Scanned Code Report

AUDITAGENT

Code Info

Organization
Gearbox-protocol

Scan ID
2

Repository
core-v3

Branch
next

Commit Hash
a94cb842d221594fff4fac92253d316c24bcad7e

Contracts in scope

contracts/core/AliasedLossPolicyV3.sol
contracts/core/BotListV3.sol
contracts/core/DefaultAccountFactoryV3.sol
contracts/core/GearStakingV3.sol
contracts/core/PriceOracleV3.sol
contracts/credit/CreditAccountV3.sol
contracts/credit/CreditConfiguratorV3.sol
contracts/credit/CreditFacadeV3.sol
contracts/credit/CreditManagerV3.sol
contracts/credit/CreditManagerV3_USDT.sol
contracts/libraries/BalancesLogic.sol
contracts/libraries/BitMask.sol
contracts/libraries/CollateralLogic.sol
contracts/libraries/CreditAccountHelper.sol
contracts/libraries/Constants.sol
contracts/libraries/CreditLogic.sol
contracts/libraries/MarketHelper.sol
contracts/libraries/OptionalCall.sol
contracts/libraries/QuotasLogic.sol
contracts/libraries/USDTFees.sol
contracts/pool/GaugeV3.sol
contracts/pool/LinearInterestRateModelV3.sol
contracts/pool/PoolQuotaKeeperV3.sol
contracts/pool/PoolV3.sol
contracts/pool/PoolV3_USDT.sol
contracts/pool/TumblerV3.sol
contracts/traits/ACLTrait.sol
contracts/traits/ContractsRegisterTrait.sol
contracts/traits/PriceFeedValidationTrait.sol
contracts/traits/ReentrancyGuardTrait.sol
contracts/traits/SanityCheckTrait.sol
contracts/traits/USDT_Transfer.sol

Code Statistics

Findings	Contracts Scanned	Lines of Code
26	32	7929

Code Summary

Gearbox Protocol V3

Gearbox Protocol is a generalized leverage protocol that allows users to take leveraged positions in DeFi. The protocol consists of several interconnected components that work together to provide a secure and efficient lending and borrowing system.

Core Components

- **Pools**: Liquidity providers deposit assets into pools to earn interest. Pools implement the ERC-4626 standard and support EIP-2612 permits.
- **Credit Managers**: Manage borrowing from pools and enforce collateral requirements.
- **Credit Accounts**: Individual leveraged positions owned by users, managed through Credit Managers.
- **Credit Facades**: User-facing contracts that provide a simplified interface for interacting with Credit Managers.
- **Adapters**: Allow Credit Accounts to interact with external DeFi protocols.
- **Price Oracle**: Provides token price data for collateral valuation.
- **Quota System**: Limits exposure to risky assets through quotas that users must "purchase".
- **Interest Rate Models**: Determine borrowing rates based on pool utilization.
- **GEAR Staking**: Governance mechanism allowing GEAR token holders to vote on protocol parameters.

Key Features

- **Leveraged Positions**: Users can open credit accounts with leverage to amplify their exposure to various DeFi strategies.
- **Multicall Operations**: Users can batch multiple operations in a single transaction, including adding/withdrawing collateral, changing debt size, and interacting with external protocols.
- **Collateral Management**: The protocol enforces collateral requirements and liquidates undercollateralized positions.
- **Quota System**: Limits exposure to risky assets through quotas that incur interest over time or one-time fees.
- **Bot Automation**: Users can authorize bots to manage their credit accounts within specified permissions.
- **Liquidations**: Undercollateralized or expired positions can be liquidated to protect lenders.

Protocol Flow

1. Liquidity providers deposit assets into pools and receive LP tokens.
2. Borrowers open credit accounts through Credit Facades, providing initial collateral.
3. Credit Managers enforce collateral requirements and manage interactions with external protocols.
4. Users can add/withdraw collateral, increase/decrease debt, and interact with external protocols through multicalls.
5. If a position becomes undercollateralized, it can be liquidated to repay the debt.

Risk Management

- **Collateral Checks**: Ensure positions remain sufficiently collateralized.
- **Liquidation Mechanism**: Allows liquidators to repay debt and seize collateral from undercollateralized positions.
- **Quota System**: Limits exposure to risky assets.
- **Price Oracle**: Provides reliable price data for collateral valuation.
- **Interest Rate Models**: Adjust borrowing rates based on utilization to incentivize optimal capital efficiency.

Governance

- **GEAR Staking**: GEAR token holders can stake their tokens to participate in governance.
- **Gauge System**: Allows GEAR holders to vote on quota rates for different assets.
- **Configurators**: Special contracts that can modify parameters of Credit Managers and Pools.

Main Entry Points and Actors

Actors:

- **Liquidity Providers**: Deposit assets into pools to earn interest.
- **Borrowers**: Open credit accounts and take leveraged positions.
- **Liquidators**: Liquidate undercollateralized positions.
- **Bots**: Automated systems that can manage credit accounts with owner permission.
- **Configurators**: Governance participants who can modify protocol parameters.
- **GEAR Stakers**: Governance participants who vote on protocol parameters.

Entry Points:

- **PoolV3**:
- `deposit(uint256 assets, address receiver)`: Deposits assets into the pool.

