SHERLOCK

Security Review For dHEDGE

Private Audit Contest Prepared For:	**dHEDGE**
Lead Security Expert:	**xiaoming90**
Date Audited:	**August 4 - August 13, 2025**
Final Commit:	**417230f**

Introduction

Explore a universe of top-tier tokenized vaults. Contest focuses on core contracts, updated Aave integration, new Pendle integration and a couple of periphery contracts designed to improve UX.

Scope

Repository: dhedge/V2-Public

Audited Commit: 5d41cbbc75758c474d835d59bdbd29363ab808e6

Final Commit: 417230fc4cd97e820f6b86f743f4bd4ffe53d3a5

Files:

- contracts/guards/assetGuards/AaveLendingPoolAssetGuard.sol
- contracts/guards/assetGuards/pendle/PendlePTAssetGuard.sol
- contracts/guards/contractGuards/AaveLendingPoolGuardV3.sol
- contracts/guards/contractGuards/pendle/PendleRouterV4ContractGuard.sol
- contracts/limitOrders/PoolLimitOrderManager.sol
- contracts/PoolFactory.sol
- contracts/PoolLogic.sol
- contracts/PoolManagerLogic.sol
- contracts/priceAggregators/ERC4626PriceAggregator.sol
- contracts/priceAggregators/PendlePTPriceAggregator.sol
- contracts/swappers/easySwapperV2/EasySwapperV2.sol
- contracts/swappers/easySwapperV2/WithdrawalVault.sol
- contracts/utils/pendle/PendlePTHandlerLib.sol

Final Commit Hash

417230fc4cd97e820f6b86f743f4bd4ffe53d3a5

Findings

Each issue has an assigned severity:

- 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.

- High issues are directly exploitable security vulnerabilities that need to be fixed.

Issues Found

High	Medium
3	6

Issues Not Fixed and Not Acknowledged

High	Medium
0	0

Security experts who found valid issues

000000
0x37
0xc0ffEE
Bigsam
j3x
ke1caM
newspacexyz
silver_eth
xiaoming90

Issue H-1: Tokens can be stolen

Source: https://github.com/sherlock-audit/2025-07-dhedge-update-judging/issues/30

Found by

000000

Summary

Tokens can be stolen by a manager which is untrusted. Issue is different than `All tokens can be stolen`, different root cause, not duplicates!

Root Cause

Missing YT validation in `redeemPyToToken()` path.

Internal Pre-conditions

-

External Pre-conditions

-

Attack Path

1. Manager calls `PoolLogic::execTransaction()` targeting `redeemPyToToken()` path in the Pendle contract guard.
2. His data decodes into `(poolLogicAddress, maliciousYTContract, ptToStealBalance, TokenOutput(supportedTokenOut, 0, supportedTokenOut, NONE_SWAP_DATA)` (some pseudo code here, you get the point).
3. First checks in `_validateSellPendlePT()` check receiver is pool logic address (pass), `tokenOut` field is supported (pass), swap data type is NONE (pass).
4. Then we call `isExpired()` on our fake YT and we expect true, our malicious contract handles that.
5. We call `PT()` on our fake YT. In the fake YT contract, this is a function which returns a fake PT address if `msg.sender` is the contract guard and the real PT to steal in the else case. Due to that, this call returns a fake PT address. Note that this fake PT has to be some token we do not have in our balance, can be our own contract or just

any token we don't have in the pool logic contract, so we return 0 on the `balanceOf()` call.

6. `intermediateSwapData` -> `(fakePT, supportedTokenOut, 0, X)`. We have `X` tokens of the supported token out, which can be 0, can be 1 million, whatever.
7. We call `redeemPyToToken()` on Pendle.
8. There, we call `SY` on our fake YT contract, return some malicious contract we own, can be our fake YT contract itself.
9. Call internal `_redeemPyToSy(fakeSY, fakeYT, ptToStealBalance, 1)`. There, first we call `PT()` on our fake YT, as sender is not the contract guard, we return the real PT to steal. Then, transfer the PT from the pool contract to our fake YT contract. Then, `needToBurnYt` is false as `isExpired()` returns true on our YT. Then, we call `redeemPY()` on our fake YT, we return some value above 0, let's say 1e18, to pass the slippage check in the internal function and we do a no-op.
10. `_redeemSyToToken(poolLogicAddress, fakeSY, 1e18, TokenOutput, false)` is called. Swap data type is `NONE`, so we go in `__redeemSy()`. There, `doPull` is false (last input), so we immediately call `redeem()` on our fake SY, we do a no-op and return some number. Back in `_redeemSyToToken()` we check the return against the slippage in `TokenOutput` we provided, which was 0, so it passes.
11. Pendle flow is over. End result is that the PT was transferred to the malicious contract, it is stolen. No funds were transferred to the pool logic contract.
12. `afterTxGuard()` is called on the Pendle contract guard which checks slippage. As source token is the fake PT, no funds were transferred from it, so `srcAmount` is 0. Destination token is the supported token, we didn't receive any, so `dstAmount` is also 0. Since both have the same value of 0$, slippage does not trigger as system thinks we didn't send out any funds.

Impact

Direct theft of funds.

PoC

No response

Mitigation

Validate the `YT` to be legit.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/dhedge/V2-Public/pull/12

Issue H-2: Lack of validation allowing malicious manager to bypass slippage control to steal funds

Source: https://github.com/sherlock-audit/2025-07-dhedge-update-judging/issues/118

Found by

000000, xiaoming90

Summary

-

Root Cause

-

Internal Pre-conditions

-

External Pre-conditions

-

Attack Path

It was observed that there is a lack of validation against the `_data` pass in by the manager in the `PendleRouterV4ContractGuard.txGuard` function.

