SHERLOCK

# Security Review For dHEDGE

Collaborative Audit Prepared For: **dHEDGE**
Lead Security Expert(s): **pkqs90**
**santipu_**
**xiaoming90**
Date Audited: **December 18 - December 29, 2025**

# Introduction

DYTM is a protocol designed to simplify leverage/margin trading using any DeFi protocol. Anyone can create a credit market tailored to their needs (yes permissionless). Think of it as a protocol which provides credit to use across different DeFi platforms.

At the core, DYTM is pretty similar to any other lending protocol. One of the major difference is that it allows for a high level of customization and flexibility. DYTM is designed to be modular and extensible, allowing developers to build on top of it and create new features and functionalities. At its core, all that a market does is to allow a user to lend and borrow specific assets. They can re-supply the assets they borrowed (assuming they swapped it for some other asset) as long as the re-supplied assets are supported by the market. They can choose to either lend their assets or escrow them (not subject to lending risks) as collateral for any asset(s) that they borrow.

## Scope

Repository: dhedge/DYTM

Audited Commit: 6c2f86a80871753263eb5d5f4287dcb46d84bd56

Final Commit: a96d7cbb86661298c0b3e8a41355213f3952ef4d

Files:

- src/abstracts/Registry.sol
- src/abstracts/storages/Context.sol
- src/abstracts/storages/OfficeStorage.sol
- src/abstracts/storages/TransientEnumerableHashTableStorage.sol
- src/extensions/delegatees/AccountSplitterAndMerger.sol
- src/extensions/delegatees/OwnableDelegatee.sol
- src/extensions/delegatees/SimpleDelegatee.sol
- src/extensions/hooks/AddressAccountBaseWhitelist.sol
- src/extensions/hooks/BaseHook.sol
- src/extensions/hooks/BorrowerWhitelist.sol
- src/extensions/hooks/SimpleAccountWhitelist.sol
- src/extensions/market-configs/SimpleMarketConfig.sol
- src/extensions/market-configs/WhitelistedTransfersMarketConfig.sol
- src/extensions/oracles/BaseOracleModule.sol
- src/extensions/oracles/dHEDGEPoolPriceAggregator.sol
- src/extensions/weights/SimpleWeights.sol

- src/interfaces/ParamStructs.sol
- src/IRM/FixedBorrowRateIRM.sol
- src/IRM/LinearKinkIRM.sol
- src/libraries/Constants.sol
- src/libraries/HooksCallHelpers.sol
- src/libraries/SharesMathLib.sol
- src/libraries/StorageAccessors.sol
- src/libraries/TokenHelpers.sol
- src/libraries/Utils.sol
- src/Office.sol
- src/periphery/DYTMPeriphery.sol
- src/periphery/OfficeERC6909ToERC20Wrapper.sol
- src/periphery/WrappedERC6909ERC20.sol
- src/types/AccountId.sol
- src/types/MarketId.sol
- src/types/ReserveKey.sol
- src/types/Types.sol

## Final Commit Hash

**a96d7cbb86661298c0b3e8a41355213f3952ef4d**

## Findings

Each issue has an assigned severity:

- High issues are directly exploitable security vulnerabilities that need to be fixed.
- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- Low/Info issues are non-exploitable, informational findings that do not pose a security risk or impact the system's integrity. These issues are typically cosmetic or related to compliance requirements, and are not considered a priority for remediation.

## Issues Found

| High | Medium | Low/Info |
|---|---|---|
| 6 | 19 | 12 |

## Issues Not Fixed and Not Acknowledged

| High | Medium | Low/Info |
|---|---|---|
| 0 | 0 | 0 |

# Issue H-1: Unliquidatable positions due to overflow in `_getQuote` [RESOLVED]

Source: https://github.com/sherlock-audit/2025-12-dhedge-dytm-dec-18th/issues/34

## Summary

An attacker can create positions that cannot be liquidated due to an overflow error in `_getQuote`.

## Vulnerability Detail

The `_getQuote` function in `dHEDGEPoolPriceAggregator` converts a base asset amount to its quote asset equivalent:

```
(uint256 basePriceUSD, uint256 baseScale) = _getPrice(base);
(uint256 quotePriceUSD, uint256 quoteScale) = _getPrice(quote);

return inAmount.mulDivDown(basePriceUSD * quoteScale, quotePriceUSD * baseScale);
```

This logic is vulnerable to overflow when the base asset is a dHEDGE pool, as `IdHEDGEPoolLogic::tokenPrice` returns prices with 18 decimals.

For instance, with the following values:

- `inAmount` = 120_000e18
- `basePriceUSD` = 1e18
- `quoteScale` = 1e36

The multiplication of these values exceeds `type(uint256).max`, triggering an overflow and reverting the transaction.

This flaw can be exploited to create unliquidatable positions. The `liquidate` function calls `isHealthyAccount`, which in turn invokes `getQuote` to convert collateral to USD. If `getQuote` overflows, `isHealthyAccount` reverts, preventing liquidation.

**Example scenario:**

1. Bob holds 120,000 shares of a dHEDGE pool, priced at 1 USD per share (`1e18`).
2. He opens a position using 115,000 shares as collateral and borrows the maximum allowed.
3. He then opens a second position with 5,000 shares and transfers them to the first position. This does not trigger a health check on the first position.
4. As a result, the first position becomes unliquidatable, as `isHealthyAccount` will revert due to the overflow in `_getQuote`.

**Note**: If the position becomes profitable, Bob can close it by repaying the debt and withdrawing the collateral.

## Impact

Attackers can create unliquidatable positions using dHEDGE pool shares as collateral. These positions can accumulate bad debt that cannot be redistributed to the rest of the market.

## Code Snippet

https://github.com/sherlock-audit/2025-12-dhedge-dytm-dec-18th/blob/main/DYTM/src/extensions/oracles/dHEDGEPoolPriceAggregator.sol#L89

## Tool Used

Manual Review

## Recommendation

Update `_getQuote` to prevent overflow when processing large values.

# Issue H-2: The default weight for the same asset allows creating bad debt [RESOLVED]

Source: https://github.com/sherlock-audit/2025-12-dhedge-dytm-dec-18th/issues/40

## Summary

When the collateral and debt token are the same, `SimpleWeights` assigns a default weight of `1e18`, enabling attackers to generate bad debt and drain liquidity from a market.

## Vulnerability Detail

The `getWeight` function in `SimpleWeights` returns a default weight of `1e18` when the asset used as collateral is the same as the asset borrowed:

```
    function getWeight(
        AccountId /* account */,
        uint256 collateralTokenId,
        ReserveKey debtAsset
    )
        external
        view
        returns (uint64 weight)
    {
        // If the assets are the same, return 1 as the weight.
        if (collateralTokenId.getAsset() == debtAsset.getAsset()) {
>>          return uint64(WAD);
        }

        // ...
    }
```

This default weight allows a user to fully borrow against the same asset used as collateral. The position becomes liquidatable after a small amount of interest accrues, and liquidation rewards enable an attacker to drain value from the system.

**Example scenario:**

1. An attacker deposits 100 USDC as collateral and borrows 100 USDC.
2. After one block, interest accrues, making the position eligible for liquidation.
3. The attacker liquidates their own position, earning a liquidation bonus (e.g., 5 USDC).
4. The system is left with 5 USDC of bad debt, absorbed by other lenders.

This attack can be repeated infinitely to drain the market of liquidity.

## Impact

Any participant can repeatedly generate bad debt and extract value via liquidation rewards, compromising market stability and draining liquidity.

## Code Snippet

https://github.com/sherlock-audit/2025-12-dhedge-dytm-dec-18th/blob/main/DYTM/src/extensions/weights/SimpleWeights.sol#L61

## Tool Used

Manual Review

## Recommendation

Remove the default weight of `1e18` for same-asset collateral-debt pairs. Enforce explicit configuration of weights and document that using `1e18` is always unsafe.

# Issue H-3: Collateral could be taken without repaying debts [RESOLVED]

Source: https://github.com/sherlock-audit/2025-12-dhedge-dytm-dec-18th/issues/51

## Vulnerability Detail

Assume that the collateral is USDC (6 decimals) where the price is $1, and the debt is WBTC (8 decimals) where the price is $100,000.

Following is the oracle math that calculates the `assetsRepaid` (asset that the liquidator needs to repay)

- USDC (in)(collateral): `basePriceUSD=1e8`, `baseScale=10(6+8)=1e14`
- WBTC (out)(debt): `quotePriceUSD=100000e8=1e13`, `quoteScale=10(8+8)=1e16`

$$\text{out} = \left\lfloor \text{in} \cdot \frac{1e8 \cdot 1e16}{1e13 \cdot 1e14} \right\rfloor = \left\lfloor \frac{in}{1000} \right\rfloor$$

So, if $\text{in} < 1000$, then $\text{out}$ will be zero, which means that the liquidator doesn't need to repay.