- `withdraw(uint256 assets, address receiver, address owner)`: Withdraws assets from the pool.
- `lendCreditAccount(uint256 borrowedAmount, address creditAccount)`: Lends funds to a credit account.
- `repayCreditAccount(uint256 repaidAmount, uint256 profit, uint256 loss)`: Updates pool state to indicate debt repayment.
- **CreditFacadeV3**:
- `openCreditAccount(address onBehalfOf, MultiCall[] calls, uint256 referralCode)`: Opens a new credit account.
- `closeCreditAccount(address creditAccount, MultiCall[] calls)`: Closes a credit account.
- `multicall(address creditAccount, MultiCall[] calls)`: Executes a batch of calls on a credit account.
- `botMulticall(address creditAccount, MultiCall[] calls)`: Allows bots to manage credit accounts.
- `liquidateCreditAccount(address creditAccount, address to, MultiCall[] calls, bytes memory lossPolicyData)`: Liquidates an unhealthy credit account.
- `partiallyLiquidateCreditAccount(address creditAccount, address token, uint256 repaidAmount, uint256` Partially liquidates a credit account.
- **GearStakingV3**:
- `deposit(uint96 amount, MultiVote[] votes)`: Stakes GEAR tokens.
- `withdraw(uint96 amount, address to, MultiVote[] votes)`: Unstakes GEAR tokens.
- `multivote(MultiVote[] votes)`: Performs a sequence of votes.
- `migrate(uint96 amount, MultiVote[] votesBefore, MultiVote[] votesAfter)`: Migrates staked GEAR to a successor contract.
- **BotListV3**:
- `setBotPermissions(address bot, address creditAccount, uint192 permissions)`: Sets bot permissions for a credit account.
- `eraseAllBotPermissions(address creditAccount)`: Removes all bot permissions for a credit account.

Complex multicall logic in CreditFacadeV3 may allow collateral check bypass

High Risk

The _multicall function in CreditFacadeV3 implements a complex batching of operations that execute both internal calls (to functions of the credit facade itself) and external adapter calls. It first stores expected balances of certain tokens using the BalancesLogic.storeBalances function and, after executing all calls, compares the current balances to these expected values using BalancesLogic.compareBalances. In particular, the code segment:

```
if (expectedBalances.length != 0) {
    address failedToken = BalancesLogic.compareBalances(creditAccount,
expectedBalances, Comparison.GREATER_OR_EQUAL);
    if (failedToken != address(0)) revert
BalanceLessThanExpectedException(failedToken);
}
```

ensures that token balances do not drop below expected values during multicall execution. However, an attacker who controls a registered adapter (or bot with delegated permissions) could design a sequence of calls that manipulate token balances in an unexpected way, for example by temporarily transferring collateral in and out or by returning deceptive values from external calls. Such manipulation might allow the attacker to bypass the collateral check, thereby exposing the protocol to the risk of undercollateralized credit accounts and potential fund misappropriation.

Lack of validation for `token` parameter in `_tryWithdrawPhantomToken` function

High Risk

In the `_tryWithdrawPhantomToken` function of the CreditFacadeV3 contract, there is a critical vulnerability where the `token` parameter is not properly validated before making an external call. The function attempts to check if the token is a phantom token by making a static call to the token address with the `getPhantomTokenInfo` selector.

```
(bool success, bytes memory returnData) =
OptionalCall.staticCallOptionalSafe({
    target: token,
    data: abi.encodeWithSelector(IPhantomToken.getPhantomTokenInfo.selector),
    gasAllowance: 30_000
});
if (!success) return (token, amount, flags);

(address target, address depositedToken) = abi.decode(returnData, (address,
address));

// ensure that `token` is recognized by the credit manager
_getTokenMaskOrRevert(token);
```

However, the function only validates that the token is recognized by the credit manager *after* making the external call and decoding the return data. This means that an attacker could pass a malicious contract address as the `token` parameter, which could return carefully crafted data that would be decoded as valid phantom token information.

The vulnerability is particularly severe because the function later makes another external call to the `target` address obtained from the first call:

```
flags = _externalCall({
    creditAccount: creditAccount,
    target: target,
    adapter: ICreditManagerV3(creditManager).contractToAdapter(target),
    callData: abi.encodeCall(IPhantomTokenWithdrawer.withdrawPhantomToken,
(token, amount)),
    flags: flags
});
```

This allows an attacker to potentially execute arbitrary code through a carefully crafted malicious contract. The impact is high as it could lead to theft of funds from credit accounts or

other severe security breaches. The likelihood is also high since the function can be called through the `withdrawCollateral` multicall operation, which is accessible to credit account owners.

Mitigation Analysis: This is a valid high severity issue. The lack of validation before making external calls could lead to arbitrary code execution.

3 of 26 Findings

CreditFacadeV3.sol

Incorrect validation in `_calcPartialLiquidationPayments` function

High Risk

In the `partiallyLiquidateCreditAccount` function of the CreditFacadeV3 contract, there is a critical vulnerability in the validation of tokens during partial liquidation. The function calls `_calcPartialLiquidationPayments` to calculate the amount of collateral to seize, but it doesn't properly validate that the token being seized is not the underlying token until after the phantom token withdrawal logic is executed.

```
(token, seizedAmount, flags) = _tryWithdrawPhantomToken(creditAccount, token,
seizedAmount, 0); // U:[FA-16A]
if (token == underlying) revert UnderlyingIsNotLiquidatableException(); // U:
[FA-16,16A]
```

This creates a vulnerability where if the token is a phantom token and its `depositedToken` is the underlying token, the check will only happen after the phantom token has already been withdrawn, potentially allowing the liquidation of the underlying token which should be forbidden.

