33AUDITS & CO.

Dein Audit Report

Introduction

A time-boxed security review of the protocol was done by 33Audit & Company, focusing on the security aspects of the smart contracts. This audit was performed by 33Audits, Nirlin, Mis4nthr0pic, Zuhaib, Sm4rty.

Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where I try to find as many vulnerabilities as possible. I can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

About 33 Audits & Company

33Audits LLC is an independent smart contract security researcher company and development group. We conduct audits a as a group of independent auditors with various experience and backgrounds. We have conducted over 15 audits with dozens of vulnerabilities found and are experienced in building and auditing smart contracts. We have over 4 years of Smart Contract development with a focus in Solidity, Rust and Move. Check our previous work here or reach out on X @solidityauditor.

About **Dein-fi**

Decentralized Insurance Network Insurance infrastructure on all chains.

Severity classification

Severity	**Impact: High**	**Impact: Medium**	**Impact: Low**
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

Scope

File
DEINToken
LiquidityMiningStakingETH
DEINTreasury

File
DEINNFTStaking
PolicyBookFabric
PolicyRegistry
SwapEvent
ReputationSystem
AaveProtocol
RewardPool
DEINStakingView
CompoundProtocol
YearnProtocol
PolicyBookRegistry
AbstractLiquidityMiningStaking
Vesting
LiquidityRegistry
PolicyBookAdmin
CompoundPool
DEINStaking
AbstractStaking
ContractsRegistry
DEINRewardsGenerator
DemandBookLiquidity
PolicyQuote
YieldGenerator
CapitalPool
DEINCoverStaking
ClaimVoting
DemandBook
PolicyBookFacade
PolicyBook
ClaimingRegistry

ID	Title	Severity	Status
[C-01]	Price feed updates malfunction if price drop or increase by 50% and then stabilize there.	Critical	Fixed
[H-01]	Price feed updates malfunction if price drop or increase by 50% and then stabilize there.	High	Fixed
[M-01]	In dein staking `applyVoterPenalty` does not update the locked amount for the vesting users.	Critical	Fixed
[M-02]	DOS of BMI Swapping functionality if the position NFT is transferred/traded	Critical	Fixed
[M-03]	DEIN tokens not swapped in exchange for BMI tokens get stuck in the `SwapEvent.sol` forever	Medium	Fixed
[M-04]	Hardcoded number of `blocks_per_day` can cause issues in case of future upgrades	Medium	Acknowledged
[M-05]	`addLiquidityETH` and `removeLiquidityETH` calls in `AbstractLiquidityMiningStaking.sol` will fail due to hardcoded slippage settings	Medium	Fixed
[M-06]	The use of the `permit()` function is susceptible to frontrunning attacks, resulting in a denial-of-service (DOS) for users	Medium	Fixed
[M-07]	Restake Reward can be used to game the reward system	Medium	Acknowledged
[L-01]	Small users can evade liquidation contribution	Low	Fixed
[L-02]	Issues in accounting on transfer of USDT when fee is enabled	Low	Acknowledged
[L-03]	OwnableUpgradeable: Does not implement 2-Step-Process for transferring ownership	Low	Fixed
[L-04]	`isLocked` returns incorrect output when empty arrays of `tokenId` is passed	Low	Fixed
[L-05]	Outdated versions of OpenZeppelin library	Low	Fixed
[I-01]	Redundant Code in Committing Withdrawn Claims and Ending Active Demands	Informational	Fixed
[I-02]	Enumerable mapping's add/remove function's return values not checked	Informational	Fixed
[I-03]	Remove unused constant defined	Informational	Fixed

ID	Title	Severity	Status
[I-04]	Incorrect Spell checks across the contracts	Informational	Fixed
[I-05]	Reentrancy Guard is not initialized in Cover Staking contract	Informational	Fixed
[I-06]	Contracts are using hardcoded addresses across all the contracts	Informational	Fixed

Issues

[C-01] Price feed updates malfunction if price drop or increase by 50% and than stabilize there.

Description

The provided code snippet details a function `_updateTokenPrice` designed to update the price of a given token in a smart contract. This function performs several key operations:

- It ensures that the price list for a token does not exceed a predefined maximum length. If it does, the oldest price is removed.
- Calculates the price change ratio based on the new token price and its average price, aiming to identify significant deviations.
- Adds the new price to the token's price list only if the change percentage is within a certain threshold (less than 50%).

