Three Sigma

Code Audit

DEEP3 LABS

Deep3 Labs On-chain AI Infrastructure

Disclaimer

Code Audit

Deep3 Labs On-chain AI Infrastructure

Disclaimer

The ensuing audit offers no assertions or assurances about the code's security. It cannot be deemed an adequate judgment of the contract's correctness on its own. The authors of this audit present it solely as an informational exercise, reporting the thorough research involved in the secure development of the intended contracts, and make no material claims or guarantees regarding the contract's post-deployment operation. The authors of this report disclaim all liability for all kinds of potential consequences of the contract's deployment or use. Due to the possibility of human error occurring during the code's manual review process, we advise the client team to commission several independent audits in addition to a public bug bounty program.

Table of Contents

Code Audit

Deep3 Labs On-chain AI Infrastructure

Table of Contents

Summary

Code Audit

Deep3 Labs On-chain AI Infrastructure

Summary

Three Sigma audited Deep3 in a 3 person week engagement. The audit was conducted from 07/04/2025 to 11/04/2025.

Protocol Description

Deep3 Labs is building the foundational AI infrastructure for Web3.0—enabling smart contracts, dApps, and blockchain platforms to integrate powerful on-chain AI with just a few lines of code. Through maturity-adjusted staking, modular governance tokens, and role-based access controls, the Deep3 protocol allows developers and users to deploy, govern, and benefit from AI models directly on-chain.

Scope

Code Audit

Deep3 Labs On-chain AI Infrastructure

Scope

Filepath	nSLOC
src/AccessControl3DL.sol	127
src/D3LVesting.sol	115
src/GD3LToken.sol	39
src/MaturityAdjustedStaking.sol	357
src/PD3LToken.sol	42
src/UD3LToken.sol	30
src/VotingUnitsRegistry.sol	28
Total	**738**

Methodology

Code Audit

Deep3 Labs On-chain AI Infrastructure

Methodology

To begin, we reasoned meticulously about the contract's business logic, checking security-critical features to ensure that there were no gaps in the business logic and/or inconsistencies between the aforementioned logic and the implementation. Second, we thoroughly examined the code for known security flaws and attack vectors. Finally, we discussed the most catastrophic situations with the team and reasoned backwards to ensure they are not reachable in any unintentional form.

Taxonomy

In this audit, we classify findings based on Immunefi's Vulnerability Severity Classification System (v2.3) as a guideline. The final classification considers both the potential impact of an issue, as defined in the referenced system, and its likelihood of being exploited. The following table summarizes the general expected classification according to impact and likelihood; however, each issue will be evaluated on a case-by-case basis and may not strictly follow it.

Impact / Likelihood	LOW	MEDIUM	HIGH
NONE	None		
LOW	Low		
MEDIUM	Low	Medium	Medium
HIGH	Medium	High	High
CRITICAL	High	Critical	Critical

Project Dashboard

Code Audit

Deep3 Labs On-chain AI Infrastructure

Project Dashboard

Application Summary

Name	Deep3 Labs
Commit	20a484ebcae74b1c63dbdf77962855d669132507
Repository	https://github.com/deep3labs/contracts
Language	Solidity
Platform	Ethereum

Engagement Summary

Timeline	07/04/2025 to 11/04/2025
№ of Auditors	3
Review Time	3 person weeks

Vulnerability Summary

Issue Classification	Found	Addressed	Acknowledged
Critical	1	1	0
High	5	5	0
Medium	3	3	0
Low	2	2	0
None	5	5	0

Category Breakdown

Suggestion	5
Documentation	0
Bug	11
Optimization	0
Good Code Practices	0

Risk Section

Code Audit

Deep3 Labs On-chain AI Infrastructure

Risk Section

No risks identified.

Findings

Code Audit

Deep3 Labs On-chain AI Infrastructure

Findings

3S-Deep3 Labs-C01

Missing checkpoint validation in **`MaturityAdjustedStaking::_processRewards`** allows infinite reward claiming

Id	3S-Deep3 Labs-C01
Classification	Critical
Impact	Critical
Likelihood	High
Category	Bug
Status	Addressed in #437db43.

Description

In the **`MaturityAdjustedStaking`** contract, the **`_processRewards`** function contains a critical vulnerability that allows users to claim rewards multiple times for the same time period by rolling back the current checkpoint.

The **`_processRewards`** function is responsible for calculating and processing rewards for a staked position up to a specified checkpoint. However, it lacks a crucial validation check to ensure that the **`toCheckpoint`** parameter is greater than or equal to the position's **`lastProcessedCheckpoint`**:

```
function _processRewards(uint256 tokenId, uint256 toCheckpoint)
    internal
    returns (uint256 rewards, uint256 lastProcessedCheckpoint, uint256
lastProcessedTimestamp)
{
    // Missing validation for toCheckpoint
    StakeInfo storage _position = _positions[tokenId];
    (rewards, lastProcessedTimestamp) = _calculateRewards(_position,
toCheckpoint);
```

```
    // update last processed timestamp
    _position.lastProcessedTime = lastProcessedTimestamp;
    // If we're not at the latest checkpoint, move to the next one
    // Otherwise, stay at the current checkpoint
    lastProcessedCheckpoint = toCheckpoint < checkpointCount() - 1 ?
toCheckpoint + 1 : toCheckpoint;
    // Update the position's lastProcessedCheckpoint
    _position.lastProcessedCheckpoint = uint32(lastProcessedCheckpoint);
    return (rewards, lastProcessedCheckpoint, lastProcessedTimestamp);
}
```

This lack of validation creates a critical exploit where an attacker can roll back their position's checkpoint and timestamp, allowing them to claim rewards multiple times for the same period.

Steps to Reproduce:

This PoC demonstrates the issue. Here's how the attack works:

1. Alice stakes tokens and later claims rewards, updating her position's **`lastProcessedCheckpoint`** to the current checkpoint (2) and **`lastProcessedTime`** to the current timestamp.

