SHERLOCK

Security Review For Notional

Collaborative Audit Prepared For: **Notional**
Lead Security Expert(s): **xiaoming90**
Date Audited: **January 7 - January 9, 2026**

Introduction

TBA

Scope

Repository: notional-finance/notional-v4

Audited Commit: 86b597c0ecfb9ddcd22c07f8e2e20b69a07f83ef

Final Commit: 9f5e91e4bd3bdf0edb659e856e1c657190c6f738

Files:

- src/interfaces/IMidas.sol
- src/oracles/MidasUSDOracle.sol
- src/proxy/AddressRegistry.sol
- src/staking/AbstractStakingStrategy.sol
- src/staking/MidasStakingStrategy.sol
- src/staking/PendlePT.sol
- src/staking/PendlePT_sUSDe.sol
- src/withdraws/Midas.sol

Final Commit Hash

9f5e91e4bd3bdf0edb659e856e1c657190c6f738

Findings

Each issue has an assigned severity:

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

Issues Found

High	Medium	Low/Info
0	1	13

Issues Not Fixed and Not Acknowledged

High	Medium	Low/Info
0	0	0

Issue M-1: Valuation of withdraw request can be improved [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit/2025-12-notional-v4-midas-update-jan-7th/issues/16

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

Vulnerability Detail

When Midas's withdraw request (WR) has not been finalized yet, the protocol will price the pending WR via the `getKnownWithdrawTokenAmount()` function below. If the Strategy Vault's asset is USDC, the function will return the price of the pending WR in USDC terms.

It was observed that the function uses `request.mTokenRate`, which was set when the redemption request is submitted, to compute the value.

```
119:     function getKnownWithdrawTokenAmount(uint256 requestId)
120:         public
121:         view
122:         override
123:         returns (bool hasKnownAmount, uint256 amount)
124:     {
125:         IRedemptionVault.Request memory request =
↪  redeemVault.redeemRequests(requestId);
126:         uint256 tokenDecimals = TokenUtils.getDecimals(WITHDRAW_TOKEN);
127:
128:         // NOTE: it is possible that the mTokenRate moves a bit but this will
↪  be used
129:         // to value the withdraw request when it is pending.
130:         hasKnownAmount = true;
131:         // The request.mTokenRate will be updated when the request is
↪  processed to the final rate.
132:         amount = _truncate((request.amountMToken * request.mTokenRate) /
↪  request.tokenOutRate, tokenDecimals);
133:         // Midas vaults return the amount in 18 decimals but we need to
↪  convert to the native
134:         // token decimals here.
135:         amount = (amount * 10 ** tokenDecimals) / 1e18 - 1;
136:     }
```

Assume that it takes 3 days for Midas side to process a redemption request. Strategy Vault's asset is USDC, and 1 USDC = 1 USD hardcoded.

1. On Day 1, the price of mHYPER/USD is 1.06. A withdraw/redeem request is created, the `request.mTokenRate` is locked at 1.06.
2. On Day 2, the price of mHYPER/USD falls to 0.70 as one of its underlying

assets/investments suffers a significant loss.

3. On Day 3, the Midas's vault admin will approve the request via the following function. Based on the mToken rate passed in by the admin, the converted USDC will be transferred to WRM.

   Note that the rate is determined by Midas's vault admin, and determine if `approveRequest` or `safeApproveRequest` is called.

```
File: RedemptionVault.sol
457:      * @param requestId request id
458:      * @param newMTokenRate new mToken rate
459:      * @param isSafe new mToken rate
460:      * @param safeValidateLiquidity if true, checks if there is enough
 ↪  liquidity
461:      * and if its not sufficient, function wont fail
462:      *
463:      * @return success true if success, false only in case if
464:      * safeValidateLiquidity == true and there is not enough liquidity
465:      */
466:     function _approveRequest(
467:         uint256 requestId,
468:         uint256 newMTokenRate,
469:         bool isSafe,
470:         bool safeValidateLiquidity
471:     )
472:         internal
473:         returns (
474:             bool /* success */
475:         )
476:     {
```

The issue is that when the price falls to 0.70, the WR is still calculated using the old rate of 1.06, which overinflated it. When the actual price falls to 0.70 on Day 2, it is more likely that on Day 3, the redeem request will be processed with a new mToken rate closer to the actual price of 0.70 rather than 1.06, unless Midas is willing to absorb the loss.

On the other hand, if the price of mToken increases significantly during a short period of time, the WR will be undervalued.

Impact

The WR will be overvalued or undervalued.

Code Snippet

https://github.com/notional-finance/notional-exponent/blob/adb8c251efef419e316649dadbeb649614bf53c8/src/withdraws/Midas.sol#L119

Tool Used

Manual Review

Recommendation

There is no reliable method to determine the actual `newMTokenRate` that Midas's vault admin will pass into the `_approveRequest()` function to process the redeem request. Thus, determining the mToken rate when the request is pending is a best-effort activity, since no one can predict what value the vault admin will submit on-chain later.

On Midas side, it exposes a mToken/USD price oracle (`IMidasDataFeed(midasVault_.mTokenDataFeed()).aggregator()`). Assume that the Midas price oracle accurately reflects the mToken's actual value. In this case, during a significant price swing, it is more likely that the final `newMTokenRate` used by the vault admin will be a value somewhere close to the rate returned by the mToken/USD price oracle rather than the old/stale rate locked when the redeem request is submitted. The mToken/USD feed will provide a value more closely aligned with the current NAV.

Discussion

T-Woodward

I'm not sure about this one. Like you said, you don't actually know what mTokenRate the request will be processed at.

From my conversations with them, their treatment varies by vault. On most vaults they will use the mTokenRate at the time of processing to pay you out, but on some vaults like mF-ONE they say that they'll pay you out as of the mTokenRate you snapped at the time of request.

From a risk standpoint I'm not sure if it really matters how we mark it. The account is frozen at this point in time anyway. I suppose that the mark could affect whether liquidations occur, but given that the account is already in the withdraw queue, it is likely that any liquidator would just let the withdraw finalize first before liquidating anyway.

