Backd Tokenomics contest Findings & Analysis Report

2022-07-19

Table of contents

Overview

About C4

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

A C4 audit 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 Backd Tokenomics smart contract system written in Solidity. The audit contest took place between May 27—June 3 2022.

Wardens

66 Wardens contributed reports to the Backd Tokenomics contest:

1. WatchPug (jtp and ming)
2. scaraven
3. Picodes
4. 0x52
5. hansfriese
6. 0xNineDec
7. csanuragjain
8. SmartSek (0xDjango and hake)
9. fatherOfBlocks
10. IIIIIII
11. Ruhum
12. shenwilly
13. 0x1f8b
14. unforgiven
15. StyxRave
16. JC
17. peritoflores

18. BowTiedWardens (BowTiedHeron, BowTiedPickle, m4rio_eth, Dravee and BowTiedFirefox)
19. MiloTruck
20. SecureZeroX
21. berndartmueller
22. defsec
23. cccz
24. sashik_eth
25. c3phas
26. Chom
27. 0x29A (0x4non and rotcivegaf)
28. Funen
29. 0xNazgul
30. Sm4rty
31. 0xf15ers (remora and twojoy)
32. asutorufos
33. delfin454000
34. gzeon
35. hake
36. sach1r0
37. simon135
38. oyc_109
39. catchup
40. Kaiziron
41. Waze
42. robee
43. cryptphi
44. dipp
45. codexploder
46. bardamu

47. masterchief
48. Kumpa
49. hyh
50. 0xKitsune
51. Tadashi
52. Dravee
53. djxploit
54. Tomio
55. 0xkatana
56. Fitraldys
57. Randyyy
58. RoiEvenHaim

This contest was judged by Alex the Entreprenerd.

Final report assembled by itsmetechjay.

Summary

The C4 analysis yielded an aggregated total of 20 unique vulnerabilities. Of these vulnerabilities, 2 received a risk rating in the category of HIGH severity and 18 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 42 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 41 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 Backd Tokenomics contest repository, and is composed of 22 smart contracts written in the Solidity programming language and includes 2,507 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 (2)

[H-01] `Minter.sol#startInflation()` can be bypassed.

Submitted by WatchPug, also found by 0x52

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/tokenomics/Minter.sol#L104-L108

```
function startInflation() external override onlyGovernance {
    require(lastEvent == 0, "Inflation has already started."
    lastEvent = block.timestamp;
    lastInflationDecay = block.timestamp;
}
```

As `lastEvent` and `lastInflationDecay` are not initialized in the `constructor()`, they will remain to the default value of `0`.

However, the permissionless `executeInflationRateUpdate()` method does not check the value of `lastEvent` and `lastInflationDecay` and used them directly.

As a result, if `executeInflationRateUpdate()` is called before `startInflation()`:

1. L190, the check of if `_INFLATION_DECAY_PERIOD` has passed since `lastInflationDecay` will be `true`, and `initialPeriodEnded` will be set to `true` right away;
2. L188, since the `lastEvent` in `totalAvailableToNow += (currentTotalInflation * (block.timestamp - lastEvent));` is `0`, the `totalAvailableToNow` will be set to `totalAvailableToNow ≈ currentTotalInflation * 52 years`, which renders the constrains of `totalAvailableToNow` incorrect and useless.

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/tokenomics/Minter.sol#L115-L117

```
function executeInflationRateUpdate() external override retu
    return _executeInflationRateUpdate();
}
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/tokenomics/Minter.sol#L187-L215

```
function _executeInflationRateUpdate() internal returns (boo
    totalAvailableToNow += (currentTotalInflation * (block.t
    lastEvent = block.timestamp;
    if (block.timestamp >= lastInflationDecay + _INFLATION_D
        currentInflationAmountLp = currentInflationAmountLp.
        if (initialPeriodEnded) {
            currentInflationAmountKeeper = currentInflationA
                annualInflationDecayKeeper
            );
            currentInflationAmountAmm = currentInflationAmou
                annualInflationDecayAmm
            );
```

```
            } else {
                currentInflationAmountKeeper =
                    initialAnnualInflationRateKeeper /
                    _INFLATION_DECAY_PERIOD;

                currentInflationAmountAmm = initialAnnualInflati
                initialPeriodEnded = true;
            }
            currentTotalInflation =
                currentInflationAmountLp +
                currentInflationAmountKeeper +
                currentInflationAmountAmm;
            controller.inflationManager().checkpointAllGauges();
            lastInflationDecay = block.timestamp;
        }
        return true;
    }
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/tokenomics/Minter.sol#L50-L51

```
    // Used for final safety check to ensure inflation is not ex
    uint256 public totalAvailableToNow;
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/tokenomics/Minter.sol#L217-L227

```
    function _mint(address beneficiary, uint256 amount) internal
        totalAvailableToNow += ((block.timestamp - lastEvent) *
        uint256 newTotalMintedToNow = totalMintedToNow + amount;
        require(newTotalMintedToNow <= totalAvailableToNow, "Min
        totalMintedToNow = newTotalMintedToNow;
        lastEvent = block.timestamp;
        token.mint(beneficiary, amount);
        _executeInflationRateUpdate();
        emit TokensMinted(beneficiary, amount);
        return true;
    }
```

Recommendation

Consider initializing `lastEvent`, `lastInflationDecay` in `constructor()`.

or

Consider adding `require(lastEvent != 0 && lastInflationDecay != 0, "...")` to `executeInflationRateUpdate()`.

danhper (Backd) confirmed

Alex the Entreprenerd (judge) commented:

> The warden has shown how `startInflation` can be bypassed, breaking the Access Control as well as the Sponsor Goal for Emissions.

> Because of this I believe High Severity to be appropriate.

> The sponsor has mitigated in a subsequent PR

[H-02] Total Supply is not guaranteed and is not deterministic.

Submitted by Picodes, also found by scaraven

The actual total supply of the token is random and depends on when `_executeInflationRateUpdate` is executed.

Proof of Concept

The `README` and tokenomic documentation clearly states that "The token supply is limited to a total of 268435456 tokens." However when executing `_executeInflationRateUpdate`, it first uses the current inflation rate to update the total available before checking if it needs to be reduced.

Therefore if no one mints or calls `executeInflationRateUpdate` for some time around the decay point, the inflation will be updated using the previous rate so the `totalAvailableToNow` will grow too much.

Mitigation Steps

You should do

```
    totalAvailableToNow += (currentTotalInflation * (block.timestamp
```

Only if the condition `block.timestamp >= lastInflationDecay + _INFLATION_DECAY_PERIOD` is false.

Otherwise you should do

```
    totalAvailableToNow += (currentTotalInflation * (lastInflationDe
```

Then update the rates, then complete with

```
    totalAvailableToNow += (currentTotalInflation * (block.timestamp
```

Note that as all these variables are either constants either already loaded in memory this is super cheap to do.

danhper (Backd) confirmed, but disagreed with severity and commented:

> I believe this should actually be high severity

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

> The warden has identified the lack of an upper bound on the inflation math which would make it so that more than the expected supply cap of the token could be minted.

> The sponsor agrees that this should be of High Severity.

> Because this breaks the protocol stated invariant of a specific cap of `268435456` tokens, I agree with High Severity.

Medium Risk Findings (18)

[M-01] DoS on KeeperGauge due to division by zero

Submitted by fatherOfBlocks

In the **_calcTotalClaimable()** function it should be validated that perPeriodTotalFees[i] != 0 since otherwise it would generate a DoS in **claimableRewards()** and **claimRewards()**.
This would be possible since if **advanceEpoch()** or **kill()** is executed by the InflationManager address, the epoch will go up without perPeriodTotalFees[newIndexEpoch] is 0.
The negative of this is that every time the **InflationManager** executes these two methods (**kill() and advanceEpoch()**) DoS is generated until you run **reportFees()**.
Another possible case is that **kill()** or **advanceEpoch()** are executed 2 times in a row and there is no way of a perPeriodTotalFees[epoch-1] updating its value, therefore it would be an irreversible DoS.

Recommended Mitigation Steps

Generate a behavior for the case that perPeriodTotalFees[i] == 0.

samwerner (Backd) confirmed, but disagreed with severity

Alex the Entreprenerd (judge) commented:

> The finding at face value is valid and it's impact is a DOS.

> A division by zero in `keeperRecords[beneficiary].feesInPeriod[i].scaledDiv(perPeriodTotalFees[i])`, will cause an instantaneous revert making it impossible to claim fees.

> After more reading I believe that the finding is valid and can cause issues if there's one epoch without fees, if for any reason the protocol skips one epoch via `advanceEpoch`, while no `perPeriodTotalFees` are increased, then, because `keeperRecords[beneficiary].nextEpochToClaim` will include the empty epoch, the `beneficiary` will not be able to receive fees.

> Given that setup is necessary for this to happen, I believe Medium Severity to be more appropriate.

[M-02] The first AMM Staker will have control over how the shares are calculated.

Submitted by 0xNineDec

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/tokenomics/AmmGauge.sol#L147

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/tokenomics/AmmGauge.sol#L154

Impact

The first staker can take control of how the subsequent shares are going to be distributed by simply staking 1wei amount of the token and frontrunning future stakers. The reasons of this are related on how the variables are updated and with the amounts that the Gauge allows users to stake (anything but zero). The origin of this vulnerability relies on the evaluation of the `totalStaked` variable on its inception.

Proof of Concept

To illustrate this attack, an environment of testing was made in order to track the token flows and how the variables are being updated and read.

The initial or border conditions taken into account are the same as the ones used by the team to perform the tests and just a few assumptions and simplifications were taken.

1. The inflation rate is fixed for simplicity (`0.001`). This is valid within a short period of time because it is not a function of how the tokens are distributed or their flows. By tracking how the inflation rate is calculated an updated, we see that it is managed by the `currentInflationAmountAmm` within the `Minter.sol` contract, which value is modified by `_executeInflationRateUpdate()` three

lines below the last code permalink. Its value depends on non-token balance related parameters (such as inflation decays and annual rates).

2. For the testing environment performed by the team, a DummyERC20 was used as testing token. The same is done on the exploit environment.
3. The controller is not used because it is used to retrieve the inflation rate and it is now fixed because of 1).

Each user state is updated whenever he calls either `stake`, `unstake` or `claimRewards`.

Steps:

- Alice is the first staker and deposits 1wei worth of DummyERC20.
- Bob takes one day to find out this new protocol and decides to stake 10 ETH amount of tokens (`10 * 10**decimals()`).
- Alice, who was scanning the mempool, frontruns Bob with the same amount he was willing to stake. Her txn is mined first.
- Then Bobs' transaction is mined for the 10 ETH worth.
- Sometime after this, the pool is checkpointed.
- A few days pass, and Bob wants to stake even more tokens. The same amount as before.
- Alice frontruns him again updating her shares.
- Bobs' transaction is mined and his shares are also updated.
- The pool is checkpointed again. And Alice managed to increase considerably her amount of shares.

Both cases were evaluated (with and without staking 1 wei first). The attack scenario outputs a 100% more shares to Alice than Bob in comparison with the ethical/non-attack situation.

The code used to perform this test is the following:

```
    it("First Depositer Exploit", async function () {
            let userShares = []
            let userIntegral = []
            let userBalance = []
```

```
let globalIntegral, totalStaked;
let aliceBob = [alice, bob];

// Starting Checkpoint
await this.ammgauge.poolCheckpoint();
await ethers.provider.send("evm_increaseTime", [1 * 24 *

const updateStates = async () => {
    userShares = []
    userIntegral = []
    userBalance = []
    for (const user of aliceBob) {
        let balances = ethers.utils.formatEther(await th
        let currentShare = ethers.utils.formatEther(awai
        let currentStakedIntegral = ethers.utils.formatE
        userShares.push(currentShare);
        userIntegral.push(currentStakedIntegral);
        userBalance.push(balances);
    }
    globalIntegral = await this.ammgauge.ammStakedIntegra
    totalStaked = await this.ammgauge.totalStaked()
    console.log("  ")
    console.log("         ALICE / BOB");
    console.log(`Shares: ${userShares}`);
    console.log(`Integr: ${userIntegral}`);
    console.log(`Balanc: ${userBalance}`);
    console.log("  ")
    console.log("Global")
    console.log(`Integral: ${ethers.utils.formatEther(gl
}

const stake = async (to, amount) => {
    await updateStates()
    console.log(" ")
    // Balance before
    let balanceBefore = await this.ammgauge.balances(to.
    // Stake
    await this.ammgauge.connect(to).stake(amount);
    expect(await this.ammgauge.balances(to.address)).to.
    // await updateStates();
    console.log(" ")
}

const unstake = async (to, amount) => {
    await updateStates()
```

```
        console.log(" ")
        // Balance before
        let balanceBefore = await this.ammgauge.balances(to.
        // Stake
        await this.ammgauge.connect(to).unstake(amount);
        expect(await this.ammgauge.balances(to.address)).to.
        await updateStates();
        console.log(" ")
    }

    // HERE IS WHERE THE SIMULATION IS PERFORMED
    let simulationTimes = 2;
    let withOneWeiDeposit = true;

    if (withOneWeiDeposit) {
        // Alice deposits first
        console.log("Alice Deposits 1wei")
        let firstUserDeposit = ethers.utils.parseEther("1");
        await stake(alice, 1);
    }

    for (let index = 1; index <= simulationTimes; index++) {
        console.log(" ")
        console.log(`Loop number ${index}`);
        console.log(" ")

        console.log("A day passes until Bob decides to depos
        await ethers.provider.send("evm_increaseTime", [1 * 

        console.log(" ")
        console.log("She scans that Bob is about to stake 10
        console.log("Alice Frontruns")
        let frontrunAmount = ethers.utils.parseEther("10");
        await stake(alice, frontrunAmount);

        console.log(" ")
        console.log("Bob stakes 10 tokens")
        await stake(bob, frontrunAmount)