The issue is compounded by the fact that the `_tryWithdrawPhantomToken` function can change both the token and the seized amount, but the validation happens after these changes. This means that even if the original token passed to the function is not the underlying, the function might end up trying to seize the underlying token after the phantom token withdrawal.

This vulnerability could allow attackers to partially liquidate credit accounts in ways that should be forbidden, potentially leading to loss of funds for borrowers or unexpected behavior in the protocol. The impact is high as it could lead to unfair liquidations, and the likelihood is medium as it requires specific conditions with phantom tokens.

Mitigation Analysis: This is a valid high severity issue. The incorrect validation order could allow liquidation of the underlying token which should be forbidden.

Direct storage manipulation via inline assembly in CreditManagerV3's account opening/closing

Medium Risk

In the openCreditAccount and closeCreditAccount functions of CreditManagerV3, the contract uses inline assembly to directly write to the CreditAccountInfo struct storage. For example, in openCreditAccount the borrower field is set as follows:

```
assembly {
    let slot := add(newCreditAccountInfo.slot, 4)
    sstore(slot, value)
}
```

This low-level manipulation bypasses the standard Solidity type safety and layout guarantees. If the underlying storage layout changes (for example, due to an upgrade or a bug in the initialization logic), this assembly code may overwrite unintended storage slots, possibly leaving a credit account in an inconsistent state. Such an inconsistency might be exploited to reuse an account with nonzero debt or to bypass proper liquidations, thereby potentially harming the protocol's stability.

DoS risk via debt update frequency check in CreditManagerV3

Medium Risk

In the manageDebt function of CreditManagerV3, there is a check to ensure that a credit account's debt is updated only once per block:

```
if (currentCreditAccountInfo.lastDebtUpdate == block.number) {
    revert DebtUpdatedTwiceInOneBlockException();
}
```

While this is intended as a reentrancy-prevention mechanism, it can be exploited as a denial-of-service vector in scenarios where an attacker is able to trigger a debt update on a target credit account, setting the lastDebtUpdate to the current block number. Any subsequent legitimate operation within the same block that would update the debt (for example, a necessary adjustment to maintain collateralization) would then revert with a DebtUpdatedTwiceInOneBlockException. This could effectively freeze operations on the credit account, leading to a denial of service and potentially preventing timely liquidations of undercollateralized accounts.

Price Oracle Fallback Vulnerability

Medium Risk

The PriceOracleV3 contract employs an optional call to invoke the skipPriceCheck() function on a given price feed using a fixed gas allowance. If the target price feed contract returns empty data (for example, via a fallback function) or behaves unexpectedly, the skipCheck flag remains false. This forces the oracle to perform stricter validations, which might result in reverts or the use of stale price data. This vulnerability can lead to mispricing of collateral and erroneous liquidation decisions.

Example snippet:

```
function _validatePriceFeed(address priceFeed, uint32 stalenessPeriod)
internal view returns (bool skipCheck) {
    if (!priceFeed.isContract()) revert
AddressIsNotContractException(priceFeed);

    try IPriceFeed(priceFeed).decimals() returns (uint8 _decimals) {
        if (_decimals != 8) revert IncorrectPriceFeedException();
    } catch {
        revert IncorrectPriceFeedException();
    }

    (bool success, bytes memory returnData) =
OptionalCall.staticCallOptionalSafe({
        target: priceFeed,
        data: abi.encodeWithSelector(IPriceFeed.skipPriceCheck.selector),
        gasAllowance: 10000
    });
    if (success) {
        skipCheck = abi.decode(returnData, (bool));
    }

    try IPriceFeed(priceFeed).latestRoundData() returns (uint80, int256
answer, uint256, uint256 updatedAt, uint80) {
        if (skipCheck) {
            if (stalenessPeriod != 0) revert IncorrectParameterException();
        } else {
            if (stalenessPeriod == 0) revert IncorrectParameterException();
            _checkAnswer(answer, updatedAt, stalenessPeriod);
        }
    } catch {
        revert IncorrectPriceFeedException();
    }
}
```

A malicious or misconfigured price feed could exploit this behavior, undermining the reliability of price data used throughout the protocol.

Mitigation Analysis: Downgraded from High to Medium. While the issue is valid, the impact is limited as it requires a malicious or misconfigured price feed. The fixed gas limit could lead to DoS but not direct fund loss.

Unsafe external calls in `_externalCall` function

Medium Risk

The `_externalCall` function in CreditFacadeV3 makes external calls to adapters without proper validation of the return data, which could lead to unexpected behavior:

```
function _externalCall(address creditAccount, address target, address
adapter, bytes memory callData, uint256 flags)
    internal
    returns (uint256)
{
    if (adapter == address(0) || target == address(0)) revert
TargetContractNotAllowedException(); // U:[FA-38]

    if (flags & EXTERNAL_CONTRACT_WAS_CALLED_FLAG == 0) {
        _setActiveCreditAccount(creditAccount); // U:[FA-38]
        flags |= EXTERNAL_CONTRACT_WAS_CALLED_FLAG; // U:[FA-38]
    }

    bool useSafePrices = abi.decode(adapter.functionCall(callData), (bool));
// U:[FA-38]
    if (useSafePrices) flags |= REVERT_ON_FORBIDDEN_TOKENS_FLAG |
USE_SAFE_PRICES_FLAG; // U:[FA-38,45]

    emit Execute({creditAccount: creditAccount, targetContract: target}); //
U:[FA-38]
    return flags;
}
```

The function makes a call to the adapter contract and directly decodes the return value as a boolean without any checks on the return data length or content. If the adapter contract returns data that cannot be properly decoded as a boolean, this could lead to unexpected reverts or incorrect behavior.

