SHERLOCK

SHERLOCK SECURITY REVIEW FOR

Prepared for:	Notional
Prepared by:	Sherlock
Lead Security Expert:	xiaoming90 + 0xleastwood
Dates Audited:	March 27 - May 15, 2023
Prepared on:	August 31, 2023

Introduction

Earn fixed income on your crypto or borrow at fixed rates for up to one year with Notional - DeFi's top fixed rate protocol.

Scope

Repository: notional-finance/contracts-v2

Branch: feature/notional-v3-sherlock-audit

Commit: b20a45c912785fab5f2b62992e5260f44dbae197

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
21	11

Issues not fixed or acknowledged

Medium	High
0	0

Security experts who found valid issues

xiaoming90
0xleastwood
bin2chen
ShadowForce
mstpr-brainbot
chaduke
lemonmon
0x00ffDa
iglyx
0xGoodess

Issue H-1: claimCOMPAndTransfer() COMP may be locked into the contract

Source: https://github.com/sherlock-audit/2023-03-notional-judging/issues/168

Found by

bin2chen, lemonmon, xiaoming90 + 0xleastwood

Summary

Malicious users can keep front-run `claimCOMPAndTransfer()` to trigger `COMPTROLLER.claimComp()` first, causing `netBalance` in `claimCOMPAndTransfer()` to be 0 all the time, resulting in `COMP` not being transferred out and locked in the contract

Vulnerability Detail

`claimCOMPAndTransfer()` use for "Claims COMP incentives earned and transfers to the treasury manager contract" The code is as follows:

```
function claimCOMPAndTransfer(address[] calldata cTokens)
    external
    override
    onlyManagerContract
    nonReentrant
    returns (uint256)
{
    uint256 balanceBefore = COMP.balanceOf(address(this));
    COMPTROLLER.claimComp(address(this), cTokens);
    uint256 balanceAfter = COMP.balanceOf(address(this));

    // NOTE: the onlyManagerContract modifier prevents a transfer to address(0)
↪   here
    uint256 netBalance = balanceAfter.sub(balanceBefore);   //<-------@only
↪   transfer out `netBalance`
    if (netBalance > 0) {
        COMP.safeTransfer(msg.sender, netBalance);
    }

    // NOTE: TreasuryManager contract will emit a COMPHarvested event
    return netBalance;
```

From the above code, we can see that this method only turns out the difference value `netBalance` But `COMPTROLLER.claimComp()` can be called by anyone, if there is a

malicious user front-run this transcation to triggers `COMPTROLLER.claimComp()` first This will cause the`netBalance` to be 0 all the time, resulting in `COMP` not being transferred out and being locked in the contract.

The following code is from `Comptroller.sol`

https://github.com/compound-finance/compound-protocol/blob/master/contracts/Comptroller.sol

```
function claimComp(address holder, CToken[] memory cTokens) public {
↪  //<----------anyone can call it
    address[] memory holders = new address[](1);
    holders[0] = holder;
    claimComp(holders, cTokens, true, true);
}
```

Impact

`COMP` may be locked into the contract

Code Snippet

https://github.com/sherlock-audit/2023-03-notional/blob/main/contracts-v2/contracts/external/actions/TreasuryAction.sol#L118C4-L123

Tool used

Manual Review

Recommendation

Transfer all balances, not using `netBalance`

Discussion

Jiaren-tang

.

Jiaren-tang

Escalate for 10 USDC. do not think this can be a high vulnerability because of the following reason

this is a medium because first, the protocol can use flashbot to avoid frontrunning

this is loss of reward not lose of user fund

this griefing attack has no gain from attacker at all

https://docs.sherlock.xyz/audits/judging/judging

> Medium: There is a viable scenario (even if unlikely) that could cause the protocol to enter a state where a material amount of funds can be lost. The attack path is possible with assumptions that either mimic on-chain conditions or reflect conditions that have a reasonable chance of becoming true in the future. The more expensive the attack is for an attacker, the less likely it will be included as a Medium (holding all other factors constant). The vulnerability must be something that is not considered an acceptable risk by a reasonable protocol team.

we consider the attack is cost high becaues the attacker needs to monitor the pending claim transaction from notional side while has no economic gain

sherlock-admin

> Escalate for 10 USDC. do not think this can be a high vulnerability because of the following reason
>
> this is a medium because first, the protocol can use flashbot to avoid frontrunning
>
> this is loss of reward not lose of user fund
>
> this griefing attack has no gain from attacker at all
>
> https://docs.sherlock.xyz/audits/judging/judging
>
> > Medium: There is a viable scenario (even if unlikely) that could cause the protocol to enter a state where a material amount of funds can be lost. The attack path is possible with assumptions that either mimic on-chain conditions or reflect conditions that have a reasonable chance of becoming true in the future. The more expensive the attack is for an attacker, the less likely it will be included as a Medium (holding all other factors constant). The vulnerability must be something that is not considered an acceptable risk by a reasonable protocol team.
>
> we consider the attack is cost high becaues the attacker needs to monitor the pending claim transaction from notional side while has no economic gain

You've created a valid escalation for 10 USDC!

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.

0xleastwood

> Escalate for 10 USDC. do not think this can be a high vulnerability because of the following reason
>
> this is a medium because first, the protocol can use flashbot to avoid frontrunning
>
> this is loss of reward not lose of user fund
>
> this griefing attack has no gain from attacker at all
>
> https://docs.sherlock.xyz/audits/judging/judging
>
> > Medium: There is a viable scenario (even if unlikely) that could cause the protocol to enter a state where a material amount of funds can be lost. The attack path is possible with assumptions that either mimic on-chain conditions or reflect conditions that have a reasonable chance of becoming true in the future. The more expensive the attack is for an attacker, the less likely it will be included as a Medium (holding all other factors constant). The vulnerability must be something that is not considered an acceptable risk by a reasonable protocol team.
>
> we consider the attack is cost high becaues the attacker needs to monitor the pending claim transaction from notional side while has no economic gain

I disagree, `COMP` rewards continuously accrue over time and hence this function can be called at any time to lose rewards. This attack is not expensive at all to perform and can be profitable in other ways that aren't immediately apparent. The difference between `high` and `medium` risk within the context of Sherlock's judging is `There is a viable scenario (even if unlikely)` vs `This vulnerability would result in a material loss of funds, and the cost of the attack is low (relative to the amount of funds lost)`. It is clear that this fits the criteria of the latter.

Trumpero

Agree with the comment by Leastwood, this issue should be high

hrishibhat

Result: High Has duplicates Given that the rewards are an integral part of protocol these would be stuck in the contract due to the exploit considering this as a valid high

sherlock-admin

Escalations have been resolved successfully!

Escalation status:

- ShadowForce: rejected

Issue H-2: repayAccountPrimeDebtAtSettlement() user lost residual cash

Source: https://github.com/sherlock-audit/2023-03-notional-judging/issues/172

Found by

bin2chen, xiaoming90 + 0xleastwood

Summary

in `repayAccountPrimeDebtAtSettlement()` Incorrect calculation of `primeCashRefund` value (always == 0) Resulting in the loss of the user's residual cash

Vulnerability Detail

when settle Vault Account will execute `settleVaultAccount()->repayAccountPrimeDebtAtSettlement()` In the `repayAccountPrimeDebtAtSettlement()` method the residual amount will be refunded to the user The code is as follows.

```
    function repayAccountPrimeDebtAtSettlement(
        PrimeRate memory pr,
        VaultStateStorage storage primeVaultState,
        uint16 currencyId,
        address vault,
        address account,
        int256 accountPrimeCash,
        int256 accountPrimeStorageValue
    ) internal returns (int256 finalPrimeDebtStorageValue, bool didTransfer) {
...

            if (netPrimeDebtRepaid < accountPrimeStorageValue) {
                // If the net debt change is greater than the debt held by the
↪   account, then only
                // decrease the total prime debt by what is held by the account.
↪   The residual amount
                // will be refunded to the account via a direct transfer.
                netPrimeDebtChange = accountPrimeStorageValue;
                finalPrimeDebtStorageValue = 0;

                int256 primeCashRefund = pr.convertFromUnderlying(
                    pr.convertDebtStorageToUnderlying(netPrimeDebtChange.sub(acc
↪   ountPrimeStorageValue)) //<--------@audit always ==0
                );
                TokenHandler.withdrawPrimeCash(
```

```
                    account, currencyId, primeCashRefund, pr, false // ETH will
↪  be transferred natively
                );
                didTransfer = true;
            } else {
```

From the above code we can see that there is a spelling error

1. netPrimeDebtChange = accountPrimeStorageValue;
2. primeCashRefund = netPrimeDebtChange.sub(accountPrimeStorageValue) so primeCashRefund always ==0

should be `primeCashRefund = netPrimeDebtRepaid - accountPrimeStorageValue`

Impact

`primeCashRefund` always == 0 , user lost residual cash

Code Snippet

https://github.com/sherlock-audit/2023-03-notional/blob/main/contracts-v2/contracts/internal/vaults/VaultAccount.sol#L575

Tool used

Manual Review

Recommendation

```
    function repayAccountPrimeDebtAtSettlement(
        PrimeRate memory pr,
        VaultStateStorage storage primeVaultState,
        uint16 currencyId,
        address vault,
        address account,
        int256 accountPrimeCash,
        int256 accountPrimeStorageValue
    ) internal returns (int256 finalPrimeDebtStorageValue, bool didTransfer) {
...

            if (netPrimeDebtRepaid < accountPrimeStorageValue) {
                // If the net debt change is greater than the debt held by the
↪  account, then only
                // decrease the total prime debt by what is held by the account.
↪  The residual amount
                // will be refunded to the account via a direct transfer.
```

```
                netPrimeDebtChange = accountPrimeStorageValue;
                finalPrimeDebtStorageValue = 0;

                int256 primeCashRefund = pr.convertFromUnderlying(
-                   pr.convertDebtStorageToUnderlying(netPrimeDebtChange.sub(acc
↪   ountPrimeStorageValue))
+                   pr.convertDebtStorageToUnderlying(netPrimeDebtRepaid.sub(acc
↪   ountPrimeStorageValue))
                );
                TokenHandler.withdrawPrimeCash(
                    account, currencyId, primeCashRefund, pr, false // ETH will
↪   be transferred natively
                );
                didTransfer = true;
            } else {
```

Discussion

jeffywu

Fixed: https://github.com/notional-finance/contracts-v2/pull/135

xiaoming9090

@0xleastwood + @xiaoming9090 : Verified. Fixed in PR https://github.com/notional-finance/contracts-v2/pull/135

Issue H-3: `VaultAccountSecondaryDebtShareStorage.maturity` will be cleared prematurely

Source: https://github.com/sherlock-audit/2023-03-notional-judging/issues/183

Found by

xiaoming90 + 0xleastwood

Summary

`VaultAccountSecondaryDebtShareStorage.maturity` will be cleared prematurely during liquidation

Vulnerability Detail

If both the `accountDebtOne` and `accountDebtTwo` of secondary currencies are zero, Notional will consider both debt shares to be cleared to zero, and the maturity will be cleared as well as shown below.

https://github.com/sherlock-audit/2023-03-notional/blob/main/contracts-v2/contracts/internal/vaults/VaultSecondaryBorrow.sol#L495

```
File: VaultSecondaryBorrow.sol
495:     function _setAccountMaturity(
496:         VaultAccountSecondaryDebtShareStorage storage accountStorage,
497:         int256 accountDebtOne,
498:         int256 accountDebtTwo,
499:         uint40 maturity
500:     ) private {
501:         if (accountDebtOne == 0 && accountDebtTwo == 0) {
502:             // If both debt shares are cleared to zero, clear the maturity
 ↪  as well.
503:             accountStorage.maturity = 0;
504:         } else {
505:             // In all other cases, set the account to the designated
 ↪  maturity
506:             accountStorage.maturity = maturity;
507:         }
508:     }
```

`VaultLiquidationAction.deleverageAccount` function

Within the `VaultLiquidationAction.deleverageAccount` function, it will call the `_reduceAccountDebt` function.

Referring to the `_reduceAccountDebt` function below. Assume that the `currencyIndex` reference to a secondary currency. In this case, the else logic in Line 251 will be executed. An important point to take note of that is critical to understand this bug is that only ONE of the prime rates will be set as it assumes that the other prime rate will not be used (Refer to Line 252 - 255). However, this assumption is incorrect.

Assume that the `currencyIndex` is `1`. Then `netUnderlyingDebtOne` parameter will be set to a non-zero value (`depositUnderlyingInternal`) at Line 261 while `netUnderlyingDebtTwo` parameter will be set to zero at Line 262. This is because, in Line 263 of the `_reduceAccountDebt` function, the `pr[0]` will be set to the prime rate, while the `pr[1]` will be zero or empty. It will then proceed to call the `VaultSecondaryBorrow.updateAccountSecondaryDebt`

https://github.com/sherlock-audit/2023-03-notional/blob/main/contracts-v2/contracts/external/actions/VaultLiquidationAction.sol#L239

```
File: VaultLiquidationAction.sol
239:     function _reduceAccountDebt(
240:         VaultConfig memory vaultConfig,
241:         VaultState memory vaultState,
242:         VaultAccount memory vaultAccount,
243:         PrimeRate memory primeRate,
244:         uint256 currencyIndex,
245:         int256 depositUnderlyingInternal,
246:         bool checkMinBorrow
247:     ) private {
248:         if (currencyIndex == 0) {
249:             vaultAccount.updateAccountDebt(vaultState,
 ↪  depositUnderlyingInternal, 0);
250:             vaultState.setVaultState(vaultConfig);
251:         } else {
252:             // Only set one of the prime rates, the other prime rate is not
 ↪  used since
253:             // the net debt amount is set to zero
254:             PrimeRate[2] memory pr;
255:             pr[currencyIndex - 1] = primeRate;
256:
257:             VaultSecondaryBorrow.updateAccountSecondaryDebt(
258:                 vaultConfig,
259:                 vaultAccount.account,
260:                 vaultAccount.maturity,
261:                 currencyIndex == 1 ? depositUnderlyingInternal : 0,
262:                 currencyIndex == 2 ? depositUnderlyingInternal : 0,
263:                 pr,
264:                 checkMinBorrow
265:             );
266:         }
```

```
267:     }
```

Within the `updateAccountSecondaryDebt` function, at Line 272, assume that `accountStorage.accountDebtTwo` is `100`. However, since `pr[1]` is not initialized, the `VaultStateLib.readDebtStorageToUnderlying` will return a zero value and set the `accountDebtTwo` to zero.