The `_validateSellPendlePT()` function is the core function for validating the manager's input `_data`. Reviewed `_validateSellPendlePT` function shows that it only checks three (3) items, which is insufficient.

1. _receiver == _poolLogic
2. _output.tokenOut is supported assets
3. Swap type is NONE

https://github.com/sherlock-audit/2025-07-dhedge-update/blob/main/V2-Public/contracts/guards/contractGuards/pendle/PendleRouterV4ContractGuard.sol#L153

```
File: PendleRouterV4ContractGuard.sol
153:   function _validateSellPendlePT(
```

```
154:     address _poolLogic,
155:     address _poolManagerLogic,
156:     address _receiver,
157:     IPAllActionTypeV3.TokenOutput memory _output
158:   ) internal view {
159:     require(_receiver == _poolLogic, "recipient is not pool");
160:
161:     require(IHasSupportedAsset(_poolManagerLogic).isSupportedAsset(_output.tok
↪  enOut), "unsupported destination asset");
162:
163:     // Forbid swaps for initial version, this can be changed later
164:     require(_output.swapData.swapType == IPAllActionTypeV3.SwapType.NONE,
↪  "only underlying");
165:   }
```

The three (3) checks are not sufficient because it does not check other data such as `market` or `yt`. Thus, a malicious manager can pass in a malicious `market` address (deployed by the attacker). When Line 91 below `IPMarket(market).readTokens()` is executed, it will return an ERC20 token address that is supported by dHedge, but not the Pendle PT tokens that are going to be swapped, utilized, or transferred out during the Pendle operation.

https://github.com/sherlock-audit/2025-07-dhedge-update/blob/main/V2-Public/contracts/guards/contractGuards/pendle/PendleRouterV4ContractGuard.sol#L91

```
File: PendleRouterV4ContractGuard.sol
075:     } else if (method == IPActionSwapPTV3.swapExactPtForToken.selector) {
076:       (
077:         address receiver,
078:         address market,
079:         ,
080:         IPAllActionTypeV3.TokenOutput memory output,
081:         IPAllActionTypeV3.LimitOrderData memory limit
082:       ) = abi.decode(
083:           getParams(_data),
084:           (address, address, uint256, IPAllActionTypeV3.TokenOutput,
↪  IPAllActionTypeV3.LimitOrderData)
085:         );
086:
087:       _validateSellPendlePT(poolLogic, _poolManagerLogic, receiver, output);
088:
089:       _validateLimitOrder(limit);
090:
091:       (, address pt, ) = IPMarket(market).readTokens();
092:
093:       // `tokenOut` the the token to receive, no matter what the swap type is
094:       intermediateSwapData = SlippageAccumulator.SwapData({
095:         srcAsset: pt,
096:         dstAsset: output.tokenOut,
```

```
097:         srcAmount: _getBalance(pt, poolLogic),
098:         dstAmount: _getBalance(output.tokenOut, poolLogic)
099:       });
100:
101:       txType = uint16(TransactionType.SellPendlePT);
```

When the code reaches the `SlippageAccumulator` part, it will evaluate to the following. Refer to the audit comments below.

`SlippageAccumulator.SwapData.srcAmount` will be zero here. Since the pool does not hold any WETH token (or other ERC20 token apart from Pendle PT token), the `_getBalance` can be tricked to return zero.

```
intermediateSwapData = SlippageAccumulator.SwapData({
  srcAsset: pt, // @audit Non-PT token supported by dHedge (e.g., WETH)
  dstAsset: output.tokenOut,
  srcAmount: _getBalance(pt, poolLogic), // @audit return 0 when `balanceOf()` is
  ↪  called.
  dstAmount: _getBalance(output.tokenOut, poolLogic)
});
```

After each Pendle operation, the slippage check will be executed by the `SlippageAccumulatorUser.afterTxGuard()` function.

In Line 42, when the `_getBalance(intermediateSwapData.srcAsset, poolLogic)` is executed, it will return zero. As such, `swapData.srcAsset` will be zero, which is an important point to note.

```
srcAmount:
↪  intermediateSwapData.srcAmount.sub(_getBalance(intermediateSwapData.srcAsset,
↪  poolLogic))
srcAmount: (0.sub(0))
srcAmount: 0
```

https://github.com/sherlock-audit/2025-07-dhedge-update/blob/main/V2-Public/contracts/utils/SlippageAccumulatorUser.sol#L42

```
File: SlippageAccumulatorUser.sol
32:   function afterTxGuard(address poolManagerLogic, address to, bytes memory /*
↪  data */) public virtual override {
33:     address poolLogic = IPoolManagerLogic(poolManagerLogic).poolLogic();
34:     require(msg.sender == poolLogic, "not pool logic");
35:
36:     slippageAccumulator.updateSlippageImpact(
37:       poolManagerLogic,
38:       to,
39:       SlippageAccumulator.SwapData({
40:         srcAsset: intermediateSwapData.srcAsset,
41:         dstAsset: intermediateSwapData.dstAsset,
```

```
42:         srcAmount:
↪  intermediateSwapData.srcAmount.sub(_getBalance(intermediateSwapData.srcAsset,
↪  poolLogic)),
43:         dstAmount: _getBalance(intermediateSwapData.dstAsset,
↪  poolLogic).sub(intermediateSwapData.dstAmount)
44:       })
45:     );
46:     intermediateSwapData = SlippageAccumulator.SwapData(address(0), address(0),
↪  0, 0);
47:   }
```

The following is the `updateSlippageImpact()` function where the actual slippage check is performed.

