SHERLOCK

Security Review For Super DCA

Public Audit Contest Prepared For: **Super DCA**
Lead Security Expert: **vinica_boy**
Date Audited: **September 29 - October 3, 2025**

Introduction

This contest reviews the contracts that make up the Super DCA Liquidity Network, powered by Uniswap V4 Hooks. It focuses on the security and economic soundness of the contract suite coordinating dynamic fees, emitting DCA tokens using a Curve-style gauge, and locking liquidity when new tokens are listed.

Scope

Repository: Super-DCA-Tech/super-dca-cashback

Audited Commit: e9fd71d431af2da205a924547bce5484daa5e9df

Final Commit: 0aa9394fcf12336425a08fb489c5b3db1f4f5445

Files:

- src/interfaces/ISuperDCATrade.sol
- src/SuperDCACashback.sol

Repository: Super-DCA-Tech/super-dca-gauge

Audited Commit: 3a15b6c145c72c97ca00f56bb07fe727e2f49330

Final Commit: 75be287521127c398263bab2856d9f08636627a9

Files:

- src/interfaces/ISuperDCAStaking.sol
- src/SuperDCAGauge.sol
- src/SuperDCAListing.sol
- src/SuperDCAStaking.sol
- src/SuperDCAToken.sol

Findings

Each issue has an assigned severity:

- High issues are directly exploitable security vulnerabilities that need to be fixed.
- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.

Issues Found

High	Medium
4	4

Issues Not Fixed and Not Acknowledged

High	Medium
0	0

Security experts who found valid issues

0x00T1
0x11singh99
0x60scs
0xAadi
0xAdwa
0xB4nkz
0xBoraichoT
0xCrypt0nite
0xCuru
0xDaniel_eth
0xDemon
0xDjango
0xHexed
0xImmortan
0xaxaxa
0xheartcode
0xleo
0xlookman
0xpetern
0xsolisec
0xv1bh4
0xzen
33audits
4Nescient
8olidity
Aamirusmani1552
Al-Qa-qa
AlexCzm
Almanax
Arav
Artur
AuditorPraise
BADROBINX
Bbash
BengalCatBalu
Bobai23
Boy2000
BoyD
Chonkov
CodexBugmeNot
Cryptor
DSbeX
DemiGods
Divine_Dragon
DuoSec
EQUATION
FonDevs
HeckerTrieuTien
Hunter
Hurricane
Icon_0x
Immanux
Ironsidesec
IzuMan
JeRRy0422
JohnTPark24
JohnWeb3
Josh4324
JuggerNaut
Kirkeelee
KiroBrejka
LonWof-Demon
MysteryAuditor
NHristov
Ollam
Opecon
Orhukl
OxSath404
PASCAL
Phaethon
PolarizedLight
Pro_King
R
Ragnarok
Razkky
SMB62
SOPROBRO
S_722
SaIntRobi
SarveshLimaye
ScarletFir
Siiiisivan
Sir_Shades
SuperDevFavour

TAdev0
TECHFUND-inc
TsvetanLobutov
VCeb
Victor_TheOracle
Whiterabbit
WillyCode20
Xmanuel
Y4nhu1
Yaneca_b
ZeroEx
Ziusz
_Ranjan2003
alekso91
alexbabits
algiz
alicrali33
aman
ami
axelot
bXiv
bam0x7
bbl4de
bughunter442
bustomer
cholakovvv
codexNature
dani3l526
deadmanwalking
denys_sosnovskyi
derastephh
djshaneden
drdee
engineer
eww
ezsia
farman1094
francoHacker
frankauditcraft
fullstop
futureHack
glitch-Hunter
gneiss
grigorovv17
har0507
harry
heavyw8t
hjo
holtzzx
illoy_sci
itsRavin
ivanalexandur
jah
jayjoshix
jo13
kangaroo
kazan
ke1caM
khaye26
kimnoic
kimonic
kom
lucky-gru
maigadoh
makeWeb3safe
marcosecure0x
merlin
mingzoox
montecristo
namx05
natachi
nganhg
ni8mare
nonso72
oct0pwn
omeiza
oot2k
patitonar
peazzycole
pindarev
pollersan
proofvoid
prosper
proxima_centuri
pv
r1ver
rashmor
resosiloris
rsam_eth
sakibcy
secret__one
sedare
shiazinho
shieldrey
silver_eth
slavina
softdev0323
soloking
songyuqi
sourav_DEV
surenyan-oks
taticuvostru
techOptimizor
the_haritz
theholymarvycodes
thekmj
theweb3mechanic
tinnohofficial
tobi0x18
typicalHuman
ubl4nk
udo
v10g1
v_2110
vinica_boy
vivekd
volleyking
whitehair0330
wickie
x0t0wt1w
xxiv
y4y
yeahChibyke
zcai