The maximum $\text{in}$ is 999, which is equal to 0.000999 USDC, or ~$0.001

Thus, for each liquidation call, the liquidator can extract ~$0.001. If the liquidator can bundle multiple calls and repeats this for 1,000,000 times, they will be able to extract ~$1000.

This will be profitable if the gas cost is less than the profit. If the protocol is deployed on L2 chain (e.g., Base) where gas is cheap, the chance of a profitable attack will increase. The issue will be aggravated if:

- The debt token's price increase (e.g., WBTC increase to 200,000 USD) OR the collateral token's price decrease
- The debt token's decimals is smaller OR the collateral token's decimals is larger

## Impact

It is possible for the liquidator to take collateral, but repay nothing under certain conditions.

## Code Snippet

https://github.com/sherlock-audit/2025-12-dhedge-dytm-dec-18th/blob/75a66499b7be9b8b9fb8518a798e52e4e0043f43/DYTM/src/Office.sol#L387

## Tool Used

Manual Review

## Recommendation

Consider having an invariant thatt if the assets repaid == 0, but collateral received > 0, it should revert as this indicates someone is attempting to extract value without repaying anything.

# Issue H-4: Overcollateral position can be liquidated [RESOLVED]

Source: https://github.com/sherlock-audit/2025-12-dhedge-dytm-dec-18th/issues/55

## Vulnerability Detail

The protocol will mark a position as unhealthy and subject it to liquidation even if it is overcollateralized, but its collateral value falls below the `minMarginAmountUSD`.

```
File: Office.sol
572:     function isHealthyAccount(AccountId account, MarketId market) public view
 ↪  returns (bool isHealthy) {
..SNIP..
625:             // If at any point the weighted collateral value equals/exceeds
 ↪  the debt value AND
626:             // the weighted collateral value exceeds the minimum margin amount
 ↪  in USD for the market,
627:             // we can conclude that the account is healthy and return early.
628:             // If the account isn't healthy even after accounting for the last
 ↪  collateral asset,
629:             // the function will return false as it never entered this block.
630:             if (weightedCollateralValueUSD >= debtValueUSD &&
 ↪  weightedCollateralValueUSD >= minMarginAmountUSD) {
631:                 return true;
632:             }
```

Assume that:

- Weight of USDC<>WETH = 0.8
- Price of WETH = 2000USD
- minMarginAmountUSD = 100 USD

Bob supplied 1 WETH, and his position's collateral value is 1600 USD (2000 * 0.8). Bob proceeds to borrow 50 USDC, and his position's debt value is 50 USD.

WETH price drops significantly (can also be other more volatile collateral assets) to 120 USD.

Bob's position's collateral value = 96 USD (120 * 0.8), while his debt value is still 50 USD. His position/account is still overcollateralized, but the protocol deems it unhealthy and liquidates it.

## Impact

The user's position will be liquidated earlier than expected, even though it is overcollateralized and poses no insolvency risk to the system, leading to them losing some collateral due to the liquidation penalty/bonus required to be paid to the liquidator.

## Code Snippet

https://github.com/sherlock-audit/2025-12-dhedge-dytm-dec-18th/blob/75a66499b7be9b8b9fb8518a798e52e4e0043f43/DYTM/src/Office.sol#L630

## Tool Used

Manual Review

## Recommendation

Consider only an undercollateralized account (collateral value * LTV-weight < debt value) to be eligible for liquidation.

# Issue H-5: Missing token validation allows migration of non-existent funds [RESOLVED]

Source: https://github.com/sherlock-audit/2025-12-dhedge-dytm-dec-18th/issues/57

## Summary

The `Office::migrateSupply` function allows arbitrary creation of collateral tokens by setting different token addresses for `fromTokenId` and `toTokenId`.

## Vulnerability Detail

The `migrateSupply` function in the Office contract enables users to withdraw collateral from one market and deposit it into another market:

```
function migrateSupply(MigrateSupplyParams calldata params)
        external
        onlyAuthorizedCaller(params.account)
        returns (uint256 assetsRedeemed, uint256 newSharesMinted)
    {
        // ...

        // Withdraw the assets from the `fromTokenId` reserve and supply them to
        ↪  the `toTokenId` reserve.
        uint256 sharesRedeemed;
>>      (assetsRedeemed, sharesRedeemed) = _withdraw({
            marketConfig: fromMarketConfig,
            account: params.account,
            tokenId: params.fromTokenId,
            assets: params.assets,
            shares: params.shares
        });
>>      newSharesMinted = _supply({
            marketConfig: toMarketConfig, tokenId: params.toTokenId, account:
            ↪  params.account, assets: assetsRedeemed
        });

        // ...
    }
```

There is no validation to ensure that the token addresses represented by `params.fromTokenId` and `params.toTokenId` are the same. This enables an attacker to specify different token addresses, causing the system to withdraw one token and deposit a different one.

For example, an attacker could call `migrateSupply` to withdraw 1e18 DAI and deposit 1e18 WETH, effectively exchanging 1 DAI for 1 WETH without incurring any cost. Then, the

attacker can simply withdraw the WETH, swap it for more DAI, and repeat the attack until draining the contract of all WETH.

## Impact

An attacker can generate free collateral by migrating from a low-value token to a high-value token.

This exploit can be repeated infinitely in the same tx and drain the liquidity from all markets.

## Code Snippet

https://github.com/sherlock-audit/2025-12-dhedge-dytm-dec-18th/blob/main/DYTM/src/Office.sol#L480

## Tool Used

Manual Review

## Recommendation

Add a validation check to ensure that the token addresses corresponding to `fromTokenId` and `toTokenId` are identical.

# Issue H-6: Liquidations can seize collateral from other markets [RESOLVED]

Source: https://github.com/sherlock-audit/2025-12-dhedge-dytm-dec-18th/issues/58

## Summary

The liquidation process does not validate that seized collateral belongs to the same market as the liquidated debt. This allows liquidators to take collateral from unrelated, healthy positions in other markets.

## Vulnerability Detail

The liquidate function allows liquidators to specify an array of `tokenId` values to seize collateral in order to repay debt:

```
function liquidate(LiquidationParams calldata params) external returns (uint256
↪  assetsRepaid) {
        // ...
        {
            IOracleModule oracleModule = marketConfig.oracleModule();

            for (uint256 i; i < params.collateralParams.length; ++i) {
                IERC20 collateralAsset =
                ↪  params.collateralParams[i].tokenId.getAsset();

                (uint256 assets,) = _withdraw({
                    marketConfig: marketConfig,
                    account: params.account,
>>                  tokenId: params.collateralParams[i].tokenId,
                    assets: params.collateralParams[i].assets,
                    shares: params.collateralParams[i].shares
                });

                // ...
            }
        }
```

There is no check to ensure that the `tokenId` specified in each `collateralParams` entry belongs to the same market as the debt being repaid. As a result, a liquidator can seize collateral from different markets where the user may have unrelated, healthy positions.

## Impact

This breaks the market isolation assumption, allowing liquidators to extract collateral from positions that are not undercollateralized in the targeted market.

## Code Snippet

https://github.com/sherlock-audit/2025-12-dhedge-dytm-dec-18th/blob/main/DYTM/src/Office.sol#L378

## Tool Used

Manual Review

## Recommendation

Ensure that all seized collateral token IDs correspond to the same market as the one in which the liquidation is occurring.

# Issue M-1: Pending interest charged at incorrect fee when updating performance rate [RESOLVED]

Source: https://github.com/sherlock-audit/2025-12-dhedge-dytm-dec-18th/issues/37

## Summary

When updating the performance fee in `SimpleMarketConfig`, any pending interest that has not yet been accrued will be charged using the new fee rather than the previous one.

## Vulnerability Detail

The `SimpleMarketConfig` contract exposes `setPerformanceFee` to update the performance fee applied to accrued interest in the `Office` contract:

```
function setPerformanceFee(uint64 newFeePercentage) external onlyOwner {
    uint64 oldFeePercentage = feePercentage;

    require(newFeePercentage <= WAD,
    ↪  SimpleMarketConfig__InvalidPercentage(newFeePercentage));

    feePercentage = newFeePercentage;

    emit SimpleMarketConfig__PerformanceFeeModified(newFeePercentage,
    ↪  oldFeePercentage);
}
```

If this function is called while there is pending, unaccrued interest, the entire pending amount will be charged at the new fee rate. This effectively allows the performance fee to apply retroactively.

## Impact

All pending interest will be accrued using the updated performance fee.

## Code Snippet

https://github.com/sherlock-audit/2025-12-dhedge-dytm-dec-18th/blob/main/DYTM/src/extensions/market-configs/SimpleMarketConfig.sol#L161

## Tool Used

Manual Review

## Recommendation

Update `setPerformanceFee` to call `accrueInterest` before modifying the fee, ensuring interest is accounted under the correct rate.

