SHERLOCK

Security Review For EaseDeFi

Public Audit Contest Prepared For: **EaseDeFi**
Lead Security Expert: **0x73696d616f**
Date Audited: **November 17 - November 21, 2025**

Introduction

EaseDefi is a protocol focused on safe insurance alternatives for DeFi protocols via multiple improvements of ours and for Nexus Mutual. stNXM is an evolution of our arNXM product which acts as a Liquid Staking Token for Nexus Mutual's underwriting

Scope

Repository: EaseDeFi/stNXM-Contracts

Audited Commit: b1431014cfadc4e5b65a9b93dfb52092313fef31

Final Commit: 748eec3756a81e1a3c2558cec7add052ade520ae

Files:

- contracts/core/stNxmOracle.sol
- contracts/core/stNXM.sol
- contracts/core/stNxmSwap.sol
- contracts/general/Ownable.sol

Final Commit Hash

748eec3756a81e1a3c2558cec7add052ade520ae

Findings

Each issue has an assigned severity:

- High issues are directly exploitable security vulnerabilities that need to be fixed.
- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.

Issues Found

High	Medium
5	7

Issues Not Fixed and Not Acknowledged

High	Medium
0	0

Security experts who found valid issues

0rpse
0x73696d616f
0x97
0xAsen
0xCrypt0nite
0xDLG
0xDaniel_eth
0xDemon
0xDgiin
0xHexed
0xKemah
0xMosh
0xRaz
0xSecurious
0xYjs
0xbzbee42
0xc0ffEE
0xeix
0xgh0stcybers3c
0xleo
0xlucky
0xpetern
0xpoison
0xsai
0xsh
0xxAristos
5am
Abba.eth
AestheticBhai
Albert_Mei
Almanax
Anas22
Baccarat
Bluedragon
Bobai23
BoyD
Bozwkd
CasinoCompiler
ChargingFoxSec
Cybrid
DevBear0411
Drynooo
HeckerTrieuTien
Ibukun
IzuMan
JohnTPark24
JuggerNaut
MaratCerby
MysteryAuditor
Orhukl
OxSath404
Pataroff
Polaris_2
PowPowPow
Protokol
R4Y4N3
ReidnerM
Sa1ntRobi
Saad
Siiiisivan
Sir_Shades
SoarinSkySagar
Solea
SuperDevFavour
Theseersec
TianZun
Tricko
Vesko210
Vigilia-Aethereum
Wojack
X0sauce
Xmanuel
ZeronautX
Ziusz
akolade
algiz
asotu3
aster
asui
axelot
bbl4de
befree3x
blockace
boredpukar
codexNature
coffiasd
coin2own
cyberEth
d33p
dandan
dani3l526
dantehrani
deadmanwalking
dee_bug
destiny_rs
djshaneden
dobrevaleri
edger
eierina
elolpuer
elyas
ethaga
ezsia
farismaulana
firmanregar
frndz0ne
fullstop
gkrastenov

h2134
hanz
hy
iamephraim
insecureMary
itsRavin
ivxylo
jayjoshix
joicygiore
joshuam33
ke1caM
khaye26
kimnoic
kowolski
lucky-gru
makarov
makeWeb3safe
maxim371
merlin
n33k
ni8mare
oxwhite
peazzycole
pecata17107
richardo
s4bot3ur
securewei
securityresearcher
shieldrey
slavina
slcoder
th3hybrid
theboiledcorn
theholymarvycodes
thimthor
tobi0x18
touristS
typicalHuman
ubl4nk
vangrim
werulez99
xKeywordx
zaida
zcai

Issue H-1: Attacker can profit by manipulating Uniswap liquidity.

Source: https://github.com/sherlock-audit/2025-11-stnxm-by-easedefi-judging/issues/38

Found by

0rpse, 0x73696d616f, 0xAsen, 0xCrypt0nite, 0xDaniel_eth, 0xDemon, 0xDgiin, 0xHexed, 0xMosh, 0xYjs, 0xbzbee42, 0xeix, 0xlucky, 0xpetern, 0xsh, 0xxAristos, Abba.eth, Albert_Mei, Bobai23, BoyD, CasinoCompiler, Cybrid, Drynooo, HeckerTrieuTien, JohnTPark24, MaratCerby, Pataroff, Protokol, R4Y4N3, Saad, Siiiisivan, Sir_Shades, SuperDevFavour, Theseersec, Wojack, Xmanuel, ZeronautX, asotu3, axelot, befree3x, blockace, boredpukar, coin2own, cyberEth, d33p, dani3l526, dantehrani, deadmanwalking, destiny_rs, dobrevaleri, eierina, ezsia, farismaulana, fullstop, gkrastenov, h2134, hy, iamephraim, insecureMary, ivxylo, jayjoshix, joshuam33, ke1caM, khaye26, kimnoic, kowolski, lucky-gru, makeWeb3safe, maxim371, ni8mare, peazzycole, s4bot3ur, securewei, theholymarvycodes, tobi0x18, typicalHuman, ubl4nk, vangrim, xKeywordx, zcai

Summary

The dexBalances() function uses Uniswap's `slot0()` to read the current spot price, which can be manipulated within a single transaction. An attacker can swap a large amount into the Uniswap pool to inflate the DEX balance in `totalAssets()` and reduce virtual shares in `totalSupply()`, causing the exchange rate to spike. This allows users to withdraw at an inflated rate and profit at the expense of remaining stakers.

Root Cause

The `totalAssets` function, which is one of the most critical functions in the erc4626 system uses a `dexBalance` function to get the amount of the wNXM held by dex. This is essential because the stNXM is an owner of that liquidity, and it should account for it while calculating the exchange rate. But the problem is that the attacker can easily manipulate a Uniswap pool using a flash loan and inflate the stNXM price.

```
function dexBalances()
        public
        view
        returns (uint256 assetsAmount, uint256 sharesAmount)
    {
        (uint160 sqrtRatio, , , , , , ) = dex.slot0();

        for (uint256 i = 0; i < dexTokenIds.length; i++) {
```

```
            (uint256 posAmount0, uint256 posAmount1) = PositionValue.total(
                nfp,
                dexTokenIds[i],
                sqrtRatio
            );

            if (isToken0) {
                sharesAmount += posAmount0;
                assetsAmount += posAmount1;
            } else {
                sharesAmount += posAmount1;
                assetsAmount += posAmount0;
            }
        }
    }


function totalAssets() public view override returns (uint256) {
        // Add staked NXM, wNXM in the contract, wNXM in the dex and Morpho,
        ↪  subtract the admin fees.

        return stakedNxm() + unstakedNxm() - adminFees;
    }
```

Flow

Setup :

