linked rewards (this is already 2x in step 2)
4. User will receive double the linked rewards due to the iteration in `https://github.com/code-423n4/2022-05-vetoken/blob/main/contracts/VE3DRewardPool.sol#L257-L258`

Recommended Mitigation Steps

Guess a check for an already existing extraRewards can be added before Line 138

Similar issue

https://github.com/code-423n4/2022-05-vetoken/blob/main/contracts/VE3DLocker.sol#L156 - not so sure of the severity for this.

https://github.com/code-423n4/2022-05-vetoken/blob/main/contracts/BaseRewardPool.sol#L126 - reported in a seperate report

[jetbrain10 (veToken Finannce) confirmed](#)

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

> The warden has shown how, due to a misconfiguration error, a leak of value can happen, and other depositors (late withdrawers) would lose the rewards that they are entitled to.

> Mitigation seems to be straightforward (add a duplicate check, or use a enumerableMap), that said, because of the risk of loss contingent on configuration, I agree with Medium Severity.

[M-04] User can lose extra rewards

Submitted by csanuragjain

rewardManager can at anytime delete the extra Rewards. This impacts the extra rewards earned by existing staker. The existing staker will have no way to claim these extra rewards. Since these extra rewards have been staked for all users making a deposit BaseRewardPool.sol#L215 so basically the extra reward gets locked with no one having access to this fund. Need to be fixed for BaseRewardPool.sol as well

Proof of Concept

1. rewardManager adds 2 extra reward token A,B using addExtraReward
2. User X makes a deposit of amount 1000 using stake function
3. This stakes extra rewards A,B for this user
4. rewardManager now calls clearExtraRewards which removes extraRewards object
5. User X has now no way to retrieve the staked extra rewards A,B of amount 1000

Recommended Mitigation Steps

If rewardManager wants to clear extra rewards then all existing stakes on extra rewards must be withdrawn and claimed so that user extra rewards are not lost.

solvetony (veToken Finance) disagreed with severity and commented:

> Not an issue of the user staked fund, but we need to add call at pool factory for `clearExtraRewards()`.

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

> The warden has shown how, due to admin privilege, extra rewards could stay stuck in the contract.

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

> Notice that the rewards would be lost forever as there doesn't seem to be any `sweep` function which instead would allow the admin to take the reward tokens.

> My recommendation is to remove the function to `clearExtraRewards` as it doesn't seem to help but it can indeed cause issues.

[M-05] Duplicate LP token could lead to incorrect deposits

Submitted by csanuragjain, also found by kirk-baird and unforgiven

It was observed that addPool function is not checking for duplicate lpToken which allows 2 or more pools to have exact same lpToken. This can cause issue with deposits.

In case of duplicate lpToken, the first pool calling depositAll will take away all lpToken and deposit them under there own pid. This leaves no balance for 2nd pool.

Proof of Concept

1. PoolManager call addPool function and uses lpToken as A
2. PoolManager again call addPool function and mistakenly provides lpToken as A
3. Now 2 pools will be created with lpToken as A
4. depositAll function is called passing first pool.
5. This takes all balance of lpToken A and depsoit it under first pool pid
6. This mean no balance is left for second pool now

Recommended Mitigation Steps

Add a global variable keeping track of all lpToken added for pool. In case of duplicate lpToken addPool function should fail.

jetbrain10 (veToken Finance) confirmed and commented:

> There is already a validation not allow to add duplicated gauges, pool manager contract, add pool function , but we have to add also a validation for lp token like gauges.

Alex the Entreprenerd (judge) commented:

> The warden has shown how, due to a lack of validation, the assumption that each lpToken is used only on one pool can be broken. This will cause accounting issues.

> For those reasons, I agree with Medium Severity.

[M-06] Incorrectly set `_maxTime` to be in line with the locking `maxTime` of each veToken could render the `deposit` of this contract to be unfunctional or even freeze assets inside the contract

Submitted by Kumpa, also found by SecureZeroX and unforgiven

Since _maxTime needs to be manually input in the constructor with no other ways of changing it, if the owner inputs the _maxTime that is higher than the capacity of each vaults of veTokens, it will cause `IVoteEscrow(escrow).increase_unlock_time(_value);` to get rejected.

In the mild case when a user initially locks the asset during depositing, the contract will simply revert the transaction.

In the severe case when a user does not lock the asset during depositing, the asset will end up locked in the contract since noone will be able to succesfully call `lockVeAsset`. This will cause the asset to be locked up with no way of withdrawing it. Even though a user will still get benefits from the minted reward, a contract will not be able to receive any benefits since it can't utilize the locked asset in `VeAssetDepositor`.

Proof of Concept

```
constructor(
    address _staker,
    address _minter,
    address _veAsset,
    address _escrow,
    uint256 _maxTime
) {
    staker = _staker;
    minter = _minter;
    veAsset = _veAsset;
    escrow = _escrow;
    feeManager = msg.sender;
    maxTime = _maxTime;
}
```

1.The owner initially sets _maxTime in constructor to be 126489600 (86400*3664*) instead of 126144000 (86400*3654*) which is the maximum time that a user can deposit in curve

```
function deposit(
    uint256 _amount,
    bool _lock,
    address _stakeAddress
) public {
    require(_amount > 0, "!>0");

    if (_lock) {
        //lock immediately, transfer directly to staker to skip an erc20 transfer
        IERC20(veAsset).safeTransferFrom(msg.sender, staker, _amount);
        _lockVeAsset();
        if (incentiveVeAsset > 0) {
            //add the incentive tokens here so they can be staked together
            _amount = _amount.add(incentiveVeAsset);
            incentiveVeAsset = 0;
        }
    } else {
        //move tokens here
        IERC20(veAsset).safeTransferFrom(msg.sender, address(this), _amount);
        //defer lock cost to another user
        uint256 callIncentive = _amount.mul(lockIncentive).div(FEE_DENOMINATOR);
        _amount = _amount.sub(callIncentive);

        //add to a pool for lock caller
        incentiveVeAsset = incentiveVeAsset.add(callIncentive);
    }
```

2.A user `deposit` veAsset but deferring the locking to save gas

3.The veAsset is transferred from a user to the contract and the contract mint a reward token to the user

```
function _lockVeAsset() internal {
    uint256 veAssetBalance = IERC20(veAsset).balanceOf(address(this));
    if (veAssetBalance > 0) {
        IERC20(veAsset).safeTransfer(staker, veAssetBalance);
    }

    //increase ammount
    uint256 veAssetBalanceStaker = IERC20(veAsset).balanceOf(staker);
    if (veAssetBalanceStaker == 0) {
        return;
    }

    //increase amount
    IStaker(staker).increaseAmount(veAssetBalanceStaker);

    uint256 unlockAt = block.timestamp + maxTime;
    uint256 unlockInWeeks = (unlockAt / WEEK) * WEEK;

    //increase time too if over 2 week buffer
    if (unlockInWeeks.sub(unlockTime) > 2) {
        IStaker(staker).increaseTime(unlockAt);
        unlockTime = unlockInWeeks;
    }
    emit LockUpdated(veAssetBalanceStaker, unlockTime);
}
```

```
function increaseTime(uint256 _value1) external returns (bool) {
    require(msg.sender == depositor, "!auth");
    IVoteEscrow(escrow).increase_unlock_time(_value1);
    return true;
}
```

```
def increase_unlock_time(_unlock_time: uint256):
    """
    @notice Extend the unlock time for `msg.sender` to `_unlock_time`
    @param _unlock_time New epoch time for unlocking
    """
    self.assert_not_contract(msg.sender)
    _locked: LockedBalance = self.locked[msg.sender]
    unlock_time: uint256 = (_unlock_time / WEEK) * WEEK  # Locktime is rounded down to weeks

    assert _locked.end > block.timestamp, "Lock expired"
    assert _locked.amount > 0, "Nothing is locked"
    assert unlock_time > _locked.end, "Can only increase lock duration"
    assert unlock_time <= block.timestamp + MAXTIME, "Voting lock can be 4 years max"

    self._deposit_for(msg.sender, 0, unlock_time, _locked, INCREASE_UNLOCK_TIME)
```

*Above shows that the deposit will get reverted if lockAmount is more than 4 years in curve's VotingEscrow

4.The veAsset will not be able to move to the staking contract because when VoterProxy calls `increase_unlock_time` in the targeted vault, the call will get reverted due to exceeded _value of `unlockAt`

Recommended Mitigation Steps

The owner should set the value of _maxTime in advance for each veAsset and not relying on manual inputting during the constructoring as the risk of misconfiging ot is high. Otherwise the contract should add an emergency measure that can help change _maxTime but this function needs to be protected with the highest security (eg. with timelock and multisig).

solvetony (veToken Finance) disagreed with severity and commented:

> We can set it by a function instead of a constructor. Middle risk.

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

> The warden has shown how, due to misconfiguration a lock may not be created, causing tokens to be stuck in the contract.

> We can confirm that a revert would happen by checking `VotingEscrow`

> Because this is contingent on a wrong configuration I believe Medium Severity to be more appropriate.

[M-07] `VE3DRewardPool` and `VE3DLocker` adds to an unbounded array which may potentially lock all rewards in the contract

Submitted by kirk-baird, also found by csanuragjain, Dravee, gzeon, IllIllI, Koustre, Ruhum, unforgiven, VAD37, and xiaoming90

https://github.com/code-423n4/2022-05-vetoken/blob/2d7cd1f6780a9bcc8387dea8fecfbd758462c152/contracts/VE3DRewardPool.sol#L102-L112

https://github.com/code-423n4/2022-05-vetoken/blob/2d7cd1f6780a9bcc8387dea8fecfbd758462c152/contracts/VE3DLocker.sol#L145-L172

Impact

The function `addReward()` allows the owner to add a new reward token to the list `rewardTokens`.

However, this is an unbounded list that when appended to cannot be shortened. The impact is it is possible to reach a state where the list is so long it cannot be iterated through due to the gas cost being larger than the block gas limit. This would cause a state where all transactions which iterate over this list will revert.

Since the modifier `updateReward()` iterates over this list it is possible that there will reach a state where the we are unable to call any functions with this modifier. The list includes

- `stake()`
- `stakeAll()`
- `stakeFor()`
- `withdraw()`
- `withdrawAll()`
- `getReward()`
- `notifyRewardAmount()`

As a result it would therefore be impossible to withdraw any rewards from this contract.

The same issue exists in `VE3DLocker`. Where rewards can be added by either `Booster` or the owner.

```
function addReward(
    address _rewardToken,
    address _veAssetDeposits,
    address _ve3TokenRewards,
    address _ve3Token
) external onlyOwner {
    rewardTokenInfo[_rewardToken].veAssetDeposits = _veAsset
    rewardTokenInfo[_rewardToken].ve3TokenRewards = _ve3Toke
    rewardTokenInfo[_rewardToken].ve3Token = _ve3Token;
    rewardTokens.add(_rewardToken);
}



function addReward(
    address _rewardsToken,
    address _veAssetDeposits,
    address _ve3Token,
    address _ve3TokenStaking,
    address _distributor,
    bool _isVeAsset
) external {
    require(_msgSender() == owner() || operators.contains(_m
    require(rewardData[_rewardsToken].lastUpdateTime == 0);
    require(_rewardsToken != address(stakingToken));
    rewardTokens.push(_rewardsToken);


    rewardData[_rewardsToken].lastUpdateTime = uint40(block.
    rewardData[_rewardsToken].periodFinish = uint40(block.ti
    rewardDistributors[_rewardsToken][_distributor] = true;


    rewardData[_rewardsToken].isVeAsset = _isVeAsset;
    // if reward is veAsset
    if (_isVeAsset) {
        require(_ve3Token != address(0));
        require(_ve3TokenStaking != address(0));
        require(_veAssetDeposits != address(0));
        rewardData[_rewardsToken].ve3Token = _ve3Token;
        rewardData[_rewardsToken].ve3TokenStaking = _ve3Toke
        rewardData[_rewardsToken].veAssetDeposits = _veAsset
    }
```

```
    }
```

Recommended Mitigation Steps

Consider having some method for removing old reward tokens which are no longer in use.

Alternatively set a hard limit on the number of reward tokens that can be added.

A different option is to allow rewards to be iterated and distributed on a per token bases rather than all tokens at once.

jetbrain10 (veToken Finance) disagreed with severity and commented:

> we're going to add a bound, same as **#222**, **#125**.

Alex the Entreprenerd (judge) commented:

> My guesstimate of the math is that each reward would add 100k gas to the `updateReward` modifier, meaning we'd need 120 reward tokens before any consideration about running out of gas would happen.

> You also don't seem to be able to add a second one (provided someone has used the contract at least once after an addition).

> I'll think about it but am thinking Medium is stretching it.

Alex the Entreprenerd (judge) commented:

> Likelyhood is very low, however per the rules if enough rewards are added then claiming and withdrawing can be bricked permanently.

> I'd recommend end users to ensure the unlikely number of 120 rewards is never reached.

> Marking the finding as Valid and of Medium Severity.

[M-08] Not updating `totalWeight` when operator is removed in `VeTokenMinter`

Submitted by sseefried, also found by shenwilly

https://github.com/code-423n4/2022-05-vetoken/blob/2d7cd1f6780a9bcc8387dea8fecfbd758462c152/contracts/VeTokenMinter.sol#L36-L38

https://github.com/code-423n4/2022-05-vetoken/blob/2d7cd1f6780a9bcc8387dea8fecfbd758462c152/contracts/VeTokenMinter.sol#L41-L4

https://github.com/code-423n4/2022-05-vetoken/blob/2d7cd1f6780a9bcc8387dea8fecfbd758462c152/contracts/Booster.sol#L598-L614

Impact

The `totalWeight` state variable of the `VeTokenMinter` contract is used to work out the amount of `veAsset` earned when the `Booster.rewardClaimed` function is called.