# Issue M-2: Interest should be accrued before updating the IRM [RESOLVED]

Source: https://github.com/sherlock-audit/2025-12-dhedge-dytm-dec-18th/issues/38

## Summary

If the interest rate model (IRM) is updated in `SimpleMarketConfig` while interest is pending accrual, the new IRM will be applied retroactively, resulting in incorrect interest calculations.

## Vulnerability Detail

The setIRM function allows the contract owner to update the IRM used for interest calculations:

```
function setIRM(IIRM newIrm) external onlyOwner {
    IIRM oldIrm = irm;

    require(address(newIrm) != address(0), SimpleMarketConfig__ZeroAddress());

    irm = newIrm;

    emit SimpleMarketConfig__IrmModified(newIrm, oldIrm);
}
```

If `setIRM` is called before interest has been accrued, the unaccrued portion will be calculated using the newly set IRM. This introduces retroactive behavior where past borrowing periods are charged based on future rate models.

## Impact

Interest is incorrectly accrued using the updated IRM instead of the one that was active during the accrual period.

## Code Snippet

https://github.com/sherlock-audit/2025-12-dhedge-dytm-dec-18th/blob/main/DYTM/src/extensions/market-configs/SimpleMarketConfig.sol#L209

## Tool Used

Manual Review

## Recommendation

Update `setIRM` to call `accrueInterest` before modifying the IRM, ensuring interest is accounted under the correct IRM.

# Issue M-3: Approvals flow is broken [RESOLVED]

Source: https://github.com/sherlock-audit/2025-12-dhedge-dytm-dec-18th/issues/39

## Summary

The current approvals implementation in `Registry` only permits spenders to transfer tokens to themselves. This breaks standard approval semantics and limits functionality.

## Vulnerability Detail

In the `Registry` contract, the `transferFrom` function deducts allowances using `_spendAllowance`. However, it incorrectly sets the `spender` parameter as the `receiver`:

```
    function transferFrom(
        AccountId sender,
        AccountId receiver,
        uint256 tokenId,
        uint256 amount
    )
        public
        virtual
        returns (bool success)
    {
        // ...
        if (
            (currentOwner != caller && !callerIsOperator)
                || (callerIsOperator && tokenId.getTokenType() ==
                ↪  TokenType.ISOLATED_ACCOUNT)
        ) {
            _spendAllowance({
>>              currentOwner: currentOwner, from: sender, spender: receiver,
↪   tokenId: tokenId, amount: amount
            });
        }

        // ...
    }
```

This misassignment leads to a failure when `_spendAllowance` is executed. The function expects the spender (the receiver) to be approved by the owner due to the `onlyAuthorizedCaller` modifier:

```
function _spendAllowance(
    address currentOwner,
    AccountId from,
    AccountId spender,
```

```
    uint256 tokenId,
    uint256 amount
)
    internal
    virtual
    onlyAuthorizedCaller(spender)
{
```

As a result, `transferFrom` only works when the spender is also the receiver. If an approved address attempts to transfer tokens to another address, the transaction reverts.

## Impact

Approved addresses cannot transfer tokens to third parties, rendering the approval mechanism functionally broken and inconsistent with expected behavior.

## Code Snippet

https://github.com/sherlock-audit/2025-12-dhedge-dytm-dec-18th/blob/main/DYTM/src/abstracts/Registry.sol#L225

## Tool Used

Manual Review

## Recommendation

Update `Registry::transferFrom` to pass `msg.sender` as the spender when calling `_spendAllowance`. Also, consider removing the `onlyAuthorizedCaller` modifier from `_spendAllowance`, as it enforces a constraint already covered by the caller context.

# Issue M-4: Flashloan enables profitable share-ratio arbitrage on liquidatable ERC4626 vaults [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit/2025-12-dhedge-dytm-dec-18th/issues/41

This issue has been acknowledged by the team but won't be fixed at this time.

## Vulnerability Detail

`Office` exposes a `flashloan()` function allowing any token held by the protocol to be borrowed and returned within a single transaction:

https://github.com/sherlock-audit/2025-12-dhedge-dytm-dec-18th/blob/main/DYTM/src/Office.sol#L551-L560

```
function flashloan(IERC20 token, uint256 assets, bytes calldata callbackData)
↪  external {
    token.safeTransfer({to: msg.sender, value: assets});
    IFlashloanReceiver(msg.sender).onFlashloanCallback({assets: assets,
    ↪  callbackData: callbackData});
    token.safeTransferFrom({from: msg.sender, to: address(this), value: assets});
}
```

When the flashloaned token is an ERC4626-style vault share (e.g., dHedgePool tokens) whose underlying asset/share ratio can be decreased within the same transaction, such as by triggering a liquidation on an unhealthy position–an attacker can exploit this:

1. Flashloan vault shares from `Office`.
2. Redeem shares -> underlying assets at the current (higher) ratio.
3. Trigger liquidation on the vault, lowering its asset/share ratio.
4. Deposit assets back into the vault -> receive **more** shares than originally borrowed.
5. Repay the flashloan, keeping the surplus shares as profit.

The attack is profitable as long as the gain exceeds entry/exit fees of the vault.

## Impact

Attacker can extracts value from other depositors of the affected ERC4626 vault at no cost beyond gas and any vault-level fees, effectively a risk-free arbitrage.

## Recommendation

Disallow flashloans for volatile ERC4626-type assets whose asset/share ratio can be manipulated intra-transaction.

# Issue M-5: Interest accrual stops when asset is marked non-borrowable, causing losses for lenders and unfair charges later [RESOLVED]

Source: https://github.com/sherlock-audit/2025-12-dhedge-dytm-dec-18th/issues/42

## Vulnerability Detail

During interest accrual, `Office` skips updating interest if the asset is not borrowable:

https://github.com/sherlock-audit/2025-12-dhedge-dytm-dec-18th/blob/main/DYTM/src/Office.sol#L741-L743

```
if (dt == 0 || !marketConfig.isBorrowableAsset(key.getAsset())) {
    return 0;
}
```

When an admin marks an asset as non-borrowable while there is outstanding debt (`reserveData.borrowed > 0`), two problems arise:

1. No interest accrues during the non-borrowable period. Borrowers have no incentive to repay, and lenders lose expected yield.
2. `lastUpdateTimestamp` is not updated when returning early. If the asset is later made borrowable again, the next accrual will calculate `dt` from the stale timestamp, retroactively charging interest for the entire non-borrowable period.

## Impact

1. Lenders lose interest during the non-borrowable window.
2. Borrowers are overcharged if the asset becomes borrowable again, as skipped time is back-calculated into interest.
3. If an asset is switched from borrowable to non-borrowable without first calling `accrueInterest`, all the pending interest will be lost.

## Recommendation

Continue accruing interest on existing debt even when `isBorrowableAsset()` returns false.

# Issue M-6: Borrowable dHedgePool tokens enable price manipulation exploit [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit/2025-12-dhedge-dytm-dec-18th/issues/43

This issue has been acknowledged by the team but won't be fixed at this time.

## Vulnerability Detail

dHedgePool tokens derive their price from `IdHEDGEPoolLogic(asset).tokenPrice()`, which relies on `pool.balanceOf()` of underlying assets. This makes the token price susceptible to manipulation via direct token donations.

If dHedgePool tokens are ever enabled as borrowable assets, an attacker can execute a leveraged price manipulation attack similar to the Cream Finance exploit:

1. User A supplies dHedgePool tokens as collateral.
2. User B borrows the same dHedgePool tokens.
3. User B transfers tokens to User A, who supplies them again.
4. Repeat steps 2-3 to build artificial leverage: User A accumulates massive collateral, User B accumulates equivalent debt.
5. Attacker donates tokens directly to the dHedgePool, inflating `tokenPrice()`.
6. User A's collateral value spikes, granting inflated borrowing power to drain other assets.
7. User B's position becomes deeply underwater and is abandoned.

The profit is extracted from the artificially inflated borrowing power on User A's position.

## Impact

If dHedgePool tokens are made borrowable, attackers can drain lending markets of other assets via price manipulation, causing total loss of lender funds.

## Recommendation

Enforce that dHedgePool tokens can **never** be set as borrowable.

# Issue M-7: Underflow during repayment [RESOLVED]

Source: https://github.com/sherlock-audit/2025-12-dhedge-dytm-dec-18th/issues/47

## Vulnerability Detail

Assume the last user decided to repay all their shares (aka max shares path). In this case, `shares` will be equal to the last user's debt share balance.

Then, when calculating the `assetsRepaid`, the math is rounded up, resulting in `assetsRepaid` being 1 wei more than expected. Subsequently, when executing `$reserveData.borrowed -= assetsRepaid`, it will cause a underflow revert due to a shortfall of 1 wei.