- Uniswap V3 pool (0.05% fee): 1,000 stNXM : 1,000 wNXM (1:1 ratio)
- stNXM vault free liquidity: 20,000 wNXM
- Total protocol: ~30,000 wNXM equivalent (including staked/morpho deposits)
- Alice holds: 10,000 stNXM shares

Step 1: Alice Requests Withdrawal

- Alice calls redeem(10,000 shares)
- Current exchange rate: ~1.0 wNXM per share
- Expected payout: ~10,000 wNXM
- Withdrawal queued for 2 days

Step 2: After 2 Days - Alice Constructs Batch Transaction Alice creates single transaction with following steps:

2a. Flashloan 10,000 wNXM from Morpho

- Borrowed: 10,000 wNXM

- Fee: 0.05% = 5 wNXM

2b. Swap 10,000 wNXM → stNXM on Uniswap

- Using constant product (x*y=k): 1,000 * 1,000 = 1,000,000
- After swap: stNXM_new * 11,000 = 1,000,000
- stNXM_new = 90.9 stNXM
- Alice receives: 1,000 - 90.9 = 909.1 stNXM
- Swap fee: 0.05% * 10,000 = 5 wNXM

So, pool state after swap:

- Before: 1,000 stNXM : 1,000 wNXM
- After: 90.9 stNXM : 11,000 wNXM

2c. Protocol Metrics Manipulated Before manipulation:

- totalAssets() = 20,000 (free) + 1,000 (dexWNXM) + ... = ~30,000 wNXM
- totalSupply() = 30,000 - 1,000 (virtual shares) = 29,000 shares
- Exchange rate = 30,000 / 29,000 = 1.034 wNXM/share
- Alice expects: 10,000 * 1.034 = 10,340 wNXM

After manipulation:

- totalAssets() = 20,000 (free) + 11,000 (dexWNXM inflated) + ... = ~40,000 wNXM
- totalSupply() = 30,000 - 90.9 (virtual shares reduced) = 29,909 shares
- Exchange rate = 40,000 / 29,909 = 1.337 wNXM/share
- Alice will receive: 10,000 * 1.337 = 13,370 wNXM

2d. Finalize Withdrawal

- Alice calls withdrawFinalize()
- Receives: 13,370 wNXM (inflated payout)
- Extra extracted: 13,370 - 10,340 = 3,030 wNXM

2e. Reverse Swap - Rebalance Pool

- Alice swaps back: 909.1 stNXM → wNXM
- Receives: ~9,000 wNXM (accounting for fees and slippage)
- Swap fee: ~4.5 wNXM

2f. Repay Flashloan

- Repay: 10,000 + 5 = 10,005 wNXM
- Alice's wNXM balance: 13,370 (withdrawal) + 9,000 (reverse swap) = 22,370 wNXM

- After repayment: 22,370 - 10,005 = 12,365 wNXM

Step 3: Calculate Profit

- Alice started with: 10,000 stNXM shares (worth ~10,340 wNXM)
- Alice ends with: 12,365 wNXM
- Net profit: 12,365 - 10,340 = 2,025 wNXM (~$141,750 @ $70/wNXM)
- Attack cost: ~15 wNXM (flashloan + swap fees)
- Remaining stakers lose: 3,030 wNXM in vault value

Internal Pre-conditions

None

External Pre-conditions

None

Attack Path

Check the root cause section.

Impact

This allows users to withdraw at an inflated rate and profit at the expense of remaining stakers.

PoC

Paste this in `stNXM.t.sol`

```
function test_ExchangeRateManipulation() public {
        // User deposits and requests withdrawal
        uint256 depositAmt = 1000 ether;
        depositWNXM(depositAmt, wnxmWhale);

        uint256 userShares = stNxm.balanceOf(wnxmWhale);
        withdrawWNXM(userShares, wnxmWhale);

        // Record expected payout before manipulation
        uint256 expectedPayout = stNxm.convertToAssets(userShares);

        // Step 2: Attacker manipulates Uniswap pool with large swap
        uint256 attackAmt = 10_000 ether;
```

```
        // Step 3: User finalizes withdrawal at inflated rate
        vm.warp(block.timestamp + 2 days + 1 hours);
         vm.startPrank(wnxmWhale);
        wNxm.approve(address(swapRouter), attackAmt);

        ISwapRouter.ExactInputSingleParams memory params = ISwapRouter
            .ExactInputSingleParams({
                tokenIn: address(wNxm),
                tokenOut: address(stNxm),
                fee: 500,
                recipient: wnxmWhale,
                deadline: block.timestamp + 1000,
                amountIn: attackAmt,
                amountOutMinimum: 0,
                sqrtPriceLimitX96: 0
            });

        swapRouter.exactInputSingle(params);
        vm.stopPrank();

        uint256 balanceBefore = wNxm.balanceOf(wnxmWhale);

        vm.prank(wnxmWhale);
        stNxm.withdrawFinalize(wnxmWhale);

        uint256 balanceAfter = wNxm.balanceOf(wnxmWhale);
        uint256 actualPayout = balanceAfter - balanceBefore;
        uint256 profit = actualPayout - expectedPayout;

        console2.log("Expected payout:", expectedPayout / 1e18);
        console2.log("Actual payout:", actualPayout / 1e18);
        console2.log("Profit:", profit / 1e18);

        // Verify vulnerability
        assertGt(
            actualPayout,
            expectedPayout,
            "User should receive inflated payout"
        );
    }
```