However, while `totalWeight` is modified inside the `VeTokenMinter` contract when function `updateveAssetWeight` is called, the `totalWeight` is not similarly reduced when function `removeOperator` is called.

The impact is that remaining operators do not receive a fair share of the total rewards and a portion of the rewards are not given out at all.

Proof of Concept

- Operator 1 is added with weight 9
- Operator 2 is added with weight 1

The `totalWeight` is now 10.

This means that Operator 1 receives 90% of the amount while Operator 2 receives 10%.

If we then call `removeOperator` on Operator 1 then 90% of the reward is no longer minted and distributed. This is unfair to the remaining operators.

The can be seen on lines [607 - 608]() of the `Booster` contract. Function `rewardClaimed` will never be called for (removed) Operator 1. But for Operator 2 they will still receive 10% of the rewards even though Operator 1 is no longer registered in the system.

Recommended Mitigation Steps

The `totalWeight` should be reduced so that the remaining operators receive a fair share of the total rewards.

Using just method calls from `VeTokenMinter` one could rectify this situation by

- adding the removed operator with `addOperator`
- setting the weight to `0` using `updateveAssetWeight`. This will have the effect of reducing the `totalWeight` by the right amount.
- removing the operator again using `removeOperator`

However, the `removeOperator` function should just be rewritten to be as follows:

```
function removeOperator(address _operator) public onlyOwner {
    totalWeight -= veAssetWeights[_operator];
    veAssetWeights[_operator] = 0;
    operators.remove(_operator);
}
```

You might also want to modify `addOperator` so that a weight can be provided as an extra argument. This saves having to call `addOperator` and then `updateveAssetWeight` which could save on gas.

[solvetony (veToken Finance) disagreed with severity and commented]():

> Confirmed. But this should be a middle risk.

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

> The warden has shown how, due to a privileged call, removing an operator, the weight used to distribute rewards will not be updated fairly. This will cause an improper distribution of rewards.

> Because the finding is limited to Loss of Yield, due to Admin Configuration, I believe Medium Severity to be more appropriate.

[M-09] in notifyRewardAmount() of VE3DRewardPool and BaseRewardPool some tokes will be locked and not distributed becasue of rounding error

Submitted by unforgiven

https://github.com/code-423n4/2022-05-vetoken/blob/2d7cd1f6780a9bcc8387dea8fecfbd758462c152/contracts/BaseRewardPool.sol#L327-L345

https://github.com/code-423n4/2022-05-vetoken/blob/2d7cd1f6780a9bcc8387dea8fecfbd758462c152/contracts/VE3DRewardPool.sol#L367-L384

Impact

Function `notifyRewardAmount()` calculates `rewardRate` for reward token/s in `VE3DRewardPool` and `BaseRewardPool`. to calculate `rewardRate` it divides `reward` amount to `duration` but because of the rounding error in division, some of `reward` amount wouldn't get distributed and stuck in contract (`rewardRate * duration < reward`). and contract don't redistributes them or don't have any mechanism to recover them. This bug can be more damaging if the precision of `rewardToken` is low or token price is high.

Proof of Concept

This is `notifyRewardAmount()` code in `BaseRewardPool` : (contract `VE3DRewardPool` is similar)

```
function notifyRewardAmount(uint256 reward) internal updateF
    historicalRewards = historicalRewards.add(reward);
    if (block.timestamp >= periodFinish) {
        rewardRate = reward.div(duration);
    } else {
        uint256 remaining = periodFinish.sub(block.timestamp
        uint256 leftover = remaining.mul(rewardRate);
        reward = reward.add(leftover);
        rewardRate = reward.div(duration);
    }
    currentRewards = reward;
    lastUpdateTime = block.timestamp;
    periodFinish = block.timestamp.add(duration);
    emit RewardAdded(reward);
}
```

As you can see it sets `rewardRate = reward.div(duration);` and this is where the rounding error happens. and even if contract distributes all in all the duration it will distribute `rewardRate * duration` which can be lower than `reward` and the extra reward amount will stuck in contract. This is `queueNewRewards()` code which calls `notifyRewardAmount()` :

```
function queueNewRewards(uint256 _rewards) external returns
    require(msg.sender == operator, "!authorized");

    _rewards = _rewards.add(queuedRewards);

    if (block.timestamp >= periodFinish) {
        notifyRewardAmount(_rewards);
        queuedRewards = 0;
        return true;
    }

    //et = now - (finish-duration)
    uint256 elapsedTime = block.timestamp.sub(periodFinish.s
    //current at now: rewardRate * elapsedTime
    uint256 currentAtNow = rewardRate * elapsedTime;
    uint256 queuedRatio = currentAtNow.mul(1000).div(_rewarc
```

```
        //uint256 queuedRatio = currentRewards.mul(1000).div(_re
        if (queuedRatio < newRewardRatio) {
            notifyRewardAmount(_rewards);
            queuedRewards = 0;
        } else {
            queuedRewards = _rewards;
        }
        return true;
    }
```

As you can see it queues `rewardToken` and when the reward amount reach some point it calls `notifyRewardAmount()` and set `queuedRewards` to `0x0`. `notifyRewardAmount()` will set `rewardRate` based on `reward amount` but because of the rounding error some of the reward token `0 =< unused < duration` will stuck in contract and pool will not distribute it. if the token has low precision or has higher price then this amount value can be very big because `notifyRewardAmount()` can be called multiple times.

Tools Used

VIM

Recommended Mitigation Steps

add extra amount to `queuedRewards` so it would be distributed on next `notifyRewardAmount()` or add other mechanism to recover it.

solvetony (veToken Finance) disagreed with severity and commented:

> Will be fixed by two solutions, adding precision and by adding another function. Middle risk.

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

> The warden has shown how, due to rounding errors, `rewardRate` may round down and cause a certain amount of tokens not to be distributed.

> In lack of a way for the `operator` to re-queue the undistributed rewards, those tokens will be lost.

> Because we know `duration` is `604800` we can see that the `rewardRate` max loss is `604799`.

> Meaning the potential max dust due to rounding down can be upwards of a number with 12 decimals
>
> 604 799 * 6 047 800 =
> 3.6577034e+12

> Because the finding is limited to loss of Yield, I believe Medium Severity to be more appropriate.

[M-10] Unable To Get Rewards If Admin Withdraws $VE3D tokens From `VeTokenMinter` Contract

Submitted by xiaoming90, also found by 0x1f8b, and VAD37

It was observed that users will not be able to get their rewards from the reward contract at certain point of time if admin withdraws $VE3D token from the `VeTokenMinter` contract.

Proof of Concept

Based on the deployment script, it was understood that at the start of the project deployment, 30 million $VE3D tokens will be pre-minted for the `VeTokenMinter` contract. Thus, the `veToken.balanceOf(VeTokenMinter.address)` will be 30 million $VE3D tokens after the deployment.

https://github.com/code-423n4/2022-05-vetoken/blob/2d7cd1f6780a9bcc8387dea8fecfbd758462c152/migrations/2_deploy_basic_contracts.js#L18

```
// vetoken minter
await deployer.deploy(VeTokenMinter, veTokenAddress);
```

```
let vetokenMinter = await VeTokenMinter.deployed();
addContract("system", "vetokenMinter", vetokenMinter.address);
global.created = true;
//mint vetoke to minter contract
const vetoken = await VeToken.at(veTokenAddress);
await vetoken.mint(vetokenMinter.address, web3.utils.toWei("300(
addContract("system", "vetoken", veTokenAddress);
```

In the `VeTokenMinter` contract, there is a function called `VeTokenMinter.withdraw` that allows the admin to withdraw $VE3D tokens from the contract. Noted that this withdraw function only perform the transfer, but did not update any of the state variables (e.g. totalSupply, maxSupply) in the contract.

https://github.com/code-423n4/2022-05-vetoken/blob/2d7cd1f6780a9bcc8387dea8fecfbd758462c152/contracts/VeTokenMinter.sol#L77

```
function withdraw(address _destination, uint256 _amount) externa
    veToken.safeTransfer(_destination, _amount);

    emit Withdraw(_destination, _amount);
}
```

Assuming that an admin withdrawed 29 million $VE3D tokens from the `VoteProxy` with the appropriate approval from the DAO or community for some valid purposes. The `veToken.balanceOf(VeTokenMinter.address)` will be 1 million $VE3D tokens after the withdrawal.

At this point, notice that `veToken.balanceOf(VeTokenMinter.address)` is 1 million, while the `VeTokenMinter.maxSupply` constant is 30 million. Therefore, there exists a discrepency between the actual amount of $VE3D tokens (1 million) stored in the contact versus the max supply (30 million).

This discrepency will cause an issue in the `VeTokenMinter.mint` function because the calculation of the amount of $VE3D tokens to be transferred is based on the fact that 30 million $VE3D tokens is always sitting in the `VeTokenMinter` contract, and thus there is always sufficient $VE3D tokens available in the `VeTokenMinter` contract to send to its users.

The `uint256 amtTillMax = maxSupply.sub(supply);` code shows that the calculation is based on `maxSupply` constant, which is 30 million.

Assume that `mint(0x001, 10 million)` is called, and the value of the state variables when stepping through this function are as follows:

- `maxSupply` constant = 30 million
- `veToken.balanceOf(VeTokenMinter.address)` = 1 million
- `supply` & `totalSupply` = 20 million
- `totalCliffs` = 1000
- `reductionPerCliff` = 30,000 (maxSupply / totalCliffs)
- `cliff` = 666 (supply/reductionPerCliff)
- `reduction` = 1000 - 666 = 334
- `_amount` = 10 million * (334/1000) = 3.340 million
- `amtTillMax` = 10 million (maxSupply - supply) (Over here the contract assume that it still has 10 million VE3D tokens more to reach the max supply)
- `(_amount > amtTillMax) = False` (since "3.340 million > 10 million" = false)
- `veToken.safeTransfer(0x001, 3.340 million)` (This will revert. Insufficent balance)

The `veToken.safeTransfer(0x001, 3.340 million` will fail and revert because `VeTokenMinter` contract does not hold sufficent amount of $VE3D tokens to transfer out. `veToken.balanceOf(VeTokenMinter.address)` = 1 million, while the contract was attempting to send out 3.340 million.

https://github.com/code-423n4/2022-05-vetoken/blob/2d7cd1f6780a9bcc8387dea8fecfbd758462c152/contracts/VeTokenMinter.sol#L48

```
function mint(address _to, uint256 _amount) external {
    require(operators.contains(_msgSender()), "not an operator")

    uint256 supply = totalSupply;
```

```
        //use current supply to gauge cliff
        //this will cause a bit of overflow into the next cliff rang
        //but should be within reasonable levels.
        //requires a max supply check though
        uint256 cliff = supply.div(reductionPerCliff);
        //mint if below total cliffs
        if (cliff < totalCliffs) {
            //for reduction% take inverse of current cliff
            uint256 reduction = totalCliffs.sub(cliff);
            //reduce
            _amount = _amount.mul(reduction).div(totalCliffs);

            //supply cap check
            uint256 amtTillMax = maxSupply.sub(supply);
            if (_amount > amtTillMax) {
                _amount = amtTillMax;
            }

            //mint
            veToken.safeTransfer(_to, _amount);
            totalSupply += _amount;
        }
    }
```

The failure/revert of `VeTokenMinter.mint` function will cascade up to `Booster.rewardClaimed`, and futher cascade up to `BaseRewardPool.getReward`. Thus, `BaseRewardPool.getReward` will stop working. As a result, the users will not be able to get any rewards from the reward contracts.

This issue will affect all projects (Curve, Pickle, Ribbon, Idle, Angle, Balancer) because `VeTokenMinter` contract is deployed once, and referenced by all the projects. Thus, the impact could be quite widespread if this occurs, and many users would be affected.

https://github.com/code-423n4/2022-05-vetoken/blob/2d7cd1f6780a9bcc8387dea8fecfbd758462c152/contracts/Booster.sol#L598

```
    function rewardClaimed(
        uint256 _pid,
```

```
        address _address,
        uint256 _amount
    ) external returns (bool) {
        address rewardContract = poolInfo[_pid].veAssetRewards;
        require(msg.sender == rewardContract || msg.sender == lockRe
        ITokenMinter veTokenMinter = ITokenMinter(minter);
        //calc the amount of veAssetEarned
        uint256 _veAssetEarned = _amount.mul(veTokenMinter.veAssetWe
            veTokenMinter.totalWeight()
        );
        //mint reward tokens
        ITokenMinter(minter).mint(_address, _veAssetEarned);

        return true;
    }
```

https://github.com/code-423n4/2022-05-vetoken/blob/2d7cd1f6780a9bcc8387dea8fecfbd758462c152/contracts/BaseRewardPool.sol#L267

```
    function getReward(address _account, bool _claimExtras)
        public
        updateReward(_account)
        returns (bool)
    {
        uint256 reward = earned(_account);
        if (reward > 0) {
            rewards[_account] = 0;
            rewardToken.safeTransfer(_account, reward);
            IDeposit(operator).rewardClaimed(pid, _account, reward);
            emit RewardPaid(_account, reward);
        }

        //also get rewards from linked rewards
        if (_claimExtras) {
            for (uint256 i = 0; i < extraRewards.length; i++) {
                IRewards(extraRewards[i]).getReward(_account);
            }
        }
        return true;
    }
```

Recommended Mitigation Steps

Remove the `VeTokenMinter.withdraw` function if possible. Otherwise, update the internal accounting of `VeTokenMinter` contract during withdrawal so that the actual balance of the $VE3D tokens is taken into consideration within the `VeTokenMinter.mint`, and the contract will not attempt to transfer more tokens than what it has.