The following is a simple scenario to demonstrate the underflow:

1. Initial state: `$reserveData.borrowed` = 0, `totalDebtSupply` = 0
2. Bob borrowed 2 assets. So 2,000,000 debt shares are minted. Due to `VIRTUAL_SHARES` = 1e6 and `VIRTUAL_ASSETS = 1`

$$\text{mintedDebtShares} = \left\lceil \frac{2 \cdot (0 + 10^6)}{0 + 1} \right\rceil = 2,000,000$$

Current state: `$reserveData.borrowed` = 2, `totalDebtSupply` = 2,000,000

3. Bob or someone repay 1 debt shares, so `$reserveData.borrowed` reduced by 1, and 1 debt share is burned. Current state: `$reserveData.borrowed` = 1, `totalDebtSupply` = 1,999,999

$$\text{assetsRepaid} = \left\lceil \frac{1 \cdot (2 + 1)}{2,000,000 + 10^6} \right\rceil = \left\lceil \frac{3}{3,000,000} \right\rceil = 1$$

4. Bob try to repay all shares by using the max share feature (`params.shares == type(uint256).max`). The `assetsRepaid` will be 2, while the `$reserveData.borrowed` is only 1, so an underflow revert will occur.

$$\text{assetsRepaid} = \left\lceil \frac{1,999,999 \cdot (1 + 1)}{1,999,999 + 10^6} \right\rceil = \left\lceil \frac{3,999,998}{2,999,999} \right\rceil = \lceil 1.33333 \ldots \rceil = 2$$

## Impact

Underflow revert might occur under certain conditions, leading to repayment to revert.

## Code Snippet

https://github.com/sherlock-audit/2025-12-dhedge-dytm-dec-18th/blob/75a66499b7be9b8b9fb8518a798e52e4e0043f43/DYTM/src/Office.sol#L304

## Tool Used

Manual Review

## Recommendation

Consider flooring it to 0 on underflow.

# Issue M-8: Missing validation allows weight and liquidation bonus sum to exceed 100%, causing immediate bad debt on liquidation [RESOLVED]

Source: https://github.com/sherlock-audit/2025-12-dhedge-dytm-dec-18th/issues/48

## Vulnerability Detail

The `SimpleWeights` contract manages collateral weights (LLTV - Liquidation Loan-to-Value) for different collateral-debt pairs. The `setWeight()` function allows the owner to configure the maximum borrowing capacity as a percentage of collateral value:

https://github.com/sherlock-audit/2025-12-dhedge-dytm-dec-18th/blob/main/DYTM/src/extensions/weights/SimpleWeights.sol#L83-L87

```
function setWeight(uint256 collateralTokenId, ReserveKey debtAsset, uint64 weight)
↪   external onlyOwner {
    require(weight <= WAD, SimpleWeights__InvalidWeight());
    rawWeights[collateralTokenId][debtAsset] = weight;
}
```

The function only validates that `weight <= WAD` (100%), but does not validate the relationship between `weight` and `liquidationBonusPercentage`. When a position becomes unhealthy and is liquidated, the liquidator receives the collateral plus a bonus. If `weight + liquidationBonusPercentage > 100%`, liquidating a position that just became unhealthy will immediately result in bad debt.

Example scenario:

- Weight (LLTV) = 80%
- Collateral value = 100
- Debt = 80.001 (position just became unhealthy)
- Liquidation bonus = 20.001%

When liquidating, the liquidator claims 100% of collateral but the protocol must also pay 20.001% as bonus. The total payout exceeds the collateral value, creating instant bad debt.

## Impact

Positions that marginally exceed the liquidation threshold will generate bad debt upon liquidation, as the required liquidation bonus payout cannot be covered by the seized collateral. This leads to protocol insolvency and losses for lenders.

## Recommendation

Add validation in `setWeight()` to ensure `weight + liquidationBonusPercentage <= WAD - buffer`. It is also recommended to add a buffer.

# Issue M-9: Liquidation cannot be executed or delayed during a liquidity crunch [RESOLVED]

Source: https://github.com/sherlock-audit/2025-12-dhedge-dytm-dec-18th/issues/50

## Vulnerability Detail

Consider a market with USDC and WETH.

1. Bob supplies 1000 USDC as LEND collateral. Reserve (USDC) = 1000
2. Alice borrows all 1000 USDC from the USDC reserve. Reserve (USDC) = 0
3. Bob has an open debt position (e.g., borrowed WETH). Later, his position becomes unhealthy due to the rise in debt value or the fall in collateral value
4. A liquidator attempts to liquidate Bob using his USDC LEND collateral
5. Liquidation reverts inside `_withdraw()` because Reserve (USDC) = 0 (Insufficient liquidity)

So Bob cannot be liquidated until Alice repays or new liquidity is added, which introduces a delay in liquidation and increases the risk of bad debts.

## Impact

Liquidation cannot be executed or delayed during a liquidity crunch.

## Code Snippet

https://github.com/sherlock-audit/2025-12-dhedge-dytm-dec-18th/blob/75a66499b7be9b8b9fb8518a798e52e4e0043f43/DYTM/src/Office.sol#L380

## Tool Used

Manual Review

## Recommendation

Most lending protocols mitigate this issue by offering the liquidator the option to receive "shares" instead of underlying assets, so that liquidation can proceed during liquidity crunches or periods of high utilization. The liquidator will hold the shares and redeem them as and when liquidity becomes available.

Consider adding a feature that allows the liquidator to receive "shares".

# Issue M-10: Switching of collateral does not perform a health check [RESOLVED]

Source: https://github.com/sherlock-audit/2025-12-dhedge-dytm-dec-18th/issues/52

## Vulnerability Detail

The admin can set different weights for the same asset, but of a different type. For instance, the admin may deem a collateral that cannot be lent out a safer asset than one that can be lent out, as in the event of high utilization, the debt position with escrowed collateral can be liquidated more efficiently.

Assume the following scenario:

- minMarginAmountUSD = $100
- debtValueUSD = $90
- collateralValueUSD = $120
- weight(ESCROW collateral, debt asset) = 0.95
- weight(LEND collateral, debt asset) = 0.7

Before switching (Collateral = ESCROW), the `weightedCollateral` is 114 (120 * 0.95). Since 114 >= 90 and 114 >= 100, the account is healthy. After switching to LEND, the `weightedCollateral` is 84 (120 * 0.7). Since 84 >= 100, the account is unhealthy.

Another scenario that might occur if `_checkHealth()` does not exist is that users can front-run the liquidation TX against their account, switch collateral to a higher weight (ESCROW) to avoid liquidation, and back-run the liquidation TX to switch it back to the original lower weight (LEND), to cause the liquidation TX to revert. Users have an incentive to switch back to LEND rather than the ESCROWED type, as LEND collateral accrues interest, whereas the ESCROWED type does not.

## Impact

Accounts could become unhealthy after switching the collateral. Malicious users could also exploit this to avoid or delay liquidation, increasing the risk of bad-debt accumulation.

## Code Snippet

https://github.com/sherlock-audit/2025-12-dhedge-dytm-dec-18th/blob/75a66499b7be9b8b9fb8518a798e52e4e0043f43/DYTM/src/Office.sol#L131

## Tool Used

Manual Review

## Recommendation

Consider adding a `_checkHealth()` function towards the end of the `switchCollateral()` function to handle this edge case since the `_withdraw()` function itself does not perform any health check.

# Issue M-11: Missing interest accrual causes stale health checks on token transfers [RESOLVED]

Source: https://github.com/sherlock-audit/2025-12-dhedge-dytm-dec-18th/issues/59

## Summary

Transfers of collateral or debt tokens perform health checks on the involved accounts. However, the absence of interest accrual prior to these checks results in stale debt calculations, potentially allowing accounts to become undercollateralized immediately after the transfer.

## Vulnerability Detail

When accounts transfer collateral or debt tokens, a health check is triggered via `_update`:

```
function _update(AccountId from, AccountId to, uint256 tokenId, uint256 amount)
↪  internal override {
        // ...
        if (
            from != Constants.ACCOUNT_ID_ZERO && to != Constants.ACCOUNT_ID_ZERO
                && (tokenType != TokenType.ISOLATED_ACCOUNT && tokenType !=
                ↪  TokenType.NONE)
        ) {
            if (amount != 0) {
                // ...

                // ...
                if (tokenType != TokenType.DEBT) {
>>                  _checkHealth(from, market);
                } else {
                    // ...
                    _verifyCallerAuthorization(to);
>>                  _checkHealth(to, market);
                }
            }
        }
    }
```

The `_checkHealth` function calls `isHealthyAccount`, which calculates the account's debt in USD:

```
function isHealthyAccount(AccountId account, MarketId market) public view returns
↪  (bool isHealthy) {
        // ...
```

```
        // First we calculate the debt value in USD.
        uint256 debtValueUSD;
        {
            // Rounding up in favour of the lenders.
            uint256 debtAmount =
>>              balanceOf(account,
 ↪  debtId).toAssetsUp(getReserveDataStorage(debtKey).borrowed,
 ↪  totalSupply(debtId));

            debtValueUSD = oracleModule.getQuote({
                inAmount: debtAmount, base: address(debtKey.getAsset()), quote:
                 ↪  Constants.USD_ISO_ADDRESS
            });
        }

        // ...
    }
```

This calculation relies on `getReserveDataStorage(debtKey).borrowed`, which does not reflect pending interest if it hasn't been accrued recently. As a result, the computed `debtAmount` may be lower than the actual debt, allowing the health check to pass incorrectly.