2. Alice then calls **`claim(tokenId, 0)`**, which processes rewards up to checkpoint 0:

- The **`_processRewards`** function updates **`lastProcessedCheckpoint`** to 1 (since it's not the latest checkpoint)

3. Alice calls **`claim(tokenId, 1)`**, which processes rewards up to checkpoint 1:

- The case **`fromCheckpoint == toCheckpoint`** in **`_calculateRewards`** is triggered

- **`lastProcessedTime`** is rolled back to an earlier timestamp (day 2)

4. Alice can now call **`claim(tokenId)`** again to claim all rewards from the rolled-back timestamp to the present, effectively claiming some rewards multiple times.

5. This process can be repeated indefinitely, allowing Alice to claim infinite rewards.

Recommendation

Add a validation check at the beginning of the **`_processRewards`** function to ensure that **`toCheckpoint`** is greater than or equal to the position's current **`lastProcessedCheckpoint`**:

```
function _processRewards(uint256 tokenId, uint256 toCheckpoint)
    internal
    returns (uint256 rewards, uint256 lastProcessedCheckpoint, uint256
lastProcessedTimestamp)
{
    StakeInfo storage _position = _positions[tokenId];
+   // Prevent checkpoint rollback
+   if (toCheckpoint < _position.lastProcessedCheckpoint) {
+       revert InvalidCheckpointOrder();
+   }

    (rewards, lastProcessedTimestamp) = _calculateRewards(_position,
toCheckpoint);
    // update last processed timestamp
    _position.lastProcessedTime = lastProcessedTimestamp;
    // If we're not at the latest checkpoint, move to the next one
    // Otherwise, stay at the current checkpoint
    lastProcessedCheckpoint = toCheckpoint < checkpointCount() - 1 ?
toCheckpoint + 1 : toCheckpoint;
    // Update the position's lastProcessedCheckpoint
    _position.lastProcessedCheckpoint = uint32(lastProcessedCheckpoint);
    return (rewards, lastProcessedCheckpoint, lastProcessedTimestamp);
}
```

Additionally, a new error should be defined in the contract:

```
error InvalidCheckpointOrder();
```

This fix ensures that users can only process rewards forward in time, preventing the rollback exploit that enables infinite reward claiming.

3S-Deep3 Labs-H01

Incorrect reward calculation scaling leads to 1,000,000x higher rewards than intended

Id	3S-Deep3 Labs-H01
Classification	High
Impact	High
Likelihood	High
Category	Bug
Status	Addressed in #5eff52a.

Description

There is a critical issue in the **`MaturityAdjustedStaking::_calculateRewards`** function where rewards are calculated incorrectly, resulting in users receiving 1,000,000 times more rewards than intended. This occurs because the reward calculation only divides by **`PRECISION`** once, when it should divide by **`PRECISION`** twice.

In the **`MaturityAdjustedStaking`** contract, the **`PRECISION`** constant is defined as **`1e6`** (1,000,000). This value is used for scaling in various calculations throughout the contract. The maturity multiplier is scaled by **`PRECISION`**, and the reward rate is also expected to be scaled by **`PRECISION`**.

The issue appears in two places within the **`_calculateRewards`** function:

1. When calculating rewards for a single checkpoint:

```
rewards = (
    _position.amount * _checkpoints[fromCheckpoint].rewardRatePerSecond *
(periodEnd - periodStart)
        * multiplier
) / PRECISION;
```

2. When calculating rewards across multiple checkpoints:

```
uint256 checkpointRewards = (
    _position.amount * _checkpoints[i].rewardRatePerSecond * (periodEnd -
periodStart) * multiplier
) / PRECISION;
```

In both cases, the calculation only divides by **`PRECISION`** once, but it should divide by **`PRECISION`** twice:

- Once to normalize the **`multiplier`** (which is scaled by **`PRECISION`**)

- Once to normalize the **`rewardRatePerSecond`** (which is also scaled by **`PRECISION`**)

This means that users are receiving **`PRECISION`** (1,000,000) times more rewards than intended. For example, if a user stakes 100 tokens for 1 day with a reward rate of 1,000 (0.001 tokens per second per staked token) and a multiplier of 1,000,000 (1.0x), they should receive:

- Expected: 100 * 0.001 * 86400 * 1.0 = 8.64 tokens

- Actual: 100 * 1,000 * 86400 * 1,000,000 / 1,000,000 = 8,640,000 tokens (1,000,000x higher)

Recommendation

The reward calculation should be modified to divide by **`PRECISION`** twice.

3S-Deep3 Labs-H02

Incorrect parameter order in **`PD3LToken::_update`** function leads to voting units transfer to wrong addresses

Id	3S-Deep3 Labs-H02
Classification	High
Impact	High
Likelihood	High
Category	Bug
Status	Addressed in #00d2c06.

Description

The **`PD3LToken::_update`** function has been implemented with an incorrect parameter signature compared to the OpenZeppelin ERC721 implementation it's overriding. This mismatch causes voting units to be transferred between the wrong addresses when tokens change ownership.

In the current implementation:

```
function _update(address from, uint256 tokenId, address newOwner) internal
virtual override returns (address) {
    address previousOwner = super._update(from, tokenId, newOwner);
    // Handle all transfers including minting (from=0) and burning (to=0)
    IVotingUnitsRegistry(votingUnitsRegistry).transferVotingUnits(from,
newOwner, _positions[tokenId].amount);
    return previousOwner;
}
```

The function signature is incorrect. In OpenZeppelin's ERC721 implementation, the correct signature is:

```
function _update(address to, uint256 tokenId, address auth) internal virtual
returns (address)
```

This means that when **`PD3LToken::_update`** calls **`super._update(from, tokenId, newOwner)`**, the parameters are being misinterpreted:

- **`from`** is being treated as the destination address (**`to`**)

- **`tokenId`** remains correctly as the token ID

- **`newOwner`** is being treated as the authorized address (**`auth`**)

The function then incorrectly calls **`transferVotingUnits(from, newOwner, _positions[tokenId].amount)`**, which transfers voting units from what it thinks is the sender to what it thinks is the receiver. However, due to the parameter mismatch:

1. The **`from`** parameter (which is actually the destination in the parent implementation) is incorrectly used as the source of voting units

2. The **`newOwner`** parameter (which is actually the authorization address in the parent implementation) is incorrectly used as the destination for voting units

This creates a critical issue where voting units are transferred in the wrong direction or between completely unrelated addresses.