        // A few days pass
        await ethers.provider.send("evm_increaseTime", [1 * 
        // The pool is checkpointed
        await this.ammgauge.poolCheckpoint();
        console.log("After 1 day the pool is checkpointed")
        await updateStates()
```

```
        }
    })
```

The simulation was both made for the attacked and non attacked situations. The values that are shown represent how the contract updates them (the `totalStaked` variable is 0 when first Alice calls the stake function after `_userCheckpoint()` rans)

WITH 1WEI STAKE (ATTACK)

time	Situation	totalStaked	Alice Shares	Bob Shares
0-	First poolCheckpoint	0	0	0
0+	Alice Deposits 1wei	0	0	0
1	Alice frontruns Bob @ 10eth	1wei	0	0
2	Bob 10eth txn is mined	10eth + 1wei	86.4	0
3	1 day later poolCheckpoint() is called	20eth + 1 wei	86.4	0
4	Alice frontruns Bob again	20eth + 1 wei	86.4	0
5	Bob 10eth txn is mined	30eth + 1wei	172.8	0
6	1 day later poolCheckpoint() is called	40eth + 1wei	172.8	86.4

WITHOUT THE 1WEI STAKE (No “first staker hijack”)

time	Situation	totalStaked	Alice Shares	Bob Shares
0-	First poolCheckpoint	0	0	0
0+	Alice stakes 10eth	0	0	0
1	Bob stakes 10eth	10eth	0	0
2	1 day later poolCheckpoint() is called	20eth	0	0
3	Alice stakes 10eth	20eth	0	0
4	Bob stakes 10eth	30eth	86.4	0
5	1 day later poolCheckpoint() is called	40eth	86.4	86.4

Recommended Mitigation Steps

Further evaluation on how the variables are updated and how does the `Integral` (both each users and global one) is calculated on the pool inception is needed to

patch this issue.

danhper (Backd) confirmed, but disagreed with severity

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

> The warden has identified a way for the `rewardsShares` to be improperly assigned.
>
> The exploit is based on the fact that if `totalStaked` is zero we effectively will skip the first loop of points.
>
> This can create situations where certain users are rewarded unfairly in comparison to their initial deposit.
>
> However, this only applies when we go from zero to non-zero for `totalStaked` henced the scenario proposed by the warden (1 day of free rewards for first depositor) is actually the worst case scenario.
>
> Having the deployer do 2 or 3 initial deposits should mitigate this attack, which ultimately is limited to a leak of value to the first user depositing.
>
> For those reasons, I believe the finding to be valid and of Medium Severity.

[M-03] THE first AMM Staker may not receive according rewards because of poor checkpoints

Submitted by 0xNineDec

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/tokenomics/AmmGauge.sol#L56

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/tokenomics/AmmGauge.sol#L140

Impact

The first staker within the `AmmGauge` may not get the rewards if the pool is not checkpointed right after he stakes and before he wants to claim the rewards.

Proof of Concept

A testing environment that reproduces how the protocol is going to be deployed and managed is used to evaluate this case under the following assumptions and simplifications.

1. The inflation rate is fixed for simplicity (`0.001`).
2. For the testing environment performed by the team, a DummyERC20 was used as testing token. The same is done on the exploit environment.
3. The minting of tokens impact both on the inflation calculation and their balance. But this test evaluates the states just before minting (claimable balances). Following how the pools are updated, they are checkpointed in the end of the `_executeInflationRateUpdate` call. Not while staking.

In order to illustrate this scenario we will show both the vulnerable and non vulnerable situations.

Vulnerable Situation:

1. Alice, Bob, Charlie and David are future users of the pool. They all notice the inception of this project and decide to stake.
2. They all stake the same amount. Their transactions are mined with 1min of difference starting from Alice and finishing with David.
3. There is no external pool checkpoint between Alice and Bob (besides the one that is triggered when Bob stakes).
4. Sometime happens and they all want to check their accumulated reward balance. Alice accumulated much less than the others.

Non Vulnerable Situation:

- The same as before but calling externally `_poolCheckpoint()` between Alice stake call and Bobs' and before checking the accumulated rewards.

The code to show this has a `secureCheckpoints` toggle that can be set as true or false to trigger (or not) the intermediate poolCheckpoints.

```
    it('First Staker Rewards Calculation', async function () {

        let secureCheckpoints = false;
        let currentShare, currentStakedIntegral, balances;
        await this.ammgauge.poolCheckpoint();
        await ethers.provider.send("evm_increaseTime", [1 * 24 *

        const updateStates = async (from) => {
            currentShare = await this.ammgauge.perUserShare(from
            currentStakedIntegral = await this.ammgauge.perUserS
            balances = await this.ammgauge.balances(from.address
        }

        const stake = async (to, amount) => {
            await updateStates(to)
            console.log(" ")
            // Balance before
            let balanceBefore = await this.ammgauge.balances(to.
            // Stake
            await this.ammgauge.connect(to).stake(amount);
            expect(await this.ammgauge.balances(to.address)).to.
            await updateStates(to);
            console.log(" ")
        }

        const unstake = async (to, amount) => {
            await updateStates(to)
            console.log(" ")
            // Balance before
            let balanceBefore = await this.ammgauge.balances(to.
            // Stake
            await this.ammgauge.connect(to).unstake(amount);
            expect(await this.ammgauge.balances(to.address)).to.
            await updateStates(to);
            console.log(" ")
        }

        // Each user stakes tokens
        let initialStaking = ethers.utils.parseEther("10")
        console.log(" ")
        console.log("USERS STAKE");
        for (const user of users) {
```

```
        await stake(user, initialStaking)
        if(secureCheckpoints){await this.ammgauge.poolCheckpoint
        await ethers.provider.send("evm_increaseTime", [60 * 60]
        }
        console.log(" ")

        await ethers.provider.send("evm_increaseTime", [ 5 * 24
        if(secureCheckpoints){await this.ammgauge.poolCheckpoint

        let claimableRewards = [];
        let claimedRewards = [];
        console.log(" ")
        console.log("USERS CLAIMABLE REWARDS AFTER 5 days");
        console.log(" ")
        for (const user of users) {
            let stepClaimable = await this.ammgauge.claimableRew
            claimableRewards.push(ethers.utils.formatEther(stepC

            let rewardsClaim = await (await this.ammgauge.claimR
            claimedRewards.push(ethers.utils.formatEther(rewards
        }

        console.log("Claimable calculated")
        console.log("   ALICE - BOB -  CHARLIE - DAVID")
        console.log(claimableRewards)

        console.log(" ")
        console.log("Effectively Claimed")
        console.log("   ALICE - BOB -  CHARLIE - DAVID")
        console.log(claimableRewards)
    })
```

The outputs for both cases are shown on the following chart. The initial staking amount is 10eth amount of the DummyERC20 token.

	Without Checkpoints	With Checkpoints
Alice	6.6	115.5
Bob	111.9	111.9
Charlie	110.1	110.1
David	108.9	108.9

Recommended Mitigation Steps

- Check how is calculated the staking variables while the pool has no tokens staked and also how the updates and checkpoints are performed.

chase-manning (Backd) confirmed, but disagreed with severity and commented:

> This can only impact one user and only in an edge case so should be Medium severity.

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

> The warden has shown how the first depositor may end up not getting the correct amount of points due to how zero is handled in `poolCheckpoint`

> Am not fully confident this should be kept separate from M-02.

> However at this time, I believe the finding to be of Medium Severity.

Alex the Entreprenerd (judge) commented:

> At this time, while the underlying solution may be the same, I believe this finding and M-02 to be distinct.

[M-04] Amount distributed can be inaccurate when updating weights

Submitted by Picodes

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/tokenomics/Minter.sol#L220

https://github.com/code-423n4/2022-04-backd/blob/c856714a50437cb33240a5964b63687c9876275b/backd/contracts/tokenomics/InflationManager.sol#L559

https://github.com/code-423n4/2022-04-backd/blob/c856714a50437cb33240a5964b63687c9876275b/backd/contracts/to

kenomics/InflationManager.sol#L572

https://github.com/code-423n4/2022-04-backd/blob/c856714a50437cb33240a5964b63687c9876275b/backd/contracts/tokenomics/InflationManager.sol#L586

Impact

When updating pool inflation rates, other pools see their `currentRate` being modified without having `poolCheckpoint` called, which leads to false computations.

This will lead to either users losing a part of their claims, but can also lead to too many tokens could be distributed, preventing some users from claiming due to the `totalAvailableToNow` requirement in `Minter`.

Proof of Concept

Imagine you have 2 AMM pools A and B, both with an `ammPoolWeight` of 100, where `poolCheckpoint` has not been called for a moment. Then, imagine calling `executeAmmTokenWeight` to reduce the weight of A to 0.

Only A is checkpointed here, so when B will be checkpointed it will call `getAmmRateForToken`, which will see a pool weight of 100 and a total weight of 100 over the whole period since the last checkpoint of B, which is false, therefore it will distribute too many tokens. This is critical has the minter expects an exact or lower than expected distribution due to the requirement of `totalAvailableToNow`.

In the opposite direction, when increasing weights, it will lead to less tokens being distributed in some pools than planned, leading to a loss for users.

Mitigation Steps

Checkpoint every `LpStakerVault`, `KeeperGauge` or `AmmGauge` when updating the weights of one of them.

chase-manning (Backd) confirmed, but disagreed with severity and commented:

> We think this should be Medium as impact is quite minor.

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

> Because `getAmmRateForToken` uses the `totalAmmTokenWeight`, then all Gauges will need to be `poolCheckpoint`ed at the time the weight is changed-
>
> This is because for systems that accrue points over time, any time the multiplier changes, the growth of points has changed, and as such a new accrual needs to be performed.
>
> I believe there are 3 ways to judge this finding:
>
> - It's Medium as the Admin is using their privilege to change the points, potentially to their favour or to someones detriment.
> - It's High because the math is wrong and the code fails (by a way of lack of approximation) to properly allocate the resources.
> - It's Medium because while the finding is correct in a vacuum, anyone can call `poolCheckpoint`, as such if a true concern for fairness is made, anyone can front-run and back-run the update of weights with the goal of offering the most accurate math possible.
>
> Given those considerations, I want to emphasize that not calling `poolCheckpoint` can cause potentially drastic leaks of value, in a way similar to the original Sushi MasterChef's lack of `massUpdatePools` could.
>
> That said, I believe the finding is of medium severity as any individual could setup a bot to monitor and prevent this and it would require the weights to be changed by governance and impact can be quite minor as long as the pools are used.

[M-05] `BkdLocker#depositFees()` can be front run to steal the newly added rewardToken

Submitted by WatchPug

Every time the `BkdLocker#depositFees()` gets called, there will be a surge of rewards per locked token for the existing stakeholders.

This enables a well-known attack vector, in which the attacker will take a large portion of the shares before the surge, then claim the rewards and exit immediately.

While the `_WITHDRAW_DELAY` can be set longer to mitigate this issue in the current implementation, it is possible for the admin to configure it to a very short period of time or even `0`.

In which case, the attack will be very practical and effectively steal the major part of the newly added rewards.

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/BkdLocker.sol#L90-L100

```
    function depositFees(uint256 amount) external override {
        require(amount > 0, Error.INVALID_AMOUNT);
        require(totalLockedBoosted > 0, Error.NOT_ENOUGH_FUNDS);
        IERC20(rewardToken).safeTransferFrom(msg.sender, address(thi

        RewardTokenData storage curRewardTokenData = rewardTokenData

        curRewardTokenData.feeIntegral += amount.scaledDiv(totalLock
        curRewardTokenData.feeBalance += amount;
        emit FeesDeposited(amount);
    }
```

Proof of Concept

Given:

- Current `totalLockedBoosted()` is `100,000 govToken`;
- Pending distribution fees amount is `1,000 rewardToken`;
- `depositFees()` is called to add `1,000 rewardToken`;
- The attacker frontrun the 1st transaction with a `lock()` transaction to deposit `100,000 govToken`, taking 50% of the pool;
- After the transaction in step 1 is mined, the attacker calls `claimFees()` and received `500 rewardToken`.

As a result, the attacker has stolen half of the pending fees which belong to the old users.

Recommendation

Consider switching the reward to a `rewardRate` -based gradual release model, such as Synthetix's StakingRewards contract.

See: https://github.com/Synthetixio/synthetix/blob/develop/contracts/StakingRewards.sol#L113-L132

chase-manning (Backd) disputed and commented:

> The withdrawal delay prevents this attack.

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

> Given the need for:
>
> - Withdrawal delay set to 0 or small
> - Need for front-running which causes leak of value
>
> I believe High Severity to be out of the question.
>
> That said, because the rewards are given out at the time of `depositFees` , and they are not linearly vested, I do believe that a front-run MEV attack can be executed, being able to immediately claim the reward tokens, at the cost / risk of having to lock the tokens.
>
> The shorter the withdrawal delay, the lower the risk for the attack..
>
> Because this is contingent on configuration, and at best it's a loss of yield for the lockers, I believe Medium Severity to be more appropriat.
>
> As an additional note, please consider the fact that if the potential value gained is high enough, an attacker could just hedge the risk of locking by shorting the tokens, effectively being delta neutral while using the rewards for profit.

> This means that if your token becomes liquid enough (a goal for any protocol), you would expect the withdrawal delay to become ineffective as hedging options become available.

> Forcing the rewards to linearly vest will prevent the front-run from being effective and will reward long term lockers.

[M-06] `Minter.sol#_executeInflationRateUpdate()` `inflationManager().checkpointAllGauges()` is called after InflationRate is updated, causing users to lose rewards

Submitted by WatchPug

When `Minter.sol#_executeInflationRateUpdate()` is called, if an `_INFLATION_DECAY_PERIOD` has past since `lastInflationDecay`, it will update the InflationRate for all of the gauges.

However, in the current implementation, the rates will be updated first, followed by the rewards being settled using the new rates on the gauges using `inflationManager().checkpointAllGauges()`.

If the `_INFLATION_DECAY_PERIOD` has passed for a long time before `Minter.sol#executeInflationRateUpdate()` is called, the users may lose a significant amount of rewards.

On a side note, `totalAvailableToNow` is updated correctly.

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/tokenomics/Minter.sol#L187-L215