So I would argue that what we do here should primarily be concerned with UX and what number makes most sense to show on the UI to the user. We could I suppose have two versions of this WRM, one where we use the current mTokenRate and one where we use the old one and deploy a different according to the vault. I don't know, what do you think Jeff?

jeffywu

Well there is a lot of uncertainty around how this might be handled since the admin has four ways to finalize the request:

```
/**
   * @notice approving requests from the `requestIds` array with the
```

```
     * current mToken rate. WONT fail even if there is not enough liquidity
     * to process all requests.
     * Does same validation as `safeApproveRequest`.
     * Transfers tokenOut to users
     * Sets request flags to Processed.
     * @param requestIds request ids array
     */
    function safeBulkApproveRequest(uint256[] calldata requestIds) external;

    /**
     * @notice approving requests from the `requestIds` array using the
      ↪  `newMTokenRate`.
     * WONT fail even if there is not enough liquidity to process all requests.
     * Does same validation as `safeApproveRequest`.
     * Transfers tokenOut to user
     * Sets request flags to Processed.
     * @param requestIds request ids array
     * @param newMTokenRate new mToken rate inputted by vault admin
     */
    function safeBulkApproveRequest(
        uint256[] calldata requestIds,
        uint256 newMTokenRate
    ) external;

    /**
     * @notice approving redeem request if not exceed tokenOut allowance
     * Burns amount mToken from contract
     * Transfers tokenOut to user
     * Sets flag Processed
     * @param requestId request id
     * @param newMTokenRate new mToken rate inputted by vault admin
     */
    function approveRequest(uint256 requestId, uint256 newMTokenRate) external;

    /**
     * @notice approving request if inputted token rate fit price diviation percent
     * Burns amount mToken from contract
     * Transfers tokenOut to user
     * Sets flag Processed
     * @param requestId request id
     * @param newMTokenRate new mToken rate inputted by vault admin
     */
    function safeApproveRequest(uint256 requestId, uint256 newMTokenRate)
        external;
```

The ”safe” version restricts the change in the mTokenRate and the ”unsafe” version allows them to set any rate they want.

In the scenario outlined in this issue, it's not clear how the requests would finalize. If we pre-maturely mark the requests down to the oracle rate, then someone could front run

the user by watching the mempool and liquidating the account right before they actually finalize the withdraw request at a higher valuation. I feel like it would be most fair to allow the request to finalize at a valuation and then proceed with a liquidation.

Issue L-1: Potential pitfalls due to counterintuitive `minReceiveAmount` on the Midas vaults [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit/2025-12-notional-v4-midas-update-jan-7th/issues/9

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

Vulnerability Detail

The Midas redemption vault expects the `minReceiveAmount` to be in 18 decimal precision. Thus, if users want to receive 100 USDC, they must set the `minReceiveAmount` to be 100e18 instead of 100e6, which is counterintuitive and a potential pitfall.

```
File: MidasStakingStrategy.sol
24:     function _executeInstantRedemption(
..SNIP..
44:         ERC20(yieldToken).checkApprove(address(redeemVault),
 ↪  yieldTokensToRedeem);
45:         redeemVault.redeemInstant(address(asset), yieldTokensToRedeem,
 ↪  minReceiveAmount);
..SNIP..
```

```
File: IRedemptionVault.sol
146:      * @param tokenOut stable coin token address to redeem to
147:      * @param amountMTokenIn amount of mToken to redeem (decimals 18)
148:      * @param minReceiveAmount minimum expected amount of tokenOut to receive
 ↪  (decimals 18)
149:      */
150:     function redeemInstant(
151:         address tokenOut,
152:         uint256 amountMTokenIn,
153:         uint256 minReceiveAmount
154:     ) external;
```

Similarly, the Midas deposit vault expects the `minReceiveAmount` to be in 18 decimal precision regardless of the native token decimals.

```
File: Midas.sol
60:     function _stakeTokens(uint256 amount, bytes memory stakeData) internal
 ↪  override {
..SNIP..
70:         // Midas requires the amount to be in 18 decimals regardless of the
 ↪  native token decimals.
```

```
71:         uint256 scaledAmount = amount * 1e18 / (10 **
↪  TokenUtils.getDecimals(STAKING_TOKEN));
72:         depositVault.depositInstant(STAKING_TOKEN, scaledAmount,
↪  minReceiveAmount, i_referrerId);
73:     }
```

```
File: IDepositVault.sol
157:      * @param tokenIn address of tokenIn
158:      * @param amountToken amount of `tokenIn` that will be taken from user
↪  (decimals 18)
159:      * @param minReceiveAmount minimum expected amount of mToken to receive
↪  (decimals 18)
160:      * @param referrerId referrer id
161:      */
162:     function depositInstant(
163:         address tokenIn,
164:         uint256 amountToken,
165:         uint256 minReceiveAmount,
166:         bytes32 referrerId
167:     ) external;
```

Impact

Users might configure the minimum expected amount in the wrong decimals.

Code Snippet

https://github.com/notional-finance/notional-exponent/blob/2bcf75fd5f77d761a9cd704d865c62c82950ff2e/src/staking/MidasStakingStrategy.sol#L45

https://github.com/notional-finance/notional-exponent/blob/adb8c251efef419e316649dadbeb649614bf53c8/src/withdraws/Midas.sol#L72

Tool Used

Manual Review

Recommendation

Consider documenting clearly that the `minReceiveAmount` for Midas related function must be in 18 decimals.