Steps to Reproduce:

The PoC shows that when Alice stakes tokens, her voting units decrease by twice the staked amount:

1. Once for the legitimate transfer of **`gD3L`** tokens from Alice to the staking contract

2. A second time due to the incorrect parameter order in **`_update`**, causing an additional, unintended transfer of voting units to the zero-address.

Recommendation

Correct the function signature and implementation to match OpenZeppelin's ERC721 **`_update`** function:

```
/// @notice ERC721 hook to update voting units transfer on token transfers
/// @param to The address receiving the token
/// @param tokenId The ID of the token being transferred
/// @param auth The address authorized to make the transfer
/// @return The address of the previous owner
function _update(address to, uint256 tokenId, address auth) internal virtual
```

```
override returns (address) {
    address previousOwner = super._update(to, tokenId, auth);
    // Handle all transfers including minting (previousOwner=0) and burning
(to=0)

IVotingUnitsRegistry(votingUnitsRegistry).transferVotingUnits(previousOwner,
to, _positions[tokenId].amount);
    return previousOwner;
}
```

This correction ensures that voting units are properly transferred from the previous owner to the new owner when tokens change hands, maintaining the integrity of the voting system.

3S-Deep3 Labs-H03

Incorrect maturity multiplier calculation leads to reward inconsistency based on claim frequency

Id	3S-Deep3 Labs-H03
Classification	High
Impact	High
Likelihood	High
Category	Bug
Status	Addressed in #b681d7e.

Description

The **`MaturityAdjustedStaking`** contract incorrectly calculates rewards by using only the end-period multiplier instead of averaging the start and end multipliers over the reward period. This implementation directly contradicts the protocol's documented reward calculation formula, creating a significant economic vulnerability where users who claim more frequently receive fewer total rewards than those who claim less frequently.

According to the protocol's official documentation, rewards should be calculated as follows:

> For any period between inflection points (t1, t2):

> rewards = position_size × base_rate × Δt × (multiplier1 + multiplier2) / 2 where:

> - Δt = t2 - t1

> - multiplier1, multiplier2: Effective multipliers at t1, t2 respectively

However, the actual implementation in the contract only uses the multiplier at the end of the period:

1. When processing a single checkpoint:

```
uint256 multiplier = _calculateMaturityMultiplier(fromCheckpoint, _position,
periodEnd);
rewards = (
    _position.amount * _checkpoints[fromCheckpoint].rewardRatePerSecond *
```

```
(periodEnd - periodStart)
        * multiplier
) / PRECISION;
```

2. When processing multiple checkpoints:

```
uint256 multiplier = _calculateMaturityMultiplier(i, _position, periodEnd);
uint256 checkpointRewards = (
    _position.amount * _checkpoints[i].rewardRatePerSecond * (periodEnd -
periodStart) * multiplier
) / PRECISION;
rewards += checkpointRewards;
```

The documentation states that the multiplier should be the average of the start and end multipliers: **`(multiplier1 + multiplier2) / 2`**, but the implementation only uses the end multiplier (**`multiplier2`**).

This creates a discrepancy where the total rewards depend on how frequently a user claims. To illustrate with a concrete example:

Assume:

- **`rewardRatePerSecond = 1`**
- **`maxMultiplier = 2e6`** (2x)
- **`_position.startTime = 0`**
- **`_position.maturityDuration = 100`**
- **`_position.amount = 1`**

Scenario 1: User claims once at maturity (timestamp = 100)

- Multiplier at end = 2e6 (fully matured)
- Rewards = 1 * 1 * (100 - 0) * 2e6 / 1e6 = 200

Scenario 2: User claims at timestamp = 40, then at maturity (timestamp = 100)

- First claim:
  - Multiplier at end = 1.4e6 (40% matured)

- Rewards = 1 * 1 * (40 - 0) * 1.4e6 / 1e6 = 56

- Second claim:

- Multiplier at end = 2e6 (fully matured)

- Rewards = 1 * 1 * (100 - 40) * 2e6 / 1e6 = 120

- Total rewards = 56 + 120 = 176

The user in Scenario 2 receives 24 fewer reward tokens (12% less) than the user in Scenario 1, despite staking the same amount for the same duration. This creates a perverse incentive for users to minimize claim frequency, which contradicts the intended design of the protocol and could lead to unexpected economic behavior.