Assume that the liquidator calls the `deleverageAccount` function to clear all the debt of the `currencyIndex` secondary currency. Line 274 will be executed, and `accountDebtOne` will be set to zero.

Note that at this point, both `accountDebtOne` and `accountDebtTwo` are zero. At Line 301, the `_setAccountMaturity` will set the `accountStorage.maturity = 0` , which clears the vault account's maturity.

An important point here is that the liquidator did not clear the `accountDebtTwo`. Yet, `accountDebtTwo` became zero in memory during the execution and caused Notional to wrongly assume that both debt shares had been cleared to zero.

https://github.com/sherlock-audit/2023-03-notional/blob/main/contracts-v2/contracts/internal/vaults/VaultSecondaryBorrow.sol#L256

```
File: VaultSecondaryBorrow.sol
256:     function updateAccountSecondaryDebt(
257:         VaultConfig memory vaultConfig,
258:         address account,
259:         uint256 maturity,
260:         int256 netUnderlyingDebtOne,
261:         int256 netUnderlyingDebtTwo,
262:         PrimeRate[2] memory pr,
263:         bool checkMinBorrow
264:     ) internal {
265:         VaultAccountSecondaryDebtShareStorage storage accountStorage =
266:
 ↪  LibStorage.getVaultAccountSecondaryDebtShare()[account][vaultConfig.vault];
267:         // Check maturity
268:         uint256 accountMaturity = accountStorage.maturity;
269:         require(accountMaturity == maturity || accountMaturity == 0);
270:
271:         int256 accountDebtOne =
 ↪  VaultStateLib.readDebtStorageToUnderlying(pr[0], maturity,
 ↪  accountStorage.accountDebtOne);
272:         int256 accountDebtTwo =
 ↪  VaultStateLib.readDebtStorageToUnderlying(pr[1], maturity,
 ↪  accountStorage.accountDebtTwo);
273:         if (netUnderlyingDebtOne != 0) {
274:             accountDebtOne = accountDebtOne.add(netUnderlyingDebtOne);
```

```
276:             _updateTotalSecondaryDebt(
277:                 vaultConfig, account,
↪  vaultConfig.secondaryBorrowCurrencies[0], maturity, netUnderlyingDebtOne,
↪  pr[0]
278:             );
279:
280:             accountStorage.accountDebtOne =
↪  VaultStateLib.calculateDebtStorage(pr[0], maturity, accountDebtOne)
281:                 .neg().toUint().toUint80();
282:         }
283:
284:         if (netUnderlyingDebtTwo != 0) {
285:             accountDebtTwo = accountDebtTwo.add(netUnderlyingDebtTwo);
286:
287:             _updateTotalSecondaryDebt(
288:                 vaultConfig, account,
↪  vaultConfig.secondaryBorrowCurrencies[1], maturity, netUnderlyingDebtTwo,
↪  pr[1]
289:             );
290:
291:             accountStorage.accountDebtTwo =
↪  VaultStateLib.calculateDebtStorage(pr[1], maturity, accountDebtTwo)
292:                 .neg().toUint().toUint80();
293:         }
294:
295:         if (checkMinBorrow) {
296:             // No overflow on negation due to overflow checks above
297:             require(accountDebtOne == 0 ||
↪  vaultConfig.minAccountSecondaryBorrow[0] <= -accountDebtOne, "min borrow");
298:             require(accountDebtTwo == 0 ||
↪  vaultConfig.minAccountSecondaryBorrow[1] <= -accountDebtTwo, "min borrow");
299:         }
300:
301:         _setAccountMaturity(accountStorage, accountDebtOne, accountDebtTwo,
↪  maturity.toUint40());
302:     }
```

The final state will be `VaultAccountSecondaryDebtShareStorage` as follows:

- `maturity` and `accountDebtOne` are zero
- `accountDebtTwo` = 100

https://github.com/sherlock-audit/2023-03-notional/blob/main/contracts-v2/contracts/global/Types.sol#L551

```
struct VaultAccountSecondaryDebtShareStorage {
    // Maturity for the account's secondary borrows. This is stored separately
↪  from
```

```
    // the vault account maturity to ensure that we have access to the proper
↪   state
    // during a roll borrow position. It should never be allowed to deviate from
↪   the
    // vaultAccount.maturity value (unless it is cleared to zero).
    uint40 maturity;
    // Account debt for the first secondary currency in either fCash or pCash
↪   denomination
    uint80 accountDebtOne;
    // Account debt for the second secondary currency in either fCash or pCash
↪   denomination
    uint80 accountDebtTwo;
}
```

Firstly, it does not make sense to have `accountDebtTwo` but no maturity in storage, which also means the vault account data is corrupted. Secondly, when `maturity` is zero, it also means that the vault account did not borrow anything from Notional. Lastly, many vault logic would break since it relies on the maturity value.

`VaultLiquidationAction.liquidateVaultCashBalance` function

The root cause lies in the implementation of the `_reduceAccountDebt` function. Since `liquidateVaultCashBalance` function calls the `_reduceAccountDebt` function to reduce the debt of the vault account being liquidated, the same issue will occur here.

Impact

Any vault logic that relies on the `VaultAccountSecondaryDebtShareStorage`'s maturity value would break since it has been cleared (set to zero). For instance, a vault account cannot be settled anymore as the following `settleSecondaryBorrow` function will always revert. Since `storedMaturity == 0` but `accountDebtTwo` is not zero, Line 399 below will always revert.

As a result, a vault account with secondary currency debt cannot be settled. This also means that the vault account cannot exit since a vault account needs to be settled before exiting, causing users' assets to be stuck within the protocol.

https://github.com/sherlock-audit/2023-03-notional/blob/main/contracts-v2/contracts/internal/vaults/VaultSecondaryBorrow.sol#L385

```
File: VaultSecondaryBorrow.sol
385:     function settleSecondaryBorrow(VaultConfig memory vaultConfig, address
↪   account) internal returns (bool) {
386:         if (!vaultConfig.hasSecondaryBorrows()) return false;
387:
388:         VaultAccountSecondaryDebtShareStorage storage accountStorage =
```

```
389:
↪  LibStorage.getVaultAccountSecondaryDebtShare()[account][vaultConfig.vault];
390:         uint256 storedMaturity = accountStorage.maturity;
391:
392:         // NOTE: we can read account debt directly since prime cash
↪  maturities never enter this block of code.
393:         int256 accountDebtOne =
↪  -int256(uint256(accountStorage.accountDebtOne));
394:         int256 accountDebtTwo =
↪  -int256(uint256(accountStorage.accountDebtTwo));
395:
396:         if (storedMaturity == 0) {
397:             // Handles edge condition where an account is holding vault
↪  shares past maturity without
398:             // any debt position.
399:             require(accountDebtOne == 0 && accountDebtTwo == 0);
400:         } else {
```

In addition, the vault account data is corrupted as there is a secondary debt without maturity, which might affect internal accounting and tracking.

Code Snippet

https://github.com/sherlock-audit/2023-03-notional/blob/main/contracts-v2/contracts/external/actions/VaultLiquidationAction.sol#L239

Tool used

Manual Review

Recommendation

Fetch the prime rate of both secondary currencies because they are both needed within the `updateAccountSecondaryDebt` function when converting debt storage to underlying.

```
    function _reduceAccountDebt(
        VaultConfig memory vaultConfig,
        VaultState memory vaultState,
        VaultAccount memory vaultAccount,
        PrimeRate memory primeRate,
        uint256 currencyIndex,
        int256 depositUnderlyingInternal,
        bool checkMinBorrow
    ) private {
        if (currencyIndex == 0) {
```

```
            vaultAccount.updateAccountDebt(vaultState,
↪   depositUnderlyingInternal, 0);
            vaultState.setVaultState(vaultConfig);
        } else {
            // Only set one of the prime rates, the other prime rate is not used
↪   since
            // the net debt amount is set to zero
            PrimeRate[2] memory pr;
-           pr[currencyIndex - 1] = primeRate;
+       pr = VaultSecondaryBorrow.getSecondaryPrimeRateStateful(vaultConfig);

            VaultSecondaryBorrow.updateAccountSecondaryDebt(
                vaultConfig,
                vaultAccount.account,
                vaultAccount.maturity,
                currencyIndex == 1 ? depositUnderlyingInternal : 0,
                currencyIndex == 2 ? depositUnderlyingInternal : 0,
                pr,
                checkMinBorrow
            );
        }
    }
```

Discussion

jeffywu

Fixed: https://github.com/notional-finance/contracts-v2/pull/127

xiaoming9090

@0xleastwood + @xiaoming9090 : Verified. Fixed in PR https://github.com/notional-finance/contracts-v2/pull/127

Issue H-4: StrategyVault can perform a full exit without repaying all secondary debt

Source: https://github.com/sherlock-audit/2023-03-notional-judging/issues/184

Found by

xiaoming90 + 0xleastwood

Summary

StrategyVault can perform a full exit without repaying all secondary debt, leaving bad debt with the protocol.

Vulnerability Detail

Noted from the codebase's comment that:

> Vaults can borrow up to the capacity using the `borrowSecondaryCurrencyToVault` and `repaySecondaryCurrencyToVault` methods. Vaults that use a secondary currency must ALWAYS repay the secondary debt during redemption and handle accounting for the secondary currency themselves.

Thus, when the StrategyVault-side performs a full exit for a vault account, Notional-side does not check that all secondary debts of that vault account are cleared (= zero) and will simply trust StrategyVault-side has already handled them properly.

Line 271 below shows that only validates the primary debt but not the secondary debt during a full exit.

https://github.com/sherlock-audit/2023-03-notional/blob/main/contracts-v2/contracts/external/actions/VaultAccountAction.sol#L271

```
File: VaultAccountAction.sol
271:         if (vaultAccount.accountDebtUnderlying == 0 &&
↪  vaultAccount.vaultShares == 0) {
272:             // If the account has no position in the vault at this point,
↪  set the maturity to zero as well
273:             vaultAccount.maturity = 0;
274:         }
275:         vaultAccount.setVaultAccount({vaultConfig: vaultConfig,
↪  checkMinBorrow: true});
276:
277:         // It's possible that the user redeems more vault shares than they
↪  lend (it is not always the case
```

```
278:         // that they will be increasing their collateral ratio here, so we
 ↪  check that this is the case). No
279:         // need to check if the account has exited in full (maturity == 0).
280:         if (vaultAccount.maturity != 0) {
281:
 ↪  IVaultAccountHealth(address(this)).checkVaultAccountCollateralRatio(vault,
 ↪  account);
282:         }
```

Impact

Leveraged vaults are designed to be as isolated as possible to mitigate the risk to the Notional protocol and its users. However, the above implementation seems to break this principle. As such, if there is a vulnerability in the leverage vault that allows someone to exploit this issue and bypass the repayment of the secondary debt, the protocol will be left with a bad debt which affects the insolvency of the protocol.

Code Snippet

https://github.com/sherlock-audit/2023-03-notional/blob/main/contracts-v2/contracts/external/actions/VaultAccountAction.sol#L271

Tool used

Manual Review

Recommendation

Consider checking that all secondary debts of a vault account are cleared before executing a full exit.

```
+ int256 accountDebtOne;
+ int256 accountDebtTwo;

+ if (vaultConfig.hasSecondaryBorrows()) {
+   (/* */, accountDebtOne, accountDebtTwo) =
 ↪  VaultSecondaryBorrow.getAccountSecondaryDebt(vaultConfig, account, pr);
+ }

- if (vaultAccount.accountDebtUnderlying == 0 && vaultAccount.vaultShares == 0) {
+ if (vaultAccount.accountDebtUnderlying == 0 && vaultAccount.vaultShares == 0
 ↪  && accountDebtOne == 0 && accountDebtTwo == 0) {
    // If the account has no position in the vault at this point, set the
 ↪  maturity to zero as well
    vaultAccount.maturity = 0;
```

```
}
vaultAccount.setVaultAccount({vaultConfig: vaultConfig, checkMinBorrow: true});
```

Discussion

jeffywu

Valid suggestion

jeffywu

Fixed: https://github.com/notional-finance/contracts-v2/pull/128

xiaoming9090

@0xleastwood + @xiaoming9090: Verified. Fixed in PR https://github.com/notional-finance/contracts-v2/pull/128

Issue H-5: Unable to transfer fee reserve assets to treasury

Source: https://github.com/sherlock-audit/2023-03-notional-judging/issues/190

Found by

xiaoming90 + 0xleastwood

Summary

Transferring fee reserve assets to the treasury manager contract will result in a revert, leading to a loss of rewards for NOTE stakers.

Vulnerability Detail

https://github.com/sherlock-audit/2023-03-notional/blob/main/contracts-v2/contracts/external/actions/TreasuryAction.sol#L135

```
File: TreasuryAction.sol
132:     /// @notice redeems and transfers tokens to the treasury manager
 ↪  contract
133:     function _redeemAndTransfer(uint16 currencyId, int256
 ↪  primeCashRedeemAmount) private returns (uint256) {
134:         PrimeRate memory primeRate =
 ↪  PrimeRateLib.buildPrimeRateStateful(currencyId);
135:         int256 actualTransferExternal = TokenHandler.withdrawPrimeCash(
136:             treasuryManagerContract,
137:             currencyId,
138:             primeCashRedeemAmount.neg(),
139:             primeRate,
140:             true // if ETH, transfers it as WETH
141:         );
142:
143:         require(actualTransferExternal > 0);
144:         return uint256(actualTransferExternal);
145:     }
```

The value returned by the TokenHandler.withdrawPrimeCash function is always less than or equal to zero. Thus, the condition `actualTransferExternal > 0` will always be false, and the `_redeemAndTransfer` function will always revert.