Issue H-1: Users with pre-campaign trades will drain cashback funds by claiming retroactive rewards for periods before campaign existed

Source: https://github.com/sherlock-audit/2025-09-super-dca-judging/issues/71

Found by

0xpetern, 0xv1bh4, BengalCatBalu, Boy2000, DSbeX, DuoSec, Ironsidesec, Josh4324, JuggerNaut, Ollam, ScarletFir, Sir_Shades, SuperDevFavour, TECHFUND-inc, Y4nhu1, Ziusz, _Ranjan2003, bustomer, codexNature, deadmanwalking, drdee, eww, fullstop, glitch-Hunter, heavyw8t, itsRavin, jayjoshix, ke1caM, khaye26, lucky-gru, maigadoh, makeWeb3safe, montecristo, peazzycole, secret__one, softdev0323, tinnohofficial, vivekd, zcai

Summary

The contract contains a `cashbackClaim.startTime` field that defines when the campaign begins, but this field is never validated or used in reward calculations. As a result, trades that started before the campaign launch can claim cashback rewards retroactively for all time periods, including those before the campaign existed. This allows early trade holders to drain disproportionate amounts of cashback funds meant for legitimate campaign participants.

Root Cause

https://github.com/sherlock-audit/2025-09-super-dca/blob/main/super-dca-cashback/src/SuperDCACashback.sol#L152

https://github.com/sherlock-audit/2025-09-super-dca/blob/main/super-dca-cashback/src/SuperDCACashback.sol#L168

The root cause is the complete absence of `cashbackClaim.startTime` validation in three critical locations:

1. `Constructor` Does not validate `_cashbackClaim.startTime`
2. `_isTradeValid()`: Does not check `trade.startTime >= cashbackClaim.startTime`
3. `_calculateEpochData()` : Calculates `timeElapsed` from `trade.startTime` instead of `max(trade.startTime, cashbackClaim.startTime)`

Internal Pre-conditions

1. A user must have created a trade in the SuperDCATrade contract with `trade.startTime < cashbackClaim.startTime`
2. The trade must meet minimum flow rate requirements (`trade.flowRate >= cashbackClaim.minRate`)
3. The trade must still be active or have ended after the campaign started
4. User must own the trade NFT

External Pre-conditions

N/A

Attack Path

1. User creates Trade #1 on January 1, 2024 (timestamp: 1704067200) with valid flow rate
2. Protocol team deploys SuperDCACashback contract on March 1, 2024 with `cashbackClaim.startTime = 1709251200` (March 1, 2024)
3. Campaign is intended to reward participants starting March 1 onwards
4. On March 15, 2024, user calls `claimAllCashback(1)`
5. `_calculateEpochData()` calculates: `timeElapsed = currentTime - trade.startTime = March 15 - January 1 = 73 days`
6. User receives rewards for 10+ completed epochs (~73 days worth) instead of just 2 epochs (~14 days)
7. User claims 5x more rewards than intended, draining funds meant for legitimate campaign participants
8. Multiple pre-campaign trade holders repeat this, potentially draining the entire cashback pool

Impact

- **Fund Drainage**: Pre-campaign trades can drain the majority of cashback funds before legitimate participants can claim
- **Unfair Advantage**: Users who happened to create trades months before campaign launch get massively disproportionate rewards (up to unlimited retroactive rewards)
- **Campaign Failure**: The cashback campaign fails to incentivize new behavior since all funds go to pre-existing trades

- **Insolvency Risk**: Contract becomes insolvent as retroactive claims exceed funded amounts

PoC

```
Timeline:

Jan 1, 2024          Mar 1, 2024           Mar 15, 2024



Trade Created      Campaign Launches      User Claims
(startTime:        (cashbackClaim.        Rewards
 timestamp 1)       startTime:
                    timestamp 2)

Expected Reward Period:  [  14 days  ]
Actual Reward Period:    [     73 days      ]
                         (Jan 1 to Mar 15)
```

Mitigation

Implement proper validation and usage of the `cashbackClaim.startTime`

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/Super-DCA-Tech/super-dca-cashback/pull/24

Issue H-2: Attackers will steal rewards from legitimate pools by making duplicate pools for listed token.

Source: https://github.com/sherlock-audit/2025-09-super-dca-judging/issues/662

Found by

0xB4nkz, 0xBoraichoT, 0xDaniel_eth, 8olidity, Aamirusmani1552, BADROBINX, BengalCatBalu, JeRRy0422, Kirkeelee, Ollam, PolarizedLight, francoHacker, harry, illoy_sci, ivanalexandur, jayjoshix, lucky-gru, maigadoh, nganhg, silver_eth, techOptimizor, v10g1, vinica_boy, y4y

Summary

The per-token reward accrual without pool validation will cause reward theft for legitimate pools as attackers will create malicious pools and trigger reward distribution.