If the correct formula from the documentation were used, both scenarios would yield the same total rewards:

Using correct formula (multiplier1 + multiplier2) / 2:

- Scenario 1: 1 * 1 * (100 - 0) * (1e6 + 2e6) / (2 * 1e6) = 150

- Scenario 2:

- First claim: 1 * 1 * (40 - 0) * (1e6 + 1.4e6) / (2 * 1e6) = 48

- Second claim: 1 * 1 * (100 - 40) * (1.4e6 + 2e6) / (2 * 1e6) = 102

- Total: 48 + 102 = 150

Recommendation

Modify the reward calculation to use the average of the start and end multipliers for each period, as specified in the documentation. This ensures that rewards are consistent regardless of claim frequency.

```
// For single checkpoint case
uint256 multiplierStart = _calculateMaturityMultiplier(fromCheckpoint,
_position, periodStart);
uint256 multiplierEnd = _calculateMaturityMultiplier(fromCheckpoint,
_position, periodEnd);
rewards = (
    _position.amount * _checkpoints[fromCheckpoint].rewardRatePerSecond *
(periodEnd - periodStart)
        * (multiplierStart + multiplierEnd)
```

```
) / (2 * PRECISION);
// For multiple checkpoints case
uint256 multiplierStart = _calculateMaturityMultiplier(i, _position,
periodStart);
uint256 multiplierEnd = _calculateMaturityMultiplier(i, _position,
periodEnd);
uint256 checkpointRewards = (
    _position.amount * _checkpoints[i].rewardRatePerSecond * (periodEnd -
periodStart)
        * (multiplierStart + multiplierEnd)
) / (2 * PRECISION);
rewards += checkpointRewards;
```

This implementation ensures that users receive the same total rewards regardless of how frequently they claim, which aligns with the documented formula and the expected behavior of a maturity-based staking system.

3S-Deep3 Labs-H04

Vested but unclaimed tokens permanently locked after termination in **`D3LVesting::terminateVesting`**

Id	3S-Deep3 Labs-H04
Classification	High
Impact	High
Likelihood	High
Category	Bug
Status	Addressed in #56d39f2.

Description

The **`D3LVesting`** contract contains a design flaw that permanently locks all vested but unclaimed tokens when a vesting schedule is terminated. This affects employees who have earned tokens through their vesting period but haven't yet claimed them, resulting in complete loss of access to these assets despite having satisfied the vesting requirements.

The issue occurs in the **`terminateVesting`** function, which calculates both vested and unvested amounts, but only transfers the unvested tokens back to the admin while setting the schedule to inactive:

```
function terminateVesting(uint256 scheduleId) external onlyRole(ADMIN_ROLE)
returns (uint256) {
    VestingSchedule storage schedule = _vestingSchedules[scheduleId];
    if (schedule.beneficiary == address(0)) revert ScheduleDoesNotExist();
    if (!schedule.isActive) revert ScheduleAlreadyTerminated();

    uint256 vestedAmount = _calculateVestedAmount(schedule);
    uint256 unvestedAmount = schedule.totalAmount - vestedAmount -
schedule.claimedAmount;

    schedule.isActive = false;
    if (unvestedAmount > 0) {
        IERC20(schedule.token).safeTransfer(msg.sender, unvestedAmount);
```

```
    }

    return unvestedAmount;
}
```

The problem is compounded by the **claimVestedTokens** function, which prevents claims when a schedule is inactive:

```
function claimVestedTokens(uint256 scheduleId) external nonReentrant returns
(uint256 amount) {
    VestingSchedule storage schedule = _vestingSchedules[scheduleId];
    if (schedule.beneficiary == address(0)) revert ScheduleDoesNotExist();
    if (schedule.beneficiary != msg.sender) revert NotScheduleBeneficiary();
    if (!schedule.isActive) revert ScheduleIsTerminated();

    // Claim calculation and transfer logic
}
```

This creates a situation where:

1. An admin terminates a vesting schedule

2. The unvested tokens are returned to the admin

3. The schedule is marked as inactive

4. The beneficiary can no longer claim their vested tokens

5. The vested tokens remain locked in the contract with no withdrawal mechanism

Steps to Reproduce:

This PoC demonstrates this scenario, confirming that:

- 75% of tokens are correctly calculated as vested

- The admin receives only the unvested tokens

- The vested tokens remain in the contract

- The employee cannot claim these tokens

- The tokens are effectively locked, as no recovery mechanism exists

Recommendation

Modify the **`terminateVesting`** function to allow the beneficiary to claim their vested tokens even after termination. There are several ways to implement this:

1. Transfer vested tokens to the beneficiary during termination.

2. Alternatively, modify **`claimVestedTokens`** to allow claims even when the schedule is inactive

3S-Deep3 Labs-H05

Mature staking positions earn zero rewards forever due to time boundary cap in **`MaturityAdjustedStaking::_intervalEndTime`**

Id	3S-Deep3 Labs-H05
Classification	High
Impact	High
Likelihood	High
Category	Bug
Status	Addressed in #3d7a392.

Description

The **`MaturityAdjustedStaking`** contract fails to provide any rewards after staking positions reach maturity. According to the documentation, users should earn rewards at the maximum multiplier rate indefinitely after reaching maturity.

The root cause is in the **`MaturityAdjustedStaking::_intervalEndTime`** function, which always caps reward periods at maturity time, with no exception for post-maturity processing:

```
function _intervalEndTime(
    uint256 checkpointIndex,
    StakeInfo storage _position
) internal view returns (uint256 endTime) {
    uint256 maturityTime = _position.startTime + _position.maturityDuration;
    // Get the next checkpoint start time, if it exists, or current time
    if (checkpointIndex + 1 < _checkpoints.length) {
        endTime = _checkpoints[checkpointIndex + 1].timestamp;
    } else {
        endTime = block.timestamp;
    }
    // If the position matures before the checkpoint ends, use maturity time
as end time
    if (maturityTime < endTime) {
```

```
            endTime = maturityTime;  //@audit Always caps at maturity time
        }
    }
```

When a user calls **`MaturityAdjustedStaking::claim`**, the execution flows through several functions:

1. The external **`claim`** function calls internal **`_claim`**

2. **`_claim`** calls **`_processRewards`**

3. **`_processRewards`** calls **`_calculateRewards`**

The flawed path is in **`_processRewards`**, which updates the position's **`lastProcessedTime`** with the value returned from **`_calculateRewards`**:

```
function _processRewards(
    uint256 tokenId,
    uint256 toCheckpoint
) internal returns (
    uint256 rewards,
    uint256 lastProcessedCheckpoint,
    uint256 lastProcessedTimestamp
) {
    StakeInfo storage _position = _positions[tokenId];
    (rewards, lastProcessedTimestamp) = _calculateRewards(_position,
toCheckpoint);
    // update last processed timestamp
    _position.lastProcessedTime = lastProcessedTimestamp;
    // ...
}
```