In Line 110, the check will only be executed if the condition `dstValue < srcValue` is true. Since `swapData.srcAsset` is zero, the `srcValue` in Line 106 will always be zero. The `assetValue()` will not revert because, as mentioned earlier, the `swapData.srcAsset` will be configured to an ERC20 token that is supported by dHedge. Thus, an oracle has been configured, and price calculation will proceed with revert.

In this case, the condition `dstValue < srcValue` will always be `false`. Thus, the malicious manager (also attacker here) can bypass the slippage control, and perform a sandwich/MEV attack against the pendle's operation to steal funds.

https://github.com/sherlock-audit/2025-07-dhedge-update/blob/main/V2-Public/contracts/utils/SlippageAccumulator.sol#L100

```
File: SlippageAccumulator.sol
100:   function updateSlippageImpact(
101:     address poolManagerLogic,
102:     address router,
103:     SwapData calldata swapData
104:   ) external onlyContractGuard(router) {
105:     if
↪  (IHasSupportedAsset(poolManagerLogic).isSupportedAsset(swapData.srcAsset)) {
106:       uint256 srcValue = assetValue(swapData.srcAsset, swapData.srcAmount);
107:       uint256 dstValue = assetValue(swapData.dstAsset, swapData.dstAmount);
108:
109:       // Only update the cumulative slippage in case the amount received is
↪  lesser than amount sent/traded.
110:       if (dstValue < srcValue) {
111:         uint128 newSlippage =
↪  srcValue.sub(dstValue).mul(SCALING_FACTOR).div(srcValue).toUint128();
112:
113:         uint128 newCumulativeSlippage =
↪  (uint256(newSlippage).add(getCumulativeSlippageImpact(poolManagerLogic)))
114:           .toUint128();
115:
116:         require(newCumulativeSlippage < maxCumulativeSlippage, "slippage
↪  impact exceeded");
```

```
117:
118:         // Update the last traded timestamp.
119:         managerData[poolManagerLogic].lastTradeTimestamp =
↪  (block.timestamp).toUint64();
120:
121:         // Update the accumulated slippage impact for the poolManager.
122:         managerData[poolManagerLogic].accumulatedSlippage =
↪  newCumulativeSlippage;
123:       }
124:     }
125:   }
```

In the above example, I have shown an example using the code logic from the `if (method == IPActionSwapPTV3.swapExactPtForToken.selector)` code block in the `PendleRouterV4ContractGuard.txGuard` function. However, this issue is not just limited to `swapExactPtForToken.selector` code block.

The following code block or Pendle operation is also vulnerable to a similar exploit described here, with a slight modification, because the root cause is the same: a lack of validation against the arbitrary `_data` passed in by the manager.

1. IPActionMiscV3.exitPostExpToToken.selector (`market` is not validated and can be spoofed)
2. IPActionMiscV3.redeemPyToToken.selector (`yt` is not validated and can be spoofed)
3. IPActionSwapPTV3.swapExactTokenForPt.selector

Impact

A malicious manager can steal the funds in a pool/vault.

PoC

No response

Mitigation

No response

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/dhedge/V2-Public/pull/12

Issue H-3: Manager Can Steal User Funds Using `setUserEMode`

Source: https://github.com/sherlock-audit/2025-07-dhedge-update-judging/issues/123

Found by

000000, ke1caM, silver_eth, xiaoming90

Summary

The protocol performs `Health Factor (HF)` checks for AAVE operations that can impact a vaults's HF. However, the `setUserEMode` operation also affects a position's HF, but it is **not checked** in `AaveLendingPoolGuardV3.sol`. This omission allows a manager to manipulate the `HF` and potentially steal user funds via `liquidations`.

Root Cause

In `AaveLendingPoolGuardV3.sol`:

https://github.com/sherlock-audit/2025-07-dhedge-update/blob/main/V2-Public/contracts/guards/contractGuards/AaveLendingPoolGuardV3.sol#L90-L96

```
function _canAffectHealthFactor(bytes4 method) internal pure returns (bool
↪  canAffect) {
  if (
    method == IAaveV3Pool.borrow.selector ||
    method == IAaveV3Pool.setUserUseReserveAsCollateral.selector ||
    method == IAaveV3Pool.withdraw.selector
  ) canAffect = true;
}
```

`afterTxGuard` checks whether an operation can affect the HF. `_canAffectHealthFactor` currently only flags `borrow`, `setUserUseReserveAsCollateral`, and `withdraw` as HF-affecting operations.

There is no check for `setUserEMode`, which can reduce a position's HF to the minimum threshold (1e18) as allowed by AAVE:

`executeSetUserEMode`

https://github.com/aave-dao/aave-v3-origin/blob/6138e1fda45884b6547d094a1ddeef43dcab4977/src/contracts/protocol/libraries/logic/EModeLogic.sol#L27-L53

`validateHealthFactor`

https://github.com/aave-dao/aave-v3-origin/blob/6138e1fda45884b6547d094a1ddeef43dcab4977/src/contracts/protocol/libraries/logic/ValidationLogic.sol#L367-L395

```
uint256 public constant HEALTH_FACTOR_LIQUIDATION_THRESHOLD = 1e18;
```

```
require(
      healthFactor >= HEALTH_FACTOR_LIQUIDATION_THRESHOLD,
      Errors.HealthFactorLowerThanLiquidationThreshold()
    );
```

Internal pre-conditions

.