Root Cause

In SuperDCAGauge.sol, the `_handleDistributionAndSettlement` function accrues rewards globally per token via `staking.accrueReward(otherToken)` without validating if the specific pool is legitimately listed. Since Uniswap V4 allows multiple pools for the same token pair (e.g., USDC/SuperDCA with different fees or configurations), and the listing contract (SuperDCAListing.sol) only permits one listing per non-SuperDCA token, attackers can create unlisted pools using the same hook to siphon rewards meant for listed pools.

https://github.com/sherlock-audit/2025-09-super-dca/blob/main/super-dca-gauge/src/SuperDCAGauge.sol#L323-L367

Internal Pre-conditions

- At least one token (e.g., USDC) must be listed via the SuperDCAListing contract for a legitimate pool.
- The SuperDCAGauge hook must be deployed and configured with staking and listing contracts.
- The staking contract must have accrued rewards for the listed token.

External Pre-conditions

- No minimum liquidity checks in the hook's _handleDistributionAndSettlement (only enforced during listing).

Attack Path

1. Attacker identifies a listed token (e.g., USDC) with an existing legitimate pool (e.g., USDC/SuperDCA with dynamic fee).
2. Attacker creates a new malicious pool for the same pair (USDC/SuperDCA) using the same SuperDCAGauge hook, with minimal or zero liquidity (bypassing listing requirements).
3. Attacker adds minimal liquidity to the malicious pool, triggering `_beforeAddLiquidity` which calls `_handleDistributionAndSettlement`.
4. In `_handleDistributionAndSettlement`, rewards are accrued for the token (USDC) globally, and the community share is donated to the malicious pool (since it has liquidity).
5. Attacker removes liquidity from the malicious pool, triggering `_beforeRemoveLiquidity` again, potentially accruing more rewards.
6. Attacker repeats add/remove operations to maximize reward theft, then withdraws the donated rewards from the malicious pool.

Impact

Legitimate pools suffer a complete loss of community rewards, as rewards are diverted to the attacker's malicious pool. The protocol's reward distribution mechanism is undermined, potentially leading to reduced incentives for legitimate liquidity providers and loss of trust.

PoC

No response

Mitigation

Add pool-specific validation in `_handleDistributionAndSettlement` to check if the pool is listed before accruing rewards. For example, query the listing contract per pool key/ID or enforce that only listed pools can trigger reward distribution. Alternatively, modify the staking contract to accrue rewards per-pool instead of per-token, or add minimum liquidity checks in the hook. Ensure the listing contract tracks pools by key/ID rather than just tokens.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/Super-DCA-Tech/super-dca-gauge/pull/37

Issue H-3: Fee collection will always fail for initial positions of SuperDCA pools that contain native tokens

Source: https://github.com/sherlock-audit/2025-09-super-dca-judging/issues/720

Found by

0x60scs, 0xB4nkz, 0xBoraichoT, 0xleo, 0xzen, 4Nescient, Aamirusmani1552, Al-Qa-qa, BADROBINX, BengalCatBalu, Boy2000, Cryptor, DuoSec, Icon_0x, IzuMan, JeRRy0422, JohnTPark24, JohnWeb3, Kirkeelee, Ollam, Orhukl, R, Razkky, SMB62, SuperDevFavour, alicrali33, bbl4de, bughunter442, deadmanwalking, drdee, farman1094, gneiss, holtzzx, jo13, kom, namx05, natachi, nonso72, oct0pwn, pindarev, shiazinho, shieldrey, silver_eth, techOptimizor, tobi0x18, udo, vinica_boy, vivekd, volleyking, wickie, y4y, yeahChibyke

Summary

Uniswap v4 represents native tokens like ETH as `address(0)` internally (ref). The protocol has stated that all uniswap v4 pools need to be supported, including ones that contain native ETH as SuperDCA counterparty. However, even though native token pools can be listed successfully, the fee collection logic for the initial locked LP position treats all tokens like ERC20s, making fee claims impossible for pools with native tokens.

Root Cause

In `SuperDCAListing.sol:317-318` the fee collection logic assumes both pool currencies are ERC20 tokens and calls `IERC20(Currency.unwrap(token)).balanceOf()` to get the before and after balances to calculate fees, but Uniswap v4 represents native ETH as `address(0)` which will revert when treated as an ERC20.

https://github.com/sherlock-audit/2025-09-super-dca/blob/main/super-dca-gauge/src/SuperDCAListing.sol#L304-L324

If either token0 or token1 is a native token like ETH, this will translate to `IERC20(Currency.unwrap(address(0)).balanceOf(recipient);` which will fail, causing fees for the initial position to be permanently lost

Internal Pre-conditions

1. Pool needs to be created and listed with native ETH paired with DCA token
2. Admin needs to attempt fee collection from ETH-paired position

External Pre-conditions

1. None

Attack Path

1. User creates DCA/ETH pool position and lists token through SuperDCAListing
2. Pool generates trading fees over time in both DCA tokens and ETH
3. Admin calls `collectFees()` to retrieve accumulated fees from position
4. Function attempts to call `IERC20(address(0)).balanceOf(recipient)` for ETH
5. Transaction reverts due to invalid function call on address(0)
6. Fees remain permanently locked in position, unable to be collected

Impact

The protocol suffers permanent loss of accumulated trading fees for all ETH-paired pools, as the fee collection mechanism becomes completely non-functional for these positions.