The `transferReserveToTreasury` function depends on `_redeemAndTransfer` function. Thus, it is not possible to transfer any asset to the treasury manager contract.

Impact

The fee collected by Notional is stored in the Fee Reserve. The fee reserve assets will be transferred to Notional's Treasury to be invested into the sNOTE pool. Without the ability to do so, the NOTE stakers will not receive their rewards.

Code Snippet

https://github.com/sherlock-audit/2023-03-notional/blob/main/contracts-v2/contracts/external/actions/TreasuryAction.sol#L135

Tool used

Manual Review

Recommendation

Negate the value returned by the `TokenHandler.withdrawPrimeCash` function.

```
        int256 actualTransferExternal = TokenHandler.withdrawPrimeCash(
            treasuryManagerContract,
            currencyId,
            primeCashRedeemAmount.neg(),
            primeRate,
            true // if ETH, transfers it as WETH
-       );
+       ).neg();
```

Discussion

jeffywu

Fixed: https://github.com/notional-finance/contracts-v2/pull/119

xiaoming9090

@0xleastwood + @xiaoming9090 : Verified. Fixed in PR https://github.com/notional-finance/contracts-v2/pull/119

Issue H-6: Excess funds withdrawn from the money market

Source: https://github.com/sherlock-audit/2023-03-notional-judging/issues/193

Found by

bin2chen, chaduke, iglyx, mstpr-brainbot, xiaoming90 + 0xleastwood

Summary

Excessive amounts of assets are being withdrawn from the money market.

Vulnerability Detail

https://github.com/sherlock-audit/2023-03-notional/blob/main/contracts-v2/contracts/internal/balances/TokenHandler.sol#L270

```
File: TokenHandler.sol
256:     function _redeemMoneyMarketIfRequired(
257:         uint16 currencyId,
258:         Token memory underlying,
259:         uint256 withdrawAmountExternal
260:     ) private {
261:         // If there is sufficient balance of the underlying to withdraw
 ↪  from the contract
262:         // immediately, just return.
263:         mapping(address => uint256) storage store =
 ↪  LibStorage.getStoredTokenBalances();
264:         uint256 currentBalance = store[underlying.tokenAddress];
265:         if (withdrawAmountExternal <= currentBalance) return;
266:
267:         IPrimeCashHoldingsOracle oracle =
 ↪  PrimeCashExchangeRate.getPrimeCashHoldingsOracle(currencyId);
268:         // Redemption data returns an array of contract calls to make from
 ↪  the Notional proxy (which
269:         // is holding all of the money market tokens).
270:         (RedeemData[] memory data) =
 ↪  oracle.getRedemptionCalldata(withdrawAmountExternal);
271:
272:         // This is the total expected underlying that we should redeem
 ↪  after all redemption calls
273:         // are executed.
274:         uint256 totalUnderlyingRedeemed =
 ↪  executeMoneyMarketRedemptions(underlying, data);
275:
```

```
276:         // Ensure that we have sufficient funds before we exit
277:         require(withdrawAmountExternal <=
 ↪  currentBalance.add(totalUnderlyingRedeemed)); // dev: insufficient redeem
278:     }
```

If the `currentBalance` is `999,900` USDC and the `withdrawAmountExternal` is `1,000,000` USDC, then there is insufficient balance in the contract, and additional funds need to be withdrawn from the money market (e.g. Compound).

Since the contract already has `999,900` USDC, only an additional `100` USDC needs to be withdrawn from the money market to fulfill the withdrawal request of `1,000,000` USDC

However, instead of withdrawing `100` USDC from the money market, Notional withdraw `1,000,000` USDC from the market as per the `oracle.getRedemptionCalldata(withdrawAmountExternal)` function. As a result, an excess of `990,000` USDC is being withdrawn from the money market

Impact

This led to an excessive amount of assets idling in Notional and not generating any returns or interest in the money market, which led to a loss of assets for the users as they would receive a lower interest rate than expected and incur opportunity loss.

Attackers could potentially abuse this to pull the funds Notional invested in the money market leading to griefing and loss of returns/interest for the protocol.

Code Snippet

https://github.com/sherlock-audit/2023-03-notional/blob/main/contracts-v2/contracts/internal/balances/TokenHandler.sol#L270

Tool used

Manual Review

Recommendation

Consider withdrawing only the shortfall amount from the money market.

```
- (RedeemData[] memory data) =
 ↪  oracle.getRedemptionCalldata(withdrawAmountExternal);
+ (RedeemData[] memory data) =
 ↪  oracle.getRedemptionCalldata(withdrawAmountExternal - currentBalance);
```

Discussion

jeffywu

Valid Issue

jeffywu

Fixed: https://github.com/notional-finance/contracts-v2/pull/125

xiaoming9090

@0xleastwood + @xiaoming9090 : Verified. Fixed in PR https://github.com/notional-finance/contracts-v2/pull/125.

Issue H-7: Possible to liquidate past the debt outstanding above the min borrow without liquidating the entire debt outstanding

Source: https://github.com/sherlock-audit/2023-03-notional-judging/issues/194

Found by

xiaoming90 + 0xleastwood

Summary

It is possible to liquidate past the debt outstanding above the min borrow without liquidating the entire debt outstanding. Thus, leaving accounts with small debt that are not profitable to unwind if it needs to liquidate.

Vulnerability Detail

https://github.com/sherlock-audit/2023-03-notional/blob/main/contracts-v2/contracts/internal/vaults/VaultValuation.sol#L251

```
File: VaultValuation.sol
250:         // NOTE: deposit amount is always positive in this method
251:         if (depositUnderlyingInternal < maxLiquidatorDepositLocal) {
252:             // If liquidating past the debt outstanding above the min
↪  borrow, then the entire
253:             // debt outstanding must be liquidated.
254:
255:             // (debtOutstanding - depositAmountUnderlying) is the post
↪  liquidation debt. As an
256:             // edge condition, when debt outstanding is discounted to
↪  present value, the account
257:             // may be liquidated to zero while their debt outstanding is
↪  still greater than the
258:             // min borrow size (which is normally enforced in notional
↪  terms -- i.e. non present
259:             // value). Resolving this would require additional complexity
↪  for not much gain. An
260:             // account within 20% of the minBorrowSize in a vault that has
↪  fCash discounting enabled
261:             // may experience a full liquidation as a result.
262:             require(
263:
↪  h.debtOutstanding[currencyIndex].sub(depositUnderlyingInternal) <
↪  minBorrowSize,
```

```
264:                 "Must Liquidate All Debt"
265:             );
```

- `depositUnderlyingInternal` is always a positive value (Refer to comment on Line 250) that represents the amount of underlying deposited by the liquidator
- `h.debtOutstanding[currencyIndex]` is always a negative value representing debt outstanding of a specific currency in a vault account
- `minBorrowSize` is always a positive value that represents the minimal borrow size of a specific currency (It is stored as uint32 in storage)

If liquidating past the debt outstanding above the min borrow, then the entire debt outstanding must be liquidated.

Assume the following scenario:

- `depositUnderlyingInternal = 70 USDC`
- `h.debtOutstanding[currencyIndex] = -100 USDC`
- `minBorrowSize = 50 USDC`

If the liquidation is successful, the vault account should be left with `-30 USDC` debt outstanding because `70 USDC` has been paid off by the liquidator. However, this should not happen under normal circumstances because the debt outstanding (`-30`) does not meet the minimal borrow size of `50 USDC` and the liquidation should revert/fail.

The following piece of validation logic attempts to ensure that all outstanding debt is liquidated if post-liquidation debt does not meet the minimal borrowing size.

```
require(
    h.debtOutstanding[currencyIndex].sub(depositUnderlyingInternal) <
↪   minBorrowSize,
    "Must Liquidate All Debt"
);
```

Plugging in the values from our scenario to verify if the code will revert if the debt outstanding does not meet the minimal borrow size.

```
require(
    (-100 USDC - 70 USDC) < 50 USDC
);
===>
require(
    (-170 USDC) < 50 USDC
);
===>
```

```
require(true) // no revert
```

The above shows that it is possible for someone to liquidate past the debt outstanding above the min borrow without liquidating the entire debt outstanding. This shows that the math formula in the code is incorrect and not working as intended.

Impact

A liquidation can bring an account below the minimum debt. Accounts smaller than the minimum debt are not profitable to unwind if it needs to liquidate (Reference)

As a result, liquidators are not incentivized to liquidate those undercollateralized positions. This might leave the protocol with bad debts, potentially leading to insolvency if the bad debts accumulate.

Code Snippet

https://github.com/sherlock-audit/2023-03-notional/blob/main/contracts-v2/contracts/internal/vaults/VaultValuation.sol#L251

Tool used

Manual Review

Recommendation

Update the formula to as follows:

```
require(
-   h.debtOutstanding[currencyIndex].sub(depositUnderlyingInternal) <
 ↪  minBorrowSize,
+   h.debtOutstanding[currencyIndex].neg().sub(depositUnderlyingInternal) >
 ↪  minBorrowSize,
    "Must Liquidate All Debt"
);
```

Plugging in the values from our scenario again to verify if the code will revert if the debt outstanding does not meet the minimal borrow size.

```
require(
    ((-100 USDC).neg() - 70 USDC) > 50 USDC
);
===>
require(
    (100 USDC - 70 USDC) > 50 USDC
```

```
);
===>
require(
    (30 USDC) > 50 USDC
);
===>
require(false) // revert
```

The above will trigger a revert as expected when the debt outstanding does not meet the minimal borrow size.

Discussion

jeffywu

Fixed: https://github.com/notional-finance/contracts-v2/pull/132

xiaoming9090

@0xleastwood + @xiaoming9090 : Verified. Fixed in PR https://github.com/notional-finance/contracts-v2/pull/132

Issue H-8: Vaults can avoid liquidations by not letting their vault account be settled

Source: https://github.com/sherlock-audit/2023-03-notional-judging/issues/199

Found by

ShadowForce, xiaoming90 + 0xleastwood

Summary

Vault liquidations will leave un-matured accounts with cash holdings which are then used to offset account debt during vault account settlements. As it stands, any excess cash received via interest accrual will be transferred back to the vault account directly. If a primary or secondary borrow currency is `ETH`, then this excess cash will be transferred natively. Consequently, the recipient may intentionally revert, causing account settlement to fail.

Vulnerability Detail

The issue arises in the `VaultAccount.repayAccountPrimeDebtAtSettlement()` function. If there is any excess cash due to interest accrual, then this amount will be refunded to the vault account. Native `ETH` is not wrapped when it should be wrapped, allowing the recipient to take control over the flow of execution.

https://github.com/sherlock-audit/2023-03-notional-0xleastwood/blob/main/contracts-v2/contracts/internal/vaults/VaultAccount.sol#L548-L596

```
File: VaultAccount.sol
548:     function repayAccountPrimeDebtAtSettlement(
549:         PrimeRate memory pr,
550:         VaultStateStorage storage primeVaultState,
551:         uint16 currencyId,
552:         address vault,
553:         address account,
554:         int256 accountPrimeCash,
555:         int256 accountPrimeStorageValue
556:     ) internal returns (int256 finalPrimeDebtStorageValue, bool
↪  didTransfer) {
557:         didTransfer = false;
558:         finalPrimeDebtStorageValue = accountPrimeStorageValue;
559:
560:         if (accountPrimeCash > 0) {
561:             // netPrimeDebtRepaid is a negative number
562:             int256 netPrimeDebtRepaid = pr.convertUnderlyingToDebtStorage(
563:                 pr.convertToUnderlying(accountPrimeCash).neg()
```

```
564:             );
565:
566:             int256 netPrimeDebtChange;
567:             if (netPrimeDebtRepaid < accountPrimeStorageValue) {
568:                 // If the net debt change is greater than the debt held by
 ↪  the account, then only
569:                 // decrease the total prime debt by what is held by the
 ↪  account. The residual amount
570:                 // will be refunded to the account via a direct transfer.
571:                 netPrimeDebtChange = accountPrimeStorageValue;
572:                 finalPrimeDebtStorageValue = 0;
573:
574:                 int256 primeCashRefund = pr.convertFromUnderlying(
575:                     pr.convertDebtStorageToUnderlying(netPrimeDebtChange.su
 ↪  b(accountPrimeStorageValue))
576:                 );
577:                 TokenHandler.withdrawPrimeCash(
578:                     account, currencyId, primeCashRefund, pr, false // ETH
 ↪  will be transferred natively
579:                 );
580:                 didTransfer = true;
581:             } else {
582:                 // In this case, part of the account's debt is repaid.
583:                 netPrimeDebtChange = netPrimeDebtRepaid;
584:                 finalPrimeDebtStorageValue =
 ↪  accountPrimeStorageValue.sub(netPrimeDebtRepaid);
585:             }
586:
587:             // Updates the global prime debt figure and events are emitted
 ↪  via the vault.
588:             pr.updateTotalPrimeDebt(vault, currencyId, netPrimeDebtChange);
589:
590:             // Updates the state on the prime vault storage directly.
591:             int256 totalPrimeDebt =
 ↪  int256(uint256(primeVaultState.totalDebt));
592:             int256 newTotalDebt = totalPrimeDebt.add(netPrimeDebtChange);
593:             // Set the total debt to the storage value
594:             primeVaultState.totalDebt = newTotalDebt.toUint().toUint80();
595:         }
596:     }
```

As seen here, a `withdrawWrappedNativeToken` is used to signify when a native `ETH` transfer will be wrapped before sending an amount. In the case of vault settlement, this is always sent to `false`.

https://github.com/sherlock-audit/2023-03-notional-0xleastwood/blob/main/contracts-v2/contracts/internal/balances/TokenHandler.sol#L220-L247

```
File: TokenHandler.sol
220:     function withdrawPrimeCash(
221:         address account,
222:         uint16 currencyId,
223:         int256 primeCashToWithdraw,
224:         PrimeRate memory primeRate,
225:         bool withdrawWrappedNativeToken
226:     ) internal returns (int256 netTransferExternal) {
227:         if (primeCashToWithdraw == 0) return 0;
228:         require(primeCashToWithdraw < 0);
229:
230:         Token memory underlying = getUnderlyingToken(currencyId);
231:         netTransferExternal = convertToExternal(
232:             underlying,
233:             primeRate.convertToUnderlying(primeCashToWithdraw)
234:         );
235:
236:         // Overflow not possible due to int256
237:         uint256 withdrawAmount = uint256(netTransferExternal.neg());
238:         _redeemMoneyMarketIfRequired(currencyId, underlying,
 ↪  withdrawAmount);
239:
240:         if (underlying.tokenType == TokenType.Ether) {
241:             GenericToken.transferNativeTokenOut(account, withdrawAmount,
 ↪  withdrawWrappedNativeToken);
242:         } else {
243:             GenericToken.safeTransferOut(underlying.tokenAddress, account,
 ↪  withdrawAmount);
244:         }
245:
246:         _postTransferPrimeCashUpdate(account, currencyId,
 ↪  netTransferExternal, underlying, primeRate);
247:     }
```

It's likely that the vault account is considered solvent in this case, but due to the inability to trade between currencies, it is not possible to use excess cash in one currency to offset debt in another.

