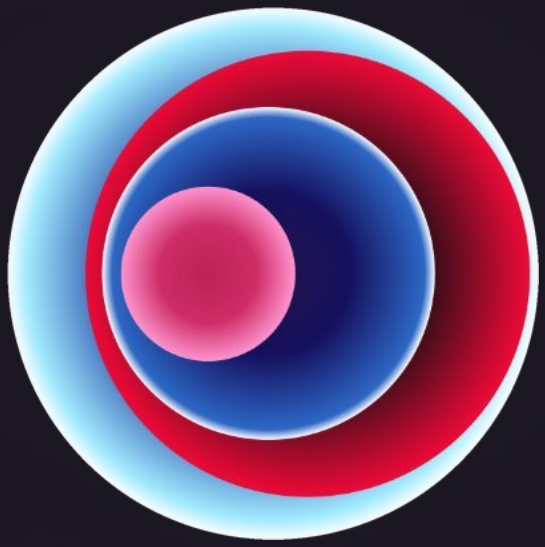
SHERLOCK

Security Review For Mellow Protocol



Public Best Efforts Audit Contest Prepared For:	**Mellow Protocol**
Lead Security Expert:	**bughuntoor**
Date Audited:	**July 14 - July 28, 2025**
Final Commit:	**72f689f**

Introduction

Mellow is modular vault infrastructure built for institutional-grade asset management on EVM chains.

Scope

Repository: mellow-finance/flexible-vaults

Audited Commit: c2d66f36333d9f29457399fa54ddc68079d7b0a9

Final Commit: 72f689f965e4ac1a4c2bcfb645a8b5416cf740c7

Files:

- src/factories/Factory.sol
- src/hooks/BasicRedeemHook.sol
- src/hooks/LidoDepositHook.sol
- src/hooks/RedirectingDepositHook.sol
- src/libraries/FenwickTreeLibrary.sol
- src/libraries/ShareManagerFlagLibrary.sol
- src/libraries/SlotLibrary.sol
- src/libraries/TransferLibrary.sol
- src/managers/BasicShareManager.sol
- src/managers/FeeManager.sol
- src/managers/RiskManager.sol
- src/managers/ShareManager.sol
- src/managers/TokenizedShareManager.sol
- src/modules/ACLModule.sol
- src/modules/BaseModule.sol
- src/modules/CallModule.sol
- src/modules/ShareModule.sol
- src/modules/SubvaultModule.sol
- src/modules/VaultModule.sol
- src/modules/VerifierModule.sol
- src/oracles/Oracle.sol
- src/permissions/BitmaskVerifier.sol

- src/permissions/Consensus.sol
- src/permissions/MellowACL.sol
- src/permissions/protocols/EigenLayerVerifier.sol
- src/permissions/protocols/ERC20Verifier.sol
- src/permissions/protocols/OwnedCustomVerifier.sol
- src/permissions/protocols/SymbioticVerifier.sol
- src/permissions/Verifier.sol
- src/queues/DepositQueue.sol
- src/queues/Queue.sol
- src/queues/RedeemQueue.sol
- src/queues/SignatureDepositQueue.sol
- src/queues/SignatureQueue.sol
- src/queues/SignatureRedeemQueue.sol
- src/vaults/Subvault.sol
- src/vaults/VaultConfigurator.sol
- src/vaults/Vault.sol

Final Commit Hash

72f689f965e4ac1a4c2bcfb645a8b5416cf740c7

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
6	9

Issues Not Fixed and Not Acknowledged

High	Medium
0	0

Security experts who found valid issues

0rpse
0x23r0
0xDLG
0xRaz
0xShoonya
0xapple
0xc0ffEE
0xfocusNode
0xloophole
0xpetern
7
8olidity
ARMoh
Arav
Brene
Caramelo10
Cybrid
DeveloperX
Etherking
Greed
GypsyKing18
HeckerTrieuTien
Ivcho332
JeRRy0422
Kirkeelee
Kodyvim
KupiaSec
MaCree
Mishkat6451
Pandabear30
Pexy
PratRed
Sabit97
SilentVoice-dev
Smacaud
Sparrow_Jac
TopStar
ZanyBonzy
akhoronko
algiz
aman
anchabadze
auditgpt
axelot
bXiv
bam0x7
blockace
boredpukar
bourdillion
bughuntoor
ccvascocc
ch13fd357r0y3r
coin2own
d4r3_w0lf
dan__vinci
davies0212
dimulski
edger
elyas
evta
freeking
holtzzx
hunt1
iFindTheBugs2
illoy_sci
jah
jasonxiale
joicygiore
jolyon
kangaroo
katz
kazan
ke1caM
khaye26
klaus
komane007
lazyrams352
lodelux
maigadoh
makeWeb3safe
natachi
odessos42
pollersan
rbd3
reedai
rsam_eth
sakibcy
sheep
silver_eth
slavina
snapishere
t.aksoy
taticuvostru
tedox
teoslaf1
theboiledcorn
v1c7
weblogicctf
who_is_rp
wickie
x0rc1ph3r
zxriptor

Issue H-1: `Consensus.checkSignatures` doesn't check duplication of signers

Source: https://github.com/sherlock-audit/2025-07-mellow-flexible-vaults-judging/issues/13

Found by

0xShoonya, 0xc0ffEE, 7, 8olidity, ARMoh, Brene, Caramelo10, Cybrid, Etherking, GypsyKing18, Ivcho332, Kodyvim, MaCree, Mishkat6451, Pandabear30, Pexy, PratRed, Sparrow_Jac, ZanyBonzy, akhoronko, aman, anchabadze, blockace, boredpukar, bughuntoor, ch13fd357r0y3r, coin2own, edger, iFindTheBugs2, illoy_sci, jasonxiale, joicygiore, jolyon, kangaroo, kazan, ke1caM, klaus, makeWeb3safe, rsam_eth, silver_eth, t.aksoy, teoslaf1, weblogicctf, zxriptor

Summary

SignatureQueue.validateOrder checks signatures using Consensus.checkSignatures. But this doesn't check duplication of signers, so malicious attacker can bypass `threshold` checking.

Root Cause

SignatureDepositQueue.deposit and SignatureRedeemQueue.redeem uses SignatureQueue.validateOrder. This checks signatures using Consensus.checkSignatures.

```
    function checkSignatures(bytes32 orderHash, Signature[] calldata signatures)
    ↪  public view returns (bool) {
        ConsensusStorage storage $ = _consensusStorage();
@>      if (signatures.length == 0 || signatures.length < $.threshold) {
            return false;
        }
        for (uint256 i = 0; i < signatures.length; i++) {
            address signer = signatures[i].signer;
            (bool exists, uint256 signatureTypeValue) = $.signers.tryGet(signer);
            if (!exists) {
                return false;
            }
            SignatureType signatureType = SignatureType(signatureTypeValue);
            if (signatureType == SignatureType.EIP712) {
                address recoveredSigner = ECDSA.recover(orderHash,
                ↪  signatures[i].signature);
                if (recoveredSigner == address(0) || recoveredSigner != signer) {
                    return false;
                }
```

```
            } else if (signatureType == SignatureType.EIP1271) {
                bytes4 magicValue = IERC1271(signer).isValidSignature(orderHash,
                ↪  signatures[i].signature);
                if (magicValue != IERC1271.isValidSignature.selector) {
                    return false;
                }
            } else {
                return false;
            }
        }
        return true;
    }
```

This checks `signatures.length < $.threshold` but this doesn't check duplication of signers. So attacker can bypass threshold and deposit or redeem using invalid order.

Internal Pre-conditions

.

External Pre-conditions

.

Attack Path

1. Attacker prepares incorrect order.
2. Attacker gets one valid signature.
3. Attacker uses `SignatureDepositQueue` or `SignatureRedeemQueue` using `signatures` that has duplication of one valid signature.
4. Attacker succeed to order.

Impact

Attacker can get profit using incorrect order bypassing threshold.

PoC

.

Mitigation

Check duplication of signers.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/mellow-finance/flexible-vaults/pull/5/files

Issue H-2: RedeemQueue Accounting Mismatch Between Batch Creation and Claim Eligibility

Source: https://github.com/sherlock-audit/2025-07-mellow-flexible-vaults-judging/issues/65

Found by

7, Arav, Brene, Cybrid, JeRRy0422, KupiaSec, Smacaud, algiz, blockace, bughuntoor, dimulski, huntl, lodelux, reedai, silver_eth, t.aksoy

Summary

The RedeemQueue contract contains a critical vulnerability where _handleReport processes fewer redemption requests than what users can claim, leading to accounting inconsistencies and potential loss of funds. The issue occurs due to a mismatch between the timestamp boundary logic in batch creation versus claim eligibility checks.

Root Cause

The vulnerability stems from inconsistent timestamp boundary handling between two functions:

_handleReport: Uses upperLookupRecent(timestamp) followed by latestEligibleIndex--, which excludes the last redemption request before the report timestamp

claim: Uses latestEligibleTimestamp from prices, allowing claims for any timestamp report timestamp

This creates a scenario where users can claim from batches that don't contain their shares, causing accounting mismatches.

```
    function claim(address receiver, uint32[] calldata timestamps) external
    ↪  nonReentrant returns (uint256 assets) {
...
          //@audit users can claim up to higher price saved
          (bool hasRequest, uint256 shares) = callerRequests.tryGet(timestamp);
           if (!hasRequest) {
               continue;
           }
           if (shares != 0) {
               uint256 index = $.prices.lowerLookup(timestamp);
               if (index >= batchIterator) {
                   continue;
               }
```

```
...
}

    function _handleReport(uint224 priceD18, uint32 timestamp) internal override {
        RedeemQueueStorage storage $ = _redeemQueueStorage();

        Checkpoints.Trace224 storage timestamps = _timestamps();
        (, uint32 latestTimestamp, uint224 latestIndex) =
        ↪  timestamps.latestCheckpoint();
        uint256 latestEligibleIndex;
        if (latestTimestamp <= timestamp) {
            latestEligibleIndex = latestIndex;
        } else {
            latestEligibleIndex = uint256(timestamps.upperLookupRecent(timestamp));
            if (latestEligibleIndex == 0) {
                return;
            }
            latestEligibleIndex--;
        }
}...
```

https://github.com/sherlock-audit/2025-07-mellow-flexible-vaults/blob/eca8836d68d65bcbfc52c6f04cf6b4b1597555bf/flexible-vaults/src/queues/RedeemQueue.sol#L219

Internal Pre-conditions

Multiple users must submit redemption requests at different timestamps _handleReport must be called with a timestamp that falls between redemption request timestamps The report timestamp must be greater than some redemption timestamps but not necessarily after all pending requests

External Pre-conditions

-

Attack Path

1. Users submit redemption requests:
   - User1: timestamp 100 (10,000 shares)
   - User2: timestamp 110 (10,000 shares)
   - User3: timestamp 120 (10,000 shares)
   - User4: timestamp 130 (10,000 shares)
2. handleReport is called with timestamp 125

3. Batch Creation:
   - upperLookupRecent(125) returns index pointing to timestamp 120
   - latestEligibleIndex-- excludes timestamp 120
   - Batch 0 created with only 20,000 shares (from timestamps 100, 110)
   - But claim eligibility allows timestamps □ 125 (including 100, 110, 120)
4. Claim:
   - User3 (timestamp 120) can claim because 120 □ 125 (latestEligibleTimestamp)
   - User3 claims 10,000 shares from Batch 0 that doesn't include their contribution
   - Batch 0 now has only 10,000 shares remaining instead of 20,000
   - User1 and User2 can only claim partial amounts, with one user unable to claim at all
   - If User3 share ratio is higher more users wouldn't be able to claim
5. Permanent Fund Lockup:
   - Next _handleReport call processes User3's 120 timestamp shares into Batch 1
   - User3 cannot claim from Batch 1
   - User3's 10,000 shares in Batch 1 become permanently unclaimable

Impact

Batch accounting would be broken Legitimate users cannot claim their full entitlement due to depleted batch Shares processed in subsequent batches become unclaimable

PoC

// File: flexible-vaults/test/unit/queues/DepositQueue.t.sol

```
// forge test --fork-url $(grep ETH_RPC .env | cut -d '=' -f2,3,4,5) --gas-limit
↪  10000000000000000 --fork-block-number 22730425 -vvv --mt testRedeemMulti
function testRedeemMulti() external {
    Deployment memory deployment = createVault(
        vaultAdmin,
        vaultProxyAdmin,
        assetsDefault
    );
    DepositQueue queue = DepositQueue(
        addDepositQueue(deployment, vaultProxyAdmin, asset)
    );

    IOracle.SecurityParams memory securityParams = IOracle.SecurityParams({
        maxAbsoluteDeviation: 6e16,
```

```
        suspiciousAbsoluteDeviation: 2e16,
        maxRelativeDeviationD18: 4e16,
        suspiciousRelativeDeviationD18: 3e16,
        timeout: 1000,
        depositInterval: 3600,
        redeemInterval: 3600
    });

    vm.prank(deployment.vaultAdmin);
    deployment.oracle.setSecurityParams(securityParams);

    pushReport(
        deployment.oracle,
        IOracle.Report({asset: asset, priceD18: 1e18})
    );

    vm.warp(block.timestamp + securityParams.timeout);

    address user1 = vm.createWallet("user1").addr;
    address user2 = vm.createWallet("user2").addr;
    address user3 = vm.createWallet("user3").addr;
    address user4 = vm.createWallet("user4").addr;
    address user5 = vm.createWallet("user5").addr;
    uint256 amount = 1 ether;

    {
        makeDeposit(user1, amount, queue);

        vm.warp(block.timestamp + securityParams.timeout);
        pushReport(
            deployment.oracle,
            IOracle.Report({asset: asset, priceD18: 1e18})
        );

        makeDeposit(user2, amount, queue);

        vm.warp(block.timestamp + securityParams.timeout);
        pushReport(
            deployment.oracle,
            IOracle.Report({asset: asset, priceD18: 1e18})
        );

        makeDeposit(user3, amount, queue);
        makeDeposit(user4, amount, queue);
        makeDeposit(user5, amount, queue);
        emit log_int(deployment.riskManager.pendingAssets(asset));

        vm.warp(block.timestamp + securityParams.depositInterval);
        pushReport(
            deployment.oracle,
```

```
            IOracle.Report({asset: asset, priceD18: 1e18})
        );
    }

    vm.warp(block.timestamp + securityParams.depositInterval);

    queue.claim(user1);
    queue.claim(user2);
    queue.claim(user3);
    queue.claim(user4);
    queue.claim(user5);

    //@audit setup RedeemQueue
    vm.startPrank(deployment.vaultAdmin);
    deployment.vault.setQueueLimit(deployment.vault.queueLimit() + 1);
    deployment.vault.createQueue(
        0,
        false,
        vaultProxyAdmin,
        asset,
        new bytes(0)
    );
    vm.stopPrank();

    RedeemQueue redeemQue = RedeemQueue(
        payable(
            deployment.vault.queueAt(
                asset,
                deployment.vault.getQueueCount(asset) - 1
            )
        )
    );

    vm.prank(deployment.feeManager.owner());
    deployment.feeManager.setFees(0, 0, 0, 0);

    //@audit start redemptions
    uint256 redeemStart = block.timestamp;
    uint256 redeemAmount = 10000000;

    makeRedeem(user1, redeemAmount, redeemQue);

    skip(100);
    makeRedeem(user2, redeemAmount, redeemQue);

    skip(100);
    makeRedeem(user3, redeemAmount, redeemQue);

    //@audit make report after second user redeem time
    vm.startPrank(address(deployment.vault));
```

```
    redeemQue.handleReport(uint224(1e18), uint32(redeemStart + 150));
    vm.stopPrank();

    (, uint256 batchShares) = redeemQue.batchAt(0);
    //@audit only first user is processes
    assertEq(batchShares, redeemAmount);

    redeemQue.handleBatches(2);
    assertEq(ERC20(asset).balanceOf(address(redeemQue)), redeemAmount);

    uint32[] memory timestamps = new uint32[](1);

    timestamps[0] = uint32(redeemStart + 100);
    vm.prank(user2);
    //@audit second user can claim even thoough only first user shares are processed
    redeemQue.claim(user2, timestamps);
    assertEq(ERC20(asset).balanceOf(address(user2)), redeemAmount);

    timestamps[0] = uint32(redeemStart);
    vm.prank(user1);
    //@audit Fails here ::[FAIL: panic: division or modulo by zero (0x12)]
    //@audit first user cant claim since user 2 removed all shres
    vm.expectRevert();
    redeemQue.claim(user1, timestamps);
    assertEq(ERC20(asset).balanceOf(address(user1)), 0);
}
```

Mitigation

Dont remove the last user from the batch processing

```
        // latestEligibleIndex--; // REMOVE THIS LINE
```

Discussion

sherlock-admin2

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

Issue H-3: Unable to withdraw native tokens because vault and redeem hooks do not handle native tokens

Source: https://github.com/sherlock-audit/2025-07-mellow-flexible-vaults-judging/issues/147

Found by

0x23r0, 0xc0ffEE, 7, Cybrid, Greed, dan__vinci, edger, huntl, katz, ke1caM, khaye26, klaus, t.aksoy, theboiledcorn, weblogicctf

Summary

The vault and redeem hook do not handle native token, preventing them from processing native token withdrawal requests. Users cannot withdraw native tokens.

Root Cause

The vault is generally implemented to support native tokens, and native tokens are included in the supported token list. However, when querying token balances during withdrawal, it does not consider the case where the asset token is a native token. This causes withdrawals to fail.

1. RedeemQueue and SignatureRedeemQueue call `vault.getLiquidAssets` to query the token balance for processing withdrawals. If no hook is used, it calls ERC20's `balanceOf`. If the asset is a native token, this function call fails and the transaction is reverted. modules/ShareModule.sol#L150

```
function getLiquidAssets() public view returns (uint256) {
    ...
    address hook = getHook(queue);
@>  return hook == address(0) ? IERC20(asset).balanceOf(address(this)) :
↪   IRedeemHook(hook).getLiquidAssets(asset);
}
```

2. If a hook is used, it calls `Hook.getLiquidAssets`. Currently, only BasicRedeemHook exists as a redeem hook. The `BasicRedeemHook.getLiquidAssets` function fails if the asset is a native token because `balanceOf` fails and gets reverted. hooks/BasicRedeemHook.sol#L35-L39

```
function getLiquidAssets(address asset) public view virtual returns (uint256
↪   assets) {
    IVaultModule vault = IVaultModule(msg.sender);
@>  assets = IERC20(asset).balanceOf(address(vault));
    uint256 subvaults = vault.subvaults();
```

```
    for (uint256 i = 0; i < subvaults; i++) {
        address subvault = vault.subvaultAt(i);
@>      assets += IERC20(asset).balanceOf(subvault);
    }
}
```

3. RedeemQueue and SignatureRedeemQueue call `vault.callHook` to process withdrawals. This function calls `Hook.callHook` via delegatecall if a redeem hook is registered. Currently, only BasicRedeemHook exists as a redeem hook. `BasicRedeemHook.callHook` fails if the asset is a native token because `balanceOf` fails and gets reverted. hooks/BasicRedeemHook.sol#L11-L19

```
function callHook(address asset, uint256 assets) public virtual {
    IVaultModule vault = IVaultModule(address(this));
@>  uint256 liquid = IERC20(asset).balanceOf(address(vault));
    ...
    for (uint256 i = 0; i < subvaults; i++) {
        address subvault = vault.subvaultAt(i);
@>      uint256 balance = IERC20(asset).balanceOf(subvault);
        ...
    }
}
```

Internal Pre-conditions

1. Using native token as asset token

External Pre-conditions

None

Attack Path

1. Attempt to withdraw after depositing native token

Impact

Cannot withdraw native token. Users cannot withdraw asset token.

PoC

The following PoC shows that `vault.getLiquidAssets` fails. Add to `flexible-vaults/test/unit/modules/ShareModule.t.sol` and run.

```
function test_poc_GetLiquidAssets() external {
    address asset = address(0xEeeeeEeeeEeEeeEeEeEeeEEEeeeeEeeeeeeeEEeE); // native
    assets.push(address(asset));

    Deployment memory deployment = createVault(vaultAdmin, vaultProxyAdmin, assets);

    vm.startPrank(vaultAdmin);
    deployment.vault.setQueueLimit(2);
    deployment.vault.createQueue(0, true, vaultProxyAdmin, address(asset), new
    ↪  bytes(0));
    deployment.vault.createQueue(0, false, vaultProxyAdmin, address(asset), new
    ↪  bytes(0));
    assertEq(deployment.vault.getQueueCount(), 2, "Queue count should be 2");
    vm.stopPrank();

    address depositQueue = deployment.vault.queueAt(address(asset), 0);
    address redeemQueue = deployment.vault.queueAt(address(asset), 1);

    vm.prank(redeemQueue);
    vm.expectRevert();
    deployment.vault.getLiquidAssets();
}
```

Mitigation

When the asset token is a native token, query the native token balance instead of using `ERC20.balanceOf`.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/mellow-finance/flexible-vaults/pull/10

Issue H-4: Protocol Fee Multiple Accrual in Oracle.submitReports

Source: https://github.com/sherlock-audit/2025-07-mellow-flexible-vaults-judging/issues/167

Found by

0x23r0, 0xc0ffEE, 7, Etherking, ZanyBonzy, blockace, evta, lodelux, silver_eth

Summary

The `Oracle.submitReports` function processes multiple asset reports in a single transaction, but the `ShareModule.handleReport` function calculates and accrues protocol fees for each report individually without considering that the timestamp should only be updated once per batch. This leads to multiple fee accruals when non-base assets are processed before the base asset, resulting in excessive protocol fees being charged.

Root Cause

The issue stems from the interaction between Oracle.submitReports and `ShareModule.handleReport`. In `ShareModule.handleReport`, the fee calculation uses the current timestamp and the last stored timestamp to calculate protocol fees:

```
// In ShareModule.handleReport (lines 281-283)
uint256 fees = feeManager_.calculateFee(address(this), asset, priceD18,
↪  shareManager_.totalShares());
if (fees != 0) {
    shareManager_.mint(feeManager_.feeRecipient(), fees);
}
```

The `FeeManager.calculateFee` function calculates protocol fees based on the time difference:

```
// In FeeManager.calculateFee (lines 81-83)
uint256 timestamp = $.timestamps[vault];
if (timestamp != 0 && block.timestamp > timestamp) {
    shares += Math.mulDiv(totalShares, $.protocolFeeD6 * (block.timestamp -
    ↪  timestamp), 365e6 days);
}
```

However, `FeeManager.updateState` only updates the timestamp when the asset is the base asset:

```
// In FeeManager.updateState (lines 116-128)
function updateState(address asset, uint256 priceD18) external {
    FeeManagerStorage storage $ = _feeManagerStorage();
    address vault = _msgSender();
    if ($.baseAsset[vault] != asset) {
        return; // Early return for non-base assets
    }
    // ... update minPriceD18 and timestamp
    $.timestamps[vault] = block.timestamp;
}
```

When multiple reports are submitted in `Oracle.submitReports`, if the first reports are for non-base assets, the timestamp remains unchanged, causing each subsequent report to calculate fees based on the same old timestamp, leading to multiple fee accruals.

Internal Pre-conditions

1. The vault must support multiple assets with one designated as the base asset
2. The `Oracle.submitReports` function must be called with multiple reports in a single transaction
3. The first reports in the batch must be for non-base assets
4. The base asset report must appear later in the batch
5. Protocol fees must be configured (protocolFeeD6 > 0)
6. Time must have passed since the last timestamp update (block.timestamp > lastTimestamp)

External Pre-conditions

.

Attack Path

1. **Setup**: Configure a vault with multiple assets where one is designated as the base asset
2. **Initial State**: Set protocol fees and ensure time has passed since the last timestamp update
3. **Submit Reports**: Call `Oracle.submitReports` with multiple reports where non-base assets appear first
4. **First Report Processing**: `ShareModule.handleReport` is called for the first non-base asset

- `FeeManager.calculateFee` calculates protocol fees based on old timestamp
- Fees are minted to the fee recipient
- `FeeManager.updateState` is called but returns early (not base asset)

5. **Subsequent Reports**: For each additional report before the base asset:
   - `FeeManager.calculateFee` uses the same old timestamp
   - Additional fees are calculated and minted
   - `FeeManager.updateState` continues to return early
6. **Base Asset Report**: When the base asset report is processed:
   - Fees are calculated again using the old timestamp
   - `FeeManager.updateState` finally updates the timestamp
7. **Result**: Multiple fee accruals have occurred, with fees being charged multiple times for the same time period

Impact

This vulnerability allows excessive protocol fees to be charged when multiple asset reports are submitted in a single transaction.

PoC

```
function testMultipleTimesFee() external {
    Deployment memory deployment = createVault(vaultAdmin, vaultProxyAdmin,
    ↪  assetsDefault);
    Oracle oracle = deployment.oracle;
    address asset = assetsDefault[0];

    vm.startPrank(vaultAdmin);
    deployment.feeManager.setFeeRecipient(vaultAdmin);
    deployment.feeManager.setFees(0,0,0, 1e5); // 10% protocol fee
    deployment.feeManager.setBaseAsset(address(deployment.vault), asset);
    vm.stopPrank();

    IOracle.Report[] memory reports = new IOracle.Report[](3);
    for (uint256 i = 0; i < 3; i++) {
        reports[i].asset = assetsDefault[2 - i];
        reports[i].priceD18 = 1e18;
    }

    vm.startPrank(vaultAdmin);
    oracle.submitReports(reports);
    for (uint256 i = 0; i < 3; i++) {
```

```
        oracle.acceptReport(reports[i].asset, reports[i].priceD18,
        ↪  uint32(block.timestamp));
    }
    vm.stopPrank();

    // Set initial shares to 1000
    vm.prank(address(deployment.vault));
    deployment.shareManager.mint(address(0x100), 1000 ether);

    // Move time forward by 1 year
    skip(365 days);

    // Submit reports for 3 assets
    vm.prank(vaultAdmin);
    oracle.submitReports(reports);

    // Check the fee recipient's shares
    uint256 feeRecipientShares = deployment.shareManager.sharesOf(vaultAdmin);
    console.log("Fee recipient shares:", feeRecipientShares);

    assertGt(feeRecipientShares, 300 ether, "Fee recipient shares mismatch");
}
```

Mitigation

The issue can be mitigated by updating the implementation related to the timestamp update. I think the timestamp should be updated even though the asset is not the base asset.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/mellow-finance/flexible-vaults/pull/6

Issue H-5: Incorrect performance fee calculation in `FeeManager`

Source: https://github.com/sherlock-audit/2025-07-mellow-flexible-vaults-judging/issues/207

Found by

0xapple, blockace, dan__vinci, davies0212, lodelux

Summary

The performance fee calculation in `FeeManager.calculateFee` uses an incorrect formula that doesn't properly convert price differences to share amounts. The current implementation leads to incorrect fee calculations.

Root Cause

The root cause is in the performance fee calculation formula in FeeManger.calculateFee function :

```
shares = Math.mulDiv(minPriceD18_ - priceD18, $.performanceFeeD6 * totalShares,
↪  1e24);
```

This formula incorrectly assumes that the price difference (`minPriceD18_ - priceD18`) can be directly multiplied by the total shares to calculate fee shares.

Following formula comparison will be help to understand issue:

- definition of price: `share = price * asset`
- current formula: 'feeShare = priceDifference * totalShare

Example1: minPriceD18=100e18, priceD18=90e18, performanceFee=1e5(10%) feeShare = 10e18 * 1e5 * totalShare / 1e24 = totalShare This means that performance fee share is the same as total share. -> Incorrect result.

Example2: Let's imagine that asset is USDC with decimal 6, and vault share price is $1. In this case, `priceD18` for `USDC` is `1e30`, not `1e18` to ensure share decimal as 18.

Internal Pre-conditions

.

External Pre-conditions

.

Attack Path

.

Impact

Incorrect performance fees calculation.

PoC

.

Mitigation

The formula should be corrected to properly convert the price difference to share amounts by including `priceD18` in the denominator:

```
shares = Math.mulDiv(minPriceD18_ - priceD18, $.performanceFeeD6 * totalShares, 1e6
↪  * priceD18);
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/mellow-finance/flexible-vaults/pull/21

Issue H-6: Redeems through RedeemQueue avoid paying management and performance fee.

Source: https://github.com/sherlock-audit/2025-07-mellow-flexible-vaults-judging/issues/491

Found by

bughuntoor, t.aksoy, tedox

Summary

When a user calls `redeem`, the amount of shares they want to redeem is immediately burned, effectively removing the shares from `activeShares`

```
IShareManager shareManager_ = IShareManager(IShareModule(vault_).shareManager());
shareManager_.burn(caller, shares);
{
    IFeeManager feeManager = IShareModule(vault_).feeManager();
    uint256 fees = feeManager.calculateRedeemFee(shares);
    if (fees > 0) {
        shareManager_.mint(feeManager.feeRecipient(), fees);
        shares -= fees;
    }
}
```

However, although the shares are burned, the funds actually remain within the vault up until a report is handled and it handles said redeems. For these reasons these funds should also be subject to performance and management fees (both for the time period between last fee accrual and creating redeem request and also after creating redeem request and up to report handling.

As the code does considers for `totalShares` only the `allocated` and minted shares, but not the ones waiting in redeem queue, the shares in redeem queue effectively bypass the necessary fees.

Root Cause

Flawed logic.

Code

https://github.com/sherlock-audit/2025-07-mellow-flexible-vaults/blob/main/flexible-vaults/src/queues/RedeemQueue.sol#L98

Attack Path

1. Reports are handled once a day.
2. Last report is at 00:00.
3. User creates a redeem request at 12:00. (12 hours since last report)
4. A new report comes at 00:00 (24 hours since previous report).
5. The user claims their redeem and does not pay any fees for the past day.

Impact

Loss of vault fees.

PoC

No response

Mitigation

When user creates a redeem request, transfer the fees to the queue contract. Burn the shares only when the request is handled (within `_handleReport`).

Discussion

sherlock-admin2

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

Issue M-1: Flawed Logic in `ShareManager` Inverts Transfer Whitelist Behavior

Source: https://github.com/sherlock-audit/2025-07-mellow-flexible-vaults-judging/issues/26

Found by

0rpse, 0x23r0, 0xDLG, 0xShoonya, 0xc0ffEE, 0xloophole, 0xpetern, 7, ARMoh, Arav, Brene, Cybrid, DeveloperX, Etherking, Greed, GypsyKing18, HeckerTrieuTien, KupiaSec, Mishkat6451, PratRed, SilentVoice-dev, Sparrow_Jac, TopStar, ZanyBonzy, algiz, axelot, bXiv, bam0x7, blockace, boredpukar, bughuntoor, coin2own, dan__vinci, davies0212, dimulski, edger, elyas, holtzzx, illoy_sci, jah, joicygiore, jolyon, ke1caM, klaus, lazyrams352, lodelux, maigadoh, natachi, odessos42, pollersan, rbd3, reedai, silver_eth, slavina, snapishere, t.aksoy, taticuvostru, teoslaf1, v1c7, who_is_rp, wickie, x0rc1ph3r, zxriptor

Summary

The `updateChecks` function in `ShareManager.sol:L139` contains flawed logic that inverts the intended behavior of the transfer whitelist feature. Instead of allowing transfers from whitelisted accounts, the code incorrectly blocks them, rendering the security feature unusable and creating behavior that is opposite to the documentation.

Vulnerability Detail

The `updateChecks` function is responsible for enforcing security policies before any share modification. When the `hasTransferWhitelist` flag is enabled, it is supposed to check if a transfer is permitted. The logic for this check is as follows:

https://github.com/sherlock-audit/2025-07-mellow-flexible-vaults/blob/main/flexible-vaults/src/managers/ShareManager.sol#L139

Here, `info` represents the sender's account data. The code reverts if the sender's `canTransfer` flag is `true`. This is a direct contradiction of the feature's documented intent in the `IShareManager.sol` interface:

```
/// @notice Whether the account is allowed to transfer (send or receive) shares
↪  when the `hasTransferWhitelist` flag is active.
bool canTransfer;
```

The documentation clearly states that `canTransfer: true` should **allow** an account to send or receive shares. The current implementation enforces the opposite, blocking whitelisted senders. Furthermore, the project's test suite (`ShareManager.t.sol`) confirms this flawed logic by explicitly expecting a revert when a whitelisted address attempts to send shares.

#PoC

```
function testUpdateChecks_TransferWhitelist_Inclusive() external {
        // --- Setup ---
        Deployment memory deployment = createVault(vaultAdmin, vaultProxyAdmin,
        ↪  assetsDefault);
        ShareManager manager = deployment.shareManager;
        address from = vm.createWallet("from").addr;
        address to = vm.createWallet("to").addr;

        vm.startPrank(deployment.vaultAdmin);

        // Enable the transfer whitelist
        manager.setFlags(
            IShareManager.Flags({
                hasMintPause: false,
                hasBurnPause: false,
                hasWhitelist: false,
                hasTransferPause: false,
                hasTransferWhitelist: true, // <-- Feature is ON
                globalLockup: 0, // No lockup
                targetedLockup: 0 // No lockup
            })
        );

        // Case 1: Neither is whitelisted. SHOULD REVERT. This is correct behavior.
        manager.setAccountInfo(from, IShareManager.AccountInfo({canDeposit: true,
        ↪  canTransfer: false, isBlacklisted: false, lockedUntil: 0}));
        manager.setAccountInfo(to, IShareManager.AccountInfo({canDeposit: true,
        ↪  canTransfer: false, isBlacklisted: false, lockedUntil: 0}));
        vm.expectRevert(abi.encodeWithSelector(IShareManager.TransferNotAllowed.se
        ↪  lector, from, to));
        manager.updateChecks(from, to);

        // Case 2: Sender is whitelisted, Receiver is not. SHOULD PASS, BUT
        ↪  REVERTS. This proves the bug.
        manager.setAccountInfo(from, IShareManager.AccountInfo({canDeposit: true,
        ↪  canTransfer: true, isBlacklisted: false, lockedUntil: 0}));
        manager.setAccountInfo(to, IShareManager.AccountInfo({canDeposit: true,
        ↪  canTransfer: false, isBlacklisted: false, lockedUntil: 0}));
        vm.expectRevert(abi.encodeWithSelector(IShareManager.TransferNotAllowed.se
        ↪  lector, from, to));
        manager.updateChecks(from, to);

        // Case 3: Sender is not, Receiver is whitelisted. SHOULD PASS. This case
        ↪  correctly passes.
        manager.setAccountInfo(from, IShareManager.AccountInfo({canDeposit: true,
        ↪  canTransfer: false, isBlacklisted: false, lockedUntil: 0}));
        manager.setAccountInfo(to, IShareManager.AccountInfo({canDeposit: true,
        ↪  canTransfer: true, isBlacklisted: false, lockedUntil: 0}));
```

```
        manager.updateChecks(from, to); // No expectRevert, this succeeds as
        ↪  intended.

        // Case 4: Both are whitelisted. SHOULD PASS, BUT REVERTS. This also
        ↪  proves the bug.
        manager.setAccountInfo(from, IShareManager.AccountInfo({canDeposit: true,
        ↪  canTransfer: true, isBlacklisted: false, lockedUntil: 0}));
        manager.setAccountInfo(to, IShareManager.AccountInfo({canDeposit: true,
        ↪  canTransfer: true, isBlacklisted: false, lockedUntil: 0}));
        vm.expectRevert(abi.encodeWithSelector(IShareManager.TransferNotAllowed.se
        ↪  lector, from, to));
        manager.updateChecks(from, to);

        vm.stopPrank();
    }
```

Impact

The primary impact is the complete failure of the transfer whitelist feature. A vault operator who enables this feature and whitelists accounts by setting `canTransfer` to `true` will find that these accounts are unexpectedly blocked from making transfers. This subverts the intended security model and can disrupt the vault's operations. Because the code behaves in the exact opposite manner of its documentation, this is an issue that can lead to confusion and operational failure.

Recommendation

To fix the inverted logic, the `updateChecks` function should only revert if **both** the sender and receiver are **not** whitelisted. This aligns the implementation with the documented behavior.

The flawed line:

```
function updateChecks(/*params*/) public view {
    // code
    .
    .
-   if (info.canTransfer || !$.accounts[to].canTransfer) {
+   if (!info.canTransfer && !$.accounts[to].canTransfer) {
    // rest of code
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/mellow-finance/flexible-vaults/pull/3

Issue M-2: ETH redemptions via `SignatureRedeemQueue` are broken due to missing `receive` function

Source: https://github.com/sherlock-audit/2025-07-mellow-flexible-vaults-judging/issues/140

Found by

0xShoonya, 7, HeckerTrieuTien, ZanyBonzy, bourdillion, dan__vinci, edger, ke1caM, klaus

Summary

The `SignatureRedeemQueue` contract lacks a `receive()` function, causing ETH redemptions to fail.

Root Cause

This is the call flow of SignatureRedeemQueue.redeem when users attempt to redeem ETH:

```
SignatureRedeemQueue.redeem(...)
    vault_.callHook(order.requested)
        ShareModule.callHook(...)
            TransferLibrary.sendAssets(asset, queue, assets)
                • if asset == ETH  →  Address.sendValue(queue, assets)
```

ShareModule.callHook() ends with

```
if (!$.isDepositQueue[queue]) {
    TransferLibrary.sendAssets(asset, queue, assets);  // ETH goes here
}
```

If `asset == ETH` the helper executes `Address.sendValue(payable(queue), assets)` – **an empty-data call** to the queue contract.

Neither `SignatureRedeemQueue` nor `SignatureQueue` (base contract) implement either `receive()` or a `payable fallback`, so the ETH transfer would revert

Internal pre-conditions

N/A

External pre-conditions

N/A

Attack Scenario

N/A

Impact

ETH users cannot redeem through Signature queues - the transaction always fails.

Mitigation

Add a `receive()` function once in the shared base contract:

```
// SignatureQueue.sol
receive() external payable {}
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/mellow-finance/flexible-vaults/pull/12

Issue M-3: stETH edge case in transfer rounding can cause denial of service for depositors and redeemers.

Source: https://github.com/sherlock-audit/2025-07-mellow-flexible-vaults-judging/issues/141

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

Found by

0x23r0, 0xShoonya, Kirkeelee, t.aksoy

Summary

The use of stETH, which can transfer 1-2 wei less than requested due to its internal share-based accounting and rounding, will cause a denial of service for depositors and redeemers as the contract will revert when attempting to transfer more stETH than it actually holds.

Root Cause

The root cause of the issue lies in how the queue contracts handle the `assets` variable during deposits, cancellations, and batch processing. When a user deposits stETH, the contract assumes that the exact amount specified by the user is received and records this value in its internal accounting. However, due to stETH's internal share-based accounting and rounding, the contract may actually receive 1-2 wei less than the requested amount. This discrepancy is not accounted for in the contract's logic. As a result, when a user later cancels their deposit or when a batch is processed in the `_handleReport` function, the contract attempts to transfer the full recorded assets amount to the user or the vault using `TransferLibrary.sendAssets`. If the contract's actual stETH balance is less than the recorded assets due to the initial shortfall, the transfer will revert. This leads to a denial of service for users attempting to cancel, claim, or process deposits/redeems, as the contract cannot fulfill the transfer with its actual balance. The fundamental problem is the assumption that the contract always receives and holds exactly the amount of stETH specified in user actions, without verifying or adjusting for the real amount received.

https://github.com/sherlock-audit/2025-07-mellow-flexible-vaults/blob/main/flexible-vaults/src/queues/DepositQueue.sol#L115

Internal Pre-conditions

1. A user deposits or redeems stETH through the queue contracts.

2. The contract receives slightly less stETH than requested due to stETH's transfer rounding.
3. The contract records the full requested amount in its internal accounting.

External Pre-conditions

1. stETH is used as the asset for the queue.
2. stETH's transfer function rounds down and delivers less than the requested amount.

Attack Path

1. User deposits or redeems stETH.
2. The contract receives less stETH than expected.
3. When the user later cancels, claims, or when a batch is processed, the contract attempts to transfer the full recorded amount.
4. The transfer reverts because the contract does not have enough stETH, causing a denial of service for all subsequent users attempting to withdraw or claim.

Impact

The users cannot withdraw, claim, or process their deposits or redemptions, resulting in a denial of service for affected users. No funds are lost, but users are unable to access their assets.

PoC

No response

Mitigation

Track the actual amount of stETH received by checking the contract balance before and after transfers, and use the real received amount for all internal accounting and subsequent transfers.

Issue M-4: Protocol Fee Exponential Compounding in ShareModule.handleReport

Source: https://github.com/sherlock-audit/2025-07-mellow-flexible-vaults-judging/issues/161

Found by

auditgpt, blockace, bughuntoor, tedox

Summary

The protocol fee calculation in `ShareModule.handleReport` function suffers from exponential compounding instead of linear accumulation. When `handleReport` is called frequently (e.g., daily), the protocol fee is applied to the current total shares which includes all previously accrued fees, leading to higher-than-intended fee extraction.

Root Cause

In ShareModule.handleReport at line 281-284:

```
uint256 fees = feeManager_.calculateFee(address(this), asset, priceD18,
↪  shareManager_.totalShares());
if (fees != 0) {
    shareManager_.mint(feeManager_.feeRecipient(), fees);
}
```

The issue is that `shareManager_.totalShares()` includes all previously minted shares, including fees that were already taken. This means each subsequent fee calculation is based on an inflated share count, leading to exponential compounding rather than linear fee accumulation.

The `FeeManager.calculateFee` function calculates fees based on the current total shares:

```
function calculateFee(address vault, address asset, uint256 priceD18, uint256
↪  totalShares) external view returns (uint256) {
    // ... other fee calculations ...
    uint256 protocolFee = (totalShares * protocolFeeRate * timeElapsed) / (365 days
    ↪  * 1e6);
    return protocolFee;
}
```

Internal Pre-conditions

1. Protocol fees are enabled in the FeeManager
2. `handleReport` function is called frequently (e.g., daily or more often)
3. The vault has existing shares that include previously accrued fees
4. The fee recipient is properly configured

External Pre-conditions

1. The vault has active deposits and share minting
2. Regular reporting cycles are established (daily/weekly reports)
3. Protocol fee rates are set to non-zero values

Attack Path

1. **Initial Setup**: A vault is created with a 5% annual protocol fee rate
2. **Daily Reports**: The `handleReport` function is called daily for 365 days
3. **Fee Calculation**: Each day, fees are calculated based on `totalShares()` which includes previous day's fees
4. **Exponential Growth**: The fee base grows each day, leading to compounding effects
5. **Result**: After 365 days, the actual fee taken is 5.11% instead of the intended 5%

Impact

Users pay higher fees than intended, with the difference increasing over time

PoC

```
// test/unit/modules/ShareModule.t.sol
function testExponentialProtocolFeeVulnerability() external {
    Deployment memory deployment = createVault(vaultAdmin, vaultProxyAdmin, assets);

    // Set up a 5% annual protocol fee (5e4 in D6 format)
    vm.prank(vaultAdmin);
    deployment.feeManager.setFees(0, 0, 0, 5e4); // 5% protocol fee

    // Set the base asset for the vault
    vm.prank(vaultAdmin);
    deployment.feeManager.setBaseAsset(address(deployment.vault), address(asset));
```

```
    // Create a deposit queue
    vm.prank(vaultAdmin);
    deployment.vault.setQueueLimit(1);
    vm.prank(vaultAdmin);
    deployment.vault.createQueue(0, true, vaultProxyAdmin, address(asset), new
    ↪  bytes(0));

    // Mint some initial shares to simulate existing deposits
    MockTokenizedShareManager(address(deployment.shareManager)).mintShares(user,
    ↪  1000 ether);

    // Record initial state
    uint256 initialTotalShares = deployment.shareManager.totalShares();
    uint256 initialFeeRecipientShares =
    ↪  deployment.shareManager.sharesOf(vaultAdmin);

    // Start from a base timestamp
    uint256 baseTimestamp = block.timestamp;

    // Call handleReport daily for 365 days
    for (uint256 day = 1; day <= 365; day++) {
        uint256 timestamp = baseTimestamp + day * 1 days;

        // Move time forward by 1 day
        vm.warp(timestamp);

        // Call handleReport as the oracle with the current timestamp
        vm.prank(address(deployment.vault.oracle()));
        deployment.vault.handleReport(address(asset), 1e18, uint32(timestamp - 1
        ↪  hours), uint32(timestamp - 1 hours));
    }

    // Check final state
    uint256 finalTotalShares = deployment.shareManager.totalShares();
    uint256 finalFeeRecipientShares = deployment.shareManager.sharesOf(vaultAdmin);

    // Calculate the total fees taken
    uint256 totalFeesTaken = finalFeeRecipientShares - initialFeeRecipientShares;

    // Calculate what the fees should be with linear compounding (5% annual)
    uint256 expectedFees = (initialTotalShares * 5) / 100; // 5% of initial shares

    console.log("Initial total shares:", initialTotalShares);
    console.log("Final total shares:", finalTotalShares);
    console.log("Initial fee recipient shares:", initialFeeRecipientShares);
    console.log("Final fee recipient shares:", finalFeeRecipientShares);
    console.log("Total fees taken:", totalFeesTaken);
    console.log("Expected fees (5% linear):", expectedFees);
}
```

Mitigation

The recommended fix would be to modify the fee calculation to calculate fess on assets amount based on total assets, and then mint corresponding shares to the fee recipient.

Discussion

sherlock-admin2

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

Issue M-5: cancelDepositRequest() always reverts due to modifying FenwickTree with wrong index

Source: https://github.com/sherlock-audit/2025-07-mellow-flexible-vaults-judging/issues/246

Found by

0rpse, 0x23r0, 0xDLG, 0xRaz, 0xShoonya, 0xc0ffEE, 0xfocusNode, 8olidity, Arav, Cybrid, Pexy, Sabit97, algiz, blockace, boredpukar, ccvascocc, d4r3_w0lf, dimulski, freeking, hunt1, jah, kazan, komane007, lodelux, maigadoh, sheep, tedox, teoslaf1

Summary

In `DepositQueue:cancelDepositRequest()`, the code uses the latest `price` checkpoint instead of `index` when attempting to remove a user's pending deposit from the Fenwick tree.

Because the timestamp-based request array has length < price, the index is out of bounds and the function always reverts, making it impossible for depositors to cancel their queued deposits.

Root Cause

The DepositQueue:cancelDepositRequest() calls:

```
(bool exists, uint32 timestamp, uint256 index) = $.prices.latestCheckpoint();
↪  //@audit index is actually the price
..
$.requests.modify(index, -int256(assets)); //@audit priceD18 instead of index will
↪  result in IndexOutOfBounds error
```

Here, `$.prices` is the oracle-price checkpoint trace, not the timestamp trace that indexes the `$.requests` Fenwick tree. As a result, the retrieved `index` does not corresponds to a valid slot in the `$.requests`, causing `modify()` to revert with IndexOutOfBounds.

Internal Pre-conditions

- At least one price report has been processed
- A user has a pending deposit and the request is still non-claimable

External Pre-conditions

None.

Attack Path

1. User submits `deposit(..)` and their funds are locked and recorded in `$.requestOf` and Fenwick tree
2. User calls `cancelDepositRequest()` , but the function reverts and the user cannot cancel their deposit

Impact

Medium severity

- Users can't cancel their own pending deposits, undermining basic UX and expected behaviour.
- Core cancellation functionality is broken, but funds aren't stolen, just stuck until the queue processes.

PoC

1. Add the following test to the `DepositQueue.t.sol`:

```
function testPoC_CancelDepositRequest_WrongIndex_Reverts() external {
        Deployment memory deployment = createVault(vaultAdmin, vaultProxyAdmin,
        ↪  assetsDefault);
        DepositQueue queue = DepositQueue(addDepositQueue(deployment,
        ↪  vaultProxyAdmin, asset));
        IOracle.SecurityParams memory securityParams =
        ↪  deployment.oracle.securityParams();

        /// @dev push a report to set the initial price
        // Just pushing a price is not enough, since the report can't be accepted.
        ↪  We need to warp the time to set a valid initial price.
        vm.warp(block.timestamp + Math.max(securityParams.timeout,
        ↪  securityParams.redeemInterval));
        pushReport(deployment.oracle, IOracle.Report({asset: asset, priceD18:
        ↪  1e18}));

        address user1 = vm.createWallet("user").addr;
        uint256 amount1 = 5 ether;

        makeDeposit(user1, amount1, queue);
        assertEq(queue.claimableOf(user1), 0, "Claimable amount should be zero
        ↪  before deposit");
```

```
        (, uint256 assets1) = queue.requestOf(user1);
        assertEq(assets1, amount1, "Assets should match the deposited amount");

        vm.prank(user1);
        vm.expectRevert("IndexOutOfBounds()");
        queue.cancelDepositRequest();
    }
```

2. Execute with:

```
forge test --mt testPoC_CancelDepositRequest_WrongIndex_Reverts --fork-url
↪  https://eth-mainnet.g.alchemy.com/public --gas-limit 10000000000000000
↪  --fork-block-number 22730425 -vvv
```

Mitigation

Instead of using the index returned from the prices.latestCheckpoint(), use the index that corresponds to the user's request timestamp. This could be achieved by modifying the cancelDepositRequest():

```
function cancelDepositRequest() external nonReentrant {
    address caller = _msgSender();
    DepositQueueStorage storage $ = _depositQueueStorage();
    Checkpoints.Checkpoint224 memory request = $.requestOf[caller];
    uint256 assets = request._value;
    if (assets == 0) {
        revert NoPendingRequest();
    }
    address asset_ = asset();
-   (bool exists, uint32 timestamp, uint256 index) = $.pric- es.latestCheckpoint();
+   (bool exists, uint32 timestamp, ) = $.prices.lates+ tCheckpoint();

    if (exists && timestamp >= request._key) {
        revert ClaimableRequestExists();
    }

    delete $.requestOf[caller];
    IVaultModule(vault()).riskManager().modifyPendingAssets(asset_,
    ↪  -int256(uint256(assets)));
+ uint256 index = _timestamps().lowerLookup(reque+ st._key);
    $.requests.modify(index, -int256(assets));
    TransferLibrary.sendAssets(asset_, caller, assets);
    emit DepositRequestCanceled(caller, assets, request._key);
}
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/mellow-finance/flexible-vaults/pull/7

Issue M-6: Targeted-lockup bypass: freshly minted shares can be transferred immediately in the same `transfer()` or `transferFrom()` call

Source: https://github.com/sherlock-audit/2025-07-mellow-flexible-vaults-judging/issues/617

Found by

rsam_eth

Summary

`TokenizedShareManager` is designed so that every newly minted share is subject to a per account targeted lock-up (`lockedUntil = now + targetedLockup`). However, because `TokenizedShareManager._update()` calls `claimShares(from)` before executing the actual ERC-20 transfer, and because `mint()` sets `lockedUntil` after the prior lock check, a user who has any allocated (yet-unclaimed) shares can:

1. trigger a mint via `claimShares` in the `_update` function (through a `transfer` or `transferFrom`),
2. have those shares credited to their balance, and
3. pass them to another address in the same transaction,

all without ever being prevented by the lock-up.

Root Cause

https://github.com/sherlock-audit/2025-07-mellow-flexible-vaults/blob/main/flexible-vaults/src/managers/TokenizedShareManager.sol#L46-L55

```
function _update(address from, address to, uint256 value) internal override {
        updateChecks(from, to); //  lock check executes here
        if (from != address(0)) {
            claimShares(from); //  mints & sets new lock, the updateChecks in next
             ↪  _update calls are
            //bypassed since from is address(0) and to is the "from" here
        }
        if (to != address(0)) {
            claimShares(to);
        }
        super._update(from, to, value); //  transfer executes after mint
    }
```

`claimShares(from)` →`mintAllocatedShares()`→ `mint()` → ERC-20 `_mint()`, the new entered `_update(address(0), from, value)` passes the `updateChecks` for `from` because in the `updateChecks` → `from == address(0)` → the `value` parameter (in the first `_update`) is minted to `from` → in the `shareManager::mint` function (3rd step in the call flow) the `lockedUntill` is assigned: https://github.com/sherlock-audit/2025-07-mellow-flexible-vaults/blob/main/flexible-vaults/src/managers/ShareManager.sol#L230-L233

```
if (targetLockup != 0) {
    uint32 lockedUntil = uint32(block.timestamp) + targetLockup;
    $.accounts[account].lockedUntil = lockedUntil;
    emit Mint(account, value, lockedUntil);
}
```

Because the original `updateChecks` has already passed, the transfer that follows moves the just-minted shares out of the locked account.

Internal Pre-conditions

- `TokenizedShareManager` operates with `targetedLockup > 0`.
- from `holds 1` allocated shares in any deposit queue.

External Pre-conditions

Attack Path

1. **State before tx**
   - `lockedUntil` for **Alice** is *in the past* → she can transfer.
   - Alice owns **100 allocated shares** (not yet active).
2. **Alice calls**

   ```
   share.transfer(bob, 100);
   ```

3. `TokenizedShareManager._update(Alice → Bob)`
   1. `updateChecks(from, to)` → passes (lock expired).
   2. `claimShares(from)` is invoked.
4. **Inside** `claimShares(Alice)`
   1. `mintAllocatedShares(Alice, 100)`
   2. `mint()` → `_mintShares()` → **ERC20** `_mint`
      triggers an **inner** `_update(0 → Alice, 100)` (no lock check).

3. After `_mint` returns, `mint` sets

```
accounts[Alice].lockedUntil = block.timestamp + targetedLockup;
```

5. **Return to outer `_update`**
    - `super._update(Alice → Bob, 100)` executes, moving the **freshly minted shares** to **Bob**.
6. **Result**
    - Transaction succeeds.
    - Bob receives 100 liquid shares immediately.
    - Alice's `lockedUntil` is refreshed, but she now holds **zero** shares, so the lock is moot.

Impact

1. **Policy bypass:** the intended cooling-off period on new shares is unenforced, if a user has allocated shares, then can just transfer them right away.
2. **Market risk:** users can instantly sell or transfer what should have been time-locked shares.

PoC

No response

Mitigation

perform a second lock check, ensuring new shares cannot move until next tx.

Discussion

sherlock-admin2

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

Issue M-7: DoS in Redemption Due to Unchecked Asset Support in Subvaults

Source: https://github.com/sherlock-audit/2025-07-mellow-flexible-vaults-judging/issues/688

Found by

7, HeckerTrieuTien, Sparrow_Jac, dan__vinci, jasonxiale, silver_eth

Summary

In Mellow Flexible Vaults, the `getLiquidAssets()` logic aggregates token balances from all subvaults without checking whether a given asset is supported by each subvault. Since subvaults are expected to restrict supported assets via the allowedAssets mapping, calling `hookPullAssets()` on a subvault holding an unsupported asset (even 1 wei) will trigger a revert due to the `NotAllowedAsset` check inside `modifySubvaultBalance()`.

This leads to a denial-of-service condition as any user can send a trivial amount of the queue asset (1 wei) to a `subvault` that doesn't support it. As a result, future redemption requests for that asset via `redeem()` will revert during the `callHook()` phase, making redemptions impossible.

Root Cause

The root cause lies in the fact that the system aggregates `IERC20(asset).balanceOf(subvault)` across all subvaults during `getLiquidAssets()` and then attempts to `hookPullAssets()` from each of them, without verifying if the asset is permitted by that subvault. If even a single subvault holds a non-zero balance of an unsupported asset, `modifySubvaultBalance()` will revert.

Internal Pre-conditions

A `Subvault` exists but does not include the given asset in its `allowedAssets`.

External Pre-conditions

nill

Attack Path

1. Attacker identifies a subvault that does not support the redemption asset.

2. Sends 1 wei of the redemption asset directly to the subvault.
3. When any user tries to redeem via `redeem()`, it calls `callHook()` internally.
4. `callHook()` attempts to aggregate and pull assets from all subvaults.
5. The presence of 1 wei in the unsubscribed subvault causes `modifySubvaultBalance()` to revert with `NotAllowedAsset`, permanently breaking redemption functionality.

Impact

Any user can prevent all redemptions for an asset by sending a trivial amount to a subvault that does not support it. This breaks a core vault function, locking user funds indefinitely unless manually cleaned up by governance.

PoC

nill

Affected Code

https://github.com/sherlock-audit/2025-07-mellow-flexible-vaults/blob/eca8836d68d65bcbfc52c6f04cf6b4b1597555bf/flexible-vaults/src/queues/SignatureRedeemQueue.sol#L13-L27

https://github.com/sherlock-audit/2025-07-mellow-flexible-vaults/blob/eca8836d68d65bcbfc52c6f04cf6b4b1597555bf/flexible-vaults/src/hooks/BasicRedeemHook.sol#L33-L41

https://github.com/sherlock-audit/2025-07-mellow-flexible-vaults/blob/eca8836d68d65bcbfc52c6f04cf6b4b1597555bf/flexible-vaults/src/managers/RiskManager.sol#L236-L253

Mitigation

Before summing up the liquid assets or attempting to pull balances from subvaults, explicitly check that the subvault allows the asset:

```
if (!vault.riskManager().isAllowedAsset(subvault, asset)) {
    continue;
}
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/mellow-finance/flexible-vaults/pull/14

Issue M-8: Malicious Users Can Perpetually Lock `feeRecipient` Shares via Targeted Lockup Reset

Source: https://github.com/sherlock-audit/2025-07-mellow-flexible-vaults-judging/issues/711

Found by

0x23r0, 0xc0ffEE, 7, Arav, dan__vinci, dimulski, klaus, lazyrams352, sakibcy

Summary

The protocol mints fee shares to the `feeRecipient` during deposits, redemptions, and protocol fee collection. However, the `ShareManager.mint()` function applies the `targetLockup` to **all recipients**, including the `feeRecipient`. Since this lock resets on every mint, a malicious user can repeatedly trigger minting operation and redeeming minimal amounts to **continually reset the lock timer**. As a result, the `feeRecipient`'s shares remain perpetually locked and unusable, leading to a DOS where protocol fees become inaccessible.

Root Cause

The protocol mints shares to the `feeRecipient` address in three cases: deposit fees, redeem fees, and protocol fees. Each of these cases calls the `mint` function in the `shareManager` contract, which directly mints tokens to the `feeRecipient` address based on calculations from the `FeeManager` contract.

1. `calculateDepositFee`: Calculates the deposit fee in shares based on the amount. **2.** `calculateRedeemFee`: Calculates the redeem fee in shares based on the amount. **3.** `calculateFee`: Calculates the combined performance and protocol fee in shares.

DepositQueue

```
        {
            IShareManager shareManager_ = vault_.shareManager();
            uint256 shares = Math.mulDiv(assets, reducedPriceD18, 1 ether);
            if (shares > 0) {
                shareManager_.allocateShares(shares);
            }
            uint256 fees = Math.mulDiv(assets, priceD18, 1 ether) - shares;
            if (fees > 0) {
@>>                 shareManager_.mint(feeManager.feeRecipient(), fees);
            }
        }
```

RedeemQueue

```
        {
            IFeeManager feeManager = IShareModule(vault_).feeManager();
            uint256 fees = feeManager.calculateRedeemFee(shares);
            if (fees > 0) {
@>>                 shareManager_.mint(feeManager.feeRecipient(), fees);
                shares -= fees;
            }
        }
```

Vault

```
        uint256 fees = feeManager_.calculateFee(address(this), asset, priceD18,
        ↪  shareManager_.totalShares());
        if (fees != 0) {
@>>             shareManager_.mint(feeManager_.feeRecipient(), fees);
        }
```

However, the `ShareManager` contract includes a `targetLockup()` time flag that applies to each user individually after every `mint`. When shares are minted, the contract updates `$.accounts[account].lockedUntil = lockedUntil`, meaning the newly minted shares are locked for the `targetLockup` duration.

```
function mint(address account, uint256 value) public onlyVaultOrQueue {
    if (value == 0) {
        revert ZeroValue();
    }
    _mintShares(account, value);
    ShareManagerStorage storage $ = _shareManagerStorage();
    uint32 targetLockup = $.flags.getTargetedLockup();
    if (targetLockup != 0) {
        uint32 lockedUntil = uint32(block.timestamp) + targetLockup;
        $.accounts[account].lockedUntil = lockedUntil;
        emit Mint(account, value, lockedUntil);
    } else {
        emit Mint(account, value, 0);
    }
}
```

If a user attempts an action (e.g., transfer or withdraw) before the `lockedUntil` time has passed, the transaction will revert with `TargetedLockupNotExpired`.

```
function updateChecks(address from, address to) public view {
    ShareManagerStorage storage $ = _shareManagerStorage();
    uint256 flags_ = $.flags;
    AccountInfo memory info;
        //..
```

```
        if (block.timestamp < info.lockedUntil) {
            revert TargetedLockupNotExpired(block.timestamp, info.lockedUntil);
        }

        //...
```

The result of this behavior is that **all shares minted to the `feeRecipient` including protocol, deposit, and redeem fees are automatically subjected to the `targetLockup` time restriction** defined in the `ShareManager`.

Because the `mint()` function enforces a lock on every recipient, **the `feeRecipient`'s newly minted shares become non-transferable and unusable until the lockup expires**.

Internal Pre-conditions

N/A

External Pre-conditions

N/A

Attack Path

1. **Attacker deposits assets** into the vault, receiving some amount of shares.
2. The attacker waits until the `feeRecipient`'s lockup period is about to expire.
3. Just before it expires, the attacker redeems a small amount of shares.
4. This redemption causes a small fee to be calculated and minted to the `feeRecipient`, which **resets its lock timer**.
5. The attacker **repeats steps 2–4**, front-running any legitimate action from the `feeRecipient` and indefinitely extending its lock period.
6. As a result, **any attempt by the `feeRecipient` to transfer, use, or withdraw funds will revert** with `TargetedLockupNotExpired`.

Impact

A malicious user can perform a DoS attack on the `feeRecipient` by continuously resetting its lock timer.

Since the `ShareManager.mint()` function applies the `targetLockup` duration to **every mint**, including those for the `feeRecipient`, an attacker can repeatedly:

- Mint some amount of shares,

- Redeem small amounts each time the lockup duration expires,
- Cause the `ShareManager` to mint a fee to the `feeRecipient`,
- Which in turn **resets the `lockedUntil` timestamp** on the `feeRecipient`.

This means that even after the lockup expires, the attacker can front-run any transaction involving the `feeRecipient` (e.g., transfer, withdraw, or claim) and extend its lock indefinitely.

As a result, the `feeRecipient` can be **permanently prevented from using or moving its funds**, effectively locking protocol fees forever.

PoC

No response

Mitigation

Skip applying the `targetLockup` when minting shares to the `feeRecipient` address.

Discussion

sherlock-admin2

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

Issue M-9: Stuck `stETH` rewards in queue contracts

Source: https://github.com/sherlock-audit/2025-07-mellow-flexible-vaults-judging/issues/739

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

Found by

0rpse, 0xc0ffEE, klaus

Summary

`stETH` rewards can be stuck in queue contracts because the contracts are not compatible with rebasing tokens, specifically `stETH`.

Root Cause

The system integrates with `stETH` but the implementation is not compatible with `stETH` (since `stETH` is rebase token). For example with the contract `DepositQueue`

1. At request deposit phase, assets amount of `stETH` is sent from depositor to the queue
2. The request then needs to wait for deposit interval and oracle timeout
3. When oracle reports valid price, the amount assets at step (1) is sent from queue to the vault

Indeed, within the time period in step (2) above, Lido system can report rewards (happening daily, positive or negative) and the `stETH` share price increases in case positive, resulting the queue's `stETH` balance increases to be more than `assets`. So, this means that the reward part is not accounted by the queue

Oppositely, in negative case, the queue's `stETH` balance can decrease causing it is unable to execute the step (3) (although Lido claims that the negative case has not happened).

Note: the issue can also happen with other contracts which holds funds, such that RedeemQueue, Vault

Internal Pre-conditions

NA

External Pre-conditions

NA

Attack Path

(Example above)

Impact

- `stETH` rewards are unhandled, leaving it stuck in the contract

PoC

Using this `MockRebaseERC20` as mock for `stETH`

```
// SPDX-License-Identifier: BUSL-1.1
pragma solidity 0.8.25;

import "@openzeppelin/contracts/token/ERC20/ERC20.sol";

contract MockRebaseERC20 is ERC20 {
    constructor() ERC20("MockERC20", "ME20") {}

    uint256 sharePrice = 1e18;

    uint256 constant DENOM = 1e18;

    function setSharePrice(uint256 newValue) public {
        sharePrice = newValue;
    }

    function toShares(uint256 value) public view returns (uint256 shares) {
        shares = value * DENOM / sharePrice;
    }

    function balanceOf(address account) public view override returns (uint256) {
        return (super.balanceOf(account) * sharePrice / DENOM);
    }

    function mint(address to, uint256 shares) external {
        _mint(to, shares);
    }

    function _approve(address owner, address spender, uint256 value, bool
    ↪  emitEvent) internal override {
        super._approve(owner, spender, toShares(value), emitEvent);
    }

    function _update(address from, address to, uint256 value) internal override {
        value = toShares(value);
        super._update(from, to, value);
```

```
    }
}
```

Add the test `test_stETH` below to the test file `test/unit/queues/DepositQueue.t.sol`

```
function test_stETH() public {
    skip(30 days);

    delete assetsDefault;
    asset = address(new MockRebaseERC20());
    assetsDefault.push(asset);

    Deployment memory deployment = createVault(vaultAdmin, vaultProxyAdmin,
    ↪  assetsDefault);
    DepositQueue queue = DepositQueue(addDepositQueue(deployment, vaultProxyAdmin,
    ↪  asset));
    IOracle.SecurityParams memory securityParams =
    ↪  deployment.oracle.securityParams();

    address alice = makeAddr('alice');
    pushReport(deployment.oracle, IOracle.Report({asset: asset, priceD18: 1e18}));
    uint nextReportTs = block.timestamp + securityParams.timeout;

    skip(1);
    makeDeposit(alice, 10 ether, queue);
    uint balanceBefore = MockRebaseERC20(asset).balanceOf(address(queue));

    // @info share price increases
    MockRebaseERC20(asset).setSharePrice(1.03e18);

    uint balanceAfter = MockRebaseERC20(asset).balanceOf(address(queue));

    assertGt(balanceAfter, balanceBefore);

    vm.warp(nextReportTs);
    pushReport(deployment.oracle, IOracle.Report({asset: asset, priceD18: 1e18}));

    balanceAfter = MockRebaseERC20(asset).balanceOf(address(queue));
    assertGt(balanceAfter, 0); // @info there is remaining asset here, which is
    ↪  basically rewards
}
```

Run the test and it succeeds.

It means that the increased asset (reward part) is still in the queue

Mitigation

Consider either:

- Explicitly handle `stETH` by transferring shares
- Or only support `wstETH`, which is non-rebase wrapped of `stETH`

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.