ETH pairs are typically high-volume so the inability to collect fees from these positions represents a significant ongoing revenue loss for the protocol, especially if the initial position contains a large amount of liquidity.

PoC

Skip

Mitigation

Consider implementing native ETH detection and adjusting the before and after balance queries:

1. Check if `Currency.unwrap()` returns `address(0)` for ETH or native chain token
2. Use `recipient.balance` for ETH balance queries instead of ERC20 calls

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/Super-DCA-Tech/super-dca-gauge/pull/39

Issue H-4: Bucket rewards will be wiped by stake/unstake before accrueRewards, lastRewardIndex resets without settling rewards, accrueRewards delta becomes 0

Source: https://github.com/sherlock-audit/2025-09-super-dca-judging/issues/1065

Found by

0x00T1, 0xAadi, 0xAdwa, 0xB4nkz, 0xCrypt0nite, 0xCuru, 0xDemon, 0xDjango, 0xHexed, 0xImmortan, 0xaxaxa, 0xleo, 0xlookman, 0xpetern, 0xsolisec, 0xv1bh4, 0xzen, 33audits, 4Nescient, 8olidity, Aamirusmani1552, Al-Qa-qa, Almanax, Arav, Artur, BADROBINX, BengalCatBalu, Bobai23, Boy2000, BoyD, Chonkov, CodexBugmeNot, DemiGods, DuoSec, FonDevs, HeckerTrieuTien, Hunter, Hurricane, Icon_0x, Ironsidesec, IzuMan, JeRRy0422, JohnTPark24, JohnWeb3, Josh4324, JuggerNaut, KiroBrejka, LonWof-Demon, MysteryAuditor, NHristov, Orhukl, OxSath404, Phaethon, R, Ragnarok, Razkky, SMB62, SOPROBRO, S_722, Sa1ntRobi, ScarletFir, Siiiisivan, SuperDevFavour, Whiterabbit, WillyCode20, Y4nhu1, Yaneca_b, ZeroEx, Ziusz, alexbabits, algiz, aman, ami, axelot, bam0x7, bbl4de, cholakovvv, dani3l526, deadmanwalking, denys_sosnovskyi, derastephh, djshaneden, drdee, farman1094, fullstop, futureHack, gneiss, grigorovv17, harry, heavyw8t, hjo, illoy_sci, itsRavin, ivanalexandur, jah, jayjoshix, jo13, kangaroo, kazan, ke1caM, kimnoic, kimonic, maigadoh, makeWeb3safe, marcosecure0x, merlin, mingzoox, ni8mare, nonso72, oct0pwn, omeiza, oot2k, patitonar, peazzycole, pindarev, pollersan, prosper, proxima_centuri, pv, r1ver, rashmor, resosiloris, rsam_eth, sedare, shiazinho, shieldrey, silver_eth, slavina, softdev0323, soloking, sourav_DEV, surenyan-oks, taticuvostru, techOptimizor, theholymarvycodes, thekmj, theweb3mechanic, tinnohofficial, tobi0x18, typicalHuman, ubl4nk, udo, v10g1, v_2110, vinica_boy, vivekd, volleyking, whitehair0330, wickie, x0t0wt1w, xxiv, yeahChibyke, zcai

Summary

The contract resets a token bucket's lastRewardIndex in stake / unstake without first settling pending rewards, this will cause a complete loss of accrued rewards for stakers as any user can call stake/unstake right before accrueReward and this will zero the bucket's delta.

Root Cause

Inside SuperDCAStaking::accrueReward we subtract the info.lastRewardIndex from rewardIndex, however inside stake/unstake we set lastRewardIndex == rewardIndex:

```
function unstake(address token, uint256 amount) external override {
    // Validate amount is non-zero and available
```

```
    if (amount == 0) revert SuperDCAStaking__ZeroAmount();

    // Check both token bucket and user balances are sufficient
    TokenRewardInfo storage info = tokenRewardInfoOf[token];
    if (info.stakedAmount < amount) revert SuperDCAStaking__InsufficientBalance();
    if (userStakes[msg.sender][token] < amount) revert
    ↪  SuperDCAStaking__InsufficientBalance();

    // Update global reward index to current time
    _updateRewardIndex();

    // Update token bucket accounting and user stakes
    info.stakedAmount -= amount;
@>    info.lastRewardIndex = rewardIndex;

    totalStakedAmount -= amount;
    userStakes[msg.sender][token] -= amount;

    // Remove token from user's set if balance reaches zero
    if (userStakes[msg.sender][token] == 0) {
        userTokenSet[msg.sender].remove(token);
    }

    // Transfer SuperDCA tokens back to user
    IERC20(DCA_TOKEN).transfer(msg.sender, amount);
    emit Unstaked(token, msg.sender, amount);
}
```

If stake/unstake were to be called just before accrueRewards the system ”forgets” about past rewards and can make the accrueRewards return zero leading to 100% lost rewards for users.