However, there are critical issues with this approach:

Price Update Deadlock: In scenarios where the token's price drops by 50% or more or increases by 50% or more suddenly, subsequent updates to the price are blocked. This condition creates a deadlock where the price cannot be updated in the smart contract, effectively causing a denial of service (DOS) for price updates. Now the price may end up staying over that 50% threshold for a long time by just stabilizing there.

For instance, if the average price is $10 and a crash happens, reducing the new price to $4, updates will continue to fail until the price recovers above $5, due to the threshold check.

In code we have this function to update the pricing

```
    function _updateTokenPrice(Token _token, uint256 _tokenPrice) internal {

          if (_tokenPriceList[_token].length() == MAX_PRICE_LIST) {

              uint256 _oldPrice = _tokenPriceList[_token].at(0);

              _tokenPriceList[_token].remove(_oldPrice);
          }

          // the average price is the input price initialy
          uint256 _tokenPriceAverage = _getAveragePriceInUSDT(_token);
```

```
        // calculate the price change ratio
        // the token price and the average can be same
        // and also can be hugely different
        // 400e36/3e18
        uint256 priceChangeRatio =
_tokenPrice.mul(PERCENTAGE_100).tryDiv(_tokenPriceAverage);

        // 100 mei se ratio krli jaye subtract away.
        uint256 priceChangePercentage =
PERCENTAGE_100.trySub(priceChangeRatio);
        if (priceChangePercentage == 0)
            priceChangePercentage = priceChangeRatio.sub(PERCENTAGE_100);
        if (priceChangePercentage != 0 && priceChangePercentage <
PRICE_CHANGE_THRESHOLD) {
            _tokenPriceList[_token].add(_tokenPrice);

            if (_token == Token.DEIN) emit
DEINPriceUpdated(block.timestamp, _tokenPrice);
            else emit ETHPriceUpdated(block.timestamp, _tokenPrice);
        }
    }
```

See these lines

```
    if (priceChangePercentage == 0)
            priceChangePercentage = priceChangeRatio.sub(PERCENTAGE_100);
        if (priceChangePercentage != 0 && priceChangePercentage <
PRICE_CHANGE_THRESHOLD) {
            _tokenPriceList[_token].add(_tokenPrice);

            if (_token == Token.DEIN) emit
DEINPriceUpdated(block.timestamp, _tokenPrice);
            else emit ETHPriceUpdated(block.timestamp, _tokenPrice);
        }
```

If the price change is not zero and is greater than 50% new price is not recorded, and it keeps comparing with the old 10$ pricing, and the price update keeps failing and essentially system keeps working at wrong price, and impact of that trickle down across the whole codebase.

Recommendations

To mitigate these issues and enhance the resilience of the price update mechanism against manipulation, the following recommendation is proposed:

- **Implement a Time-Weighted Average Price (TWAP) Oracle:** Utilizing a TWAP oracle for price updates can significantly reduce the impact of price manipulation. TWAP oracles average out the

price over a period, making it more challenging for bad actors to influence the price with a single large transaction.

[M-01] In dein staking `applyVoterPenalty` does not update the locked amount for the vesting users.

Context: There are two kinds of users, normal and vesting users, for vesting users there is locked amount that is set equal to staked amount at the time of staking, but when applying penalty locked amount is not updated and only staked amount is updated. **Description:** when we apply the penalty in the deinstaking.sol with `applyVoterPenalty()` it only updates the the stakers[_tokenId] mapping but what if the penalty is being applied to vesting user, there is no check for that and locked amount is also not updated.

The problem with that is, in claim function;