Mitigation

Replace slot0() with time-weighted average price:

```
function dexBalances() public view returns (uint256 assetsAmount, uint256
↪  sharesAmount) {
    // Use TWAP over 30 minutes instead of spot price
    uint32 twapPeriod = 1800; // 30 minutes
    uint160 sqrtRatio = OracleLibrary.getSqrtRatioAtTick(
        OracleLibrary.consult(address(dex), twapPeriod)
    );

    for (uint256 i = 0; i < dexTokenIds.length; i++) {
        (uint256 posAmount0, uint256 posAmount1) = PositionValue.total(
            nfp,
            dexTokenIds[i],
            sqrtRatio
        );
        // ... rest of the function
    }
}
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/EaseDeFi/stNXM-Contracts/commit/40630a0d88fdd663803afb73dfb9437ae01707ed

0xSimao

The fix to H-01 is having the vault using a 30-minute TWAP to calculate `dexBalances()`, which affects both `totalAssets()` and `totalSupply()`, meaning the system operates on stale pricing. If the current price is cheaper than TWAP, withdrawals receive more assets than fair value (insolvency risk); if more expensive, deposits acquire shares at a discount.

Additionally, an attacker could swap all stNXM out of the pool causing `totalSupply()` to reach 0, but `totalAssets()` decreases by the corresponding amount (to 0 or near 0, but still keeping the correct price due to the usage of virtual shares/assets in the OZ ERC4626 implementation, and correct rounding), bounding any arbitrage to the price difference rather than an exploit.

The 2-day withdrawal delay places another protection, as it breaks atomicity and allows TWAP to normalize before finalization. Deposits remain instant and unprotected, but a way to exploit them wasn't found.

Considering the lack of atomicity, difficulty in manipulating twap, and ease of arbitrage the team has decided it's not worth the extra complication and idiosyncratic behavior

Issue H-2: The vault can be drained

Source:
https://github.com/sherlock-audit/2025-11-stnxm-by-easedefi-judging/issues/127

Found by

0xeix, Tricko, asui, oxwhite, typicalHuman

Summary

The funds of vault can be drained by admin

Root Cause

stakeNxm is onlyOwner function and there is no validation of requested TokenId

Internal Pre-conditions

.

External Pre-conditions

.

Attack Path

1. `stakeNxm()` is owner-only but calls _stakeNxm from the vault contract, so inside depositTo() the msg.sender is address(this).
2. `depositTo()` only checks that `msg.sender (the vault)` is owner or approved for `requestTokenId`, and that the token belongs to the target pool.
3. An admin can keep ownership of an NFT (say tokenId 105), call `stakingNFT.approve(address(vault), 105)`, then invoke `stakeNxm(amount, pool, tranche, 105)`.
4. The vault unwraps its wNXM and stakes it, but the stake shares are credited to tokenId `105`, which is still owned by the admin.
5. When the admin later uses Nexus' withdraw on tokenId `105`, they receive the stake that was funded entirely by vault assets.
6. The vault, meanwhile, thinks it gained a staked position (it adds `tokenIdToPool[105]` and counts the tranche in `tokenIdToTranches[105]`), so `stakedNxm()` and `totalAssets()` are inflated with assets the vault does not actually control.

The Readme states:

> Are there any limitations on values set by admins (or other roles) in the codebase, including restrictions on array lengths?

> Owner (NOT proxy owner) should generally not be able to do anything that would allow them to steal from the vault.

Impact

1. Loss of funds
2. Incorrect accounting
3. Potential insolvency

PoC

No response

Mitigation

I think the vault's ownership of requested tokenId should be checked . Require `stakingNFT.ownerOf(_requestTokenId) == address(this)` before calling depositTo().

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/EaseDeFi/stNXM-Contracts/commit/beea701e46d16e9b68bbe61bf75efba406191895

Issue H-3: Owner can steal funds on withdraw by burning wrong Uniswap V3 position liquidity

Source: https://github.com/sherlock-audit/2025-11-stnxm-by-easedefi-judging/issues/176

Found by

blockace, elolpuer

Summary

From README: *If the owner is able to steal the funds that cannot be recovered by the protocol/proxy, then it may be a valid issue (if it leads to Med/High impact).* Owner can pass a `tokenId` from a different Uniswap V3 pool to `decreaseLiquidity()`, which will burn vault shares because the function does not verify the token's pool. This results in burning vault-held shares and allows the owner to gain more assets.

Root Cause

decreaseLiquidity() function does not check if `tokenId` is valid for `wNxm/stNxm` pool:

```
function decreaseLiquidity(uint256 tokenId, uint128 liquidity) external onlyOwner
↪  update {
    //@audit no check for valid tokenId
    INonfungiblePositionManager.DecreaseLiquidityParams memory params = ...

    nfp.decreaseLiquidity(params);
    (uint256 amount0, uint256 amount1) = nfp.collect(
        INonfungiblePositionManager.CollectParams(tokenId, address(this),
        ↪  type(uint128).max, type(uint128).max)
    );
    uint256 stNxmAmount = isToken0 ? amount0 : amount1;

    // Burn stNXM that was removed.
    if (stNxmAmount > 0) _burn(address(this), stNxmAmount);
    ...
}
```

This will allow owner to pass a `tokenId` from a different Uniswap V3 pool and burn shares to receive more assets during withdrawal.

Internal Pre-conditions

1. Vault has some withdraw requests

External Pre-conditions

No response

Attack Path

1. Owner creates a withdrawal request using their shares (e.g., when large whales also create withdrawal requests) and waits until it can be finalized.
2. Owner creates 2 ERC-20 tokens and deploys a Uniswap V3 pool with them.
3. Adds liquidity to the pool and mints an ERC-721 position.
4. Transfers the NFT position to the `stNXM` contract.
5. Calls `decreaseLiquidity()` with this position to burn vault shares belonging to others (from others withdrawal requests).
6. Since the function does not verify that the position belongs to the `wNxm/stNxm` pool, it burns vault shares.
7. With a reduced `totalSupply()`, the owner can now withdraw more assets.
8. Other users lose their ability to finalize withdrawals because the `burn()` will be DoSed in `withdrawFinalize()` cause this shares already burned by owner.

Impact

Owner steals tokens from vault, other user who had active withdrawal requests lose their tokens.

PoC

No response

Mitigation

Check that `tokenId` belongs to `wNxm/stNxm` pool.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/EaseDeFi/stNXM-Contracts/commit/807170825a6b993412c51e569cd35a191842685e

Issue H-4: Owner can steal funds from contract by calling stakeNxm() with fake pool address

Source: https://github.com/sherlock-audit/2025-11-stnxm-by-easedefi-judging/issues/251

Found by

0xRaz, CasinoCompiler, Cybrid, DevBear0411, Drynooo, JohnTPark24, Pataroff, asui, blockace, dandan, elolpuer, farismaulana, firmanregar, n33k, tobi0x18

Summary

Owner can pass any address to `_stakeNxm()`, including a fake staking pool with a matching interface, to add this pool into `tokenIdToPool` mapping. Then owner can set himself as the fee beneficiary and call `getRewards()`. The fake pool can return arbitrary values from `withdraw()`, inflating rewards so that `adminFees` equals the contract's entire `wNxm` balance, allowing the owner to withdraw all funds.

Root Cause

Owner can pass any address to stakeNxm() to add fake pool to storage:

```
function stakeNxm(uint256 _amount, address _poolAddress, uint256 _trancheId,
↪  uint256 _requestTokenId)
    external
    onlyOwner
    update
{
    _stakeNxm(_amount, _poolAddress, _trancheId, _requestTokenId);
}

function _stakeNxm(uint256 _amount, address _poolAddress, uint256 _trancheId,
↪  uint256 _requestTokenId) internal {
    wNxm.unwrap(_amount);
    // Make sure it's the most recent token controller address.
    nxm.approve(nxmMaster.getLatestAddress("TC"), _amount);

    IStakingPool pool = IStakingPool(_poolAddress);
    uint256 tokenId = pool.depositTo(_amount, _trancheId, _requestTokenId,
    ↪  address(this));
    tokenIdToTranches[tokenId].push(_trancheId);
    // if new nft token is minted we need to keep track of
    // tokenId and poolAddress inorder to calculate assets
    // under management
```