Additionally, the function doesn't handle the case where the adapter might return more data than just a boolean. In such cases, the decoding would still succeed but might ignore important information.

This vulnerability could be exploited if a malicious or incorrectly implemented adapter is added to the system, potentially leading to denial of service or unexpected behavior in the protocol. The impact is medium as it requires a compromised or incorrectly implemented adapter, and

the likelihood is medium as adapter contracts are typically carefully reviewed before being added to the system.

Mitigation Analysis: This is a valid medium severity issue. Unsafe external calls without proper validation could lead to unexpected behavior.

Potential reentrancy in `_tryWithdrawPhantomToken` function

Medium Risk

The `_tryWithdrawPhantomToken` function in CreditFacadeV3 contains a potential reentrancy vulnerability:

```
function _tryWithdrawPhantomToken(address creditAccount, address token,
uint256 amount, uint256 flags)
    internal
    returns (address, uint256, uint256)
{
    // ... [static call to check if token is a phantom token] ...

    // ensure that `token` is recognized by the credit manager
    _getTokenMaskOrRevert(token); // U:[FA-36A]

    uint256 balanceBefore =
IERC20(depositedToken).safeBalanceOf(creditAccount);
    flags = _externalCall({
        creditAccount: creditAccount,
        target: target,
        adapter: ICreditManagerV3(creditManager).contractToAdapter(target),
        callData:
abi.encodeCall(IPhantomTokenWithdrawer.withdrawPhantomToken, (token,
amount)),
        flags: flags
    }); // U:[FA-36A]

    emit WithdrawPhantomToken(creditAccount, token, amount); // U:[FA-36A]
    return (depositedToken,
IERC20(depositedToken).safeBalanceOf(creditAccount) - balanceBefore, flags);
}
```

The function makes an external call to withdraw a phantom token and then calculates the amount of the deposited token that was received by subtracting the balance before the call from the balance after the call. However, if the external contract is malicious or compromised, it could potentially reenter the CreditFacadeV3 contract during the call.

While the function itself is protected by the `nonReentrant` modifier in the calling functions, the reentrancy could still cause issues if the malicious contract manipulates the balance of the deposited token between the external call and the balance check. This could lead to incorrect calculations of the seized amount.

The vulnerability is mitigated by the fact that adapters and target contracts are typically carefully reviewed before being added to the system, but it still represents a potential risk. The impact is medium as it could lead to incorrect calculations, and the likelihood is low as it requires a compromised adapter or target contract.

Mitigation Analysis: This is a valid medium severity issue. The potential reentrancy vulnerability could lead to incorrect calculations.

Incorrect handling of quota interest in `manageDebt` function

Medium Risk

In the `manageDebt` function of the CreditManagerV3 contract, there is a vulnerability in the handling of quota interest when decreasing debt to zero:

```
if (amount == maxRepayment) {
    newDebt = 0;
    newCumulativeIndex = collateralDebtData.cumulativeIndexNow;
    profit = collateralDebtData.accruedFees;
    newCumulativeQuotaInterest = 0;
    currentCreditAccountInfo.quotaFees = 0;
} else {
    // ... [calculation for partial repayment] ...
}

if (collateralDebtData.quotedTokens.length != 0) {
    // zero-debt is a special state that disables collateral checks so having
quotas on
    // the account should be forbidden as they entail debt in a form of quota
interest
    if (newDebt == 0) revert DebtToZeroWithActiveQuotasException(); // U:[CM-
11A]

    // quota interest is accrued in credit manager regardless of whether
anything has been repaid,
    // so they are also accrued in the quota keeper to keep the contracts in
sync

IPoolQuotaKeeperV3(collateralDebtData._poolQuotaKeeper).accrueQuotaInterest({
        creditAccount: creditAccount,
        tokens: collateralDebtData.quotedTokens
    }); // U:[CM-11A]
}
```

The issue is that when a user attempts to fully repay their debt (`amount == maxRepayment`), the function sets `newDebt = 0` and `newCumulativeQuotaInterest = 0`, but then immediately checks if there are any quoted tokens. If there are, it reverts with `DebtToZeroWithActiveQuotasException`.

This creates a situation where users with active quotas cannot fully repay their debt in a single transaction, even if they have enough funds to do so. They would first need to remove all their quotas and then repay the debt, which requires multiple transactions and could lead to additional costs or complications.

The vulnerability is in the logic flow: the function decides to set the debt to zero before checking if this is actually allowed given the account's quotas. This could lead to confusion and frustration for users who are trying to close their positions. The impact is medium as it affects usability but doesn't directly lead to loss of funds, and the likelihood is high as it would affect any user with active quotas trying to fully repay their debt.

Mitigation Analysis: This is a valid medium severity issue. The incorrect handling of quota interest could prevent users from fully repaying their debt in a single transaction.