Look here

```
    function claim(uint256 _tokenId, uint256 _amount)
        public
        override
        updateReward(_tokenId)
        forceUpdateTokensPrice
    {
        //check ownership
        require(ownerOf(_tokenId) == _msgSender(), "DEINStaking: not a
token owner");

        // clain is only if it is vesting
        require(stakers[_tokenId].isVesting, "DEINStaking: no vesting
exist");

        // vesitng allowed in case locking period not ended,otherwise user
can withdraw all
        require(!canWithdraw(_tokenId), "DEINStaking: withdrawal is
ready");

        VestingInfo storage vesting = vestings[_tokenId];
        // this due amount will be greater than the actual value of
        uint256 _dueAmount =
            deinStakingView.getDueVestedAmount(
                stakers[_tokenId].lockingPeriod,
                vesting.locked,
                vesting.claimed
            );

        if (_dueAmount > 0) {

            require(_dueAmount >= _amount, "DEINStaking: invalid amount");

            require(

ICompoundPool(compoundPool).canWithdraw(getAmountbyLiquidation(_amount),
false),
```

```
                "DEINStaking: no liquidity"
            );

            uint256 amountByLiquidiation = _withdraw(_tokenId, _amount);
            uint256 _staked = stakers[_tokenId].staked;
            if (_amount == _staked) {
                //withdraw full position
                delete vestings[_tokenId];
                _removeTokenPosition(_tokenId, amountByLiquidiation);
            } else {
                stakers[_tokenId].staked = _staked.sub(_amount);

                vesting.claimed = vesting.claimed.add(_amount);

                deinToken.transfer(_msgSender(), amountByLiquidiation);

                emit CLAIMED(amountByLiquidiation, _tokenId,
_msgSender());
            }
        }
    }
```

It uses `getDueVestedAmount()` that takes in the locked amount and return the due vested amount, now the problem here is when the penalty is applied locked amount is not updated and now user can claim more than he should have been entitled too.

Recommendations: In `applyVoterPenalty` function add additional check for the vesting user and update the locked amount too.

Resolution:

[M-02] DOS of BMI Swapping functionality if the position NFT is transferred/traded

Context: A BMI and BMIStk holder can swap his/her Bmi tokens for the dein token via the swapEvent.sol contract, but if the NFT is transferred, it completely DOS the user from the subsequent swap.

Description: When a user swaps via the SwapEven.sol, he can either lock for 24 months or 36 months and for the first time a unique position nft id is minted to the user. For a vesting user -> which is the swapping user, the 24 months and 36-months lock id's are recorded in this mapping:

```
  // Alice(swapper) => 24(locking period) => 1(id)
    // Alice(swapper) => 36(locking period) => 2(id)
    mapping(address => mapping(uint256 => uint256)) public override
tokenToVesting;
```

Now if a Alice transfers his NFT with id 1 to BOB, this mapping should be accordingly updated, which in our case is not updated in the _beforeTokenTransfer function.

And this lines get's the id for alice, even though it is held by BOB.

```
        uint256 tokenID = _deinStaking.tokenToVesting(_msgSender(), lock);
```

Now even though this ID is not held by Alice for 24 month locking period, the system will assume it is, and will try to add it to existing staking position:

```
        IStaking.StakingPosition stakingPosition =
            tokenID == 0 ? IStaking.StakingPosition.NEW :
IStaking.StakingPosition.CURRENT;
        uint256 stakingInput = tokenID == 0 ? lock : tokenID;
        _deinStaking.stakeFor(_msgSender(), amountBMI, stakingInput,
stakingPosition, true);
        ```

and when `stakeFor` function is called and the ownership check fails down
the line here in deinStaking.sol:

```solidity
   function _addToStake(
        address _staker,
        uint256 _tokenId,
        uint256 _amountDEIN,
        bool _isVesting
    ) internal updateReward(_tokenId) forceUpdateTokensPrice {
        // checking if the owner of staking index is the the staker passed
in
        require(ownerOf(_tokenId) == _staker, "DEINStaking: not a token
owner");
```

POC

To check this issue, add the following test case in swapEven.t.sol

```
 function testSwapBMIAfterAliceTransferNFTToBOB(uint96 amountBMI) public {
        vm.assume(amountBMI > 0 && amountBMI < 1_000_000);
        bmi.mintArbitrary(USER1, amountBMI * DECIMALS18);
        vm.prank(USER1);
        bmi.approve(address(swapEvent), type(uint256).max);
        uint256 deinUBefore = dein.balanceOf(USER1);
        uint256 deinSBefore = dein.balanceOf(address(deinStaking));
        uint256 deinSEBefore = dein.balanceOf(address(swapEvent));
        uint256 bmiUBefore = bmi.balanceOf(USER1);
        uint256 bmiSEBefore = bmi.balanceOf(address(swapEvent));
        vm.prank(USER1);
        swapEvent.swapTokens(PERCENTAGE_100, 0);
        uint256 deinUAfter = dein.balanceOf(USER1);
        uint256 deinSAfter = dein.balanceOf(address(deinStaking));
```