```
    if (tokenIdToPool[tokenId] == address(0)) {
        tokenIds.push(tokenId);
        tokenIdToPool[tokenId] = _poolAddress;
    }
}
```

Next owner can update `beneficiary` to himself:

```
function changeBeneficiary(address _newBeneficiary) external onlyOwner {
    beneficiary = _newBeneficiary;
}
```

And then call `getRewards()` so pool will return fake `rewards` value and `adminFees` will be inflated:

```
function getRewards() external update returns (uint256 rewards) {
    for (uint256 i = 0; i < tokenIds.length; i++) {
        uint256 tokenId = tokenIds[i];
        rewards += _withdrawFromPool(tokenIdToPool[tokenId], tokenId, false, true,
        ↪  tokenIdToTranches[tokenId]);
    }
    rewards += collectDexFees();
    adminFees += rewards * adminPercent / DIVISOR;
    ...
}

function _withdrawFromPool(
    ...
) internal returns (uint256 amount) {
    IStakingPool pool = IStakingPool(_poolAddress);
    (uint256 withdrawnStake, uint256 withdrawnRewards) =
        pool.withdraw(_tokenId, _withdrawStake, _withdrawRewards, _trancheIds);
    amount = withdrawnStake + withdrawnRewards;
    wNxm.wrap(amount);
}
```

Next he will call `withdrawAdminFees()` to withdraw funds:

```
function withdrawAdminFees() external update {
    if (adminFees > 0) wNxm.transfer(beneficiary, adminFees);
    adminFees = 0;
}
```

Internal Pre-conditions

No response

External Pre-conditions

No response

Attack Path

1. User deposits `100e18 wNXM` into the `stNXM` contract.
2. Owner creates a contract that matches the `StakingPool` interface.
3. Owner calls `stakeNxm()` with his pool address and `amount = 100e18`.
4. The contract unwraps `wNXM`, `pool.depositTo()` returns some `tokenId`, which is stored together with the pool address.
5. Owner calls `changeBeneficiary()` and sets the beneficiary to himself (assume a 10% fee).
6. Owner calls `getRewards()`. The contract iterates over all pools, and the fake pool returns `withdrawnRewards = 100e18`.
7. The contract wraps `100e18 NXM` into `wNXM`.
8. `adminFees` increases by `10e18 wNXM` (the 10% fee).
9. The owner repeats steps 3 and 6 with slightly adjusted amounts, adding almost the entire `100e18` to `adminFees`.
10. Owner calls `withdrawAdminFees()` and steals the funds from the contract.

Impact

Owner will steal funds from the contract.

PoC

No response

Mitigation

Ensure that `poolAddress` is valid in `stakeNxm()`.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/EaseDeFi/stNXM-Contracts/commit/448a741b44d3588c490b62134ca9e330e270a8ed

Issue H-5: owner can hide a staked nft via removeTokenIdAtIndex and inflate shares using this to make profit

Source: https://github.com/sherlock-audit/2025-11-stnxm-by-easedefi-judging/issues/878

Found by

0xKemah, 0xxAristos, 5am, AestheticBhai, Baccarat, Cybrid, Drynooo, IzuMan, PowPowPow, Solea, TianZun, Ziusz, asui, bbl4de, boredpukar, deadmanwalking, elolpuer, hanz, iamephraim, ivxylo, jayjoshix, peazzycole, slcoder, thimthor, tobi0x18, xKeywordx, zaida

Summary

The ability for the owner to remove a staked Nexus NFT from the tokenIds array without withdrawing its stake will cause manipulation and profit, as the owner can hide an NFT's stake from `stakedNxm()/totalAssets()`, deposit while the vault appears under□collateralized, then re□expose the nft and redeem the over□minted shares at a higher price

Root Cause

```
function removeTokenIdAtIndex(uint256 _index) external onlyOwner {
    uint256 tokenId = tokenIds[_index];
    tokenIds[_index] = tokenIds[tokenIds.length - 1];
    tokenIds.pop();
    // remove mapping to pool
    delete tokenIdToPool[tokenId];
}
```

https://github.com/EaseDeFi/stNXM-Contracts/blob/b1431014cfadc4e5b65a9b93dfb52092313fef31/contracts/core/stNXM.sol#L723 when a token id is removed the akedNxm() computes the vault's Nexus stake by looping only over tokenIds and, for each, using tokenIdToPool[token] plus tokenIdToTranches[token] to attribute a fraction of activeStake to the vault Removing an NFT from tokenIds causes all of its stake to disappear from stakedNxm() and thus from totalAssets() = stakedNxm() + unstakedNxm() - adminFees, while the NFT itself (and its Nexus stake) still belongs to the vault. Later, the owner can call stakeNxm(.., _requestTokenId = tokenId) to reuse the same NFT, at which point _stakeNxm will re□insert the tokenId into tokenIds and tokenIdToPool, restoring that stake into stakedNxm() and totalAssets()

Internal Pre-conditions

na just owner needs to have funds

External Pre-conditions

na

Attack Path

to show this simple bug assume -

- Real underlying: 10,000 wNXM in NFT k + 5,000 wNXM across other tracked NFTs = 15,000.
- tokenIds includes k, and tokenIdToPool[k] points to the correct Nexus pool.
- stakedNxm() correctly sums 15,000, so totalAssets() 15,000 and totalSupply() = 15,000 (1:1 price).

Owner hides the large NFT from accounting

- Owner calls removeTokenIdAtIndex(i) where tokenIds[i] == k.
- Function swaps out H, pops the array, and deletes tokenIdToPool[k], but does not withdraw or modify the underlying stake in Nexus.
- stakedNxm() now loops only over the remaining tokenIds, so it sees only 5,000 wNXM; the 10,000 wNXM in NFT k is no longer included.
- totalAssets_view drops to 5,000 while the real underlying remains 15,000. Owner deposits while totalAssets() is artificially low

Owner calls deposit(10,000, owner).

ERC□4626 computes minted shares with stale values: shares = depositAmount × totalSupply / totalAssets shares = 10,000 × 15,000 / 5,000 = 30,000 shares At the true totalAssets = 15,000, the owner should only receive 10,000 shares; they instead over□mint by 20,000 shares (3x). Owner re-adds the hidden NFT to accounting. Later, the owner calls stakeNxm(_smallAmount, poolForK, someTranche, k). _stakeNxm then calls pool.depositTo(..., _requestTokenId = k, address(this)), which returns the same tokenId = k. Because tokenIdToPool[k] was deleted earlier, _stakeNxm treats this as a new NFT: it pushes k back into tokenIds and sets tokenIdToPool[k] = poolForK. From this point on, stakedNxm() once again includes the full stake of NFT k, causing totalAssets() to jump to reflect □25,000 underlying (5,000 other + 10,000 hidden + 10,000 owner deposit, ignoring any small new stake). Owner redeems at the corrected higher price. With the full underlying now visible, the ERC-4626 share price is computed using totalAssets □ 25,000 and totalSupply = 45,000 (15,000 original + 30,000 owner). The owner then redeems a large portion (or all) of the 30,000 shares minted earlier, receiving significantly more wNXM than the 10,000 they originally deposited because the underlying NFT k is now fully counted in totalAssets(). The excess payout is effectively funded by diluting other stakers' proportional claim over the true underlying value.

Impact

All stNXM holders other than the owner face loss as admin is basically stealing with this even if the owner is changed even doing this one time will make the protocol loss a huge chunk

PoC

No response

Mitigation

No response

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/EaseDeFi/stNXM-Contracts/commit/beea701e46d16e9b68bbe61bf75efba406191895

Issue M-1: pushing same trancheId to the tokenIdToTranches will inflate stakedNxm accounting.

Source: https://github.com/sherlock-audit/2025-11-stnxm-by-easedefi-judging/issues/178

Found by

0xAsen, 0xDLG, 0xYjs, 0xc0ffEE, 0xleo, AestheticBhai, Albert_Mei, Cybrid, Drynooo, MysteryAuditor, Orhukl, OxSath404, R4Y4N3, Solea, Vesko210, Wojack, Ziusz, algiz, asui, bbl4de, blockace, boredpukar, codexNature, coffiasd, cyberEth, elyas, frndz0ne, h2134, itsRavin, ivxylo, oxwhite, richardo, slcoder, theboiledcorn, thimthor, werulez99, zaida

Vulnerability Details

Tranche IDs are used to track accounting and expiry dates, and it is normal for stakers to use the same tranche ID to update their position.

When owner stakes NXM, the given `_trancheId` is added into `tokenIdToTranches` list.