10 of 26 Findings PriceOracleV3.sol PriceFeedValidationTrait.sol

Fixed Gas Limit in Price Feed Validation Leading to DoS

Medium Risk

Within the PriceFeedValidationTrait, the _validatePriceFeed function performs a static call to a price feed to retrieve a flag (skipPriceCheck) and imposes a hard-coded gas limit of 10,000 gas:

```
(bool success, bytes memory returnData) =
OptionalCall.staticCallOptionalSafe({
    target: priceFeed,
    data: abi.encodeWithSelector(IPriceFeed.skipPriceCheck.selector),
    gasAllowance: 10_000
});
```

This fixed gas stipend might be insufficient in scenarios where the target price feed contract requires a bit more gas to execute its logic. In such cases, even a correctly functioning price feed could fail the validation, causing operations relying on up-to-date price data to revert. An attacker or a misconfiguration in an upgraded price feed could exploit this mechanism to induce a denial of service in the protocol by causing price checks to fail unexpectedly.

Mitigation Analysis: This is a valid medium severity issue. The fixed gas limit could lead to DoS if price feeds require more gas.

First depositor can front-run deposit ratio

Low Risk

In the current implementation of the Pool, there is no allocation of a small number of shares to seed the pool at deployment time. This allows the first depositor to potentially gain a disproportionately large share of the pool by depositing a small amount of underlying. The next depositor, who deposits significantly more underlying, might receive fewer shares than expected, effectively offering an arbitrage advantage to the initial depositor. The code snippet below demonstrates the deposit function:

```
function deposit(uint256 assets, address receiver) public override returns
(uint256 shares) {
    ...
    if (assetsReceived == 0 || shares == 0) {
        revert AmountCantBeZeroException();
    }
    // no special seeding is done to mitigate the first depositor advantage
    _deposit(receiver, assets, assetsReceived, shares);
}
```

Because shares are priced proportionally to the pool's total assets and supply, the first depositor can permanently hold a larger fraction of total shares than is typical in ERC4626 design if no mitigating step is taken.

Mitigation Analysis: This is a valid issue. The first depositor front-running attack is a known vulnerability in ERC4626 implementations.

Potential key collision in PriceOracleV3 due to assembly usage in _getTokenReserveKey

Low Risk

PriceOracleV3 stores reserve price feed parameters in a mapping keyed by a computed address derived via inline assembly. The function _getTokenReserveKey is defined as follows:

```
function _getTokenReserveKey(address token) internal pure returns (address key) {
    assembly {
        mstore(0x0, or("RESERVE", shl(0x28, token)))
        key := keccak256(0x0, 0x1b)
    }
}
```

This nonstandard mechanism for generating a unique key from a token address depends on the precise implementation of the bitwise operations and the constant string literal. If an attacker is able to craft a token address that causes the computed key to collide with that of another token's reserve key, it may lead to an unintended overwrite or misconfiguration of reserve price feed parameters. Although the probability of such a collision is extremely low due to the use of keccak256, the nonstandard implementation and reliance on inline assembly introduces some risk if not handled with extreme care.

Precision Loss in Linear Interest Rate Model Calculations

• Low Risk

The LinearInterestRateModelV3 contract calculates borrow rates based on pool utilization using a piecewise linear approach with utilization values expressed in WAD and interest rates in RAY. The mathematical operations involved, including multiplications and divisions, may incur precision loss, especially when utilization values approach the defined thresholds (U_1 and U_2). Although such precision loss is unlikely to have a dramatic impact on most operations, it could result in minor discrepancies in interest rate computations that affect borrowing costs or yields under extreme conditions.

Example snippet:

```
uint256 U_WAD = (WAD * (expectedLiquidity - availableLiquidity)) /
expectedLiquidity;

if (U_WAD <= U_1_WAD) {
    return R_base_RAY + ((R_slope1_RAY * U_WAD) / U_1_WAD);
}

if (U_WAD <= U_2_WAD) {
    return R_base_RAY + R_slope1_RAY + (R_slope2_RAY * (U_WAD - U_1_WAD)) /
(U_2_WAD - U_1_WAD);
}

return R_base_RAY + R_slope1_RAY + R_slope2_RAY + R_slope3_RAY * (U_WAD -
U_2_WAD) / (WAD - U_2_WAD);
```

While the precision loss does not compromise security, it could lead to slight errors in borrow rate determination under edge-case scenarios.

Mitigation Analysis: This is a valid precision loss issue. While it's unlikely to have significant impact, it's correctly classified as Low severity.

Lack of validation for `collateralHints` in `fullCollateralCheck` function

• Low Risk

In the `fullCollateralCheck` function of the CreditManagerV3 contract, there is a vulnerability related to the handling of `collateralHints`:

```
function fullCollateralCheck(
    address creditAccount,
    uint256 enabledTokensMask,
    uint256[] calldata collateralHints,
    uint16 minHealthFactor,
    bool useSafePrices
)
    external
    override
    nonReentrant // U:[CM-5]
    creditFacadeOnly // U:[CM-2]
    returns (uint256)
{
    CollateralDebtData memory cdd = _calcDebtAndCollateral({
        creditAccount: creditAccount,
        minHealthFactor: minHealthFactor,
        collateralHints: collateralHints,
        enabledTokensMask: enabledTokensMask,
        task: CollateralCalcTask.FULL_COLLATERAL_CHECK_LAZY,
        useSafePrices: useSafePrices
    }); // U:[CM-18]
    // ...
}
```

The function accepts an array of `collateralHints` without any validation and passes it directly to the `_calcDebtAndCollateral` function. These hints are used to optimize the order in which collateral tokens are checked, but if invalid hints are provided, it could lead to unexpected behavior or inefficiencies.