```
        uint256 deinSEAfter = dein.balanceOf(address(swapEvent));
        uint256 bmiUAfter = bmi.balanceOf(USER1);
        uint256 bmiSEAfter = bmi.balanceOf(address(swapEvent));
        // user1 transfers the tokenid to user 2 that is probably id 1
        assertEq(deinStaking.ownerOf(1), USER1);
        vm.startPrank(USER1);
        // transfer the token 1 to user 2 erc1155
        // alice transfer nft to the bob(user2)
        deinStaking.safeTransferFrom(USER1, USER2, 1, 1, "");
        vm.stopPrank();
        // user 2 should have the token 1
        // assertEq(deinStaking.ownerOf(1), USER2);
        // now call the swap function again
        bmi.mintArbitrary(USER1, amountBMI * DECIMALS18);
        vm.prank(USER1);
          //expect revert with not a token owner error
        // but is enable to do it due to problem with not updating the
mapping and revert down the pipeline in stakefor function
        vm.expectRevert("DEINStaking: not a token owner");
        // now alice tries to swap all of his tokens for the dein tokens
with vesting period of 24 months
        swapEvent.swapTokens(PERCENTAGE_100, 0);
        vm.stopPrank();
        assertEq(bmi.balanceOf(USER1), amountBMI * DECIMALS18);
    }
```

Recommendations: Update the mapping in _beforeTokenTransfer() function.

Resolution:

Resolution:

[M-03] DEIN tokens not swapped in exchange of BMI tokens get stuck in the `SwapEvent.sol` forever

Links: https://github.com/dein-fi/dein-core/blob/audit-code-freeze/contracts/SwapEvent.sol

Description: The `PREDEPOSITED_AMOUNT` in DEIN tokens is transferred to the `SwapEvent.sol` contract at the event's start. Users engage in swapping their existing BMI tokens for DEIN tokens and subsequently stake them directly for periods of 24/36 months.

Consider the scenario where only 80% of BMI holders participate in the token swap, and the remaining 20% fail to do so, resulting in unclaimed DEIN tokens. As a result, since no function is defined for the remaining DEIN tokens to be claimed at the end of the event, these unclaimed tokens become permanently trapped within the contract. There is also no `burn()` function exposed to burn these tokens after the event is over.

Impact: Unclaimed DEIN tokens get stuck in the contract indefinitely.

Recommendations: A good fix could involve implementing a function to transfer these tokens to another address controlled by the owner at the conclusion of the event like DEINTreasury. This would ensure that the unclaimed DEIN tokens are not left stranded within the contract indefinitely. Additionally, it may be beneficial to consider incorporating a `burn()` function for DEIN tokens.

Resolution:

Resolution:

[M-04] Hardcoded number of `blocks_per_day` can cause issues in case of future upgrades

Context:

https://github.com/dein-fi/dein-core/blob/bb1700b6ff24bf4f1770e49a5a44277383d628a9/contracts/Globals.sol#L22

Description:

Given that the `blocks_per_day` is hardcoded as a constant, any deviation from the expected number of blocks (even as small as a 10% difference) can lead to significant discrepancies in the calculated number of blocks per month and year.

This is especially concerning considering hardforks or upgrades to the network. Also if the contracts get deployed to an L2 this will cause inaccuracies as L2s have different block times.

Recommendations:

Implement a setter function that is only callable by the owner that will allow the admin to update the value of `blocks_per_day` in case of any network changes or discrepancies in the future.

Resolution:

[M-05] `addLiquidityETH` and `removeLiquidityETH` calls in `AbstractLiquidityMiningStaking.sol` with fail due to hardcoded slippage settings

Links: https://github.com/dein-fi/dein-core/blob/bb1700b6ff24bf4f1770e49a5a44277383d628a9/contracts/abstract/AbstractLiquidityMiningStaking.sol#L164-L171 https://github.com/dein-fi/dein-core/blob/bb1700b6ff24bf4f1770e49a5a44277383d628a9/contracts/abstract/AbstractLiquidityMiningStaking.sol#L290-L297

Description: There are two issues with the `addLiquidityETH` and `removeLiquidityETH` functions. The first one is that the `deadline` parameter is set to `type(uint256).max`, creating a problem as this transaction can be mined whenever the miner chooses. In essence, the transaction remains valid indefinitely, which is not a recommended practice. Additionally, the slippage parameter is hardcoded to 1% based on comments and discussions with the development team. This approach raises concerns as slippage should be adjustable to reflect current market conditions and should never be hardcoded. This lack of flexibility may lead to complications and is not conducive to adapting to varying liquidity scenarios.

Recommendations: Consider allowing users to pass the `slippage` and `deadline` parameters themselves.

Resolution:

[M-06] The use of the `permit()` function is susceptible to frontrunning attacks, resulting in a denial-of-service (DOS) for users

Links: https://github.com/dein-fi/dein-core/blob/bb1700b6ff24bf4f1770e49a5a44277383d628a9/contracts/LeveragePool.sol#L618 https://github.com/dein-fi/dein-core/blob/bb1700b6ff24bf4f1770e49a5a44277383d628a9/contracts/abstract/AbstractLiquidityMiningStaking.sol#L49 https://github.com/dein-fi/dein-core/blob/bb1700b6ff24bf4f1770e49a5a44277383d628a9/contracts/abstract/AbstractLiquidityMiningStaking.sol#L86 https://github.com/dein-fi/dein-core/blob/bb1700b6ff24bf4f1770e49a5a44277383d628a9/contracts/DEINCoverStaking.sol#L244

Description: When `permit()` is executed, it primarily checks the validity of the signature and ensures that the deadline has not passed. It's important to note that, by design, the token disregards the `msg.sender` of the `permit()` call. This, coupled with the visibility of transactions in the mempool, makes it susceptible to frontrunning by any user or bot.

A malicious actor can monitor the mempool, waiting for a signature to be sent alongside specific function calls. Upon identifying such a signature, the attacker can frontrun it by directly calling `permit()`. This poses a DoS attack to users, as it disrupts the expected functionality following the `permit()`.

The affected functions in the code include `stakeDEINXWithPermit()`, `provideLiquidityWithPermit()`, `stakeWithPermit()`, and `requestWithdrawalWithPermit()`.

Impact: The functionality that follows the `permit()` function is never executed; essentially, the function call reverts.

POC: Read the blog on trust-security.xyz that explains the issue in depth: https://www.trust-security.xyz/post/permission-denied

Recommendations: Wrap any calls that use `permit()` in a try/catch block. This way, even if an attacker front-runs the transaction, the function will continue to execute following the call to `permit()`.

Here's an example with the `stakeDEINXWithPermit` function:

```
function stakeDEINXWithPermit(
    uint256 deinXAmount,
    address policyBookAddress,
    uint256 lock,
    uint256 _deadline,
    uint8 v,
    bytes32 r,
    bytes32 s
) external override nonReentrant {
    // Try permit() before staking to use the nonce if possible
    try IERC20PermitUpgradeable(policyBookAddress).permit(
        _msgSender(),
        address(this),
        deinXAmount,
        _deadline,
```

```
            v,
            r,
            s
        ) {
            // Permit was successful
        } catch {
            // Permit failed, continue with the staking if possible
        }
        _stakeDEINX(_msgSender(), deinXAmount, policyBookAddress, lock, true);
    }
```

Resolution:

[M-07]- Restake Reward can be used to game the reward system.

Context: Restaking the reward allows users to earn more rewards than less amount of time. **Description:** `restakeReward()` function allows user to restake their reward in two ways, either in an existing position or a new position, now the existing position can have any locking period [1,6,12,24,36] months,

Consider the following scenario:

1. User 1 stakes into a locking position with 12 months, and get the staked multiplier of 3.
2. Now user accumulates the reward and calls the restake reward function.
3. User 1 can restake this into new position with new locking period of existing position
4. Let's say the 11 months have passed since the original position was created, now user wants to create the position for 1 month.
5. But user can game in a way, he can restake in existing position for only last one month and get the staked multiplier of 3, which if he created the new position he could only get the staked multiplier of 1.

Recommendations: Assess the restaking mechanism, one way to go is that reward can be restaked into new position only not existing. **Resolution:**

[L-01] Protocol can frontrun the call to `ClaimingRegistry::withdrawClaim()` to force user to pay high commissions

Links: https://github.com/dein-fi/dein-core/blob/bb1700b6ff24bf4f1770e49a5a44277383d628a9/contracts/ClaimingRegistry.sol#L779-L795 https://github.com/dein-fi/dein-core/blob/bb1700b6ff24bf4f1770e49a5a44277383d628a9/contracts/CapitalPool.sol#L538-L543 https://github.com/dein-fi/dein-core/blob/bb1700b6ff24bf4f1770e49a5a44277383d628a9/contracts/CapitalPool.sol#L223-L228

Description: The `ClaimingRegistry::withdrawClaim()` function is called when a user seeks to claim a refund for a specific `claimId`. This function is permissionless, allowing anyone to execute it. However, if the caller is not the rightful claimant, a commission fee is imposed on the user.

It is essential to highlight that these commission rates lack upper limits. Consequently, a malicious protocol owner could manipulate the system by setting exorbitant commission rates. This can lead to front-running

user transactions, forcing them into paying commissions and essentially depriving them of their rewards.

Impact: The protocol can front-run user transactions, compelling them to pay high commission fee.

Recommendations: A quick fix would involve adding an upper limit for the commission fee and introducing a delay in the `ClaimingRegistry::withdrawClaim()` function. If users do not invoke this function within the specified delay period, it can be called by other users once the period expires. å

[L-02] Small users can evade liquidation contribution

Context: The liquidation amount is socialised among all the users when the withdraw happen, so a small user can evade this by creating a small position.

Description: In the withdraw function and claim function, internal function _withdraw is called, which also socialise the liquidation amount among the users if there is any.

The amount to be deducted from the user is calculated in the `getAmountbyLiquidation()` function which is as following and have added the comment on how this can go wrong.

```
    function getAmountbyLiquidation(uint256 _amount) public view override
returns (uint256) {
and than use that percentage to calculate the amount he should contribute.
        uint256 userLiquidationAmount =
liquidationAmount.mul(_amount).tryDiv(totalPool);

        return _amount.sub(userLiquidationAmount);
    }
```

This will need a lot of dein to be staked and relatively very small liquidation so is a low.

Recommendations: One approach to solve this problem is to change the formula such that first we calculate the percentage a user ows and than deduct that percentage from his balance.

Resolution:

[L-03] Issues in accounting on transfer of USDT when fee is enabled

Links:

https://github.com/dein-fi/dein-core/blob/bb1700b6ff24bf4f1770e49a5a44277383d628a9/contracts/CapitalPool.sol#L180-L181
https://github.com/dein-fi/dein-core/blob/bb1700b6ff24bf4f1770e49a5a44277383d628a9/contracts/CapitalPool.sol#L193-L198

Type of Issue:

Logical

Description:

The stablecoin utilized in the DEIN protocol is USDT, which is inherently a **fee-on-transfer** token. Currently, no fees are enabled for transfers, but the admin can always set a fee in the future. If such an instance occurs, the current accounting of stablecoins within smart contracts will break completely, as it does not accommodate fee-on-transfer tokens.

Impact:

The accounting of token transfers across the DEIN protocol will become inconsistent.

POC

Refer the code at - https://etherscan.io/token/0xdac17f958d2ee523a2206206994597c13d831ec7#code#L127

Recommendations:

Even though this scenario is very unlikely, incorporating robust code to handle such situations is a prudent approach. Essentially, tracking token amounts based on balance changes rather than relying solely on parameters passed to the function is recommended.

Resolution:

[L-04] OwnableUpgradeable: Does not implement 2-Step-Process for transferring ownership

Links: https://github.com/dein-fi/dein-core/tree/bb1700b6ff24bf4f1770e49a5a44277383d628a9/contracts

Description: The contract does not implement a 2-Step-Process for transferring ownership. Ownership of the contract can easily be lost if an incorrect address is passed as an argument to the `transferOwnership` function.

Since the privileged roles have critical function roles assigned to them. Assigning the ownership to the wrong user can be disastrous.

Consider using the Ownable2StepUpgradeable contract from OZ instead.

Two-step ownership transfer uses `transferOwnership` to transfer the ownership and the new owner must call `acceptOwnership` in order for the transfer to be completed. Refer to the above Ownable2StepUpgradeable.sol for more details.

Impact: Loss of ownership can occur in scenarios where the current owner makes a mistake in transferring it.

Recommendations: Implement 2-Step-Process for transferring ownership via Ownable2StepUpgradeable.

Resolution:

[L-05] `isLocked` returns incorrect output when empty arrays of `tokenId` is passed

Links: https://github.com/dein-fi/dein-core/blob/bb1700b6ff24bf4f1770e49a5a44277383d628a9/contracts/DEINNFTStaking.sol#L76

Description: The current implementation of the `isLocked()` function returns whether the specific `tokenId` is locked for a given user. If the array is empty, the expected behavior is that all values should be `false`. However, the current implementation returns `true` in this incorrect case.