```
    function _executeInflationRateUpdate() internal returns (bool) {
        totalAvailableToNow += (currentTotalInflation * (block.times
        lastEvent = block.timestamp;
        if (block.timestamp >= lastInflationDecay + _INFLATION_DECAY_
            currentInflationAmountLp = currentInflationAmountLp.scal
            if (initialPeriodEnded) {
                currentInflationAmountKeeper = currentInflationAmoun
```

```
                annualInflationDecayKeeper
            );
            currentInflationAmountAmm = currentInflationAmountAm
                annualInflationDecayAmm
            );
        } else {
            currentInflationAmountKeeper =
                initialAnnualInflationRateKeeper /
                _INFLATION_DECAY_PERIOD;

            currentInflationAmountAmm = initialAnnualInflationRa
            initialPeriodEnded = true;
        }
        currentTotalInflation =
            currentInflationAmountLp +
            currentInflationAmountKeeper +
            currentInflationAmountAmm;
        controller.inflationManager().checkpointAllGauges();
        lastInflationDecay = block.timestamp;
    }
    return true;
}
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/tokenomics/InflationManager.sol#L110-L125

```
function checkpointAllGauges() external override returns (bool)
    uint256 length = _keeperGauges.length();
    for (uint256 i; i < length; i = i.uncheckedInc()) {
        IKeeperGauge(_keeperGauges.valueAt(i)).poolCheckpoint();
    }
    address[] memory stakerVaults = addressProvider.allStakerVau
    for (uint256 i; i < stakerVaults.length; i = i.uncheckedInc(
        IStakerVault(stakerVaults[i]).poolCheckpoint();
    }

    length = _ammGauges.length();
    for (uint256 i; i < length; i = i.uncheckedInc()) {
        IAmmGauge(_ammGauges.valueAt(i)).poolCheckpoint();
    }
    return true;
}
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/tokenomics/KeeperGauge.sol#L110-L117

```
    function poolCheckpoint() public override returns (bool) {
        if (killed) return false;
        uint256 timeElapsed = block.timestamp - uint256(lastUpdated)
        uint256 currentRate = IController(controller).inflationManag
        perPeriodTotalInflation[epoch] += currentRate * timeElapsed;
        lastUpdated = uint48(block.timestamp);
        return true;
    }
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/tokenomics/InflationManager.sol#L507-L519

```
    function getKeeperRateForPool(address pool) external view overri
        if (minter == address(0)) {
            return 0;
        }
        uint256 keeperInflationRate = Minter(minter).getKeeperInflat
        // After deactivation of weight based dist, KeeperGauge hand
        if (weightBasedKeeperDistributionDeactivated) return keeperI
        if (totalKeeperPoolWeight == 0) return 0;
        bytes32 key = _getKeeperGaugeKey(pool);
        uint256 poolInflationRate = (currentUInts256[key] * keeperIn
            totalKeeperPoolWeight;
        return poolInflationRate;
    }
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/tokenomics/Minter.sol#L173-L176

```
        function getKeeperInflationRate() external view override ret
            if (lastEvent == 0) return 0;
            return currentInflationAmountKeeper;
        }
```

Proof of Concept

Given:

- currentInflationAmountAmm: 12,000 Bkd (1000 per month)
- annualInflationDecayAmm: 50%
- initialPeriodEnded: true
- lastInflationDecay: 11 months ago
- _INFLATION*DECAY*PERIOD: 1 year
- Alice deposited as the one and only staker in the `AmmGauge` pool;
- 1 month later;
- `Minter.sol#_executeInflationRateUpdate()` is called;
- Alice `claimableRewards()` and received `500` Bkd tokens.

Expected Results:

- Alice to receive `1000` Bkd tokens as rewards.

Actual Results:

- Alice received `500` Bkd tokens as rewards.

Recommendation

Consider moving the call to `checkpointAllGauges()` to before the `currentInflationAmountKeeper` is updated.

```
    function _executeInflationRateUpdate() internal returns (bool) {
        totalAvailableToNow += (currentTotalInflation * (block.times
        lastEvent = block.timestamp;
        if (block.timestamp >= lastInflationDecay + _INFLATION_DECAY_
            controller.inflationManager().checkpointAllGauges();
            currentInflationAmountLp = currentInflationAmountLp.scal
            if (initialPeriodEnded) {
                currentInflationAmountKeeper = currentInflationAmoun
                    annualInflationDecayKeeper
                );
                currentInflationAmountAmm = currentInflationAmountAm
```

```
                annualInflationDecayAmm
            );
        } else {
            currentInflationAmountKeeper =
                initialAnnualInflationRateKeeper /
                _INFLATION_DECAY_PERIOD;

            currentInflationAmountAmm = initialAnnualInflationRa
            initialPeriodEnded = true;
        }
        currentTotalInflation =
            currentInflationAmountLp +
            currentInflationAmountKeeper +
            currentInflationAmountAmm;
        lastInflationDecay = block.timestamp;
    }
    return true;
}
```

chase-manning (Backd) confirmed, but disagreed with severity and commented:

> This should be medium risk, as should only have a minor impact on users getting less rewards and no over-minting can occur.

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

> The warden has shown how, due to an incorrect order of operations, rates for new rewards can be enacted before the old rewards are distributed, causing a loss of rewards for end users.

> There are 2 ways to judge this finding:
>
> - The system is failing at distributing the proper amount of rewards, hence the finding is of High Severity
> - End users can risk a loss of value if these functions are not called often enough as the difference between the old rates and the new rates will cause a loss of reward for the end users.
>
> Ultimately the impact is a loss of value, further minimized by the fact that anyone can call `poolCheckpoint` as well as `executeInflationRateUpdate` meaning the losses can be minimized.

> So from a coding standpoint I believe the bug should be fixed before deployment, but from a impact point of view the impact can be minimized to make “loss of yield” mostly rounding errors.

> Given the impact I believe the finding to be of Medium Severity.

[M-07] FeeBurner initiates swap without any slippage checks if Chainlink oracle fails

Submitted by Ruhum

https://github.com/code-423n4/2022-05-backd/blob/main/protocol/contracts/tokenomics/FeeBurner.sol#L43-L88

https://github.com/code-423n4/2022-05-backd/blob/main/protocol/contracts/swappers/SwapperRouter.sol#L414-L425

https://github.com/code-423n4/2022-05-backd/blob/main/protocol/contracts/swappers/SwapperRouter.sol#L439

Impact

While the SwapperRouter contract isn't explicitly in scope, it's a dependency of the FeeBurner contract which *is* in scope. So I think it's valid to make this submission.

The SwapperRouter contract uses the chainlink oracle to compute the minimum amount of tokens it should expect from the swap. The value is then used for the slippage check. But, if the chainlink oracle fails, for whatever reason, the contract uses `0` for the slippage check instead. Thus there's a scenario where swaps initiated by the FeeBurner contract can be sandwiched.

Proof of Concept

1. multiple swaps initiated through `FeeBurner.burnToTarget()`
2. SwapperRouter calls `_minTokenAmountOut()` to determine `min_out` parameter.
3. `minTokenAmountOut()` returns `0` when Chainlink oracle fails

Recommended Mitigation Steps

Either revert the transaction or initiate the transaction with a default slippage of 99%. In the case of Curve, you can get the expected amount through `get_dy()` and then multiply the value by 0.99. Use that as the `min_out` value and you don't have to worry about chainlink

chase-manning (Backd) disputed and commented:

> This is intended functionality. If there is no oracle for a token, we still want to swap it, even if this presents a possible sandwich attack. It should be rare for a token to not have an oracle, and when it does we would rather accept slippage as opposed to not being able to swap it at all.

Alex the Entreprenerd (judge) commented:

> I acknowledge the sponsor reply that they want to offer a service to the end user in allowing any swappable token to be used.

> While I believe the intent of the sponsor is respectable, the reality of the code is that it indeed allows for price manipulation and extraction of value, personally I would recommend end users to perform their own swaps to ensure a more reliable outcome.

> That said, because the code can be subject to leak of value, I believe Medium Severity to be appropriate.

[M-08] Users can claim extremely large rewards or lock rewards from LpGauge due to uninitialised `poolLastUpdate` variable

Submitted by scaraven

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/tokenomics/LpGauge.sol#L115-L119

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/

StakerVault.sol#L326-L328

Impact

A user can claim all of the available governance tokens or prevent any rewards from being claimed in `LpGauge.sol` if sufficient time is left between deploying the contract and initialising it in the `StakerVault.sol` contract by calling `initalizeLPGauge()` OR if a new `LPGauge` contract is deployed and added to `StakerVault` using `prepareLPGauge`.

Inside `LPGauge.sol` when calling `_poolCheckPoint()`, the `lastUpdated` variable is not initalised so defaults to a value of `0`, therefore if the user has managed to stake tokens in the `StakerVault` then the calculated `poolStakedIntegral` will be very large (as block.timestamp is very large). Therefore a user can mint most current available governance tokens for themselves when they claim their rewards (or prevent any governance tokens from being claimed).

Proof of Concept

1. LP Gauge and StakerVault contracts are deployed
2. Before the `initializeLpGauge()`, user A will stake 1 token with `stakeFor()` thereby increasing `_poolTotalStaked` by 1. As the `lpgauge` address is equal to the zero address, `_userCheckPoint()` will not be called and `poolLastUpdate` will remain at 0.
3. The user can then directly call `_userCheckPoint()` and be allocated a very large number of shares. This works because `poolLastUpdate` is 0 but the staked amount in the vault is larger than 0
4. Once `initializeLPGauge()` is called, the user can then call `claimRewards()` and receive a very large portion of tokens or if `poolStakedIntegral` exceeds the mint limit set by `Minter.sol` then no one else can claim governance tokens from the lpGauge.

OR

5. A new LP Gauge contract is deployed and added to the vault using `prepareGauge()`. Follow steps 2 to 4.

Recommended Mitigation Steps

Initialise `poolLastUpdate` in the constructor

danhper (Backd) confirmed

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

> The warden has identified a way to exploit the uninitialized value for the variable `poolLastUpdate`
>
> Because this is contingent on setup, and the impact is theft of rewards, I believe Medium Severity to be more appropriate.

scaraven (warden) commented:

> Hi, correct me if I am wrong but there is a second scenario where this situation can be exploited as outlined in my report. If governance deploy a new LPGauge they must do it through `prepareLPGauge()` as `initializeLpGauge()` will revert. This introduces a significant delay (3 days if I'm correct) before the LPGauge can actually be initialised with `executeLPGauge()`. Therefore a user can easily monitor this contract and call `poolCheckpoint()` as soon as soon as `prepareLPGauge()` is called and the contract has no way to prevent this.

Alex the Entreprenerd (judge) commented:

> @scaraven
>
> I don't see any enforced delay in `initializeLpGauge`

```
ftrace | funcSig
function _setConfig(bytes32 key↑, address value↑) internal returns (address) {
    emit ConfigUpdatedAddress(key↑, currentAddresses[key↑], value↑);
    currentAddresses[key↑] = value↑;
    pendingAddresses[key↑] = address(0);
    deadlines[key↑] = 0;
    return value↑;
}

ftrace | funcSig
function _setConfig(bytes32 key↑, uint256 value↑) internal returns (uint256) {
    uint256 oldValue = currentUInts256[key↑];
    currentUInts256[key↑] = value↑;
    pendingUInts256[key↑] = 0;
    deadlines[key↑] = 0;
    emit ConfigUpdatedNumber(key↑, oldValue, value↑);
    return value↑;
}
```

> `_setConfig` bypasses the need for a `MIN_DELAY`

> Meaning the exploit is not practical for the first gauge

> For Gauge that needs to be substituted, that will indeed require a 3 day wait period, `_poolCheckpoint` can only be effective after the first deposit has happened as well, see: https://github.com/code-423n4/2022-05-backd-findings/issues/100 Which effects how initial rewards are calculated.

> Due to the math being off it may be necessary to influence `uint256 currentRate = inflationManager.getLpRateForStakerVault(address(stakerVault));`

> This, the need for a second gauge, and the race condition between the first depositor and other depositors are external conditions, leading me to believe that Medium Severity is appropriate

scaraven (warden) commented:

> Cool, thanks for taking the time to explain

[M-09] BkdLocker depositFees can be blocked

Submitted by csanuragjain

burnFees will fail if none of the pool tokens have underlying token as native ETH token. This is shown below. Since burnFees fails so no fees is deposited in BKDLocker.

Proof of Concept

1. Assume RewardHandler.sol has currently amount 5 as address(this).balance (ethBalance) (even attacker can send a small balance to this contract to do this dos attack)
2. None of the pools have underlying as address(0) so no ETH tokens and only ERC20 tokens are present
3. Now feeBurner.burnToTarget is called passing current ETH balance of amount 5 with all pool tokens
4. feeBurner loops through all tokens and swap them to WETH. Since none of the token is ETH so burningEth_ variable is false
5. Now the below require condition fails since burningEth_ is false