While the CreditFacadeV3 contract does validate `collateralHints` in its `_setFullCheckParams` function, there's no guarantee that all calls to `fullCollateralCheck` will go through this validation. If a future version of the credit facade or another contract with the credit facade role calls this function with invalid hints, it could lead to issues.

The vulnerability is relatively minor as it would likely only result in inefficiencies rather than direct security issues, but it still represents a potential risk. The impact is low as it would

primarily affect gas usage, and the likelihood is low as it requires a contract with the credit facade role to provide invalid hints.

Mitigation Analysis: This is a valid low severity issue. Lack of validation for collateralHints could lead to inefficiencies.

Potential precision loss in `_calcPartialLiquidationPayments` function

• Low Risk

In the `_calcPartialLiquidationPayments` function of the CreditFacadeV3 contract, there is a potential precision loss issue:

```
function _calcPartialLiquidationPayments(uint256 amount, address token, bool
isExpired)
    internal
    view
    returns (uint256 repaidAmount, uint256 feeAmount, uint256 seizedAmount)
{
    address priceOracle = ICreditManagerV3(creditManager).priceOracle();
    (
        ,
        uint16 feeLiquidation,
        uint16 liquidationDiscount,
        uint16 feeLiquidationExpired,
        uint16 liquidationDiscountExpired
    ) = ICreditManagerV3(creditManager).fees();
    seizedAmount = IPriceOracleV3(priceOracle).convert(amount, underlying,
token) * PERCENTAGE_FACTOR
        / (isExpired ? liquidationDiscountExpired : liquidationDiscount); //
U:[FA-15]
    feeAmount = amount * (isExpired ? feeLiquidationExpired : feeLiquidation)
/ PERCENTAGE_FACTOR; // U:[FA-15]
    unchecked {
        // unchecked subtraction is safe because credit configurator ensures
that liquidation fee is below 100%
        repaidAmount = amount - feeAmount; // U:[FA-15]
    }
}
```

The function calculates the amount of collateral to seize by converting the repaid amount from the underlying token to the collateral token and then dividing by the liquidation discount. This division could lead to precision loss, especially for tokens with different decimal places.

Additionally, the function calculates the fee amount by multiplying the repaid amount by the fee percentage and then dividing by `PERCENTAGE_FACTOR`. This division could also lead to precision loss, especially for small amounts.

While these precision losses are generally small and unlikely to cause significant issues, they could still lead to slightly unfair liquidations where liquidators receive slightly more or less

collateral than they should. The impact is low as the precision loss would be minimal, and the likelihood is medium as it would affect all partial liquidations to some degree.

Mitigation Analysis: This is a valid low severity issue. Potential precision loss in calculations could lead to slightly unfair liquidations.

16 of 26 Findings

contracts/credit/CreditManagerV3.sol

Deprecated OpenZeppelin Function Usage

Low Risk

The contract `CreditManagerV3.sol` uses the deprecated `safeApprove` function from OpenZeppelin's libraries:

```
CreditAccountHelper.safeApprove({creditAccount: creditAccount, token: token,
spender: spender, amount: amount}); // U:[CM-15]
```

OpenZeppelin has deprecated several functions, including `safeApprove`, and replaced them with newer versions. The `safeApprove` function can lead to issues when trying to change an existing allowance from a non-zero value to another non-zero value, as it requires resetting the allowance to zero first.

Consider using `safeIncreaseAllowance` and `safeDecreaseAllowance` instead, or the newer `forceApprove` function which was designed to replace `safeApprove`.

Mitigation Analysis: This is a valid low severity issue. Using deprecated OpenZeppelin functions could lead to issues when changing allowances.

17 of 26 Findings

contracts/core/GearStakingV3.sol

contracts/libraries/CreditAccountHelper.sol

Unsafe ERC20 Operation Usage

• Low Risk

The contracts use direct ERC20 operations without safety checks, which can lead to unexpected behavior:

In GearStakingV3.sol:

```
IERC20(_gear()).approve(successor, uint256(amount));
```

In CreditAccountHelper.sol:

```
try ICreditAccountV3(creditAccount).execute(token,
abi.encodeCall(IERC20.approve, (spender, amount))) returns (
```

Direct ERC20 operations can be problematic because:

1. Some tokens don't return a boolean value as expected
2. Some tokens may revert on failure instead of returning false
3. Some tokens like USDT return non-standard values

It is recommended to use OpenZeppelin's SafeERC20 library which handles these edge cases properly by wrapping the operations in a way that works with non-compliant tokens.

Mitigation Analysis: This is a valid low severity issue. Unsafe ERC20 operations could lead to unexpected behavior with non-compliant tokens.