Impact

Liquidations require vaults to be settled if `block.timestamp` is past the maturity date, hence, it is not possible to deleverage vault accounts, leading to bad debt accrual.

Code Snippet

https://github.com/sherlock-audit/2023-03-notional-0xleastwood/blob/main/contracts-v2/contracts/internal/vaults/VaultAccount.sol#L548-L596

https://github.com/sherlock-audit/2023-03-notional-0xleastwood/blob/main/contracts-v2/contracts/internal/balances/TokenHandler.sol#L220-L247

Tool used

Manual Review

Recommendation

Consider wrapping `ETH` under all circumstances. This will prevent vault accounts from intentionally reverting and preventing their account from being settled.

Discussion

jeffywu

Valid Issue, in our fixes we remove transfers from Settle Vault Account.

jeffywu

Fixed in: https://github.com/notional-finance/contracts-v2/pull/135

0xleastwood

@0xleastwood + @xiaoming9090: PR 135 attempted to fix the issue by removing the transfer of excess assets from the settlement. It will store the excess assets to be refunded in `s.primaryCash` so that it can be read to `vaultAccount.tempCashBalance` later when exiting the vault (`VaultAccountAction.exitVault`). The transfer of excess assets will be performed during the exit vault.

However, in an edge case where a liquidator deleverages another account and happens to have primary cash in their vault account, their `s.primaryCash` (excess cash) will be wiped.

Following is the function call flow for reference: `deleverageAccount` -> `_transferVaultSharesToLiquidator` -> `liquidator.setVaultAccount` -> Effect: liquidator's `s.primaryCash = 0`

0xleastwood

@0xleastwood + @xiaoming9090: Verified fix as per comments in #207. Fixed in https://github.com/notional-finance/contracts-v2/pull/135

Issue H-9: Possible to create vault positions ineligible for liquidation

Source: https://github.com/sherlock-audit/2023-03-notional-judging/issues/202

Found by

xiaoming90 + 0xleastwood

Summary

Users can self-liquidate their secondary debt holdings in such a way that it is no longer possible to deleverage their vault account as `checkMinBorrow` will fail post-maturity.

Vulnerability Detail

When deleveraging a vault account, the liquidator will pay down account debt directly and the account will not accrue any cash. Under most circumstances, it is not possible to put an account's debt below its minimum borrow size.

However, there are *two* exceptions to this:

- Liquidators purchasing cash from a vault account. This only applies to non-prime vault accounts.
- A vault account is being settled and `checkMinBorrow` is skipped to ensure an account can always be settled.

https://github.com/sherlock-audit/2023-03-notional-0xleastwood/blob/main/contracts-v2/contracts/external/actions/VaultLiquidationAction.sol#L57-L119

```
File: VaultLiquidationAction.sol
057:     function deleverageAccount(
058:         address account,
059:         address vault,
060:         address liquidator,
061:         uint16 currencyIndex,
062:         int256 depositUnderlyingInternal
063:     ) external payable nonReentrant override returns (
064:         uint256 vaultSharesToLiquidator,
065:         int256 depositAmountPrimeCash
066:     ) {
067:         require(currencyIndex < 3);
068:         (
069:             VaultConfig memory vaultConfig,
070:             VaultAccount memory vaultAccount,
```

```
071:             VaultState memory vaultState
072:         ) = _authenticateDeleverage(account, vault, liquidator);
073:
074:         PrimeRate memory pr;
075:         // Currency Index is validated in this method
076:         (
077:             depositUnderlyingInternal,
078:             vaultSharesToLiquidator,
079:             pr
080:         ) =
 ↪  IVaultAccountHealth(address(this)).calculateDepositAmountInDeleverage(
081:             currencyIndex, vaultAccount, vaultConfig, vaultState,
 ↪  depositUnderlyingInternal
082:         );
083:
084:         uint16 currencyId = vaultConfig.borrowCurrencyId;
085:         if (currencyIndex == 1) currencyId =
 ↪  vaultConfig.secondaryBorrowCurrencies[0];
086:         else if (currencyIndex == 2) currencyId =
 ↪  vaultConfig.secondaryBorrowCurrencies[1];
087:
088:         Token memory token = TokenHandler.getUnderlyingToken(currencyId);
089:         // Excess ETH is returned to the liquidator natively
090:         (/* */, depositAmountPrimeCash) =
 ↪  TokenHandler.depositUnderlyingExternal(
091:             liquidator, currencyId,
 ↪  token.convertToExternal(depositUnderlyingInternal), pr, false
092:         );
093:
094:         // Do not skip the min borrow check here
095:         vaultAccount.vaultShares =
 ↪  vaultAccount.vaultShares.sub(vaultSharesToLiquidator);
096:         if (vaultAccount.maturity == Constants.PRIME_CASH_VAULT_MATURITY) {
097:             // Vault account will not incur a cash balance if they are in
 ↪  the prime cash maturity, their debts
098:             // will be paid down directly.
099:             _reduceAccountDebt(
100:                 vaultConfig, vaultState, vaultAccount, pr, currencyIndex,
 ↪  depositUnderlyingInternal, true
101:             );
102:             depositAmountPrimeCash = 0;
103:         }
104:
105:         // Check min borrow in this liquidation method, the deleverage
 ↪  calculation should adhere to the min borrow
106:         vaultAccount.setVaultAccountForLiquidation(vaultConfig,
 ↪  currencyIndex, depositAmountPrimeCash, true);
107:
```


```
108:         emit VaultDeleverageAccount(vault, account, currencyId,
↪  vaultSharesToLiquidator, depositAmountPrimeCash);
109:         emit VaultLiquidatorProfit(vault, account, liquidator,
↪  vaultSharesToLiquidator, true);
110:
111:         _transferVaultSharesToLiquidator(
112:             liquidator, vaultConfig, vaultSharesToLiquidator,
↪  vaultAccount.maturity
113:         );
114:
115:         Emitter.emitVaultDeleverage(
116:             liquidator, account, vault, currencyId, vaultState.maturity,
117:             depositAmountPrimeCash, vaultSharesToLiquidator
118:         );
119:     }
```

`currencyIndex` represents which currency is being liquidated and `depositUnderlyingInternal` the amount of debt being reduced. Only one currency's debt can be updated here.

https://github.com/sherlock-audit/2023-03-notional-0xleastwood/blob/main/contracts-v2/contracts/external/actions/VaultLiquidationAction.sol#L239-L267

```
File: VaultLiquidationAction.sol
239:     function _reduceAccountDebt(
240:         VaultConfig memory vaultConfig,
241:         VaultState memory vaultState,
242:         VaultAccount memory vaultAccount,
243:         PrimeRate memory primeRate,
244:         uint256 currencyIndex,
245:         int256 depositUnderlyingInternal,
246:         bool checkMinBorrow
247:     ) private {
248:         if (currencyIndex == 0) {
249:             vaultAccount.updateAccountDebt(vaultState,
↪  depositUnderlyingInternal, 0);
250:             vaultState.setVaultState(vaultConfig);
251:         } else {
252:             // Only set one of the prime rates, the other prime rate is not
↪  used since
253:             // the net debt amount is set to zero
254:             PrimeRate[2] memory pr;
255:             pr[currencyIndex - 1] = primeRate;
256:
257:             VaultSecondaryBorrow.updateAccountSecondaryDebt(
258:                 vaultConfig,
259:                 vaultAccount.account,
260:                 vaultAccount.maturity,
```

```
261:                 currencyIndex == 1 ? depositUnderlyingInternal : 0,
262:                 currencyIndex == 2 ? depositUnderlyingInternal : 0,
263:                 pr,
264:                 checkMinBorrow
265:             );
266:         }
267:     }
```

In the case of vault settlement, through self-liquidation, users can setup their debt and cash holdings post-settlement, such that both `accountDebtOne` and `accountDebtTwo` are non-zero and less than `vaultConfig.minAccountSecondaryBorrow`. The objective would be to have zero primary debt and `Y` secondary debt and `X` secondary cash. Post-settlement, cash is used to offset debt (`Y - X < minAccountSecondaryBorrow`) and due to the lack of `checkMinBorrow` in `VaultAccountAction.settleVaultAccount()`, both secondary currencies can have debt holdings below the minimum amount.

Now when `deleverageAccount()` is called on a prime vault account, debts are paid down directly. However, if we are only able to pay down one secondary currency at a time, `checkMinBorrow` will fail in `VaultSecondaryBorrow.updateAccountSecondaryDebt()` because both debts are checked.

https://github.com/sherlock-audit/2023-03-notional-0xleastwood/blob/main/contracts-v2/contracts/internal/vaults/VaultSecondaryBorrow.sol#L295-L299

```
File: VaultSecondaryBorrow.sol
295:         if (checkMinBorrow) {
296:             // No overflow on negation due to overflow checks above
297:             require(accountDebtOne == 0 ||
 ↪  vaultConfig.minAccountSecondaryBorrow[0] <= -accountDebtOne, "min borrow");
298:             require(accountDebtTwo == 0 ||
 ↪  vaultConfig.minAccountSecondaryBorrow[1] <= -accountDebtTwo, "min borrow");
299:         }
```

No prime fees accrue on secondary debt, hence, this debt will never reach a point where it is above the minimum borrow amount.

Impact

Malicious actors can generate vault accounts which cannot be liquidated. Through opening numerous vault positions, Notional can rack up significant exposure and accrue bad debt as a result.

Code Snippet

https://github.com/sherlock-audit/2023-03-notional-0xleastwood/blob/main/contracts-v2/contracts/external/actions/VaultLiquidationAction.sol#L57-L119

https://github.com/sherlock-audit/2023-03-notional-0xleastwood/blob/main/contracts-v2/contracts/external/actions/VaultLiquidationAction.sol#L239-L267

https://github.com/sherlock-audit/2023-03-notional-0xleastwood/blob/main/contracts-v2/contracts/internal/vaults/VaultSecondaryBorrow.sol#L295-L299

Tool used

Manual Review

Recommendation

Either allow for multiple currencies to be liquidated or ensure that `checkMinBorrow` is performed only on the currency which is being liquidated.

Discussion

T-Woodward

True. Also I don't think you need to settle the accounts. If you call liquidateVaultCashBalance I believe min borrow checks will be skipped on all currencies, so you could just deleverage twice to get positive cash balances on each secondary currency and then liquidateVaultCashBalance on each to get both secondary currency debts below min.

0xleastwood

Escalate for 10 USDC

I think this was wrongly classified as `medium` severity during the judging contest phase. I had confirmed this as a `high` severity finding with the Notional team prior to submission.

Ultimately, the finding states that users can create any number of vault positions where they only have debt in each secondary currency and no primary currency. They are then able to liquidate their debt below `minAccountSecondaryBorrow` and therefore prevent any future liquidation. Repeated abuse of this could lead to protocol insolvency.

sherlock-admin

> Escalate for 10 USDC
>
> I think this was wrongly classified as `medium` severity during the judging contest phase. I had confirmed this as a `high` severity finding with the

Notional team prior to submission.

Ultimately, the finding states that users can create any number of vault positions where they only have debt in each secondary currency and no primary currency. They are then able to liquidate their debt below `minAccountSecondaryBorrow` and therefore prevent any future liquidation. Repeated abuse of this could lead to protocol insolvency.

You've created a valid escalation for 10 USDC!

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.

Trumpero

Agree with the escalation that this issue should be a high

hrishibhat

Result: High Unique Considering this issue as a valid high

sherlock-admin

Escalations have been resolved successfully!

Escalation status:

- 0xleastwood: accepted

jeffywu

Fixed: https://github.com/notional-finance/contracts-v2/pull/138

xiaoming9090

@0xleastwood + @xiaoming9090: Verified. Fixed in PR https://github.com/notional-finance/contracts-v2/pull/138

Issue H-10: Partial liquidations are not possible

Source: https://github.com/sherlock-audit/2023-03-notional-judging/issues/204

Found by

xiaoming90 + 0xleastwood

Summary

Due to an incorrect implementation of `VaultValuation.getLiquidationFactors()`, Notional requires that a liquidator reduces an account's debt below `minBorrowSize`. This does not allow liquidators to partially liquidate a vault account into a healthy position and opens up the protocol to an edge case where an account is always ineligible for liquidation.