```
require(burningEth_ || msg.value == 0, Error.INVALID_VALUE);
```

6. This fails the burnFees function.

Recommended Mitigation Steps

ETH should not be sent if none of pool underlying token is ETH. Change it to something like below:

```
bool ethFound=false;
for (uint256 i; i < pools.length; i = i.uncheckedInc()) {
            ILiquidityPool pool = ILiquidityPool(pools[i]);
            address underlying = pool.getUnderlying();
            if (underlying != address(0)) {
                _approve(underlying, address(feeBurner));
            } else
{
ethFound=true;
}
            tokens[i] = underlying;
        }

if(ethFound){
```

```
            feeBurner.burnToTarget{value: ethBalance}(tokens, target
    } else {
    feeBurner.burnToTarget(tokens, targetLpToken);
    }
```

samwerner (Backd) confirmed

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

> The warden has shown how, due to a flaw in the logic, if ETH is present in the contract and no pool is denominated in ETH, then the contract will revert.
>
> This can be done as a DOS attack or for griefing.
>
> However, remediation would simply require adding a pool denominated in ETH, to ensure that the logic goes through
>
> For this reason, I believe Medium Severity to be more appropriate

[M-10] There are multiple ways for admins/governance to rug users

Submitted by lllllll, also found by 0x1f8b

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/zaps/PoolMigrationZap.sol#L61

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/BkdLocker.sol#L70-L75

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/RewardHandler.sol#L50

Impact

A malicious admin can steal user funds or lock their balances forever.

Even if the user is benevolent the fact that there is a rug vector available may negatively impact the protocol's reputation.

Proof of Concept

Unlike the original Convex code that goes to great lengths to prevent users having the ability to transfer funds/mint things, this project introduces multiple roles and new abilities that require users to place more trust in governance:

1. Admins can initiate migrations and set the `newPool_` to be a contract that forwards funds to accounts they control

```
File: protocol/contracts/zaps/PoolMigrationZap.sol   #1

61            ILiquidityPool newPool_ = _underlyingNewPools[under
62            uint256 ethValue_ = underlying_ == address(0) ? und
63            newPool_.depositFor{value: ethValue_}(msg.sender, u
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/zaps/PoolMigrationZap.sol#L61-L63

2. Admins can add infinite `newRewardToken`s:

```
File: protocol/contracts/BkdLocker.sol   #2

70        function migrate(address newRewardToken) external overr
71            _replacedRewardTokens.remove(newRewardToken);
72            _replacedRewardTokens.set(rewardToken, block.timest
73            lastMigrationEvent = block.timestamp;
74            rewardToken = newRewardToken;
75        }
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/BkdLocker.sol#L70-L75

so that `_userCheckpoint()` s, which are required to withdraw funds, revert because they run out of gas iterating over the tokens:

```
File: protocol/contracts/BkdLocker.sol   #3

292       function _userCheckpoint(
293           address user,
294           uint256 amountAdded,
295           uint256 newTotal
296       ) internal {
297           RewardTokenData storage curRewardTokenData = rewar
298
299           // Compute the share earned by the user since they
300           uint256 userBalance = balances[user];
301           if (userBalance > 0) {
302               curRewardTokenData.userShares[user] += (curRew
303                   curRewardTokenData.userFeeIntegrals[user])
304                       userBalance.scaledMul(boostFactors[use
305                   );
306
307               // Update values for previous rewardTokens
308               if (lastUpdated[user] < lastMigrationEvent) {
309                   uint256 length = _replacedRewardTokens.len
310                   for (uint256 i; i < length; i = i.unchecke
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/BkdLocker.sol#L292-L310

3. Admins can set a malicious `feeBurner` , via the addressProvider, which just takes the fees for itself

```
File: protocol/contracts/RewardHandler.sol   #4

35        function burnFees() external override {
36            IBkdLocker bkdLocker = IBkdLocker(addressProvider.g
37            IFeeBurner feeBurner = addressProvider.getFeeBurner
38            address targetLpToken = bkdLocker.rewardToken();
39            address[] memory pools = addressProvider.allPools()
40            uint256 ethBalance = address(this).balance;
41            address[] memory tokens = new address[](pools.lengt
42            for (uint256 i; i < pools.length; i = i.uncheckedIn
```

```
43                ILiquidityPool pool = ILiquidityPool(pools[i]);
44                address underlying = pool.getUnderlying();
45                if (underlying != address(0)) {
46                    _approve(underlying, address(feeBurner));
47                }
48                tokens[i] = underlying;
49            }
50            feeBurner.burnToTarget{value: ethBalance}(tokens, t
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/RewardHandler.sol#L35-L50

4. Admins can set an oracle that provides the wrong answers:

```
File: protocol/contracts/swappers/SwapperRouter.sol   #5

452             try _addressProvider.getOracleProvider().getPriceE
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/swappers/SwapperRouter.sol#L452

Recommended Mitigation Steps

The trust-minimizing approach that Convex took was to not allow admins to change addresses. In order for things to change, an admin is allowed to completely shut everything down, and during the shut down state, users are still able to withdraw their funds. Later, the admins spin up a whole new set of contracts, and let users migrate things themselves. Something similar can be done here by having the DAO accept proposals to spawn specific contracts, and hook up specific addresses in certain ways in the new deployment.

danhper (Backd) disputed and commented:

> This is a design decision. Many protocols are fully upgradeable (Compound, Aave, Maker) and some decide to be fully immutable (Convex, Curve). The governance process will be formalized a little later and have safeguards in place but we are not

> planning on following Convex's approach nor think that it is the "correct" way to design a protocol.

Alex the Entreprenerd (judge) commented:

> The warden has listed multiple ways in which Admin Privilege can be used against users of the protocol.

> While the sponsor may be fully aware of these mechanics, it is very important that these type of risks are clearly explained to end-users as they can ultimately bear the consequences of those risks.

> Because the findings are contingent on a malicious admin, I believe Medium Severity to be appropriate.

[M-11] Usage of deprecated transfer to send ETH

Submitted by peritoflores, also found by JC and StyxRave

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/swappers/SwapperRouter.sol#L140

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/swappers/SwapperRouter.sol#L280

Impact

Usage of deprecated transfer Swap can revert.

Proof of Concept

The original `transfer` used to send eth uses a fixed stipend 2300 gas. This was used to prevent reentrancy. However this limit your protocol to interact with others contracts that need more than that to process the transaction.

A good article about that: https://consensys.net/diligence/blog/2019/09/stop-using-soliditys-transfer-now/.

Recommended Mitigation Steps

Used call instead. For example

```
        (bool success, ) = msg.sender.call{amount}("");
        require(success, "Transfer failed.");
```

chase-manning (Backd) confirmed

Alex the Entreprenerd (judge) commented:

> While submission is lazy in that it doesn't show the ways in which it could revert, (for example most of the times even a transfer to a gnosis-safe will not revert as the gas stipend is sufficient)

> It's true that `transfer`s gas stipend may run out, causing reverts

> For this reason I agree with Med Severity.

[M-12] Users can claim more fees than expected if governance migrates current rewardToken again by fault.

Submitted by hansfriese, also found by csanuragjain and unforgiven

https://github.com/code-423n4/2022-05-backd/tree/main/protocol/contracts/BkdLocker.sol#L70-L75

https://github.com/code-423n4/2022-05-backd/tree/main/protocol/contracts/BkdLocker.sol#L302-L322

Impact

Users can claim more fees than expected if governance migrates current rewardToken again by fault.

Proof of Concept

In the migrate() function, there is no requirement newRewardToken != rewardToken. If this function is called with the same “rewardToken” parameter, ”_replacedRewardTokens” will contain the current “rewardToken” also. Then when the user claims fees, “userShares” will be added two times for the same token at L302-L305, L314-L317.

It’s because “curRewardTokenData.userFeeIntegrals[user]” is updated at L332 after the “userShares” calculation for past rewardTokens. So the user can get paid more fees than he should.

Tools Used

Solidity Visual Developer of VSCode

Recommended Mitigation Steps

You need to add this require() at L71.

require(newRewardToken != rewardToken, Error.SAME*AS*CURRENT);

danhper (Backd) confirmed

Alex the Entreprenerd (judge) commented:

> The warden has identified how a governance migration from and to the same token can cause the rewards to be double-counted.
>
> Because the exploit is contingent on:
>
> - Admin Privilege
> - Would cause issues with Yield
>
> I believe Medium Severity to be appropriate.

[M-13] Inconsistency in view functions can lead to users believing they’re due for more BKD rewards

Submitted by SmartSek

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/tokenomics/AmmConvexGauge.sol#L107-L111

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/tokenomics/AmmConvexGauge.sol#L129-L134

Impact

The view functions used for a user to check their claimable rewards vary in their implementation. This can cause users to believe they are due X amount but will receive Y.

Proof of Concept

If the `inflationRecipient` is set, then `poolStakedIntegral` will be incremented in `claimableRewards()` but not in any other function like `allClaimableRewards()` or `poolCheckpoint()`.

If a user calls `claimableRewards()` after the `inflationRepient` has been set, `claimableRewards()` will return a larger value than `allClaimableRewards()` or the amount actually returned by `claimRewards()`.

Recommended Mitigation Steps

To make the logic consistent, `claimableRewards()` needs `if (inflationRecipient == address(0))` added to it.

chase-manning (Backd) confirmed

Alex the Entreprenerd (judge) commented:

> The warden has identified an incorrect implementation in how rewards are shown, while no assets are at risk, the issue identified is a programming mistake that could cause further issues.

> For this reason I agree with Medium Severity

> I'm still fairly conflicted on severity as the impact is going to be quite small, however at this time, because the "code is wrong", I still think Medium Severity to be valid.

[M-14] StakerVault.unstake(), StakerVault.unstakeFor() would revert with a uint underflow error of StakerVault.strategiesTotalStaked, StakerVault._poolTotalStaked.

Submitted by hansfriese

https://github.com/code-423n4/2022-05-backd/tree/main/protocol/contracts/StakerVault.sol#L98-L102

https://github.com/code-423n4/2022-05-backd/tree/main/protocol/contracts/StakerVault.sol#L342-L346

https://github.com/code-423n4/2022-05-backd/tree/main/protocol/contracts/StakerVault.sol#L391-L395

Impact

StakerVault.unstake(), StakerVault.unstakeFor() would revert with a uint underflow error of StakerVault.strategiesTotalStaked, StakerVault._poolTotalStaked.

Proof of Concept

Currently it saves totalStaked for strategies and non-strategies separately.

uint underflow error could occur in these cases.

Scenario 1.

1. Address A(non-strategy) stakes some amount x and it will be added to StakerVault_poolTotalStaked.
2. This address A is approved as a strategy by StakerVault.inflationManager.
3. Address A tries to unstake amount x, it will be deducted from StakerVault.strategiesTotalStaked because this address is a strategy already.

Even if it would succeed for this strategy but it will revert for other strategies because StakerVault.strategiesTotalStaked is less than correct staked amount for strategies.

Scenario 2. There is a transfer between strategy and non-strategy using StakerVault.transfer(), StakerVault.transferFrom() functions. In this case, StakerVault.strategiesTotalStaked and StakerVault._poolTotalStaked must be changed accordingly but there is no such logic.

Tools Used

Solidity Visual Developer of VSCode

Recommended Mitigation Steps

You need to modify 3 functions. StakerVault.addStrategy(), StakerVault.transfer(), StakerVault.transferFrom().

1. You need to move staked amount from StakerVault._poolTotalStaked to StakerVault.strategiesTotalStaked every time when StakerVault.inflationManager approves a new strategy.

   You can modify addStrategy() at L98-L102 like this.

   function addStrategy(address strategy) external override returns (bool) { require(msg.sender == address(inflationManager), Error.UNAUTHORIZED*ACCESS); require(!strategies[strategy], Error.ADDRESS*ALREADY_SET);

   strategies[strategy] = true; _poolTotalStaked -= balances[strategy]; strategiesTotalStaked += balances[strategy];

   return true; }

2. You need to add below code at L126 of transfer() function.

   if(strategies[msg.sender] != strategies[account]) { if(strategies[msg.sender]) { // from strategy to non-strategy _poolTotalStaked += amount; strategiesTotalStaked -= amount; } else { // from non-strategy to strategy _poolTotalStaked -= amount; strategiesTotalStaked += amount; } }

3. You need to add below code at L170 of transferFrom() function.

    if(strategies[src] != strategies[dst]) { if(strategies[src]) { // from strategy to non-strategy _poolTotalStaked += amount; strategiesTotalStaked -= amount; } else { // from non-strategy to strategy _poolTotalStaked -= amount; strategiesTotalStaked += amount; } }

danhper (Backd) confirmed

Alex the Entreprenerd (judge) commented:

> The warden has identified a way for funds to be stuck due to underflow.
>
> While the odds of this happening are fairly low, the grief can be executed by frontrunning the `addStrategy` call, as well as by mistake.
>
> Additionally, a strategy doesn't seem to be removable making the loss of those hypothetical tokens permanent.
>
> For those reasons I believe Medium Severity to be appropriate.

[M-15] Potential DoS when removing keeper gauge

Submitted by shenwilly

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/tokenomics/InflationManager.sol#L609-L618

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/tokenomics/KeeperGauge.sol#L82

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/actions/topup/TopUpActionFeeHandler.sol#L95-L98

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/actions/topup/TopUpAction.sol#L807

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/actions/topup/TopUpAction.sol#L653

Impact

When `_removeKeeperGauge` is called, there is no guarantee that the keeper gauge isn't currently in use by any `TopUpActionFeeHandler`. If it's still in use, any top up action executions will be disabled as reporting fees in `KeeperGauge.sol` will revert:

```
function reportFees(
    address beneficiary,
    uint256 amount,
    address lpTokenAddress
) external override returns (bool) {
    ...
    require(!killed, Error.CONTRACT_PAUSED); // gauge is killed
    ...
    return true;
}
```

If this happened during extreme market movements, some positions that require a top up will not be executed and be in risk of being liquidated.

Proof of Concept

- Alice registers a top up action.
- Governance calls `InflationManager.removeKeeperGauge`, replacing an old keeper gauge. However, governance forgot to call `TopUpActionFeeHandler.prepareKeeperGauge` so `TopUpActionFeeHandler.getKeeperGauge` still points to the killed gauge.
- Market moved and Alice's position should now be executed by keepers, however any attempt to execute will revert: > Keeper calls TopUpAction.execute(); > _payFees(); > IActionFeeHandler(feeHandler).payFees(); > IKeeperGauge(keeperGauge).reportFees(); > reverts as gauge is already killed

- Governance noticed and calls `prepareKeeperGauge` with a 3 days delay.
- Alice's position got liquidated before the change is executed.

Recommended Mitigation Steps

Consider adding an on-chain check to ensure that the keeper gauge is not in use before removing them.

danhper (Backd) confirmed

Alex the Entreprenerd (judge) commented:

> I believe the warden has shown a situation in which calling `_removeKeeperGauge` can cause `payFees` to revert, making it impossible (for a time) for fees to be paid.

> I do not believe the impact will extend beyond:
>
> - Potential loss of yield (or delay)
> - Need for governance to set a new gauge

> I disagree with the statement: `Alice's position got liquidated before the change is executed.` as no liquidation should be contingent on fees being paid from this function.

> Because the finding shows how the system for fees can be stopped due to external conditions, I agree with Medium Seveirty

[M-16] it's possible to initialize contract BkdLocker for multiple times by sending startBoost=0 and each time different values for other parameters

Submitted by unforgiven, also found by 0x1f8b and scaraven

function `initialize()` of `BkdLocker` suppose to be called one time and contract initialize one time. but if it's called by `startBoost=0` then it's possible to call it again with different values for other parameters. there are some logics based on the values function `initilize()` sets which is in calculating boost and withdraw delay. by

initializing multiple times different users get different values for those logics and because rewards are distributed based on boosts so those logics will be wrong too.

Proof of Concept

This is `initiliaze()` code in `BkdLocker`:

```
function initialize(
    uint256 startBoost,
    uint256 maxBoost,
    uint256 increasePeriod,
    uint256 withdrawDelay
) external override onlyGovernance {
    require(currentUInts256[_START_BOOST] == 0, Error.CONTRA
    _setConfig(_START_BOOST, startBoost);
    _setConfig(_MAX_BOOST, maxBoost);
    _setConfig(_INCREASE_PERIOD, increasePeriod);
    _setConfig(_WITHDRAW_DELAY, withdrawDelay);
}
```

As you can see it checks the initialization statue by `currentUInts256[_START_BOOST]` 's value but it's not correct way to do and initializer can set `currentUInts256[_START_BOOST]` value as `0` and set other parameters values and call this function multiple times with different values for `_MAX_BOOST` and `_INCREASE_PERIOD` and `_WITHDRAW_DELAY`. setting different values for those parameters can cause different calculation in `computeNewBoost()` and `prepareUnlock()`. function `computeNewBoost()` is used to calculate users boost parameters which is used on reward distribution. so by changing `_MAX_BOOST` the rewards will be distributed wrongly between old users and new users.

Tools Used

VIM

Recommended Mitigation Steps

Add some other variable to check the status of initialization of contract.

danhper (Backd) confirmed, but disagreed with severity

Alex the Entreprenerd (judge) commented:

> The warden has shown how, under specific circumstances, the `BkdLocker` contact can be initialized multiple times, with the specific goal of changing configuration parameters.

> From my understanding these configs are meant to be set only once (there are no available external setters that governance can call), effectively sidestepping the "perceived immutability" that the locker seems to be offering.

> The attack is contingent on malicious Governance, for that reason I believe Medium Severity to be appropriate.

> The impact of the attack can cause:
>
> - Loss of Yield
> - Unfair distribution of rewards
> - Abuse of rewards math for governance advantage

> End users can verify that the exploit is not applicable by ensuring that `startBoost` is greater than 0.

[M-17] Strategy in StakerVault.sol can steal more rewards even though it's designed strategies shouldn't get rewards.

Submitted by hansfriese

https://github.com/code-423n4/2022-05-backd/tree/main/protocol/contracts/StakerVault.sol#L95

https://github.com/code-423n4/2022-05-backd/tree/main/protocol/contracts/tokenomics/LpGauge.sol#L52-L63

Impact

Strategy in StakerVault.sol can steal more rewards even though it's designed strategies shouldn't get rewards.

Also there will be a problem with a rewarding system in LpGauge.sol so that some normal users wouldn't get rewards properly.

Proof of Concept

1. Strategy A staked amount x and x will be added to StakerVault.strategiesTotalStaked.

   contracts\StakerVault.sol#L312

2. Strategy A transferred the amount x to non-strategy B and StakerVault.strategiesTotalStaked, StakerVault. _poolTotalStaked won't be updated. contracts\StakerVault.sol#L111
3. After some time for the larger LpGauge.poolStakedIntegral, B claims rewards using the LpGauge.claimRewards() function. contracts\tokenomics\LpGauge.sol#L52

   Inside LpGauge.userCheckPoint(), it's designed not to calculate LpGauge.perUserShare for strategy, but it will pass this condition because B is not a strategy. contracts\tokenomics\LpGauge.sol#L90

   Furthermore, when calculate rewards, LpGauge.poolStakedIntegral will be calculated larger than a normal user stakes same amount. It's because StakerVault._poolTotalStaked wasn't updated when A transfers x amount to B so LpGauge.poolTotalStaked is less than correct value. contracts\tokenomics\LpGauge.sol#L113-L117

   Finally B can get more rewards than he should and the reward system will pay more rewards than it's designed.

Tools Used

Solidity Visual Developer of VSCode

Recommended Mitigation Steps

I think there will be two methods to fix.

Method 1 is to forbid a transfer between strategy and non-strategy so that strategy can't move funds to non-strategy.

Method 2 is to update StakerVault.strategiesTotalStaked and StakerVault._poolTotalStaked correctly so that strategy won't claim more rewards than he should even though he claims rewards using non-strategy.

Method 1. You need to modify two functions. StakerVault.transfer(), StakerVault.transferFrom().

1. You need to add this require() at L112 for transfer().

   require(strategies[msg.sender] == strategies[account], Error.FAILED_TRANSFER);

2. You need to add this require() at L144 for transferFrom().

   require(strategies[src] == strategies[dst], Error.FAILED_TRANSFER);

Method 2. I've explained about this method in my medium risk report "StakerVault.unstake(), StakerVault.unstakeFor() would revert with a uint underflow error of StakerVault.strategiesTotalStaked, StakerVault._poolTotalStaked" I will copy the same code for your convenience.

You need to modify 3 functions. StakerVault.addStrategy(), StakerVault.transfer(), StakerVault.transferFrom().

1. You need to move staked amount from StakerVault._poolTotalStaked to StakerVault.strategiesTotalStaked every time when StakerVault.inflationManager approves a new strategy.

   You can modify addStrategy() at L98-L102 like this.

   function addStrategy(address strategy) external override returns (bool) { require(msg.sender == address(inflationManager), Error.UNAUTHORIZED*ACCESS); require(!strategies[strategy], Error.ADDRESS*ALREADY_SET);

```
strategies\[strategy] = true;
\_poolTotalStaked -= balances\[strategy];
strategiesTotalStaked += balances\[strategy];

return true;
```

}

2. You need to add below code at L126 of transfer() function.

   if(strategies[msg.sender] != strategies[account]) { if(strategies[msg.sender]) { // from strategy to non-strategy _poolTotalStaked += amount; strategiesTotalStaked -= amount; } else { // from non-strategy to strategy _poolTotalStaked -= amount; strategiesTotalStaked += amount; } }

3. You need to add below code at L170 of transferFrom() function.

   if(strategies[src] != strategies[dst]) { if(strategies[src]) { // from strategy to non-strategy _poolTotalStaked += amount; strategiesTotalStaked -= amount; } else { // from non-strategy to strategy _poolTotalStaked -= amount; strategiesTotalStaked += amount; } }

chase-manning (Backd) confirmed, but disagreed with severity and commented:

> We think this is a medium risk.

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

> I believe there's validity to the finding but at the same time I believe the impact is a loss of yield more so than an unfair gain of yield for a strategy.
>
> Specifically the POC is reliant on Depositing as a user, then transferring tokens to a strategy.
>
> I believe this will break the accounting per the POC shown (`strategiesTotalStaked` will be incorrect).
>
> Then the rewards will be claimable to the strategy as if it were a user, meaning that the extra checks to prevent strategies from gaining staking rewards will be sidestepped.
>
> I believe those tokens will be lost unless all strategies have a way to sweep non-protected tokens.
>
> Because the warden showed how to break accounting, I believe Medium Severity to be valid, that said I don't believe the warden has shown any meaningful economic

> attack beside end-users losing their own tokens and the rewards attached to them.

[M-18] Fees from delisted pool still in reward handler will become stuck after delisting

Submitted by 0x52

Unclaimed fees from pool will be stuck.

Proof of Concept

When delisting a pool the pool's reference is removed from address provider:

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/Controller.sol#L63

Burning fees calls a dynamic list of all pools which no longer contains the delisted pool:

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/RewardHandler.sol#L39

Since the list no longer contains the pool those fees will not be processed and will remain stuck in the contract

Recommended Mitigation Steps

Call burnFees() before delisting a pool.

danhper (Backd) confirmed

Alex the Entreprenerd (judge) commented:

> The warden has shown how, by removing a pool before calling `burnFees`, the removed pool will not receive the portion of fees that it should.

> Because this finding related to loss of yield, I believe Medium Severity to be appropriate.

Low Risk and Non-Critical Issues

For this contest, 42 reports were submitted by wardens detailing low risk and non-critical issues. The **report highlighted below** by **IIIIIII** received the top score from the judge.

The following wardens also submitted reports: SmartSek, BowTiedWardens, SecureZeroX, berndartmueller, cccz, Ruhum, cryptphi, defsec, dipp, fatherOfBlocks, unforgiven, codexploder, Chom, bardamu, masterchief, Kumpa, 0x29A, c3phas, Funen, hansfriese, Picodes, shenwilly, WatchPug, 0xNazgul, 0x1f8b, Sm4rty, 0xf15ers, asutorufos, delfin454000, gzeon, hake, sach1r0, simon135, StyxRave, oyc_109, hyh, catchup, Kaiziron, MiloTruck, sashik_eth, *and* Waze.

Low Risk Issues

	Issue	Instances
1	`migrate()` still does transfers when the transfer is to the same pool, and this can be done multiple times	1
2	Non-exploitable reentrancy	1
3	Users can DOS themselves by executing `prepareUnlock(0)` many times	1
4	Unused/empty `receive()` / `fallback()` function	3
5	`safeApprove()` is deprecated	4
6	Missing checks for `address(0x0)` when assigning values to `address` state variables	8
7	`_prepareDeadline()`, `_setConfig()`, and `_executeDeadline()` should be `private`	1

Total: 19 instances over 7 issues

[L-01] `migrate()` still does transfers when the transfer is to the same pool, and this can be done multiple times

There's no check that the old address isn't the same as the new address, and there's no check that the migration has already happened

There is 1 instance of this issue:

```
File: protocol/contracts/zaps/PoolMigrationZap.sol   #1

52        function migrate(address oldPoolAddress_) public overri
53            ILiquidityPool oldPool_ = ILiquidityPool(oldPoolAdd
54            IERC20 lpToken_ = IERC20(oldPool_.getLpToken());
55            uint256 lpTokenAmount_ = lpToken_.balanceOf(msg.sen
56            require(lpTokenAmount_ != 0, "No LP Tokens");
57            require(oldPool_.getWithdrawalFee(msg.sender, lpTok
58:           lpToken_.safeTransferFrom(msg.sender, address(this)
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/zaps/PoolMigrationZap.sol#L52-L58

[L-02] Non-exploitable reentrancy

There is no reentrancy guard in this function, and if used with a token that has transfer callbacks, such as an ERC777, the caller can reenter before `balances` is updated. I don't currently see a way to exploit this

There is 1 instance of this issue:

```
File: protocol/contracts/tokenomics/AmmGauge.sol   #1

130           uint256 oldBal = IERC20(ammToken).balanceOf(address
131           IERC20(ammToken).safeTransfer(dst, amount);
132           uint256 newBal = IERC20(ammToken).balanceOf(address
133           uint256 unstaked = oldBal - newBal;
134           balances[msg.sender] -= unstaked;
135:          totalStaked -= unstaked;
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/tokenomics/AmmGauge.sol#L130-L135

[L-03] Users can DOS themselves by executing `prepareUnlock(0)` many times

There's no check on the amount, and every call add another entry to an array. When the user finally calls `executeUnlocks()` they'll run out of gas

There is 1 instance of this issue:

```
File: protocol/contracts/BkdLocker.sol   #1

118      function prepareUnlock(uint256 amount) external overrid
119          require(
120              totalStashed[msg.sender] + amount <= balances[m
121              "Amount exceeds locked balance"
122:         );
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/BkdLocker.sol#L118-L122

[L-04] Unused/empty `receive()` / `fallback()` function

If the intention is for the Ether to be used, the function should call another function, otherwise it should revert (e.g. `require(msg.sender == address(weth))`)

There are 3 instances of this issue:

```
File: protocol/contracts/zaps/PoolMigrationZap.sol   #1

31:       receive() external payable {}
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/zaps/PoolMigrationZap.sol#L31

```
File: protocol/contracts/RewardHandler.sol   #2
```

```
30:         receive() external payable {}
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/RewardHandler.sol#L30

```
File: protocol/contracts/tokenomics/FeeBurner.sol    #3

35:         receive() external payable {} // Recieve function for
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/tokenomics/FeeBurner.sol#L35

[L-05] `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 4 instances of this issue:

```
File: protocol/contracts/zaps/PoolMigrationZap.sol    #1

27:                 IERC20(underlying_).safeApprove(address(newPoo
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/zaps/PoolMigrationZap.sol#L27

```
File: protocol/contracts/RewardHandler.sol    #2

52:             IERC20(targetLpToken).safeApprove(address(bkdLocke
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/RewardHandler.sol#L52

```
File: protocol/contracts/RewardHandler.sol   #3

64:             IERC20(token).safeApprove(spender, type(uint256).ma
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/RewardHandler.sol#L64

```
File: protocol/contracts/tokenomics/FeeBurner.sol   #4

118:             IERC20(token_).safeApprove(spender_, type(uint256)
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/tokenomics/FeeBurner.sol#L118

[L-06] Missing checks for `address(0x0)` when assigning values to `address` state variables

There are 8 instances of this issue:

```
File: protocol/contracts/StakerVault.sol

72:             token = _token;
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/StakerVault.sol#L72

```
File: protocol/contracts/BkdLocker.sol

49:             rewardToken = _rewardToken;
```

```
74:             rewardToken = newRewardToken;
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/BkdLocker.sol#L49

```
File: protocol/contracts/tokenomics/VestedEscrowRevocable.sol

43:             treasury = treasury_;
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/tokenomics/VestedEscrowRevocable.sol#L43

```
File: protocol/contracts/tokenomics/AmmGauge.sol

39:             ammToken = _ammToken;
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/tokenomics/AmmGauge.sol#L39

```
File: protocol/contracts/tokenomics/VestedEscrow.sol

65:             fundAdmin = fundAdmin_;
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/tokenomics/VestedEscrow.sol#L65

```
File: protocol/contracts/tokenomics/KeeperGauge.sol

48:             pool = _pool;
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/tokenomics/KeeperGauge.sol#L48

```
File: protocol/contracts/tokenomics/BkdToken.sol

21:           minter = _minter;
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/tokenomics/BkdToken.sol#L21

[L-07] `_prepareDeadline()`, `_setConfig()`, and `_executeDeadline()` should be `private`

I flagged this in the last Backd contest, but it doesn't seem to have been addressed, so bringing it up again: These functions have the ability to bypass the timelocks of every setting. No contract besides the `Preparable` contract itself should need to call these functions, and having them available will lead to exploits. The contracts that currently call `_setConfig()` in their constructors should be given a new function `_initConfig()` for this purpose. The `Vault` calls some of these functions as well, and should be changed to manually inspect the deadline rather than mucking with the internals, which is error-prone. The mappings should also be made `private`, and there should be public getters to read their values

There is 1 instance of this issue:

```
File: protocol/contracts/utils/Preparable.sol   #1

115       /**
116        * @notice Execute uint256 config update (with time del
117        * @dev Needs to be called after the update was prepare
118        * @return New value.
119        */
120       function _executeUInt256(bytes32 key) internal returns
121           _executeDeadline(key);
122           uint256 newValue = pendingUInts256[key];
123           _setConfig(key, newValue);
```

```
124             return newValue;
125         }
126
127         /**
128          * @notice Execute address config update (with time del
129          * @dev Needs to be called after the update was prepare
130          * @return New value.
131          */
132         function _executeAddress(bytes32 key) internal returns
133             _executeDeadline(key);
134             address newValue = pendingAddresses[key];
135             _setConfig(key, newValue);
136             return newValue;
137:        }
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/utils/Preparable.sol#L115-L137

Non-Critical Issues

	Issue	Instances
1	Unneeded import	1
2	Return values of `approve()` not checked	1
3	Large multiples of ten should use scientific notation (e.g. `1e6`) rather than decimal literals (e.g. `1000000`), for readability	2
4	Missing event for critical parameter change	3
5	Use a more recent version of solidity	1
6	Use a more recent version of solidity	16
7	Constant redefined elsewhere	10
8	Inconsistent spacing in comments	3
9	File is missing NatSpec	5
10	NatSpec is incomplete	17
11	Event is missing `indexed` fields	10
12	Not using the named return variables anywhere in the function is confusing	2

	Issue	Instances
9	Typos	6

Total: 80 instances over 13 issues

[N-01] Unneeded import

There is 1 instance of this issue:

```
File: protocol/contracts/tokenomics/BkdToken.sol   #1

8:   import "../../libraries/ScaledMath.sol";
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/tokenomics/BkdToken.sol#L8

[N-02] Return values of `approve()` not checked

Not all `IERC20` implementations `revert()` when there's a failure in `approve()`. The function signature has a `boolean` return value and they indicate errors that way instead. By not checking the return value, operations that should have marked as failed, may potentially go through without actually approving anything

There is 1 instance of this issue:

```
File: protocol/contracts/tokenomics/VestedEscrow.sol   #1

25:           IERC20(rewardToken_).approve(msg.sender, type(uint
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/tokenomics/VestedEscrow.sol#L25

[N-03] Large multiples of ten should use scientific notation

(e.g. `1e6`) rather than decimal literals (e.g. `1000000`), for readability

There are 2 instances of this issue:

```
File: protocol/contracts/utils/CvxMintAmount.sol   #1

10:       uint256 private constant _CLIFF_SIZE = 100000 * 1e18;
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/utils/CvxMintAmount.sol#L10

```
File: protocol/contracts/utils/CvxMintAmount.sol   #2

12:       uint256 private constant _MAX_SUPPLY = 100000000 * 1e1
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/utils/CvxMintAmount.sol#L12

[N-04] Missing event for critical parameter change

There are 3 instances of this issue:

```
File: protocol/contracts/tokenomics/InflationManager.sol   #1

58        function setMinter(address _minter) external override
59            require(minter == address(0), Error.ADDRESS_ALREAD
60            require(_minter != address(0), Error.INVALID_MINTE
61            minter = _minter;
62            return true;
63:       }
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/tokenomics/InflationManager.sol#L58-L63

```
File: protocol/contracts/tokenomics/VestedEscrow.sol   #2

68          function setAdmin(address _admin) external override {
69              require(_admin != address(0), Error.ZERO_ADDRESS_N
70              require(msg.sender == admin, Error.UNAUTHORIZED_AC
71              admin = _admin;
72:         }
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/tokenomics/VestedEscrow.sol#L68-L72

```
File: protocol/contracts/tokenomics/VestedEscrow.sol   #3

74          function setFundAdmin(address _fundadmin) external ove
75              require(_fundadmin != address(0), Error.ZERO_ADDRE
76              require(msg.sender == admin, Error.UNAUTHORIZED_AC
77              fundAdmin = _fundadmin;
78:         }
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/tokenomics/VestedEscrow.sol#L74-L78

[N-05] Use a more recent version of solidity

Use a solidity version of at least 0.8.12 to get `string.concat()` to be used instead of `abi.encodePacked(<str>,<str>)`

There is 1 instance of this issue:

```
File: protocol/contracts/tokenomics/InflationManager.sol   #1

2:    pragma solidity 0.8.10;
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/tokenomics/InflationManager.sol#L2

[N-06] 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 16 instances of this issue:

See original submission for details.

[N-07] 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:

```
File: protocol/contracts/Controller.sol

/// @audit seen in protocol/contracts/StakerVault.sol
21:       IAddressProvider public immutable override addressProv
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/Controller.sol#L21

```
File: protocol/contracts/RewardHandler.sol

/// @audit seen in protocol/contracts/StakerVault.sol
20:       IController public immutable controller;

/// @audit seen in protocol/contracts/Controller.sol
21:       IAddressProvider public immutable addressProvider;
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/RewardHandler.sol#L20

```
File: protocol/contracts/tokenomics/Minter.sol

/// @audit seen in protocol/contracts/RewardHandler.sol
55:       IController public immutable controller;
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/tokenomics/Minter.sol#L55

```
File: protocol/contracts/tokenomics/InflationManager.sol

/// @audit seen in protocol/contracts/RewardHandler.sol
24:       IAddressProvider public immutable addressProvider;
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/tokenomics/InflationManager.sol#L24

```
File: protocol/contracts/tokenomics/AmmGauge.sol

/// @audit seen in protocol/contracts/tokenomics/Minter.sol
20:       IController public immutable controller;
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/tokenomics/AmmGauge.sol#L20

```
File: protocol/contracts/tokenomics/KeeperGauge.sol

/// @audit seen in protocol/contracts/tokenomics/AmmGauge.sol
30:       IController public immutable controller;
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/tokenomics/KeeperGauge.sol#L30

```
File: protocol/contracts/tokenomics/LpGauge.sol

/// @audit seen in protocol/contracts/tokenomics/KeeperGauge.sol
19:       IController public immutable controller;

/// @audit seen in protocol/contracts/StakerVault.sol
21:       IInflationManager public immutable inflationManager;
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/tokenomics/LpGauge.sol#L19

```
File: protocol/contracts/access/RoleManager.sol

/// @audit seen in protocol/contracts/tokenomics/InflationManage
25:       IAddressProvider public immutable addressProvider;
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/access/RoleManager.sol#L25

[N-08] 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 3 instances of this issue:

```
File: protocol/contracts/utils/CvxMintAmount.sol   #1

11:       uint256 private constant _CLIFF_COUNT = 1000; // 1,000
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/utils/CvxMintAmount.sol#L11

```
File: protocol/contracts/utils/CvxMintAmount.sol   #2

14:           IERC20(address(0x4e3FBD56CD56c3e72c1403e103b45Db9da
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/utils/CvxMintAmount.sol#L14

```
File: protocol/contracts/tokenomics/InflationManager.sol   #3

532:      //TOOD: See if this is still needed somewhere
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/tokenomics/InflationManager.sol#L532

[N-09] File is missing NatSpec

There are 5 instances of this issue:

```
File: protocol/contracts/utils/CvxMintAmount.sol
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/utils/CvxMintAmount.sol

```
File: protocol/contracts/tokenomics/VestedEscrowRevocable.sol
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/tokenomics/VestedEscrowRevocable.sol

File: protocol/contracts/tokenomics/VestedEscrow.sol

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/tokenomics/VestedEscrow.sol

File: protocol/contracts/access/Authorization.sol

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/access/Authorization.sol

File: protocol/contracts/access/RoleManager.sol

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/access/RoleManager.sol

[N-10] NatSpec is incomplete

There are 17 instances of this issue:

File: protocol/contracts/StakerVault.sol

```
/// @audit Missing: '@param strategy'
93          /**
94           * @notice Registers an address as a strategy to be ex
95           * @dev This should be used if a strategy deposits int
96           * @return `true` if success.
97           */
98:         function addStrategy(address strategy) external overri
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/StakerVault.sol#L93-L98

File: protocol/contracts/Controller.sol

```
/// @audit Missing: '@param payer'
117         /**
118          * @return the total amount of ETH require by `payer`
119          * positions registered in all actions
120          */
121:        function getTotalEthRequiredForGas(address payer) exte
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/Controller.sol#L117-L121

File: protocol/contracts/utils/Preparable.sol

```
/// @audit Missing: '@param key'
33          /**
34           * @notice Prepares an uint256 that should be committe
35           * after `_MIN_DELAY` elapsed
36           * @param value The value to prepare
37           * @return `true` if success.
38           */
39          function _prepare(
40              bytes32 key,
41              uint256 value,
42              uint256 delay
43:         ) internal returns (bool) {

/// @audit Missing: '@param delay'
33          /**
34           * @notice Prepares an uint256 that should be committe
35           * after `_MIN_DELAY` elapsed
36           * @param value The value to prepare
37           * @return `true` if success.
38           */
39          function _prepare(
40              bytes32 key,
41              uint256 value,
42              uint256 delay
43:         ) internal returns (bool) {

/// @audit Missing: '@param key'
57          /**
```

```
58          * @notice Prepares an address that should be committe
59          * after `_MIN_DELAY` elapsed
60          * @param value The value to prepare
61          * @return `true` if success.
62          */
63         function _prepare(
64             bytes32 key,
65             address value,
66             uint256 delay
67:        ) internal returns (bool) {

/// @audit Missing: '@param delay'
57         /**
58          * @notice Prepares an address that should be committe
59          * after `_MIN_DELAY` elapsed
60          * @param value The value to prepare
61          * @return `true` if success.
62          */
63         function _prepare(
64             bytes32 key,
65             address value,
66             uint256 delay
67:        ) internal returns (bool) {

/// @audit Missing: '@param key'
81         /**
82          * @notice Reset a uint256 key
83          * @return `true` if success.
84          */
85:        function _resetUInt256Config(bytes32 key) internal ret

/// @audit Missing: '@param key'
93         /**
94          * @notice Reset an address key
95          * @return `true` if success.
96          */
97:        function _resetAddressConfig(bytes32 key) internal ret