18 of 26 Findings

contracts/core/AliasedLossPolicyV3.sol contracts/core/BotListV3.sol
contracts/core/DefaultAccountFactoryV3.sol contracts/core/GearStakingV3.sol
contracts/core/PriceOracleV3.sol contracts/credit/CreditAccountV3.sol
contracts/credit/CreditConfiguratorV3.sol contracts/credit/CreditFacadeV3.sol
contracts/credit/CreditManagerV3.sol contracts/credit/CreditManagerV3_USDT.sol
contracts/libraries/Constants.sol contracts/pool/GaugeV3.sol
contracts/pool/LinearInterestRateModelV3.sol contracts/pool/PoolQuotaKeeperV3.sol
contracts/pool/PoolV3.sol contracts/pool/PoolV3_USDT.sol
contracts/pool/TumblerV3.sol contracts/traits/ACLTrait.sol
contracts/traits/ContractsRegisterTrait.sol contracts/traits/PriceFeedValidationTrait.sol
contracts/traits/ReentrancyGuardTrait.sol contracts/traits/SanityCheckTrait.sol
contracts/traits/USDT_Transfer.sol

Non-Specific Solidity Pragma Version

• Low Risk

All contracts in the codebase use a floating pragma version:

```
pragma solidity ^0.8.17;
```

Using a floating pragma (`^`) allows the contracts to be compiled with any compiler version starting from 0.8.17 up to (but not including) 0.9.0. This introduces several risks:

1. Different compiler versions may introduce different bugs or optimizations that could affect the behavior of the contract
2. It makes it harder to ensure reproducible builds across different environments
3. Newer compiler versions might introduce breaking changes that could affect the contract's behavior

It is recommended to lock the pragma to a specific version (e.g., `pragma solidity 0.8.17;`) to ensure consistent compilation results and reduce the risk of unexpected behavior due to compiler differences.

Mitigation Analysis: This is a valid low severity issue. Non-specific Solidity pragma versions can lead to inconsistent compilation results.

19 of 26 Findings

contracts/credit/CreditFacadeV3.sol contracts/libraries/CollateralLogic.sol contracts/libraries/MarketHelper.sol

Magic Numbers Instead Of Constants

Low Risk

Several contracts use magic numbers (hardcoded literal values) instead of named constants:

In CreditFacadeV3.sol:

```
_setFullCheckParams(fullCheckParams, mcall.callData[4:]); // U:[FA-24]
```

The value `4` appears multiple times, representing the byte offset in calldata.

In CollateralLogic.sol:

```
lt = uint16(packedQuota >> 96);
```

The value `96` represents a bit shift amount for unpacking data.

In MarketHelper.sol:

```
if (_version(pool) < 3_10) return _marketConfigurator(pool).treasury();
```

The value `3_10` represents a version number.

Using magic numbers makes the code less readable and more error-prone. If these values need to be changed in the future, developers would need to find and update all occurrences, which could lead to inconsistencies.

Define named constants for these values to improve code readability, maintainability, and reduce the risk of errors when values need to be updated.

Mitigation Analysis: This is a valid low severity issue. Magic numbers make the code less readable and more error-prone.

20 of 26 Findings

contracts/credit/CreditFacadeV3.sol contracts/credit/CreditManagerV3.sol contracts/pool/PoolV3.sol

Incorrect `nonReentrant` modifier Placement

• Low Risk

The `nonReentrant` modifier is not placed as the first modifier in several functions across multiple contracts:

In CreditFacadeV3.sol:

```
nonReentrant // U:[FA-4]
```

In CreditManagerV3.sol:

```
nonReentrant // U:[CM-5]
```

In PoolV3.sol:

```
nonReentrant // U:[LP-2B]
```

When the `nonReentrant` modifier is not placed first in the list of modifiers, other modifiers will execute before the reentrancy check. This creates a potential vulnerability window where reentrancy could occur within those other modifiers.

As a best practice, the `nonReentrant` modifier should always be the first modifier in the list to ensure that the reentrancy check is performed before any other code executes. This prevents potential reentrancy attacks that could exploit logic in other modifiers.

Mitigation Analysis: This is a valid low severity issue. Incorrect modifier placement could create a potential vulnerability window.

Unnecessary Modifier Usage

• Low Risk

The contracts define modifiers that are used only once or could be replaced with a simple require statement:

In GearStakingV3.sol:

```
modifier migratorOnly() {
```

In CreditAccountV3.sol:

```
modifier factoryOnly() {
```

Creating modifiers for access control checks that are only used once or twice increases the contract size and gas costs unnecessarily. These modifiers likely contain simple checks that could be implemented directly in the functions that use them.

Consider replacing these modifiers with direct require statements in the functions where they are used, especially if they are only used once. This would reduce contract size and potentially save gas during deployment and execution.

Mitigation Analysis: This is a valid low severity issue. Unnecessary modifiers increase contract size and gas costs.

Empty Code Block Detection

• Low Risk

The `DefaultAccountFactoryV3.sol` contract contains an empty function implementation:

```
function serialize() external view override returns (bytes memory) {}
```

Empty code blocks can indicate incomplete implementations, forgotten code, or unnecessary functions. They can also be confusing for developers trying to understand the contract's behavior.

If this function is intentionally empty (perhaps because it's required by an interface but not needed in this implementation), consider adding a comment explaining why it's empty. If it's meant to be implemented but was overlooked, complete the implementation. If it's truly unnecessary, consider removing it if possible.

Mitigation Analysis: This is a valid low severity issue. Empty code blocks can indicate incomplete implementations.