Vulnerability Detail

While `VaultValuation.getLiquidationFactors()` might allow for the resultant outstanding debt to be below the minimum borrow amount and non-zero, `deleverageAccount()` will revert due to `checkMinBorrow` being set to `true`. Therefore, the only option is for liquidators to wipe the outstanding debt entirely but users can set up their vault accounts such that that `maxLiquidatorDepositLocal` is less than each of the vault currency's outstanding debt.

https://github.com/sherlock-audit/2023-03-notional-0xleastwood/blob/main/contracts-v2/contracts/internal/vaults/VaultValuation.sol#L240-L270

```
File: VaultValuation.sol
240:         int256 maxLiquidatorDepositLocal = _calculateDeleverageAmount(
241:             vaultConfig,
242:             h.vaultShareValueUnderlying,
243:             h.totalDebtOutstandingInPrimary.neg(),
244:             h.debtOutstanding[currencyIndex].neg(),
245:             minBorrowSize,
246:             exchangeRate,
247:             er.rateDecimals
248:         );
249:
250:         // NOTE: deposit amount is always positive in this method
251:         if (depositUnderlyingInternal < maxLiquidatorDepositLocal) {
252:             // If liquidating past the debt outstanding above the min
 ↪  borrow, then the entire
253:             // debt outstanding must be liquidated.
254:
255:             // (debtOutstanding - depositAmountUnderlying) is the post
 ↪  liquidation debt. As an
```

```
256:             // edge condition, when debt outstanding is discounted to
 ↪  present value, the account
257:             // may be liquidated to zero while their debt outstanding is
 ↪  still greater than the
258:             // min borrow size (which is normally enforced in notional
 ↪  terms -- i.e. non present
259:             // value). Resolving this would require additional complexity
 ↪  for not much gain. An
260:             // account within 20% of the minBorrowSize in a vault that has
 ↪  fCash discounting enabled
261:             // may experience a full liquidation as a result.
262:             require(
263:
 ↪  h.debtOutstanding[currencyIndex].sub(depositUnderlyingInternal) <
 ↪  minBorrowSize,
264:                 "Must Liquidate All Debt"
265:             );
266:         } else {
267:             // If the deposit amount is greater than maxLiquidatorDeposit
 ↪  then limit it to the max
268:             // amount here.
269:             depositUnderlyingInternal = maxLiquidatorDepositLocal;
270:         }
```

If `depositUnderlyingInternal >= maxLiquidatorDepositLocal`, then the liquidator's deposit is capped to `maxLiquidatorDepositLocal`. However, `maxLiquidatorDepositLocal` may put the vault account's outstanding debt below the minimum borrow amount but not to zero.

However, because it is not possible to partially liquidate the account's debt, we reach a deadlock where it isn't possible to liquidate all outstanding debt and it also isn't possible to liquidate debt partially. So even though it may be possible to liquidate an account into a healthy position, the current implementation doesn't always allow for this to be true.

Impact

Certain vault positions will never be eligible for liquidation and hence Notional may be left with bad debt. Liquidity providers will lose funds as they must cover the shortfall for undercollateralised positions.

Code Snippet

https://github.com/sherlock-audit/2023-03-notional-0xleastwood/blob/main/contracts-v2/contracts/external/actions/VaultLiquidationAction.sol#L76-L82

https://github.com/sherlock-audit/2023-03-notional-0xleastwood/blob/main/contracts-v2/contracts/external/actions/VaultAccountHealth.sol#L260-L264

https://github.com/sherlock-audit/2023-03-notional-0xleastwood/blob/main/contracts-v2/contracts/internal/vaults/VaultValuation.sol#L240-L270

Tool used

Manual Review

Recommendation

`VaultValuation.getLiquidationFactors()` must be updated to allow for partial liquidations.

```
File: VaultValuation.sol
251:         if (depositUnderlyingInternal < maxLiquidatorDepositLocal) {
252:             // If liquidating past the debt outstanding above the min
 ↪  borrow, then the entire
253:             // debt outstanding must be liquidated.
254:
255:             // (debtOutstanding - depositAmountUnderlying) is the post
 ↪  liquidation debt. As an
256:             // edge condition, when debt outstanding is discounted to
 ↪  present value, the account
257:             // may be liquidated to zero while their debt outstanding is
 ↪  still greater than the
258:             // min borrow size (which is normally enforced in notional
 ↪  terms -- i.e. non present
259:             // value). Resolving this would require additional complexity
 ↪  for not much gain. An
260:             // account within 20% of the minBorrowSize in a vault that has
 ↪  fCash discounting enabled
261:             // may experience a full liquidation as a result.
262:             require(
263:
 ↪  h.debtOutstanding[currencyIndex].neg().sub(depositUnderlyingInternal) >=
 ↪  minBorrowSize,
                         ||
 ↪  h.debtOutstanding[currencyIndex].neg().sub(depositUnderlyingInternal) == 0
264:                 "Must Liquidate All Debt"
265:             );
266:         } else {
267:             // If the deposit amount is greater than maxLiquidatorDeposit
 ↪  then limit it to the max
268:             // amount here.
269:             depositUnderlyingInternal = maxLiquidatorDepositLocal;
```

```
270:         }
```

Discussion

T-Woodward

This is true. This is an issue when a vaultAccount is borrowing from the prime maturity, but not an issue when an account is borrowing from an fCash maturity.

0xleastwood

Escalate for 10 USDC

We are able to set up a vault account in a way such that liquidating an amount equal or up to `maxLiquidatorDepositLocal` will leave the resultant currency debt above `minBorrowSize` and hence the liquidation will fail. Therefore, we would be in a deadlock position where a user is unable to sufficiently liquidate a vault of considerable size and hence the vault could accrue bad debt. While this only applies to prime vaults, `high` severity is still justified here because it is not possible to liquidate a vault if these conditions are held.

sherlock-admin

> Escalate for 10 USDC
>
> We are able to set up a vault account in a way such that liquidating an amount equal or up to `maxLiquidatorDepositLocal` will leave the resultant currency debt above `minBorrowSize` and hence the liquidation will fail. Therefore, we would be in a deadlock position where a user is unable to sufficiently liquidate a vault of considerable size and hence the vault could accrue bad debt. While this only applies to prime vaults, `high` severity is still justified here because it is not possible to liquidate a vault if these conditions are held.

You've created a valid escalation for 10 USDC!

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.

Trumpero

Agree with the escalation that this should be a high issue.

hrishibhat

Result: High Unique Considering this a high issue based points raised in the issue and the escalation

sherlock-admin

Escalations have been resolved successfully!

Escalation status:

- 0xleastwood: accepted

jeffywu

Fixed: https://github.com/notional-finance/contracts-v2/pull/132

xiaoming9090

@0xleastwood + @xiaoming9090 : Verified. Fixed in PR https://github.com/notional-finance/contracts-v2/pull/132

Issue H-11: Vault accounts with excess cash can avoid being settled

Source: https://github.com/sherlock-audit/2023-03-notional-judging/issues/207

Found by

xiaoming90 + 0xleastwood

Summary

If excess cash was transferred out from an account during account settlement, then the protocol will check the account's collateral ratio and revert if the position is unhealthy. Because it may not be possible to settle a vault account, liquidators cannot reduce account debt by purchasing vault shares because `_authenticateDeleverage()` will check to see if a vault has matured.

Vulnerability Detail

Considering an account's health is determined by a combination of its outstanding debt, cash holdings and the total underlying value of its vault shares, transferring out excess cash may actually put an account in an unhealthy position.

https://github.com/sherlock-audit/2023-03-notional-0xleastwood/blob/main/contracts-v2/contracts/external/actions/VaultAccountAction.sol#L41-L68

```
File: VaultAccountAction.sol
41:     function settleVaultAccount(address account, address vault) external
 ↪  override nonReentrant {
42:         requireValidAccount(account);
43:         require(account != vault);
44:
45:         VaultConfig memory vaultConfig =
 ↪  VaultConfiguration.getVaultConfigStateful(vault);
46:         VaultAccount memory vaultAccount =
 ↪  VaultAccountLib.getVaultAccount(account, vaultConfig);
47:
48:         // Require that the account settled, otherwise we may leave the
 ↪  account in an unintended
49:         // state in this method because we allow it to skip the min borrow
 ↪  check in the next line.
50:         (bool didSettle, bool didTransfer) =
 ↪  vaultAccount.settleVaultAccount(vaultConfig);
51:         require(didSettle, "No Settle");
52:
53:         vaultAccount.accruePrimeCashFeesToDebt(vaultConfig);
```

```
54:
55:         // Skip Min Borrow Check so that accounts can always be settled
56:         vaultAccount.setVaultAccount({vaultConfig: vaultConfig,
↪  checkMinBorrow: false});
57:
58:         if (didTransfer) {
59:             // If the vault did a transfer (i.e. withdrew cash) we have to
↪  check their collateral ratio. There
60:             // is an edge condition where a vault with secondary borrows has
↪  an emergency exit. During that process
61:             // an account will be left some cash balance in both currencies.
↪  It may have excess cash in one and
62:             // insufficient cash in the other. A withdraw of the excess in
↪  one side will cause the vault account to
63:             // be insolvent if we do not run this check. If this scenario
↪  indeed does occur, the vault itself must
64:             // be upgraded in order to facilitate orderly exits for all of
↪  the accounts since they will be prevented
65:             // from settling.
66:
↪  IVaultAccountHealth(address(this)).checkVaultAccountCollateralRatio(vault,
↪  account);
67:         }
68:     }
```

It is important to note that all vault liquidation actions require a vault to first be settled. Hence, through self-liquidation, sophisticated vault accounts can have excess cash in one currency and significant debt holdings in the vault's other currencies.

https://github.com/sherlock-audit/2023-03-notional-0xleastwood/blob/main/contracts-v2/contracts/external/actions/VaultLiquidationAction.sol#L197-L237

```
File: VaultLiquidationAction.sol
197:     function _authenticateDeleverage(
198:         address account,
199:         address vault,
200:         address liquidator
201:     ) private returns (
202:         VaultConfig memory vaultConfig,
203:         VaultAccount memory vaultAccount,
204:         VaultState memory vaultState
205:     ) {
206:         // Do not allow invalid accounts to liquidate
207:         requireValidAccount(liquidator);
208:         require(liquidator != vault);
209:
```

```
210:         // Cannot liquidate self, if a vault needs to deleverage itself as
 ↪  a whole it has other methods
211:         // in VaultAction to do so.
212:         require(account != msg.sender);
213:         require(account != liquidator);
214:
215:         vaultConfig = VaultConfiguration.getVaultConfigStateful(vault);
216:         require(vaultConfig.getFlag(VaultConfiguration.DISABLE_DELEVERAGE)
 ↪  == false);
217:
218:         // Authorization rules for deleveraging
219:         if (vaultConfig.getFlag(VaultConfiguration.ONLY_VAULT_DELEVERAGE)) {
220:             require(msg.sender == vault);
221:         } else {
222:             require(msg.sender == liquidator);
223:         }
224:
225:         vaultAccount = VaultAccountLib.getVaultAccount(account,
 ↪  vaultConfig);
226:
227:         // Vault accounts that are not settled must be settled first by
 ↪  calling settleVaultAccount
228:         // before liquidation. settleVaultAccount is not permissioned so
 ↪  anyone may settle the account.
229:         require(block.timestamp < vaultAccount.maturity, "Must Settle");
230:
231:         if (vaultAccount.maturity == Constants.PRIME_CASH_VAULT_MATURITY) {
232:             // Returns the updated prime vault state
233:             vaultState =
 ↪  vaultAccount.accruePrimeCashFeesToDebtInLiquidation(vaultConfig);
234:         } else {
235:             vaultState = VaultStateLib.getVaultState(vaultConfig,
 ↪  vaultAccount.maturity);
236:         }
237:     }
```

Consider the following example:

Alice has a valid borrow position in the vault which is considered risky. She has a small bit of secondary cash but most of her debt is primary currency denominated. Generally speaking her vault is healthy. Upon settlement, the small bit of excess secondary cash is transferred out and her vault is undercollateralised and eligible for liquidation. However, we are deadlocked because it is not possible to settle the vault because `checkVaultAccountCollateralRatio()` will fail, and it's not possible to purchase the excess cash and offset the debt directly via `liquidateVaultCashBalance()` or `deleverageAccount()` because `_authenticateDeleverage()` will revert if a vault has not yet been settled.

Impact

Vault accounts can create positions which will never be eligible for liquidation and the protocol may accrue bad debt.

Code Snippet

https://github.com/sherlock-audit/2023-03-notional-0xleastwood/blob/main/contracts-v2/contracts/external/actions/VaultAccountAction.sol#L41-L68

https://github.com/sherlock-audit/2023-03-notional-0xleastwood/blob/main/contracts-v2/contracts/external/actions/VaultLiquidationAction.sol#L197-L237

Tool used

Manual Review

Recommendation

Consider adding a liquidation method which settles a vault account and allows for a liquidator to purchase vault shares, offsetting outstanding debt, before performing collateral ratio checks.

Discussion

T-Woodward

We're changing settlement to just leave excess cash in the account's cash balance and not transfer out. This allows us to drop the collateral check and remove this issue.

0xleastwood

Escalate for 10 USDC

This issue identifies a deadlock situation where it is not possible to liquidate a vault unless it has first been settled. But it is also not possible to purchase vault cash and offset debt. Hence, a vault can be considered healthy prior to settlement, but because some excess cash is transferred out and the protocol performs collateral checks via `checkVaultAccountCollateralRatio()`, vault settlement may fail.

I believe the severity of this situation to also be `high` risk because sophisticated users are able to avoid liquidation.

sherlock-admin

> Escalate for 10 USDC
>
> This issue identifies a deadlock situation where it is not possible to liquidate a vault unless it has first been settled. But it is also not possible

> to purchase vault cash and offset debt. Hence, a vault can be considered healthy prior to settlement, but because some excess cash is transferred out and the protocol performs collateral checks via `checkVaultAccountCollateralRatio()`, vault settlement may fail.
>
> I believe the severity of this situation to also be `high` risk because sophisticated users are able to avoid liquidation.