```
tokenIdToTranches[tokenId].push(_trancheId);
```

The problem is the function always adds the tranche ID to the list, without validating if it already exist. This leads to duplicate tranch IDs in the list.

Now, when stakedNxm is calculated it will return accumulation of assets for every tranche in the list;

```
uint256[] memory trancheIds = tokenIdToTranches[token];
```

```
for (uint256 j = 0; j < trancheIds.length; j++) {
    uint256 tranche = trancheIds[j];
    (,, uint256 stakeShares,) = IStakingPool(pool).getDeposit(token, tranche);

    if (tranche < currentTranche) {
        (, uint256 amountAtExpiry, uint256 sharesAtExpiry) =
        ↪  IStakingPool(pool).getExpiredTranche(tranche);
        // Pool has been properly expired here.
        if (sharesAtExpiry > 0) assets += (amountAtExpiry * stakeShares) /
        ↪  sharesAtExpiry;
        // The tranche ended but expirations have not been processed.
        else assets += (activeStake * stakeShares) / stakeSharesSupply;
    } else {
        assets += (activeStake * stakeShares) / stakeSharesSupply;
```

Since there are duplicate tranche IDs in the list assets amount calculated will be inflated.

And as a result `totalAssets` will be affected, which in turn breaks deposit/withdraw and other functionalities depending on `totalAssets`.

Note that there is a public resetTranches() function, but that is not forced and a user who can profit from it will likely not call it before redeeming. A redeem can also happen right after the inflation, since the resetTranches is not guaranteed.

Impact

higher `totalAssets` as a result of inflated `stakedNxm`. This will lead to users receiving incorrect minting & redeemig amounts.

This does not need the owner to be malicious, it will happen on normal conditions.

Well, even if there is a keeper bot to call resetTranches() in that case owner can intentionally be malicious and inflate the totalAssets and redeem right after duplicating the same tranche ID.

Owner malicious or not, Either way, this is an issue.

Recomended mitigation

push(add) only when the trancheId is not on the `tokenIdToTranches` list.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/EaseDeFi/stNXM-Contracts/commit/41e13ca1433bc4f800547b90c18b6e12e8208451

Issue M-2: Missing Tranche Tracking After `extendDeposit()` Causes Temporary Asset Underreporting

Source: https://github.com/sherlock-audit/2025-11-stnxm-by-easedefi-judging/issues/212

Found by

0x73696d616f, 0x97, 0xDLG, 0xgh0stcybers3c, 0xsai, AestheticBhai, Drynooo, HeckerTrieuTien, IzuMan, Orhukl, OxSath404, Solea, Vesko210, Ziusz, asui, axelot, blockace, edger, ivxylo, oxwhite, slcoder, theboiledcorn, thimthor, werulez99, zaida

Summary

The `extendDeposit()` function moves a staking position from one tranche to another in the Nexus Mutual pool but fails to update the `tokenIdToTranches` mapping. When `stakedNxm()` is subsequently called, it only queries the old tranche (which now has 0 deposits) and misses the extended position in the new tranche. This causes `totalAssets()` to underreport vault assets until `resetTranches()` is called, creating a temporary but real accounting discrepancy.

Location: stNXM.sol

Vulnerable Code:

```
function extendDeposit(uint256 _tokenId, uint256 _initialTrancheId, uint256
↪  _newTrancheId, uint256 _topUpAmount)
    external
    onlyOwner
    update
{
    address stakingPool = tokenIdToPool[_tokenId];

    if (_topUpAmount > 0) {
        wNxm.unwrap(_topUpAmount);
        nxm.approve(nxmMaster.getLatestAddress("TC"), _topUpAmount);
    }

    IStakingPool(stakingPool).extendDeposit(_tokenId, _initialTrancheId,
    ↪  _newTrancheId, _topUpAmount);
    //  Missing: tokenIdToTranches[_tokenId].push(_newTrancheId);
}
```

Correct Pattern (from `_stakeNxm()`):

```
function _stakeNxm(uint256 _amount, address _poolAddress, uint256 _trancheId,
↪  uint256 _requestTokenId) internal {
```

```
    // ... approval logic ...
    uint256 tokenId = pool.depositTo(_amount, _trancheId, _requestTokenId,
    ↪  address(this));
    tokenIdToTranches[tokenId].push(_trancheId); //  Correctly tracks new tranche
    // ...
}
```

Impact

Accounting Error:

- After `extendDeposit()`, the new tranche is NOT tracked in `tokenIdToTranches`
- `stakedNxm()` loops through tracked tranches and queries Nexus: `getDeposit(tokenId, tranche)`
- For the new tranche: `getDeposit(tokenId, newTranche)` returns the extended funds but is never queried
- Result: `totalAssets()` underreports by the amount of the extended stake

Exploitation Scenario:

```
1. Initial state: 10,000 wNXM staked in tranche T1
   - totalAssets = 10,000
   - totalSupply = 10,000 shares
   - Share price = 1.0