Inside **`_calculateRewards`**, the flawed sequence happens when a position crosses maturity:

```
function _calculateRewards(
    StakeInfo storage _position,
    uint256 toCheckpoint
) internal view returns (uint256 rewards, uint256 lastProcessedTime) {
    // Track the last processed time locally
```

```
    uint256 currentProcessedTime = _position.lastProcessedTime;

    // Processing each checkpoint
    for (uint256 i = fromCheckpoint; i <= toCheckpoint; ) {
        periodStart = max(_intervalStartTime(i, _position),
currentProcessedTime);
        periodEnd = _intervalEndTime(i, _position);  // Will return maturity
time

        if (periodEnd > periodStart) {
            // Calculate rewards...
            currentProcessedTime = periodEnd;  // Set to maturity time
        }
        // ...
    }

    return (rewards, currentProcessedTime);  // Returns maturity time
}
```

When a position reaches maturity:

1. **`_intervalEndTime`** returns the maturity time

2. **`currentProcessedTime`** is set to maturity time

3. This maturity time is returned and stored as the position's **`lastProcessedTime`**

For all subsequent claims after maturity:

1. The initial **`periodStart`** equals the stored **`lastProcessedTime`** (maturity time)

2. **`periodEnd`** is also capped at maturity time

3. This results in **`periodEnd - periodStart = 0`**, meaning no duration for reward calculation

4. Consequently, zero rewards are earned for the entire post-maturity period

Steps to Reproduce:

This PoC confirms this behavior, showing that after a position reaches maturity:

1. The **`lastProcessedTime`** remains stuck at maturity time

2. No rewards are earned post-maturity

Recommendation

Modify the **`_intervalEndTime`** function to allow reward periods to extend beyond maturity when processing post-maturity rewards.

3S-Deep3 Labs-M01

Inconsistent function parameters between **`MaturityAdjustedStaking`** and its interface leads to potential integration issues

Id	3S-Deep3 Labs-M01
Classification	Medium
Impact	Medium
Likelihood	Medium
Category	Bug
Status	Addressed in #95c7e3b.

Description

The **`MaturityAdjustedStaking`** contract implements the **`IMaturityAdjustedStaking`** interface, which defines the functions **`setStakingParameters`** and **`claim`**. However, the parameters for these functions differ between the implementation and the interface, which can lead to integration issues and unexpected behavior when interacting with the contract.

In the **`IMaturityAdjustedStaking`** interface, the **`setStakingParameters`** function is defined as:

```
function setStakingParameters(
    uint256 cliffDuration,
    uint256 maturityDuration,
    uint256 maxMultiplier,
    uint256 rewardRatePerSecond
) external returns (uint256 checkpointIndex);
```

In the **`MaturityAdjustedStaking`** contract, the **`setStakingParameters`** function is implemented as:

```
function setStakingParameters(
    uint256 newRewardRate,
```

```
    uint256 newCliffDuration,
    uint256 newMaturityDuration,
    uint256 newMaxBonus
) external override onlyRole(PARAMETER_ADMIN_ROLE) returns (uint256
checkpointIndex);
```

Similarly, the **`claim`** function in the interface is defined as:

```
function claim(uint256 tokenId, uint256 endTime)
    external
    returns (uint256 rewards, uint256 lastProcessedCheckpoint, uint256
lastProcessedTime);
```

While in the **`MaturityAdjustedStaking`** contract, it is implemented as:

```
function claim(uint256 tokenId, uint256 toCheckpoint)
    public
    override
    isAuthorized(tokenId)
    nonReentrant
    returns (uint256 rewards, uint256 lastProcessedCheckpoint, uint256
lastProcessedTime);
```

Recommendation

Ensure that the function signatures in the **`MaturityAdjustedStaking`** contract match those defined in the **`IMaturityAdjustedStaking`** interface.

3S-Deep3 Labs-M02

Missing validation for **newMaxBonus** parameter leads to DoS of reward claims

Id	3S-Deep3 Labs-M02
Classification	Medium
Impact	High
Likelihood	Low
Category	Bug
Status	Addressed in #0f570c9.

Description

In the **MaturityAdjustedStaking::setStakingParameters** function, there is no validation to ensure that the **newMaxBonus** parameter is greater than or equal to **PRECISION** (1e6). This oversight can lead to a denial of service condition for the contract's reward claiming functionality.

The issue occurs because the **_calculateMaturityMultiplier** function, which is called during reward calculations, assumes that the maximum multiplier is always greater than or equal to **PRECISION**. When calculating the linear interpolation between **PRECISION** and the maximum multiplier, it performs the following calculation:

```
return PRECISION + ((cp.maxMultiplier - PRECISION) * elapsed) /
_position.maturityDuration;
```

If **cp.maxMultiplier** is less than **PRECISION**, this subtraction will underflow, causing the transaction to revert. This happens in the **_calculateRewards** function, which is called by **_claim** when users attempt to claim their rewards.

What makes this issue particularly problematic is that once a checkpoint with an invalid **maxMultiplier** is created, it becomes part of the permanent history in the **_checkpoints** array. Even if the parameter admin later corrects the value by setting a valid **maxMultiplier**, any position that spans the problematic checkpoint will still fail when attempting to claim rewards, as the calculation will still need to process the invalid checkpoint.