Internal Pre-conditions

Somebody called stake unstake before the hook _beforeAddLiquidity calls _handleDistributionAndSettlement

External Pre-conditions

-

Attack Path

1. Lets say lastIndex is 50, Index now will be 100, user or attacker calls stake/unstake just before _beforeAddLiquidity is triggered.

2. lastIndex becomes 100
3. _beforeAddLiquidity calls into accrueReward, because delta is `uint256 delta = rewardIndex - info.lastRewardIndex;` and stake/unstake was called before than, delta results is a small number or even 0 so basically no rewards

Impact

High, the system forgets about past rewards, users will lose up to 100% of their rewards.

PoC

Inside SuperDCAStaking.t.sol::TokenRewardInfos, copy paste this test:

```
function test_PoC_UnstakeBeforeAccrue_WipesBucketDelta() public {
    // Set up a single bucket stake
    vm.prank(user);
    staking.stake(tokenA, 100e18);

    // Let rewards accrue
    uint256 start = staking.lastMinted();
    uint256 secs = 100;
    vm.warp(start + secs);

    //what should be paid
    uint256 expectedMint = secs * rate;
    assertGt(expectedMint, 0, "sanity");

    // Unstake before accrue resets bucket lastRewardIndex and wipes  delta
    vm.prank(user);
    staking.unstake(tokenA, 1);

    // Accrue now => pays 0 due to wiped delta
    vm.prank(gauge);
    uint256 paid = staking.accrueReward(tokenA);
    assertEq(paid, 0, "pending bucket rewards wiped by unstake reset");
}
```

Run with `forge test --match-test test_PoC_UnstakeBeforeAccrue_WipesBucketDelta -vvvv`

Console.logs:

```
  ← [Return] 0
   [0] VM::assertEq(0, 0, "pending bucket rewards wiped by unstake reset")
  ↪  [staticcall]
       ← [Return]
   ← [Stop]
```

Mitigation

Make the contract remember the passed rewards even after stake unstake has been called

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/Super-DCA-Tech/super-dca-gauge/pull/41

Issue M-1: Unfair distribution of rewards for LPs

Source: https://github.com/sherlock-audit/2025-09-super-dca-judging/issues/489

Found by

patitonar, vinica_boy

Summary

The pool's reward distribution occurs during each liquidity operation through the beforeAddLiquidity and beforeRemoveLiquidity hooks. The Uniswap pool's donate mechanism is used, which distributes rewards to in-range LPs at the current slot0.tick. The problem with this design is that rewards accrued over a certain period are distributed to the current in-range position, regardless of which positions provided active liquidity during the accrual period. This can be exploited by an attacker who performs the following steps (can be done in a single transaction):

1. Some time has passed and rewards have been accumulated, but not distributed.
2. Attacker makes a swap to move the tick to a predefined tick where their position provides liquidity in a small adjacent tick range.
3. Triggers the reward distribution mechanism by collecting fees (e.g., modifying liquidity with a 0 liquidity delta, which triggers the beforeRemoveLiquidity hook).
4. His position would accrue most of the fees because of the super tight range it has provided liquidity.
5. Swaps back to move the tick to its original value. The attacker incurs gas fees and swap fees, but depending on the accrued rewards, this can be profitable. Even if not profitable, this would still lead to steal of rewards from other honest LPs.

In a normal workflow scenario it would look like this - rewards accrue over a certain time period without any liquidity operations and numerous swaps move the tick in range of tick t0 to tick t1. All LPs that have provided active liquidity during that period should be rewarded, but if there is a certain market movement and price/tick move to a different tick t2 and liquidity operation is triggered at that point, only positions which have tick t2 in range would accrue rewards for all period since the last distribution.

Root Cause

The main problem is that the protocol cannot account for which LPs have provided actual in-range liquidity during reward accrual and distributes reward when there is an liquidity operation.

handleDistributionAndSettlement()

Internal Pre-conditions

N/A

External Pre-conditions

N/A

Attack Path

Shared in the Summary section.

Impact

Steal of LPs rewards.