2. Owner calls: extendDeposit(tokenId, T1, T2, 5000)
   - T2 now has 10,000 + 5000 = 15,000 wNXM
   - But tokenIdToTranches still = [T1]

3. stakedNxm() calculation:
   - Queries getDeposit(tokenId, T1) → 0 (was moved)
   - Never queries T2 (not tracked)
   - Returns 0

4. totalAssets now reports: 0 (artificially low!)
   - Share price drops to 0

5. Attacker deposits 5,000 wNXM:
   - Gets 5,000 / 0 = very high number of shares
   - Extreme share inflation

6. Someone calls resetTranches():
   - Discovers T2 has deposits
   - Updates tokenIdToTranches = [T2]
   - totalAssets corrects back to 15,000
```

```
7. Attacker redeems shares at corrected price
   - Extracts significantly more value than deposited
```

Severity Justification:

- Temporary accounting error that self-corrects within 91 days
- Requires specific conditions (owner using `extendDeposit`)
- Exploitable but correction is permissionless and quick
- Financial impact: Medium (temporary share inflation/deflation)

POC

```
function testExtendDepositAccountingError() public {
    // Setup: Initial staking position
    uint256 initialStake = 10000 ether;
    depositWNXM(initialStake, wnxmWhale);

    uint256 currentTranche = block.timestamp / 91 days;
    stakeNxm(initialStake, riskPools[0], currentTranche, tokenIds[0]);

    // Record initial state
    uint256 initialAssets = stNxm.totalAssets();
    uint256 initialSupply = stNxm.totalSupply();

    console.log("Before extend:");
    console.log("  totalAssets:", initialAssets);
    console.log("  totalSupply:", initialSupply);
    console.log("  tracked tranches:", stNxm.tokenIdToTranches(tokenIds[0]).length);

    // Owner extends deposit to new tranche
    uint256 extensionAmount = 5000 ether;
    uint256 newTranche = currentTranche + 4;

    vm.startPrank(multisig);
    stNxm.extendDeposit(tokenIds[0], currentTranche, newTranche, extensionAmount);
    vm.stopPrank();

    // Check accounting after extension (BEFORE resetTranches)
    uint256 assetsAfterExtend = stNxm.totalAssets();
    uint256[] memory tranchesAfter = stNxm.tokenIdToTranches(tokenIds[0]);

    console.log("\nAfter extend (before resetTranches):");
    console.log("  totalAssets:", assetsAfterExtend);
    console.log("  tracked tranches length:", tranchesAfter.length);

    // BUG: totalAssets doesn't increase
    // Extended funds exist in Nexus but aren't counted
```

```
    uint256 expectedIncrease = extensionAmount; // Should increase by at least this
    require(assetsAfterExtend < initialAssets + expectedIncrease, "Assets should
    ↪  have increased more");

    // Verify funds actually exist in Nexus pool
    IStakingPool pool = IStakingPool(riskPools[0]);
    (,, uint256 sharesInNewTranche,) = pool.getDeposit(tokenIds[0], newTranche);
    require(sharesInNewTranche > 0, "Deposit exists in new tranche");

    // Now call resetTranches - this fixes the accounting
    stNxm.resetTranches();

    uint256 assetsAfterReset = stNxm.totalAssets();
    uint256[] memory tranchesAfterReset = stNxm.tokenIdToTranches(tokenIds[0]);

    console.log("\nAfter resetTranches:");
    console.log("  totalAssets:", assetsAfterReset);
    console.log("  tracked tranches length:", tranchesAfterReset.length);

    // NOW assets are correct
    require(assetsAfterReset >= initialAssets + (extensionAmount / 2), "Assets
    ↪  corrected after reset");
    require(tranchesAfterReset.length >= 2, "New tranche now tracked");
}
```

Recommendation

Add tranche tracking immediately after extending the deposit:

```
function extendDeposit(uint256 _tokenId, uint256 _initialTrancheId, uint256
↪  _newTrancheId, uint256 _topUpAmount)
    external
    onlyOwner
    update
{
    address stakingPool = tokenIdToPool[_tokenId];

    if (_topUpAmount > 0) {
        wNxm.unwrap(_topUpAmount);
        nxm.approve(nxmMaster.getLatestAddress("TC"), _topUpAmount);
    }

    IStakingPool(stakingPool).extendDeposit(_tokenId, _initialTrancheId,
    ↪  _newTrancheId, _topUpAmount);

    // Fix: Track the new tranche immediately
    tokenIdToTranches[_tokenId].push(_newTrancheId);
```

```
    // Optional: Consider removing _initialTrancheId if fully moved
    // This prevents querying a tranche with 0 deposits
}
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/EaseDeFi/stNXM-Contracts/commit/40630a0d88fdd663803afb73dfb9437ae01707ed

Issue M-3: Expired tranche cannot be extended due to lack of token allowance

Source: https://github.com/sherlock-audit/2025-11-stnxm-by-easedefi-judging/issues/305

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

Found by

0xYjs, h2134, hy, ni8mare, securityresearcher

Summary

The StNXM contract contains a functional issue in the `extendDeposit()` flow where deposit extension fails for expired tranches due to missing NXM token allowance.

Root Cause

In `extendDeposit()`, when handling expired tranches, StakingPool attempts to withdraw and then redeposit funds, but StNXM contract fails to provide sufficient NXM allowance to TokenController for the complete operation.

Code Flow:

1. StNXM calls extendDeposit:

stNXM.sol#L379:

```
IStakingPool(stakingPool).extendDeposit(_tokenId, _initialTrancheId, _newTrancheId,
↪  _topUpAmount);
```

2. StakingPool handles expired tranche:

StakingPool#extendDeposit():

```
      // if the initial tranche is expired, withdraw everything and make a new
      ↪  deposit
      // this requires the user to have grante sufficient allowance
      if (initialTrancheId < _firstActiveTrancheId) {

        uint[] memory trancheIds = new uint[](1);
        trancheIds[0] = initialTrancheId;

@>      (uint withdrawnStake, /* uint rewardsToWithdraw */) = withdraw(
          tokenId,
          true, // withdraw the deposit
```

```
          true, // withdraw the rewards
          trancheIds
        );

@>      depositTo(withdrawnStake + topUpAmount, targetTrancheId, tokenId,
↪   msg.sender);

        return;
        // done! skip the rest of the function.
      }
```

3. StNXM only approves top-up amount

stNXM.sol#L374-L377:

```
if (_topUpAmount > 0) {
    wNxm.unwrap(_topUpAmount);
    nxm.approve(nxmMaster.getLatestAddress("TC"), _topUpAmount);
}
```

The issue occurs because StNXM only approves the `_topUpAmount` for TokenController, but the `depositTo()` call requires approval for the full amount (withdrawnStake + topUpAmount). When TokenController attempts transfer the assets, it fails due to insufficient allowance.