You've created a valid escalation for 10 USDC!

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.

Trumpero

Agree with the escalation, this issue should be a high.

hrishibhat

Result: High Unique Considering this issue as a valid high based on the escalation that certain users can avoid liquidations

sherlock-admin

Escalations have been resolved successfully!

Escalation status:

- 0xleastwood: accepted

jeffywu

Fixed in: https://github.com/notional-finance/contracts-v2/pull/135

0xleastwood

@0xleastwood + @xiaoming9090: PR 135 attempted to fix the issue by removing the transfer of excess assets from the settlement. It will store the excess assets to be refunded in `s.primaryCash` so that it can be read to `vaultAccount.tempCashBalance` later when exiting the vault (`VaultAccountAction.exitVault`). The transfer of excess assets will be performed during the exit vault.

However, in an edge case where a liquidator deleverages another account and happens to have primary cash in their vault account, their `s.primaryCash` (excess cash) will be wiped.

Following is the function call flow for reference: `deleverageAccount` -> `_transferVaultSharesToLiquidator` -> `liquidator.setVaultAccount` -> Effect: liquidator's `s.primaryCash = 0`

jeffywu

I'm not sure this is the case:

```
function _transferVaultSharesToLiquidator(
    address receiver,
    VaultConfig memory vaultConfig,
    uint256 vaultSharesToLiquidator,
    uint256 maturity
) private {
    // Liquidator will receive vault shares that they can redeem by calling
    ↪  exitVault. If the liquidator has a
    // leveraged position on then their collateral ratio will increase
    VaultAccount memory liquidator = VaultAccountLib.getVaultAccount(receiver,
    ↪  vaultConfig);
    // The liquidator must be able to receive the vault shares (i.e. not be in
    ↪  the vault at all or be in the
    // vault at the same maturity). If the liquidator has fCash in the current
    ↪  maturity then their collateral
    // ratio will increase as a result of the liquidation, no need to check
    ↪  their collateral position.
    require(liquidator.maturity == 0 || liquidator.maturity == maturity,
    ↪  "Maturity Mismatch"); // dev: has vault shares
    liquidator.maturity = maturity;
    liquidator.vaultShares = liquidator.vaultShares.add(vaultSharesToLiquidator);
    liquidator.setVaultAccount({vaultConfig: vaultConfig, checkMinBorrow: true,
    ↪  emitEvents: false});
}
```

1. In `getVaultAccount` the vault cash balance will be loaded into `tempCashBalance`
2. In `setVaultAccount` we call `_setVaultAccount`
3. The first line of `_setVaultAccount` requires that the tempCashBalance is equal to zero. This will revert the deleverage transaction before the cash balance is cleared.

0xleastwood

You're right! I've checked this and agree that the flow of execution would look like:

```
VaultLiquidationAction.deleverageAccount
 VaultAccount.getVaultAccount
     liquidator.tempCashBalance = s.primaryCash // Set to non-zero value
 VaultLiquidationAction._transferVaultSharesToLiquidator
     liquidator.setVaultAccount
         VaultAccount._setVaultAccount
             require(vaultAccount.tempCashBalance == 0) // Revert
```

@0xleastwood + @xiaoming9090: Verified. Fixed in https://github.com/notional-finance/contracts-v2/pull/135.

Issue M-1: Lack of ERC20 approval on depositing to external money markets Compound V2

Source: https://github.com/sherlock-audit/2023-03-notional-judging/issues/28

Found by

ShadowForce, mstpr-brainbot

Summary

Notional's current rebalancing process for depositing prime cash underlyings into Compound V2 generates the mint function without approval.

Vulnerability Detail

Governance rebalances the prime cash underlyings by depositing them into various external money market platforms. Currently, Notional only supports Compound V2, for which an oracle and rebalancing strategy have been developed. When Compound V2 deposit call data is generated in the oracle, it only generates the cTokens' mint function without approval. However, it should first approve and then call the mint, as cToken takes the underlying from the Notional proxy and mints the cToken to the Notional proxy.

Upon careful examination of the v2 code, this finding passes tests because the old Notional V2 proxy already has approval for some Compound V2 cTokens. Since the Notional V2 code is not in scope for this contest and the approval situation is not mentioned in the protocol documentation, this finding should be considered valid. Furthermore, if the protocol wants to launch new cTokens for which V2 does not already have approval, the process will fail due to the lack of approval.

Impact

This finding should be considered valid for several reasons:

The issue is not mentioned in the documentation provided by the protocol team. It is crucial for the documentation to be comprehensive, as it serves as a guide for developers and users to understand the intricacies of the protocol. The Notional V2 code is out of scope for the contest. Therefore, the fact that the old Notional V2 proxy already has approval for some Compound V2 cTokens should not be considered a mitigating factor, as the focus should be on the current implementation and its potential issues. Most importantly, this issue could impact the functionality of Notional when attempting to launch new cTokens for Compound V2 that do not already have an allowance in the proxy. The lack of approval would cause the process to fail, effectively limiting the growth and adaptability of the

protocol. In summary, this finding is valid due to its absence in the provided documentation, its relevance to the current implementation rather than the out-of-scope Notional V2 code, and its potential to limit the protocol's functionality when dealing with new cTokens for Compound V2. I'll call it as medium severity after all considerations.

It is important to note that this finding may not be applicable to non-implemented protocol oracles such as AAVE-Euler. In these cases, there is a possibility to create multiple call data deposits, allowing for a more flexible approach. Governance can first generate one call data to approve the required allowances and then generate a subsequent call data to initiate the deposit process.

Code Snippet

https://github.com/sherlock-audit/2023-03-notional/blob/main/contracts-v2/contracts/external/pCash/adapters/CompoundV2AssetAdapter.sol#L45-L48

Tool used

Manual Review

Recommendation

put a check on allowance before deposit something like this:

```
if (IERC20.allowance(address(NotionalProxy), address(cToken))) {
      callData[0] = abi.encodeWithSelector(
        IERC20.approve.selector,
        address(NotionalProxy),
        address(cToken)
      );
    }
```

Discussion

jeffywu

Compound V2 will not be used for rebalancing, however, even if that were not the case this issue would be invalid. Notional currently already has an approval with all listed cTokens and that would still stand after the migration.

mstpr

Escalate for 10 USDC

Current code scope is only covering CompoundV2 and there are rebalancing, deposit and redeeming functions already coded up. It is certain that this code

scope is intended to work with only CompoundV2 since it has full functionality. Protocol team states that CompoundV2 will not be used for rebalancing but there are functions for it and there are not alternative money markets built in aswell so one can easily say that there is only CompoundV2 to interact with prime cash at the current scope.

Approvals from NotionalV2 handles the lack of approval here however, as stated in the contest here: USDT can be used in NotionalV3 which was not existed in NotionalV2 hence, there are no approvals. If USDT would be integrated to the NotionalV3 current code would fail since there is no approval for USDT from NotionalV3 to CompoundV2.

If CompoundV2 will not be used for rebalancing, then why there are rebalancing functions for only CompoundV2 and not others? I think protocol team should've stated that.

sherlock-admin

> Escalate for 10 USDC
>
> Current code scope is only covering CompoundV2 and there are rebalancing, deposit and redeeming functions already coded up. It is certain that this code scope is intended to work with only CompoundV2 since it has full functionality. Protocol team states that CompoundV2 will not be used for rebalancing but there are functions for it and there are not alternative money markets built in aswell so one can easily say that there is only CompoundV2 to interact with prime cash at the current scope.
>
> Approvals from NotionalV2 handles the lack of approval here however, as stated in the contest here: USDT can be used in NotionalV3 which was not existed in NotionalV2 hence, there are no approvals. If USDT would be integrated to the NotionalV3 current code would fail since there is no approval for USDT from NotionalV3 to CompoundV2.
>
> If CompoundV2 will not be used for rebalancing, then why there are rebalancing functions for only CompoundV2 and not others? I think protocol team should've stated that.

You've created a valid escalation for 10 USDC!

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.

Jiaren-tang

Escalate for 10 USDC. I agree with mstpr that this issue should be a valid issue

adding any token or external market would break the rebalance flow because of lack of token approval

sherlock-admin

> Escalate for 10 USDC. I agree with mstpr that this issue should be a valid issue
>
> adding any token or external market would break the rebalance flow because of lack of token approval

You've created a valid escalation for 10 USDC!

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.

hrishibhat

Result: Medium Has duplicates Considering this issue as a valid medium based on the information provided in the readme makes this issue possible when the additional token is added.

sherlock-admin

Escalations have been resolved successfully!

Escalation status:

- mstpr: accepted
- ShadowForce: accepted

Issue M-2: Fee on transfer tokens will break the withdrawing process

Source: https://github.com/sherlock-audit/2023-03-notional-judging/issues/34

Found by

mstpr-brainbot

Summary

If a currency has a built-in transfer fee, withdrawing prime cash may be impossible due to accounting discrepancies.

Vulnerability Detail

Example: Alice has 100 pUSDT, equivalent to 105 USDT, and assume that all the underlying USDT is in Compound V3 (in form of cUSDT), earning interest.

When Alice withdraws the prime cash using the `withdraw()` function in `AccountsAction.sol`, the function checks if the corresponding underlying (105 USDT) is available in the contract. Since all the USDT is lent out in Compound, Notional initiates the redemption process. The redemption process attempts to withdraw 105 USDT worth of cUSDT from Compound. However, due to transfer fees on USDT, redeeming 105 USDT worth of cUSDT results in approximately 104.9 USDT. The require check ensures that Notional must withdraw 105 USDT or more, but in reality, only 104.9 USDT is withdrawn, causing the function to revert consistently.

Impact

Since this is an unlikely scenario I'll label it as medium.

However, if fee on transfer tokens will be used this can be a high finding since withdrawals will not go through at all. USDT can open it's transfer functionality so that should be also taken into consideration if such thing happens.

Code Snippet

https://github.com/sherlock-audit/2023-03-notional/blob/main/contracts-v2/contracts/external/actions/AccountAction.sol#L173

https://github.com/sherlock-audit/2023-03-notional/blob/main/contracts-v2/contracts/internal/balances/TokenHandler.sol#L220-L247

https://github.com/sherlock-audit/2023-03-notional/blob/main/contracts-v2/contracts/internal/balances/TokenHandler.sol#L249-L278

revert lines https://github.com/sherlock-audit/2023-03-notional/blob/main/contracts-v2/contracts/internal/balances/TokenHandler.sol#L383

https://github.com/sherlock-audit/2023-03-notional/blob/main/contracts-v2/contracts/internal/balances/TokenHandler.sol#L277

Tool used

Manual Review

Recommendation

Instead of promising the underlying amount on withdrawals, just return the withdrawn pcashs corresponding yield tokens underlying amount and let users endorse the loss

Discussion

jeffywu

This is somewhat true, although if this really happened and we needed to manage it the PrimeCashHoldingsOracle could return a lower external balance to account for the transfer fee.

Jiaren-tang

Escalate for 10 USDC. fee on transfer is not in scope and this issue should not be a seperate medium

the onchain context is:

> DEPLOYMENT: Currently Mainnet, considering Arbitrum and Optimisim in the near future. ERC20: Any Non-Rebasing token. ex. USDC, DAI, USDT (future), wstETH, WETH, WBTC, FRAX, CRV, etc. ERC721: None ERC777: None FEE-ON-TRANSFER: None planned, some support for fee on transfer

clearly none of the supported ERC20 token is fee-on-transfer token

and the protocol clearly indicate

> FEE-ON-TRANSFER: None planned, some support for fee on transfer

sherlock-admin

> Escalate for 10 USDC. fee on transfer is not in scope and this issue should not be a seperate medium

the onchain context is:

> DEPLOYMENT: Currently Mainnet, considering Arbitrum and Optimisim in the near future. ERC20: Any Non-Rebasing token. ex. USDC, DAI, USDT (future), wstETH, WETH, WBTC, FRAX, CRV, etc. ERC721: None ERC777: None FEE-ON-TRANSFER: None planned, some support for fee on transfer

clearly none of the supported ERC20 token is fee-on-transfer token

and the protocol clearly indicate

> FEE-ON-TRANSFER: None planned, some support for fee on transfer

You've created a valid escalation for 10 USDC!

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.

hrishibhat

Result: Medium Unique Considering this a valid issue as the readme indicates support of Fee-on-Transfer token is intended.

sherlock-admin

Escalations have been resolved successfully!

Escalation status:

- ShadowForce: rejected

Issue M-3: convertFromStorage() fails to use rounding-up when converting a negative storedCashBalance into signedPrimeSupplyValue.

Source: https://github.com/sherlock-audit/2023-03-notional-judging/issues/70

Found by

chaduke

Summary

`convertFromStorage()` fails to use rounding-up when converting a negative `storedCashBalance` into `signedPrimeSupplyValue`.

Vulnerability Detail

`convertFromStorage()` is used to convert `storedCashBalance` into `signedPrimeSupplyValue`. When `storedCashBalance` is negative, it represents a debt - positive prime cash owed.

https://github.com/sherlock-audit/2023-03-notional/blob/main/contracts-v2/contracts/internal/pCash/PrimeRateLib.sol#L60-L74

Unfortunately, when converting a negative `storedCashBalance` into `signedPrimeSupplyValue`, the following division will apply a rounding-down (near zero) mode, leading to a user to owe less than it is supposed to be.

```
return storedCashBalance.mul(pr.debtFactor).div(pr.supplyFactor);
```

This is not acceptable. Typically, rounding should be in favor of the protocol, not in favor of the user to prevent draining of the protocol and losing funds of the protocol.