External pre-conditions

.

Attack Path

1. Manager creates an AAVE position and enables E-Mode using `setUserEMode`.
2. Manager supplies collateral and borrows an exact amount that sets up for E-Mode removal.
3. Manager disables E-Mode using `setUserEMode`, reducing the position's HF to `1e18`.
4. In the next block, the manager can `liquidate` the position, knowing it is eligible for `liquidation`.
5. Manager `liquidates` the position and collects the `liquidation bonus`.
6. Manager steals assets from the Vault, compromising protocol security and causing loss of user funds.

Impact

A manager can steal user funds by exploiting `AAVE E-Mode` and liquidations. Users lose more than 1% and more than $10 of their principal.

PoC

1. Setup Collateral: $200,000 in ETH E-Mode: LTV 93%, Liquidation Threshold 95%, Penalty 1% Normal: LTV 80.5%, Liquidation Threshold 83%, Penalty 5%

2. Debt for HF = 1 without E-Mode 1.00 = ($200,000 × 0.83) / Debt Debt = $166,000

3. E-Mode Maximum borrow in E-Mode = $200,000 × 0.93 = $186,000 Required borrow = $166,000

4. HF in E-Mode and after E-Mode removal In E-Mode: ($200,000 × 0.95) / $166,000 = 1.145 After E-Mode removal: ($200,000 × 0.83) / $166,000 = 1

5. After E-Mode removal `Health Factor` is equal to `1e18` and in the next block due to interest accural the position can be liquidated

Debt to be liquidated: $166,000 × 50% = $83,000 Collateral seized: $83,000 (debt) + ($83,000 × 5% penalty) = $87,150 Liquidation bonus to liquidator: $4,150 (5% of $83,000)

6. After liquidation Remaining collateral: $200,000 - $87,150 = $112,850 Remaining debt: $166,000 - $83,000 = $83,000 New Health Factor: ($112,850 × 0.83) / $83,000 = 1.127

Total Direct Loss from Liquidation Bonus: $4,150

Mitigation

Add `setUserEMode` to operations that affect `Health Factor`:

```
  function _canAffectHealthFactor(bytes4 method) internal pure returns (bool
  ↪  canAffect) {
    if (
      method == IAaveV3Pool.borrow.selector ||
      method == IAaveV3Pool.setUserUseReserveAsCollateral.selector ||
      method == IAaveV3Pool.withdraw.selector ||
++    method == IAaveV3Pool.setUserEMode.selector
    ) canAffect = true;
  }
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/dhedge/V2-Public/pull/9

Issue M-1: Malicious actors can DoS assets unrolling

Source: https://github.com/sherlock-audit/2025-07-dhedge-update-judging/issues/33

Found by

000000

Summary

Malicious actors can DoS assets unrolling

Root Cause

When unrolling assets in `WithdrawalVault::_unrollAssets()` (which happens when a user inits a withdrawal through `EasySwapperV2`), the following logic is executed:

```
for (uint256 i; i < supportedAssets.length; ++i) {
     address asset = supportedAssets[i].asset;
     uint16 assetType = IHasAssetInfo(poolFactory).getAssetType(asset);

     // Unrolling logic based on asset type...
}
```

For Uniswap positions, the following is executed:

```
else if (assetType == 7) {
     unrolledAssets = EasySwapperV3Helpers.getUnsupportedV3Assets(_dHedgeVault,
     ↪  asset);
}
```

Which runs the following:

```
function getUnsupportedV3Assets(address pool, address nonfungiblePositionManager)
↪  internal view returns (address[] memory assets) {
    uint256 nftCount =
    ↪  INonfungiblePositionManager(nonfungiblePositionManager).balanceOf(pool);
    // Each position has two assets
    assets = new address[](nftCount * 2);
    for (uint256 i = 0; i < nftCount; ++i) {
        uint256 tokenId = INonfungiblePositionManager(nonfungiblePositionManager).t
        ↪  okenOfOwnerByIndex(pool, i);
        (, , address token0, address token1, , , , , , , , ) = INonfungiblePosition
        ↪  Manager(nonfungiblePositionManager).positions(tokenId);

        assets[i * 2] = token0;
```

```
            assets[i * 2 + 1] = token1;
        }
    }
```

This allows an attacker to create unbounded iteration, causing DoS and loss of funds due to high gas expense.

Internal Pre-conditions

-

External Pre-conditions

-

Attack Path

1. Bob will init a withdrawal.
2. Alice interacts with the NFT manager by minting a ton of NFT positions to the pool, minting is a permissionless process.
3. The iteration will go OOG, causing loss of funds due to the gas and DoS.

Another one:

1. Bob will init a withdrawal.
2. Alice interacts with the NFT manager and mints him a position with malicious tokens which are then added to the `srcAssets` set.
3. When iterated over in `recoverAssets()`, they maliciously revert on transfers calls, causing DoS.

Impact

Loss of funds due to gas and DoS.

PoC

No response

Mitigation

Handle storage for UniV3 positions internally and do not rely on the NFT manager storage.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/dhedge/V2-Public/pull/14

Issue M-2: Withdrawals can fail due to approval overwrite

Source: https://github.com/sherlock-audit/2025-07-dhedge-update-judging/issues/60

Found by

000000

Summary

Withdrawals can fail due to approval overwrite

Root Cause

When withdrawing from Aave, there is a chance the collateral we are withdrawing from Aave is a PT. If withdraw data has been provided, we exit from the PT to the underlying token of the PT, i.e. for `PT-USDE`, it would be `USDE`. Then, we swap all of our collaterals to the debt token to repay our flashloan. The issue is that since both the PT and its underlying token are supported assets in the system and can be added as collateral to Aave, then here:

```
for (uint256 i; i < executionData.srcTokensLength; ++i) {
    transactions[executionData.txCount].to =
    ↪  address(swapProps.srcData[0].srcTokenSwapDetails[i].token);
    transactions[executionData.txCount].txData =
    ↪  abi.encodeWithSelector(IERC20Extended.approve.selector, swapper,
    ↪  swapProps.srcData[0].srcTokenSwapDetails[i].amount);
    executionData.txCount++;
}
```

We would approve USDE twice, one for the `PT-USDE` underlying (as we exit from the PT to USDE), and one from any USDE that has been provided as collateral to Aave. The first approval would be overwritten by the second one. Then, here:

```
transactions[executionData.txCount].to = swapper;
transactions[executionData.txCount].txData =
↪  abi.encodeWithSelector(ISwapper.swap.selector, swapProps);
```

When the actual swap takes place, the swapper wouldn't have enough approval to support the swap we are requesting, causing a revert.

Internal Pre-conditions

1. Both the `PT-Underlying` and `Underlying` are added as collateral to Aave. This is completely expected as both assets are specifically whitelisted (`USDE` and its PT and `SUSDE` and its PT for example), can be verified through the information in the contest README about whitelisted tokens. **Also note** that managers are not trusted, so they do not have to conform to some special trusted rules.

External Pre-conditions

-

Attack Path

No specific attack path, root cause mentions the necessary information.

Impact

DoS of proper withdrawals, break of core protocol functionality, Medium.

PoC

No response

Mitigation

Increase the approval instead of overwriting it, just be careful with USDT on mainnet.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/dhedge/V2-Public/pull/11

Issue M-3: Limit Order Slippage Validation Vulnerability in EasySwapperV2 Causes Fund Loss For Users

Source: https://github.com/sherlock-audit/2025-07-dhedge-update-judging/issues/69

Found by

newspacexyz

Summary

A vulnerability exists in the `completeLimitOrderWithdrawal` function of `EasySwapperV2.sol` where the slippage validation logic incorrectly validates the `_expectedDestTokenAmount` parameter. The function validates the total destination token balance in the WithdrawalVault after swaps, rather than the actual swap output, creating a vulnerability where front-running attacks by keepers can cause users to lose tokens through excessive slippage while still passing the validation check.

Root Cause

The vulnerability is located in the `completeLimitOrderWithdrawal` function at lines 321-326 in EasySwapperV2.sol:

```
function completeLimitOrderWithdrawal(
  IWithdrawalVault.MultiInSingleOutData calldata _swapData,
  uint256 _expectedDestTokenAmount
) external returns (uint256 destTokenAmount) {
  return _completeWithdrawal(msg.sender, _swapData, _expectedDestTokenAmount,
  ↪  WithdrawalVaultType.LIMIT_ORDER);
}
```

The problem is that `_expectedDestTokenAmount` represents the **total** destination token balance in the WithdrawalVault after swaps, not the **minimum output from the current swap operation**. This creates a vulnerability where:

1. A user creates multiple limit orders for different pools
2. Some limit orders are executed by keepers, depositing destination tokens into the WithdrawalVault
3. When the user calls `completeLimitOrderWithdrawal`, the validation checks the total balance against `_expectedDestTokenAmount`
4. If a keeper front-runs the user's transaction by executing another limit order, the total balance may still exceed `_expectedDestTokenAmount` even if the current swap suffers significant slippage

Internal Pre-conditions

1. A user must have created multiple limit orders for different pools
2. At least one limit order must have been executed by a keeper

External Pre-conditions

.

Attack Path

- User creates limit orders for PoolA and PoolB
- Limit order for PoolA is executed by a keeper, depositing 1000 USDC worth of tokens into the user's WithdrawalVault
- User calls `completeLimitOrderWithdrawal` with `_expectedDestTokenAmount = 1000` (expecting at least 1000 USDC total)
- A keeper executes the limit order for PoolB, which deposits 500 USDC into the user's WithdrawalVault, that are unrolled to USDC
- The user's swap operation suffers significant slippage, e.g., 50% slippage
- The swap only produces 500 USDC instead of the expected 1000 USDC
- However, the total vault balance is now 500 + 500 = 1000 USDC
- The validation `balanceAfterSwaps >= _expectedDestTokenAmount` passes (1000 >= 1000)
- The user receives 1000 USDC but has lost 500 USDC due to slippage

Impact

Users can lose tokens because of the incorrect slippage protection mechanism.

PoC

.

Mitigation

The `completeLimitOrderWithdrawal` function should be modified to validate the swap output instead of the total balance.