Internal Pre-conditions

- StNXM contract holds a staking NFT with an expired tranche
- Owner attempts to extend the expired tranche to a future tranche

External Pre-conditions

None required

Attack Path

1. Owner calls `extendDeposit()` with an expired initial tranche;
2. Withdrawal from expired tranche completes successfully;
3. Redeposit attempt fails due to missing allowance;
4. Transaction reverts after partial completion;
5. Funds remain in StNXM contract, extension incomplete.

Impact

Deposit extension operations for expired tranches fail completely, leaving funds stranded in the StNXM contract after successful withdrawal. This breaks the core functionality of tranche extension, rendering a fundamental feature of the staking system inoperable for expired positions.

PoC

No response

Mitigation

Approve sufficient NXM allowance to TokenController in the StNXM contract to cover the full deposit amount (withdrawnStake + topUpAmount) when extending an expired tranche.

Issue M-4: Admin fees are applied to NMX tokens during migration

Source: https://github.com/sherlock-audit/2025-11-stnxm-by-easedefi-judging/issues/444

Found by

dandan, h2134, merlin

Summary

The context of deployment of `stNXM.sol`:

1)Launch contract and initialize 2)Pause contract 3)Transfer over NFTs, NXM, and ownership from the old arNXM 4)Initialize External 5) Unpause

A malicious user or even the owner can charge admin fees on all NXM tokens from the old `arNXM` between steps 3 and 4.

Root Cause

The update modifier can be exploited by a malicious user or the owner to charge admin fees on NXM tokens migrating from arNXM to stNXM:

```
if (balance > lastBalance && lastStaked <= staked) {
            adminFees += (balance - lastBalance) * adminPercent / DIVISOR;
        }
```

Internal Pre-conditions

1. Launch the stNXM contract, initialize it, and pause it.
2. Transfer NXM tokens from the old arNXM contract.

External Pre-conditions

None

Attack Path

1) At this step, the `stNXM` contract is paused and NXM tokens are held on its balance.
2) A malicious user or the owner calls `resetTranches`:

```
 function resetTranches() public update { // update
        uint256 firstTranche = (block.timestamp / 91 days) - 1;

        for (uint256 i = 0; i < tokenIds.length; i++) {  // tokenIds isn't
        ↪  initialized
}}
```

3. The `resetTranches` function called because it is not paused and it executes the `update` modifier. `balance > lastBalance` is true because `lastBalance = 0` and `balance` equals all tokens from arNXM. `stakedNxm` returns 0; `lastStaked <= staked` is true (0 <= 0)

```
modifier update() {
     uint256 nxmBalance = nxm.balanceOf(address(this));
     if (nxmBalance > 0) wNxm.wrap(nxmBalance);

     uint256 balance = wNxm.balanceOf(address(this));
     uint256 staked = stakedNxm(); // 0

     // amount of NXM from arNXM > 0 && 0 <= 0
     if (balance > lastBalance && lastStaked <= staked) {
         adminFees += (balance - lastBalance) * adminPercent / DIVISOR;
     }

     _;

     lastBalance = wNxm.balanceOf(address(this));
     lastStaked = stakedNxm();
 }
```

4. The `initializeExternals` function is executed.

Impact

All migrated NXM tokens can be unfairly reduced by admin fees, causing users to lose part of their balance during the migration from arNXM to stNXM.

PoC

No response

Mitigation

Consider changing the `update` modifier:

```
- if (balance > lastBalance && lastStaked <= staked) {
+ if (balance > lastBalance && staked != 0 && lastStaked <= staked) {
            adminFees += (balance - lastBalance) * adminPercent / DIVISOR;
        }
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/EaseDeFi/stNXM-Contracts/commit/86decc1d35b729b342e3d5ceee29ba62becf35c0

Issue M-5: Uniswap Pool price can be manipulated so stNXM adds liquidity to a super imbalanced pool and loses funds

Source: https://github.com/sherlock-audit/2025-11-stnxm-by-easedefi-judging/issues/457

Found by

0x73696d616f, 0xc0ffEE, 0xeix, 0xlucky, 0xpoison, Abba.eth, Bluedragon, BoyD, Bozwkd, Ibukun, Orhukl, Polaris_2, Protokol, SoarinSkySagar, Vesko210, Vigilia-Aethereum, akolade, blockace, dee_bug, farismaulana, gkrastenov, hanz, ivxylo, kowolski, makarov, ni8mare, oxwhite, shieldrey, theboiledcorn, theholymarvycodes, typicalHuman, werulez99

Summary

stNXM::initializeExternals() adds liquidity without slippage checks, so it may add to an extremely unbalanced pool and get all funds stolen.

An attacker can do this by frontrunning the stNXM::initializeExternals() call and swapping in the `dex` to move the price to the lowest possible, so the tick becomes -887272, then liquidity is added in the full range at this tick, which adds the full `wNXM` amount to the `dex`, and 0 `stNXM`.

After this, the attacker can swap just a bit of `stNXM`, such as 0.01 ether, `and steal the whole` wNXM' from the DEX.

The attacker can even spam this for no cost to make sure it goes through, as soon as it detects the new dex created, which emits and event.

Root Cause

In `stNXM.sol:120`, there is no slippage protection when minting.

Internal Pre-conditions

None

External Pre-conditions

None

Attack Path

1. Admin creates dex and initializes tick to price 1.
2. Attacker pushes tick to -887270 with a null swap, frontrunning the admin.
3. Admin calls `stNXM::initializeExternals()`, adding liquidity to the unbalanced pool, which adds only `wNXM`.
4. Attacker gets all `wNXM` from the pool by swapping only some `stNXM` that they got from the swap contract or from `stNXM::deposit()` itself.

Impact

All `wNXM` in the dex is stolen.

PoC

Add this test to `stNXM.t.sol`:

```
function test_POC_swap_steal() public {
    vm.startPrank(address(stNxmSwap));
    stNxm.approve(address(swapRouter), 0.01 ether);
    ISwapRouter.ExactInputSingleParams memory params =
    ↪  ISwapRouter.ExactInputSingleParams(
        address(stNxm), address(wNxm), 500, address(stNxmSwap), 1000000000000000,
        ↪  0.01 ether, 0, 0
    );
    swapRouter.exactInputSingle(params);
    vm.stopPrank();
}
```

In the setup function, add the following after the line `IUniswapV3Pool(dex).initialize(7922816251426433759354395033 6);`:

```
vm.startPrank(address(this));
wNxm.approve(address(swapRouter), 1 ether);
dex.swap(
    address(this),
    true,
    int256(1 ether),
    4295128740,
    abi.encode("")
);
vm.stopPrank();
```

Add the swap function to the UniswapV3Pool.sol interface file:

```
interface IUniswapV3Pool is IUniswapV3PoolState {
    function swap(
        address recipient,
        bool zeroForOne,
        int256 amountSpecified,
        uint160 sqrtPriceLimitX96,
        bytes calldata data
    ) external returns (int256 amount0, int256 amount1);
}
```

It shows the attacker stealing all the `wNXM`.

Mitigation

Swap to place the correct price in the pool before adding liquidity to the dex.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/EaseDeFi/stNXM-Contracts/commit/0e3f06d7e23be25e278db034560423d77ba040f0

Issue M-6: Wrong price validation in oracle causes early DoS and allows late-stage price manipulation

Source: https://github.com/sherlock-audit/2025-11-stnxm-by-easedefi-judging/issues/708

Found by

0x73696d616f, 0xAsen, 0xMosh, 0xSecurious, 0xc0ffEE, 0xeix, 0xgh0stcybers3c, ChargingFoxSec, HeckerTrieuTien, JuggerNaut, ReidnerM, Sa1ntRobi, Sir_Shades, Solea, SuperDevFavour, TianZun, Vesko210, Wojack, X0sauce, ZeronautX, aster, befree3x, boredpukar, destiny_rs, djshaneden, ethaga, h2134, iamephraim, joicygiore, khaye26, kimnoic, maxim371, merlin, ni8mare, pecata17107, securewei, shieldrey, slavina, th3hybrid, touristS, typicalHuman, werulez99

Summary

Sanity check's reliance on `elapsedTime` will reject normal price shift in early stages of the protocol and allow price manipulation later on

Root Cause

In oracle contract implementation dedicated for Morpho market, `sanePrice` function validates price of `stNXM` in `wNXM` terms in the following way:

```
function sanePrice(uint256 _price) public view returns (bool) {
        // Amount of 1 year it's been
@>      uint256 elapsedTime = block.timestamp - startTime;
        // If price is lower than equal it's not too high.
        if (_price < 1e18) return true;

@>      uint256 apy = (_price - 1e18) * 31_536_000 / elapsedTime;
        return apy <= saneApy;
    }
```

Such APY formula will magnify small normal shifts in TWAP price in early stages of the protocol, when `elapsedTime` is small. Consider the following situation:

1. `elapsedTime` is equal to 7 days
2. For this period of time, price was higher at around 1% deviation mark - `1.01e18`, which is normal price shifts and is definitely not price manipulation
3. The resulting APY at 7 day mark (highest denominator) - around `5.2e17`, which will cause a revert on this line. Quoting an oracle with such price deviation on DEX would revert as well, since denominator is smaller

For big `elapsedTime` values, big price deviations will be rendered as valid:

1. `elapsedTime` is equal to 3 years
2. Price at the moment of quoting is `2.5e18` - `1.5e18` more than target `1e18` price
3. Resulting APY is at borderline - `5e17`, which allows such price to pass

First scenario will break the Morpho market usage, since any operation that would trigger `_isHealthy` check would revert due to price being rejected in oracle

Internal Pre-conditions

None

External Pre-conditions

None

Attack Path

There is no real attack path, since normal operations will cause Morpho disturbance

Impact

For intial stages, when `elapsedTime` is small - Morpho usage is easily disturbed by normal price movements For later stages, when `elapsedTime` is big - abnormal prices that may happen during low liquidity periods can pass

PoC

No response

Mitigation

No response

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/EaseDeFi/stNXM-Contracts/commit/7a9abd4e78bfb29416e9677040aa3650999ee353

Issue M-7: Missing Uniswap V3 cardinality initialization in `StNxmOracle` leads to permanent Denial of Service

Source: https://github.com/sherlock-audit/2025-11-stnxm-by-easedefi-judging/issues/899

Found by

0x73696d616f, 0xAsen, Almanax, Anas22, Sir_Shades, befree3x, hanz, khaye26, thimthor

Summary

The `StNxmOracle` contract calculates the price of `stNXM` using a 30-minute Time-Weighted Average Price (TWAP) from a Uniswap V3 pool. However, newly created Uniswap V3 pools are initialized with an `observationCardinality` of 1 by default, which only stores the current block's state.

Because the Oracle does not expand this cardinality during initialization, calls to `OracleLibrary.consult` requesting data from 1800 seconds (30 minutes) ago will inevitably fail. This causes the price() function to revert consistently, rendering the oracle unusable and bricking any dependent systems (like Morpho markets) immediately upon deployment.

Root Cause

In `stNxmOracle.sol`, the constructor sets the `dex` address but fails to call `increaseObservationCardinalityNext` on the Uniswap V3 pool.

By default, Uniswap V3 pools have an `observationCardinality` of 1. To successfully query a TWAP over a window of `TWAP_PERIOD` (1800 seconds), the pool must be configured to store enough historical observation slots to cover that duration given the chain's block time. Without this initialization, `OracleLibrary.consult` reverts when trying to interpolate a past timestamp.

Internal Pre-conditions

1. `StNxmOracle` is deployed with a reference to a Uniswap V3 pool.
2. The `StNxmOracle` attempts to query a TWAP period of 1800 seconds (`TWAP_PERIOD`).

External Pre-conditions

1. The linked Uniswap V3 pool has the default `observationCardinality` of 1 (or any value insufficient to cover 30 minutes of history).

Attack Path

1. **Deployment:** The protocol deploys `StNxmOracle`.
2. **Interaction:** A user or the Morpho protocol calls `stNxmOracle.price()` to value collateral.
3. **Execution:**
    - The function calls `OracleLibrary.consult(dex, 1800)`.
    - The library attempts to fetch the observation from 1800 seconds ago.
    - The Uniswap pool checks its observation buffer. Since cardinality is 1, it cannot provide history.
4. **Failure:** The transaction reverts.
5. **DoS:** The Oracle remains unusable until an external party manually calls `increaseObservationCardinalityNext` on the pool and waits for the buffer to fill.

Impact

The protocol cannot execute Oracle updates. This results in an immediate Denial of Service (DoS) for the lending integration. Morpho cannot retrieve a valid price, preventing users from borrowing and preventing liquidators from securing the protocol.

PoC

No response

Mitigation

In the `StNxmOracle` constructor, verify the pool's cardinality and increase it if necessary to support the 30-minute TWAP. A safe buffer is recommended (e.g., 100 slots).

```
    constructor(address _dex, address _wNxm, address _stNxm) {
        dex = IUniswapV3Pool(_dex);
        wNxm = _wNxm;
        stNxm = _stNxm;
        startTime = block.timestamp;

+       // Increase cardinality to ensure enough history for 30m TWAP
```

```
+       // 100 slots is usually sufficient for 30 mins on Mainnet (12s blocks)
+       // But safer to go higher or calculate exact needs.
+       dex.increaseObservationCardinalityNext(100);
    }
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/EaseDeFi/stNXM-Contracts/commit/98c005807c5f74d61de7280d572c4aac7ba2f8cb

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.