23 of 26 Findings

contracts/credit/CreditConfiguratorV3.sol contracts/credit/CreditFacadeV3.sol contracts/traits/PriceFeedValidationTrait.sol

Unoptimized Numeric Literal Format

Low Risk

The contracts use underscore notation for numeric literals instead of scientific notation:

In CreditConfiguratorV3.sol:

```
gasAllowance: 30_000
```

In CreditFacadeV3.sol:

```
gasAllowance: 30_000
```

In PriceFeedValidationTrait.sol:

```
gasAllowance: 10_000
```

While using underscores for readability is good practice, for certain values (especially those with many zeros), scientific notation (`e` notation) can be more readable and less error-prone. For example, `1e18` is clearer and less prone to miscounting zeros than `1000000000000000000`.

For the specific examples shown, the values are relatively small, so the current format may be acceptable. However, for consistency and future-proofing, consider using scientific notation for larger values throughout the codebase.

Mitigation Analysis: This is a valid low severity issue. Unoptimized numeric literal format can be less readable for large values.

24 of 26 Findings

contracts/core/AliasedLossPolicyV3.sol contracts/core/GearStakingV3.sol
contracts/credit/CreditConfiguratorV3.sol contracts/credit/CreditFacadeV3.sol
contracts/credit/CreditManagerV3.sol contracts/pool/GaugeV3.sol
contracts/pool/PoolQuotaKeeperV3.sol contracts/pool/TumblerV3.sol

Loop contains `require` / `revert` statements

• Low Risk

Multiple contracts contain loops with `require` or `revert` statements inside them:

```
// Examples from various contracts
while (remainingTokensMask != 0) { ... }
for (uint256 i = 0; i < len;) { ... }
for (uint256 i; i < num; ++i) { ... }
```

Using `require` or `revert` statements inside loops can cause the entire transaction to fail if a single item fails to meet the condition. This creates an "all or nothing" approach that may not be desirable in many cases.

A better pattern is to implement a "forgive and continue" approach where:
1. The loop tracks which items failed processing
2. The loop continues processing remaining items even if some fail
3. After the loop completes, the function returns information about which items failed

This approach allows partial success and gives users more detailed feedback about what went wrong, rather than reverting the entire transaction.

Mitigation Analysis: This is a valid low severity issue. Using require/revert in loops creates an 'all or nothing' approach.

Incorrect Order of Division and Multiplication

• Low Risk

The contract performs division before multiplication, which can lead to precision loss:

```
? quotaChange / int96(uint96(PERCENTAGE_FACTOR)) *
int96(uint96(PERCENTAGE_FACTOR))
```

In Solidity's integer arithmetic, division operations truncate any remainder. When division is performed before multiplication, this truncation can lead to significant precision loss.

For example, if `quotaChange` is 10 and `PERCENTAGE_FACTOR` is 100, the expression would evaluate as:
1. `10 / 100 = 0` (integer division truncates to 0)
2. `0 * 100 = 0`

The correct approach would be to perform multiplication before division to preserve precision:

```
? quotaChange * int96(uint96(PERCENTAGE_FACTOR)) /
int96(uint96(PERCENTAGE_FACTOR))
```

However, in this specific case, it appears the code is intentionally checking if the value is divisible by `PERCENTAGE_FACTOR` without a remainder. If that's the case, the code should be commented to clarify this intention.

Mitigation Analysis: This is a valid low severity issue. Division before multiplication can lead to precision loss.

Lack of timelock on Tumbler’s setRate(...) function

Best Practices

The Tumbler contract allows a privileged configurator to directly set any quota interest rate for tokens, and can do so at any time without a timelock. Because these rates can have a large impact on the leverage mechanics of the protocol, having a governance-controlled or configurable timelock might reduce the risk of accidental or malicious changes.

```
function setRate(address token, uint16 rate) external override
configuratorOnly {
    if (!_tokensSet.contains(token)) {
        revert TokenIsNotQuotedException();
    }
    if (rate == 0) {
        revert IncorrectParameterException();
    }
    _setRate(token, rate);
}
```

Disclaimer

Kindly note, the Audit Agent is currently in beta stage and no guarantee is being given as to the accuracy and/or completeness of any of the outputs the Audit Agent may generate, including without limitation this Report. The results set out in this Report may not be complete nor inclusive of all vulnerabilities. The Audit Agent is provided on an 'as is' basis, without warranties or conditions of any kind, either express or implied, including without limitation as to the outputs of the code scan and the security of any smart contract verified using the Audit Agent.

Blockchain technology remains under development and is subject to unknown risks and flaws. This Report does not indicate the endorsement of any particular project or team, nor guarantee its security. Neither you nor any third party should rely on this Report in any way, including for the purpose of making any decisions to buy or sell a product, service or any other asset.

To the fullest extent permitted by law, Nethermind disclaims any liability in connection with this Report, its content, and any related services and products and your use thereof, including, without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement. Nethermind does not warrant, endorse, guarantee, or assume responsibility for any product or service advertised or offered by a third party through the product, any open source or third-party software, code, libraries, materials, or information linked to, called by, referenced by or accessible through the report, its content, and the related services and products, any hyperlinked websites, any websites or mobile applications appearing on any advertising, and Nethermind will not be a party to or in any way be responsible for monitoring any transaction between you and any third-party providers of products or services. As with the purchase or use of a product or service through any medium or in any environment, you should use your best judgment and exercise caution where appropriate.

FOR AVOIDANCE OF DOUBT, THE REPORT, ITS CONTENT, ACCESS, AND/OR USAGE THEREOF, INCLUDING ANY ASSOCIATED SERVICES OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, INVESTMENT, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.