If interest is accrued shortly after the transfer, the account may immediately become liquidatable. In extreme cases, this could lead to bad debt if the undercollateralization is significant.

## Impact

Accounts may appear healthy during token transfers due to stale debt calculations. This can result in immediate liquidations or bad debt once interest is accrued.

## Code Snippet

https://github.com/sherlock-audit/2025-12-dhedge-dytm-dec-18th/blob/main/DYTM/src/Office.sol#L811 https://github.com/sherlock-audit/2025-12-dhedge-dytm-dec-18th/blob/main/DYTM/src/Office.sol#L817

## Tool Used

Manual Review

## Recommendation

Accrue interest on the market before performing health checks during collateral or debt token transfers.

# Issue M-12: Unfair liquidations due to pending interest [RESOLVED]

Source: https://github.com/sherlock-audit/2025-12-dhedge-dytm-dec-18th/issues/60

## Summary

Health checks during liquidation understate collateral value when interest on collateral tokens has not been accrued. This allows accounts that are marginally healthy to be liquidated.

## Vulnerability Detail

Before a liquidation, the protocol verifies that the account is unhealthy using `isHealthyAccount`:

```
function liquidate(LiquidationParams calldata params) external returns (uint256
 ↪  assetsRepaid) {
        // ...

        // Check that the account is unhealthy before liquidating.
        require(
>>          !isHealthyAccount(params.account, params.market),
 ↪  Office__AccountStillHealthy(params.account, params.market)
        );
}
```

Within `isHealthyAccount`, the collateral value is calculated by converting collateral shares to assets using the total supplied value in the market:

```
assetAmount = balanceOf(account, collateralIds[i])
    .toAssetsDown(getReserveDataStorage(collateralKey).supplied,
    ↪  totalSupply(collateralIds[i]));
```

If interest has not recently been accrued for a given collateral token, the value of `getReserveDataStorage(collateralKey).supplied` will be lower than it should be. This causes `assetAmount` to be underestimated, potentially leading to a failed health check even if the account would be healthy after interest accrual.

While the protocol accrues interest for the debt token before liquidation, it does not do so for the collateral tokens held by the account, which causes this discrepancy.

## Impact

Accounts with sufficient collateral may be prematurely and unfairly liquidated due to stale interest accrual on collateral tokens. This undermines liquidation fairness and can lead to user fund losses.

## Code Snippet

https://github.com/sherlock-audit/2025-12-dhedge-dytm-dec-18th/blob/main/DYTM/src/Office.sol#L613

## Tool Used

Manual Review

## Recommendation

Accrue interest for all collateral tokens held by the liquidated account before executing the health check.

# Issue M-13: Withdrawing collateral or borrowing debt can be griefed [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit/2025-12-dhedge-dytm-dec-18th/issues/61

This issue has been acknowledged by the team but won't be fixed at this time.

## Summary

Withdrawals and borrows rely on available market liquidity. This creates an opportunity for griefing attacks, where an adversary can temporarily exhaust liquidity via a sandwich strategy, preventing legitimate users from completing their transactions.

## Vulnerability Detail

When a user attempts to withdraw or borrow tokens, the protocol checks if sufficient liquidity is available in the market:

```
// Revert if there isn't enough liquidity in the reserve.
require($reserveData.borrowed <= $reserveData.supplied,
↪  Office__InsufficientLiquidity(key));
```

This check can be exploited by an attacker who strategically borrows all available liquidity just before the victim's transaction is executed. An example attack flow:

1. Bob submits a transaction to withdraw all his USDC collateral.
2. Alice monitors the mempool and sees Bob's transaction.
3. Alice frontruns Bob by depositing collateral and borrowing all available USDC.
4. Bob's transaction fails due to insufficient liquidity.
5. Alice then repays the borrowed USDC in the same block, withdraws her collateral, and restores the original market state.

This results in no financial cost to the attacker (e.g., no interest is accrued within the block) except for the gas paid, while disrupting the victim's ability to access their funds.

## Impact

Attackers or MEV bots can block legitimate user transactions for withdrawals or borrows by temporarily draining market liquidity.

## Code Snippet

https://github.com/sherlock-audit/2025-12-dhedge-dytm-dec-18th/blob/main/DYTM/src/Office.sol#L713 https://github.com/sherlock-audit/2025-12-dhedge-dytm-dec-18th/blob/main/DYTM/src/Office.sol#L239

## Tool Used

Manual Review

## Recommendation

Consider introducing logic to prevent borrowing and repaying in the same block, or explicitly acknowledge this griefing vector as a known limitation.

# Issue M-14: Bad debt socialization can be skipped [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit/2025-12-dhedge-dytm-dec-18th/issues/62

This issue has been acknowledged by the team but won't be fixed at this time.

## Summary

A malicious liquidator can intentionally leave a minimal amount of collateral in an underwater account to prevent bad debt from being socialized, avoiding losses if they are also a lender in the affected market.

## Vulnerability Detail

During liquidation, if an account holds bad debt and no remaining collateral, the protocol initiates bad debt socialization:

```
    // If the position has accrued bad debt, we need to socialise the losses and
    ↪  remove the debt
    // from the account.
    debtSharesUnpaid = balanceOf(params.account, debtId);
    if (
>>      getRegistryStorageStruct().marketWiseData[params.account][params.market].co
↪   llateralIds.length() == 0
            && debtSharesUnpaid > 0
    ) {
        debtAssetsUnpaid = debtSharesUnpaid.toAssetsUp($reserveData.borrowed,
        ↪  totalSupply(debtId));

        $reserveData.borrowed =
        ↪  $reserveData.borrowed.saturatingSub(debtAssetsUnpaid);
        $reserveData.supplied =
        ↪  $reserveData.supplied.saturatingSub(debtAssetsUnpaid);

        _burn({from: params.account, tokenId: debtId, amount: debtSharesUnpaid});
    }
```

This logic only executes if the account has no remaining collateral in the market. A liquidator can exploit this by leaving a trivial amount (e.g., 1 wei) of collateral in the account, which skips the bad debt handling.

This is particularly problematic if the liquidator is also a lender in the market. In such a case, they can avoid incurring a proportional loss from socialized bad debt, exit their position, and then trigger socialization later, shifting the losses onto other lenders.

## Impact

Malicious liquidators can bypass bad debt socialization, leading to unfair distribution of losses.

## Code Snippet

https://github.com/sherlock-audit/2025-12-dhedge-dytm-dec-18th/blob/main/DYTM/src/Office.sol#L445

## Tool Used

Manual Review

## Recommendation

Acknowledge this limitation in the protocol design, as it's a design flaw inherited from the Morpho codebase.

# Issue M-15: Flashloan attack allows resetting dHEDGE pools share price [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit/2025-12-dhedge-dytm-dec-18th/issues/63

This issue has been acknowledged by the team but won't be fixed at this time.

## Summary

If the total supply of a dHEDGE pool is deposited into `Office`, an attacker can exploit flash loans to reset the pool's share price, leading to bad debt and mass liquidations.

## Vulnerability Detail

When the entire supply of dHEDGE pool shares is deposited into `Office`, the following attack becomes feasible:

1. The attacker acquires dHEDGE pool shares and uses them as collateral to borrow another asset from `Office`.
2. They take a flash loan of the entire pool share supply.
3. Using the flash loaned shares, they redeem all pool tokens from the dHEDGE vault, draining its underlying assets and resetting the share price.
4. They deposit a small amount of underlying assets to mint new shares at the now-reset share price (e.g., 1:1).
5. The flash loan is repaid using these newly minted shares.
6. The original collateral is now worth significantly less, while the borrowed amount remains unchanged, resulting in protocol bad debt.

Additionally, other positions using the affected collateral may become liquidatable, increasing the attacker's profit.

This exploit is only viable under the following conditions:

- The entire vault supply is held within `Office`.
- No streaming or manager fees exist on the pool, or these fees are automatically deposited into `Office` upon minting.
- No exit fees are enforced on withdrawals.
- The vault allows full withdrawal of all underlying assets in a single transaction.

## Impact

An attacker can reset the share price of a dHEDGE pool, causing widespread liquidations and generating bad debt in the protocol.