The following POC shows a rounding-down will happen for a negative value division. The result of the following test is -3.

```
function testMod() public {

        int256 result = -14;
        result = result / 4;
        console2.logInt(result);
    }
```

Impact

`convertFromStorage()` fails to use rounding-up when converting a negative `storedCashBalance` into `signedPrimeSupplyValue`. The protocol is losing some dusts amount, but it can be accumulative or a vulnerability that can be exploited.

Code Snippet

Tool used

VSCode

Manual Review

Recommendation

Use rounding-up instead.

```
function convertFromStorage(
        PrimeRate memory pr,
        int256 storedCashBalance
    ) internal pure returns (int256 signedPrimeSupplyValue) {
        if (storedCashBalance >= 0) {
            return storedCashBalance;
        } else {
            // Convert negative stored cash balance to signed prime supply value
            // signedPrimeSupply = (negativePrimeDebt * debtFactor) /
 ↪  supplyFactor

            // cashBalance is stored as int88, debt factor is uint80 * uint80 so
 ↪  there
            // is no chance of phantom overflow (88 + 80 + 80 = 248) on mul
-           return storedCashBalance.mul(pr.debtFactor).div(pr.supplyFactor);
+         return (storedCashBalance.mul(pr.debtFactor).sub(pr.supplyFactor-1)).d
 ↪  iv(pr.supplyFactor);
        }
    }
```

Issue M-4: Compound exchange rate can be manipulated to withdraw more underlying tokens from NotionalV3

Source: https://github.com/sherlock-audit/2023-03-notional-judging/issues/122

Found by

mstpr-brainbot

Summary

Prime cash underlying values can be manipulated via donating the underlying to cToken contract to inflate the cToken price and then withdrawing from Notional for more.

Vulnerability Detail

Prime supply underlying value is calculated via few formulas and math operations let's check them:

To calculate the underlying interest rate: (currentUnderlyingValue - lastUnderlyingValue) / lastUnderlyingValue which we can say that underlying value can be in forms of DAI or cDAI (since the other protocol handlers are not in the audit scope I'll treat it as there is only CompoundV2 available for prime cash external lenders) We can derive this formula for more context: (((exchangeRate * totalCDAIHoldings) + totalDAIHolds) - ((previousExchangeRate * totalCDAIHoldings) + totalDAIHolds)) / ((previousExchangeRate * totalCDAIHoldings) + totalDAIHolds) = underlyingInterestRate

To calculate the underlying scalar: lastScalar * (1 + underlyingInterestRate) = underlyingScalar

To calculate the supply factor: supplyScalar * underlyingScalar = supplyFactor

To calculate the underlying final: primeCashValue * supplyFactor = underlyingFinal

now considering these, if someone can manipulate the currentUnderlying (total underlying tokens that Notional holds) up to some level that they can make the prime cash value higher, than there will be a profit. Currently, NotionalV3 deposits the funds to CompoundV2 which the exchange rate is calculated as follows:

(totalCash + totalBorrows - totalReserves) / totalCDAISupply

where as totalCash is the balanceOf(token) in the cToken contract. This means that the donations can manipulate the exchange rate and make Notional V3 underlying oracle trick that the tokens worths more. That means the exchange rate can be calculated as this:

(airDropAmount + totalCash + totalBorrows - totalReserves) / totalCDAISupply

if we can find a scenario where underlyingFinal > airDropAmount + initialPrimeCash, someone can do a flash loan attack

Let's draw a scenario where this can happen , Assume cDAI has very low liquidity because Compound incentives the v3 usage, Notional is withdrawing its CompoundV2 positions slowly to DAI and getting ready to deploy them to CompoundV3. For this scenario case assume there are total of 100K cDAI which all of them hold by Notional (this can change, just for simplicity I am keeping it), now if attacker donates 50K DAI to cDAI attacker will make the new exchange rate 1.5. Here we go:

1- There are 220K DAI idle and 100K cDAI in Notional (for easiness let's assume 1cDAI == 1DAI) 2- Alice flashloans 4.83M DAI and deposits to Notional receiving 4.83M pDAI (we assume supplyFactor is 1, which is possible if we are early stages after migration, no interest accrued). Alice will make the total idle DAI balance in Notional as 5M after her deposit 3- Alice deposits the 50K DAI to cDAI contract to make the exchange rate 1.5 (100KDAI/100KCDAI was the initial state of cDAI, now 150K DAI / 100K CDAI = 1.5) 4- Alice withdraws the 4.83M pDAI for 5M DAI

Let's prove the math, lastUnderlying = (100K cDAI * 1 + 5M) = 5.1M

currentUnderlying = (100K cDAI * 1.5 + 5M) = 5.150M

underlyingInterestRate = (5.15M - 5.1M) / 5.1M = 0.045

underlyingScalar = 1 * (0.045+ 1) = 1.045

supplyScalar = 1 * 1.045 = 1.045

Result: Alice deposited 4.78M DAI which she got 4.78M pDAI for it, now 4.78M pDAI worths 4.78M * 1.045 = 5M DAI (approx)

Alice flashloaned 4.78M + 50K = 4.83M total flash loaned amount

5M - 4.83M = 170K DAI profit after her attack

In the end, Alice stole the 220K idle balance.

Impact

Although the above example considers the compound cDAI has very low liquidity, the same scenario could happen with big numbers but that would require Notional to hold lots of liquidity. Since compound is literally incentivizing the compoundv3 over compoundv2 I assumed that scenario is not far away from reality. We also assumed supplyScalar was 1, which is possible since the prime market will start as 1 but also it doesn't really matter since we multiply with lastScalar and supplyScalar on both sides of equation, again it was 1 for simplicity.

Code Snippet

https://github.com/sherlock-audit/2023-03-notional/blob/main/contracts-v2/contracts/external/actions/AccountAction.sol#L173-L210

https://github.com/sherlock-audit/2023-03-notional/blob/main/contracts-v2/contracts/internal/balances/BalanceHandler.sol#L525

https://github.com/sherlock-audit/2023-03-notional/blob/main/contracts-v2/contracts/internal/pCash/PrimeCashExchangeRate.sol#L618-L641

https://github.com/sherlock-audit/2023-03-notional/blob/main/contracts-v2/contracts/internal/pCash/PrimeCashExchangeRate.sol#L535-L596

https://github.com/sherlock-audit/2023-03-notional/blob/main/contracts-v2/contracts/internal/balances/BalanceHandler.sol#L143-L160

https://github.com/sherlock-audit/2023-03-notional/blob/main/contracts-v2/contracts/internal/balances/TokenHandler.sol#L231-L234

https://github.com/sherlock-audit/2023-03-notional/blob/main/contracts-v2/contracts/internal/pCash/PrimeRateLib.sol#L301-L306

Compound stuff

Tool used

Manual Review

Recommendation

Acknowledge that CompoundV2 exchange rate is exploitable via donating, if the CompoundV2 cTokens has low liquidity, donate attacks will be quite cheap which can cause pDAI manipulations aswell. If the CompoundV2 cToken liquidity goes below some level abort the CompoundV2 strategy and migrate to other money markets where liquidity is decent and exchange rate is not manipulatable.

Discussion

jeffywu

This attack vector is known and mitigations have been put in place against it (we've already withdrawn from Compound V2). Furthermore, we've only listed large cap Compound V2 markets so this attack would be very expensive against prime cash.

mstpr

Escalate for 10 USDC

At the moment, the contest only includes the CompoundV2 adapter. From this, we could infer that the primary cash will be deposited into CompoundV2. In the future,

CompoundV3 and Aave may also be integrated. However, within the current scope of the code, it only operates with CompoundV2. Consequently, we can safely assert that prime cash will be lent to CompoundV2. This is evident from the scope of the code, which includes all the necessary implementations such as rebalancing, redeeming, and depositing.

While I concur that the attack necessitates low liquidity in the CompoundV2 markets and that it's challenging to implement on-chain countermeasures, I believe this risk should be outlined by the protocol team in their documentation. Looking at the Sherlock docs, it's clear that the protocol team should account for possible attack vectors.

Which is exactly what this issue fits, the attack exists and protocol team did not include the attack in the docs. According to this, this finding should be a medium

sherlock-admin

> Escalate for 10 USDC
>
> At the moment, the contest only includes the CompoundV2 adapter. From this, we could infer that the primary cash will be deposited into CompoundV2. In the future, CompoundV3 and Aave may also be integrated. However, within the current scope of the code, it only operates with CompoundV2. Consequently, we can safely assert that prime cash will be lent to CompoundV2. This is evident from the scope of the code, which includes all the necessary implementations such as rebalancing, redeeming, and depositing.
>
> While I concur that the attack necessitates low liquidity in the CompoundV2 markets and that it's challenging to implement on-chain countermeasures, I believe this risk should be outlined by the protocol team in their documentation. Looking at the Sherlock docs, it's clear that the protocol team should account for possible attack vectors.
>
> Which is exactly what this issue fits, the attack exists and protocol team did not include the attack in the docs. According to this, this finding should be a medium

You've created a valid escalation for 10 USDC!

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.

hrishibhat

Result: Medium Unique Considering this issue as a valid medium, based on the conditions required and given that the Compound v2 integration was still in scope for the contest.

sherlock-admin

Escalations have been resolved successfully!

Escalation status:

- mstpr: accepted

Issue M-5: Cannot permissionless settle the vault account if the user use a blacklisted account

Source: https://github.com/sherlock-audit/2023-03-notional-judging/issues/155

Found by

ShadowForce

Summary

Cannot permissionless settle the vault account if the user use a blacklisted account

Vulnerability Detail

In VaultAccoutnAction.sol, one of the critical function is

```
/// @notice Settles a matured vault account by transforming it from an fCash
↪  maturity into
/// a prime cash account. This method is not authenticated, anyone can settle a
↪  vault account
/// without permission. Generally speaking, this action is economically
↪  equivalent no matter
/// when it is called. In some edge conditions when the vault is holding prime
↪  cash, it is
/// advantageous for the vault account to have this called sooner. All vault
↪  account actions
/// will first settle the vault account before taking any further actions.
/// @param account the address to settle
/// @param vault the vault the account is in
function settleVaultAccount(address account, address vault) external override
↪  nonReentrant {
    requireValidAccount(account);
    require(account != vault);

    VaultConfig memory vaultConfig =
↪  VaultConfiguration.getVaultConfigStateful(vault);
    VaultAccount memory vaultAccount = VaultAccountLib.getVaultAccount(account,
↪  vaultConfig);

    // Require that the account settled, otherwise we may leave the account in
↪  an unintended
    // state in this method because we allow it to skip the min borrow check in
↪  the next line.
    (bool didSettle, bool didTransfer) =
↪  vaultAccount.settleVaultAccount(vaultConfig);
```

```
    require(didSettle, "No Settle");

    vaultAccount.accruePrimeCashFeesToDebt(vaultConfig);

    // Skip Min Borrow Check so that accounts can always be settled
    vaultAccount.setVaultAccount({vaultConfig: vaultConfig, checkMinBorrow:
↪   false});

    if (didTransfer) {
        // If the vault did a transfer (i.e. withdrew cash) we have to check
↪   their collateral ratio. There
        // is an edge condition where a vault with secondary borrows has an
↪   emergency exit. During that process
        // an account will be left some cash balance in both currencies. It may
↪   have excess cash in one and
        // insufficient cash in the other. A withdraw of the excess in one side
↪   will cause the vault account to
        // be insolvent if we do not run this check. If this scenario indeed
↪   does occur, the vault itself must
        // be upgraded in order to facilitate orderly exits for all of the
↪   accounts since they will be prevented
        // from settling.

↪   IVaultAccountHealth(address(this)).checkVaultAccountCollateralRatio(vault,
↪   account);
    }
}
```

as the comment suggests, this function should be called permissionless

and the comment is, which means there should not be able to permissionless reject account settlement

```
/// will first settle the vault account before taking any further actions.
```

this is calling

```
(bool didSettle, bool didTransfer) =
↪   vaultAccount.settleVaultAccount(vaultConfig);
```

which calls

```
/// @notice Settles a matured vault account by transforming it from an fCash
↪   maturity into
/// a prime cash account. This method is not authenticated, anyone can settle a
↪   vault account
```


```
/// without permission. Generally speaking, this action is economically
↪  equivalent no matter
/// when it is called. In some edge conditions when the vault is holding prime
↪  cash, it is
/// advantageous for the vault account to have this called sooner. All vault
↪  account actions
/// will first settle the vault account before taking any further actions.
/// @param account the address to settle
/// @param vault the vault the account is in
function settleVaultAccount(address account, address vault) external override
↪  nonReentrant {
    requireValidAccount(account);
    require(account != vault);

    VaultConfig memory vaultConfig =
↪  VaultConfiguration.getVaultConfigStateful(vault);
    VaultAccount memory vaultAccount = VaultAccountLib.getVaultAccount(account,
↪  vaultConfig);

    // Require that the account settled, otherwise we may leave the account in
↪  an unintended
    // state in this method because we allow it to skip the min borrow check in
↪  the next line.
    (bool didSettle, bool didTransfer) =
↪  vaultAccount.settleVaultAccount(vaultConfig);
    require(didSettle, "No Settle");
```

basically this calls

```
// Calculates the net settled cash if there is any temp cash balance that is net
↪  off
// against the settled prime debt.
bool didTransferPrimary;
(accountPrimeStorageValue, didTransferPrimary) =
↪  repayAccountPrimeDebtAtSettlement(
    vaultConfig.primeRate,
    primeVaultState,
    vaultConfig.borrowCurrencyId,
    vaultConfig.vault,
    vaultAccount.account,
    vaultAccount.tempCashBalance,
    accountPrimeStorageValue
);
```