PoC

N/A

Mitigation

Non-trivial mitigation as we need a way to account for the active LP position which deserve part of the accumulated rewards. One option would be to lock LPs positions and distribute rewards on a fixed time epochs instead of distributing rewards on liquidity operations. This would allow to verify which positions have actually provided in-range liquidity for that epoch and distribute rewards based on that.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/Super-DCA-Tech/super-dca-gauge/pull/52

Issue M-2: First Depositor can flash loan the protocol too artificially increase the token rewards without incurring any risk

Source: https://github.com/sherlock-audit/2025-09-super-dca-judging/issues/852

Found by

0xBoraichoT, 0xaxaxa, 0xleo, 0xpetern, 0xsolisec, 33audits, 4Nescient, 8olidity, Almanax, Bbash, Bobai23, BoyD, Chonkov, CodexBugmeNot, Divine_Dragon, DuoSec, EQUATION, HeckerTrieuTien, Hurricane, Immanux, Ironsidesec, JeRRy0422, JohnTPark24, MysteryAuditor, Opecon, Orhukl, OxSath404, R, Razkky, SOPROBRO, S_722, SuperDevFavour, TAdev0, TECHFUND-inc, TsvetanLobutov, WillyCode20, Y4nhu1, ZeroEx, Ziusz, aman, ami, axelot, bbl4de, dani3l526, denys_sosnovskyi, drdee, engineer, ezsia, frankauditcraft, fullstop, gneiss, grigorovv17, har0507, heavyw8t, itsRavin, jayjoshix, ke1caM, kimnoic, kimonic, lucky-gru, maigadoh, makeWeb3safe, merlin, namx05, nonso72, peazzycole, pollersan, proofvoid, r1ver, resosiloris, sakibcy, silver_eth, songyuqi, the_haritz, theweb3mechanic, tobi0x18, vinica_boy, vivekd, volleyking, whitehair0330, wickie, yeahChibyke, zcai

Summary

When a user stakes SuperDCATokens , the timestamp is updated to prevent a flashloan attack however when the first user stake the timestamp update is not updated due to the fact that the timestamp is skipped when totalStakedAmount is 0 (as seen in the code snippet below) and this is 0 during the time when the first depositor makes an initial deposit.

```
/**
 * @notice Updates the global reward index based on elapsed time and total staked
 ↪  amount.
 * @dev The 1e18 scaling factor provides mathematical precision for fractional
 ↪  rewards.
 */
function _updateRewardIndex() internal {
    // Return early if no stakes exist or no time has passed
    if (totalStakedAmount == 0) return;
    uint256 elapsed = block.timestamp - lastMinted;
    if (elapsed == 0) return;

    // Calculate mint amount based on elapsed time and mint rate
    uint256 mintAmount = elapsed * mintRate;

    // Update global index: previous_index + (mint_amount * 1e18 / total_staked)
    rewardIndex += Math.mulDiv(mintAmount, 1e18, totalStakedAmount);
```

```
    lastMinted = block.timestamp;
    emit RewardIndexUpdated(rewardIndex);
}
```

This allows a user to execute a flash loan to borrow DCATokens, stake them in the contract, claim rewards, and repay the flash loan–all within a single transaction.

Root Cause

The `_updateRewardIndex()` function is not fully executed when `totalStakedAmount == 0`. Let's walk through what happens in the `stake()` function:

```
function stake(address token, uint256 amount) external override {
    ...

    // Update global reward index to current time
    _updateRewardIndex();

    // Transfer SuperDCA tokens from user to contract
    IERC20(DCA_TOKEN).transferFrom(msg.sender, address(this), amount);

    // Update token bucket accounting and user stakes
    TokenRewardInfo storage info = tokenRewardInfoOf[token];
    info.stakedAmount += amount;
    info.lastRewardIndex = rewardIndex;

    totalStakedAmount += amount;
    userStakes[msg.sender][token] += amount;

    // Add token to user's active token set if new
    userTokenSet[msg.sender].add(token);

    emit Staked(token, msg.sender, amount);
}
```

Notice that `_updateRewardIndex()` is called **before** `totalStakedAmount` is updated. So, when the **first user stakes**, `totalStakedAmount` is still `0`, causing `_updateRewardIndex()` to **exit early** without updating the `lastMinted` timestamp.

This means:

- The user's stake is registered.
- The reward index for the token is set to the current global `rewardIndex`.
- However, `lastMinted` **is still the contract deployment timestamp** (or the last updated time, if any), and not the time of the stake.

⛒ Vulnerability

A malicious user can exploit this as follows:

1. Use a **flash loan** to borrow DCA tokens.
2. **Stake** those tokens (triggering `_updateRewardIndex()`, which exits early).
3. Trigger **reward accrual** via the `gauge` contract.
4. The `gauge` calls `accrueReward()`, which in turn calls `_updateRewardIndex()` again:

```
function _updateRewardIndex() internal {
    if (totalStakedAmount == 0) return; // Skipped during first stake
    uint256 elapsed = block.timestamp - lastMinted;
    if (elapsed == 0) return;

    uint256 mintAmount = elapsed * mintRate;
    rewardIndex += Math.mulDiv(mintAmount, 1e18, totalStakedAmount);
    lastMinted = block.timestamp;

    emit RewardIndexUpdated(rewardIndex);
}
```

At this point:

- `totalStakedAmount` is **non-zero**, so the function proceeds.
- `elapsed` includes the full time since contract deployment (or last mint), since `lastMinted` was never updated earlier.
- As a result, the `rewardIndex` increases significantly.