On a side note, [Convex's Minter contract](), will mint the `CRX` gov tokens to the users on the fly. See [https://github.com/convex-eth/platform/blob/1f11027d429e454dacc4c959502687eaeffdb74a/contracts/contracts/Cvx.sol#L76](). Thus, there will not be a case where there is not sufficient `CRV` tokens in the contract to send to it users.

However, in VeToken Protocol, it attempts to transfer the portion of pre-minted $VE3D tokens (30 millions) to the users. See [https://github.com/code-423n4/2022-05-vetoken/blob/2d7cd1f6780a9bcc8387dea8fecfbd758462c152/contracts/VeTokenMinter.sol#L72](). Thus, it is possible that there is not enough $VE3D tokens to send to its users if the admin withdraw the pre-minted $VE3D tokens.

[solvetony (veToken Finance) confirmed and commented]():

> We might need to withdraw, so we need to fix it.

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

> I have to acknowledge that `mint` can fail if not enough token is present, however I must disagree with the finding.

> The warden is saying that `totalSupply` is going to be a big number, while the number is updated exclusively after minting happens.

> For this reason I think the report is mostly invalid. No math issues will happen due to minting to the contract and then withdrawing as the variables will not be set in unrealistic values.

> However, it is true that `mint`ing an amount greater than what the contract hold will brick the system. For that reason I'll consider the report as valid and Medium,

> however I believe the POC for the high severity finding to be incorrect.

[M-11] Misconfiguration of Fees Incentive Might Cause Tokens To Be Stuck In `Booster` Contract

Submitted by xiaoming90, also found by 0xNazgul, berndartmueller, cccz, FSchmoede, Funen, kirk-baird, Kumpa, and VAD37

https://github.com/code-423n4/2022-05-vetoken/blob/2d7cd1f6780a9bcc8387dea8fecfbd758462c152/contracts/Booster.sol#L193

https://github.com/code-423n4/2022-05-vetoken/blob/2d7cd1f6780a9bcc8387dea8fecfbd758462c152/contracts/Booster.sol#L576

Proof of Concept

The `Booster.setFeeInfo` function is responsible for setting the allocation of gauge fees between lockers and $VE3D stakers. `lockFeesIncentive` and `stakerLockFeesIncentive` should add up to `10000`, which is equivalent to `100%`.

However, there is no validation check to ensure that that `_lockFeesIncentive` and `_stakerLockFeesIncentive` add up to `10000`. Thus, it entirely depends on the developer to get these two values right.

As such, it is possible to set `lockFeesIncentive + takerLockFeesIncentive` to be less than `100%`. This might happen due to human error. For instance, a typo (forget a few zero) or newly joined developer might not be aware of the fee denomination and called `setFeeInfo(40, 60)` instead of `setFeeInfo(4000, 6000)`.

https://github.com/code-423n4/2022-05-vetoken/blob/2d7cd1f6780a9bcc8387dea8fecfbd758462c152/contracts/Booster.sol#L193

```
uint256 public constant FEE_DENOMINATOR = 10000;

// Set reward token and claim contract, get from Curve's registr
function setFeeInfo(uint256 _lockFeesIncentive, uint256 _stakerI
    require(msg.sender == feeManager, "!auth");

    lockFeesIncentive = _lockFeesIncentive;
    stakerLockFeesIncentive = _stakerLockFeesIncentive;
    ..SNIP..
}
```

Assume that `setFeeInfo(40, 60)` is called instead of of `setFeeInfo(4000, 6000)`, only `1%` of the fee collected will be transferred to the users and the remaining `99%` of the fee collected will be stuck in the `Booster` contract.

https://github.com/code-423n4/2022-05-vetoken/blob/2d7cd1f6780a9bcc8387dea8fecfbd758462c152/contracts/Booster.sol#L576

```
function earmarkFees() external returns (bool) {
    //claim fee rewards
    IStaker(staker).claimFees(feeDistro, feeToken);
    //send fee rewards to reward contract
    uint256 _balance = IERC20(feeToken).balanceOf(address(this))

    uint256 _lockFeesIncentive = _balance.mul(lockFeesIncentive)
    uint256 _stakerLockFeesIncentive = _balance.mul(stakerLockFe
        FEE_DENOMINATOR
    );
    if (_lockFeesIncentive > 0) {
        IERC20(feeToken).safeTransfer(lockFees, _lockFeesIncenti
        IRewards(lockFees).queueNewRewards(_lockFeesIncentive);
    }
    if (_stakerLockFeesIncentive > 0) {
        IERC20(feeToken).safeTransfer(stakerLockRewards, _staker
        IRewards(stakerLockRewards).queueNewRewards(feeToken, _s
    }
    return true;
}
```

Can we retrieve or "save" the tokens stuck in `Booster` contract?

Any veAsset (e.g. CRV, ANGLE) sitting on the `Booster` contract is claimable. However, in this case, the `feeToken` is likely not a veAsset, thus the remaining gauge fee will be stuck in the `Booster` contract perpetually. For instance, in Curve, the gauge fee is paid out in 3CRV, the LP token for the TriPool. (Source)

Impact

Users will lost their gauge fee if this happens.

Recommended Mitigation Steps

Implement validation check to ensure that `lockFeesIncentive` and `takerLockFeesIncentive` add up to 100% to eliminate any risk of misconfiguration.

```
uint256 public constant FEE_DENOMINATOR = 10000;

// Set reward token and claim contract, get from Curve's registr
function setFeeInfo(uint256 _lockFeesIncentive, uint256 _stakerI
    require(msg.sender == feeManager, "!auth");
    require(_lockFeesIncentive + _stakerLockFeesIncentive == FEE

    lockFeesIncentive = _lockFeesIncentive;
    stakerLockFeesIncentive = _stakerLockFeesIncentive;
    ..SNIP..
}
```

solvetony (veToken Finance) confirmed and commented:

> Will try to provide a fix based on recommendation.

Alex the Entreprenerd (judge) commented:

> The warden has shown that, due to a lack of checks, a misconfiguration can happen that will cause tokens to be stuck (although temporarily) in the `Booster`.

> I believe a simple check to ensure:

- Caller incentive is not unfairly high (e.g. 100%)
- Total sums up to 100%

> Would be a great way to give additional guarantees to the contract.

> Agree with Medium Severity.

[M-12] Malicious operator can rug pull

*Submitted by oyc*109_

https://github.com/code-423n4/2022-05-vetoken/blob/2d7cd1f6780a9bcc8387dea8fecfbd758462c152/contracts/VoterProxy.sol#L138-L143

https://github.com/code-423n4/2022-05-vetoken/blob/2d7cd1f6780a9bcc8387dea8fecfbd758462c152/contracts/VoterProxy.sol#L274-L285

Vulnerability Details

A compromised or malicious operator can withdraw all tokens by calling the function withdrawAll

```
/2022-05-vetoken/contracts/VoterProxy.sol
138:     function withdrawAll(address _token, address _gauge) ex
139:         require(msg.sender == operator, "!auth");
140:         uint256 amount = balanceOfPool(_gauge).add(IERC20(_
141:         withdraw(_token, _gauge, amount);
142:         return true;
143:     }
```

The operator can also call an arbitary address with any value

```
/2022-05-vetoken/contracts/VoterProxy.sol
274:     function execute(
275:         address _to,
276:         uint256 _value,
277:         bytes calldata _data
```

```
    ) external returns (bool, bytes memory) {
        require(msg.sender == operator, "!auth");

        (bool success, bytes memory result) = _to.call{valu
        require(success, "!success");

        return (success, result);
    }
```

[jetbrain10 (veToken Finance) disputed and commented](#):

> The operator will be the multi-sig wallet.

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

> The warden has shown a risk for depositors in that the `operator` can sweep all tokens out of the contract.

> While this comment is longer than the report, the finding is valid and of medium severity.

[M-13] Unused rewards(because of totalSupply()==0 for some period) will be locked forever in VE3DRewardPool and BaseRewardPool

Submitted by unforgiven, also found by csanuragjain

https://github.com/code-423n4/2022-05-vetoken/blob/2d7cd1f6780a9bcc8387dea8fecfbd758462c152/contracts/BaseRewardPool.sol#L152-L162

https://github.com/code-423n4/2022-05-vetoken/blob/2d7cd1f6780a9bcc8387dea8fecfbd758462c152/contracts/VE3DRewardPool.sol#L170-L183

Impact

The `VE3DRewardPool` and `BaseRewardPool` contract is supposed to distribute rewards to stackers, but if in some period, `totalSupply()` was equal to `0`, then for that time period, rewards will not added to `rewardPerTokenStored` and those period rewards would not distribute to any address and those rewards will stuck in contract forever.

Proof of Concept

This is `notifyRewardAmount()` code in `BaseRewardPool` contract:
(`VE3DRewardPool` code is similar)

```
function notifyRewardAmount(uint256 reward) internal updat
  historicalRewards = historicalRewards.add(reward);
  if (block.timestamp >= periodFinish) {
      rewardRate = reward.div(duration);
  } else {
      uint256 remaining = periodFinish.sub(block.timestamp
      uint256 leftover = remaining.mul(rewardRate);
      reward = reward.add(leftover);
      rewardRate = reward.div(duration);
  }
  currentRewards = reward;
  lastUpdateTime = block.timestamp;
  periodFinish = block.timestamp.add(duration);
  emit RewardAdded(reward);
}
```

As you can see, in the line `rewardRate = reward.div(duration);` the value of `rewardRate` has been set to the division of available `reward` to `duration`. so if we distribute `rewardRate` amount in every second between stackers, then all rewards will be used by contract. contract uses `updateReward()` modifier to update `rewardPerTokenStored` (this variable keeps track of distributed tokens) and this modifier uses `rewardPerToken()` to update `BaseRewardPool`:

```
modifier updateReward(address account) {
  rewardPerTokenStored = rewardPerToken();
  lastUpdateTime = lastTimeRewardApplicable();
  if (account != address(0)) {
      rewards[account] = earned(account);
```

```
            userRewardPerTokenPaid[account] = rewardPerTokenStor
        }
        emit RewardUpdated(account, rewards[account], rewardPerT
        _;
    }
```

This is `rewardPerToken()` code in `BaseRewardPool`:

```
    function rewardPerToken() public view returns (uint256) {
        if (totalSupply() == 0) {
            return rewardPerTokenStored;
        }
        return
            rewardPerTokenStored.add(
                lastTimeRewardApplicable().sub(lastUpdateTime).m
                    totalSupply()
                )
            );
    }
```

If for some period `totalSupply()` was `0` then contract won't increase `rewardPerTokenStored` and those periods reward stuck in contract forever, because there is no mechanism to calculate them and withdraw them in contract. For example if `operator` deploy and initialize the pool immediately before others having a chance of stacking their tokens, and use `queueNewRewards()` to queue the rewards then The rewards for early period of pool will be locked forever.

Tools Used

VIM

Recommended Mitigation Steps

Add some mechanism to recalculate `rewardRate` or calculated undistributed rewards(calculated undistributed reward based on `rewardRate` and when `totalSupply()` is `0`).

[solvetony (veToken Finance) disagreed with severity and commented](#):

> Recover function would solve this issue. Middle risk.

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

> The warden has found a valid problem, however a coded POC would have gone a long way.

> In order to judge the issue I had to code it for myself to be able to demonstrate the problem.

> Anyhow this is a Brownie dump of me setting up the contract (removing transferFrom to get it done rapidly) Showing how skipping 1/3 of reward duration will cause a loss of 1/3 of the yield.

> Meaning that the accumulator used for rewards is not redistributing the old rewards

```
>>> x.lastTimeRewardApplicable()
0
>>> x.queueNewRewards(1e18, {"from": a[0]})
Transaction sent: 0x0b959c886d959038396aa0a9a82cef39c6bc817e4e48
  Gas price: 0.0 gwei   Gas limit: 12000000   Nonce: 5
  BaseRewardPool.queueNewRewards confirmed   Block: 15208165   G

<Transaction '0x0b959c886d959038396aa0a9a82cef39c6bc817e4e48026(
>>> x.lastTimeRewardApplicable()
1658703966
>>> chain.time()
1658703975
>>> 1658703966 - 1658703975
-9
>>> x.lastTimeRewardApplicable()
1658703966
>>> x.lastUpdateTime()
1658703966
>>> x.periodFinish()
1659308766
>>> 1658703966 - 1659308766
-604800
>>> chain.sleep(604800 // 3) \#\# Sleep for third of time
>>> chain.time()
1658905644
```

```
>>> 1658905644- 1659308766
-403122
>>> x.stake(1e18, {"from": a[0]})
Transaction sent: 0x4899faa962b45d2f746db4f045b9858fdd506185ceca
  Gas price: 0.0 gwei   Gas limit: 12000000   Nonce: 6
  BaseRewardPool.stake confirmed   Block: 15208166   Gas used: 7

<Transaction '0x4899faa962b45d2f746db4f045b9858fdd506185cecab019
>>> x.balanceOf(a[0])
1000000000000000000
>>> x.earned(a[0])
0
>>> chain.sleep(x.duration())
>>> x.earned(a[0])
0
>>> x.getReward(a[0], False)
Transaction sent: 0xba12fac5aa76e59d51fa277245cd96db53d7ca2685bc
  Gas price: 0.0 gwei   Gas limit: 12000000   Nonce: 7
  BaseRewardPool.getReward confirmed   Block: 15208167   Gas use

<Transaction '0xba12fac5aa76e59d51fa277245cd96db53d7ca2685bdf97a
>>> history[-1].return_value
666502976190414339
>>>
```

> Which means, that the warden has found a valid vulnerability and any time spent with a totalSupply of 0 will cause those rewards to be lost.