## Code Snippet

https://github.com/sherlock-audit/2025-12-dhedge-dytm-dec-18th/blob/main/DYTM/src/Office.sol#L552

## Tool Used

Manual Review

## Recommendation

Restrict flash loans for share-based tokens such as dHEDGE pools and ERC-4626 vaults to prevent this form of share price manipulation.

# Issue M-16: First borrower can prevent other users from borrowing by inflating the total shares [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit/2025-12-dhedge-dytm-dec-18th/issues/64

This issue has been acknowledged by the team but won't be fixed at this time.

## Summary

When a new borrowable token is enabled in a market, the first borrower can manipulate the total borrow shares accounting by repeatedly borrowing and repaying minimal amounts. This inflates the total borrow shares, ultimately preventing further borrowing from the market.

## Vulnerability Detail

Upon enabling a new borrowable token in a market, a malicious actor can execute a loop where they:

- Borrow 1 wei of the token.
- Repay 1 wei in shares, not assets.

The following PoC illustrates this attack and can be added to `Office_borrow.t.sol`:

```
function test_InflateBorrowShares()
    external
    whenReserveExists
    givenAccountIsIsolated
    whenCallerIsOperator
    givenEnoughLiquidityInTheReserve
    givenAccountIsHealthy
    whenUsingSuppliedPlusEscrowedAssets
{
    vm.startPrank(caller);

    assertEq(office.getReserveData(usdcKey).borrowed, 0);
    assertEq(office.totalSupply(usdcKey.toDebtId()), 0);

    for(uint i = 0; i < 190; i++) {
        // Borrow 1 wei of asset
        office.borrow(BorrowParams(account, usdcKey, caller, 1, ""));

        // Repay 1 wei of shares
        office.repay(RepayParams(account, usdcKey, TokenType(0), 0, 1, ""));
    }
```

```
    // We check the total borrow shares, which have been inflated a lot
    assertEq(office.getReserveData(usdcKey).borrowed, 0);
    assertEq(office.totalSupply(usdcKey.toDebtId()),
    ↪  1569273864571236344288756396854560802108984835527147512214371777);

    // Next normal borrow call will revert
    vm.expectRevert();
    office.borrow(BorrowParams(account, usdcKey, caller, 100e6, ""));
}
```

This attack requires no capital, only transaction fees, and renders the market unusable for future borrowers.

## Impact

An attacker can prevent others from borrowing a specific token by artificially inflating the total borrow shares, effectively disabling lending of that token in the market.

## Code Snippet

https://github.com/sherlock-audit/2025-12-dhedge-dytm-dec-18th/blob/main/DYTM/src/Office.sol#L301

## Tool Used

Manual Review

## Recommendation

Implement a minimum borrow amount to avoid share inflation in low-debt conditions, or document the issue and encourage market officers to use hooks to mitigate this behavior.

# Issue M-17: Small collateral balances block bad debt socialization during liquidation [RESOLVED]

Source: https://github.com/sherlock-audit/2025-12-dhedge-dytm-dec-18th/issues/65

## Summary

A strict check on withdrawals prevents seizing 0 collateral with a negligible share balance. This can block liquidation and, in turn, prevent bad debt socialization, even when the account is effectively underwater.

## Vulnerability Detail

The `Office` contract includes the following check to prevent burning non-zero shares for zero assets:

```
// It's possible that due to rounding errors, the `assetsWithdrawn` is zero even
↪  though `sharesRedeemed` is non-zero.
// In such a case, we don't want shares to be burned and instead revert.
// While `sharesRedeemed` cannot be zero (due to rounding errors), we still check
↪  for it to be safe.
require(assetsWithdrawn != 0 && sharesRedeemed != 0,
↪  Office__ZeroAssetsOrSharesWithdrawn(account, key));
```

This check is generally appropriate, but it creates an edge case during liquidation:

1. An account (e.g., Bob) is underwater with two collateral tokens: WETH and USDC.
2. Bob holds only 1 wei of USDC shares, which maps to 0 USDC assets due to the virtual share rate (e.g., `1e6 shares = 1 wei assets`).
3. When a liquidator attempts to seize both collaterals, the withdraw call for USDC fails because `assetsWithdrawn == 0`, causing the transaction to revert.
4. Even if the liquidation proceeds with WETH only, the remaining 1 wei of USDC collateral shares prevents triggering the bad debt socialization logic, which only executes when no collateral remains.

This behavior can also be induced preemptively by transferring 1 wei of a collateral token to an underwater account.

## Impact

Bad debt socialization is blocked if the liquidated account holds a negligible collateral balance that cannot be withdrawn due to the `assetsWithdrawn != 0` check.

## Code Snippet

https://github.com/sherlock-audit/2025-12-dhedge-dytm-dec-18th/blob/main/DYTM/src/Office.sol#L729

## Tool Used

Manual Review

## Recommendation

Skip the `assetsWithdrawn != 0` check when the withdrawal occurs in the context of a liquidation, allowing all collateral tokens to be fully cleared regardless of precision loss.

# Issue M-18: Incorrect borrow rate when donating to reserve [RESOLVED]

Source: https://github.com/sherlock-audit/2025-12-dhedge-dytm-dec-18th/issues/66

## Summary

The `donateToReserve` function does not accrue interest before increasing the `supplied` value. As a result, the next interest accrual uses an incorrect borrow rate based on the new supply, undercharging interest for the prior utilization period.

## Vulnerability Detail

The `donateToReserve` function allows an officer to increase the reserve supply by donating assets:

```
    function donateToReserve(ReserveKey key, uint256 assets) external
    ↪  onlyOfficer(key.getMarketId()) {
        OfficeStorage.ReserveData storage $reserveData = getReserveDataStorage(key);

        // It isn't worth donating to a reserve which has no assets supplied as all
        ↪  it does is
        // increase the initial exchange rate of the reserve.
        require($reserveData.supplied != 0, Office__ReserveIsEmpty(key));

>>      $reserveData.supplied += assets;

        // Transfer the donation from the caller to the Office contract.
        key.getAsset().safeTransferFrom({from: msg.sender, to: address(this),
        ↪  value: assets});

        emit Office__AssetDonated({key: key, assets: assets});
    }
```

The function modifies the `supplied` value without first accruing interest. This causes the subsequent call to `accrueInterest` to compute the borrow rate using the inflated supply value, which does not reflect the utilization over the elapsed time.

**Example scenario:**

- Initial state: 100 USDC supplied, 80 USDC borrowed → 80% utilization, borrow rate should be 10%.
- Time passes, no interest accrual occurs.
- Officer donates 60 USDC via `donateToReserve`, increasing `supplied` to 160.

- The next interest accrual computes utilization as `80/160 = 50%`, resulting in a 3% borrow rate applied retroactively to the entire period.

This inaccurately reduces the borrow interest charged, underpricing debt and resulting in a loss to lenders.

## Impact

The borrow rate used for interest accrual is lower than it should be due to calling `donateToReserve`, which distorts utilization and reduces interest charged on outstanding debt.

## Code Snippet

https://github.com/sherlock-audit/2025-12-dhedge-dytm-dec-18th/blob/main/DYTM/src/Office.sol#L536

## Tool Used

Manual Review

## Recommendation

Invoke `accrueInterest` within `donateToReserve` before modifying the `supplied` value to ensure borrow rates reflect accurate utilization over time.

# Issue M-19: MEV sandwich attack allows theft of reserve donations [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit/2025-12-dhedge-dytm-dec-18th/issues/67

This issue has been acknowledged by the team but won't be fixed at this time.

## Summary

When `donateToReserve` is called by the market officer, a MEV bot can sandwich the transaction to capture the majority of the donated assets.

## Vulnerability Detail

The `donateToReserve` function in the `Office` contract directly increases the supplied amount for the reserve, which immediately affects the exchange rate:

```
    function donateToReserve(ReserveKey key, uint256 assets) external
    ↪  onlyOfficer(key.getMarketId()) {
        OfficeStorage.ReserveData storage $reserveData = getReserveDataStorage(key);

        // It isn't worth donating to a reserve which has no assets supplied as all
        ↪  it does is
        // increase the initial exchange rate of the reserve.
        require($reserveData.supplied != 0, Office__ReserveIsEmpty(key));

>>      $reserveData.supplied += assets;

        // Transfer the donation from the caller to the Office contract.
        key.getAsset().safeTransferFrom({from: msg.sender, to: address(this),
        ↪  value: assets});

        emit Office__AssetDonated({key: key, assets: assets});
    }
```

However, a MEV bot can exploit the timing to extract most of the donation through a sandwich attack:

1. A market has 100,000 supplied USDC, and the officer plans to donate 10,000 USDC as an incentive to lenders.
    - supplied = 100,000 USDC
    - shares = 100,000 (1:1 exchange rate).
2. A MEV bot sees the pending tx in the mempool and frontruns by depositing 10M USDC into the market, minting 10M shares.