Recommendation

Add a validation check in the **`setStakingParameters`** function to ensure that **`newMaxBonus`** is always greater than or equal to **`PRECISION`**:

```
function setStakingParameters(
    uint256 newRewardRate,
    uint256 newCliffDuration,
    uint256 newMaturityDuration,
    uint256 newMaxBonus
) external override onlyRole(PARAMETER_ADMIN_ROLE) returns (uint256
checkpointIndex) {
+   if (newMaxBonus < PRECISION) revert InvalidMaxBonus();
    _currentCliffDuration = newCliffDuration;
    _currentMaturityDuration = newMaturityDuration;
    checkpointIndex = _createCheckpoint(newRewardRate, newMaxBonus);
}
```

Additionally, add the corresponding error to the interface:

```
error InvalidMaxBonus();
```

This validation will prevent the creation of checkpoints with invalid multiplier values that could break the reward claiming functionality.

3S-Deep3 Labs-M03

Lack of automatic reward claiming during token transfers leads to loss of rewards

Id	3S-Deep3 Labs-M03
Classification	Medium
Impact	High
Likelihood	Low
Category	Bug
Status	Addressed in #38b18c5.

Description

The **`MaturityAdjustedStaking`** contract implements a staking system where users can stake tokens and earn rewards over time. These staking positions are represented as NFTs (ERC721 tokens) that can be transferred between users. However, there is a critical issue in the token transfer mechanism.

When a staking position NFT is transferred from one user to another, the pending rewards are not automatically claimed for the original owner. Instead, the pending rewards remain associated with the position itself. This means that when the NFT is transferred, the new owner gains the ability to claim all accumulated rewards, including those earned during the previous owner's holding period.

The issue is located in the **`PD3LToken::_update`** function, which is called during token transfers. This function updates the voting units but does not claim pending rewards for the original owner before the transfer. The rewards claiming functionality is implemented in **`MaturityAdjustedStaking::_claim`**, but it's not automatically called during transfers.

Consider the following scenario:

1. Alice stakes 1000 tokens and receives a staking position NFT (tokenId = 1)

2. Over time, Alice's position accumulates 50 reward tokens that are claimable but not yet claimed

3. Alice transfers the NFT to Bob (via **`safeTransferFrom`** or similar ERC721 transfer function)

4. Bob now owns the staking position NFT

5. Bob calls **`MaturityAdjustedStaking::claim(1)`** and receives all 50 reward tokens that were earned during Alice's ownership period

6. Alice has lost her 50 reward tokens that she rightfully earned

This issue affects all transfers of staking position NFTs and can lead to significant loss of rewards for users who forget to claim their rewards before transferring their positions.

Recommendation

In the function **`PD3LToken::_update`**, if it is a transfer between addresses then the function **`_claim(uint256 tokenId)`** should be called at the beginning.

3S-Deep3 Labs-L01

Cross-contract reentrancy during **`_safeMint`** allows temporary double voting power

Id	3S-Deep3 Labs-L01
Classification	Low
Impact	Low
Likelihood	High
Category	Bug
Status	Addressed in #034e316.

Description

The **`stake`** function in the **`MaturityAdjustedStaking`** contract is responsible for creating a new staking position by minting a token and transferring the staked amount from the user to the contract. However, the **`_safeMint`** function is called before **`stakingToken.safeTransferFrom`**, which allows **`msg.sender`** to reenter during **`_safeMint`** and potentially interact with other contracts to vote, having temporarily doubled voting power. This is because the voting power is added to the user before the staked tokens are transferred, creating a window where the user has both the staked tokens and the voting power. Although the impact is limited by a correct implementation of OpenZeppelin Votes and its checkpoint functionality, the governance code is out of scope, and the usage of the votes cannot be validated by this audit.

The relevant code snippet is:

```
function stake(uint256 amount) external override nonReentrant returns
(uint256 tokenId) {
    if (amount == 0) revert ZeroStakeAmount();
    uint256 currentTime = block.timestamp;
    tokenId = _nextTokenId++;
    // Calculate effective start time (includes cliff period)
    uint256 effectiveStartTime = currentTime + _currentCliffDuration;
    // Store position info
```

```
    _positions[tokenId] = StakeInfo({
        amount: amount,
        startTime: effectiveStartTime,
        lastProcessedCheckpoint: uint32(_checkpoints.length - 1),
        lastProcessedTime: effectiveStartTime,
        maturityDuration: _currentMaturityDuration
    });
    _safeMint(_msgSender(), tokenId);
    stakingToken.safeTransferFrom(_msgSender(), address(this), amount);
    emit Staked(_msgSender(), _msgSender(), tokenId, amount,
effectiveStartTime);
}
```

Recommendation

To prevent reentrancy and ensure that the voting power is only added after the tokens are transferred, move the **`_safeMint`** call after **`stakingToken.safeTransferFrom`**. The updated function should be:

```
function stake(uint256 amount) external override nonReentrant returns
(uint256 tokenId) {
    if (amount == 0) revert ZeroStakeAmount();
    uint256 currentTime = block.timestamp;
    tokenId = _nextTokenId++;
    // Calculate effective start time (includes cliff period)
    uint256 effectiveStartTime = currentTime + _currentCliffDuration;
    // Store position info
    _positions[tokenId] = StakeInfo({
        amount: amount,
        startTime: effectiveStartTime,
        lastProcessedCheckpoint: uint32(_checkpoints.length - 1),
        lastProcessedTime: effectiveStartTime,
        maturityDuration: _currentMaturityDuration
    });
-   _safeMint(_msgSender(), tokenId);
    stakingToken.safeTransferFrom(_msgSender(), address(this), amount);
+   _safeMint(_msgSender(), tokenId);
    emit Staked(_msgSender(), _msgSender(), tokenId, amount,
effectiveStartTime);
```

```
}
```

3S-Deep3 Labs-L02

Incorrect parameter order in **`_calculateMaturityMultiplier`** function call leads to potential function unusability

Id	3S-Deep3 Labs-L02
Classification	Low
Impact	Low
Likelihood	High
Category	Bug
Status	Addressed in #5cf9d8f.