```
function completeLimitOrderWithdrawal(
  IWithdrawalVault.MultiInSingleOutData calldata _swapData,
  uint256 _expectedSwapOutput
) external returns (uint256 destTokenAmount) {
    // Send balance + _expectedSwapOutput to `WithdrawalVault.swapToSingleAsset`
}
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/dhedge/V2-Public/pull/10

Issue M-4: `_beforeTokenTransfer()` doesn't handle Burning, which allows DoSing ALL limit orders from being executed and forcing them to be executed at a loss

Source: https://github.com/sherlock-audit/2025-07-dhedge-update-judging/issues/72

Found by

0x37, 0xc0ffEE, j3x, silver_eth

Summary

`_beforeTokenTransfer()` doesn't handle the Burning case (as opposed to the Minting), which allows DoSing ALL limit orders of a certain pool from being executed and force them to be executed at a loss, preventing users from getting profit.

Root Cause

In case it's a minting, `_beforeTokenTransfer()` skips the cooldown check.

But that's not the cause with the burning, which can lead to exploiting this check to prevent ANY limit orders of a certain pool from being executed (we'll explain how later).

Unless the `address(0)` is whitelisted which leads to the execution of this line, but i doubt this will be the case!

How can an attacker exploit the cooldown period?

Let's take a look at the `getExitRemainingCooldown()` function which is used in the check that we're gonna exploit:

```
function getExitRemainingCooldown(address _depositor) public view returns (uint256
↪  remaining) {
    uint256 cooldownFinished =
    ↪  lastDeposit[_depositor].add(lastExitCooldown[_depositor]);

    if (cooldownFinished < block.timestamp) return 0;

    remaining = cooldownFinished.sub(block.timestamp);
}
```

So an attacker needs to manipulate `lastExitCooldown[easySwapperV2]`, so that when we add it to `lastDeposit[easySwapperV2]`, it will be >= `block.timestamp`.

We can do it by calling `PoolLogic::depositFor()` in that dHedge vault, which will call the internal function _depositFor() (we do it by depositing just 1 unit of a token, that's 0.000001 USD in the case of USDC).

Then, `_depositFor()` will set the `lastExitCooldown[easySwapperV2]` mapping right here. In case the pool has liquidity in it (which is gonna be the case most of the times), the cooldown will be calculated like this, and cause the `require(getExitRemainingCooldown(_from) == 0, "cooldown active");` check to fail the execution.

We know that when users create a limit order, the keeper will execute these orders when the price conditions are met - by calling `executeLimitOrdersSafe()` (or `executeLimitOrders()`), which calls the internal function `_executeLimitOrder()`, which calls the internal function `_processLimitOrderExecution()`.

When the keeper tries to execute the limit orders (using `executeLimitOrders()` OR `executeLimitOrdersSafe()` - both will lead to the failure of all the orders), the easySwapper.initLimitOrderWithdrawalFor() line will cause a revert, since `EasySwapperV2.sol::initLimitOrderWithdrawalFor()` calls PoolLogic::withdrawToSafe, and `withdrawToSafe()` burns the pool tokens here, which means they are transferred to `address(0)`, which leads to the trigger of the `_beforeTokenTransfer()` hook, which leads to `getExitRemainingCooldown(easySwapperV2)` being greater than 0 to cause the execution to fail for all orders, since the `_from` address here is the easySwapper's address for all the orders.

Internal Pre-conditions

- `address(0)` is not a whitelisted receiver (which i don't think it will be, it doesn't make sense to whitelist the 0 address).
- The pool has to be public OR the easy swapper is an allowed member - see here

External Pre-conditions

N/A

Attack Path

The attacker can have a bot that automates calling `depositFor(easySwapperV2)` each time the cooldown period is close to getting finished, that will DoS any limit orders from getting executed for a very long period of time. The deposit amount is just 1 unit (unit and not token), which is a neglectable loss. If we assume the token is deposit token is USDC, and the attacker calls `depositFor()` 50 times a day, that's only `0.01825` USD a year.

Explaining Why just 1 unit is enough

- Attacker deposits `_amount = 1` (0.000001 USDC or USDT).
- Then this calculation will become usdAmount = _assetValue(_asset, _amount) = 1 * 10 (0.000001 USD, USDC price = 1 USD, and the decimals of the price are 18 in the AssetHandler).
- For a non-empty pool, liquidityMinted = usdAmount * totalSupplyBefore / fundValue
- From the PoC : the pool has 4 WETH (price = 1200 USD if we use the developer's setup, and actually that price doesn't matter since it will get simplified in all cases) and 1000 USDC, so `fundValue = (4 * 1200 + 1000) * 1018 = 5800 * 1018`.
- `totalSupplyBefore = 5800 * 1018` (from initial deposits where liquidityMinted = usdAmount).
- So, `liquidityMinted = 1012 * 5800 * 1018 / 5800 * 1018 = 1012`.
- Since `1012 > 100_000`, the check require(liquidityMinted >= 100_000) passes.

Impact

- The attacker can prevent any limit orders of a certain pool from getting executed for a very long period of time with very minimal loss, since if the attacker calls `depositFor()` 50 times a day, that's only `0.01825` USD a year.
- The attacker can prevent users from getting profit: If the attacker notices that some orders will result in a profit for some users due to currentPriceD18 > limitOrder_.takeProfitPriceD18, he can DoS the execution until the price drops, and then stop the attack when he makes sure the order will be executed at a loss (`currentPriceD18 <= limitOrder_.stopLossPriceD18`).