> Because this is contingent on:

- No deposits before adding rewards
- Lasts only until a deposit has happened
- Is related to loss of yield

> I believe Medium Severity to be more appropriate.

[M-14] Deposited staking tokens can be lost if rewards token info added by mistake in addReward() in VE3DRewardPool and there is no checking to ensure this would not happen (ve3Token for one reward was equal to stacking token)

Submitted by unforgiven

https://github.com/code-423n4/2022-05-vetoken/blob/2d7cd1f6780a9bcc8387dea8fecfbd758462c152/contracts/VE3DRewardPool.sol#L102-L112

https://github.com/code-423n4/2022-05-vetoken/blob/2d7cd1f6780a9bcc8387dea8fecfbd758462c152/contracts/VE3DRewardPool.sol#L297-L318

Impact

If `owner` by mistake or bad intention add a reward token that `ve3Token` of `new reward` token was equal to `old reward` token address then `old reward` token balance of `VE3DRewardPool` can be lost by calling `getReward(account, _claimExtras=False,_stake=False)` because of the line `314` which is `IERC20(rewardTokenInfo[_rewardToken].ve3Token).safeTransfer(_account, ve3TokenBalance);`

Proof of Concept

This is `addReward()` code in `VE3DRewardPool`:

```
function addReward(
    address _rewardToken,
    address _veAssetDeposits,
    address _ve3TokenRewards,
    address _ve3Token
) external onlyOwner {
    rewardTokenInfo[_rewardToken].veAssetDeposits = _veAsset
    rewardTokenInfo[_rewardToken].ve3TokenRewards = _ve3Toke
    rewardTokenInfo[_rewardToken].ve3Token = _ve3Token;
    rewardTokens.add(_rewardToken);
}
```

As you can see there is no check for values and it sets new `rewardToken` parameters. This is `getReward()` code:

```
function getReward(
    address _account,
    bool _claimExtras,
    bool _stake
) public updateReward(_account) {
    address _rewardToken;
    for (uint256 i = 0; i < rewardTokens.length(); i++) {
        _rewardToken = rewardTokens.at(i);

        uint256 reward = earnedReward(_rewardToken, _account
        if (reward > 0) {
            rewardTokenInfo[_rewardToken].rewards[_account]
            IERC20(_rewardToken).safeApprove(rewardTokenInfo
            IERC20(_rewardToken).safeApprove(
                rewardTokenInfo[_rewardToken].veAssetDeposit
                reward
            );
            IVeAssetDeposit(rewardTokenInfo[_rewardToken].ve
                reward,
                false
            );

            uint256 ve3TokenBalance = IERC20(rewardTokenInfo
                address(this)
            );
            if (_stake) {
                IERC20(rewardTokenInfo[_rewardToken].ve3Toke
                    rewardTokenInfo[_rewardToken].ve3TokenRe
                    0
                );
                IERC20(rewardTokenInfo[_rewardToken].ve3Toke
                    rewardTokenInfo[_rewardToken].ve3TokenRe
                    ve3TokenBalance
                );
                IRewards(rewardTokenInfo[_rewardToken].ve3To
                    _account,
                    ve3TokenBalance
                );
            } else {
                IERC20(rewardTokenInfo[_rewardToken].ve3Toke
                    _account,
                    ve3TokenBalance
                );
            }
            emit RewardPaid(_account, ve3TokenBalance);
```

```
            }
        }

        //also get rewards from linked rewards
        if (_claimExtras) {
            uint256 length = extraRewards.length;
            for (uint256 i = 0; i < length; i++) {
                IRewards(extraRewards[i]).getReward(_account);
            }
        }
    }
```

As you can see it loops through all `rewardTokens` and in line `314` it transfers all balance of `rewardTokenInfo[_rewardToken].ve3Token` of contract to `account` address. So if `owner` by mistake or bad intention add a new `rewardToken` that `rewardTokenInfo[newRewardToken].ve3Token == oldRewardToken` then by calling `getReward(account, False, False)` contract will loop through all reward tokens and when it reaches the `newRewardToken` it will transfer all the balance of contract in `rewardTokenInfo[newRewardToken].ve3Token` address to account address and `rewardTokenInfo[newRewardToken].ve3Token` is `oldRewardToken`, so it will transfer all the `oldRewardToken` balance of contract(which was supposed to be distributed among all stackers) to account address. This is like a backdoor that gives `owner` the ability to withdraw rewards tokens and if owner makes a simple mistake then one can easily withdraw that contract balance of reward token.

Tools Used

VIM

Recommended Mitigation Steps

In `addReward()` check that there is no mistake or any malicious values added to rewards.

solvetony (veToken Finance) disagreed with severity and commented:

> We would try to decrease chance of that, most likely by implementing validation, to reduce manual check for owner.

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

> The warden has shown how due to misconfiguration all rewards could be sent to the first claimer, because this is contingent on a misconfiguration, I believe Medium Severity to be more appropriate.

Alex the Entreprenerd (judge) commented:

> While this report and #179 can be grouped as similar, I believe the warden has shown two different potential risks and at this time am choosing to leave the two findings as separate.

[M-15] Owner should be allowed to change feeManager

Submitted by csanuragjain

Once Fee Manager has been set initially by owner, then owner has no power to change it. Owner should be allowed to change fees manager in case if he feels current fee manager is behaving maliciously

Proof of Concept

1. Observe the setFeeManager function and see that only feeManager is allowed to change it once set initially

```
function setFeeManager(address _feeM) external {
        require(msg.sender == feeManager, "!auth");
        feeManager = _feeM;
        emit FeeManagerUpdated(_feeM);
    }
```

Recommended Mitigation Steps

Change the setFeeManager function like below. Same can be done with other important functionality involving setArbitrator and setVoteDelegate

```
require(msg.sender == owner, "!auth");
```

jetbrain10 (veToken Finance) confirmed, but disagreed with severity

Alex the Entreprenerd (judge) commented:

> I agree with the finding, the feeManager can also potentially funnel funds away so it's best to allow governance to replace them if need be.

> Due to the finding being tied to admin privilege, I agree with Medium Severity.

[M-16] Admin Privilege in minting to arbitrary address allows operator to dilute tokens

Submitted by Alex the Entreprenerd, also found by lllllll

`veTokenMinter` allows any operator to mint new tokens, that's fine in the context of it being used for:

- having `VeAssetDepositor` deposit for the user and mint
- Having the `Booster` mint reward tokens

However because of the open ended system that allows any address to be set as `operator`, the system allows the admin to set themselves as the operator and to mint an excess amount of tokens, diluting other users.

Because this seems to be used exclusively by the `VeAssetDepositor` and the `Booster` hardcoding these two addresses would provide stronger security guarantees.

Proof of Concept

addOperator(malicious, {"from": gov})

mint(malicious, AMOUNT, {"from": malicious})

Recommended Mitigation Steps

Set the minters as immutable to provide stronger security guarantees.

jetbrain10 (veToken Finance) acknowledged, but disagreed with severity and commented:

> Admin will be controlled by DAO to prevent this happen.

Alex the Entreprenerd (judge) commented:

> Because am judging the contest am forfeiting any winnings.

> I do believe that the system would be best if the minters where hardcoded (it would also save gas).

[M-17] Missing sane bounds on asset weights

Submitted by lllllll

The admin may fat-finger a change, or be malicious, and have the weights be extreme - ranging from zero to `type(uint256).max`, which would cause the booster to pay out unexpected amounts

Proof of Concept

No bounds checks in the update function:

```
File: contracts/VeTokenMinter.sol   \#1

41      function updateveAssetWeight(address veAssetOperator, u
42          require(operators.contains(veAssetOperator), "not a
43          totalWeight -= veAssetWeights[veAssetOperator];
44          veAssetWeights[veAssetOperator] = newWeight;
45          totalWeight += newWeight;
46      }
```

https://github.com/code-423n4/2022-05-vetoken/blob/2d7cd1f6780a9bcc8387dea8fecfbd758462c152/contracts/VeTokenMinter.sol#L41-L46

The value is used by the reward contract to determine how much to mint:

```
File: contracts/Booster.sol   \#2

598       function rewardClaimed(
599           uint256 _pid,
600           address _address,
601           uint256 _amount
602       ) external returns (bool) {
603           address rewardContract = poolInfo[_pid].veAssetRew
604           require(msg.sender == rewardContract || msg.sender
605           ITokenMinter veTokenMinter = ITokenMinter(minter);
606           //calc the amount of veAssetEarned
607           uint256 _veAssetEarned = _amount.mul(veTokenMinter
608               veTokenMinter.totalWeight()
609           );
610           //mint reward tokens
611           ITokenMinter(minter).mint(_address, _veAssetEarned
```

https://github.com/code-423n4/2022-05-vetoken/blob/2d7cd1f6780a9bcc8387dea8fecfbd758462c152/contracts/Booster.sol#L598-L611

Wrong values will lead to excessive inflation/deflation.

Recommended Mitigation Steps

Have sane upper/lower limits on the values.

solvetony (veToken Finance) confirmed, but disagreed with severity and commented:

> We may consider adding this.

Alex the Entreprenerd (judge) commented:

> The warden has shown how due to a lack of checks certain assets may provide a disproportionate amount of rewards.

> Because this is contingent on an admin mistake, and the impact would be loss or gain of Yield; I believe Medium Severity to be appropriate.

[M-18] Governance can arbitrarily burn VeToken from any address

Submitted by shenwilly

Governance can burn any amount of `VeToken` from any address.

Unlike `VE3Token` which is minted when users deposit veAsset and burned when users withdraw, the `burn` function in the governance token `VeToken.sol` is unnecessary and open up the risk of malicious/compromised governance burning user's token.

Recommended Mitigation Steps

Consider removing the function, or modify the burn function so it only allows `msg.sender` to burn the token:

```
function burn(uint256 _amount) external {
    _burn(msg.sender, _amount);
}
```

[solvetony (veToken Finance) acknowledged and commented](#):

> We might update readme on that case.

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

> The warden has shown how the operator, which may be the DAO or a privileged Multisig, can burn any tokens.

> While the functionality is part of the system for VE3Token as the system uses it to track underlying ownership, burning of balances from arbitrary addresses is a dangerous form of admin privilege.

> I'd recommend deleting the burn function.

[M-19] `VE3DRewardPool` claim in loop depend on pausable token

Submitted by VAD37

https://github.com/code-423n4/2022-05-vetoken/blob/2d7cd1f6780a9bcc8387dea8fecfbd758462c152/contracts/VE3DRewardPool.sol#L296-L299

https://github.com/code-423n4/2022-05-vetoken/blob/1be2f03670e407908f175c08cf8cc0ce96c55baf/contracts/VeAssetDepositor.sol#L134-L152

Vulnerability Details

Project veToken is supposed to be a generalized version of Convex for non-Curve token. There is only one contract for all rewards token in the platform.

All ve3Token rewards are bundled together inside `ve3DLocker` and `ve3DRewardPool` in a loop. Instead of having its own unique contract like `VeAssetDepositer` or `VoterProxy` for each token.

Impact

If one token has pausable transfer, user cannot claim rewards or withdraw if they have multiple rewards include that pause token.

Right now the project intends to support only 6 tokens, including Ribbon token which has pausable transfer controlled by Ribbon DAO.

Normally, this would not be an issue in Convex where only a few pools would be affected by single coin. Since, veAsset are bundled together into single reward pool, it becomes a major problem.

Proof of Concept

- Token like Ribbon pause token transfer by DAO due to an unfortunate event.
- `VE3DRewardPool` try call `getReward()` , `VeAssetDepositor` try deposit token from earned rewards does not work anymore because `IERC20.transfer` is

blocked. This effectively reverts current function if user have this token reward > 0.

Recommended Mitigation Steps

It would be a better practice if we had a second `getReward()` function that accepts an array of token that we would like to interact with.

It saves gas and only requires some extra work on frontend website. Instead of current implementation, withdraw all token bundles together.

solvetony (veToken Finance) confirmed and commented:

> It might happen rarely, but we might fix that.

Alex the Entreprenerd (judge) commented:

> The warden has shown how, in certain cases, because a reward token could be pausable, this can cause the entire claim process to break as `getReward` doesn't allow the caller to specify which tokens to receive.

> I have to agree that the odds are low, however the finding is valid and because it's reliant on external conditions I believe Medium Severity to be appropriate.

[M-20] Contracts should be robust to upgrades of underlying gauges and eventually changes of the underlying tokens

Submitted by Picodes

For some veAsset project (for example Angle's gauges, gauge contracts are upgradable, so interfaces and underlying LP tokens are subject to change, blocking and freezing the system. Note that this is not hypothetic as it happened a few weeks ago: see this snapshot vote. Therefore, the system should be robust to a change in the pair gauge / token.

Note that is doable in the current setup for the veToken team to rescue the funds in such case, hence it is only a medium issue. You'd have to do as follow: a painful shutdown of the `Booster` (which would lead to an horrible situation where you'd

have to preserve backwards compatibility for LPs to save their funds in the new Booster), an operator change in `VoterProxy` to be able to call `execute`.

Recommended Mitigation Steps

To deal with upgradeable contracts, either the `VoterProxy` needs to be upgradable to deal with any situation that may arise, either you need to add upgradeable "intermediate" contracts between the `staker` and the gauge that could be changed to preserve the logic.

jetbrain10 (veToken Finance) confirmed and commented:

> Same as answer **#49** for angle Voter Proxy, will make it upgrade able and make all future VoterProxy can be upgrade able as well if veAsset project agrees.

Alex the Entreprenerd (judge) commented:

> The warden has shown how, due to integrating with underlying upgradeable contracts, the Sponsors Contracts could get bricked or forced into a shutdown.

> I believe the finding to be equivalent to showing how the system could end up being attached to a bad gauge, and the warden has shown historical proof that this has happened and could happen again.

> For those reasons, as well as the Sponsor Confirming, I believe the finding to be valid and of Medium Severity.

[M-21] `VoterProxy` incorrectly assumes a 1-1 mapping between the gauge and the LP tokens.

Submitted by Picodes

https://github.com/code-423n4/2022-05-vetoken/blob/2d7cd1f6780a9bcc8387dea8fecfbd758462c152/contracts/VoterProxy.sol#L270

https://github.com/code-423n4/2022-05-vetoken/blob/2d7cd1f6780a9bcc8387dea8fecfbd758462c152/contracts/VoterProxy.sol#L140

Impact

When calling `withdrawAll`, to compute the amount to withdraw, the contract checks the balance of gauge tokens and assume that `1 gauge token = 1 LP token` by doing `uint256 amount = balanceOfPool(_gauge).add(IERC20(_token).balanceOf(address(this)));`. Overall this assumption may not hold and this would lead to a loss of funds when calling `withdrawAll`.

Proof of Concept

Indeed this is false in some cases, check for example https://etherscan.io/address/0x3785Ce82be62a342052b9E5431e9D3a839cfB581 where the total supply is not the same as the balance of LP tokens held by the contract. You can also check the contract code where you can see there is a `staking_factor` between the balance and the underlying LP token balance.

Recommended Mitigation Steps

Use the total supply of `pool.token` which is a better proxy to know how much to withdraw when withdrawing all.

jetbrain10 (veToken Finance) commented:

> Are you referring we need to calc the staking _factor by ourself?

Alex the Entreprenerd (judge) commented:

> @jetbrain10 the warden says that some Deposits will not return the same amount of Lp Tokens.

> See this example from the contract linked by the Warden: https://etherscan.io/tx/0xd3eab573697d4fb92ebe4d91d35b03795d384ac45f7a723b321c98f6da2420cb

> I think this means the contracts may break when integrating with Angle Protocol.

> As far as I'm aware CRV, Balancer will return a 1-1 between the Deposit Token and the Gauge Token.

Alex the Entreprenerd (judge) commented:

> The warden has shown evidence of the GaugeLP-Token being "rebased" from the "usual" 1-1 to a ratio (assuming due to cost / increasing in value in underlying).

> Because:

- The Sponsor system is meant to integrate with multiple protocols
- The warden has shown a specific example (ANGLE) of the math bring broken

> Considering that this is contingent on the sponsor launching an integration with the Angle Protocol, using Gauges that "rebase", I believe the finding to be Valid and of Medium Severity.

> Most likely remediation will require poking the gauge for exchange rates and also checking available tokens after withdrawing.

jetbrain10 (veToken Finance) acknowledged

[M-22] VE3DRewardPool allows the same reward address to be added multiple times to the `extraRewards` array

Submitted by Ruhum

https://github.com/code-423n4/2022-05-vetoken/blob/main/contracts/VE3DRewardPool.sol#L134-L139

https://github.com/code-423n4/2022-05-vetoken/blob/main/contracts/VE3DRewardPool.sol#L214-L216

https://github.com/code-423n4/2022-05-vetoken/blob/main/contracts/BaseRewardPool.sol#L121

Impact

When the same address is included twice it might cause issues depending on the contract. I checked the current convex contract to see which addresses were added to the `extraRewards` array. For [cvxCRV Rewards](#) there's only [0x7091dbb7fcbA54569eF1387Ac89Eb2a5C9F6d2EA](#) which is the VirtualBalanceRewardPool contract.

Proof of Concept

1. `rewardManager` adds the same address twice through `addExtraReward()`
2. VE3DRewardPool calls `stake()` twice with the same amount:

```
function stake(uint256 _amount) public updateReward(msg.senc
    require(_amount > 0, "RewardPool : Cannot stake 0");

    //also stake to linked rewards
    uint256 length = extraRewards.length;
    for (uint256 i = 0; i < length; i++) {
        IRewards(extraRewards[i]).stake(msg.sender, _amount)
    }

    //add supply
    _totalSupply = _totalSupply.add(_amount);
    //add to sender balance sheet
    _balances[msg.sender] = _balances[msg.sender].add(_amour
    //take tokens from sender
    stakingToken.safeTransferFrom(msg.sender, address(this),

    emit Staked(msg.sender, _amount);
}
```

Recommended Mitigation Steps

Prevent the same addresses from being added multiple times to the `extraRewards` array.

[jetbrain10 (vetoken Finance) disagreed with severity and commented](#):

> Will add code to check if this is same reward address.

Alex the Entreprenerd (judge) commented:

> The warden has shown how, due to a misconfiguration, rewards could be disbursed twice, breaking protocol invariants.

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

[M-23] BaseRewardPool's `rewardPerTokenStored` can be inflated and rewards can be stolen

Submitted by sorrynotsorry

https://github.com/code-423n4/2022-05-vetoken/blob/2d7cd1f6780a9bcc8387dea8fecfbd758462c152/contracts/BaseRewardPool.sol#L180

https://github.com/code-423n4/2022-05-vetoken/blob/2d7cd1f6780a9bcc8387dea8fecfbd758462c152/contracts/BaseRewardPool.sol#L137-L142

https://github.com/code-423n4/2022-05-vetoken/blob/2d7cd1f6780a9bcc8387dea8fecfbd758462c152/contracts/BaseRewardPool.sol#L157-L159

Impact

The first user who calls BaseRewardPool's `stake()` function with 1 wei can inflate the `rewardPerTokenStored`. And the same user can call `withdraw` and drain the rewards.

Proof of Concept

When a user call `stake()` with 1 wei, it updates the `_totalSupply` as 1 wei and the `rewards` through `updateReward` modifier. This modifier calls `rewardPerToken()` to assign the return to `rewardPerTokenStored` and assigns it to the account via

`userRewardPerTokenPaid[account] = rewardPerTokenStored;`

`rewardPerToken()` formula is as below;

```
function rewardPerToken() public view returns (uint256) {
        if (totalSupply() == 0) {
            return rewardPerTokenStored;
        }
        return
            rewardPerTokenStored.add(
                lastTimeRewardApplicable().sub(lastUpdateTime).m
                    totalSupply()
                )
            );
    }
```

Since it depends on the denominator as totalSupply(), the whole multiplying will be divided by 1 wei which will inflate the `rewardPerTokenStored` astronomically. And there is no obstacle for the user to withdraw it in the withdraw function.

https://github.com/code-423n4/2022-05-vetoken/blob/2d7cd1f6780a9bcc8387dea8fecfbd758462c152/contracts/BaseRewardPool.sol#L180

https://github.com/code-423n4/2022-05-vetoken/blob/2d7cd1f6780a9bcc8387dea8fecfbd758462c152/contracts/BaseRewardPool.sol#L137-L142

https://github.com/code-423n4/2022-05-vetoken/blob/2d7cd1f6780a9bcc8387dea8fecfbd758462c152/contracts/BaseRewardPool.sol#L157-L159

Recommended Mitigation Steps

The team might consider to add boundries to reward the stakers to be consistent inside the limits.

solvetony (veToken Finance) acknowledged, but disagreed with severity and commented:

> This is the rare case when only one staker in the pool.

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

> This report would have heavily benefited by a coded POC.

> I'm pasting my raw data from terminal (typos and stuff included)

Compare for unfairness

> Seems to be unfair toward the early depositor, but not too crazily

```
>>> x = BaseRewardPool.deploy(1, ETH_ADDRESS, ETH_ADDRESS, a[0],
Transaction sent: 0x981137ae6eead737b5a56a58f03046f184b9482c7b85
  Gas price: 0.0 gwei   Gas limit: 12000000   Nonce: 4
  BaseRewardPool.constructor confirmed   Block: 15221212   Gas u
  BaseRewardPool deployed at: 0xe0aA552A10d7EC8760Fc6c246D391E69

>>> x.stake(1, {"from": a[0]})
Transaction sent: 0x03971a1e78ab54979f834746436488f1cb49c5a48395
  Gas price: 0.0 gwei   Gas limit: 12000000   Nonce: 5
  BaseRewardPool.stake confirmed   Block: 15221213   Gas used: 7

<Transaction '0x03971a1e78ab54979f834746436488f1cb49c5a483957f0c
>>> x.queueNewRewards(1e18, {"from": a[0]})
Transaction sent: 0xb88fe369ed4b49f8043d67354f3429f41a403fa98b4e
  Gas price: 0.0 gwei   Gas limit: 12000000   Nonce: 6
  BaseRewardPool.queueNewRewards confirmed   Block: 15221214   G

<Transaction '0xb88fe369ed4b49f8043d67354f3429f41a403fa98b4e74a8
>>> chain.sleep(x.duration() // 10)
>>> chain.time
<bound method Chain.time of <Chain object (chainid=1, height=152
>>> chain.time()
1658940912
>>> chain.sleep(x.duration() // 10)
>>> chain.time()
1659001396
>>> x.stake(1, {"from": a[0]})
Transaction sent: 0xea095333668d2f89979e55bfd7252e6c117d875bfbb6
  Gas price: 0.0 gwei   Gas limit: 12000000   Nonce: 7
  BaseRewardPool.stake confirmed   Block: 15221215   Gas used: 7
```

```
<Transaction '0xea095333668d2f89979e55bfd7252e6c117d875bfbb628ee
>>> x.stake(10, {"from": a[1]})
Transaction sent: 0x85f2de5caee675c9c9889533804c65cb6b337f3dcb71
  Gas price: 0.0 gwei   Gas limit: 12000000   Nonce: 5
  BaseRewardPool.stake confirmed   Block: 15221216   Gas used: 1

<Transaction '0x85f2de5caee675c9c9889533804c65cb6b337f3dcb714af7
>>> chain.sleep(x.duration())
>>> first = x.getReward({"from": a[0]).return_value
  File "<console>", line 1,
    first = x.getReward({"from": a[0]).return_value
                                       ^
SyntaxError: closing parenthesis ')' does not match opening pare
>>> first = x.getReward({"from": a[0]).return_value
  File "<console>", line 1,
    first = x.getReward({"from": a[0]).return_value
                                       ^
SyntaxError: closing parenthesis ')' does not match opening pare
>>> first = x.getReward({"from": a[0]}).return_value
Transaction sent: 0x6cd0057d7a1c9586b6bb75324d1871fbce02b661c2bc
  Gas price: 0.0 gwei   Gas limit: 12000000   Nonce: 8
  BaseRewardPool.getReward confirmed   Block: 15221217   Gas use

>>> second = x.getReward({"from": a[1]}).return_value
Transaction sent: 0x2fc7122fe33c2b332d32c1bc634ef9ec5d94133fc411
  Gas price: 0.0 gwei   Gas limit: 12000000   Nonce: 6
  BaseRewardPool.getReward confirmed   Block: 15221218   Gas use

>>> first
True
>>> second
True
>>> x.userRewardPerTokenPaid(a[0])
26671530533507225058333333333333333
>>> x.userRewardPerTokenPaid(a[1])
26671530533507225058333333333333333

\#\# Second User (10 deposited)
>>> history[-1].events
{'RewardUpdated': [OrderedDict([('user', '0x33A4622B82D4c04a53e1
\#\# First user 2 deposited (early 1 /10th of time)
>>> history[-2].events
{'RewardUpdated': [OrderedDict([('user', '0x66aB6D9362d4F355962
>>>
```

Deposit, front-run rewards and withdraw

```
>>> x.stake(1, {"from": a[0]})
Transaction sent: 0x03971a1e78ab54979f834746436488f1cb49c5a4839
  Gas price: 0.0 gwei   Gas limit: 12000000   Nonce: 5
  BaseRewardPool.stake confirmed   Block: 15221254   Gas used: 7

<Transaction '0x03971a1e78ab54979f834746436488f1cb49c5a483957f0c
>>> x.queueNewRewards(1e18, {"from": a[0]})
Transaction sent: 0xb88fe369ed4b49f8043d67354f3429f41a403fa98b4e
  Gas price: 0.0 gwei   Gas limit: 12000000   Nonce: 6
  BaseRewardPool.queueNewRewards confirmed   Block: 15221255   G

<Transaction '0xb88fe369ed4b49f8043d67354f3429f41a403fa98b4e74a8
>>> x.withdraw(1, False, {"from": a[0]})
Transaction sent: 0x28e3687fb982c9473cd4c79e5a04e66f90cc1ab7182c
  Gas price: 0.0 gwei   Gas limit: 12000000   Nonce: 7
  BaseRewardPool.withdraw confirmed   Block: 15221256   Gas used

<Transaction '0x28e3687fb982c9473cd4c79e5a04e66f90cc1ab7182d6d55
>>> history[-1].events
{'RewardUpdated': [OrderedDict([('user', '0x66aB6D9362d4F3559627
>>> 46296296296292 / 1e18
4.6296296296292e-05
>>>
```

Let's compare against a more non-malicious realistic deposit of 1e18 (Same as above but stake and withdraw for 1e18)

```
{'RewardUpdated': [OrderedDict([('user', '0x66aB6D9362d4F3559627
```

Seems to receive 3 times less rewards

I think there may be a lot more to uncover, and this finding would have benefitted by more time invested in coding a POC.

With the information that I have, it does seem like, due to the supply math, that early depositors can inflate the amount of rewards they receive by depositing a small amount.

> It may be advisable to discuss this with the Synthetix team to see historically how this can be griefed, and it may be ideal to require an initial deposit of at least 1e18 tokens on the first deposit.

> Because the finding is:

- Limited to loss of yield
- Lacks Coded POC showing how to steal all rewards (my due diligence denies that statement)

> I think Medium Severity is more appropriate

[M-24] User can lose funds

Submitted by csanuragjain

If `\_rewardToken` is set as `\_stakingToken` by mistake then user funds would get lost (staking token will get sent as reward token). Need to be fixed for BaseRewardPool.sol as well.

Proof of Concept

1. User A and B makes deposit of amount 100 each
2. Owner calls addReward and queueNewRewards to add 1 reward amount with _rewardToken as stakingToken (by mistake)
3. After some time reward is calculated as 5 for User A (total reward amount is same as staking amount which is 100+100+1).
4. User A makes the withdraw and obtains 105 amount and now User B is stuck since contract does not have enough funds

Recommended Mitigation Steps

Add below check in constructor

```
require(_stakingToken!=_rewardToken, "Incorrect reward token");
```

jetbrain10 (veToken Finance) disagreed with severity and commented:

> It's set by owner, so there is no issue, but it is nice to have and it is a quick fix.

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

> The warden has shown how, due to a misconfiguration, the deposit token `stakingToken` could be transferred away to early withdrawers.

> For end users the basic due diligence would be to ensure that the `stakingToken` is not added as reward.

> Because the finding is contingent on a malicious admin / misconfiguration, I believe Medium Severity to be appropriate.

[M-25] Consistently check account balance before and after transfers for Fee-On-Transfer discrepancies

Submitted by Dravee, also found by pauliax

https://github.com/code-423n4/2022-05-vetoken/blob/2d7cd1f6780a9bcc8387dea8fecfbd758462c152/contracts/Booster.sol#L356

https://github.com/code-423n4/2022-05-vetoken/blob/2d7cd1f6780a9bcc8387dea8fecfbd758462c152/contracts/VE3DRewardPool.sol#L337

Vulnerability Details

As arbitrary ERC20 tokens can be passed, the amount here should be calculated every time to take into consideration a possible fee-on-transfer or deflation.

Also, it's a good practice for the future of the solution.

Affected code:

- File: Booster.sol

```
    function deposit(
        uint256 _pid,
        uint256 _amount,
        bool _stake
    ) public returns (bool) {
...
        IERC20(lptoken).safeTransferFrom(msg.sender, staker
...
            ITokenMinter(token).mint(address(this), _amount
...
            IERC20(token).safeApprove(rewardContract, _amou
            IRewards(rewardContract).stakeFor(msg.sender, _
...
            ITokenMinter(token).mint(msg.sender, _amount);
...
        emit Deposited(msg.sender, _pid, _amount);
...
```

- File: VE3DRewardPool.sol

```
    function donate(address _rewardToken, uint256 _amount)
        IERC20(_rewardToken).safeTransferFrom(msg.sender, a
        rewardTokenInfo[_rewardToken].queuedRewards += _amo
    }
```

Recommended Mitigation Steps

Use the balance before and after the transfer to calculate the received amount instead of assuming that it would be equal to the amount passed as a parameter.

solvetony (veToken Finance) confirmed and commented:

> Confirmed in donate function, we are have to add a check for the reward token list However, I don't think we do anything with lp token as we already use the exact lp token address

Alex the Entreprenerd (judge) commented:

> While I think the scenario is highly unlikely, the warden has shown how, in the case of a `feeOnTransfer` token being added as reward, the claiming of reward could

> be potentially broken.

> Remediation can be as simple as checking for the actual transferred amount in the `donate` function, or simply not allowing the tokens.

> However, because the warden has shown how the system could be bricked due to a configuration issue, I believe Medium Severity to be appropriate.

[M-26] `VE3DLocker.sol` Wrong implementation of inversely traverse for loops always reverts

Submitted by WatchPug, also found by Dravee, gzeon, and TerrierLover

https://github.com/code-423n4/2022-05-vetoken/blob/2d7cd1f6780a9bcc8387dea8fecfbd758462c152/contracts/VE3DLocker.sol#L305-L329

https://github.com/code-423n4/2022-05-vetoken/blob/2d7cd1f6780a9bcc8387dea8fecfbd758462c152/contracts/VE3DLocker.sol#L349-L373

https://github.com/code-423n4/2022-05-vetoken/blob/2d7cd1f6780a9bcc8387dea8fecfbd758462c152/contracts/VE3DLocker.sol#L376-L396

https://github.com/code-423n4/2022-05-vetoken/blob/2d7cd1f6780a9bcc8387dea8fecfbd758462c152/contracts/VE3DLocker.sol#L399-L415

Vulnerability Details

```
function totalSupplyAtEpoch(uint256 _epoch) external view returr
    uint256 epochStart = uint256(epochs[_epoch].date).div(rewarc
        rewardsDuration
    );
    uint256 cutoffEpoch = epochStart.sub(lockDuration);

    //traverse inversely to make more current queries more gas e
```

```
        for (uint256 i = _epoch; i + 1 != 0; i--) {
            Epoch storage e = epochs[i];
            if (uint256(e.date) <= cutoffEpoch) {
                break;
            }
            supply = supply.add(epochs[i].supply);
        }

        return supply;
    }
```

In `VE3DLocker.sol`, there are multiple instances in which an inversely traverse for loop is used "to make more current queries more gas efficient".

For example:

- `totalSupplyAtEpoch()`
- `balanceAtEpochOf()`
- `pendingLockAtEpochOf()`
- `totalSupply()`

The implementation of the inversely traverse for loop is inherited from Convex's original version: https://github.com/convex-eth/platform/blob/main/contracts/contracts/CvxLockerV2.sol#L333-L334

However, Convex's locker contract is using Solidity 0.6.12, in which the arithmetic operations will overflow/underflow without revert.

As the solidity version used in the current implementation of `VE3DLocker.sol` is `0.8.7`, and there are some breaking changes in Solidity v0.8.0, including:

> Arithmetic operations revert on underflow and overflow.

Ref: https://docs.soliditylang.org/en/v0.8.7/080-breaking-changes.html#silent-changes-of-the-semantics

Which makes the current implementation of inversely traverse for loops always reverts.

More specifically:

1. `for (uint i = locks.length - 1; i + 1 != 0; i--) {` will revert when `locks.length == 0` at `locks.length - 1` due to underflow;
2. `for (uint256 i = _epoch; i + 1 != 0; i--) {` will loop until `i == 0` and reverts at `i--` due to underflow.

As a result, all these functions will be malfunctioning and all the internal and external usage of these function will always revert.

Recommended Mitigation Steps

Change `VE3DLocker.sol#L315` to:

```
for (uint256 i = locks.length; i > 0; i--) {
    uint256 lockEpoch = uint256(locks[i - 1].unlockTime).sub(loc
    //lock epoch must be less or equal to the epoch we're basing
    if (lockEpoch <= epochTime) {
        if (lockEpoch > cutoffEpoch) {
            amount = amount.add(locks[i - 1].amount);
```

Change `VE3DLocker.sol#L360` to:

```
for (uint256 i = locks.length; i > 0; i--) {
    uint256 lockEpoch = uint256(locks[i - 1].unlockTime).sub(loc

    //return the next epoch balance
    if (lockEpoch == nextEpoch) {
        return locks[i - 1].amount;
    } else if (lockEpoch < nextEpoch) {
        //no need to check anymore
        break;
    }
```

Change `VE3DLocker.sol#L387` to:

```
for (uint256 i = epochindex; i > 0; i--) {
```

```
        Epoch storage e = epochs[i - 1];
```

Change `VE3DLocker.sol#L406` to:

```
    for (uint256 i = _epoch + 1; i > 0; i--) {
        Epoch storage e = epochs[i - 1];
        if (uint256(e.date) <= cutoffEpoch) {
            break;
        }
        supply = supply.add(e.supply);
    }
```

[solvetony (veToken Finance) confirmed and commented](#):

> Looks valid.

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

> The warden has shed light into a underflow that will cause a few functions which logic was ported over from 0.6.12, to revert due to new checks introduced in a more recent version of solidity.

> While the bug is easily fixable, the functions are broken.

> Normally I would be conflicted between a Low and Medium severity as these functions are mostly `view` and seem to have no particular impact.

> However, due to my familiarity with other Lockers, and the CVX Voting Strategy used I believe the finding implications are that:

- Integrating strategies (tokenizedLocks of the VE3DLocker) would be bricked
- Snapshot wouldn't be usable as it would need to know the balance of a user at a specific epoch

> So am confident in a Medium Severity and have contemplated raising to High, as impact is pretty dramatic.

> Personally am very confident governance couldn't work without some of the `view` functions impacted, however, considering that the Warden (and other dups) did not bring in any mention of governance, I think Medium Severity is appropriate.

> I highly recommend the sponsor ensures their governance-related function don't revert as that would be very problematic.

[M-27] Booster's shutdownPool can freeze user funds

Submitted by hyh, also found by jonatascm

`shutdownPool()` marks shutdown successful even if it's not (i.e. when withdrawAll() call wasn't successful). As withdrawing logic expect that the pool in shutdown has already provided the funds, and makes no additional attempts to retrieve them, user funds will be frozen permanently as there are no mechanics in place to turn shutdown off for a pool.

Setting severity to medium as that's user principal funds freeze scenario conditional on any issues in withdrawAll().

Proof of Concept

In the case of unsuccessful withdrawAll() call the pool nevertheless will be marked as shutdown:

https://github.com/code-423n4/2022-05-vetoken/blob/2d7cd1f6780a9bcc8387dea8fecfbd758462c152/contracts/Booster.sol#L312-L320

```
        //withdraw from gauge
        try IStaker(staker).withdrawAll(pool.lptoken, pool.gauge

        pool.shutdown = true;
        gaugeMap[pool.gauge] = false;

        emit PoolShuttedDown(_pid);
        return true;
    }
```

It will block the withdrawals as there will be not enough funds to fulfil the claim:

https://github.com/code-423n4/2022-05-vetoken/blob/2d7cd1f6780a9bcc8387dea8fecfbd758462c152/contracts/Booster.sol#L408-L422

```
        //pull from gauge if not shutdown
        // if shutdown tokens will be in this contract
        if (!pool.shutdown) {
            IStaker(staker).withdraw(lptoken, gauge, _amount);
        }

        //some gauges claim rewards when withdrawing, stash then
        //do not call if shutdown since stashes wont have access
        address stash = pool.stash;
        if (stash != address(0) && !isShutdown && !pool.shutdowr
            IStash(stash).stashRewards();
        }

        //return lp tokens
        IERC20(lptoken).safeTransfer(_to, _amount);
```

This way user funds will be frozen as the system will not attempt to withdraw from the pool, while there will be no funds to transfer to the user and _withdraw() will be reverting on L422 safeTransfer call.

Recommended Mitigation Steps

The shutdownPool logic can to be updated:

https://github.com/code-423n4/2022-05-vetoken/blob/2d7cd1f6780a9bcc8387dea8fecfbd758462c152/contracts/Booster.sol#L307-L320

Consider unifying the logic with shutdownSystem() and marking the pool shutdown only if withdraw was successful, for example:

```
    //shutdown pool
    function shutdownPool(uint256 _pid, bool forced) external returr
        require(msg.sende == poolManager, "!auth");
```

```
        PoolInfo storage pool = poolInfo[_pid];

        bool withdrawSuccess = false;

        //withdraw from gauge
        try IStaker(staker).withdrawAll(pool.lptoken,pool.gauge){
            withdrawSuccess = true;
        } catch {}

        if (withdrawSuccess || forced) {
                pool.shutdown = true;
                gaugeMap[pool.gauge] = false;
            emit PoolShuttedDown(_pid);
            return true;
        }

        return false;
    }
```

jetbrain10 (veToken Finance) acknowledged, but disagreed with severity and commented:

> Convex has the same, and we may need force shutdown.

Alex the Entreprenerd (judge) commented:

> My issue with the submission is that it makes the following statement: `Setting severity to medium as that's user principal funds freeze scenario conditional on any issues in withdrawAll()`.

> Without mentioning any possible scenario in which `withdrawAll` could fail

> Given the information that I have we could have reverts if:

- Token is paused (e.g. Tether Blacklist)
- Gauge breaks (not much you can do there, pausing probably won't help recover funds)
- Misterious bugs in the code, none were mentioned above.

> Will think about severity.

Alex the Entreprenerd (judge) commented:

> After reviewing the rest of the contest, I remember marking a finding about Paused tokens with Medium Severity, so while I think the submission could have been made stronger by providing real-world examples of how the tokens could be stuck, to maintain consistency I'll mark this finding as valid as well.

> As while it's unspecified how a revert could happen, the try catch irreversible system does mean that a failed rescue bricks the tokens, perhaps a governance only secondary rescue system may be helpful for emergency scenarios.

[M-28] ExtraRewardStashV2's stashRewards can become unavailable

Submitted by hyh

There is no check for the reward token amount to be transferred out in stashRewards(). As reward token list is external (controlled with `IGauge(gauge).reward_tokens`), and an arbitrary token can end up there, in the case when such token doesn't allow for zero amount transfers, the stashRewards() managed extra rewards retrieval can become unavailable.

I.e. stashRewards() can be blocked for even an extended period of time, so all other extra rewards gathering will not be possible. This cannot be controlled by the system as pool reward token list is external.

Setting the severity to medium as reward gathering is a base functionality of the system and its availability is affected.

Proof of Concept

`stashRewards()` attempts to send the `amount` to rewardArbitrator() without checking:

https://github.com/code-423n4/2022-05-vetoken/blob/2d7cd1f6780a9bcc8387dea8fecfbd758462c152/contracts/ExtraRewardStashV2.sol#L193-L203

```
        if (activeCount > 1) {
            //take difference of before/after(only send new tokens)
            uint256 amount = IERC20(token).balanceOf(address(this));
            amount = amount.sub(before);

            //send to arbitrator
            address arb = IDeposit(operator).rewardArbitrator();
            if (arb != address(0)) {
                IERC20(token).safeTransfer(arb, amount);
            }
        }
```

If `IStaker(staker).withdraw()` produced no new tokens for any reason, the `amount = amount.sub(before)` above can be zero:

https://github.com/code-423n4/2022-05-vetoken/blob/2d7cd1f6780a9bcc8387dea8fecfbd758462c152/contracts/ExtraRewardStashV2.sol#L188-L189

```
        uint256 before = IERC20(token).balanceOf(address(this));
        IStaker(staker).withdraw(token);
```

As reward `token` can be arbitrary, it can also be reverting on an attempt to transfer zero amounts:

https://github.com/d-xo/weird-erc20#revert-on-zero-value-transfers

If this be the case then the whole stashRewards() call will be failing until `IStaker(staker).withdraw()` manage to withdraw some `tokens` or such `token` be removed from gauge's reward token list. Both events aren't directly controllable by the system.

Recommended Mitigation Steps

Consider running the transfer only when amount is positive:

```
-   if (activeCount > 1) {
+   if (amount > 0 && activeCount > 1) {
```

```
        //take difference of before/after(only send new tokens)
        uint256 amount = IERC20(token).balanceOf(address(this));
        amount = amount.sub(before);

        //send to arbitrator
        address arb = IDeposit(operator).rewardArbitrator();
        if (arb != address(0)) {
            IERC20(token).safeTransfer(arb, amount);
        }
    }
```

[jetbrain10 (veToken Finance) confirmed](#)

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

> The warden has shown how, due to a lack of check for zero-transfer, for specific tokens, the `stashRewads` function can be made to revert, causing it not to process any reward token

> Because this is contingent on the token having zero-balance and reverting, I think Medium Severity to be appropriate.

[M-29] Centralisation RIsk: `VoterProxy` owner may set the `operate` to an address they own and drain all token balances

Submitted by kirk-baird

https://github.com/code-423n4/2022-05-vetoken/blob/2d7cd1f6780a9bcc8387dea8fecfbd758462c152/contracts/VoterProxy.sol#L274-L285

https://github.com/code-423n4/2022-05-vetoken/blob/2d7cd1f6780a9bcc8387dea8fecfbd758462c152/contracts/VoterProxy.sol#L123-L143

Impact

The `owner` of `VoterProxy` is able to call `setOperator()` (if the previous operator is shutdown). This allows them to then call `execute()`, `withdraw()` or `withdrawAll()`.

Execute makes a call to any arbitrary contract with arbitrary data. This may therefore call any ERC20 token, and gauge or the `VoterEscrow` account and withdraw protocol funds.

The functions `withdraw()` and `withdrawAll()` can also be abused to take all funds deposited in the gauges and transfer them to the owner's malicious address.

This poses a significant centralisation risk if the owner private key is compromised or the owner decides to rug pull.

Proof of Concept

After the owner has updated the `operator` via `setOperator()` they are able to call `VoterProxy.execute()` to execute any call to any smart contract.

```
function execute(
    address _to,
    uint256 _value,
    bytes calldata _data
) external returns (bool, bytes memory) {
    require(msg.sender == operator, "!auth");


    (bool success, bytes memory result) = _to.call{value: _v
    require(success, "!success");


    return (success, result);
}
```

Similarly, for `withdraw()` and `withdrawAll()`

```
function withdraw(
    address _token,
    address _gauge,
```

```
        uint256 _amount
    ) public returns (bool) {
        require(msg.sender == operator, "!auth");
        uint256 _balance = IERC20(_token).balanceOf(address(this
        if (_balance < _amount) {
            _amount = _withdrawSome(_gauge, _amount.sub(_balance
            _amount = _amount.add(_balance);
        }
        IERC20(_token).safeTransfer(msg.sender, _amount);
        return true;
    }


    function withdrawAll(address _token, address _gauge) externa
        require(msg.sender == operator, "!auth");
        uint256 amount = balanceOfPool(_gauge).add(IERC20(_toker
        withdraw(_token, _gauge, amount);
        return true;
    }
```

Recommended Mitigation Steps

This issue may be mitigated removing the ability for the `owner` to change the operator in `VoterProxy`.

If the functionality is require ensure it is behind a time lock and multisig / dao.

jetbrain10 (veToken Finance) acknowledged, but disagreed with severity and commented:

> Admin will be controlled by DAO and muti-sig wallet.

Alex the Entreprenerd (judge) commented:

> The warden has shown how, due to a couple of permissioned functions, funds may be swept out of the `VoterProxy`.

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

> For end users I highly recommend making sure that not only the multi-sig is strong (high threshold), but also that is behind a time-lock to ensure you have time to react in case of emergency.

[M-30] Incorrect deployment parameters

Submitted by Picodes

https://github.com/code-423n4/2022-05-vetoken/blob/2d7cd1f6780a9bcc8387dea8fecfbd758462c152/migrations/25_deploy_angle_pools.js#L68

https://github.com/code-423n4/2022-05-vetoken/blob/2d7cd1f6780a9bcc8387dea8fecfbd758462c152/migrations/25_deploy_angle_pools.js#L80

Impact

The address of G-Uni tokens in the deployment scripts are not up to date.

Proof of Concept

For example for agEUR/USDC it is 0xedecb43233549c51cc3268b5de840239787ad56c and not 0x2bD9F7974Bc0E4Cb19B8813F8Be6034F3E772add.

Recommended Mitigation Steps

For safety why not fetching directly the LP token from the staking contract?

jetbrain10 (veToken Finance) confirmed

Alex the Entreprenerd (judge) commented:

> The warden has shown how a configuration file shows that the settings for the project are using an old address.

> While the finding pertains to a setup script (generally out of scope), given that:

- The sponsor has confirmed

- The finding is valid in that using older deployments will cause at the very least a loss of yield
- We already had an instance of bringing an out-of-scope file into scope via Sponsor-Confirming (See: #209)

> With the information I have, I believe the finding to be of Medium Severity and believe the sponsor will mitigate by updating to the Warden suggested addresses.

Low Risk and Non-Critical Issues

For this contest, 51 reports were submitted by wardens detailing low risk and non-critical issues. The report highlighted below by **IllIllI** received the top score from the judge.

The following wardens also submitted reports: kirk-baird, reassor, WatchPug, cryptphi, minhquanym, SmartSek, pauliax, Dravee, SecureZeroX, TerrierLover, sseefried, xiaoming90, 0xNazgul, csanuragjain, horsefacts, berndartmueller, dipp, Hawkeye, MiloTruck, sashik_eth, unforgiven, hansfriese, hyh, robee, 0x29A, catchup, ellahi, FSchmoede, gzeon, simon135, 0x1f8b, 0xf15ers, Deivitto, Funen, asutorufos, BouSalman, cccz, cogitoergosumsw, ElKu, GimelSec, shenwilly, sorrynotsorry, c3phas, Picodes, _Adam, delfin454000, oyc_109, 0xDjango, Chom, *and* z3s.

Low Risk Issues

	Issue	Instances
L-01	`getAPY()` returns the wrong answer during leap years	1
L-02	Missing checks for `address(0x0)` when assigning values to `address` state variables	26

Total: 27 instances over 2 issues

[L-01] `getAPY()` returns the wrong answer during leap years

There are more blocks in a year during a leap year. Using a static value for the number of blocks in a year will eventually lead to the wrong answer

There is 1 instance of this issue:

```
File: contracts/BaseRewardPool.sol   \#1

343      function getAPY() external view returns (uint256) {
344          return rewardRate.mul(BLOCKS_PER_YEAR).mul(1e18).di
345:     }
```

https://github.com/code-423n4/2022-05-vetoken/blob/2d7cd1f6780a9bcc8387dea8fecfbd758462c152/contracts/BaseRewardPool.sol#L343-L345

[L-02] Missing checks for `address(0x0)` when assigning values to `address` state variables

There are 26 instances of this issue. (For in-depth details on this and all further low & non-critical issues with multiple instances, see the warden's full report.

Non-critical Issues

	Issue	Instances
N-01	Adding a `return` statement when the function defines a named return variable, is redundant	1
N-02	`public` functions not called by the contract should be declared `external` instead	8
N-03	`constant` s should be defined rather than using magic numbers	10
N-04	Numeric values having to do with time should use time units for readability	1
N-05	Large multiples of ten should use scientific notation (e.g. `1e6`) rather than decimal literals (e.g. `1000000`), for readability	1
N-06	Missing event for critical parameter change	5
N-07	Use a more recent version of solidity	6

	Issue	Instances
N-08	Constant redefined elsewhere	10
N-09	Inconsistent spacing in comments	17
N-10	Typos	8
N-11	File is missing NatSpec	3
N-12	NatSpec is incomplete	1
N-13	Avoid the use of sensitive terms	1
N-14	`safeApprove()` is deprecated	12

Total: 84 instances over 14 issues

[N-01] Adding a `return` statement when the function defines a named return variable, is redundant

There is 1 instance of this issue:

```
File: contracts/VoterProxy.sol   \#1

119:          return balance;
```

https://github.com/code-423n4/2022-05-vetoken/blob/2d7cd1f6780a9bcc8387dea8fecfbd758462c152/contracts/VoterProxy.sol#L119

[N-02] `public` functions not called by the contract should be declared `external` instead

Contracts are allowed to override their parents' functions and change the visibility from `external` to `public`.

There are 8 instances of this issue.

[N-03] `constant` s should be defined rather than using magic numbers

There are 10 instances of this issue.

[N-04] Numeric values having to do with time should use time units for readability

There are units for seconds, minutes, hours, days, and weeks

There is 1 instance of this issue:

```
File: contracts/VeAssetDepositor.sol   \#1

/// @audit 86400
18:       uint256 private constant WEEK = 7 * 86400;
```

https://github.com/code-423n4/2022-05-vetoken/blob/2d7cd1f6780a9bcc8387dea8fecfbd758462c152/contracts/VeAssetDepositor.sol#L18

[N-05] Large multiples of ten should use scientific notation (e.g. `1e6`) rather than decimal literals (e.g. `1000000`), for readability

There is 1 instance of this issue:

```
File: contracts/VeTokenMinter.sol   \#1

15:       uint256 public constant maxSupply = 30 * 1000000 * 1e1
```

https://github.com/code-423n4/2022-05-vetoken/blob/2d7cd1f6780a9bcc8387dea8fecfbd758462c152/contracts/VeTokenMinter.sol#L15

[N-06] Missing event for critical parameter change

There are 5 instances of this issue.

[N-07] Use a more recent version of solidity

Use a solidity version of at least 0.8.13 to get the ability to use `using for` with a list of free functions

There are 6 instances of this issue.

[N-08] Constant redefined elsewhere

Consider defining in only one contract so that values cannot become out of sync when only one location is updated. A cheap way to store constants in a single location is to create an `internal constant` in a `library`. If the variable is a local cache of another contract's value, consider making the cache variable internal or private, which will require external users to query the contract with the source of truth, so that callers don't get out of sync.

There are 10 instances of this issue.

[N-09] Inconsistent spacing in comments

Some lines use `// x` and some use `//x`. The instances below point out the usages that don't follow the majority, within each file

There are 17 instances of this issue.

[N-10] Typos

There are 8 instances of this issue.

[N-11] File is missing NatSpec

There are 3 instances of this issue.

[N-12] NatSpec is incomplete

There is 1 instance of this issue:

```
File: contracts/VoterProxy.sol   \#1

/// @audit Missing: '@param bytes'
191       /**
192        * @notice  Verifies that the hash is valid
193        * @dev     Snapshot Hub will call this function when
194        *          snapshot.js on behalf of this contract. Sr
195        *          function with the hash and the signature c
196        * @param _hash Hash of the message that was sent to S
197        * @return EIP1271 magic value if the signature is val
198        */
199:      function isValidSignature(bytes32 _hash, bytes memory)
```

https://github.com/code-423n4/2022-05-vetoken/blob/2d7cd1f6780a9bcc8387dea8fecfbd758462c152/contracts/VoterProxy.sol#L191-L199

[N-13] Avoid the use of sensitive terms

Use alternative variants, e.g. allowlist/denylist instead of whitelist/blacklist

There is 1 instance of this issue:

```
File: contracts/VE3DRewardPool.sol   \#1
204:         //fees dont apply until whitelist+veVeAsset lock be
```

https://github.com/code-423n4/2022-05-vetoken/blob/2d7cd1f6780a9bcc8387dea8fecfbd758462c152/contracts/VE3DRewardPool.sol#L204

[N-14] `safeApprove()` is deprecated

Deprecated in favor of `safeIncreaseAllowance()` and `safeDecreaseAllowance()`. If only setting the initial allowance to the value that means infinite, `safeIncreaseAllowance()` can be used instead

There are 12 instances of this issue.

jetbrain10 (veToken Finance) commented:

> High quality.

Alex the Entreprenerd (judge) commented:

> 2L, 5R, 7NC

> *(Note: See* **original submission** *for judge's full commentary.)*

Gas Optimizations

For this contest, 48 reports were submitted by wardens detailing gas optimizations. The **report highlighted below** by **IllIllI** received the top score from the judge.

The following wardens also submitted reports: **0x1f8b**, **MiloTruck**, **sashik_eth**, **_Adam**, **Funen**, **Tomio**, **jonatascm**, **robee**, **cogitoergosumsw**, **SecureZeroX**, **WatchPug**, **SmartSek**, **catchup**, **fatherOfBlocks**, **FSchmoede**, **0x29A**, **0xKitsune**, **Dravee**, **gzeon**, **GalloDaSballo**, **horsefacts**, **0xf15ers**, **0xNazgul**, **c3phas**, **delfin454000**, **Kaiziron**, **simon135**, **TomJ**, **0xkatana**, **asutorufos**, **Cityscape**, **ElKu**, **ellahi**, **hansfriese**, **minhquanym**, **oyc_109**, **pauliax**, **Randyyy**, **reassor**, **RoiEvenHaim**, **sach1r0**, **saian**, **TerrierLover**, **Waze**, **z3s**, **Hawkeye**, *and* **Ruhum**.

Summary

	Issue	Instances
G-01	Update to state var should only happen once in a function	1
G-02	Multiple `address` mappings can be combined into a single `mapping` of an `address` to a `struct`, where appropriate	1
G-03	State variables only set in the constructor should be declared `immutable`	11
G-04	Avoid contract existence checks by using solidity version 0.8.10 or later	124

	Issue	Instances
G-05	State variables should be cached in stack variables rather than re-reading them from storage	39
G-06	Multiple accesses of a mapping/array should use a local variable cache	30
G-07	`<x> += <y>` costs more gas than `<x> = <x> + <y>` for state variables	3
G-08	`internal` functions only called once can be inlined to save gas	2
G-09	`<array>.length` should not be looked up in every loop of a `for`-loop	7
G-10	`++i` / `i++` should be `unchecked{++i}` / `unchecked{i++}` when it is not possible for them to overflow, as is the case when used in `for`- and `while`-loops	13
G-11	`keccak256()` should only need to be called on a specific string literal once	1
G-12	Using `bool`s for storage incurs overhead	5
G-13	Using `> 0` costs more gas than `!= 0` when used on a `uint` in a `require()` statement	7
G-14	It costs more gas to initialize variables to zero than to let the default of zero be applied	14
G-15	`++i` costs less gas than `i++`, especially when it's used in `for`-loops (`--i` / `i--` too)	13
G-16	Splitting `require()` statements that use `&&` saves gas	2
G-17	Using `private` rather than `public` for constants, saves gas	9
G-18	Don't compare boolean expressions to boolean literals	7
G-19	Don't use `SafeMath` once the solidity version is 0.8.0 or greater	6
G-20	Duplicated `require()` / `revert()` checks should be refactored to a modifier or function	15
G-21	`require()` or `revert()` statements that check input arguments should be at the top of the function	2

	Issue	Instances
G-22	Empty blocks should be removed or emit something	3
G-23	Use custom errors rather than `revert()` / `require()` strings to save deployment gas	67
G-24	Functions guaranteed to revert when called by normal users can be marked `payable`	7
G-25	Don't use `_msgSender()` if not supporting EIP-2771	2

Total: 391 instances over 25 issues

[G-01] Update to state var should only happen once in a function

The assignment of `totalWeight` can be optimized by summing the old and new weights

There is 1 instance of this issue:

```
File: contracts/VeTokenMinter.sol   \#1

43            totalWeight -= veAssetWeights[veAssetOperator];
44            veAssetWeights[veAssetOperator] = newWeight;
45:           totalWeight += newWeight;
```

https://github.com/code-423n4/2022-05-vetoken/blob/2d7cd1f6780a9bcc8387dea8fecfbd758462c152/contracts/VeTokenMinter.sol#L43-L45

[G-02] Multiple `address` mappings can be combined into a single `mapping` of an `address` to a `struct`, where appropriate

Saves a storage slot for the mapping. Depending on the circumstances and sizes of types, can avoid a Gsset (**20000 gas**) per mapping combined. Reads and

subsequent writes can also be cheaper when a function requires both values and they both fit in the same storage slot. Finally, if both fields are accessed in the same function, can save **~42 gas per access** due to not having to recalculate the key's keccak256 hash (Gkeccak256 - **30 gas**) and that calculation's associated stack operations.

There is 1 instance of this issue:

```
File: contracts/BaseRewardPool.sol   \#1

75        mapping(address => uint256) public userRewardPerTokenP
76        mapping(address => uint256) public rewards;
77:       mapping(address => uint256) private _balances;
```

https://github.com/code-423n4/2022-05-vetoken/blob/2d7cd1f6780a9bcc8387dea8fecfbd758462c152/contracts/BaseRewardPool.sol#L75-L77

[G-03] State variables only set in the constructor should be declared `immutable`

Avoids a Gsset (**20000 gas**) in the constructor, and replaces each Gwarmacces (**100 gas**) with a `PUSH32` (**3 gas**).

There are 11 instances of this issue. (For in-depth details on this and all further gas optimizations with multiple instances, see the warden's [full report](#)*.)*

[G-04] Avoid contract existence checks by using solidity version 0.8.10 or later

Prior to 0.8.10 the compiler inserted extra code, including `EXTCODESIZE` (**700 gas**), to check for contract existence for external calls. In more recent solidity versions, the compiler will not insert these checks if the external call has a return value

There are 124 instances of this issue.

[G-05] State variables should be cached in stack variables rather than re-reading them from storage

The instances below point to the second+ access of a state variable within a function. Caching of a state variable replace each Gwarmaccess (**100 gas**) with a much cheaper stack read. Other less obvious fixes/optimizations include having local memory caches of state variable structs, or having local caches of state variable contracts/addresses.

There are 39 instances of this issue.

[G-06] Multiple accesses of a mapping/array should use a local variable cache

The instances below point to the second+ access of a value inside a mapping/array, within a function. Caching a mapping's value in a local `storage` variable when the value is accessed [multiple times](), saves **~42 gas per access** due to not having to recalculate the key's keccak256 hash (Gkeccak256 - **30 gas**) and that calculation's associated stack operations. Caching an array's struct avoids recalculating the array offsets into memory

There are 30 instances of this issue.

[G-07] `<x> += <y>` costs more gas than `<x> = <x> + <y>` for state variables

There are 3 instances of this issue.

[G-08] `internal` functions only called once can be inlined to save gas

Not inlining costs **20 to 40 gas** because of two extra `JUMP` instructions and additional stack operations needed for function calls.

There are 2 instances of this issue.

[G-09] `<array>.length` should not be looked up in every loop of a `for`-loop

The overheads outlined below are *PER LOOP*, excluding the first loop

- storage arrays incur a Gwarmaccess (**100 gas**)
- memory arrays use `MLOAD` (**3 gas**)
- calldata arrays use `CALLDATALOAD` (**3 gas**)

Caching the length changes each of these to a `DUP<N>` (**3 gas**), and gets rid of the extra `DUP<N>` needed to store the stack offset

There are 7 instances of this issue.

[G-10] `++i` / `i++` should be `unchecked{++i}` / `unchecked{i++}` when it is not possible for them to overflow, as is the case when used in `for`- and `while`-loops

The `unchecked` keyword is new in solidity version 0.8.0, so this only applies to that version or higher, which these instances are. This saves **30-40 gas** per loop

There are 13 instances of this issue.

[G-11] `keccak256()` should only need to be called on a specific string literal once

It should be saved to an immutable variable, and the variable used instead. If the hash is being used as a part of a function selector, the cast to `bytes4` should also only be done once

There is 1 instance of this issue:

```
File: contracts/Booster.sol   \#1

488:                bytes4(keccak256("set_rewards_receiver(address
```

https://github.com/code-423n4/2022-05-vetoken/blob/2d7cd1f6780a9bcc8387dea8fecfbd758462c152/contracts/Booster.sol#L488

[G-12] Using `bool` s for storage incurs overhead

```
// Booleans are more expensive than uint256 or any type that
// word because each write operation emits an extra SLOAD to
// slot's contents, replace the bits taken up by the boolean
// back. This is the compiler's defense against contract upg
// pointer aliasing, and it cannot be disabled.
```

https://github.com/OpenZeppelin/openzeppelin-contracts/blob/58f635312aa21f947cae5f8578638a85aa2519f5/contracts/security/ReentrancyGuard.sol#L23-L27

Use `uint256(1)` and `uint256(2)` for true/false to avoid a Gwarmaccess (100 gas), and to avoid Gsset (**20000 gas**) when changing from 'false' to 'true', after having been 'true' in the past

There are 5 instances of this issue.

[G-13] Using `> 0` costs more gas than `!= 0` when used on a `uint` in a `require()` statement

This change saves 6 gas per instance

There are 7 instances of this issue.

[G-14] It costs more gas to initialize variables to zero than to let the default of zero be applied

There are 14 instances of this issue.

[G-15] `++i` costs less gas than `i++`, especially when it's used in `for`-loops (`--i` / `i--` too)

Saves **6 gas per loop**

There are 13 instances of this issue.

[G-16] Splitting `require()` statements that use `&&` saves gas

See [this issue](#) which describes the fact that there is a larger deployment gas cost, but with enough runtime calls, the change ends up being cheaper

There are 2 instances of this issue.

[G-17] Using `private` rather than `public` for constants, saves gas

If needed, the value can be read from the verified contract source code. Savings are due to the compiler not having to create non-payable getter functions for deployment calldata, and not adding another entry to the method ID table

There are 9 instances of this issue.

[G-18] Don't compare boolean expressions to boolean literals

`if (<x> == true)` **=>** `if (<x>)` , `if (<x> == false)` **=>** `if (!<x>)`

There are 7 instances of this issue.

[G-19] Don't use `SafeMath` once the solidity version is 0.8.0 or greater

Version 0.8.0 introduces internal overflow checks, so using `SafeMath` is redundant and adds overhead

There are 6 instances of this issue.

[G-20] Duplicated `require()` / `revert()` checks should be refactored to a modifier or function

Saves deployment costs

There are 15 instances of this issue.

[G-21] `require()` or `revert()` statements that check input arguments should be at the top of the function

Checks that involve constants should come before checks that involve state variables

There are 2 instances of this issue.

[G-22] Empty blocks should be removed or emit something

The code should be refactored such that they no longer exist, or the block should do something useful, such as emitting an event or reverting. If the contract is meant to be extended, the contract should be `abstract` and the function signatures be added without any default implementation. If the block is an empty if-statement block to avoid doing subsequent checks in the else-if/else conditions, the else-if/else conditions should be nested under the negation of the if-statement, because they involve different classes of checks, which may lead to the introduction of errors when the code is later modified (`if(x){}else if(y){...}else{...}` => `if(!x){if(y){...}else{...}}`)

There are 3 instances of this issue.

[G-23] Use custom errors rather than `revert()` / `require()` strings to save deployment gas

Custom errors are available from solidity version 0.8.4. The instances below match or exceed that version

There are 67 instances of this issue.

[G-24] Functions guaranteed to revert when called by normal users can be marked `payable`

If a function modifier such as `onlyOwner` is used, the function will revert if a normal user tries to pay the function. Marking the function as `payable` will lower the gas cost for legitimate callers because the compiler will not include checks for whether a payment was provided. The extra opcodes avoided are `CALLVALUE` (2), `DUP1` (3), `ISZERO` (3), `PUSH2` (3), `JUMPI` (10), `PUSH1` (3), `DUP1` (3), `REVERT` (0), `JUMPDEST` (1), `POP` (2), which costs an average of about **21 gas per call** to the function, in addition to the extra deployment cost

There are 7 instances of this issue.

[G-25] Don't use `_msgSender()` if not supporting EIP-2771

Use `msg.sender` if the code does not implement [EIP-2771 trusted forwarder](#) support

There are 2 instances of this issue.

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

> Most thorough submission, would like for the warden to add:
>
> - Total Gas Saved per finding next to headline
> - POC for some of the contentious stuff (refactor storage, pack, use bool), because some of the advice while generally sound, requires a refactoring that leverages the concept.
>
> Overall best report for the contest
>
> Total Gas Saved 46633
>
> *(Note: See* [original submission](#) *for judge's full commentary.)*

Disclosures

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

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

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

Top