    - supplied = 10,100,000 USDC
    - shares = 10,100,000
3. Then, the officer's tx is executed, which donates 10,000 USDC to lenders.
    - supplied = 10,110,000 USDC
    - shares = 10,100,000
4. Just after, the MEV bot withdraws all its collateral shares.
    - assetsWithdrawn = `10,000,000 * 10,110,000 / 10,100,000` = ~10,009,900 USDC
    - supplied = 100,100 USDC
    - shares = 100,000

As a result, the MEV bot earns approximately 9,900 USDC, while legitimate lenders receive only 100 USDC of the intended 10,000 USDC donation, just 1% of the total.

## Impact

Any actor monitoring the mempool can sandwich `donateToReserve` transactions and capture the majority of the donated assets, undermining the intended benefit to lenders.

## Code Snippet

https://github.com/sherlock-audit/2025-12-dhedge-dytm-dec-18th/blob/main/DYTM/src/Office.sol#L536

## Tool Used

Manual Review

## Recommendation

Officers should be instructed to use MEV-protected RPC endpoints when calling `donateToReserve`.

Additionally, pausing markets via hooks before the donation could mitigate this issue, though this approach may also be subject to exploitation if an attacker anticipates the sequence.

# Issue L-1: Incorrect decimals returned for low-address assets in `_getDecimals` [RESOLVED]

Source: https://github.com/sherlock-audit/2025-12-dhedge-dytm-dec-18th/issues/35

## Summary

The `_getDecimals` function can return an incorrect decimal value for assets with addresses lower than `_ADDRESS_RESERVED_RANGE` when those assets have fewer than 18 decimals. This results in incorrect price calculations within the oracle.

## Vulnerability Detail

In `BaseOracleModule`, the `_getDecimals` function returns 18 decimals by default for any asset with an address lower than `_ADDRESS_RESERVED_RANGE` (`0xffffffff`):

```
function _getDecimals(address asset) internal view returns (uint8 decimals) {
    if (uint160(asset) <= _ADDRESS_RESERVED_RANGE) {
        return 18;
    }
    (bool success, bytes memory data) =
    ↪  asset.staticcall(abi.encodeCall(IERC20Metadata.decimals, ()));
    return success && data.length == 32 ? abi.decode(data, (uint8)) : 18;
}
```

This logic assumes that any asset with a low address has 18 decimals, but there is no guarantee that this is true. Although the likelihood of an ERC20 token being deployed at such an address is very low, certain system or base-token contracts might exist at these locations.

For instance, the native POL token resides at address `0x0000000000000000000000000000000000001010`, which falls below `_ADDRESS_RESERVED_RANGE`. While POL has 18 decimals and does not cause an issue, future tokens at similar addresses could have fewer decimals, resulting in incorrect pricing by the oracle.

## Impact

Assets with addresses below `_ADDRESS_RESERVED_RANGE` and fewer than 18 decimals will have their prices calculated incorrectly, leading to an underprice for these assets.

## Code Snippet

https://github.com/sherlock-audit/2025-12-dhedge-dytm-dec-18th/blob/main/DYTM/src/extensions/oracles/BaseOracleModule.sol#L31C1-L33C10

## Tool Used

Manual Review

## Recommendation

Refactor `_getDecimals` to attempt calling the `decimals` method first. Only fall back to the address check if that call fails.

# Issue L-2: Assets and allowances can be stolen from `SimpleDelegatee` [RESOLVED]

Source: https://github.com/sherlock-audit/2025-12-dhedge-dytm-dec-18th/issues/36

## Summary

The `SimpleDelegatee` contract allows arbitrary calls, enabling anyone to steal assets or exploit allowances left behind by users.

## Vulnerability Detail

`SimpleDelegatee` is designed to handle delegation callbacks from the `Office` contract. It exposes an `aggregate` function that performs arbitrary external calls:

```
function aggregate(Call[] memory calls) public returns (bytes[] memory returnData) {
        uint256 length = calls.length;
        returnData = new bytes[](length);
        Call memory call;
        for (uint256 i = 0; i < length;) {
            call = calls[i];
            (bool success, bytes memory data) = call.target.call(call.callData);

            require(success, SimpleDelegatee__CallFailed(call));

            returnData[i] = data;

            unchecked {
                ++i;
            }
        }
    }
```

The function lacks access control and allows any user to call it. If a user mistakenly leaves tokens in the contract or grants allowances to it, an attacker can craft arbitrary calls to transfer those assets or use the approvals.

Although the contract is not meant to retain assets or approvals, this design presents a critical risk for users who are unaware of its behavior or use it incorrectly.

## Impact

Assets held by the contract can be stolen. Any ERC20 allowances granted to the contract can be used maliciously.

## Code Snippet

https://github.com/sherlock-audit/2025-12-dhedge-dytm-dec-18th/blob/main/DYTM/src/extensions/delegatees/SimpleDelegatee.sol#L39

## Tool Used

Manual Review

## Recommendation

Document this risk clearly in `SimpleDelegatee`, and advise users to use `OwnableDelegatee` if they intend to store assets or grant allowances.

# Issue L-3: Incorrect `DYTMPeriphery::getExchangeRate()` unit scaling misreports lending/debt share price [RESOLVED]

Source: https://github.com/sherlock-audit/2025-12-dhedge-dytm-dec-18th/issues/44

## Vulnerability Detail

`DYTMPeriphery` exposes market/account information and derives exchangeRate by calling `getExchangeRate()`, which is intended to return "assets per 1 share" for a given `tokenId`.

However, `getExchangeRate()` uses `sharesToAssets(tokenId, 10 ** asset.decimals())` as the "1 share" unit:

https://github.com/sherlock-audit/2025-12-dhedge-dytm-dec-18th/blob/main/DYTM/src/periphery/DYTMPeriphery.sol#L302-L310

```
function getExchangeRate(uint256 tokenId) public returns (uint256 exchangeRate) {
    ReserveKey key = tokenId.getReserveKey();
    IERC20Metadata asset = IERC20Metadata(address(key.getAsset()));

    return sharesToAssets(tokenId, 10 ** (asset.decimals()));
}
```

For LENDING/DEBT token types, share accounting includes an additional 6-decimal offset, meaning the correct "1 share" base-unit is `10 ** (asset.decimals() + 6)`. As a result, the returned exchange rate is scaled incorrectly (while ESCROW-style tokens without the offset are unaffected).

## Impact

Mispriced exchange rate for lending/debt tokens (typically off by 10^6).

## Recommendation

Compute the "1 share" unit based on token type: use `10 ** (asset.decimals() + 6)` for lending/debt tokens, and `10 ** asset.decimals()` for escrow tokens.

# Issue L-4: Transient storage not cleared between delegation calls causes stale queue entries [RESOLVED]

Source: https://github.com/sherlock-audit/2025-12-dhedge-dytm-dec-18th/issues/45

## Vulnerability Detail

`TransientEnumerableHashTableStorage` uses transient storage to track `(account, market)` pairs in a queue during `delegationCall()` execution. The `__length` counter and hashtable entries persist in transient storage until the end of the transaction.

When a transaction contains multiple `delegationCall()` invocations, the second call inherits stale `(account, market)` pairs from the first call's queue. These leftover entries are then re-processed during health checks, even though they are unrelated to the current operation.

Since transient storage is only cleared at the end of a transaction (not between internal calls), subsequent `delegationCall()` operations within the same transaction will iterate over an increasingly polluted queue.

## Impact

1. Potential reverts: If stale entries reference accounts/markets in unexpected states, health checks may fail unexpectedly.

## Recommendation

Document that only one `delegationCall()` per transaction is supported.

# Issue L-5: Unable to disable hooks due to zero-address check in `setHooks()` [RESOLVED]

Source: https://github.com/sherlock-audit/2025-12-dhedge-dytm-dec-18th/issues/46

## Vulnerability Detail

`SimpleMarketConfig.setHooks()` allows the owner to update the hooks contract. However, it includes a zero-address check that prevents setting hooks to `address(0)`:

https://github.com/sherlock-audit/2025-12-dhedge-dytm-dec-18th/blob/main/DYTM/src/extensions/market-configs/SimpleMarketConfig.sol#L221-L229

```
function setHooks(IHooks newHooks) external onlyOwner {
    IHooks oldHooks = hooks;

@>  require(address(newHooks) != address(0), SimpleMarketConfig__ZeroAddress());

    hooks = newHooks;

    emit SimpleMarketConfig__HooksModified(newHooks, oldHooks);
}
```

Setting `hooks` to `address(0)` would be a natural and gas-efficient way to disable all hook functionality. With the current implementation, the owner must deploy and set a dummy no-op hooks contract instead.

## Recommendation

Remove the zero-address check in `setHooks()` to allow disabling hooks by setting to `address(0)`.