So, executing limit orders is time-sensitive, due to this condition:

```
if (currentPriceD18 > limitOrder_.stopLossPriceD18 && currentPriceD18 <
↪  limitOrder_.takeProfitPriceD18)
  revert LimitOrderNotFillable(currentPriceD18, limitOrder_.stopLossPriceD18,
  ↪  limitOrder_.takeProfitPriceD18);
```

So since the attacker can [for a long period of time] prevent the execution of orders who currently pass the condition, until the condition no longer passes, we don't know when it will pass again.

And according to sherlock docs:

> **Could Denial-of-Service (DOS), griefing, or locking of contracts count as Medium (or High) severity issue?** To judge the severity we use two separate criteria: 1 - The issue causes funds to be locked for more than a week. 2 - The issue impacts the availability

of time-sensitive functions (cutoff functions are not considered time-sensitive). If at least one of these is describing the case, the issue can be Medium. If both apply, ***the issue can be considered High severity***

PoC

Add the following test to the `./test/integration/common/limitOrders/PoolLimitOrderManagerTestSetup.t.sol` file

```
function test_ExploitCooldownPeriod() public {
    // Preparations before creating a test pool
    IAssetHandlerMock assetHandler =
    ↪  IAssetHandlerMock(IPoolFactoryMock(poolFactory).getAssetHandler());
    vm.startPrank(assetHandler.owner());
    assetHandler.setChainlinkTimeout(86400 * 365);
    // This is instead of setting setExitCooldown to 0 (less code)
    IPoolFactoryMock(poolFactory).addReceiverWhitelist(address(poolLimitOrderManage
    ↪  rProxy)); // as you will see, this won't help preventing the issue

    // Create a pool with WETH and USDC as supported assets
    IHasSupportedAssetMock.Asset[] memory supportedAssets = new
    ↪  IHasSupportedAssetMock.Asset[](2);
    supportedAssets[0] = IHasSupportedAssetMock.Asset({asset: weth, isDeposit:
    ↪  true});
    supportedAssets[1] = IHasSupportedAssetMock.Asset({asset: usdc, isDeposit:
    ↪  true});

    address testPool = IPoolFactoryMock(poolFactory).createFund(
        false, user, "User", "Test Settlement Order", "TSO", 0, 0, supportedAssets
    );

    // Deposit WETH and USDC into pool
    deal(weth, user, 4e18);
    deal(usdc, user, 1000e6);
    vm.startPrank(user);
    IERC20(weth).approve(testPool, 4e18);
    IERC20(usdc).approve(testPool, 1000e6);
    IPoolLogic(testPool).deposit(weth, 4e18);
    IPoolLogic(testPool).deposit(usdc, 1000e6);
    assertEq(IERC20(weth).balanceOf(testPool), 4e18);
    assertEq(IERC20(usdc).balanceOf(testPool), 1000e6);
    assertGt(IERC20(testPool).totalSupply(), 0);

    assertEq(IERC20(weth).balanceOf(user), 0);
    assertEq(IERC20(usdc).balanceOf(user), 0);
    uint256 userTestPoolBalance = IERC20(testPool).balanceOf(user);
    assertGt(userTestPoolBalance, 0);
```

```
    // Create limit order in a same fashion as in _executeLimitOrder
    IPoolLogic(testPool).approve(address(poolLimitOrderManagerProxy),
    ↪  userTestPoolBalance);

    poolLimitOrderManagerProxy.createLimitOrder(
        PoolLimitOrderManager.LimitOrderInfo({
            amount: userTestPoolBalance,
            stopLossPriceD18: 0,
            takeProfitPriceD18: 1200e18,
            user: user,
            pool: testPool,
            pricingAsset: pricingAsset
        })
    );

    _setPricingAssetPriceD8(1200e8);

    // Execute limit order first to create settlement order
    PoolLimitOrderManager.LimitOrderExecution[] memory limitOrders =
        new PoolLimitOrderManager.LimitOrderExecution[](1);
    limitOrders[0] = PoolLimitOrderManager.LimitOrderExecution({
        orderId: _getLimitOrderId(user, testPool),
        complexAssetsData: _getEmptyPoolComplexAssetsData(testPool),
        amount: type(uint256).max
    });

    // attacker's depositFor tx
    address attacker = address(0x1337);
    deal(usdc, attacker, 1);
    vm.startPrank(attacker);
    IERC20(usdc).approve(testPool, 1);

    IPoolLogic(testPool).depositFor(address(PoolLimitOrderManager(poolLimitOrderMan
    ↪  agerProxy).easySwapper()), usdc, 1); // just 1 unit of usdc
    skip(6 hours); // to avoid the "can withdraw soon" error
    vm.stopPrank();

    vm.startPrank(keeper);
    vm.expectRevert("cooldown active");
    poolLimitOrderManagerProxy.executeLimitOrders(limitOrders);

}
```

Output:

```
                                [3839] 0xffFb5fB14606EB3a548C113026355020dDF27535::re
↪  ceiverWhitelist(0x0000000000000000000000000000000000000000) [staticcall]
                                    [3020] 0x61f9f48Bafe633e2ADfdbB9573419220bDEaEE59:
↪  :receiverWhitelist(0x0000000000000000000000000000000000000000) [delegatecall]
                                        ← [Return] false
                                    ← [Return] false
                                ← [Revert] revert: cooldown active
                            ← [Revert] revert: cooldown active
                        ← [Revert] revert: cooldown active
                    ← [Revert] revert: cooldown active
                ← [Revert] revert: cooldown active
            ← [Revert] revert: cooldown active
        ← [Revert] revert: cooldown active
    ← [Revert] revert: cooldown active
```

You notice in the logs that the contract checked if the `address(0)` was whitelisted, and since it wasn't, it proceeded to this check and reverted with `"cooldown active"`.