Then, in `accrueReward()`:

```
function accrueReward(address token) external override onlyGauge returns (uint256
↪   rewardAmount) {
    _updateRewardIndex();

    TokenRewardInfo storage info = tokenRewardInfoOf[token];
    if (info.stakedAmount == 0) return 0;

    uint256 delta = rewardIndex - info.lastRewardIndex;
    if (delta == 0) return 0;

    rewardAmount = Math.mulDiv(info.stakedAmount, delta, 1e18);
    info.lastRewardIndex = rewardIndex;

    return rewardAmount;
}
```

- A **non-zero delta** is calculated.

- The user's pool receives **rewards for elapsed time they were not actually staked**, since the stake timestamp was never correctly recorded.

Finally, the user repays the flash loan by unstaking– all within a **single transaction** – while extracting DCA token rewards they didn't earn.

Internal Pre-conditions

totalStakedAmount should equal 0.

External Pre-conditions

A pool should be deployed with listed token and DCA token with pool hook set to Gauge contract.

Attack Path

1. Michael has just deployed and initialised a USDc-DAC pool, with the hook contract being the guage contract.
2. Michael has just listed USDc token via the SuperDCAListing.sol contract which passed succesfully, and add liquidity to the pool.
3. Michael then takes a flashloan of 100e18 SuperDCAToken, which he stakes in the SuperDCAStaking contract by calling the stake function [`stake(address token, uint256 amount) external override`] with the USDc token and amount of 100e18.
4. Michael has now staked he then performs a swap operation in the pool to trigger the accrueReward call via the SuperGauge contract.
5. The stakingContract mints rewards to the pool.
6. Michael unstakes his position and proceeds to pay the flashloan.
7. Michael can claim SuperDCAToken by removing liquidity he added during step 2.

Impact

High, rewards are artificially minted.

PoC

No response

Mitigation

No response

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/Super-DCA-Tech/super-dca-gauge/pull/48

Issue M-3: Manager can retroactively apply new rate to past time, misallocating emissions - Invariant Broken

Source: https://github.com/sherlock-audit/2025-09-super-dca-judging/issues/928

Found by

0xB4nkz, 0xCrypt0nite, 0xDjango, 0xaxaxa, 0xheartcode, 0xlookman, 0xpetern, Aamirusmani1552, Al-Qa-qa, Bbash, BengalCatBalu, Boy2000, BoyD, DemiGods, DuoSec, HeckerTrieuTien, Ironsidesec, JeRRy0422, JuggerNaut, LonWof-Demon, NHristov, Ollam, R, Sir_Shades, VCeb, Victor_TheOracle, Xmanuel, alekso91, cholakovvv, denys_sosnovskyi, engineer, farman1094, har0507, heavyw8t, itsRavin, jayjoshix, ke1caM, khaye26, maigadoh, merlin, ni8mare, peazzycole, r1ver, shieldrey, silver_eth, soloking, tinnohofficial, tobi0x18, vinica_boy, vivekd, whitehair0330, x0t0wt1w

Summary

The staking system is supposed to guarantee:

- Exact increment formula:
  - On any call that triggers `_updateRewardIndex()` with `elapsed = block.timestamp - lastMinted` and totalStakedAmount > 0:
    - rewardIndex increases by `Math.mulDiv(elapsed * mintRate, 1e18, totalStakedAmount)` **exactly**.

But in `SuperDCAStaking.sol`, `setMintRate()` overwrites `mintRate` immediately without first applying rewards for the elapsed time. This breaks the invariant: the next reward update uses the **new** rate for the **past** interval, instead of the old rate. A manager (or attacker with control) can backdate a rate change and distort emissions.

Root Cause

In `setMintRate()` (around line 171 in `SuperDCAStaking.sol`):

```
mintRate = newMintRate; //  overwrites before old rewards are applied
```

Since `_updateRewardIndex()` isn't called first, the entire `elapsed = now - lastMinted` window gets charged at the *new* rate.

Internal Pre-conditions

1. `totalStakedAmount > 0`.

2. Time has passed since `lastMinted`.
3. Authorized address calls `setMintRate(newRate)` before any update.

External Pre-conditions

- None.

Attack Path

1. Stakers are earning at rate 10/sec.
2. 1000 seconds pass, but no update occurs.
3. Manager calls `setMintRate(1000)`.
4. On next accrual, `_updateRewardIndex()` applies the whole 1000 seconds at 1000/sec, not 10/sec.