# Issue L-6: No use case for approving allowance for an IsolatedAccount [RESOLVED]

Source: https://github.com/sherlock-audit/2025-12-dhedge-dytm-dec-18th/issues/49

## Vulnerability Detail

There is no use case for approving allowance for an IsolatedAccount. Code can be simplified.

https://github.com/sherlock-audit/2025-12-dhedge-dytm-dec-18th/blob/main/DYTM/src/abstracts/Registry.sol#L182-L199

```
function approve(
    AccountId account,
    AccountId spender,
    uint256 tokenId,
    uint256 amount
)
```

## Recommendation

Change the `spender` to an `address` type.

# Issue L-7: Transfer of zero value might lead to liquidation to revert [RESOLVED]

Source: https://github.com/sherlock-audit/2025-12-dhedge-dytm-dec-18th/issues/53

## Vulnerability Detail

At the end of the liquidation function, the code will attempt to transfer debt assets from the liquidator to the `Office` contract. However, for some ERC20 tokens (e.g., the old AAVE token, LEND), it reverts when transferring 0 value.

The `debtAsset` can be zero when the users being liquidated hold collateral assets similar to the debt asset.

```
File: Office.sol
348:     function liquidate(LiquidationParams calldata params) external returns
 ↪  (uint256 assetsRepaid) {
..SNIP..
463:         // Transfer the debt asset from the liquidator to this contract thus
 ↪  reducing/repaying the debt.
464:         debtAsset.safeTransferFrom({from: msg.sender, to: address(this),
 ↪  value: liquidatorRepaymentObligation});
```

## Impact

Liquidation will be DOSed for certain tokens.

## Code Snippet

https://github.com/sherlock-audit/2025-12-dhedge-dytm-dec-18th/blob/75a66499b7be9b8b9fb8518a798e52e4e0043f43/DYTM/src/Office.sol#L464

## Tool Used

Manual Review

## Recommendation

Consider the following changes:

```
+ if (liquidatorRepaymentObligation > 0) {
        debtAsset.safeTransferFrom({from: msg.sender, to: address(this), value:
         ↪  liquidatorRepaymentObligation});
```

```
+ }
```

# Issue L-8: Lack of minimum debt size [RESOLVED]

Source: https://github.com/sherlock-audit/2025-12-dhedge-dytm-dec-18th/issues/54

## Vulnerability Detail

It was observed that the protocol enforces a minimum margin/collateral amount. However, many lending protocols enforce a minimum debt amount for opened positions rather than a minimum margin/collateral amount.

The reason is to prevent users from opening dust debt positions, since liquidating such positions can be unprofitable, and the liquidation bonus may not be sufficient to cover gas costs. In this case, there is no incentive for the liquidator to liquidate those positions, and bad debt may accumulate.

## Impact

Some dust positions might not be profitable to liquidate, increasing the risk of bad-debt accumulation.

## Code Snippet

https://github.com/sherlock-audit/2025-12-dhedge-dytm-dec-18th/blob/75a66499b7be9b8b9fb8518a798e52e4e0043f43/DYTM/src/Office.sol#L212

## Tool Used

Manual Review

## Recommendation

Consider implementing the minimum debt size.

# Issue L-9: Residual right and allowance when transferring an isolated account [RESOLVED]

Source: https://github.com/sherlock-audit/2025-12-dhedge-dytm-dec-18th/issues/56

## Vulnerability Detail

In the current design, if Alice sets Bob as an operator for her Isolated Account A but forgets to remove Bob from the operator list before transferring it to someone else, she will not be able to revoke Bob's operator access afterward since she is no longer the owner of the Isolated Account A.

If Alice later receives back her Isolated Account A (for example, if she had temporarily loaned it out), Bob will automatically regain operator privileges, which may or may not align with Alice's original intention.

```
File: Registry.sol
182:     function approve(
..SNIP..
191:     {
192:         address caller = msg.sender;
193:         address currentOwner = ownerOf(account);
194:
195:         require(currentOwner == caller, Registry__IsNotAccountOwner(account,
↪  caller));
196:
197:         _approve({currentOwner: currentOwner, from: account, spender: spender,
↪  tokenId: tokenId, amount: amount});
198:         success = true;
199:     }
```

In addition, the `approve()` function has the same issue. The approved allowance is also not reset when transferring an isolated account.

## Impact

Operator privileges might be automatically granted when an isolated account gets returned to the previous owner, which might not be intended. There will be a residual allowance when transferring an isolated account.

## Code Snippet

https://github.com/sherlock-audit/2025-12-dhedge-dytm-dec-18th/blob/75a66499b7be9b8b9fb8518a798e52e4e0043f43/DYTM/src/abstracts/Registry.sol#L74

https://github.com/sherlock-audit/2025-12-dhedge-dytm-dec-18th/blob/75a66499b7be9b8b9fb8518a798e52e4e0043f43/DYTM/src/abstracts/Registry.sol#L182

## Tool Used

Manual Review

## Recommendation

Consider implementing measures to address rights and allowances when transferring an isolated account.

# Issue L-10: Read-only reentrancy during delegation calls [RESOLVED]

Source: https://github.com/sherlock-audit/2025-12-dhedge-dytm-dec-18th/issues/68

## Vulnerability Detail

During a delegation call, health checks are deferred until the end of the call. This allows users to create temporary undercollateralized positions within the delegation context.

This behavior introduces a read-only reentrancy vector, where an attacker could exploit external integrations or hooks that rely on real-time collateral status. By having temporary undercollateralized positions, the attacker may trigger unintended behaviors in those systems, potentially leading to fund loss or incorrect state transitions.

## Impact

Read-only reentrancy can be used to manipulate integrations or hooks into performing actions based on temporarily undercollateralized accounts, potentially leading to loss of funds or unintended operations.

## Code Snippet

https://github.com/sherlock-audit/2025-12-dhedge-dytm-dec-18th/blob/main/DYTM/src/Office.sol#L82

## Tool Used

Manual Review

## Recommendation

This behavior should be clearly documented to ensure that integrations are aware that delegation calls can temporarily bypass collateral checks, and must not rely on them for real-time enforcement.

# Issue L-11: Wrong rounding direction during liquidations [RESOLVED]

Source: https://github.com/sherlock-audit/2025-12-dhedge-dytm-dec-18th/issues/69

## Vulnerability Detail

During liquidations, the protocol rounds up the amount of debt shares repaid:

```
debtSharesRepaid = assetsRepaid.toSharesUp($reserveData.borrowed,
↪  totalSupply(debtId));
```

This contradicts the standard best practice of rounding in favor of the protocol, which typically involves rounding down in such contexts.

## Impact

Incorrect rounding direction during liquidations may introduce small but exploitable discrepancies that can serve as a basis for more complex attack vectors.

## Code Snippet

https://github.com/sherlock-audit/2025-12-dhedge-dytm-dec-18th/blob/main/DYTM/src/Office.sol#L436

## Tool Used

Manual Review

## Recommendation

Change the rounding direction to round down when calculating debtSharesRepaid, in order to favor the protocol.

# Issue L-12: Isolated accounts can bypass transfer restrictions [RESOLVED]

Source: https://github.com/sherlock-audit/2025-12-dhedge-dytm-dec-18th/issues/70

## Vulnerability Detail

When transferring isolated accounts, the `canTransferShares` check is explicitly skipped:

```
    function _update(AccountId from, AccountId to, uint256 tokenId, uint256 amount)
    ↪  internal override {
        // ...
        if (
            from != Constants.ACCOUNT_ID_ZERO && to != Constants.ACCOUNT_ID_ZERO
>>              && (tokenType != TokenType.ISOLATED_ACCOUNT && tokenType !=
↪   TokenType.NONE)
        ) {
            if (amount != 0) {
                MarketId market = tokenId.getMarketId();

                // Check if the transfer is allowed as per market config.
                require(
                    getMarketConfig(market).canTransferShares({from: from, to: to,
                    ↪  tokenId: tokenId, shares: amount}),
                    Office__TransferNotAllowed({from: from, to: to, tokenId:
                    ↪  tokenId, shares: amount})
                );

                // ...
            }
        }
    }
```

This logic allows transfers of isolated accounts without enforcing transfer restrictions defined by `canTransferShares`.

As a result, a user with transfer privileges (e.g., whitelisted) can create an isolated account containing restricted assets and transfer ownership of that account to a non-whitelisted address, effectively bypassing the restrictions.

## Impact

Non-whitelisted addresses can gain control of accounts containing assets under transfer restrictions, circumventing intended access controls.

## Code Snippet

https://github.com/sherlock-audit/2025-12-dhedge-dytm-dec-18th/blob/main/DYTM/src/Office.sol#L796

## Tool Used

Manual Review

## Recommendation

Update the transfer logic or associated hooks to enforce transfer restrictions based on the owner of the isolated account.

Also, document this issue so that all officers are aware of it and can implement hooks to prevent this bypass.

# Disclaimers

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

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