Mitigation

This issue can be avoided by checking if `_from` is a trusted EasySwapperV2, and `_to` is the 0 address, which means the current tx is a limit order execution tx.

Important Note about the README

The readme note about the `depositFor()` talks just about the impact in the context of the `PoolLogic` contract, i don't think the team are aware of the impact on the `PoolLimitOrderManager` contract which can force limit orders to be executed at a loss, therefore i don't think that rule includes this issue.

The note in the readme was mentioned under `Please discuss any design choices you made.`, and i don't think that allowing an attacker decide when orders are executed is a design choice.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/dhedge/V2-Public/pull/13

Issue M-5: Incorrect Streaming Fee Calculation

Source: https://github.com/sherlock-audit/2025-07-dhedge-update-judging/issues/128

Found by

000000, 0x37, Bigsam, kelcaM, newspacexyz, silver_eth

Summary

The `lastFeeMintTime` is not updated when the `streamingFee` is zero. This creates an issue when the `managementFeeNumerator` is zero for a period of time. If the manager later increases `managementFeeNumerator` to a positive value, the streaming fee calculation retroactively includes the period when the fee was supposed to be zero, resulting in users overpaying fees.

Root Cause

`lastFeeMintTime` is updated only when `streamingFee > 0`, ignoring the fact that `managementFeeNumerator` could have been zero:

https://github.com/sherlock-audit/2025-07-dhedge-update/blob/main/V2-Public/contracts/PoolLogic.sol#L814

```
if (streamingFee > 0) lastFeeMintTime = block.timestamp;
```

Because of this, past periods with zero management fee are incorrectly counted in future fee calculations.

Internal pre-conditions

1. Manager sets managementFeeNumerator to zero for a period of time.
2. Later increases it to a positive value.

External pre-conditions

.

Attack Path

.

Impact

Streaming fee is incorrectly charged for periods when `managementFeeNumerator` was zero. Users overpay fees, benefiting the manager unfairly.

PoC

I will skip two 2 week period before the fees are increased just to show the key of the issue.

`lastFeeMintTime` = 1000, `managementFeeNumerator` = 0, `block.timestamp` = 2000 → no fee should accrue, but `lastFeeMintTime` is not updated.

`managementFeeNumerator` = 100, `block.timestamp` = 3000, `lastFeeMintTime` = 1000 → the elapsed time of 2000 seconds is incorrectly includes (1000) in streaming fee calculation.

Users pay an inflated streaming fee covering the period when no fee should have been applied.

Mitigation

Update `lastFeeMintTime` when `managementFeeNumerator` is zero:

```
if (streamingFee > 0 || managementFeeNumerator == 0) lastFeeMintTime =
↪  block.timestamp;
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/dhedge/V2-Public/pull/8

Issue M-6: PoolLogic::onERC721Received doesnt confirm approved msg.sender

Source: https://github.com/sherlock-audit/2025-07-dhedge-update-judging/issues/135

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

Found by

silver_eth

Summary the function onERC721Received doesnt have access control meaning anyone can call it with a trusted operator to access the specified operator guard for the fiat money guards for example, the call will process to _verifyERC721

```
function _verifyERC721(address _operator, address _from, uint256 _tokenId) internal
↪  returns (bool verified) {
  // Leverage NFTs should be minted from Flat Money protocol, not transferred from
  ↪  other addresses
  require(_from == address(0), "nft not minted");

  // Get currently tracked NFTs
  uint256[] memory tokenIds = getOwnedTokenIds(msg.sender);

  // Loop through tracked NFTs and check the ownership of each ID. `ownerOf` call
  ↪  fails if owner is address(0) which means position was burnt.
  // Catch block removes this NFT from tracked NFTs. No checks on the owner are
  ↪  made because what's in tracker belongs to the vault by default.
  for (uint256 i; i < tokenIds.length; ++i) {
    try
      ILeverageModule(IDelayedOrder(_operator).vault().moduleAddress(FlatcoinModul
      ↪  eKeys._LEVERAGE_MODULE_KEY))
        .ownerOf(tokenIds[i])
    returns (
      address // solhint-disable-next-line no-empty-blocks
    ) {} catch {
      nftTracker.removeUintId({
        _guardedContract: _operator,
        _nftType: nftType,
        _pool: msg.sender,
        _nftID: tokenIds[i]
      });
    }
  }

  // This is the only place where NFT IDs are added to the tracker.
  nftTracker.addUintId({
    _guardedContract: _operator,
```

```
      _nftType: nftType,
      _pool: msg.sender,
      _nftID: _tokenId,
      _maxPositions: positionsLimit
    });

    verified = true;
  }
```

the nftTracker also doesnt ensure that a tokenId is not added twice so an attacker can either make the call with a nft already owned by the fund or with a nft owned by another address to make sure that the nft is not removed on the next call and as a result inflate the positionLimit note: the sponsors mentioned that 721s will not be supported assets but are still going to be held by the fund

Root Cause no access control on external call

Internal Pre-conditions none

External Pre-conditions none

Attack Path attacker calls onERC721Received with a trusted operator specifying tokens that have owners (either the fund or another address) until the positionsLimit is reached operator attempts to send in 721s for protocol operation but the call reverts due to limit having been reached

Impact dos of operator in sending nft tokens to the pool

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.