Issue L-2: Residual allowance after `redeemInstant` is executed [RESOLVED]

Source: https://github.com/sherlock-audit/2025-12-notional-v4-midas-update-jan-7th/issues/10

Vulnerability Detail

Assume that the caller wants to redeem 100 mToken (Yield Token). In this case, Line 44 will grant 100 mToken allowance to the `redeemVault`. If the fee is 0.5%, Midas will charge 0.5 mToken, and the remaining 99.5 mToken will be converted to USDC.

```
File: MidasStakingStrategy.sol
24:     function _executeInstantRedemption(
25:         address sharesOwner,
26:         uint256 yieldTokensToRedeem,
27:         bytes memory redeemData
28:     )
..SNIP..
41:         uint256 assetsBefore = TokenUtils.tokenBalance(asset);
42:         (uint256 minReceiveAmount) = abi.decode(redeemData, (uint256));
43:
44:         ERC20(yieldToken).checkApprove(address(redeemVault),
↪  yieldTokensToRedeem);
45:         redeemVault.redeemInstant(address(asset), yieldTokensToRedeem,
↪  minReceiveAmount);
```

The following is taken from the Midas redemption vault's source code. Within the `_redeemInstant()` function, 99.5 mToken will be burned. However, when mToken is burned, the allowance is not consumed. Thus, when `MidasStakingStrategy` grants 100 mToken allowance to the redemption vault, only 0.5 mToken will be consumed, and there will be a residual allowance of 99.5 mToken.

```
File: RedemptionVault.sol
551:     function _redeemInstant(
..SNIP..
599:         mToken.burn(user, calcResult.amountMTokenWithoutFee);
600:         if (calcResult.feeAmount > 0)
601:             _tokenTransferFromUser(
602:                 address(mToken),
603:                 feeReceiver,
604:                 calcResult.feeAmount,
605:                 18
606:             );
```

The following is a simple test script to demonstrate this:

```solidity
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

import "forge-std/Test.sol";
import "forge-std/console2.sol";

interface IERC20 {
    function approve(address spender, uint256 amount) external returns (bool);
    function allowance(address owner, address spender) external view returns
    ↪  (uint256);
    function balanceOf(address) external view returns (uint256);
}

interface IRedeemVault {
    // no return value
    function redeemInstant(address tokenOut, uint256 amountMTokenIn, uint256
    ↪  minReceiveAmount) external;
}

contract RedeemInstantAllowanceTest is Test {
    address constant MHYPER = 0x9b5528528656DBC094765E2abB79F293c21191B9;
    address constant REDEEM_VAULT = 0x6Be2f55816efd0d91f52720f096006d63c366e98;
    address constant USDC = 0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48;

    address bob;

    function setUp() public {
        vm.createSelectFork(vm.rpcUrl("mainnet"));
        bob = makeAddr("bob");
    }

    function test_redeemInstant_printResidualAllowance() public {
        uint256 amountIn = 100e18;

        deal(MHYPER, bob, amountIn, true);

        vm.prank(bob);
        IERC20(MHYPER).approve(REDEEM_VAULT, amountIn);

        uint256 usdcBefore = IERC20(USDC).balanceOf(bob);

        vm.prank(bob);
        IRedeemVault(REDEEM_VAULT).redeemInstant(USDC, amountIn, 0);

        uint256 usdcAfter = IERC20(USDC).balanceOf(bob);
        uint256 usdcOut = usdcAfter - usdcBefore;

        console2.log("Bob USDC received:", usdcOut);
```

```
        console2.log("Residual mHYPER allowance:", IERC20(MHYPER).allowance(bob,
        ↪  REDEEM_VAULT));
    }
}
```

```
Ran 1 test for test/RedeemInstantAllowance.t.sol:RedeemInstantAllowanceTest
[PASS] test_redeemInstant_printResidualAllowance() (gas: 561198)
Logs:
  Bob USDC received: 106254523
  Residual mHYPER allowance: 99500000000000000000
```

Impact

Residual allowance will remain even after the redemption operation.

Code Snippet

https://github.com/notional-finance/notional-exponent/blob/2bcf75fd5f77d761a9cd704d865c62c82950ff2e/src/staking/MidasStakingStrategy.sol#L45

Tool Used

Manual Review

Recommendation

Consider explicitly resetting the allowance back to zero immediately after the `redeemVault.redeemInstant()` function is executed. Alternatively, consider only granting the fee amount (e.g., 0.5 mToken in this scenario), but this solution is more engineering efforts.

Issue L-3: Subtracting 1 from `tokensClaimed` is unnecessary [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit/2025-12-notional-v4-midas-update-jan-7th/issues/11

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

Vulnerability Detail

It is unnecessary to subtract 1 unit in Line 111 below because the `tokensClaimed` calculation below follows the approach done within the Midas vaults, except for the -1.

```
File: Midas.sol
094:     function _finalizeWithdrawImpl(
095:         address, /* account */
096:         uint256 requestId
097:     )
098:         internal
099:         override
100:         returns (uint256 tokensClaimed)
101:     {
102:         IRedemptionVault.Request memory request =
 ↪  redeemVault.redeemRequests(requestId);
103:         require(request.status == MidasRequestStatus.Processed);
104:         uint256 tokenDecimals = TokenUtils.getDecimals(WITHDRAW_TOKEN);
105:
106:         // Once the request is processed, the tokens are transferred to this
 ↪  contract so we calculate the
107:         // amount of tokens claimed and return it. This is not ideal since we
 ↪  don't pull the tokens but it
108:         // is how the Midas vault works.
109:         tokensClaimed = _truncate((request.amountMToken * request.mTokenRate)
 ↪  / request.tokenOutRate, tokenDecimals);
110:         // Convert to the native token decimals, subtract 1 unit to account
 ↪  for any rounding errors.
111:         tokensClaimed = (tokensClaimed * 10 ** tokenDecimals) / 1e18 - 1;
112:     }
```

The `_truncate` in Line 109 within the `_finalizeWithdrawImpl()` function is exactly what is done within the `RedemptionVault._approveRequest()` function. So, this part is fine. The `tokensClaimed` after `_truncate` is guaranteed to be a multiple of `10(18-tokenDecimals)` (for tokenDecimals < 18).

Let's review the second part, which converts 18 decimals to native token decimals in Line 111 within the `_finalizeWithdrawImpl()` function.

```
File: RedemptionVault.sol
```

```
466:     function _approveRequest(
467:         uint256 requestId,
468:         uint256 newMTokenRate,
469:         bool isSafe,
470:         bool safeValidateLiquidity
471:     )
..SNIP..
489:         uint256 amountTokenOutWithoutFee = _truncate(
490:             (request.amountMToken * newMTokenRate) / request.tokenOutRate,
491:             tokenDecimals
492:         );
..SNIP..
506:             _tokenTransferFromTo(
507:                 request.tokenOut,
508:                 requestRedeemer,
509:                 request.sender,
510:                 amountTokenOutWithoutFee,
511:                 tokenDecimals
512:             );
..SNIP..
519:         request.status = RequestStatus.Processed;
520:         request.mTokenRate = newMTokenRate;
521:         redeemRequests[requestId] = request;
```

Within the Midas redemption vault, the conversion is done within the `DecimalsCorrectionLibrary.convertFromBase18()` -> `DecimalsCorrectionLibrary.convert()` function.

```
File: ManageableVault.sol
441:     function _tokenTransferFromTo(
442:         address token,
443:         address from,
444:         address to,
445:         uint256 amount,
446:         uint256 tokenDecimals
447:     ) internal {
448:         uint256 transferAmount = amount.convertFromBase18(tokenDecimals);
..SNIP..
455:         IERC20(token).safeTransferFrom(from, to, transferAmount);
456:     }
```

```
File: DecimalsCorrectionLibrary.sol
18:     function convert(
19:         uint256 originalAmount,
20:         uint256 originalDecimals, // @audit-info 18
21:         uint256 decidedDecimals // @audit-info 6
22:     ) internal pure returns (uint256) {
23:         if (originalAmount == 0) return 0;
24:         if (originalDecimals == decidedDecimals) return originalAmount;
25:
```

```
26:         uint256 adjustedAmount;
27:
28:         if (originalDecimals > decidedDecimals) {
29:             adjustedAmount =
30:                 originalAmount /
31:                 (10**(originalDecimals - decidedDecimals));
32:         } else {
```

Notional uses:

```
(tokensClaimed * 10 ** tokenDecimals) / 1e18
```

Which is exactly the same as:

```
tokensClaimed.convertFromBase18(tokenDecimals)
```

So without `-1`, Notional would return exactly the number of tokens Midas transferred to the WRM.

In addition, -1 introduced a small risk of revert due to underflow. If tokensClaimed is small, the numerator will round down to 0, and subtracting 1 will cause an underflow and revert.

```
tokensClaimed = (tokensClaimed * 10 ** tokenDecimals) / 1e18 - 1;
```

The same issue applies to the `MidasWithdrawRequestManager.getKnownWithdrawTokenAmount()` function.

Impact

There is a risk of revert and underflow for small requests, and requests are slightly understated by 1 wei.

Code Snippet

https://github.com/notional-finance/notional-exponent/blob/adb8c251efef419e316649dadbeb649614bf53c8/src/withdraws/Midas.sol#L111

https://github.com/notional-finance/notional-exponent/blob/adb8c251efef419e316649dadbeb649614bf53c8/src/withdraws/Midas.sol#L135

Tool Used

Manual Review

Recommendation

Consider removing the -1.

```
- tokensClaimed = (tokensClaimed * 10 ** tokenDecimals) / 1e18 - 1;
+ tokensClaimed = (tokensClaimed * 10 ** tokenDecimals) / 1e18;
```

Since the `DecimalsCorrectionLibrary`, which Midas uses, is already imported into the protocol, it is also recommended to use the same library function to ensure 100% alignment.

```
- tokensClaimed = (tokensClaimed * 10 ** tokenDecimals) / 1e18;
+ tokensClaimed = tokensClaimed.convertFromBase18(tokenDecimals);
```

In this case, the computed `tokensClaimed` will match exactly the amount of tokens Midas transferred to the WRM, and there is no ambiguity.

Discussion

jeffywu

Confirmed, however, I'd prefer to just keep it as is. I think leaving a bit of dust back will not have any harm.

Issue L-4: Incorrect `updatedAt` returned from `Midas USDOracle` [RESOLVED]

Source: https://github.com/sherlock-audit/2025-12-notional-v4-midas-update-jan-7th/issues/12

Vulnerability Detail

Within the `MidasUSDOracle._calculateBaseToQuote()`, the price returned uses "answer" from both:

1. `midasOracle` (mHYPER/USD)
2. `usdcToUSDOracle` (USDC/USD)

However, it was observed that the `updatedAt` returned by this function only depends on the `updatedAt` returned by the `midasOracle`.

Assume that the `midasOracle.updatedAt` is very recent (passes freshness checks), but `usdcToUSDOracle` is stale (not updated for a long time). In this case, the oracle still returns the price with the more recent timestamp.

```
File: MidasUSDOracle.sol
27:     function _calculateBaseToQuote()
28:         internal
29:         view
30:         override
31:         returns (uint80 roundId, int256 answer, uint256 startedAt, uint256
 ↪  updatedAt, uint80 answeredInRound)
32:     {
33:         int256 mTokenRate;
34:         (roundId, mTokenRate, startedAt, updatedAt, answeredInRound) =
 ↪  midasOracle.latestRoundData();
35:         // Offset the USDC/USD rate to ensure that the USDC/USD rate is
 ↪  hardcoded to 1 in the trading
36:         // module. In the vaults we request the mToken/USDC rate so we want to
 ↪  offset any USDC variations here.
37:         int256 usdcToUSD;
38:         (, usdcToUSD,,,) = usdcToUSDOracle.latestRoundData();
39:         require(usdcToUSD > 0 && mTokenRate > 0, "Chainlink Rate Error");
40:         answer = (mTokenRate * usdcToUSD * 1e18) / (usdcToUSDDecimals *
 ↪  midasDecimals);
41:     }
```

When the `TradingModule.getOraclePrice()` performs a stale price check here, it will wrongly assume that the price is fresh, while in reality it is not because one of its factors (USDC/USD) is stale.

Impact

Incorrect `updatedAt` is being returned, preventing the downstream module from detecting a stale price.

Code Snippet

https://github.com/notional-finance/notional-exponent/blob/86b597c0ecfb9ddcd22c07f8e2e20b69a07f83ef/src/oracles/MidasUSDOracle.sol#L27

Tool Used

Manual Review

Recommendation

Consider setting `updatedAt` to the oldest timestamp from both answers. For instance, `updatedAt = min(updatedAt_midas, updatedAt_usdc)`.

Issue L-5: Hardcoded USDC/USD oracle address [RESOLVED]

Source: https://github.com/sherlock-audit/2025-12-notional-v4-midas-update-jan-7th/issues/13

Vulnerability Detail

The `MidasUSDOracle` oracle return (mHYPER/USD * USDC/USD) rate in 18 decimals. Note: mHYPER/USD => base/quote in this report.

```
File: MidasUSDOracle.sol
11: contract MidasUSDOracle is AbstractCustomOracle {
..SNIP..
27:     function _calculateBaseToQuote()
  ..SNIP..
32:     {
33:         int256 mTokenRate;
34:         (roundId, mTokenRate, startedAt, updatedAt, answeredInRound) =
 ↪  midasOracle.latestRoundData();
35:         // Offset the USDC/USD rate to ensure that the USDC/USD rate is
 ↪  hardcoded to 1 in the trading
36:         // module. In the vaults we request the mToken/USDC rate so we want to
 ↪  offset any USDC variations here.
37:         int256 usdcToUSD;
38:         (, usdcToUSD,,,) = usdcToUSDOracle.latestRoundData();
39:         require(usdcToUSD > 0 && mTokenRate > 0, "Chainlink Rate Error");
40:         answer = (mTokenRate * usdcToUSD * 1e18) / (usdcToUSDDecimals *
 ↪  midasDecimals);
41:     }
42: }
43:
```

This value is used within the `TradingModule.getOraclePrice(baseToken=mHYPER, quoteToken=USDC)`. Note that the USDC/USD will cancel off and be equal to one in the formula.

```
answer = [(mHYPER/USD * USDC/USD * 1e18) * 1e18 * 1e18] / [(USDC/USD * 1e18) *
 ↪  1e18)]
answer = mHYPER/USD * 1e18 (hardcoded 1 USDC = 1 USD)
```

https://github.com/notional-finance/leveraged-vaults/blob/7e0abc3e118db0abb20c7521c6f53f1762fdf562/contracts/trading/TradingModule.sol#L284

```
function getOraclePrice(address baseToken, address quoteToken)
..SNIP..
    answer =
        (basePrice * quoteDecimals * RATE_DECIMALS) /
```

```
        (quotePrice * baseDecimals);
    decimals = RATE_DECIMALS;
}
```

So when fetching the price of mHYPER/USDC via the `TradingModule.getOraclePrice()` function, it assumes that 1 USDC is always equal to 1 USD. For this to work, the USDC/USD price oracle used by the `MidasUSDOracle` and `TradingModule` must be the same. Otherwise, the math will not work.

However, it was observed that the USDC/USD oracle is hardcoded to `0x8fFfFfd4AfB6115b954Bd326cbe7B4BA576818f6` within the `MidasUSDOracle`. Thus, if the USDC/USD oracle in the `TradingModule` gets updated to another price oracle, the price returned from `TradingModule.getOraclePrice(baseToken=mHYPER, quoteToken=USDC)` will be invalid.

```
File: MidasUSDOracle.sol
11: contract MidasUSDOracle is AbstractCustomOracle {
12:     using TypeConvert for uint256;
13:
14:     AggregatorV2V3Interface public constant usdcToUSDOracle =
15:         AggregatorV2V3Interface(0x8fFfFfd4AfB6115b954Bd326cbe7B4BA576818f6);
16:     AggregatorV2V3Interface public immutable midasOracle;
17:     int256 public immutable usdcToUSDDecimals;
18:     int256 public immutable midasDecimals;
19:
20:     constructor(string memory description_, IMidasVault midasVault_)
↪  AbstractCustomOracle(description_, address(0)) {
21:         require(block.chainid == CHAIN_ID_MAINNET);
22:         usdcToUSDDecimals = int256(10 ** usdcToUSDOracle.decimals());
23:         midasOracle = AggregatorV2V3Interface(IMidasDataFeed(midasVault_.mToken┘
↪  DataFeed()).aggregator());mHYPER/USD oracle
24:         midasDecimals = int256(10 ** midasOracle.decimals());
25:     }
```

Impact

The price returned by `TradingModule.getOraclePrice(baseToken=mHYPER, quoteToken=USDC)` might be invalid if the USDC/USD price oracle is updated.

Code Snippet

https://github.com/notional-finance/notional-exponent/blob/86b597c0ecfb9ddcd22c07f8e2e20b69a07f83ef/src/oracles/MidasUSDOracle.sol#L15

Tool Used

Manual Review

Recommendation

Instead of hardcoding the USDC/USD in the `MidasUSDOracle`, consider fetching the USDC/USD oracle address from TradingModule's `priceOracles[USDC].oracle`.

Discussion

T-Woodward

Noted

Issue L-6: `TradingModule.getOraclePrice(mHYPER, XXXXX)` will not work as intended [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit/2025-12-notional-v4-midas-update-jan-7th/issues/14

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

Vulnerability Detail

Based on the name of the oracle (`MidasUSDOracle`), one might assume that this oracle returns the price of mHYPER/USD. However, the oracle returns the price of (mHYPER/USD * USDC/USD). Note: mHYPER/USD => base/quote in this report.

However, if one executes `TradingModule.getOraclePrice(mHYPER, WETH)`, it will not return the price of mHYPER/WETH. Instead, it will return the price of (mHYPER/WETH * USDC/USD).

Impact

If the protocol intends to support a pair with mHYPER and non-USDC tokens, using `MidasUSDOracle` directly within the `TradingModule.getOraclePrice()` will not return the correct price.

Code Snippet

https://github.com/notional-finance/notional-exponent/blob/86b597c0ecfb9ddcd22c07f8e2e20b69a07f83ef/src/oracles/MidasUSDOracle.sol#L11

Tool Used

Manual Review

Recommendation

Be aware of this pitfall, and consider documenting this.

Issue L-7: Decimal precision of the rate returned from the `MidasWithdrawRequestManager.getExchangeRate()` function deviates from other WRMs

Source: https://github.com/sherlock-audit/2025-12-notional-v4-midas-update-jan-7th/issues/15

Vulnerability Detail

All other WithdrawRequestManager's `getExchangeRate()` returns the exchange rate in Native precision. However, the `MidasWithdrawRequestManager.getExchangeRate()` function returns the exchange rate in 18 decimals, although the underlying asset is USDC denominated in 6 decimals precision.

```
File: Midas.sol
143:     function getExchangeRate() public view override returns (uint256 rate) {
144:         IMidasVault.TokenConfig memory tokenConfig =
 ↪  redeemVault.tokensConfig(WITHDRAW_TOKEN);
145:         rate = IMidasDataFeed(redeemVault.mTokenDataFeed()).getDataInBase18();
146:         if (!tokenConfig.stable) {
147:             uint256 tokenRate =
 ↪  IMidasDataFeed(tokenConfig.dataFeed).getDataInBase18();
148:             rate = 1e18 * rate / tokenRate;
149:         }
150:     }
```

Impact

Anyone who relies on WithdrawRequestManager's `getExchangeRate()` might receive results with inconsistent decimal precision.

Code Snippet

https://github.com/notional-finance/notional-exponent/blob/adb8c251efef419e316649dadbeb649614bf53c8/src/withdraws/Midas.sol#L143

Tool Used

Manual Review

Recommendation

Ensure that the rate returned from all WithdrawRequestManager's `getExchangeRate()` follows a consistent decimal precision. Consider using `convertFromBase18(tokenDecimals)` to convert the rate in 18 decimals back to native precision.

Discussion

jeffywu

Will Fix

Issue L-8: Handling of cancelled redeem request [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit/2025-12-notional-v4-midas-update-jan-7th/issues/17

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

Vulnerability Detail

Per Midas's redemption vault code, it is possible for a vault admin to cancel a redeem request.

```
File: RedemptionVault.sol
359:     function rejectRequest(uint256 requestId) external onlyVaultAdmin {
360:         Request memory request = redeemRequests[requestId];
361:
362:         _validateRequest(request.sender, request.status);
363:
364:         redeemRequests[requestId].status = RequestStatus.Canceled;
365:
366:         emit RejectRequest(requestId, request.sender);
367:     }
```

Consider a scenario where a redeem request is cancelled, and its side-effect/impact on the existing withdraw requests (WR)

1. Within the protocol, the `getKnownWithdrawTokenAmount()` function will continue to treat it as pending and still return a positive value even if it is likely that a cancel request will not be paid out. If that is the case, the WR will be overvalued.
2. During redemption, the mToken was transferred into the redemption vault when the request was created. When a request is cancelled, Midas does not automatically refund the mToken that was not processed, and will mark the request's status to `Canceled`. Since Notional will only consider the WR to be processed if `request.status == MidasRequestStatus.Processed`, the finalization will always fail, and the WR will remain stuck.

Note. Once Midas cancels a redeem request, the admin cannot execute the `_approveRequest()` function to process the redeem request anymore. So, the change in the redeem request's status is final.

```
119:     function getKnownWithdrawTokenAmount(uint256 requestId)
120:         public
121:         view
122:         override
123:         returns (bool hasKnownAmount, uint256 amount)
124:     {
```

```
125:         IRedemptionVault.Request memory request =
↪  redeemVault.redeemRequests(requestId);
126:         uint256 tokenDecimals = TokenUtils.getDecimals(WITHDRAW_TOKEN);
127:
128:         // NOTE: it is possible that the mTokenRate moves a bit but this will
↪  be used
129:         // to value the withdraw request when it is pending.
130:         hasKnownAmount = true;
131:         // The request.mTokenRate will be updated when the request is
↪  processed to the final rate.
132:         amount = _truncate((request.amountMToken * request.mTokenRate) /
↪  request.tokenOutRate, tokenDecimals);
133:         // Midas vaults return the amount in 18 decimals but we need to
↪  convert to the native
134:         // token decimals here.
135:         amount = (amount * 10 ** tokenDecimals) / 1e18 - 1;
136:     }
```

Impact

When a redeem request is cancelled, the WR might be overvalued, and the WR remains stuck.

Code Snippet

https://github.com/notional-finance/notional-exponent/blob/adb8c251efef419e316649dadbeb649614bf53c8/src/withdraws/Midas.sol#L119

Tool Used

Manual Review

Recommendation

Consider the edge case above and whether additional processing is needed to handle these scenarios. Check with Midas's protocol team to understand under what conditions a redeem request would be cancelled, whether a refund will be issued, whether there is any policy/SLA, and what happens after a cancellation. Their answer will determine how a cancelled redeem request should be valued and handled.

Discussion

T-Woodward

Yeah have done. Basically, they have said that redeem request cancellation should never happen. In the event that it does, the current behavior is the best we could hope for - account is frozen and no one will liquidate. At this point we would need to do something manual and likely would need to upgrade the contract.

I don't think there's really a way to handle this correctly because it's not clear why this would happen, what it would mean, or what would happen next. Additionally, Midas will already have the mTokens so we can't just clear the withdraw request. I think if anything, maybe you add a function that would allow us to change the withdraw request status from cancelled to processed because that's probably what we would eventually do. Midas would just send us the tokens and we manually change the request status.

Issue L-9: Midas operational limits can block deposit and redemption [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit/2025-12-notional-v4-midas-update-jan-7th/issues/18

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

Vulnerability Detail

Midas protocol enforces the following limits for its deposit and instant redemption features:

The deposit function relies on `depositInstant()` function, which is subject to the following limit:

1. Instant Daily Limit (`_requireAndUpdateLimit`) - 20 million mToken at the time of audit
2. Per-token allowance (`_requireAndUpdateAllowance`) - 116 million USDC at the time of audit
3. Max supply cap (`_validateMaxSupplyCap`) - `2256-1` at the time of audit

The instant redemption path relies on `redeemInstant` function, which is subject to the following limit:

1. Instant Daily Limit (`_requireAndUpdateLimit`) - 20 million mToken at the time of audit
2. Per-token allowance (`_requireAndUpdateAllowance`) - 202 million USDC at the time of audit

Impact

Users will not be able to deposit and perform instant redemption if any of the above limits are reached

Code Snippet

https://github.com/notional-finance/notional-exponent/blob/adb8c251efef419e316649dadbeb649614bf53c8/src/withdraws/Midas.sol#L72

https://github.com/notional-finance/notional-exponent/blob/2bcf75fd5f77d761a9cd704d865c62c82950ff2e/src/staking/MidasStakingStrategy.sol#L45

Tool Used

Manual Review

Recommendation

Since the limits are currently set high, it is unlikely that any deposit or redemption will exceed them. Thus, this is purely informational, and the team might want to consider operationally monitoring the allowance/limits or documenting them.

Discussion

T-Woodward

Noted

Issue L-10: Decoding will revert if asset != staking token [RESOLVED]

Source: https://github.com/sherlock-audit/2025-12-notional-v4-midas-update-jan-7th/issues/19

Vulnerability Detail

Assume that the staking token is USDC, while the Strategy Vault's asset is USDT.

In the `MidasStakingStrategy._mintYieldTokens` function, it takes the `depositData` and force it into "stakeData" with the `(receiver, minReceiveAmount)` format. So, if the users provide a `StakingTradeParams` data, it will be ignored and overwritten.

```
File: MidasStakingStrategy.sol
17:     function _mintYieldTokens(uint256 assets, address receiver, bytes memory
↪  depositData) internal override {
18:         (uint256 minReceiveAmount) = abi.decode(depositData, (uint256));
19:         // Ensure that we encode the proper receiver here for KYC checks.
20:         bytes memory stakeData = abi.encode(receiver, minReceiveAmount);
21:         super._mintYieldTokens(assets, receiver, stakeData);
22:     }
```

The `withdrawRequestManager.stakeTokens(address(asset), assets, depositData)` will be executed later, which will execute `_preStakingTrade()` function internally in Line 108

```
File: AbstractWithdrawRequestManager.sol
095:     function stakeTokens(
096:         address depositToken,
097:         uint256 amount,
098:         bytes calldata data
099:     )
..SNIP..
106:         uint256 initialYieldTokenBalance =
↪  ERC20(YIELD_TOKEN).balanceOf(address(this));
107:         ERC20(depositToken).safeTransferFrom(msg.sender, address(this),
↪  amount);
108:         (uint256 stakeTokenAmount, bytes memory stakeData) =
↪  _preStakingTrade(depositToken, amount, data);
109:         _stakeTokens(stakeTokenAmount, stakeData);
```

When the `_preStakingTrade()` function is executed, since `depositToken != STAKING_TOKEN`, it will attempt to decode the data as if it were a `StakingTradeParams` data in Line 342. As a result, the `abi.decode` will revert.

```
File: AbstractWithdrawRequestManager.sol
330:     function _preStakingTrade(
```

```
331:         address depositToken,
332:         uint256 depositAmount,
333:         bytes calldata data
334:     )
335:         internal
336:         returns (uint256 amountBought, bytes memory stakeData)
337:     {
338:         if (depositToken == STAKING_TOKEN) {
339:             amountBought = depositAmount;
340:             stakeData = data;
341:         } else {
342:             StakingTradeParams memory params = abi.decode(data,
↪  (StakingTradeParams));
343:             stakeData = params.stakeData;
344:
```

Impact

Staking will revert if Strategy Vault's asset is not equal to the staking token asset.

Code Snippet

https://github.com/notional-finance/notional-exponent/blob/2bcf75fd5f77d761a9cd704d865c62c82950ff2e/src/staking/MidasStakingStrategy.sol#L18

Tool Used

Manual Review

Recommendation

For the Midas integration, review whether the protocol only supports a setup where the Strategy Vault's asset equals the staking token asset. If not, the approach of encoding the account/receiver needs to be updated (e.g., decode `depositData` as `StakingTradeParams` and overwrite the `params.stakeData = abi.encode(receiver, minReceiveAmount)`).

Discussion

T-Woodward

This could be an issue for mHYPERBTC. wBTC and cbBTC are payment tokens on deposit, but only cbBTC is a payment token on redeem. So currently we would not be able to pair this vault against wBTC borrow which will be necessary.

_executeInstantRedemption also does not work if asset != staking token.

Issue L-11: Hardcoded `MidasAccessControl` and `MIDAS_GREENLISTED_ROLE` [RESOLVED]

Source: https://github.com/sherlock-audit/2025-12-notional-v4-midas-update-jan-7th/issues/20

Vulnerability Detail

It was observed that the `MidasAccessControl` and `MIDAS_GREENLISTED_ROLE` are hardcoded.

```
File: IMidas.sol
64: IMidasAccessControl constant MidasAccessControl =
↪  IMidasAccessControl(0x0312A9D1Ff2372DDEdCBB21e4B6389aFc919aC4B);
65: bytes32 constant MIDAS_GREENLISTED_ROLE =
↪  0xd2576bd6a4c5558421de15cb8ecdf4eb3282aac06b94d4f004e8cd0d00f3ebd8;
```

```
File: Midas.sol
60:     function _stakeTokens(uint256 amount, bytes memory stakeData) internal
↪  override {
61:         // NOTE: account must be encoded by the vault in the stake data, not
↪  provided by the user.
62:         (address account, uint256 minReceiveAmount) = abi.decode(stakeData,
↪  (address, uint256));
63:         if (depositVault.greenlistEnabled()) {
64:             // Ensures that any KYC checks are respected.
65:             require(MidasAccessControl.hasRole(MIDAS_GREENLISTED_ROLE,
↪  account), "Midas: account is not greenlisted");
66:         }
```

```
File: Midas.sol
75:     function _initiateWithdrawImpl(
..SNIP..
84:     {
85:         if (redeemVault.greenlistEnabled()) {
86:             // Ensures that any KYC checks are respected.
87:             require(MidasAccessControl.hasRole(MIDAS_GREENLISTED_ROLE,
↪  account), "Midas: account is not greenlisted");
88:         }
```

If Midas migrates to a new access control contract or uses a different access control contract for their vaults, the checks will be incorrect and based on the outdated access control contract.

Impact

Code Snippet

https://github.com/notional-finance/notional-exponent/blob/adb8c251efef419e316649dadbeb649614bf53c8/src/withdraws/Midas.sol#L65

https://github.com/notional-finance/notional-exponent/blob/adb8c251efef419e316649dadbeb649614bf53c8/src/withdraws/Midas.sol#L87

Tool Used

Manual Review

Recommendation

Consider deriving the access control contract address and role bytes from the vaults directly:

- `IMidasAccessControl ac = redeemVault.accessControl();`
- `bytes32 greenListedRole = ac.GREENLISTED_ROLE();`

Discussion

T-Woodward

Seems a fine change to me

Issue L-12: Midas blacklist and sanction [RESOLVED]

Source: https://github.com/sherlock-audit/2025-12-notional-v4-midas-update-jan-7th/issues/21

Vulnerability Detail

It was observed that the protocol performs the `require(MidasAccessControl.hasRole(MIDAS_GREENLISTED_ROLE, sharesOwner), ...);` checks in `MidasWithdrawRequestManager._stakeTokens` and `MidasWithdrawRequestManager._initiateWithdrawImpl` functions to prevent non-KYCed users from depositing or redeeming when Midas's greenlist is enabled.

```
File: Midas.sol
60:     function _stakeTokens(uint256 amount, bytes memory stakeData) internal
 ↪  override {
61:         // NOTE: account must be encoded by the vault in the stake data, not
 ↪  provided by the user.
62:         (address account, uint256 minReceiveAmount) = abi.decode(stakeData,
 ↪  (address, uint256));
63:         if (depositVault.greenlistEnabled()) {
64:             // Ensures that any KYC checks are respected.
65:             require(MidasAccessControl.hasRole(MIDAS_GREENLISTED_ROLE,
 ↪  account), "Midas: account is not greenlisted");
66:         }
..SNIP..
75:     function _initiateWithdrawImpl(
..SNIP..
85:         if (redeemVault.greenlistEnabled()) {
86:             // Ensures that any KYC checks are respected.
87:             require(MidasAccessControl.hasRole(MIDAS_GREENLISTED_ROLE,
 ↪  account), "Midas: account is not greenlisted");
88:         }
```

However, Midas also imposes other restrictions on users, but these were not checked in the `MidasWithdrawRequestManager`.

- sanctioned (`sanctionsList.isSanctioned(user)`)
- blacklisted (`BLACKLISTED_ROLE`)

Impact

Since the `MidasWithdrawRequestManager` does not check if the account is sanctioned or blacklisted by Midas, they could utilize Notional contracts to deposit or redeem funds, which could potentially cause Notional contracts to be sanctioned or blacklisted by Midas too.

Code Snippet

https://github.com/notional-finance/notional-exponent/blob/adb8c251efef419e316649dadbeb649614bf53c8/src/withdraws/Midas.sol#L63

https://github.com/notional-finance/notional-exponent/blob/2bcf75fd5f77d761a9cd704d865c62c82950ff2e/src/staking/MidasStakingStrategy.sol#L35

Tool Used

Manual Review

Recommendation

Consider implementing additional checks to ensure that addresses blacklisted or sanctioned by Midas cannot use Notional contracts as a proxy to transfer funds.

On a side note, be aware of the risk that greenlisting, blacklisting, and sanctioning can also be applied to Notional contracts, which can lead to funds being stuck due to inability to redeem them.

Discussion

T-Woodward

Yes valid. We should add these checks as well

Issue L-13: `minAmount` feature of Midas's DepositVault and RedemptionVault [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit/2025-12-notional-v4-midas-update-jan-7th/issues/22

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

Vulnerability Detail

It was observed that the Midas's DepositVault and RedemptionVault has a `minAmount` feature. At the time of the audit, the `minAmount` was set to 0 for both vaults, effectively disabling it.

The following is a code snippet extracted from Midas's deposit and redeem vaults.

```
File: ManageableVault.sol
282:     /**
283:      * @inheritdoc IManageableVault
284:      */
285:     function setMinAmount(uint256 newAmount) external onlyVaultAdmin {
286:         minAmount = newAmount;
287:         emit SetMinAmount(msg.sender, newAmount);
288:     }
```

```
File: DepositVault.sol
657:         if (!isFreeFromMinAmount[userCopy]) {
658:             _validateMinAmount(userCopy, result.mintAmount);
659:         }
```

```
File: RedemptionVault.sol
756:         if (!isFreeFromMinAmount[user]) {
757:             uint256 minRedeemAmount = isFiat ? minFiatRedeemAmount : minAmount;
758:             require(minRedeemAmount <= amountMTokenIn, "RV: amount < min");
759:         }
```

However, it may be possible that Midas set the `minAmount` to non-zero in the future. Often, this measure is enabled to guard against dust/small deposits and redemption, which might overwhelm the operation.

The following attempts to assess the impact if this `minAmount` increases to non-zero in the future:

1. Increasing `minAmount` later can retroactively brick a small Notional position. Some small positions may not be withdrawn if their size is below Midas's `minAmount`. This applies to both WithdrawRequest initiation and Instant redemption. Note that even if the user's positions are large in the first place, they might still become

dust/small due to partial liquidation, partial redemption, partial repayment, and fee charged by Notional via decaying effective yield token etc.

2. The deposit will revert, but the impact is not too serious, as users can always retry with a larger deposit.
3. Liquidator might perform partial liquidation, and receive a withdraw request whose position size is small. In this case, they might not be able to redeem the withdraw request.

Impact

Redemption and deposit might revert if the position size is small/dust.

Code Snippet

https://github.com/notional-finance/notional-exponent/blob/2bcf75fd5f77d761a9cd704d865c62c82950ff2e/src/staking/MidasStakingStrategy.sol#L45

https://github.com/notional-finance/notional-exponent/blob/adb8c251efef419e316649dadbeb649614bf53c8/src/withdraws/Midas.sol#L91

https://github.com/notional-finance/notional-exponent/blob/adb8c251efef419e316649dadbeb649614bf53c8/src/withdraws/Midas.sol#L72

Tool Used

Manual Review

Recommendation

If possible, consider asking Midas to exempt Notional from the minimum amount restriction since Midas's vault admin can add Notional contracts into the `isFreeFromMinAmount` mapping of their vaults to bypass any minimum amount checks. Note that both Notional's `MidasWithdrawRequestManager` and `MidasStakingStrategy` contracts need to be whitelisted.

Otherwise, more engineering efforts will be required, and more complex solutions will be needed to address the three impacts mentioned earlier in the report.

Discussion

T-Woodward

Noted. Will ask Midas to add us to the mapping

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.