```
    function isLocked(uint256[] calldata tokenIDs, address user)
        external
        view
        override
        returns (bool _isLocked)
    {
        _isLocked = true;
        for (uint256 i = 0; i < tokenIDs.length; i = i.uncheckedInc()) {
            if (!isLocked(tokenIDs[i], user)) {
                _isLocked = false;
                break;
            }
        }
    }
```

In isLocked() function, the variable _isLocked is initialized to true. Then, it iterates over each tokenId in the tokenIDs array. If any of the tokens are not locked for the given user, _isLocked is set to false. However, if the tokenIDs array is empty, the loop is never entered, and _isLocked remains true by default.

There is no direct impact as such.

Recommendations: Add a require check to verify that the `tokenIDs` array is not empty.

```
    function isLocked(uint256[] calldata tokenIDs, address user)
        external
        view
        override
        returns (bool _isLocked)
    {
+       require(tokenIDs.length > 0, "TokenIDs array cannot be empty"); // Check if the array is not empty
        _isLocked = true;
        for (uint256 i = 0; i < tokenIDs.length; i = i.uncheckedInc()) {
            if (!isLocked(tokenIDs[i], user)) {
                _isLocked = false;
                break;
            }
        }
    }
```

Resolution:

[L-06] Outdated versions of OpenZeppelin library

Links: https://github.com/dein-fi/dein-core/blob/bb1700b6ff24bf4f1770e49a5a44277383d628a9/package.json#L42-L43 https://github.com/dein-fi/dein-core/blob/bb1700b6ff24bf4f1770e49a5a44277383d628a9/contracts/tokens/DEINToken.sol#L5

Description: The current set of contracts utilizes the following versions, released on November 8, 2022:

```
"@openzeppelin/contracts": "^4.8.0",
"@openzeppelin/contracts-upgradeable": "^4.5.2",
```

However, numerous security issues have been reported and subsequently addressed in these contracts since their release.

Additionally, it should be noted that the contracts currently use `draft-ERC20Permit`, which is not recommended for production use. The suitable version for production is available in the latest stable release of OpenZeppelin libraries.

Recommendations: To fix the issue, use the latest stable version of the OZ contracts. Reference - https://github.com/OpenZeppelin/openzeppelin-contracts/releases

Resolution:

[I-01] Redundant Code in Committing Withdrawn Claims and Ending Active Demands

Context: DemandBook.sol#L808

Description: In DemandBook.sol, Both commitWithdrawnClaim and `endActiveDemand` functions call the internal `_endDemand` function with the same parameters, `demandIndex` and `block.timestamp`. The `_endDemand` function sets the end time of a demand based on certain conditions, effectively ending the demand.

```
    function commitWithdrawnClaim(uint256 demandIndex) external override
onlyClaimingRegistry {
        _endDemand(demandIndex, block.timestamp);
    }

    function endActiveDemand(uint256 demandIndex) external override
onlyClaimingRegistry {
        _endDemand(demandIndex, block.timestamp);
    }
```

`commitWithdrawnClaim` is invoked when a claim is withdrawn, whereas `endActiveDemand` is invoked when ending an active demand. Despite the differing contexts, the primary goal of both functions remains

consistent: to end the active demand. They achieve this by invoking the `_endDemand` function, effectively ending the demand regardless of the scenario.

It does not pose an immediate security threat, it can lead to confusion, maintenance issues, and potential inefficiencies in the codebase.

Recommendations:

Consolidate the functionality of `commitWithdrawnClaim` and `endActiveDemand` functions into a single function.

Resolution:

[I-02] Enumerable mapping's add/remove function's return values not checked

Description: Enumerable mapping is used very extensively across the codebase, but the return value for the add and remove is not checked, basically they return false if the action fails.

There are certain functions where they are removing the items from set but not checking if it exists in the set. For example, If in the `blacklistDistributor` function in PolicyBookAdmin, there isn't any check if the address is already removed from `whitelistedDistributor`. And there is no check for return value of remove.

So, the call will be successful and events will be emitted for the same even if the remove call failed.