/// @audit Missing: '@param key'
115        /**
116         * @notice Execute uint256 config update (with time de
117         * @dev Needs to be called after the update was prepar
118         * @return New value.
119         */
120:       function _executeUInt256(bytes32 key) internal returns
```

```
/// @audit Missing: '@param key'
127         /**
128          * @notice Execute address config update (with time de
129          * @dev Needs to be called after the update was prepar
130          * @return New value.
131          */
132:        function _executeAddress(bytes32 key) internal returns
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/utils/Preparable.sol#L33-L43

```
File: protocol/contracts/AddressProvider.sol

/// @audit Missing: '@return'
79          * @param action Address of action to add.
80          */
81:         function addAction(address action) external override o

/// @audit Missing: '@param freezable'
207         /**
208          * @notice Initializes an address
209          * @param key Key to initialize
210          * @param initialAddress Address for `key`
211          */
212         function initializeAddress(
213             bytes32 key,
214             address initialAddress,
215             bool freezable
216:         ) public override onlyGovernance {

/// @audit Missing: '@param key'
264         /**
265          * @notice Execute update of `key`
266          * @return New address.
267          */
268:        function executeAddress(bytes32 key) external override

/// @audit Missing: '@param key'
274         /**
275          * @notice Reset `key`
276          * @return true if it was reset
277          */
278:        function resetAddress(bytes32 key) external override o
```

```
/// @audit Missing: '@param token'
396       /**
397        * @notice Tries to get the staker vault for a given t
398        * @return A boolean set to true if the vault exists a
399        */
400:       function tryGetStakerVault(address token) external vie
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/AddressProvider.sol#L79-L81

```
File: protocol/contracts/tokenomics/InflationManager.sol

/// @audit Missing: '@param lpToken'
236       /**
237        * @notice Execute update of lp pool weight (with time
238        * @dev Needs to be called after the update was prepar
239        * @return New lp pool weight.
240        */
241:       function executeLpPoolWeight(address lpToken) external

/// @audit Missing: '@param token'
321       /**
322        * @notice Execute update of lp pool weight (with time
323        * @dev Needs to be called after the update was prepar
324        * @return New lp pool weight.
325        */
326:       function executeAmmTokenWeight(address token) external
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/tokenomics/InflationManager.sol#L236-L241

[N-11] Event is missing `indexed` fields

Each `event` should use three `indexed` fields if there are three or more fields

There are 10 instances of this issue:

See original submission for details.

[N-12] Not using the named return variables anywhere in the function is confusing

Consider changing the variable to be an unnamed one

There are 2 instances of this issue:

```
File: protocol/contracts/tokenomics/FeeBurner.sol   #1

/// @audit received
43          function burnToTarget(address[] memory tokens_, addres
44              public
45              payable
46              override
47:             returns (uint256 received)
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/tokenomics/FeeBurner.sol#L43-L47

```
File: protocol/contracts/tokenomics/FeeBurner.sol   #2

/// @audit received
96          function _depositInPool(address underlying_, ILiquidit
97              internal
98:             returns (uint256 received)
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/tokenomics/FeeBurner.sol#L96-L98

[N-13] Typos

There are 6 instances of this issue:

```
File: protocol/contracts/BkdLocker.sol

/// @audit invlude
```

```
173:        * @dev This does not invlude the gov. tokens queued f
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/BkdLocker.sol#L173

```
File: protocol/contracts/tokenomics/InflationManager.sol

/// @audit TOOD
532:      //TOOD: See if this is still needed somewhere
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/tokenomics/InflationManager.sol#L532

```
File: protocol/contracts/tokenomics/FeeBurner.sol

/// @audit Emmited
29:       event Burned(address targetLpToken, uint256 amountBurn

/// @audit successfull
29:       event Burned(address targetLpToken, uint256 amountBurn

/// @audit Recieve
35:       receive() external payable {} // Recieve function for

/// @audit Transfering
84:           // Transfering LP tokens back to sender
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/tokenomics/FeeBurner.sol#L29

Alex the Entreprenerd (judge) commented:

> 1. migrate() still does transfers when the transfer is to the same pool, and this can be done multiple times

> Interesting finding, also wonder if this could cause issues with fees, but in lack of POC, I think this is a valid Low Severity finding
> **2. Non-exploitable reentrancy**
> Agree with severity and finding, would rephrase to “code doesn’t conform to CEI”

> **3. Users can DOS themselves by executing prepareUnlock(0) many times**
> This should be downgraded to non-critical because it probably requires tens of thousands of calls, that said the finding is valid

> **4. Unused/empty receive()/fallback() function**
> I fail to see the need for the extra checks given that the contracts are meant to handle ETH

> **5. safeApprove() is deprecated**
> Technically valid, however the code is using `safeApprove` correctly, only once, from zero to X

> **6. Missing checks for address(0x0) when assigning values to address state variables**
> Valid

> **7. _prepareDeadline(), _setConfig(), and _executeDeadline() should be private**
> Disagree with the alarmist side, but there’s validity to this finding.

> **Non-critical Issues**
> Agree with the findings although it feels like a bot wrote this.

> Overall a really exhaustive report, 3 findings are interesting the rest doesn’t stand out, however the thoroughness of the report does.

Gas Optimizations

For this contest, 41 reports were submitted by wardens detailing gas optimizations. The report highlighted below by **IIIIIII** received the top score from the judge.

The following wardens also submitted reports: MiloTruck, Picodes, robee, defsec, 0xKitsune, sashik_eth, fatherOfBlocks, berndartmueller, SmartSek, Tadashi, Dravee, djxploit, csanuragjain, scaraven, c3phas, Tomio, 0xkatana, 0x1f8b,

SecureZeroX, Chom, Fitraldys, Sm4rty, asutorufos, hake, Randyyy, RoiEvenHaim, Funen, StyxRave, Waze, oyc_109, 0xf15ers, simon135, sach1r0, 0xNazgul, Kaiziron, gzeon, catchup, hansfriese, 0x29A, *and* delfin454000.

	Issue	Instances
1	Multiple `address` mappings can be combined into a single `mapping` of an `address` to a `struct`, where appropriate	5
2	State variables only set in the constructor should be declared `immutable`	1
3	State variables can be packed into fewer storage slots	2
4	State variables should be cached in stack variables rather than re-reading them from storage	32
5	Multiple accesses of a mapping/array should use a local variable cache	7
6	The result of external function calls should be cached rather than re-calling the function	2
7	`<x> += <y>` costs more gas than `<x> = <x> + <y>` for state variables	14
8	`internal` functions only called once can be inlined to save gas	6
9	Add `unchecked {}` for subtractions where the operands cannot underflow because of a previous `require()`	1
10	`<array>.length` should not be looked up in every loop of a `for`-loop	8
11	`++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	1
12	`require()` / `revert()` strings longer than 32 bytes cost extra gas	1
13	Using `bool`s for storage incurs overhead	7
14	Using `> 0` costs more gas than `!= 0` when used on a `uint` in a `require()` statement	7
15	Usage of `uints` / `ints` smaller than 32 bytes (256 bits) incurs overhead	3
16	Using `private` rather than `public` for constants, saves gas	11
17	Duplicated `require()` / `revert()` checks should be refactored to a modifier or function	7
18	`require()` or `revert()` statements that check input arguments should be at the top of the function	3
19	Empty blocks should be removed or emit something	3

	Issue	Instances
20	Use custom errors rather than `revert()` / `require()` strings to save deployment gas	109
21	Functions guaranteed to revert when called by normal users can be marked `payable`	53

Total: 283 instances over 21 issues

[1] 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 are 5 instances of this issue:

```
File: protocol/contracts/StakerVault.sol

50          mapping(address => uint256) public balances;
51          mapping(address => uint256) public actionLockedBalance
52
53:         mapping(address => mapping(address => uint256)) intern
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/StakerVault.sol#L50-L53

```
File: protocol/contracts/BkdLocker.sol

27          mapping(address => uint256) public balances;
28          mapping(address => uint256) public boostFactors;
29          mapping(address => uint256) public lastUpdated;
30          mapping(address => WithdrawStash[]) public stashedGovT
```

```
31:         mapping(address => uint256) public totalStashed;
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/BkdLocker.sol#L27-L31

```
File: protocol/contracts/tokenomics/AmmGauge.sol

27          mapping(address => uint256) public perUserStakedIntegr
28:         mapping(address => uint256) public perUserShare;
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/tokenomics/AmmGauge.sol#L27-L28

```
File: protocol/contracts/tokenomics/VestedEscrow.sol

44          mapping(address => uint256) public initialLocked;
45          mapping(address => uint256) public totalClaimed;
46:         mapping(address => address) public holdingContract;
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/tokenomics/VestedEscrow.sol#L44-L46

```
File: protocol/contracts/tokenomics/LpGauge.sol

25          mapping(address => uint256) public perUserStakedIntegr
26:         mapping(address => uint256) public perUserShare;
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/tokenomics/LpGauge.sol#L25-L26

[2] 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 is 1 instance of this issue:

```
File: protocol/contracts/tokenomics/VestedEscrow.sol   #1

39:       uint256 public totalTime;
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/tokenomics/VestedEscrow.sol#L39

[3] State variables can be packed into fewer storage slots

If variables occupying the same slot are both written the same function or by the constructor, avoids a separate Gsset (**20000 gas**). Reads of the variables can also be cheaper

There are 2 instances of this issue:

```
File: protocol/contracts/tokenomics/VestedEscrow.sol   #1

/// @audit Variable ordering with 8 slots instead of the current
/// @audit  uint256(32):totalTime, uint256(32):initialLockedSupp
34:       address public admin;
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/tokenomics/VestedEscrow.sol#L34

```
File: protocol/contracts/tokenomics/KeeperGauge.sol   #2

/// @audit Variable ordering with 5 slots instead of the current
/// @audit  mapping(32):keeperRecords, mapping(32):perPeriodTota
```

```
27:       mapping(address => KeeperRecord) public keeperRecords;
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/tokenomics/KeeperGauge.sol#L27

[4] 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 32 instances of this issue:

```
File: protocol/contracts/StakerVault.sol

/// @audit token
330:          uint256 oldBal = IERC20(token).balanceOf(address(t

/// @audit token
333:              ILiquidityPool pool = addressProvider.getPoolF

/// @audit token
337:          IERC20(token).safeTransferFrom(msg.sender, address

/// @audit token
338:          uint256 staked = IERC20(token).balanceOf(address(t

/// @audit token
375:          uint256 oldBal = IERC20(token).balanceOf(address(t

/// @audit token
381:          IERC20(token).safeTransfer(dst, amount);

/// @audit token
383:          uint256 unstaked = oldBal.uncheckedSub(IERC20(toke
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/StakerVault.sol#L330

```
File: protocol/contracts/BkdLocker.sol

/// @audit totalLockedBoosted
97:           curRewardTokenData.feeIntegral += amount.scaledDiv
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/BkdLocker.sol#L97

```
File: protocol/contracts/tokenomics/Minter.sol

/// @audit currentInflationAmountLp
91:               currentInflationAmountLp +

/// @audit currentInflationAmountLp
208:                  currentInflationAmountLp +

/// @audit currentInflationAmountKeeper
92:               currentInflationAmountKeeper +

/// @audit currentInflationAmountKeeper
209:                  currentInflationAmountKeeper +

/// @audit currentInflationAmountAmm
93:               currentInflationAmountAmm;

/// @audit currentInflationAmountAmm
210:                  currentInflationAmountAmm;

/// @audit totalAvailableToNow
220:          require(newTotalMintedToNow <= totalAvailableToNow
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/tokenomics/Minter.sol#L91

```
File: protocol/contracts/tokenomics/InflationManager.sol

/// @audit minter
501:          uint256 lpInflationRate = Minter(minter).getLpInfl

/// @audit minter
511:          uint256 keeperInflationRate = Minter(minter).getKe

/// @audit minter
526:          uint256 ammInflationRate = Minter(minter).getAmmIn

/// @audit totalKeeperPoolWeight
517:              totalKeeperPoolWeight;

/// @audit totalKeeperPoolWeight
575:          totalKeeperPoolWeight = totalKeeperPoolWeight > 0

/// @audit totalLpPoolWeight
502:          uint256 poolInflationRate = (currentUInts256[key]

/// @audit totalLpPoolWeight
589:          totalLpPoolWeight = totalLpPoolWeight > 0 ? totalL

/// @audit totalAmmTokenWeight
528:              totalAmmTokenWeight;

/// @audit totalAmmTokenWeight
602:          totalAmmTokenWeight = totalAmmTokenWeight > 0 ? to
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/tokenomics/InflationManager.sol#L501

```
File: protocol/contracts/tokenomics/AmmGauge.sol

/// @audit ammStakedIntegral
159:          perUserStakedIntegral[user] = ammStakedIntegral;

/// @audit totalStaked
90:                   (block.timestamp - uint256(ammLastUpdated)

/// @audit totalStaked
148:              ammStakedIntegral += (currentRate * timeElapse
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/tokenomics/AmmGauge.sol#L159

```
File: protocol/contracts/tokenomics/VestedEscrow.sol

/// @audit unallocatedSupply
84:           require(unallocatedSupply > 0, "No reward tokens i
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/tokenomics/VestedEscrow.sol#L84

```
File: protocol/contracts/tokenomics/KeeperGauge.sol

/// @audit epoch
87:           keeperRecords[beneficiary].feesInPeriod[epoch] += 

/// @audit epoch
88:           perPeriodTotalFees[epoch] += amount;