Description

The **`positionMultiplier`** function in the **`MaturityAdjustedStaking`** contract is intended to calculate the maturity multiplier for a staking position at a specific timestamp within a checkpoint. It calls the internal function **`_calculateMaturityMultiplier`** with the parameters **`tokenId`**, **`_positions[tokenId]`**, and **`timestamp`**. However, the **`_calculateMaturityMultiplier`** function expects parameters in the order of **`uint256 checkpointIndex`**, **`StakeInfo storage _position`**, and **`uint256 timestamp`**. This mismatch in parameter order can lead to incorrect calculations or reversion of the function call, rendering the **`positionMultiplier`** function unusable.

The relevant code snippet is:

```
function positionMultiplier(uint256 tokenId, uint256 checkpointIndex,
uint256 timestamp)
    external
    view
    override
    isToken(tokenId)
    isCheckpoint(checkpointIndex)
    returns (uint256)
{
    return _calculateMaturityMultiplier(tokenId, _positions[tokenId],
```

```
timestamp);
}
```

Recommendation

Correct the parameter order in the call to **`_calculateMaturityMultiplier`** within the **`positionMultiplier`** function to match the expected function signature.

3S-Deep3 Labs-N01

Redundant storage of vesting schedules leads to increased gas costs

Id	3S-Deep3 Labs-N01
Classification	None
Category	Suggestion
Status	Addressed in #b89bccc.

Description

The **`D3LVesting`** contract is designed to manage vesting schedules for tokens, allowing for the creation, claiming, and termination of vesting schedules. Within the contract, the **`createVestingSchedule`** function is responsible for setting up new vesting schedules. However, the contract redundantly stores vesting schedule data in both an array **`_schedules`** and a mapping **`_vestingSchedules`**. The array **`_schedules`** is used solely to determine the key for the mapping **`_vestingSchedules`**, which is the primary data structure accessed by other functions in the contract. This redundancy results in unnecessary storage operations, leading to increased gas costs.

Recommendation

To optimize gas usage, replace the **`_schedules`** array with a simple counter that tracks the number of vesting schedules created. This counter can be used as the key for the **`_vestingSchedules`** mapping, eliminating the need for duplicate storage. Alternatively, the **`_schedules`** array can be kept instead of **`_vestingSchedules`**, and the other functions be refactored accordingly.

3S-Deep3 Labs-N02

Redundant checkpoint validation in **`MaturityAdjustedStaking::claimableRewards`** leads to code duplication

Id	3S-Deep3 Labs-N02
Classification	None
Category	Suggestion
Status	Addressed in #975446e.

Description

The **`claimableRewards`** function in the **`MaturityAdjustedStaking`** contract is designed to calculate the unclaimed rewards for a staking position up to a specific checkpoint. However, the function currently duplicates the logic for validating the checkpoint index, which is already encapsulated in the **`isCheckpoint`** modifier. This redundancy not only leads to code duplication but also increases the risk of inconsistencies if the validation logic needs to be updated in the future.

The relevant code snippet is:

```
function claimableRewards(uint256 tokenId, uint256 checkpointIndex)
    external
    view
    override
    isToken(tokenId)
    returns (uint256 rewards, uint256 lastProcessedTime)
{
    // This function calculates rewards up to a specific checkpoint index
    if (checkpointIndex >= _checkpoints.length) {
        revert InvalidCheckpointIndex();
    }
    StakeInfo storage _position = _positions[tokenId];
    return _calculateRewards(_position, checkpointIndex);
```

```
}
```

Recommendation

Remove the redundant checkpoint validation logic from the **`claimableRewards`** function and utilize the **`isCheckpoint`** modifier to ensure consistent and centralized validation.

3S-Deep3 Labs-N03

Inconsistency between documentation and implementation for **`UD3LToken`** minting permissions

Id	3S-Deep3 Labs-N03
Classification	None
Category	Suggestion
Status	Addressed in #8e18c63.

Description

There is a discrepancy between the project documentation and the actual implementation of the **`UD3LToken`** contract regarding token minting permissions. According to the documentation, "the only mechanism by which new utility tokens are minted is through our innovative staking program".

In contrast, the implementation assigns the **`MINTER_ROLE`** to the **`initialOwner`** during deployment—granting minting privileges not exclusively to the staking contract.

The **`mint`** function in **`UD3LToken`** is restricted by the **`onlyRole(MINTER_ROLE)`** modifier, allowing any address with this role to mint tokens. Since the **`initialOwner`** holds this role by default, they are capable of minting tokens independently of the staking mechanism outlined in the documentation.

Recommendation

To align the implementation with the documentation, consider one of these approaches:

1. Update the documentation to accurately reflect that the contract owner also has minting privileges, clarifying the governance model around this capability.

2. Modify the deployment process to immediately revoke the **`MINTER_ROLE`** from the **`initialOwner`** after granting it to the staking contract, ensuring tokens can only be minted through the staking program.

3S-Deep3 Labs-N04

Redundant function calls in inheritance chain leads to unnecessary gas consumption

Id	3S-Deep3 Labs-N04
Classification	None
Category	Suggestion
Status	Addressed in #8d2a83f.

Description

In the **`AccessControlD3L`** contract, there is an issue with the implementation of the **`_grantRole`** and **`_revokeRole`** functions. The contract inherits from both **`AccessControlEnumerable`** and **`AccessControlDefaultAdminRules`**, and attempts to maintain the functionality of both parent contracts by explicitly calling their implementations.

However, the implementation doesn't account for the inheritance chain of these parent contracts. Specifically:

1. When **`AccessControlDefaultAdminRules::_grantRole`** is called, it already calls **`super._grantRole()`** which resolves to **`AccessControlEnumerable::_grantRole`**

2. Similarly, when **`AccessControlDefaultAdminRules::_revokeRole`** is called, it already calls **`super._revokeRole()`** which resolves to **`AccessControlEnumerable::_revokeRole`**

This means that in both the **`_grantRole`** and **`_revokeRole`** functions in **`AccessControlD3L`**, the explicit calls to **`AccessControlEnumerable::_grantRole`** and **`AccessControlEnumerable::_revokeRole`** are redundant and result in duplicate function calls.