```
/// @notice Removes a distributor address from the distributor whitelist
/// @param _distributor distributor address that will be blacklist
function blacklistDistributor(address _distributor) external override
onlyOwner {
    _whitelistedDistributors.remove(_distributor); //@audit add a check
here for if the distributor is already removed or not?
    delete distributorFees[_distributor];

    emit DistributorBlacklisted(_distributor);
}
```

Recommendations:

Check the return value for when adding or removing to the `_whiteListedDistributors` mapping.

Resolution:

[I-03] Remove unused constant defined

Links: https://github.com/dein-fi/dein-core/blob/bb1700b6ff24bf4f1770e49a5a44277383d628a9/contracts/CompoundPool.sol#L33

Description: In the `CompoundPool.sol` contract, there is a constant `USD_DECIMALS` defined on #L33. However, this constant appears to be unused throughout the contract.

```
    uint256 public constant USD_DECIMALS = 10**8;
```

Recommendations: Consider removing the declaration of the `USD_DECIMALS` constant

Resolution:

[I-04] Incorrect Spell checks across the contracts

Context: https://github.com/dein-fi/dein-core/blob/audit-code-freeze/contracts/tokens/DEINToken.sol

Description: Decentralized is mispelled in the token contract which will cause a redeployment of the token. Deposit is mispeleed in the YieldGenerator contract. Demand is misspelled in createDemand function in DemandBook contract.

```
contract DEINToken is ERC20Permit {
    uint256 public constant TOTAL_SUPPLY = 1 * (10**9) * (10**18);

    /// @dev Initializes the DEIN token contract.
    /// @param tokenReceiver The initial recipient of all the DEIN tokens.
    constructor(address tokenReceiver)
        ERC20Permit("DECENRALIZED INSURANCE")
        ERC20("DECENRALIZED INSURANCE", "DEIN")
    {
        _mint(tokenReceiver, TOTAL_SUPPLY);
    }
}
```

Recommendations: Fix these typos to avoid redeployment or avoid confusion when calling these functions from the front end.

Resolution:

[I-05] Reentrancy Guard is not initialized in Cover Staking contract

Context: DEINCoverStaking.sol#L10 AbstractStaking.sol#L6

Description: The AbstractStaking and DEINCoverStaking inherit ReentrancyGuardUpgradeable from Openzeppelin but it is not initialized in the `__Staking_init()` and `__DEINCoverStaking_init()` functions.

```
import "@openzeppelin/contracts-
upgradeable/security/ReentrancyGuardUpgradeable.sol";

    function __Staking_init() internal onlyInitializing {
        __Ownable_init();

        lockingPeriods = [1, 6, 12, 24, 36];
```

```
        stakingIndex = 1;
        _setStakedMultiplier();
    }
```

The same thing happening in DeinStaking.sol, it inherits from AbstractStaking.sol which inherits ReentrancyGuardUpgradeable, but it is never initialised there either

Recommendations: Call the __ReentrancyGuard_init() in the init functions of respective contracts.

Resolution:

[I-06] Contracts is using hardcoded address across all the contracts

Links: https://github.com/dein-fi/dein-core/blob/bb1700b6ff24bf4f1770e49a5a44277383d628a9/contracts/LiquidityRegistry.sol#L408 https://github.com/dein-fi/dein-core/blob/bb1700b6ff24bf4f1770e49a5a44277383d628a9/contracts/PolicyBookAdmin.sol#L125 https://github.com/dein-fi/dein-core/blob/bb1700b6ff24bf4f1770e49a5a44277383d628a9/contracts/PolicyBookAdmin.sol#L238 https://github.com/dein-fi/dein-core/blob/bb1700b6ff24bf4f1770e49a5a44277383d628a9/contracts/PolicyBookRegistry.sol#L125-L128

Description: As a best practice, hardcoding addresses directly within a function is generally discouraged in software development. It is advisable to use constants to store such values, promoting a more modular and easily adaptable codebase. This approach enhances code readability, promotes easier updates, and facilitates the management of various parameters, ensuring a more robust and maintainable software solution.

Recommendations: Define the value as a constant to enhance readability and understanding.

Resolution: