SHERLOCK

SHERLOCK SECURITY REVIEW FOR

Prepared for: Notional
Prepared by: Sherlock
Lead Security Expert: xiaoming90
Dates Audited: January 8 - January 18, 2024
Prepared on: February 16, 2024

Introduction

Maximum Returns. Minimum Risk. Lend, borrow, and earn leveraged yield with DeFi's leading fixed rate lending protocol.

Scope

Repository: notional-finance/contracts-v3

Branch: audit/rebalancing-audit

Commit: d9835667a2d80ccab89233d04c3b5f9eb7ba4585

Repository: notional-finance/wrapped-fcash

Branch: feat/support-notional-v3

Commit: 3157e4c5eb105eaccbd827a8fc016a48fcc44dd7

For the detailed scope, see the contest details.

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

Medium	High
15	3

Issues not fixed or acknowledged

Medium	High
0	0

Issue H-1: Lender transactions can be front-run, leading to lost funds

Source: https://github.com/sherlock-audit/2023-12-notional-update-5-judging/issues/6

Found by

bin2chen, coffiasd

Summary

Users can mint wfCash tokens via `mintViaUnderlying` by passing a variable `minImpliedRate` to guard against trade slippage. If the market interest is lower than expected by the user, the transaction will revert due to slippage protection. However, if the user mints a share larger than maxFCash, the `minImpliedRate` check is not performed.

Vulnerability Detail

User invoke `mintViaUnderlying` to mint shares: https://github.com/sherlock-audit/2023-12-notional-update-5/blob/3bf2fb5d992dfd5aa7343d7788e881d3a4294b13/wrapped-fcash/contracts/wfCashLogic.sol#L25-L33

```
function mintViaUnderlying(
    uint256 depositAmountExternal,
    uint88 fCashAmount,
    address receiver,
    uint32 minImpliedRate//@audit when lendAmount bigger than maxFCash lack of
 ↪  minRate protect.
) external override {
    (/* */, uint256 maxFCash) = getTotalFCashAvailable();
    _mintInternal(depositAmountExternal, fCashAmount, receiver, minImpliedRate,
 ↪  maxFCash);
}
```

let's dive into `_mintInternal` we can see if `maxFCash < fCashAmount` , the vaule of `minImpliedRate` is not checked https://github.com/sherlock-audit/2023-12-notional-update-5/blob/3bf2fb5d992dfd5aa7343d7788e881d3a4294b13/wrapped-fcash/contracts/wfCashLogic.sol#L60-L68

```
if (maxFCash < fCashAmount) {
    // NOTE: lending at zero
```

```
    uint256 fCashAmountExternal = fCashAmount * precision /
↪   uint256(Constants.INTERNAL_TOKEN_PRECISION);//@audit-info fCashAmount *
↪   (underlyingTokenDecimals) / 1e8
    require(fCashAmountExternal <= depositAmountExternal);

    // NOTE: Residual (depositAmountExternal - fCashAmountExternal) will be
↪   transferred
    // back to the account
    NotionalV2.depositUnderlyingToken{value: msgValue}(address(this),
↪   currencyId, fCashAmountExternal);//@audit check this.
}
```

Imagine the following scenario:

- lender deposit Underlying token to mint some shares and set a `minImpliedRate` to protect the trsanction
- alice front-run her transaction invoke `mint` to mint some share
- the shares of lender mint now is bigger than `maxFCash`
- now the lender `lending at zero`

```
function testDepositViaUnderlying() public {
    address alice = makeAddr("alice");
    deal(address(asset), LENDER, 8800 * precision, true);
    deal(address(asset), alice, 5000 * precision, true);

    //alice deal.
    vm.stopPrank();
    vm.startPrank(alice);
    asset.approve(address(w), type(uint256).max);

    //==============================LENDER START==============================//
    vm.stopPrank();
    vm.startPrank(LENDER);
    asset.approve(address(w), type(uint256).max);
    //user DAI balance before:
    assertEq(asset.balanceOf(LENDER), 8800e18);

    (/* */, uint256 maxFCash) = w.getTotalFCashAvailable();
    console2.log("current maxFCash:",maxFCash);

    //LENDER mintViaUnderlying will revert due to slippage.
    uint32 minImpliedRate = 0.15e9;
    vm.expectRevert("Trade failed, slippage");
    w.mintViaUnderlying(5000e18,5000e8,LENDER,minImpliedRate);
    //==============================LENDER END==============================//
```

```
    //=====================alice frontrun to mint some shares.============//
    vm.stopPrank();
    vm.startPrank(alice);
    w.mint(5000e8,alice);

    //=========================LENDER TX ================================//
    vm.stopPrank();
    vm.startPrank(LENDER);
    asset.approve(address(w), type(uint256).max);
    //user DAI balance before:
    assertEq(asset.balanceOf(LENDER), 8800e18);

    //LENDER mintViaUnderlying will success.
    w.mintViaUnderlying(5000e18,5000e8,LENDER,minImpliedRate);

    console2.log("lender mint token:",w.balanceOf(LENDER));
    console2.log("lender cost DAI:",8800e18 - asset.balanceOf(LENDER));
}
```

From the above test, we can observe that if `maxFCasha` is greater than `5000e8`, the lender's transaction will be reverted due to "Trade failed, slippage." Subsequently, if Alice front-runs by invoking `mint` to create some shares before the lender, the lender's transaction will succeed. Therefore, the lender's `minImpliedRate` check will be bypassed, leading to a loss of funds for the lender.

Impact

lender lost of funds

Code Snippet

https://github.com/sherlock-audit/2023-12-notional-update-5/blob/3bf2fb5d992dfd5aa7343d7788e881d3a4294b13/wrapped-fcash/contracts/wfCashLogic.sol#L60-L68

```
if (maxFCash < fCashAmount) {
    // NOTE: lending at zero
    uint256 fCashAmountExternal = fCashAmount * precision /
↪   uint256(Constants.INTERNAL_TOKEN_PRECISION);//@audit-info fCashAmount *
↪   (underlyingTokenDecimals) / 1e8
    require(fCashAmountExternal <= depositAmountExternal);

    // NOTE: Residual (depositAmountExternal - fCashAmountExternal) will be
↪   transferred
```

```
    // back to the account
    NotionalV2.depositUnderlyingToken{value: msgValue}(address(this),
↪   currencyId, fCashAmountExternal);//@audit check this.
}
```

Tool used

Foundry Manual Review

Recommendation

add a check inside `_mintInternal`

```
        if (maxFCash < fCashAmount) {
+           require(minImpliedRate ==0,"Trade failed, slippage");
            // NOTE: lending at zero
            uint256 fCashAmountExternal = fCashAmount * precision /
↪   uint256(Constants.INTERNAL_TOKEN_PRECISION);//@audit-info fCashAmount *
↪   (underlyingTokenDecimals) / 1e8
            require(fCashAmountExternal <= depositAmountExternal);

            // NOTE: Residual (depositAmountExternal - fCashAmountExternal) will
↪   be transferred
            // back to the account
            NotionalV2.depositUnderlyingToken{value: msgValue}(address(this),
↪   currencyId, fCashAmountExternal);//@audit check this.
        }
```

Discussion

sherlock-admin2

1 comment(s) were left on this issue during the judging contest.

takarez commented:

> valid because { This is valid as the watson explained how the slippage chack can be bypassed via front-running the users deposit}

sherlock-admin

The protocol team fixed this issue in PR/commit https://github.com/notional-finance/wrapped-fcash/pull/16.

sherlock-admin

The Lead Senior Watson signed-off on the fix.

Issue H-2: Residual ETH will not be sent back to users during the minting of wfCash

Source: https://github.com/sherlock-audit/2023-12-notional-update-5-judging/issues/25

Found by

eol, twicek, xiaoming90

Summary

Residual ETH will not be sent back to users, resulting in a loss of assets.

Vulnerability Detail

At Line 67, residual ETH within the `depositUnderlyingToken` function will be sent as Native ETH back to the `msg.sender`, which is this wfCash Wrapper contract.

https://github.com/sherlock-audit/2023-12-notional-update-5/blob/main/wrapped-fcash/contracts/wfCashLogic.sol#L67

```
File: wfCashLogic.sol
35:     function _mintInternal(
..SNIP..
60:         if (maxFCash < fCashAmount) {
61:             // NOTE: lending at zero
62:             uint256 fCashAmountExternal = fCashAmount * precision /
 ↪  uint256(Constants.INTERNAL_TOKEN_PRECISION);
63:             require(fCashAmountExternal <= depositAmountExternal);
64:
65:             // NOTE: Residual (depositAmountExternal - fCashAmountExternal)
 ↪  will be transferred
66:             // back to the account
67:             NotionalV2.depositUnderlyingToken{value:
 ↪  msgValue}(address(this), currencyId, fCashAmountExternal);
..SNIP..
87:         // Residual tokens will be sent back to msg.sender, not the
 ↪  receiver. The msg.sender
88:         // was used to transfer tokens in and these are any residual tokens
 ↪  left that were not
89:         // lent out. Sending these tokens back to the receiver risks them
 ↪  getting locked on a
90:         // contract that does not have the capability to transfer them off
91:         _sendTokensToReceiver(token, msg.sender, isETH, balanceBefore);
```

Within the `depositUnderlyingToken` function Line 108 below, the `returnExcessWrapped` parameter is set to `false`, which means it will not wrap the residual ETH, and that Native ETH will be sent back to the caller (wrapper contract)

https://github.com/sherlock-audit/2023-12-notional-update-5/blob/main/contracts-v3/contracts/external/actions/AccountAction.sol#L108

```
File: AccountAction.sol
89:     function depositUnderlyingToken(
90:         address account,
91:         uint16 currencyId,
92:         uint256 amountExternalPrecision
93:     ) external payable nonReentrant returns (uint256) {
..SNIP..
File: AccountAction.sol
105:         int256 primeCashReceived = balanceState.depositUnderlyingToken(
106:             msg.sender,
107:             SafeInt256.toInt(amountExternalPrecision),
108:             false // there should never be excess ETH here by definition
109:         );
```

balanceBefore = amount of WETH before the deposit, balanceAfter = amount of WETH after the deposit.

When the `_sendTokensToReceiver` is executed, these two values are going to be the same since it is Native ETH that is sent to the wrapper instead of WETH. As a result, the Native ETH that the wrapper received is not forwarded to the users.

https://github.com/sherlock-audit/2023-12-notional-update-5/blob/main/wrapped-fcash/contracts/wfCashLogic.sol#L331

```
File: wfCashLogic.sol
331:     function _sendTokensToReceiver(
332:         IERC20 token,
333:         address receiver,
334:         bool isETH,
335:         uint256 balanceBefore
336:     ) private returns (uint256 tokensTransferred) {
337:         uint256 balanceAfter = isETH ? WETH.balanceOf(address(this)) :
 ↪  token.balanceOf(address(this));
338:         tokensTransferred = balanceAfter - balanceBefore;
339:
340:         if (isETH) {
341:             // No need to use safeTransfer for WETH since it is known to be
 ↪  compatible
342:             IERC20(address(WETH)).transfer(receiver, tokensTransferred);
343:         } else if (tokensTransferred > 0) {
```

```
344:             token.safeTransfer(receiver, tokensTransferred);
345:         }
346:     }
```

Impact

Loss of assets as the residual ETH is not sent to the users.

Code Snippet

https://github.com/sherlock-audit/2023-12-notional-update-5/blob/main/wrapped-fcash/contracts/wfCashLogic.sol#L67

https://github.com/sherlock-audit/2023-12-notional-update-5/blob/main/contracts-v3/contracts/external/actions/AccountAction.sol#L108

https://github.com/sherlock-audit/2023-12-notional-update-5/blob/main/wrapped-fcash/contracts/wfCashLogic.sol#L331

Tool used

Manual Review

Recommendation

If the underlying is ETH, measure the Native ETH balance before and after the `depositUnderlyingToken` is executed. Forward any residual Native ETH to the users, if any.

Discussion

sherlock-admin2

1 comment(s) were left on this issue during the judging contest.

takarez commented:

> valid because { This is a valid findings as the watson was able to explained how excess ETH will be stuck and not sent back to user as intended; hight findings}

sherlock-admin

The protocol team fixed this issue in PR/commit https://github.com/notional-finance/wrapped-fcash/pull/20.

sherlock-admin

The Lead Senior Watson signed-off on the fix.

Issue H-3: Residual ETH not sent back when `batchBalanceAndTradeA` executed

Source: https://github.com/sherlock-audit/2023-12-notional-update-5-judging/issues/30

Found by

bin2chen, xiaoming90

Summary

Residual ETH was not sent back when `batchBalanceAndTradeAction` function was executed, resulting in a loss of assets.

Vulnerability Detail

Per the comment at Line 122 below, when there is residual ETH, native ETH will be sent from Notional V3 to the wrapper contract. In addition, per the comment at Line 109, it is often the case to have an excess amount to be refunded to the users.

https://github.com/sherlock-audit/2023-12-notional-update-5/blob/main/wrapped-fcash/contracts/wfCashLogic.sol#L122

```
File: wfCashLogic.sol
094:     function _lendLegacy(
File: wfCashLogic.sol
109:         // If deposit amount external is in excess of the cost to purchase
↪  fCash amount (often the case),
110:         // then we need to return the difference between postTradeCash -
↪  preTradeCash. This is done because
111:         // the encoded trade does not automatically withdraw the entire
↪  cash balance in case the wrapper
112:         // is holding a cash balance.
113:         uint256 preTradeCash = getCashBalance();
114:
115:         BalanceActionWithTrades[] memory action =
↪  EncodeDecode.encodeLegacyLendTrade(
116:             currencyId,
117:             getMarketIndex(),
118:             depositAmountExternal,
119:             fCashAmount,
120:             minImpliedRate
121:         );
```

```
122:         // Notional will return any residual ETH as the native token. When
↪  we _sendTokensToReceiver those
123:         // native ETH tokens will be wrapped back to WETH.
124:         NotionalV2.batchBalanceAndTradeAction{value:
↪  msgValue}(address(this), action);
125:
126:         uint256 postTradeCash = getCashBalance();
127:
128:         if (preTradeCash != postTradeCash) {
129:             // If ETH, then redeem to WETH (redeemToUnderlying == false)
130:             NotionalV2.withdraw(currencyId, _safeUint88(postTradeCash -
↪  preTradeCash), !isETH);
131:         }
132:     }
```

This is due to how the `depositUnderlyingExternal` function within Notional V3 is implemented. The `batchBalanceAndTradeAction` will trigger the `depositUnderlyingExternal` function. Within the `depositUnderlyingExternal` function at Line 196, excess ETH will be transferred back to the account (wrapper address) in Native ETH term.

Note that for other ERC20 tokens, such as DAI or USDC, the excess will be added to the wrapper's cash balance, and this issue will not occur.

https://github.com/sherlock-audit/2023-12-notional-update-5/blob/main/contracts-v3/contracts/internal/balances/TokenHandler.sol#L196

```
File: TokenHandler.sol
181:     function depositUnderlyingExternal(
182:         address account,
183:         uint16 currencyId,
184:         int256 _underlyingExternalDeposit,
185:         PrimeRate memory primeRate,
186:         bool returnNativeTokenWrapped
187:     ) internal returns (int256 actualTransferExternal, int256
↪  netPrimeSupplyChange) {
188:         uint256 underlyingExternalDeposit =
↪  _underlyingExternalDeposit.toUint();
189:         if (underlyingExternalDeposit == 0) return (0, 0);
190:
191:         Token memory underlying = getUnderlyingToken(currencyId);
192:         if (underlying.tokenType == TokenType.Ether) {
193:             // Underflow checked above
194:             if (underlyingExternalDeposit < msg.value) {
195:                 // Transfer any excess ETH back to the account
196:                 GenericToken.transferNativeTokenOut(
```

```
197:                     account, msg.value - underlyingExternalDeposit,
↪  returnNativeTokenWrapped
198:                 );
199:             } else {
200:                 require(underlyingExternalDeposit == msg.value, "ETH
↪  Balance");
201:             }
202:
203:             actualTransferExternal = _underlyingExternalDeposit;
```

In the comment, it mentioned that any residual ETH in native token will be wrapped back to WETH by the `_sendTokensToReceiver`.

https://github.com/sherlock-audit/2023-12-notional-update-5/blob/main/wrapped-fcash/contracts/wfCashLogic.sol#L122

```
File: wfCashLogic.sol
094:     function _lendLegacy(
..SNIP..
122:         // Notional will return any residual ETH as the native token. When
↪  we _sendTokensToReceiver those
123:         // native ETH tokens will be wrapped back to WETH.
```

However, the current implementation of the `_sendTokensToReceiver`, as shown below, does not wrap the Native ETH to WETH. Thus, the residual ETH will not be sent back to the users and stuck in the contract.

https://github.com/sherlock-audit/2023-12-notional-update-5/blob/main/wrapped-fcash/contracts/wfCashLogic.sol#L331

```
File: wfCashLogic.sol
331:     function _sendTokensToReceiver(
332:         IERC20 token,
333:         address receiver,
334:         bool isETH,
335:         uint256 balanceBefore
336:     ) private returns (uint256 tokensTransferred) {
337:         uint256 balanceAfter = isETH ? WETH.balanceOf(address(this)) :
↪  token.balanceOf(address(this));
338:         tokensTransferred = balanceAfter - balanceBefore;
339:
340:         if (isETH) {
341:             // No need to use safeTransfer for WETH since it is known to be
↪  compatible
342:             IERC20(address(WETH)).transfer(receiver, tokensTransferred);
343:         } else if (tokensTransferred > 0) {
344:             token.safeTransfer(receiver, tokensTransferred);
```


```
345:         }
346:     }
```

Impact

Loss of assets as the residual ETH is not sent to the users.

Code Snippet

https://github.com/sherlock-audit/2023-12-notional-update-5/blob/main/wrapped-fcash/contracts/wfCashLogic.sol#L122

https://github.com/sherlock-audit/2023-12-notional-update-5/blob/main/contracts-v3/contracts/internal/balances/TokenHandler.sol#L196

https://github.com/sherlock-audit/2023-12-notional-update-5/blob/main/wrapped-fcash/contracts/wfCashLogic.sol#L331

Tool used

Manual Review

Recommendation

If the underlying is ETH, measure the Native ETH balance before and after the `batchBalanceAndTradeAction` is executed. Forward any residual Native ETH to the users, if any.

Discussion

sherlock-admin2

1 comment(s) were left on this issue during the judging contest.

takarez commented:

> valid because { it seem to be submitted by the same user as issue 025}

jeffywu

Is there a POC for this? My reading is that `EncodeDecode.encodeLegacyLendTrade` will not withdraw any cash and then on line 131:

```
if (preTradeCash != postTradeCash) {
            // If ETH, then redeem to WETH (redeemToUnderlying == false)
```

```
            NotionalV2.withdraw(currencyId, _safeUint88(postTradeCash -
            ↪  preTradeCash), !isETH);
}
```

We explicitly ask for the residual cash `postTradeCash - preTradeCash` and then have it returned in WETH. Therefore, the funds would be returned to the user as they are wrapped in WETH.

jeffywu

Ok, upon further clarification I see where the issue stems from.

sherlock-admin

The protocol team fixed this issue in PR/commit https://github.com/notional-finance/wrapped-fcash/pull/20.

sherlock-admin

The Lead Senior Watson signed-off on the fix.

Issue M-1: Wfcash deposit might give incorrect amount of shares (wfcash) in some cases.

Source: https://github.com/sherlock-audit/2023-12-notional-update-5-judging/issues/14

Found by

tvdung94

Summary

Wfcash deposit might not give correct amount of wfcash when there is no fcash available on the markets.

Vulnerability Detail

The preview function, which calculates the amount of share for minting, does not take the edge case, when there is not available fcash on the markets, into account. In this case, amount of shares should be equivalent to the amount of asset (ratio 1:1). However, the code will get amount of share on notionalv2 instead, which is incorrect.

Impact

Users will get less/ incorrect amount of shares when using this deposit function.

Code Snippet

https://github.com/sherlock-audit/2023-12-notional-update-5/blob/main/wrapped-fcash/contracts/wfCashERC4626.sol#L89-L106

Tool used

Manual Review

Recommendation

Consider adding code to handle the case when there is not enough available fcash on the markets.

Discussion

sherlock-admin2

1 comment(s) were left on this issue during the judging contest.

takarez commented:

> invalid because { invalid as using NotionalV2 is the right and intended behavior}

T-Woodward

_previewDeposit logic is not consistent with _mintInternal logic

sherlock-admin

The protocol team fixed this issue in PR/commit https://github.com/notional-finance/wrapped-fcash/pull/18.

sherlock-admin

The Lead Senior Watson signed-off on the fix.

Issue M-2: _isExternalLendingUnhealthy() using stale factors

Source: https://github.com/sherlock-audit/2023-12-notional-update-5-judging/issues/18

Found by

bin2chen

Summary

In `checkRebalance() -> _isExternalLendingUnhealthy() -> getTargetExternalLendingAmount(factors)` using stale `factors` will lead to inaccurate `targetAmount`, which in turn will cause `checkRebalance()` that should have been rebalance to not execute.

Vulnerability Detail

rebalancingBot uses `checkRebalance()` to return the `currencyIds []` that need to be `rebalance`.

call order : `checkRebalance() -> _isExternalLendingUnhealthy() -> ExternalLending.getTargetExternalLendingAmount(factors)`

```
    function _isExternalLendingUnhealthy(
        uint16 currencyId,
        IPrimeCashHoldingsOracle oracle,
        PrimeRate memory pr
    ) internal view returns (bool isExternalLendingUnhealthy, OracleData memory
↪   oracleData, uint256 targetAmount) {
...

@>      PrimeCashFactors memory factors =
↪   PrimeCashExchangeRate.getPrimeCashFactors(currencyId);
        Token memory underlyingToken =
↪   TokenHandler.getUnderlyingToken(currencyId);

        targetAmount = ExternalLending.getTargetExternalLendingAmount(
            underlyingToken, factors, rebalancingTargetData, oracleData, pr
        );
```

A very important logic is to get `targetAmount`. The calculation of this value depends on `factors`. But currently used is `PrimeCashFactors memory factors = PrimeCashExchangeRate.getPrimeCashFactors(currencyId);`. This is not the latest. It

has not been aggregated yet. The correct one should be `( /* */,factors) = PrimeCashExchangeRate.getPrimeCashRateView();`.

Impact

Due to the incorrect `targetAmount`, it may cause the `currencyId` that should have been re-executed `Rebalance` to not execute `rebalance`, increasing the risk of the protocol.

Code Snippet

https://github.com/sherlock-audit/2023-12-notional-update-5/blob/main/contracts-v3/contracts/external/actions/TreasuryAction.sol#L414

Tool used

Manual Review

Recommendation

```
    function _isExternalLendingUnhealthy(
        uint16 currencyId,
        IPrimeCashHoldingsOracle oracle,
        PrimeRate memory pr
    ) internal view returns (bool isExternalLendingUnhealthy, OracleData memory
↪   oracleData, uint256 targetAmount) {
...

-       PrimeCashFactors memory factors =
↪   PrimeCashExchangeRate.getPrimeCashFactors(currencyId);
+       ( /* */,PrimeCashFactors memory factors) =
↪   PrimeCashExchangeRate.getPrimeCashRateView();
        Token memory underlyingToken =
↪   TokenHandler.getUnderlyingToken(currencyId);

        targetAmount = ExternalLending.getTargetExternalLendingAmount(
            underlyingToken, factors, rebalancingTargetData, oracleData, pr
        );
```

Discussion

sherlock-admin2

1 comment(s) were left on this issue during the judging contest.

takarez commented:

> invalid because { not convince with the impact}

jeffywu

Prime factors are accrued immediately prior to the call: https://github.com/sherlock-audit/2023-12-notional-update-5/blob/main/contracts-v3/contracts/external/actions/TreasuryAction.sol#L319

bin2chen66

Escalate

This issue will not affect the `rebalance()` method, as @jeffywu mentioned, `rebalance()` uses the latest factors.

The problem I described affects `checkRebalance()`. According to the description

> /// @notice View method used by Gelato to check if rebalancing can be executed and get the execution payload. function checkRebalance() external view override returns (bool canExec, bytes memory execPayload) {

This method is for the bot. If it uses stale factors, the bot will think that the requirements are not yet met, and there is no chance to execute `rebalance()`.

thank you.

sherlock-admin2

> Escalate
>
> This issue will not affect the `rebalance()` method, as @jeffywu mentioned, `rebalance()` uses the latest factors.
>
> The problem I described affects `checkRebalance()`. According to the description
>
> > /// @notice View method used by Gelato to check if rebalancing can be executed and get the execution payload. function checkRebalance() external view override returns (bool canExec, bytes memory execPayload) {
>
> This method is for the bot. If it uses stale factors, the bot will think that the requirements are not yet met, and there is no chance to execute `rebalance()`.
>
> thank you.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

bin2chen66

different

rebalance call order: `rebalance() -> _rebalanceCurrency() (will refresh factors first) -> _isExternalLendingUnhealthy()  will get latest factors`

checkRebalance call order: `checkRebalance() -> _isExternalLendingUnhealthy() will get stale factors without refresh factors first`

then If `checkRebalance ()` is incorrect, `rebalance ()` may not have a chance to execute

jeffywu

Understood, this is a valid escalation.

nevillehuang

Since this could directly undermine frequency of rebalancing, agree it can be medium severity.

Evert0x

@bin2chen66 do I understand correctly that this only impacts the bot Notional is planning to use?

> This method is for the bot. If it uses stale factors, the bot will think that the requirements are not yet met, and there is no chance to execute rebalance().

bin2chen66

@Evert0x Hi, Yes, as far as I know, the logic of the bot is very simple, `checkRebalance ()` is called periodically to get `execPayload`, such as execPayload=[1,2,3]. if execPayload is not empty then the bot calls `rebalance (execPayload)` . Whether there are any other uses, needs to be confirmed by the sponsor.

Evert0x

@bin2chen66 my judgment will be to keep the issue invalid

> Incorrect values in View functions are by default considered low.
> Exception: In case any of these incorrect values returned by the view functions are used as a part of a larger function which would result in loss of funds then it would be a valid medium/high depending on the impact.

The part "used as a part of a larger function which would result in loss of funds" doesn't apply here as the bot is an off-chain component.

bin2chen66

@Evert0x I have an objection. This `rebalance()` depends entirely on `checkRebalance()`. Why is it not part of this function? How do you chew words? please respect the facts

Evert0x

@bin2chen66 it's not part of the function as an off-chain component (the bot) is expected to consume the output of the view function and initiate a transaction to call `rebalance()`

> Rebalancing Bot: is an off chain bot that detects when rebalancing can occur via checkRebalance and will rebalance those currencies accordingly.

However, in the README it's clearly stated that the output of the view function is of importance to the functioning of the protocol.

Planning to accept escalation and make Medium.

Evert0x

Result: Medium Unique

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- bin2chen66: accepted

sherlock-admin

The protocol team fixed this issue in PR/commit https://github.com/notional-finance/contracts-v3/pull/33.

sherlock-admin

The Lead Senior Watson signed-off on the fix.

Issue M-3: recover() using the standard transfer may not be able to retrieve some tokens

Source: https://github.com/sherlock-audit/2023-12-notional-update-5-judging/issues/19

Found by

AuditorPraise, MohammedRizwan, bin2chen, twicek

Summary

in `SecondaryRewarder.recover()` Using the standard `IERC20.transfer()` If `REWARD_TOKEN` is like `USDT`, it will not be able to transfer out, because this kind of `token` does not return `bool` This will cause it to always `revert`

Vulnerability Detail

`SecondaryRewarder.recover()` use for

> Allows the Notional owner to recover any tokens sent to the address or any reward tokens remaining on the contract in excess of the total rewards emitted.

```
    function recover(address token, uint256 amount) external onlyOwner {
        if (Constants.ETH_ADDRESS == token) {
            (bool status,) = msg.sender.call{value: amount}("");
            require(status);
        } else {
@>          IERC20(token).transfer(msg.sender, amount);
        }
    }
```

Using the standard `IERC20.transfer()` method to execute the transfer A `token` of a type similar to `USDT` has no return value This will cause the execution of the transfer to always fail

Impact

If `REWARD_TOKEN` is like `USDT`, it will not be able to transfer out.

Code Snippet

https://github.com/sherlock-audit/2023-12-notional-update-5/blob/main/contracts-v3/contracts/external/adapters/SecondaryRewarder.sol#L152C3-L159

Tool used

Manual Review

Recommendation

```
    function recover(address token, uint256 amount) external onlyOwner {
        if (Constants.ETH_ADDRESS == token) {
            (bool status,) = msg.sender.call{value: amount}("");
            require(status);
        } else {
-           IERC20(token).transfer(msg.sender, amount);
+           GenericToken.safeTransferOut(token,msg.sender,amount);
        }
    }
```

Discussion

nevillehuang

@jeffywu @T-Woodward Were there any publicly available information stating USDT won't be use as a potential reward tokens at the point of the contest?

sherlock-admin2

> Escalate Upon closer examination and in alignment with the Sherlock rules, it becomes evident that issues of this nature are categorically classified under informational issues. Furthermore, should we acknowledge the concerns surrounding `safeTransferOut` due to `USDT` peculiarities, it is imperative to underscore that, at most, this warrants a classification of low severity. This is primarily because the core functionality of the protocol remains unaffected and fully operational without getting bricked.

You've deleted an escalation for this issue.

AuditorPraise

> Escalate Upon closer examination and in alignment with the Sherlock rules, it becomes evident that issues of this nature are categorically classified under informational issues. Furthermore, should we acknowledge the concerns surrounding `safeTransferOut` due to `USDT` peculiarities, it is imperative to underscore that, at most, this warrants a classification of low severity. This is primarily because the core functionality of the protocol remains unaffected and fully operational without getting bricked.

"Non-Standard tokens: Issues related to tokens with non-standard behaviors, such as weird-tokens are not considered valid by default unless these tokens are explicitly mentioned in the README"

contest readme::

```
Do you expect to use any of the following tokens with non-standard behaviour
↪  with the smart contracts?

USDC and USDT are the primary examples.
```

0xMR0

Escalate

This is indeed a valid issue since the non-standard behavior of USDT is not acceptable to protocol team and it is explicitly mentioned in contest readme.

Further, comment by @AuditorPraise is correct and enough for the validation of this issue.

sherlock-admin2

> Escalate
>
> This is indeed a valid issue since the non-standard behavior of USDT is not acceptable to protocol team and it is explicitly mentioned in contest readme.
>
> Further, comment by @AuditorPraise is correct and enough for the validation of this issue.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

Hash01011122

IMO In my opinion, the issue with `safeTransfer` has been widely recognized since 2022. Furthermore, it seems unfair to Watson who are discovering different vulnerabilities along with their attack paths. For instance, the issue of hardcoded `chainId` was previously classified as high severity. However, as seen in this issue, it was downgraded to low severity due to its widespread awareness and ease of discovery.

As mentioned in sherlock rules: Low/Informational Issues: While Sherlock acknowledges that it would be great to include & reward low-impact/informational

issues, we strongly feel that Watsons should focus on finding the most critical vulnerabilities that will potentially cause millions of dollars of losses on mainnet. Sherlock understands that it could be missing out on some potential "value add" for protocol, but it's only because the real task of finding critical vulnerabilities requires 100% of the attention of Watsons. While low/informational issues are not rewarded individually if a Watson identifies an attack vector that combines multiple lows to cause significant loss/damage that would still be categorized as a valid medium/high.

AuditorPraise

> IMO In my opinion, the issue with `safeTransfer` has been widely recognized since 2022.

The issue isn't about `safeTransfer` but USDT. It's no one's fault that devs still make such mistakes... But that doesn't reduce the risks associated with making such a mistake. The impact it has had on protocols since 2022 till now remains the same.

Funds could be stuck

So why should it be an informational now?

You can't compare chain Id issue to USDT being stuck in a contract as a result of the devs not using safeTransfer.

nevillehuang

@Hash01011122 You are circling too much around sherlock rules, and should look at it more factually instead of subjectively. In the contest details/code logic/documentation, no place does it mention that USDT cannot be a reward token, so I believe your argument is basically invalid. I believe no further discussions is required from my side, imo, this should remain medium severity.

Evert0x

Result: Medium Has Duplicates

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- 0xMR0: accepted

sherlock-admin

The protocol team fixed this issue in PR/commit https://github.com/notional-finance/contracts-v3/pull/28.

sherlock-admin

The Lead Senior Watson signed-off on the fix.

Issue M-4: Malicious users could block liquidation or perform DOS

Source: https://github.com/sherlock-audit/2023-12-notional-update-5-judging/issues/22

Found by

bin2chen, xiaoming90

Summary

The current implementation uses a "push" approach where reward tokens are sent to the recipient during every update, which introduces additional attack surfaces that the attackers can exploit. An attacker could intentionally affect the outcome of the transfer to gain a certain advantage or carry out certain attack.

The worst-case scenario is that malicious users might exploit this trick to intentionally trigger a revert when someone attempts to liquidate their unhealthy accounts to block the liquidation, leaving the protocol with bad debts and potentially leading to insolvency if it accumulates.

Vulnerability Detail

Per the Audit Scope Documentation provided by the protocol team on the contest page, the reward tokens can be any arbitrary ERC20 tokens

> We are extending this functionality to allow nTokens to be incentivized by a secondary reward token. On Arbitrum, this will be ARB as a result of the ARB STIP grant. In the future, this may be any arbitrary ERC20 token

Line 231 of the `_claimRewards` function below might revert due to various issues such as:

- tokens with blacklisting features such as USDC (users might intentionally get into the blacklist to achieve certain outcomes)
- tokens with hook, which allow the target to revert the transaction intentionally
- unexpected error in the token's contract

https://github.com/sherlock-audit/2023-12-notional-update-5/blob/main/contracts-v3/contracts/external/adapters/SecondaryRewarder.sol#L231

```
File: SecondaryRewarder.sol
216:     function _claimRewards(address account, uint256 nTokenBalanceBefore,
↪  uint256 nTokenBalanceAfter) private {
```

```
217:         uint256 rewardToClaim = _calculateRewardToClaim(account,
↪  nTokenBalanceBefore, accumulatedRewardPerNToken);
218:
219:         // Precision here is:
220:         //  nTokenBalanceAfter (INTERNAL_TOKEN_PRECISION)
221:         //  accumulatedRewardPerNToken (INCENTIVE_ACCUMULATION_PRECISION)
222:         // DIVIDE BY
223:         //  INTERNAL_TOKEN_PRECISION
224:         //  => INCENTIVE_ACCUMULATION_PRECISION (1e18)
225:         rewardDebtPerAccount[account] = nTokenBalanceAfter
226:             .mul(accumulatedRewardPerNToken)
227:             .div(uint256(Constants.INTERNAL_TOKEN_PRECISION))
228:             .toUint128();
229:
230:         if (0 < rewardToClaim) {
231:             GenericToken.safeTransferOut(REWARD_TOKEN, account,
↪  rewardToClaim);
232:             emit RewardTransfer(REWARD_TOKEN, account, rewardToClaim);
233:         }
234:     }
```

If a revert occurs, the following functions are affected:

```
_claimRewards -> claimRewardsDirect

_claimRewards -> claimRewards -> Incentives.claimIncentives
_claimRewards -> claimRewards -> Incentives.claimIncentives ->
↪  BalancerHandler._finalize
_claimRewards -> claimRewards -> Incentives.claimIncentives ->
↪  BalancerHandler._finalize -> Used by many functions

_claimRewards -> claimRewards -> Incentives.claimIncentives ->
↪  BalancerHandler.claimIncentivesManual
_claimRewards -> claimRewards -> Incentives.claimIncentives ->
↪  BalancerHandler.claimIncentivesManual -> nTokenAction.nTokenClaimIncentives
↪  (External)
_claimRewards -> claimRewards -> Incentives.claimIncentives ->
↪  BalancerHandler.claimIncentivesManual -> nTokenAction.nTokenClaimIncentives
↪  (External) -> claimNOTE (External)
```

Impact

Many of the core functionalities of the protocol will be affected by the revert. Specifically, the `BalancerHandler._finalize` has the most impact as this function is called by almost every critical functionality of the protocol, including deposit,

withdrawal, and liquidation.

The worst-case scenario is that malicious users might exploit this trick to intentionally trigger a revert when someone attempts to liquidate their unhealthy accounts to block the liquidation, leaving the protocol with bad debts and potentially leading to insolvency if it accumulates.

Code Snippet

https://github.com/sherlock-audit/2023-12-notional-update-5/blob/main/contracts-v3/contracts/external/adapters/SecondaryRewarder.sol#L231

Tool used

Manual Review

Recommendation

The current implementation uses a "push" approach where reward tokens are sent to the recipient during every update, which introduces additional attack surfaces that the attackers can exploit.

Consider adopting a pull method for users to claim their rewards instead so that the transfer of reward tokens is disconnected from the updating of reward balances.

Discussion

jeffywu

I believe this is a duplicate of another issue.

sherlock-admin2

1 comment(s) were left on this issue during the judging contest.

takarez commented:

> valid because { valid user can avoid liquidation}

jeffywu

Acknowledged this is a very minor risk in the case this prevents a liquidation. However, I think the fix here is to put a try / catch around the reward block and it results in a loss of rewards for the blacklisted account rather than changing the entire UX of the process.

sherlock-admin

The protocol team fixed this issue in PR/commit https://github.com/notional-finance/contracts-v3/pull/30.

sherlock-admin

The Lead Senior Watson signed-off on the fix.

Issue M-5: Low precision is used when checking spot price deviation

Source: https://github.com/sherlock-audit/2023-12-notional-update-5-judging/issues/32

Found by

xiaoming90

Summary

Low precision is used when checking spot price deviation, which might lead to potential manipulation or create the potential for an MEV opportunity due to valuation discrepancy.

Vulnerability Detail

Assume the following:

- The max deviation is set to 1%
- `nTokenOracleValue` is 1,000,000,000
- `nTokenSpotValue` is 980,000,001

https://github.com/sherlock-audit/2023-12-notional-update-5/blob/main/contracts-v3/contracts/global/Constants.sol#L47

```
File: Constants.sol
46:     // Basis for percentages
47:     int256 internal constant PERCENTAGE_DECIMALS = 100;
```

https://github.com/sherlock-audit/2023-12-notional-update-5/blob/main/contracts-v3/contracts/internal/nToken/nTokenCalculations.sol#L65

```
File: nTokenCalculations.sol
61:             int256 maxValueDeviationPercent = int256(
62:
↪   uint256(uint8(nToken.parameters[Constants.MAX_MINT_DEVIATION_LIMIT]))
63:             );
64:             // Check deviation limit here
65:             int256 deviationInPercentage =
↪   nTokenOracleValue.sub(nTokenSpotValue).abs()
```

```
66:                     .mul(Constants.PERCENTAGE_DECIMALS).div(nTokenOracleValue);
67:             require(deviationInPercentage <= maxValueDeviationPercent, "Over
↪  Deviation Limit");
```

Based on the above formula:

```
nTokenOracleValue.sub(nTokenSpotValue).abs().mul(Constants.PERCENTAGE_DECIMALS).
↪  div(nTokenOracleValue);
((nTokenOracleValue - nTokenSpotValue) * Constants.PERCENTAGE_DECIMALS) /
↪  nTokenOracleValue
((1,000,000,000 - 980,000,001) * 100) / 1,000,000,000
(19,999,999 * 100) / 1,000,000,000
1,999,999,900 / 1,000,000,000 = 1.9999999 = 1
```

The above shows that the oracle and spot values have deviated by 1.99999%, which is close to 2%. However, due to a rounding error, it is rounded down to 1%, and the TX will not revert.

Impact

The purpose of the deviation check is to ensure that the spot market value is not manipulated. If the deviation check is not accurate, it might lead to potential manipulation or create the potential for an MEV opportunity due to valuation discrepancy.

Code Snippet

https://github.com/sherlock-audit/2023-12-notional-update-5/blob/main/contracts-v3/contracts/internal/nToken/nTokenCalculations.sol#L65

Tool used

Manual Review

Recommendation

Consider increasing the precision.

For instance, increasing the precision from `Constants.PERCENTAGE_DECIMALS` (100) to 1e8 would have caught the issue mentioned earlier in the report even after the rounding down.

```
nTokenOracleValue.sub(nTokenSpotValue).abs().mul(1e8).div(nTokenOracleValue);
((nTokenOracleValue - nTokenSpotValue) * 1e8) / nTokenOracleValue
```

```
((1,000,000,000 - 980,000,001) * 1e8) / 1,000,000,000
(19,999,999 * 1e8) / 1,000,000,000 = 1999999.9 = 1999999
```

1% of 1e8 = 1000000

```
require(deviationInPercentage <= maxValueDeviationPercent, "Over Deviation
↪  Limit")
require(1999999 <= 1000000, "Over Deviation Limit") => Revert
```

Discussion

sherlock-admin2

1 comment(s) were left on this issue during the judging contest.

takarez commented:

> valid because { This is valid}

sherlock-admin

The protocol team fixed this issue in PR/commit https://github.com/notional-finance/contracts-v3/pull/34.

sherlock-admin

The Lead Senior Watson signed-off on the fix.

Issue M-6: The use of spot data when discounting is subjected to manipulation

Source: https://github.com/sherlock-audit/2023-12-notional-update-5-judging/issues/36

Found by

xiaoming90

Summary

The use of spot data when discounting is subjected to manipulation. As a result, malicious users could receive more cash than expected during redemption by performing manipulation. Since this is a zero-sum, the attacker's gain is the protocol loss.

Vulnerability Detail

When redeeming wfCash before maturity, the `_sellfCash` function will be executed.

Assume that there is insufficient fCash left on the wrapper to be sold back to the Notional AMM. In this case, the `getPrincipalFromfCashBorrow` view function will be used to calculate the number of prime cash to be withdrawn for a given fCash amount and sent to the users.

Note that the `getPrincipalFromfCashBorrow` view function uses the spot data (spot interest rate, spot utilization, spot totalSupply/totalDebt, etc.) internally when computing the prime cash to be withdrawn for a given fCash. Thus, it is subjected to manipulation.

https://github.com/sherlock-audit/2023-12-notional-update-5/blob/main/contracts-v3/contracts/external/CalculationViews.sol#L440

```
File: wfCashLogic.sol
274:     /// @dev Sells an fCash share back on the Notional AMM
275:     function _sellfCash(
276:         address receiver,
277:         uint256 fCashToSell,
278:         uint32 maxImpliedRate
279:     ) private returns (uint256 tokensTransferred) {
280:         (IERC20 token, bool isETH) = getToken(true);
281:         uint256 balanceBefore = isETH ? WETH.balanceOf(address(this)) :
 ↪  token.balanceOf(address(this));
282:         uint16 currencyId = getCurrencyId();
283:
```

```
284:         (uint256 initialCashBalance, uint256 fCashBalance) = getBalances();
285:         bool hasInsufficientfCash = fCashBalance < fCashToSell;
286:
287:         uint256 primeCashToWithdraw;
288:         if (hasInsufficientfCash) {
289:             // If there is insufficient fCash, calculate how much prime
    ↪  cash would be purchased if the
290:             // given fCash amount would be sold and that will be how much
    ↪  the wrapper will withdraw and
291:             // send to the receiver. Since fCash always sells at a discount
    ↪  to underlying prior to maturity,
292:             // the wrapper is guaranteed to have sufficient cash to send to
    ↪  the account.
293:             (/* */, primeCashToWithdraw, /* */, /* */) =
    ↪  NotionalV2.getPrincipalFromfCashBorrow(
294:                 currencyId,
295:                 fCashToSell,
296:                 getMaturity(),
297:                 0,
298:                 block.timestamp
299:             );
300:             // If this is zero then it signifies that the trade will fail.
301:             require(primeCashToWithdraw > 0, "Redeem Failed");
302:
303:             // Re-write the fCash to sell to the entire fCash balance.
304:             fCashToSell = fCashBalance;
305:         }
```

Within the CalculationViews.getPrincipalFromfCashBorrow view function, it will rely on the InterestRateCurve.calculatefCashTrade function to compute the cash to be returned based on the current interest rate model.

The Notional AMM uses a two-kink interest rate model, and the interest rate is computed based on the utilization rate (https://docs.notional.finance/notional-v3/prime-money-market/interest-rate-model), as shown below. The interest rate is used to discount the fCash back to cash before maturity. In summary, a higher interest rate will cause the fCash to be discounted more, and the current cash value will be smaller. The opposite is also true (Lower interest rate => Higher cash returned).

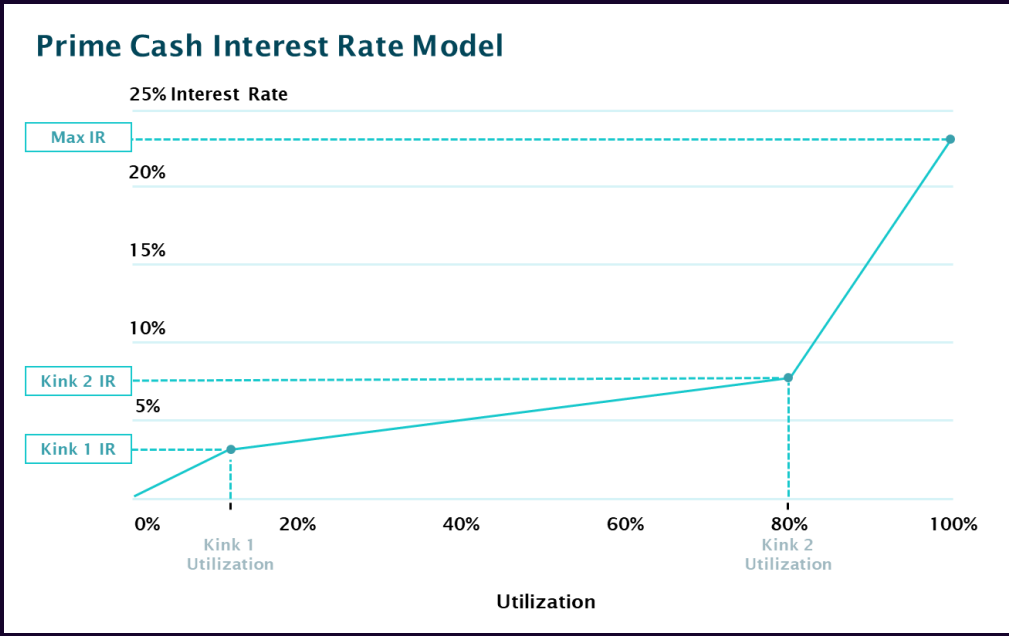
Assume that the current utilization rate is slightly above Kink 1. When Bob redeems his wfCash, the interest rate used falls within the gentle slope between Kink 1 and Kink 2. Let the interest rate based on current utilization be 4%. The amount of fCash will be discounted back with 4% interest rate to find out the cash value (present value) and the returned value is x.

Observed that before Kink 1, the interest rate changed sharply. If one could nudge the utilization toward the left (toward zero) and cause the utilization to fall between

Kink 0 and Kink 1, the interest rate would fall sharply. Since the utilization is computed as on `utilization = totalfCash/totalCashUnderlying`, one could deposit prime cash to the market to increase the denominator (`totalCashUnderlying`) to bring down the utilization rate.

Bob deposits a specific amount of prime cash (either by its own funds or flash-loan) to reduce the utilization rate, which results in a reduction in interest rate. Assume that the interest rate reduces to 1.5%. The amount of fCash will be discounted with a lower interest rate of 1.5%, which will result in higher cash value, and the returned value/received cash is y.

$y > x$. So Bob received $y - x$ more cash compared to if he had not performed the manipulation. Since this is a zero-sum, Bob's gain is the protocol loss.



Impact

Malicious users could receive more cash than expected during redemption by performing manipulation. Since this is a zero-sum, the attacker's gain is the protocol loss.

Code Snippet

https://github.com/sherlock-audit/2023-12-notional-update-5/blob/main/contracts-v3/contracts/external/CalculationViews.sol#L440

Tool used

Manual Review

Recommendation

Avoid using spot data when computing the amount of assets that the user is entitled to during redemption. Consider using a TWAP/Time-lagged oracle to guard against potential manipulation.

Discussion

jeffywu

This is valid, we should apply some slippage check against the `maxImpliedRate` that the user submits to this method against the calculation method.

sherlock-admin2

1 comment(s) were left on this issue during the judging contest.

takarez commented:

> valid because { valid and seem to be by same watson as issue 028;duplicate}

T-Woodward

I would reduce the severity on this because there's no direct financial incentive for someone to manipulate the rate here. No actual trade occurs, so there's nothing to frontrun/sandwich. The attacker would incur a cost to manipulate this and receive no offsetting direct benefit.

I think the proper mitigation for this is what @jeffywu has suggested which is to implement a slippage check on this calculation.

nevillehuang

@T-Woodward @jeffywu There seems to be some discussion in severity for this and #28. What is the worst case scenario for manipulating spot data? Is my understanding of it being it will take alot of funds to manipulate to gain some profits in terms of cash received correct?

T-Woodward

@nevillehuang manipulation would cost a lot and there would be no chance to gain any profit. The attacker would lose money. The negative impact is that a regular user would also lose money. But given that an attacker can not profit from this, I recommend medium severity, not high severity.

sherlock-admin

The protocol team fixed this issue in PR/commit https://github.com/notional-finance/wrapped-fcash/pull/14.

sherlock-admin

The Lead Senior Watson signed-off on the fix.

Issue M-7: External lending can exceed the threshold

Source: https://github.com/sherlock-audit/2023-12-notional-update-5-judging/issues/38

Found by

xiaoming90

Summary

Due to an incorrect calculation of the max lending amount, external lending can exceed the external withdrawal threshold. If this restriction/threshold is not adhered to, users or various core functionalities within the protocol will have issues redeeming or withdrawing their prime cash.

Vulnerability Detail

The following is the extract from the Audit Scope Documentation provided by the protocol team on the contest page that describes the external withdraw threshold:

> External Withdraw Threshold: ensures that Notional has sufficient liquidity to withdraw from an external lending market. If Notional has 1000 units of underlying lent out on Aave, it requires 1000 * externalWithdrawThreshold units of underlying to be available on Aave for withdraw. This ensures there is sufficient buffer to process the redemption of Notional funds. If available liquidity on Aave begins to drop due to increased utilization, Notional will automatically begin to withdraw its funds from Aave to ensure that they are available for withdrawal on Notional itself.

To ensure the redeemability of Notional's funds on external lending markets, Notional requires there to be redeemable funds on the external lending market that are a multiple of the funds that Notional has lent on that market itself.

Assume that the `externalWithdrawThreshold` is 200% and the underlying is USDC. Therefore, `PERCENTAGE_DECIMALS/externalWithdrawThreshold = 100/200 = 0.5` (Line 83-84 below). This means that the number of USDC to be available on AAVE for withdrawal must be two (2) times the number of USDC Notional lent out on AAVE (A multiple of 2).

The `externalUnderlyingAvailableForWithdraw` stores the number of liquidity in USDC on the AAVE pool available to be withdrawn.

If `externalUnderlyingAvailableForWithdraw` is 1000 USDC and `currentExternalUnderlyingLend` is 400 USDC, this means that the remaining 600

USDC liquidity on the AAVE pool is not owned by Notional.

The `maxExternalUnderlyingLend` will be `600 * 0.5 = 300`. Thus, the maximum amount that Notional can lend externally at this point is 300 USDC.

Assume that after Notional has lent 300 USDC externally to the AAVE pool.

The `currentExternalUnderlyingLend` will become `400+300=700`, and the `externalUnderlyingAvailableForWithdraw` will become `1000+300=1300`

Following is the percentage of USDC in AAVE that belong to Notional

```
700/1300 = 0.5384615385 (53%).
```

At this point, the invariant is broken as the number of USDC to be available on AAVE for withdrawal is less than two (2) times the number of USDC lent out on AAVE after the lending.

https://github.com/sherlock-audit/2023-12-notional-update-5/blob/main/contracts-v3/contracts/internal/balances/ExternalLending.sol#L81

```
File: ExternalLending.sol
36:     function getTargetExternalLendingAmount(
..SNIP..
79:         uint256 maxExternalUnderlyingLend;
80:         if (oracleData.currentExternalUnderlyingLend <
 ↪  oracleData.externalUnderlyingAvailableForWithdraw) {
81:             maxExternalUnderlyingLend =
82:                 (oracleData.externalUnderlyingAvailableForWithdraw -
 ↪  oracleData.currentExternalUnderlyingLend)
83:                 .mul(uint256(Constants.PERCENTAGE_DECIMALS))
84:                 .div(rebalancingTargetData.externalWithdrawThreshold);
85:         } else {
86:             maxExternalUnderlyingLend = 0;
87:         }
```

The root cause is that when USDC is deposited to AAVE to get aUSDC, the total USDC in the pool increases. Therefore, using the current amount of USDC in the pool to determine the maximum deposit amount is not an accurate measure of liquidity risk.

Impact

To ensure the redeemability of Notional's funds on external lending markets, Notional requires there to be redeemable funds on the external lending market that are a multiple of the funds that Notional has lent on that market itself.

If this restriction is not adhered to, users or various core functionalities within the protocol will have issues redeeming or withdrawing their prime cash. For instance, users might not be able to withdraw their assets from the protocol due to insufficient liquidity, or liquidation cannot be carried out due to lack of liquidity, resulting in bad debt accumulating within the protocol and negatively affecting the protocol's solvency.

Code Snippet

https://github.com/sherlock-audit/2023-12-notional-update-5/blob/main/contracts-v3/contracts/internal/balances/ExternalLending.sol#L81

Tool used

Manual Review

Recommendation

To ensure that a deposit does not exceed the threshold, the following formula should be used to determine the maximum deposit amount:

Let's denote:

T as the externalWithdrawThreshold L as the currentExternalUnderlyingLend W as the externalUnderlyingAvailableForWithdraw D as the Deposit (the variable we want to solve for)

$$T = \frac{L + D}{W + D}$$

Solving D, the formula for calculating the maximum deposit (D) is

$$D = \frac{TW - L}{1 - T}$$

Using back the same example in the "Vulnerability Detail" section.

The maximum deposit amount is as follows:

```
D = (TW - L) / (1 - T)
D = (0.5 * 1000 - 400) / (1 - 0.5)
D = (500 - 400) / 0.5 = 200
```

If 200 USDC is lent out, it will still not exceed the threshold of 200%, which demonstrates that the formula is working as intended in keeping the multiple of two (200%) constant before and after the deposit.

```
(400 + 200) / (1000 + 200) = 0.5
```

Discussion

sherlock-admin2

1 comment(s) were left on this issue during the judging contest.

takarez commented:

> valid because { valid and a duplicate of 035 and even seem to be by same user}

sherlock-admin

The protocol team fixed this issue in PR/commit https://github.com/notional-finance/contracts-v3/pull/27.

sherlock-admin

The Lead Senior Watson signed-off on the fix.

Issue M-8: Unexpected behavior when calling certain ERC4626 functions

Source: https://github.com/sherlock-audit/2023-12-notional-update-5-judging/issues/43

Found by

CL001, xiaoming90

Summary

Unexpected behavior could occur when certain ERC4626 functions are called during the time windows when the fCash has matured but is not yet settled.

Vulnerability Detail

When the fCash has matured, the global settlement does not automatically get executed. The global settlement will only be executed when the first account attempts to settle its own account. The code expects the `pr.supplyFactor` to return zero if the global settlement has not been executed yet after maturity.

Per the comment at Line 215, the design of the `_getMaturedCashValue` function is that it expects that if fCash has matured AND the fCash has not yet been settled, the `pr.supplyFactor` will be zero. In this case, the cash value will be zero.

https://github.com/sherlock-audit/2023-12-notional-update-5/blob/main/wrapped-fcash/contracts/wfCashBase.sol#L215

```
File: wfCashBase.sol
209:     function _getMaturedCashValue(uint256 fCashAmount) internal view
↪  returns (uint256) {
210:         if (!hasMatured()) return 0;
211:         // If the fCash has matured we use the cash balance instead.
212:         (uint16 currencyId, uint40 maturity) = getDecodedID();
213:         PrimeRate memory pr = NotionalV2.getSettlementRate(currencyId,
↪  maturity);
214:
215:         // fCash has not yet been settled
216:         if (pr.supplyFactor == 0) return 0;
..SNIP..
```

During the time window where the fCash has matured, and none of the accounts triggered an account settlement, the `_getMaturedCashValue` function at Line 33 below will return zero, which will result in the `totalAssets()` function returning zero.

https://github.com/sherlock-audit/2023-12-notional-update-5/blob/main/wrapped-fcash/contracts/wfCashERC4626.sol#L33

```
File: wfCashERC4626.sol
29:     function totalAssets() public view override returns (uint256) {
30:         if (hasMatured()) {
31:             // We calculate the matured cash value of the total supply of
 ↪  fCash. This is
32:             // not always equal to the cash balance held by the wrapper
 ↪  contract.
33:             uint256 primeCashValue = _getMaturedCashValue(totalSupply());
34:             require(primeCashValue < uint256(type(int256).max));
35:             int256 externalValue = NotionalV2.convertCashBalanceToExternal(
36:                 getCurrencyId(), int256(primeCashValue), true
37:             );
38:             return externalValue >= 0 ? uint256(externalValue) : 0;
..SNIP..
```

Impact

The `totalAssets()` function is utilized by key ERC4626 functions within the wrapper, such as the following functions. The side effects of this issue are documented below:

- `convertToAssets` (Impact = returned value is always zero assets regardless of the inputs)
- `convertToAssets` > `previewRedeem` (Impact = returned value is always zero assets regardless of the inputs)
- `convertToAssets` > `previewRedeem` > `maxWithdraw` (Impact = max withdrawal is always zero)
- `convertToShares` (Impact = Division by zero error, Revert)
- `convertToShares` > `previewWithdraw` (Impact = Revert)

In addition, any external protocol integrating with wfCash will be vulnerable within this time window as an invalid result (zero) is returned, or a revert might occur. For instance, any external protocol that relies on any of the above-affected functions for computing the withdrawal/minting amount or collateral value will be greatly impacted as the value before the maturity might be 10000, but it will temporarily reset to zero during this time window. Attackers could take advantage of this time window to perform malicious actions.

Code Snippet

https://github.com/sherlock-audit/2023-12-notional-update-5/blob/main/wrapped-fcash/contracts/wfCashBase.sol#L215

https://github.com/sherlock-audit/2023-12-notional-update-5/blob/main/wrapped-fcash/contracts/wfCashERC4626.sol#L33

Tool used

Manual Review

Recommendation

Document the unexpected behavior of the affected functions that could occur during the time windows when the fCash has matured but is not yet settled so that anyone who calls these functions is aware of them.

Discussion

jeffywu

Valid issue and I would consider increasing the severity to high. While this is not a risk for the leveraged vault framework (transactions revert inside the time window between fCash maturity and settlement rate initialization), this may cause severe issues for another protocol which relies on the `convertToAssets` method as a valuation method.

I believe the proper solution here is to `revert` while the settlement rate is not set.

nevillehuang

@jeffywu Since it is impossible to know how it will impact other protocols for wrong integrations, I will maintain is medium severity.

sherlock-admin

The protocol team fixed this issue in PR/commit https://github.com/notional-finance/wrapped-fcash/pull/13.

sherlock-admin

The Lead Senior Watson signed-off on the fix.

Issue M-9: Rebalance will be delayed due to revert

Source: https://github.com/sherlock-audit/2023-12-notional-update-5-judging/issues/44

Found by

xiaoming90

Summary

The rebalancing of unhealthy currencies will be delayed due to a revert, resulting in an excess of liquidity being lent out in the external market. This might affect the liquidity of the protocol, potentially resulting in withdrawal or liquidation having issues executed due to insufficient liquidity.

Vulnerability Detail

Assume that Notional supports 5 currencies (A, B, C, D, E), and the Gelato bot is configured to call the `checkRebalance` function every 30 minutes.

Assume that the current market condition is volatile. Thus, the inflow and outflow of assets to Notional, utilization rate, and available liquidity at AAVE change frequently. As a result, the target amount that should be externally lent out also changes frequently since the computation of this value relies on the spot market information.

At T1, when the Gelato bot calls the `checkRebalance()` view function, it returns that currencies A, B, and C are unhealthy and need to be rebalanced.

Shortly after receiving the execution payload from the `checkRebalance()`, the bot submits the rebalancing TX to the mempool for execution at T2.

When the rebalancing TX is executed at T3, one of the currencies (Currency A) becomes healthy. As a result, the require check at Line 326 will revert and the entire rebalancing transaction will be cancelled. Thus, currencies B and C that are still unhealthy at this point will not be rebalanced.

If this issue occurs frequently or repeatedly over a period of time, the rebalancing of unhealthy currencies will be delayed.

https://github.com/sherlock-audit/2023-12-notional-update-5/blob/main/contracts-v3/contracts/external/actions/TreasuryAction.sol#L326

```
File: TreasuryAction.sol
315:     function _rebalanceCurrency(uint16 currencyId, bool useCooldownCheck)
 ↪  private {
```

```
316:         RebalancingContextStorage memory context =
↪  LibStorage.getRebalancingContext()[currencyId];
317:         // Accrues interest up to the current block before any rebalancing
↪  is executed
318:         IPrimeCashHoldingsOracle oracle =
↪  PrimeCashExchangeRate.getPrimeCashHoldingsOracle(currencyId);
319:         PrimeRate memory pr =
↪  PrimeRateLib.buildPrimeRateStateful(currencyId);
320:
321:         bool hasCooldownPassed = _hasCooldownPassed(context);
322:         (bool isExternalLendingUnhealthy, OracleData memory oracleData,
↪  uint256 targetAmount) =
323:             _isExternalLendingUnhealthy(currencyId, oracle, pr);
324:
325:         // Cooldown check is bypassed when the owner updates the
↪  rebalancing targets
326:         if (useCooldownCheck) require(hasCooldownPassed ||
↪  isExternalLendingUnhealthy);
```

Impact

The rebalancing of unhealthy currencies will be delayed, resulting in an excess of liquidity being lent out to the external market. This might affect the liquidity of the protocol, potentially resulting in withdrawal or liquidation having issues executed due to insufficient liquidity.

Code Snippet

https://github.com/sherlock-audit/2023-12-notional-update-5/blob/main/contracts-v3/contracts/external/actions/TreasuryAction.sol#L326

Tool used

Manual Review

Recommendation

If one of the currencies becomes healthy when the rebalance TX is executed, consider skipping this currency and move on to execute the rebalance on the rest of the currencies that are still unhealthy.

```
function _rebalanceCurrency(uint16 currencyId, bool useCooldownCheck) private {
    RebalancingContextStorage memory context =
↪  LibStorage.getRebalancingContext()[currencyId];
```

```
    // Accrues interest up to the current block before any rebalancing is
↪   executed
    IPrimeCashHoldingsOracle oracle =
↪   PrimeCashExchangeRate.getPrimeCashHoldingsOracle(currencyId);
    PrimeRate memory pr = PrimeRateLib.buildPrimeRateStateful(currencyId);

    bool hasCooldownPassed = _hasCooldownPassed(context);
    (bool isExternalLendingUnhealthy, OracleData memory oracleData, uint256
↪   targetAmount) =
        _isExternalLendingUnhealthy(currencyId, oracle, pr);

    // Cooldown check is bypassed when the owner updates the rebalancing targets
-   if (useCooldownCheck) require(hasCooldownPassed ||
↪   isExternalLendingUnhealthy);
+   if (useCooldownCheck && !hasCooldownPassed && !isExternalLendingUnhealthy)
↪   return;
```

Discussion

sherlock-admin2

1 comment(s) were left on this issue during the judging contest.

takarez commented:

> valid because { valid medium findings due to the occurrence}

sherlock-admin

The protocol team fixed this issue in PR/commit https://github.com/notional-finance/contracts-v3/pull/32.

sherlock-admin

The Lead Senior Watson signed-off on the fix.

Issue M-10: Rebalance might be skipped even if the external lending is unhealthy

Source: https://github.com/sherlock-audit/2023-12-notional-update-5-judging/issues/46

Found by

xiaoming90

Summary

The deviation between the target and current lending amount (`offTargetPercentage`) will be underestimated due to incorrect calculation. As a result, a rebalancing might be skipped even if the existing external lending is unhealthy.

Vulnerability Detail

The formula used within the `_isExternalLendingUnhealthy` function below calculating the `offTargetPercentage` can be simplified as follows for the readability of this issue.

$$offTargetPercentage = \frac{|\ currentExternalUnderlyingLend - targetAmount\ |}{currentExternalUnderlyingLend + targetAmount} \times 100\%$$

Assume that the `targetAmount` is 100 and `currentExternalUnderlyingLend` is 90. The off-target percentage will be 5.26%, which is incorrect.

```
offTargetPercentage = abs(90 - 100) / (100 + 90) = 10 / 190 = 0.0526 = 5.26%
```

The correct approach is to calculate the off-target percentages as a ratio of the difference to the target:

$$offTargetPercentage = \frac{|\ currentExternalUnderlyingLend - targetAmount\ |}{targetAmount} \times 100\%$$

```
offTargetPercentage = abs(90 - 100) / (100) = 10 / 100 = 0.0526 = 10%
```

https://github.com/sherlock-audit/2023-12-notional-update-5/blob/main/contracts-v3/contracts/external/actions/TreasuryAction.sol#L425

```
File: TreasuryAction.sol
405:     function _isExternalLendingUnhealthy(
406:         uint16 currencyId,
407:         IPrimeCashHoldingsOracle oracle,
408:         PrimeRate memory pr
409:     ) internal view returns (bool isExternalLendingUnhealthy, OracleData
↪  memory oracleData, uint256 targetAmount) {
410:         oracleData = oracle.getOracleData();
411:
412:         RebalancingTargetData memory rebalancingTargetData =
413:
↪  LibStorage.getRebalancingTargets()[currencyId][oracleData.holding];
414:         PrimeCashFactors memory factors =
↪  PrimeCashExchangeRate.getPrimeCashFactors(currencyId);
415:         Token memory underlyingToken =
↪  TokenHandler.getUnderlyingToken(currencyId);
416:
417:         targetAmount = ExternalLending.getTargetExternalLendingAmount(
418:             underlyingToken, factors, rebalancingTargetData, oracleData, pr
419:         );
420:
421:         if (oracleData.currentExternalUnderlyingLend == 0) {
422:             // If this is zero then there is no outstanding lending.
423:             isExternalLendingUnhealthy = false;
424:         } else {
425:             uint256 offTargetPercentage =
↪  oracleData.currentExternalUnderlyingLend.toInt()
426:                 .sub(targetAmount.toInt()).abs()
427:                 .toUint()
428:                 .mul(uint256(Constants.PERCENTAGE_DECIMALS))
429:
↪  .div(targetAmount.add(oracleData.currentExternalUnderlyingLend));
430:
431:             // prevent rebalance if change is not greater than 1%,
↪  important for health check and avoiding triggering
432:             // rebalance shortly after rebalance on minimum change
433:             isExternalLendingUnhealthy =
434:                 (targetAmount < oracleData.currentExternalUnderlyingLend)
↪  && (offTargetPercentage > 0);
435:         }
436:     }
```

Impact

The deviation between the target and current lending amount (`offTargetPercentage`) will be underestimated by approximately half the majority of the time. As a result, a rebalance intended to remediate the unhealthy external lending might be skipped since the code incorrectly assumes that it has not hit the off-target percentage. External lending beyond the target will affect the liquidity of the protocol, potentially resulting in withdrawal or liquidation, having issues executed due to insufficient liquidity.

Code Snippet

https://github.com/sherlock-audit/2023-12-notional-update-5/blob/main/contracts-v3/contracts/external/actions/TreasuryAction.sol#L425

Tool used

Manual Review

Recommendation

Consider calculating the off-target percentages as a ratio of the difference to the target:

$$offTargetPercentage = \frac{|\ currentExternalUnderlyingLend - targetAmount\ |}{targetAmount} \times 100\%$$

Discussion

sherlock-admin2

1 comment(s) were left on this issue during the judging contest.

takarez commented:

> valid because { valid }

T-Woodward

Would classify this as low severity because it just depends on what you take offTargetPercentage to mean. You can still accomplish the same thing by setting your governance parameters according to what offTargetPercentage means with the current code, so I wouldn't really say this is an issue.

However, I think it should be classified as low severity rather than disputed because the watson's suggestion for how to define offTargetPercentage is more intuitive.

sherlock-admin

The protocol team fixed this issue in PR/commit https://github.com/notional-finance/contracts-v3/pull/29.

sherlock-admin

The Lead Senior Watson signed-off on the fix.

Issue M-11: getOracleData() maxExternalDeposit not accurate

Source: https://github.com/sherlock-audit/2023-12-notional-update-5-judging/issues/49

Found by

bin2chen

Summary

in `getOracleData()` The calculation of `maxExternalDeposit` lacks consideration for `reserve.accruedToTreasury`. This leads to `maxExternalDeposit` being too large, causing `Treasury.rebalance()` to fail.

Vulnerability Detail

in `getOracleData()`

```
    function getOracleData() external view override returns (OracleData memory
↪   oracleData) {
...
        (/* */, uint256 supplyCap) =
↪   IPoolDataProvider(POOL_DATA_PROVIDER).getReserveCaps(underlying);
        // Supply caps are returned as whole token values
        supplyCap = supplyCap * UNDERLYING_PRECISION;
        uint256 aTokenSupply =
↪   IPoolDataProvider(POOL_DATA_PROVIDER).getATokenTotalSupply(underlying);

        // If supply cap is zero, that means there is no cap on the pool
        if (supplyCap == 0) {
            oracleData.maxExternalDeposit = type(uint256).max;
        } else if (supplyCap <= aTokenSupply) {
            oracleData.maxExternalDeposit = 0;
        } else {
            // underflow checked as consequence of if / else statement
@>          oracleData.maxExternalDeposit = supplyCap - aTokenSupply;
        }
```

However, AAVE's restrictions are as follows: ValidationLogic.sol#L81-L88

```
  require(
    supplyCap == 0 ||
      ((IAToken(reserveCache.aTokenAddress).scaledTotalSupply() +
```

```
    ↪  uint256(reserve.accruedToTreasury)).rayMul(reserveCache.nextLiquidityIndex)
    ↪  + amount) <=
          supplyCap * (10 ** reserveCache.reserveConfiguration.getDecimals()),
        Errors.SUPPLY_CAP_EXCEEDED
      );
    }
```

The current implementation lacks subtraction of `uint256(reserve.accruedToTreasury)).rayMul(reserveCache.nextLiquidityIndex)`.

Impact

An overly large `maxExternalDeposit` may cause `rebalance()` to be unable to execute.

Code Snippet

https://github.com/sherlock-audit/2023-12-notional-update-5/blob/main/contracts-v3/contracts/external/pCash/AaveV3HoldingsOracle.sol#L160

Tool used

Manual Review

Recommendation

subtract `uint256(reserve.accruedToTreasury)).rayMul(reserveCache.nextLiquidityIndex)`

Discussion

sherlock-admin2

1 comment(s) were left on this issue during the judging contest.

takarez commented:

> valid because { valid }

sherlock-admin

The protocol team fixed this issue in PR/commit https://github.com/notional-finance/contracts-v3/pull/31.

sherlock-admin

The Lead Senior Watson signed-off on the fix.

Issue M-12: getTargetExternalLendingAmount() when targetUtilization == 0 no check whether enough externalUnderlyingAvailableForWithdraw

Source: https://github.com/sherlock-audit/2023-12-notional-update-5-judging/issues/51

Found by

bin2chen

Summary

in `getTargetExternalLendingAmount()` When `targetUtilization == 0`, it directly returns `targetAmount=0`. It lacks the judgment of whether there is enough `externalUnderlyingAvailableForWithdraw`. This may cause `_rebalanceCurrency()` to `revert` due to insufficient balance for `withdraw`.

Vulnerability Detail

when `setRebalancingTargets()` , we can setting all the targets to zero to immediately exit money it will call `_rebalanceCurrency() -> _isExternalLendingUnhealthy() -> getTargetExternalLendingAmount()`

```
    function getTargetExternalLendingAmount(
        Token memory underlyingToken,
        PrimeCashFactors memory factors,
        RebalancingTargetData memory rebalancingTargetData,
        OracleData memory oracleData,
        PrimeRate memory pr
    ) internal pure returns (uint256 targetAmount) {
        // Short circuit a zero target
@>      if (rebalancingTargetData.targetUtilization == 0) return 0;

....
        if (targetAmount < oracleData.currentExternalUnderlyingLend) {
            uint256 forRedemption = oracleData.currentExternalUnderlyingLend -
↪   targetAmount;
            if (oracleData.externalUnderlyingAvailableForWithdraw <
↪   forRedemption) {
                // increase target amount so that redemptions amount match
↪   externalUnderlyingAvailableForWithdraw
                targetAmount = targetAmount.add(
                    // unchecked - is safe here, overflow is not possible due to
↪   above if conditional
```

```
                    forRedemption -
↪   oracleData.externalUnderlyingAvailableForWithdraw
                );
            }
        }
```

When `targetUtilization==0`, it returns `targetAmount ==0`. It lacks the other judgments of whether the current `externalUnderlyingAvailableForWithdraw` is sufficient. Exceeding `externalUnderlyingAvailableForWithdraw` may cause `_rebalanceCurrency()` to revert.

For example: `currentExternalUnderlyingLend = 100` `externalUnderlyingAvailableForWithdraw = 99` If `targetUtilization` is modified to `0`, then `targetAmount` should be `1`, not `0`. `0` will cause an error due to insufficient available balance for withdrawal.

So, it should still try to withdraw as much deposit as possible first, wait for replenishment, and then withdraw the remaining deposit until the deposit is cleared.

Impact

A too small `targetAmount` may cause AAVE withdraw to fail, thereby causing the inability to `setRebalancingTargets()` to fail.

Code Snippet

https://github.com/sherlock-audit/2023-12-notional-update-5/blob/main/contracts-v3/contracts/internal/balances/ExternalLending.sol#L44

Tool used

Manual Review

Recommendation

Remove `targetUtilization == 0` directly returning 0.

The subsequent logic of the method can handle `targetUtilization == 0` normally and will not cause an error.

```
    function getTargetExternalLendingAmount(
        Token memory underlyingToken,
        PrimeCashFactors memory factors,
        RebalancingTargetData memory rebalancingTargetData,
        OracleData memory oracleData,
        PrimeRate memory pr
```

```
    ) internal pure returns (uint256 targetAmount) {
        // Short circuit a zero target
-       if (rebalancingTargetData.targetUtilization == 0) return 0;
```

Discussion

sherlock-admin2

1 comment(s) were left on this issue during the judging contest.

takarez commented:

> valid because {valid medium and a duplicate of 052}

nevillehuang

@jeffywu Is the impact of this significant and can compound to cause a material enough loss due to inability to rebalance? Also who is responsible for maintaining a high enough `externalUnderlyingAvailableForWithdraw`?

jeffywu

Hmm, I guess this would be tripped if governance tries to set the target utilization to zero. Likely, this occurs if we want to pull funds very quickly from an external lending protocol. The check is important to ensure that we pull as much funds as we can without reverting.

This issue would significantly hamper our ability to get out of a lending protocol quickly. A higher severity could be justified.

sherlock-admin

The protocol team fixed this issue in PR/commit https://github.com/notional-finance/contracts-v3/pull/26.

sherlock-admin

The Lead Senior Watson signed-off on the fix.

Issue M-13: getTargetExternalLendingAmount() targetAmount may far less than the correct value

Source: https://github.com/sherlock-audit/2023-12-notional-update-5-judging/issues/52

Found by

bin2chen

Summary

When calculating `ExternalLending.getTargetExternalLendingAmount()`, it restricts `targetAmount` greater than `oracleData.maxExternalDeposit`. However, it does not take into account that `oracleData.maxExternalDeposit` includes the protocol deposit `currentExternalUnderlyingLend` This may result in the returned quantity being far less than the correct quantity.

Vulnerability Detail

in `getTargetExternalLendingAmount()` It restricts `targetAmount` greater than `oracleData.maxExternalDeposit`.

```
    function getTargetExternalLendingAmount(
        Token memory underlyingToken,
        PrimeCashFactors memory factors,
        RebalancingTargetData memory rebalancingTargetData,
        OracleData memory oracleData,
        PrimeRate memory pr
    ) internal pure returns (uint256 targetAmount) {
...

        targetAmount = SafeUint256.min(
            // totalPrimeCashInUnderlying and totalPrimeDebtInUnderlying are in
↪   8 decimals, convert it to native
            // token precision here for accurate comparison. No underflow
↪   possible since targetExternalUnderlyingLend
            // is floored at zero.

↪   uint256(underlyingToken.convertToExternal(targetExternalUnderlyingLend)),
            // maxExternalUnderlyingLend is limit enforced by setting
↪   externalWithdrawThreshold
            // maxExternalDeposit is limit due to the supply cap on external
↪   pools
```

```
@>          SafeUint256.min(maxExternalUnderlyingLend,
↪  oracleData.maxExternalDeposit)
        );
```

this is : `targetAmount = min(targetExternalUnderlyingLend, maxExternalUnderlyingLend, oracleData.maxExternalDeposit)`

The problem is that when calculating `oracleData.maxExternalDeposit`, it does not exclude the existing deposit `currentExternalUnderlyingLend` of the current protocol.

For example: `currentExternalUnderlyingLend = 100` `targetExternalUnderlyingLend = 100` `maxExternalUnderlyingLend = 10000` `oracleData.maxExternalDeposit = 0` (All AAVE deposits include the current deposit `currentExternalUnderlyingLend`)

If according to the current calculation result: `targetAmount=0`, this will result in needing to withdraw `100`. (currentExternalUnderlyingLend - targetAmount)

In fact, only when the calculation result needs to increase the `deposit` (targetAmount > currentExternalUnderlyingLend), it needs to be restricted by `maxExternalDeposit`.

The correct one should be neither deposit nor withdraw, that is, `targetAmount=currentExternalUnderlyingLend = 100`.

Impact

A too small `targetAmount` will cause the withdrawal of deposits that should not be withdrawn, damaging the interests of the protocol.

Code Snippet

https://github.com/sherlock-audit/2023-12-notional-update-5/blob/main/contracts-v3/contracts/internal/balances/ExternalLending.sol#L89C1-L97C11

Tool used

Manual Review

Recommendation

Only when `targetAmount > currentExternalUnderlyingLend` is a deposit needed, it should be considered that it cannot exceed `oracleData.maxExternalDeposit`

```
    function getTargetExternalLendingAmount(
        Token memory underlyingToken,
        PrimeCashFactors memory factors,
```

```
        RebalancingTargetData memory rebalancingTargetData,
        OracleData memory oracleData,
        PrimeRate memory pr
    ) internal pure returns (uint256 targetAmount) {
...

-        targetAmount = SafeUint256.min(
-            // totalPrimeCashInUnderlying and totalPrimeDebtInUnderlying are in
 ↪  8 decimals, convert it to native
-            // token precision here for accurate comparison. No underflow
 ↪  possible since targetExternalUnderlyingLend
-            // is floored at zero.
-
 ↪  uint256(underlyingToken.convertToExternal(targetExternalUnderlyingLend)),
-            // maxExternalUnderlyingLend is limit enforced by setting
 ↪  externalWithdrawThreshold
-            // maxExternalDeposit is limit due to the supply cap on external
 ↪  pools
-            SafeUint256.min(maxExternalUnderlyingLend,
 ↪  oracleData.maxExternalDeposit)
-        );

+        targetAmount = SafeUint256.min(uint256(underlyingToken.convertToExternal
 ↪  (targetExternalUnderlyingLend)),maxExternalUnderlyingLend);
+        if (targetAmount > oracleData.currentExternalUnderlyingLend) { //when
 ↪  deposit ,  must check maxExternalDeposit
+            uint256 forDeposit = targetAmount -
 ↪  oracleData.currentExternalUnderlyingLend;
+            if (forDeposit > oracleData.maxExternalDeposit) {
+                targetAmount = targetAmount.sub(
+                    forDeposit - oracleData.maxExternalDeposit
+                );
+            }
+        }
```

Discussion

sherlock-admin2

1 comment(s) were left on this issue during the judging contest.

takarez commented:

> valid because { This is valid medium ; watson explained how an incorrect value can be returned}

sherlock-admin

The protocol team fixed this issue in PR/commit https://github.com/notional-finance/contracts-v3/pull/25.

sherlock-admin

The Lead Senior Watson signed-off on the fix.

Issue M-14: `wfCashERC4626`

Source: https://github.com/sherlock-audit/2023-12-notional-update-5-judging/issues/58

Found by

eol

Summary

The `wfCash` vault is credited less prime cash than the `wfCash` it mints to the depositor when its underlying asset is a fee-on-transfer token. This leads to the vault being insolvent because it has issued more shares than can be redeemed.

Vulnerability Detail

When minting `wfCash` shares more than the `maxFCash` available, the `wfCashERC4626` vault lends the deposited assets at 0% interest. The assets are deposited by the vault into Notional and the depositor gets 1:1 `wfCash` in return. This works fine for assets that are not fee-on-transfer tokens.
https://github.com/sherlock-audit/2023-12-notional-update-5/blob/main/wrapped-fcash/contracts/wfCashLogic.sol#L60-L68

```
if (maxFCash < fCashAmount) {
    // NOTE: lending at zero
    uint256 fCashAmountExternal = fCashAmount * precision /
↪   uint256(Constants.INTERNAL_TOKEN_PRECISION);
    require(fCashAmountExternal <= depositAmountExternal);

    // NOTE: Residual (depositAmountExternal - fCashAmountExternal) will be
↪   transferred
    // back to the account
    NotionalV2.depositUnderlyingToken{value: msgValue}(address(this),
↪   currencyId, fCashAmountExternal);
} else if (isETH || hasTransferFee || getCashBalance() > 0) {
```

For fee-on-transfer tokens, the vault is credited prime cash based on the actual amount it received, which is `deposit amount - transfer fee`.
https://github.com/sherlock-audit/2023-12-notional-update-5/blob/main/contracts-v3/contracts/internal/balances/TokenHandler.sol#L204-L214

```
} else {
    // In the case of deposits, we use a balance before and after check
    // to ensure that we record the proper balance change.
```

```
    actualTransferExternal = GenericToken.safeTransferIn(
        underlying.tokenAddress, account, underlyingExternalDeposit
    ).toInt();
}

netPrimeSupplyChange = _postTransferPrimeCashUpdate(
    account, currencyId, actualTransferExternal, underlying, primeRate
);
```

However, the vault mints `fCashAmount` of shares to the depositor. https://github.com/sherlock-audit/2023-12-notional-update-5/blob/main/wrapped-fcash/contracts/wfCashLogic.sol#L84-L85

```
// Mints ERC20 tokens for the receiver
_mint(receiver, fCashAmount);
```

In the case of lending at 0% interest, `fCashAmount` is equal to `depositAmount` but at 1e8 precision.

To simplify the example, let us assume that there are no other depositors. When the sole depositor redeems all their `wfCash` shares at maturity, they will be unable to redeem all their shares because the `wfCash` vault does not hold enough prime cash.

Impact

Although the example used to display the vulnerability is for the case of lending at 0% interest, the issue exists for minting any amount of shares.

The `wfCashERC4626` vault will become insolvent and unable to buy back all shares. The larger the total amount deposited, the larger the deficit. The deficit is equal to the transfer fee. Given a total deposit amount of 100M USDT and a transfer fee of 2% (assuming a transfer fee was set and enabled for USDT), 2M USDT will be the deficit.

The last depositors to redeem their shares will be shouldering the loss.

Code Snippet

https://github.com/sherlock-audit/2023-12-notional-update-5/blob/main/wrapped-fcash/contracts/wfCashLogic.sol#L60-L68 https://github.com/sherlock-audit/2023-12-notional-update-5/blob/main/contracts-v3/contracts/internal/balances/TokenHandler.sol#L204-L214 https://github.com/sherlock-audit/2023-12-notional-update-5/blob/main/wrapped-fcash/contracts/wfCashLogic.sol#L84-L85

Tool used

Manual Review

Recommendation

Consider adding the following:

1. A flag in `wfCashERC4626` that signals that the vault's asset is a fee-on-transfer token.
2. In `wfCashERC4626._previewMint()` and `wfCashERC46262._previewDeposit`, all calculations related to `assets` should account for the transfer fee of the token.

Discussion

sherlock-admin2

1 comment(s) were left on this issue during the judging contest.

takarez commented:

> vlaid because { This is valid and a duplicate of 016 due to the same underlying cause of zero lending }

jeffywu

I think the issue is valid, although the contention that the vault becomes insolvent is not completely true. When lending at 0% interest the vault also accrues variable rate interest, which may or may not accrue sufficient value to recover the loss from the transfer fee.

Agree that some sort of flag here would be appropriate to ensure that these tokens do not trigger an immediate loss is appropriate.

jeffywu

Fix will be that we do not allow tokens with fee on transfer to perform lend at zero.

gjaldon

Escalate

> I think the issue is valid, although the contention that the vault becomes insolvent is not completely true. When lending at 0% interest the vault also accrues variable rate interest, which may or may not accrue sufficient value to recover the loss from the transfer fee.

I think this is a High because the vault still stays insolvent even when the vault accrues variable rate interest. This is because any variable rate interest accrual is only applied to the prime cash's exchange rate (like a cToken.)

The issue here is that the total `wfCash` minted is supposed to be 1:1 with the vault's primeCash balance in Notional. However, there is more `wfCash` minted than the vault's prime cash balance in Notional because the Vault's `wfCash` minting does not subtract the transfer fees when computing for the amount to mint, which leads to the discrepancy.

Any variable interest rate accrual will not increase the prime cash balance of the Vault because it is non-rebasing. That means that the Vault's prime cash balance in Notional will not increase due to accrual and it will remain less than the Vault's total `wfCash` minted and the vault will remain insolvent.

For example: Vault `wfCash` supply - 1000 Vault's prime cash balance in Notional - 975 (1000 - 2.5% transfer fee) Variable interest rate accrual - increases the exchange rate from, for example, 110% to 115%

Total `wfCash` that can be redeemed is still capped at 975 even with the variable interest rate accrual. Vault remains insolvent because not all shares can be redeemed.

sherlock-admin2

> Escalate
>
> > I think the issue is valid, although the contention that the vault becomes insolvent is not completely true. When lending at 0% interest the vault also accrues variable rate interest, which may or may not accrue sufficient value to recover the loss from the transfer fee.
>
> I think this is a High because the vault still stays insolvent even when the vault accrues variable rate interest. This is because any variable rate interest accrual is only applied to the prime cash's exchange rate (like a cToken.)
>
> The issue here is that the total `wfCash` minted is supposed to be 1:1 with the vault's primeCash balance in Notional. However, there is more `wfCash` minted than the vault's prime cash balance in Notional because the Vault's `wfCash` minting does not subtract the transfer fees when computing for the amount to mint, which leads to the discrepancy.
>
> Any variable interest rate accrual will not increase the prime cash balance of the Vault because it is non-rebasing. That means that the Vault's prime cash balance in Notional will not increase due to accrual and it will remain less than the Vault's total `wfCash` minted and the vault will remain insolvent.
>
> For example: Vault `wfCash` supply - 1000 Vault's prime cash balance in Notional - 975 (1000 - 2.5% transfer fee) Variable interest rate accrual - increases the exchange rate from, for example, 110% to 115%

> Total `wfCash` that can be redeemed is still capped at 975 even with the variable interest rate accrual. Vault remains insolvent because not all shares can be redeemed.

The escalation could not be created because you are not exceeding the escalation threshold.

You can view the required number of additional valid issues/judging contest payouts in your Profile page, in the Sherlock webapp.

JeffCX

Escalate

> I think the issue is valid, although the contention that the vault becomes insolvent is not completely true. When lending at 0% interest the vault also accrues variable rate interest, which may or may not accrue sufficient value to recover the loss from the transfer fee.

I think this is a High because the vault still stays insolvent even when the vault accrues variable rate interest. This is because any variable rate interest accrual is only applied to the prime cash's exchange rate (like a cToken.)

The issue here is that the total wfCash minted is supposed to be 1:1 with the vault's primeCash balance in Notional. However, there is more wfCash minted than the vault's prime cash balance in Notional because the Vault's wfCash minting does not subtract the transfer fees when computing for the amount to mint, which leads to the discrepancy.

Any variable interest rate accrual will not increase the prime cash balance of the Vault because it is non-rebasing. That means that the Vault's prime cash balance in Notional will not increase due to accrual and it will remain less than the Vault's total wfCash minted and the vault will remain insolvent.

For example: Vault wfCash supply - 1000 Vault's prime cash balance in Notional - 975 (1000 - 2.5% transfer fee) Variable interest rate accrual - increases the exchange rate from, for example, 110% to 115%

Total wfCash that can be redeemed is still capped at 975 even with the variable interest rate accrual. Vault remains insolvent because not all shares can be redeemed.

sherlock-admin2

> Escalate
>
> > I think the issue is valid, although the contention that the vault becomes insolvent is not completely true. When lending at 0% interest the vault also accrues variable rate interest, which may

> or may not accrue sufficient value to recover the loss from the transfer fee.
>
> I think this is a High because the vault still stays insolvent even when the vault accrues variable rate interest. This is because any variable rate interest accrual is only applied to the prime cash's exchange rate (like a cToken.)
>
> The issue here is that the total wfCash minted is supposed to be 1:1 with the vault's primeCash balance in Notional. However, there is more wfCash minted than the vault's prime cash balance in Notional because the Vault's wfCash minting does not subtract the transfer fees when computing for the amount to mint, which leads to the discrepancy.
>
> Any variable interest rate accrual will not increase the prime cash balance of the Vault because it is non-rebasing. That means that the Vault's prime cash balance in Notional will not increase due to accrual and it will remain less than the Vault's total wfCash minted and the vault will remain insolvent.
>
> For example: Vault wfCash supply - 1000 Vault's prime cash balance in Notional - 975 (1000 - 2.5% transfer fee) Variable interest rate accrual - increases the exchange rate from, for example, 110% to 115%
>
> Total wfCash that can be redeemed is still capped at 975 even with the variable interest rate accrual. Vault remains insolvent because not all shares can be redeemed.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

You may delete or edit your escalation comment anytime before the 48-hour escalation window closes. After that, the escalation becomes final.

jeffywu

I won't opine the severity of the issue but I will say the understanding of the prime cash balance is not entirely correct in the escalation comment:

fCash is denominated in underlying (i.e. 1 fETH = 1 ETH at maturity) while prime cash is a non-rebasing token with a monotonically increasing exchange rate to ETH. If we mint 1 fETH "share" while lending at zero interest but only deposit 0.975 ETH worth of pCash prior to maturity...it would be appropriate to say that the vault is insolvent at this moment in time.

However, if the 0.975 ETH worth of pCash accrues interest over X days and is now worth 1 ETH and no other deposits / withdraws occur, the vault is no longer insolvent. The rebasing / non-rebasing nature of pCash does not affect the

solvency issue, the vault has full access to the underlying value of the pCash to accommodate withdraws.

gjaldon

@jeffywu isn't it that a positive cash balance is non-rebasing and any interest accruals do not change the account's cash balance? If I'm not mistaken, any interest accruals only change the prime cash factors: ref: https://github.com/sherlock-audit/2023-12-notional-update-5/blob/main/contracts-v3/contracts/internal/pCash/PrimeCashExchangeRate.sol#L589-L613

```
function getPrimeCashRateStateful(
    uint16 currencyId,
    uint256 blockTime
) internal returns (PrimeRate memory rate) {
    PrimeCashFactors memory factors = getPrimeCashFactors(currencyId);

    // Only accrue if the block time has increased
    if (factors.lastAccrueTime < blockTime) {
        uint256 primeSupplyToReserve;
        uint256 currentUnderlyingValue = getTotalUnderlyingStateful(currencyId);
        (factors, primeSupplyToReserve) = _updatePrimeCashScalars(
            currencyId, factors, currentUnderlyingValue, blockTime
        );
        _setPrimeCashFactorsOnAccrue(currencyId, primeSupplyToReserve, factors);
    } else {
        require(factors.lastAccrueTime == blockTime); // dev: revert invalid
↪   blocktime
    }

    rate = PrimeRate({
        supplyFactor: factors.supplyScalar.mul(factors.underlyingScalar).toInt(),
        debtFactor: factors.debtScalar.mul(factors.underlyingScalar).toInt(),
        oracleSupplyRate: factors.oracleSupplyRate
    });
}
```

`_updatePrimeCashScalars()` computes the interest accrual and updates the factors. `PrimeRate.supplyFactor` is the exchange rate that's used to compute for the total ETH that can be withdrawn.

Given a Vault that had deposited 1000 tokens into pCash and the following are true:

Vault has 1000 wfCash total supply Vault has pCash balance of 975 in Notional (1000 - 2.5% transfer fee) PR.supplyFactor is 100%

Let's say the PR.supplyFactor has increased to 120% due to interest accrual. When the Vault withdraws, it calls `AccountAction.withdraw()` which also ends up calling

BalanceHandler._finalize().

```
transferAmountExternal = TokenHandler.withdrawPrimeCash(
    account,
    receiver,
    balanceState.currencyId,
    balanceState.primeCashWithdraw,
    balanceState.primeRate,
    withdrawWrapped // if true, withdraws ETH as WETH
);

{
    // No changes to total cash after this point
    int256 totalCashChange =
↪   balanceState.netCashChange.add(balanceState.primeCashWithdraw);

    if (
        checkAllowPrimeBorrow &&
        totalCashChange < 0 &&
        balanceState.storedCashBalance.add(totalCashChange) < 0
    ) {
        // ... snip ...
        require(accountContext.allowPrimeBorrow, "No Prime Borrow");
        checkDebtCap = true;
    }

    if (totalCashChange != 0) {
        balanceState.storedCashBalance =
↪   balanceState.storedCashBalance.add(totalCashChange);
        mustUpdate = true;
    }
}
```

In `BalanceHandler.finalize()`, `balanceState.primeCashWithdraw` is the amount being withdrawn, which is -1000 (all the wfCash supply.) It is negative since withdrawal amounts are turned negative to represent decrease in cash balance. `totalCashChange` will be equal to `balanceState.primeCashWithdraw`, which is -1000. `balanceState.storedCashBalance` will be the Vault's cash balance of 975. Adding both will result in -25 and will lead to the `"No Prime Borrow"` revert since there is not enough cash balance for the withdrawal and prime borrowing is not enabled for the vault.

In TokenHandler.withdrawPrimeCash() is where the exchange rate is applied so that we get the principal plus the interest accrual.

```
// TokenHandler.withdrawPrimeCash()
```

```
    Token memory underlying = getUnderlyingToken(currencyId);
    netTransferExternal = convertToExternal(
        underlying,
        primeRate.convertToUnderlying(primeCashToWithdraw)
    );
```

primeRate.convertToUnderlying converts the prime cash value (principal) into its accrued value (principal + interest accrual.)

```
function convertToUnderlying(
    PrimeRate memory pr,
    int256 primeCashBalance
) internal pure returns (int256) {
    int256 result =
↪   primeCashBalance.mul(pr.supplyFactor).div(Constants.DOUBLE_SCALAR_PRECISION);
    return primeCashBalance < 0 ? SafeInt256.min(result, -1) : result;
}
```

We see above how `primeRate.supplyFactor`, which is an exchange rate that represents interest accrual for lending, is multiplied to the prime cash balance.

I think the details above show that the prime cash balance (`balanceState.storedCashBalance`) does not increase due to interest accrual. With the example above, it will stay at 975 even when interest accrues. This is because interest accruals are only applied to the Prime Cash factors and do not modify every account's cash balances, which would be expensive.

Am I missing anything @jeffywu?

jeffywu

Yeah, but 0.975 pETH units can be worth 1 ETH which is what is required for the vault to be solvent. If you call withdraw and expect to get 1 ETH, the vault will be able to satisfy the request and therefore it would be solvent.

gjaldon

I see. I think the user would expect interest accrual to make their 1 ETH larger than 1 ETH. Also, even though 0.975 pETH can become 1 ETH eventually due to variable interest accrual, there is still the 0.025 pETH that can not be redeemed. A Vault is insolvent when all of its shares can not be redeemed. When there are multiple depositors, this can lead to situations where the last depositor to withdraw will be unable to withdraw any assets.

jeffywu

Well the user could only expect up to 1 ETH because they have a claim on 1 fETH.

They are not lending variable they are lending fixed

On Mon, Feb 5, 2024 at 4:06 PM G @.***> wrote:

> I see. I think the user would expect interest accrual to make their 1 ETH larger than 1 ETH. Also, even though 0.975 pETH can become 1 ETH eventually due to variable interest accrual, there is still the 0.025 pETH that can not be redeemed. A Vault is insolvent when all of its shares can not be redeemed. When there are multiple depositors, this can lead to situations where the last depositor to withdraw will be unable to withdraw any assets.
>
> — Reply to this email directly, view it on GitHub https://github.com/sherlock-audit/2023-12-notional-update-5-judging/issues/58#issuecomment-1928538045, or unsubscribe https://github.com/notifications/unsubscribe-auth/AAHOUGSOW62TRY3ZX3MLMDTYSFXX5AVCNFSM6AAAAABCAQ65JCVHI2DSMVQWIX3LMV43OSLTON2WKQ3PNVWWK3TUHMYTSMRYGUZTQMBUGU . You are receiving this because you were mentioned.Message ID: <sherlock-audit/2023-12-notional-update-5-judging/issues/58/1928538045@ github.com>

gjaldon

Fair point. I think you are looking at it from the perspective of calling `Vault.withdraw()` where the user sets the amount of assets to be withdrawn. When using `Vault.redeem()`, the user sets the amount of shares to be redeemed. In this case, the amount to be redeemed would be the total shares of 1000. This is what I mean by insolvency, since only up to 975 shares can be redeemed and not all the shares are redeemable.

Thank you for the time and the discussion @jeffywu. I appreciate it and will do my best to carry over all my learnings to future Notional contests.

Evert0x

Planning to reject escalation and keep issue state as is @gjaldon

Evert0x

Result: Medium Unique

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- JEFFCX: rejected

sherlock-admin

The protocol team fixed this issue in PR/commit https://github.com/notional-finance/wrapped-fcash/pull/17.

sherlock-admin

The Lead Senior Watson signed-off on the fix.

Issue M-15: ExternalLending

Source: https://github.com/sherlock-audit/2023-12-notional-update-5-judging/issues/65

Found by

eol

Summary

When the Treasury rebalances and has to redeem aTokens from AaveV3, it checks that the actual amount withdrawn is greater than or equal to the set `withdrawAmount`. This check will always fail for fee-on-transfer tokens since the `withdrawAmount` does not account for the transfer fee.

Vulnerability Detail

When the Treasury rebalances and has to redeem aWETH from AaveV3 it executes calls that were encoded in `AaveV3HoldingsOracle._getRedemptionCalldata()`: https://github.com/sherlock-audit/2023-12-notional-update-5/blob/main/contracts-v3/contracts/external/pCash/AaveV3HoldingsOracle.sol#L61-L81

```
address[] memory targets = new address[](UNDERLYING_IS_ETH ? 2 : 1);
bytes[] memory callData = new bytes[](UNDERLYING_IS_ETH ? 2 : 1);
targets[0] = LENDING_POOL;
callData[0] = abi.encodeWithSelector(
    ILendingPool.withdraw.selector, underlyingToken, withdrawAmount,
↪   address(NOTIONAL)
);

if (UNDERLYING_IS_ETH) {
    // Aave V3 returns WETH instead of native ETH so we have to unwrap it here
    targets[1] = address(Deployments.WETH);
    callData[1] = abi.encodeWithSelector(WETH9.withdraw.selector,
↪   withdrawAmount);
}

data = new RedeemData[](1);
// Tokens with less than or equal to 8 decimals sometimes have off by 1 issues
↪   when depositing
// into Aave V3. Aave returns one unit less than has been deposited. This
↪   adjustment is applied
// to ensure that this unit of token is credited back to prime cash holders
↪   appropriately.
```

```
uint8 rebasingTokenBalanceAdjustment = UNDERLYING_DECIMALS <= 8 ? 1 : 0;
data[0] = RedeemData(
    targets, callData, withdrawAmount, ASSET_TOKEN,
↪   rebasingTokenBalanceAdjustment
);
```

Note that the third field in the `RedeemData` struct is the `expectedUnderlying` field which is set to the `withdrawAmount` and that `withdrawAmount` is a value greater than zero.

Execution of redemption is done in `ExternalLending.executeMoneyMarketRedemptions()`. https://github.com/sherlock-audit/2023-12-notional-update-5/blob/main/contracts-v3/contracts/internal/balances/ExternalLending.sol#L163-L172

```
for (uint256 j; j < data.targets.length; j++) {
    GenericToken.executeLowLevelCall(data.targets[j], 0, data.callData[j]);
}

// Ensure that we get sufficient underlying on every redemption
uint256 newUnderlyingBalance = TokenHandler.balanceOf(underlyingToken,
↪   address(this));
uint256 underlyingBalanceChange = newUnderlyingBalance.sub(oldUnderlyingBalance);
// If the call is not the final redemption, then expectedUnderlying should
// be set to zero.
require(data.expectedUnderlying <= underlyingBalanceChange);
```

The `require` statement expects that the change in token balance is always greater than or equal to `data.expectedUnderlying`. However, `data.expectedUnderlying` was calculated with: https://github.com/sherlock-audit/2023-12-notional-update-5/blob/main/contracts-v3/contracts/external/actions/TreasuryAction.sol#L452

```
redeemAmounts[0] = currentAmount - targetAmount;
```

It does not account for transfer fees. In effect, that check will always revert when the underlying being withdrawn is a fee-on-transfer token.

Impact

Rebalancing will always fail when redemption of a fee-on-transfer token is executed. This will also break withdrawals of prime cash from Notional of fee-on-transfer tokens that require redemption from AaveV3. https://github.com/sherlock-audit/2023-12-notional-update-5/blob/main/contracts-v3/contracts/internal/balances/TokenHandler.sol#L243-L244

```
uint256 withdrawAmount = uint256(netTransferExternal.neg());
ExternalLending.redeemMoneyMarketIfRequired(currencyId, underlying,
↪  withdrawAmount);
```

This means that these tokens can only be deposited into AaveV3 but can never redeemed. This can lead to insolvency of the protocol.

Code Snippet

https://github.com/sherlock-audit/2023-12-notional-update-5/blob/main/contracts-v3/contracts/external/pCash/AaveV3HoldingsOracle.sol#L61-L81
https://github.com/sherlock-audit/2023-12-notional-update-5/blob/main/contracts-v3/contracts/external/actions/TreasuryAction.sol#L452
https://github.com/sherlock-audit/2023-12-notional-update-5/blob/main/contracts-v3/contracts/internal/balances/ExternalLending.sol#L163-L172
https://github.com/sherlock-audit/2023-12-notional-update-5/blob/main/contracts-v3/contracts/internal/balances/TokenHandler.sol#L243-L244

Tool used

Manual Review

Recommendation

When computing for the `withdrawAmount / data.expectedUnderlying`, it should account for the transfer fees. The pseudocode for the computation may look like so:

Discussion

sherlock-admin2

1 comment(s) were left on this issue during the judging contest.

takarez commented:

> valid because { valid medium findings}

jeffywu

While this is true, I think the protocol would simply opt to not have external lending for fee on transfer tokens. Since this feature is turned on or off by the protocol I don't see how this warrants a high severity.

nevillehuang

@jeffywu would this be a duplicate of #58? If I'm not wrong the fix seems different so could be separate.

jeffywu

No, it is separate code.

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.