For example, in the current implementation of **`_grantRole`**:

```
function _grantRole(bytes32 role, address account)
    internal
    virtual
    override(AccessControlEnumerable, AccessControlDefaultAdminRules)
    returns (bool)
```

```
{
    bool granted;
    granted = AccessControlDefaultAdminRules._grantRole(role, account);
    if (granted) {
        AccessControlEnumerable._grantRole(role, account); // Redundant call
    }
    return granted;
}
```

When **AccessControlDefaultAdminRules::_grantRole(role, account)** is called, it already executes **super._grantRole(role, account)** which calls **AccessControlEnumerable::_grantRole(role, account)**. Therefore, the explicit call to **AccessControlEnumerable::_grantRole(role, account)** is redundant and results in the same function being called twice.

The same issue exists in the **_revokeRole** function.

Recommendation

Remove the redundant calls to **AccessControlEnumerable::_grantRole** and **AccessControlEnumerable::_revokeRole** in the **AccessControlD3L** contract:

```
/// @inheritdoc AccessControlEnumerable
function _grantRole(bytes32 role, address account)
    internal
    virtual
    override(AccessControlEnumerable, AccessControlDefaultAdminRules)
    returns (bool)
{
    // Just call AccessControlDefaultAdminRules._grantRole which will
    // already call AccessControlEnumerable._grantRole through super
    return AccessControlDefaultAdminRules._grantRole(role, account);
}
/// @inheritdoc AccessControlEnumerable
function _revokeRole(bytes32 role, address account)
    internal
    virtual
    override(AccessControlEnumerable, AccessControlDefaultAdminRules)
    returns (bool)
{
    // Just call AccessControlDefaultAdminRules._revokeRole which will
```

```
    // already call AccessControlEnumerable._revokeRole through super
    return AccessControlDefaultAdminRules._revokeRole(role, account);
}
```

This change will maintain the same functionality while eliminating the redundant function calls, resulting in more efficient gas usage.

3S-Deep3 Labs-N05

Rewards and staked tokens sent to caller instead of position owner

Id	3S-Deep3 Labs-N05
Classification	None
Category	Suggestion
Status	Addressed in #19aad6e.

Description

In the **`MaturityAdjustedStaking`** contract, there are two instances where tokens are sent to the caller of a function (**`msg.sender`**) instead of the actual owner of the staking position:

1. In the **`_claim`** internal function, rewards are minted to the caller instead of the position owner:

```
function _claim(uint256 tokenId, uint256 toCheckpoint)
    internal
    returns (uint256 rewards, uint256 lastProcessedCheckpoint, uint256
lastProcessedTime)
{
    (rewards, lastProcessedCheckpoint, lastProcessedTime) =
_processRewards(tokenId, toCheckpoint);
    if (rewards > 0) {
        rewardToken.mint(_msgSender(), rewards);
        emit RewardsClaimed(_msgSender(), ownerOf(tokenId), tokenId,
toCheckpoint, lastProcessedTime, rewards);
    }
}
```

2. In the **`unstake`** function, the staked tokens are returned to the caller instead of the position owner:

```
function unstake(uint256 tokenId)
    external
    override
    isAuthorized(tokenId)
```

```
    nonReentrant
    returns (uint256 rewards, uint256 lastProcessedCheckpoint, uint256
lastProcessedTime)
{
    StakeInfo memory _position = _positions[tokenId];
    (rewards, lastProcessedCheckpoint, lastProcessedTime) = _claim(tokenId);
    delete _positions[tokenId];
    _burn(tokenId);
    stakingToken.safeTransfer(_msgSender(), _position.amount);
    emit Unstaked(_msgSender(), _msgSender(), tokenId, _position.amount);
}
```

The **isAuthorized** modifier ensures that the caller is either the owner of the position or an approved operator, but it doesn't guarantee that the caller is the actual owner. This means that approved operators (who may be contracts or other users) will receive the rewards and staked tokens instead of the actual position owner.

While this doesn't represent a critical security vulnerability since the caller must be authorized by the owner, it does create a UX issue where users might not receive their tokens directly if they use approved operators to manage their positions.

Recommendation

Modify the **_claim** and **unstake** functions to send tokens to the position owner rather than the caller:

```
function _claim(uint256 tokenId, uint256 toCheckpoint)
    internal
    returns (uint256 rewards, uint256 lastProcessedCheckpoint, uint256
lastProcessedTime)
{
    (rewards, lastProcessedCheckpoint, lastProcessedTime) =
_processRewards(tokenId, toCheckpoint);
    if (rewards > 0) {
        // Mint to the owner of the position instead of the caller
        address owner = ownerOf(tokenId);
        rewardToken.mint(owner, rewards);
        emit RewardsClaimed(_msgSender(), owner, tokenId, toCheckpoint,
lastProcessedTime, rewards);
    }
```

```
}


function unstake(uint256 tokenId)
    external
    override
    isAuthorized(tokenId)
    nonReentrant
    returns (uint256 rewards, uint256 lastProcessedCheckpoint, uint256
lastProcessedTime)
{
    StakeInfo memory _position = _positions[tokenId];
    address owner = ownerOf(tokenId);
    (rewards, lastProcessedCheckpoint, lastProcessedTime) = _claim(tokenId);
    delete _positions[tokenId];
    _burn(tokenId);
    // Transfer to the owner of the position instead of the caller
    stakingToken.safeTransfer(owner, _position.amount);
    emit Unstaked(_msgSender(), owner, tokenId, _position.amount);
}
```

This ensures that tokens are always sent to the rightful owner, regardless of who initiates the transaction.