Impact

- The intended invariant ("past time is priced at the old rate") is broken.
- Rewards are overpaid or underpaid depending on rate change.
- This can be abused by a malicious manager, or simply distort rewards if done accidentally.

PoC

In the above example,

- Should mint: `1000 * 10 = 10,000`.
- Actually mints: `1000 * 1000 = 1,000,000`.

Mitigation

Apply old rewards first, then update the rate:

```
_updateRewardIndex();  //  apply old rate to past interval
mintRate = newMintRate;
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/Super-DCA-Tech/super-dca-gauge/pull/46

Issue M-4: System underpays cashback on BNB: hard-coded “USDC = 6 decimals” causes 1e12× underpayment when USDC is 18-dec

Source: https://github.com/sherlock-audit/2025-09-super-dca-judging/issues/1045

Found by

0x11singh99, 0xCrypt0nite, 4Nescient, Aamirusmani1552, Al-Qa-qa, AlexCzm, AuditorPraise, BADROBINX, BengalCatBalu, Boy2000, Kirkeelee, PASCAL, Pro_King, SMB62, Sa1ntRobi, SarveshLimaye, VCeb, Xmanuel, ZeroEx, bXiv, dani3l526, heavyw8t, jayjoshix, kangaroo, lucky-gru, rsam_eth, tobi0x18

Summary

Root cause: Cashback math assumes USDC has **6 decimals** and converts from 1e18 precision by dividing / 1e12. **Impact:** On BNB Chain, where the widely used USDC is **18-dec**, every cashback claim is scaled down by 1e12 (orders-of-magnitude underpayment). **Affected party:** All users claiming cashback on chains where **USDC ≠ 6** decimals (notably BNB). **Attack path:** No attacker required; the protocol miscalculates payouts deterministically upon claim.

Root Cause

In `SuperDCACashback.sol`, amounts computed in 1e18 precision are converted to “USDC precision” via a fixed divider:

```
function _convertToUSDCPrecision(uint256 amount) internal pure returns (uint256) {
    return amount / 1e12; // assumes USDC has 6 decimals
}
```

Comments and storage semantics also bake in “USDC (6 decimals)”. This breaks on chains where the configured USDC token uses **18 decimals** (e.g., BNB), producing amounts 1e12× too small.

Internal Pre-conditions

1. Cashback math internally uses **1e18** precision for rates/amounts.
2. The contract calls `_convertToUSDCPrecision(amount)` which divides by **1e12** unconditionally.
3. USDC token address is set to a token whose **decimals()** can differ from 6.

External Pre-conditions

1. Deployment targets a chain where the configured USDC has **18 decimals** (BNB's common USDC).
2. Users run trades and later claim cashback.

Attack Path

1. A user completes eligible epochs and calls `claim…()` on BNB.
2. The contract computes the correct 1e18-scaled amount (say, `1e18` for "1 token").
3. It divides by `1e12`, yielding `1e6` base units.
4. Because `the token has 18 decimals`, **1e6** base units = **1e-12** tokens, not 1 token → the user is underpaid by **1e12×**.

Impact

High: Users on BNB receive 1e12× less cashback than owed (direct, systemic loss of user funds). This is chain-wide and persists until fixed.

Concrete Example (BNB)

- Internal calculation says the user earned **1e18** "USDC units" (meant to represent 1 full token).
- `_convertToUSDCPrecision` returns `1e18 / 1e12 = 1e6`.
- On an **18-dec** token, `1e6` base units = **0.000000000001** tokens, not `1.0`.
- Underpayment factor: **1,000,000,000,000×**.

PoC

No response

Mitigation

Make conversion token-decimal-aware and remove the 6-dec assumption:

```
import {IERC20Metadata} from
↪  "openzeppelin-contracts/contracts/token/ERC20/extensions/IERC20Metadata.sol";
import {Math} from "openzeppelin-contracts/contracts/utils/math/Math.sol";
IERC20 public USDC;
uint8 public immutable usdcDecimals;
```

```
constructor(address _usdc, /*...*/) {
    USDC = IERC20(_usdc);
    usdcDecimals = IERC20Metadata(_usdc).decimals();
    // ...
}

// Convert from 1e18 internal precision to token decimals exactly
function _toTokenDecimals(uint256 amt1e18) internal view returns (uint256) {
    return Math.mulDiv(amt1e18, 10 ** usdcDecimals, 1e18);
}
```

Replace all calls to `_convertToUSDCPrecision` with `_toTokenDecimals`. Update comments/storage semantics to refer to “token decimals”, not “USDC(6)”.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/Super-DCA-Tech/super-dca-cashback/pull/26

Disclaimers

Sherlock does not provide guarantees nor warranties relating to the security of the project.

Usage of all smart contract software is at the respective users' sole risk and is the users' responsibility.