/// @audit epoch
131:              endEpoch = epoch;
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/tokenomics/KeeperGauge.sol#L87

```
File: protocol/contracts/access/RoleManager.sol

/// @audit _roles[role].members
148:          return _roles[role].members[account];
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/access/RoleManager.sol#L148

[5] 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 7 instances of this issue:

```
File: protocol/contracts/BkdLocker.sol

/// @audit stashedWithdraws[i]
142:                 totalAvailableToWithdraw += stashedWithdra
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/BkdLocker.sol#L142

```
File: protocol/contracts/tokenomics/VestedEscrow.sol

/// @audit amounts[i]
96:             address recipient_ = amounts[i].recipient;
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/tokenomics/VestedEscrow.sol#L96

```
File: protocol/contracts/tokenomics/KeeperGauge.sol

/// @audit keeperRecords[beneficiary]
84:             keeperRecords[beneficiary].firstEpochSet = tru

/// @audit keeperRecords[beneficiary]
85:             keeperRecords[beneficiary].nextEpochToClaim =

/// @audit keeperRecords[beneficiary]
```

```
87:             keeperRecords[beneficiary].feesInPeriod[epoch] +=

/// @audit keeperRecords[beneficiary]
139:            keeperRecords[beneficiary].nextEpochToClaim = endE
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/tokenomics/KeeperGauge.sol#L84

```
File: protocol/contracts/access/RoleManager.sol

/// @audit _roles[role]
148:            return _roles[role].members[account];
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/access/RoleManager.sol#L148

[6] The result of external function calls should be cached rather than re-calling the function

The instances below point to the second+ call of the function within a single function

There are 2 instances of this issue:

```
File: protocol/contracts/StakerVault.sol    #1

/// @audit _controller.addressProvider()
62:             Authorization(_controller.addressProvider().getRol
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/StakerVault.sol#L62

```
File: protocol/contracts/tokenomics/InflationManager.sol    #2

/// @audit i.uncheckedInc()
```

```
121:          for (uint256 i; i < length; i = i.uncheckedInc())
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/tokenomics/InflationManager.sol#L121

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

There are 14 instances of this issue:

```
File: protocol/contracts/StakerVault.sol

343:              strategiesTotalStaked += staked;

345:              _poolTotalStaked += staked;

392:              strategiesTotalStaked -= unstaked;

394:              _poolTotalStaked -= unstaked;
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/StakerVault.sol#L343

```
File: protocol/contracts/BkdLocker.sol

152:          totalLocked -= totalAvailableToWithdraw;

230:          totalLocked += amount;
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/BkdLocker.sol#L152

```
File: protocol/contracts/tokenomics/Minter.sol
```

```
154:             issuedNonInflationSupply += amount;

188:             totalAvailableToNow += (currentTotalInflation * (b

218:             totalAvailableToNow += ((block.timestamp - lastEve
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/tokenomics/Minter.sol#L154

```
File: protocol/contracts/tokenomics/VestedEscrowRevocable.sol

67:             _vestedBefore += vested;
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/tokenomics/VestedEscrowRevocable.sol#L67

```
File: protocol/contracts/tokenomics/AmmGauge.sol

113:             totalStaked += staked;

135:             totalStaked -= unstaked;

148:                 ammStakedIntegral += (currentRate * timeElapse
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/tokenomics/AmmGauge.sol#L113

```
File: protocol/contracts/tokenomics/LpGauge.sol

115                 poolStakedIntegral += (currentRate * (block.ti
116                     poolTotalStaked
117:                );
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/tokenomics/LpGauge.sol#L115-L117

[8] `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 6 instances of this issue:

```
File: protocol/contracts/AddressProvider.sol

433:       function _addKnownAddressKey(bytes32 key, AddressProvi
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/AddressProvider.sol#L433

```
File: protocol/contracts/RewardHandler.sol

62:       function _approve(address token, address spender) inte
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/RewardHandler.sol#L62

```
File: protocol/contracts/tokenomics/KeeperGauge.sol

146:       function _mintRewards(address beneficiary, uint256 amo
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/tokenomics/KeeperGauge.sol#L146

```
File: protocol/contracts/tokenomics/FeeBurner.sol

96          function _depositInPool(address underlying_, ILiquidit
97              internal
98:             returns (uint256 received)

125:        function _swapperRouter() internal view returns (ISwap
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/tokenomics/FeeBurner.sol#L96-L98

```
File: protocol/contracts/tokenomics/LpGauge.sol

106:        function _mintRewards(address beneficiary, uint256 amo
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/tokenomics/LpGauge.sol#L106

[9] Add `unchecked {}` for subtractions where the operands cannot underflow because of a previous `require()`

`require(a <= b); x = b - a` => `require(a <= b); unchecked { x = b - a }`

There is 1 instance of this issue:

```
File: protocol/contracts/tokenomics/VestedEscrow.sol   #1

63:             totalTime = endtime_ - starttime_;
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/tokenomics/VestedEscrow.sol#L63

[10] `<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 8 instances of this issue:

```
File: protocol/contracts/StakerVault.sol

259:          for (uint256 i; i < actions.length; i = i.unchecke
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/StakerVault.sol#L259

```
File: protocol/contracts/zaps/PoolMigrationZap.sol

22:           for (uint256 i; i < newPools_.length; ++i) {

39:           for (uint256 i; i < oldPoolAddresses_.length; ) {
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/zaps/PoolMigrationZap.sol#L22

```
File: protocol/contracts/RewardHandler.sol

42:           for (uint256 i; i < pools.length; i = i.uncheckedI
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/RewardHandler.sol#L42

```
File: protocol/contracts/tokenomics/InflationManager.sol

116:          for (uint256 i; i < stakerVaults.length; i = i.unc
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/tokenomics/InflationManager.sol#L116

```
File: protocol/contracts/tokenomics/VestedEscrow.sol

94:           for (uint256 i; i < amounts.length; i = i.unchecke
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/tokenomics/VestedEscrow.sol#L94

```
File: protocol/contracts/tokenomics/FeeBurner.sol

56:           for (uint256 i; i < tokens_.length; i = i.unchecke
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/tokenomics/FeeBurner.sol#L56

```
File: protocol/contracts/access/RoleManager.sol

82:           for (uint256 i; i < roles.length; i = i.uncheckedI
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/access/RoleManager.sol#L82

[11] `++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 is 1 instance of this issue:

```
File: protocol/contracts/zaps/PoolMigrationZap.sol   #1

22:           for (uint256 i; i < newPools_.length; ++i) {
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/zaps/PoolMigrationZap.sol#L22

[12] `require()` / `revert()` strings longer than 32 bytes cost extra gas

There is 1 instance of this issue:

```
File: protocol/contracts/tokenomics/Minter.sol   #1

150           require(
151               issuedNonInflationSupply + amount <= nonInflat
152               "Maximum non-inflation amount exceeded."
153:          );
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/tokenomics/Minter.sol#L150-L153

[13] 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 7 instances of this issue:

```
File: protocol/contracts/StakerVault.sol

58:         mapping(address => bool) public strategies;
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/StakerVault.sol#L58

```
File: protocol/contracts/tokenomics/Minter.sol

41:         bool public initialPeriodEnded;
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/tokenomics/Minter.sol#L41

```
File: protocol/contracts/tokenomics/InflationManager.sol

31:         bool public weightBasedKeeperDistributionDeactivated;

41:         mapping(address => bool) public gauges;
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/tokenomics/InflationManager.sol#L31

```
File: protocol/contracts/tokenomics/AmmGauge.sol

31:       bool public killed;
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/tokenomics/AmmGauge.sol#L31

```
File: protocol/contracts/tokenomics/VestedEscrow.sol

42:       bool public initializedSupply;
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/tokenomics/VestedEscrow.sol#L42

```
File: protocol/contracts/tokenomics/KeeperGauge.sol

37:       bool public override killed;
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/tokenomics/KeeperGauge.sol#L37

[14] 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:

```
File: protocol/contracts/BkdLocker.sol

91:           require(amount > 0, Error.INVALID_AMOUNT);

92:           require(totalLockedBoosted > 0, Error.NOT_ENOUGH_F

137:          require(length > 0, "No entries");
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/BkdLocker.sol#L91

```
File: protocol/contracts/tokenomics/AmmGauge.sol

104:          require(amount > 0, Error.INVALID_AMOUNT);

125:          require(amount > 0, Error.INVALID_AMOUNT);
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/tokenomics/AmmGauge.sol#L104

```
File: protocol/contracts/tokenomics/VestedEscrow.sol

84:           require(unallocatedSupply > 0, "No reward tokens i
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/tokenomics/VestedEscrow.sol#L84

```
File: protocol/contracts/tokenomics/KeeperGauge.sol

140:          require(totalClaimable > 0, Error.ZERO_TRANSFER_NO
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/tokenomics/KeeperGauge.sol#L140

[15] Usage of `uints` / `ints` smaller than 32 bytes (256 bits) incurs overhead

> When using elements that are smaller than 32 bytes, your contract's gas usage may be higher. This is because the EVM operates on 32 bytes at a time. Therefore, if the element is smaller than that, the EVM must use more operations in order to reduce the size of the element from 32 bytes to the desired size.

https://docs.soliditylang.org/en/v0.8.11/internals/layout_in_storage.html Use a larger size then downcast where needed

There are 3 instances of this issue:

```
File: protocol/contracts/StakerVault.sol   #1

295:       function decimals() external view override returns (ui
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/StakerVault.sol#L295

```
File: protocol/contracts/tokenomics/AmmGauge.sol   #2

32:        uint48 public ammLastUpdated;
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/tokenomics/AmmGauge.sol#L32

```
File: protocol/contracts/tokenomics/KeeperGauge.sol   #3

34:        uint48 public lastUpdated;
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/tokenomics/KeeperGauge.sol#L34

[16] 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 11 instances of this issue:

```
File: protocol/contracts/tokenomics/Minter.sol

25:       uint256 public immutable initialAnnualInflationRateLp;

26:       uint256 public immutable annualInflationDecayLp;

30:       uint256 public immutable initialPeriodKeeperInflation;

31:       uint256 public immutable initialAnnualInflationRateKee

32:       uint256 public immutable annualInflationDecayKeeper;

36:       uint256 public immutable initialPeriodAmmInflation;

37:       uint256 public immutable initialAnnualInflationRateAmm

38:       uint256 public immutable annualInflationDecayAmm;

44:       uint256 public immutable nonInflationDistribution;
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/tokenomics/Minter.sol#L25

```
File: protocol/contracts/tokenomics/VestedEscrow.sol

37:       uint256 public immutable startTime;

38:       uint256 public immutable endTime;
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/tokenomics/VestedEscrow.sol#L37

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

Saves deployment costs

There are 7 instances of this issue:

```
File: protocol/contracts/StakerVault.sol

223:          require(addressProvider.isAction(msg.sender), Erro
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/StakerVault.sol#L223

```
File: protocol/contracts/utils/Preparable.sol

98:           require(deadlines[key] != 0, Error.NOTHING_PENDING
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/utils/Preparable.sol#L98

```
File: protocol/contracts/AddressProvider.sol

260:          require(!meta.frozen, Error.ADDRESS_FROZEN);
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/AddressProvider.sol#L260

```
File: protocol/contracts/tokenomics/InflationManager.sol
```

```
270:              require(IStakerVault(stakerVault).getLpGauge()

365:          require(length == weights.length, "Invalid length
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/tokenomics/InflationManager.sol#L270

```
File: protocol/contracts/tokenomics/AmmGauge.sol

125:          require(amount > 0, Error.INVALID_AMOUNT);
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/tokenomics/AmmGauge.sol#L125

```
File: protocol/contracts/tokenomics/VestedEscrow.sol

76:           require(msg.sender == admin, Error.UNAUTHORIZED_AC
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/tokenomics/VestedEscrow.sol#L76

[18] `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 3 instances of this issue:

```
File: protocol/contracts/Controller.sol   #1

35:           require(_inflationManager != address(0), Error.INV
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/Controller.sol#L35

```
File: protocol/contracts/tokenomics/InflationManager.sol   #2

60:           require(_minter != address(0), Error.INVALID_MINTE
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/tokenomics/InflationManager.sol#L60

```
File: protocol/contracts/tokenomics/VestedEscrowRevocable.sol

54:           require(_recipient != treasury, "Treasury cannot b
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/tokenomics/VestedEscrowRevocable.sol#L54

[19] 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:

```
File: protocol/contracts/zaps/PoolMigrationZap.sol   #1
```

```
31:         receive() external payable {}
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/zaps/PoolMigrationZap.sol#L31

```
File: protocol/contracts/RewardHandler.sol   #2

30:         receive() external payable {}
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/RewardHandler.sol#L30

```
File: protocol/contracts/tokenomics/FeeBurner.sol   #3

35:         receive() external payable {} // Recieve function for
```

https://github.com/code-423n4/2022-05-backd/blob/2a5664d35cde5b036074edef3c1369b984d10010/protocol/contracts/tokenomics/FeeBurner.sol#L35

[20] 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 109 instances of this issue:

See original submission for details.

21. 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 53 instances of this issue:

See original submission for details.

chase-manning (Backd) commented:

> I consider this report to be of particularly high quality.

Alex the Entreprenerd (judge) commented:

> View a detailed breakdown of the judge's considerations.

> This is hands down the best gas report I've ever reviewed, there's only two improvements I'd recommend:
>
> - Post all the extra details as separate gists to make it easier to scan vs read
> - Show total gas saved in one line for each finding

> This would make it easier to score, review for the sponsor and immediately gives a sense of value to the findings

> Additionally a couple of the findings, which are completely valid, would require more detailed POC to be actionable and helpful, hence I gave them no points.

> That said this is the best I've seen so far!

> Total Gas Saved: 7415

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