calling

```
function repayAccountPrimeDebtAtSettlement(
```

```
    PrimeRate memory pr,
    VaultStateStorage storage primeVaultState,
    uint16 currencyId,
    address vault,
    address account,
    int256 accountPrimeCash,
    int256 accountPrimeStorageValue
) internal returns (int256 finalPrimeDebtStorageValue, bool didTransfer) {
    didTransfer = false;
    finalPrimeDebtStorageValue = accountPrimeStorageValue;

    if (accountPrimeCash > 0) {
        // netPrimeDebtRepaid is a negative number
        int256 netPrimeDebtRepaid = pr.convertUnderlyingToDebtStorage(
            pr.convertToUnderlying(accountPrimeCash).neg()
        );

        int256 netPrimeDebtChange;
        if (netPrimeDebtRepaid < accountPrimeStorageValue) {
            // If the net debt change is greater than the debt held by the
↪   account, then only
            // decrease the total prime debt by what is held by the account. The
↪   residual amount
            // will be refunded to the account via a direct transfer.
            netPrimeDebtChange = accountPrimeStorageValue;
            finalPrimeDebtStorageValue = 0;

            int256 primeCashRefund = pr.convertFromUnderlying(
                pr.convertDebtStorageToUnderlying(netPrimeDebtChange.sub(account
↪   PrimeStorageValue))
            );
            TokenHandler.withdrawPrimeCash(
                account, currencyId, primeCashRefund, pr, false // ETH will be
↪   transferred natively
            );
            didTransfer = true;
        } else {
            // In this case, part of the account's debt is repaid.
            netPrimeDebtChange = netPrimeDebtRepaid;
            finalPrimeDebtStorageValue =
↪   accountPrimeStorageValue.sub(netPrimeDebtRepaid);
        }
```

the token withdrawal logic above try to push ETH to accout

```
TokenHandler.withdrawPrimeCash(
```

```
    account, currencyId, primeCashRefund, pr, false // ETH will be transferred
↪   natively
);
```

this is calling

```
function withdrawPrimeCash(
      address account,
      uint16 currencyId,
      int256 primeCashToWithdraw,
      PrimeRate memory primeRate,
      bool withdrawWrappedNativeToken
  ) internal returns (int256 netTransferExternal) {
      if (primeCashToWithdraw == 0) return 0;
      require(primeCashToWithdraw < 0);

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

      // Overflow not possible due to int256
      uint256 withdrawAmount = uint256(netTransferExternal.neg());
      _redeemMoneyMarketIfRequired(currencyId, underlying, withdrawAmount);

      if (underlying.tokenType == TokenType.Ether) {
          GenericToken.transferNativeTokenOut(account, withdrawAmount,
↪   withdrawWrappedNativeToken);
      } else {
          GenericToken.safeTransferOut(underlying.tokenAddress, account,
↪   withdrawAmount);
      }

      _postTransferPrimeCashUpdate(account, currencyId, netTransferExternal,
↪   underlying, primeRate);
  }
```

note the function call

```
if (underlying.tokenType == TokenType.Ether) {
    GenericToken.transferNativeTokenOut(account, withdrawAmount,
↪   withdrawWrappedNativeToken);
} else {
    GenericToken.safeTransferOut(underlying.tokenAddress, account,
↪   withdrawAmount);
}
```

if the token type is not ETHER,

we are transfer the underlying ERC20 token to the account

```
GenericToken.safeTransferOut(underlying.tokenAddress, account, withdrawAmount);
```

the token in-scoped is

```
ERC20:  Any Non-Rebasing token. ex. USDC, DAI, USDT (future), wstETH, WETH,
↪  WBTC, FRAX, CRV, etc.
```

USDC is common token that has blacklisted

if the account is blacklisted, the transfer would revert and the account cannot be settled!

Impact

what are the impact,

per comment

```
/// will first settle the vault account before taking any further actions.
```

if that is too vague, I can list three, there are more!

1. there are certain action that need to be done after the vault settlement, for example, liqudation require the vault settlement first

https://github.com/notional-finance/contracts-v2/blob/b20a45c912785fab5f2b62992e5260f44dbae197/contracts/external/actions/VaultLiquidationAction.sol#L229

2. there are case that require force vault settlement, actually one example is notional need to force the settle the vault during migration! (this is just the case to show user should be able to permissionless reject settlement)

Code Snippet

https://github.com/notional-finance/contracts-v2/blob/b20a45c912785fab5f2b62992e5260f44dbae197/contracts/internal/balances/TokenHandler.sol#L241

Tool used

Manual Review

Recommendation

maybe let admin bypass the withdrawPrimeCash and force settle the account to not let settlement block further action!

Discussion

jeffywu

Valid issue, transfers during supposedly permissionless settlement can indeed cause issues.

jeffywu

Fixed: https://github.com/notional-finance/contracts-v2/pull/135

0xleastwood

@0xleastwood + @xiaoming9090: PR 135 attempted to fix the issue by removing the transfer of excess assets from the settlement. It will store the excess assets to be refunded in `s.primaryCash` so that it can be read to `vaultAccount.tempCashBalance` later when exiting the vault (`VaultAccountAction.exitVault`). The transfer of excess assets will be performed during the exit vault.

However, in an edge case where a liquidator deleverages another account and happens to have primary cash in their vault account, their `s.primaryCash` (excess cash) will be wiped.

Following is the function call flow for reference: `deleverageAccount` -> `_transferVaultSharesToLiquidator` -> `liquidator.setVaultAccount` -> Effect: liquidator's `s.primaryCash = 0`

0xleastwood

@0xleastwood + @xiaoming9090: Verified fix as per comments in #207. Fixed in https://github.com/notional-finance/contracts-v2/pull/135

Issue M-6: getAccountPrimeDebtBalance() always return 0

Source: https://github.com/sherlock-audit/2023-03-notional-judging/issues/173

Found by

bin2chen

Summary

Spelling errors that result in `getAccountPrimeDebtBalance()` Always return 0

Vulnerability Detail

`getAccountPrimeDebtBalance()` use for Show current debt

```
function getAccountPrimeDebtBalance(uint16 currencyId, address account) external
 ↪  view override returns (
    int256 debtBalance
) {
    mapping(address => mapping(uint256 => BalanceStorage)) storage store =
 ↪  LibStorage.getBalanceStorage();
    BalanceStorage storage balanceStorage = store[account][currencyId];
    int256 cashBalance = balanceStorage.cashBalance;

    // Only return cash balances less than zero
    debtBalance = cashBalance < 0 ? debtBalance : 0;   //<------@audit wrong,
 ↪  Always return 0
}
```

In the above code we can see that due to a spelling error, `debtBalance` always ==0 should use `debtBalance = cashBalance < 0 ? cashBalance : 0;`

Impact

`getAccountPrimeDebtBalance()` is the external method to check the debt If a third party integrates with notional protocol, this method will be used to determine whether the user has debt or not and handle it accordingly, which may lead to serious errors in the third party's business

Code Snippet

https://github.com/sherlock-audit/2023-03-notional/blob/main/contracts-v2/contracts/external/Views.sol#L496

Tool used

Manual Review

Recommendation

```
    function getAccountPrimeDebtBalance(uint16 currencyId, address account)
↪   external view override returns (
        int256 debtBalance
    ) {
        mapping(address => mapping(uint256 => BalanceStorage)) storage store =
↪   LibStorage.getBalanceStorage();
        BalanceStorage storage balanceStorage = store[account][currencyId];
        int256 cashBalance = balanceStorage.cashBalance;

        // Only return cash balances less than zero
-       debtBalance = cashBalance < 0 ? debtBalance : 0;
+       debtBalance = cashBalance < 0 ? cashBalance : 0;
    }
```

Discussion

jeffywu

Fixed in: https://github.com/notional-finance/contracts-v2/pull/134

xiaoming9090

@0xleastwood + @xiaoming9090 : Verified. Fixed in PR https://github.com/notional-finance/contracts-v2/pull/134

Issue M-7: A single external protocol can DOS rebalancing process

Source: https://github.com/sherlock-audit/2023-03-notional-judging/issues/175

Found by

0xGoodess, chaduke, xiaoming90 + 0xleastwood

Summary

A failure in an external money market can DOS the entire rebalance process in Notional.

Vulnerability Detail

https://github.com/sherlock-audit/2023-03-notional/blob/main/contracts-v2/contracts/external/pCash/ProportionalRebalancingStrategy.sol#L23

```
File: ProportionalRebalancingStrategy.sol
23:     function calculateRebalance(
24:         IPrimeCashHoldingsOracle oracle,
25:         uint8[] calldata rebalancingTargets
26:     ) external view override onlyNotional returns (RebalancingData memory
↪  rebalancingData) {
27:         address[] memory holdings = oracle.holdings();
..SNIP..
40:         for (uint256 i; i < holdings.length;) {
41:             address holding = holdings[i];
42:             uint256 targetAmount = totalValue * rebalancingTargets[i] /
↪  uint256(Constants.PERCENTAGE_DECIMALS);
43:             uint256 currentAmount = values[i];
44:
45:             redeemHoldings[i] = holding;
46:             depositHoldings[i] = holding;
..SNIP..
61:         }
62:
63:         rebalancingData.redeemData =
↪  oracle.getRedemptionCalldataForRebalancing(redeemHoldings, redeemAmounts);
64:         rebalancingData.depositData =
↪  oracle.getDepositCalldataForRebalancing(depositHoldings, depositAmounts);
65:     }
```

During a rebalance, the `ProportionalRebalancingStrategy` will loop through all the holdings and perform a deposit or redemption against the external market of the

holdings.

Assume that Notional integrates with four (4) external money markets (Aave V2, Aave V3, Compound V3, Morpho). In this case, whenever a rebalance is executed, Notional will interact with all four external money markets.

https://github.com/sherlock-audit/2023-03-notional/blob/main/contracts-v2/contracts/external/actions/TreasuryAction.sol#L304

```
File: TreasuryAction.sol
304:     function _executeDeposits(Token memory underlyingToken, DepositData[]
↪  memory deposits) private {
..SNIP..
316:             for (uint256 j; j < depositData.targets.length; ++j) {
317:                 // This will revert if the individual call reverts.
318:                 GenericToken.executeLowLevelCall(
319:                     depositData.targets[j],
320:                     depositData.msgValue[j],
321:                     depositData.callData[j]
322:                 );
323:             }
```

https://github.com/sherlock-audit/2023-03-notional/blob/main/contracts-v2/contracts/internal/balances/TokenHandler.sol#L357

```
File: TokenHandler.sol
357:     function executeMoneyMarketRedemptions(
..SNIP..
373:             for (uint256 j; j < data.targets.length; j++) {
374:                 // This will revert if the individual call reverts.
375:                 GenericToken.executeLowLevelCall(data.targets[j], 0,
↪  data.callData[j]);
376:             }
```

However, as long as one external money market reverts, the entire rebalance process will be reverted and Notional would not be able to rebalance its underlying assets.

The call to the external money market can revert due to many reasons, which include the following:

- Changes in the external protocol's interfaces (e.g. function signatures modified or functions added or removed)
- The external protocol is paused
- The external protocol has been compromised
- The external protocol suffers from an upgrade failure causing an error in the

new contract code.

Impact

Notional would not be able to rebalance its underlying holding if one of the external money markets causes a revert. The probability of this issue occurring increases whenever Notional integrates with a new external money market

The key feature of Notional V3 is to allow its Treasury Manager to rebalance underlying holdings into various other money market protocols.

This makes Notional more resilient to issues in external protocols and future-proofs the protocol. If rebalancing does not work, Notional will be unable to move its fund out of a vulnerable external market, potentially draining protocol funds if this is not mitigated.

Another purpose of rebalancing is to allow Notional to allocate Notional V3's capital to new opportunities or protocols that provide a good return. If rebalancing does not work, the protocol and its users will lose out on the gain from the investment.

On the other hand, if an external monkey market that Notional invested in is consistently underperforming or yielding negative returns, Notional will perform a rebalance to reallocate its funds to a better market. However, if rebalancing does not work, they will be stuck with a suboptimal asset allocation, and the protocol and its users will incur losses.

Code Snippet

https://github.com/sherlock-audit/2023-03-notional/blob/main/contracts-v2/contracts/external/pCash/ProportionalRebalancingStrategy.sol#L23

https://github.com/sherlock-audit/2023-03-notional/blob/main/contracts-v2/contracts/external/actions/TreasuryAction.sol#L304

https://github.com/sherlock-audit/2023-03-notional/blob/main/contracts-v2/contracts/internal/balances/TokenHandler.sol#L357

Tool used

Manual Review

Recommendation

Consider implementing a more resilient rebalancing process that allows for failures in individual external money markets. For instance, Notional could catch reverts from individual money markets and continue the rebalancing process with the remaining markets.

Discussion

jeffywu

Fixed: https://github.com/notional-finance/contracts-v2/pull/117

0xleastwood

@0xleastwood + @xiaoming9090: The MM redemption failure has been handled in PR https://github.com/notional-finance/contracts-v2/pull/117 so that a single redemption failure/revert will not DOS the entire rebalance process. However, it was observed that a single failure/revert when depositing into MMs will still DOS the entire rebalance process. Thus, this issue is partially fixed.

jeffywu

@0xleastwood, having the contract revert during deposits on a single money market is intended here. Our reasoning is that it is always safer to hold underlying funds rather than put them on money markets and forgo the additional yield. Therefore, redemptions will always succeed as much as possible, but deposits will fail. The effect here is that the contract will always hold excess underlying if there are any money market failures, sort of like automatically going into "safe mode" if anything is reverting.

Furthermore, having deposits succeed but redemptions fail will cause issues with calculations of how much to deposit and redeem if the contract is attempting to go from one money market (which fails redemption) and another money market (where we try to deposit funds that were never redeemed). We decided the added complexity here was not worth the additional benefit.

0xleastwood

> @0xleastwood, having the contract revert during deposits on a single money market is intended here. Our reasoning is that it is always safer to hold underlying funds rather than put them on money markets and forgo the additional yield. Therefore, redemptions will always succeed as much as possible, but deposits will fail. The effect here is that the contract will always hold excess underlying if there are any money market failures, sort of like automatically going into "safe mode" if anything is reverting.
>
> Furthermore, having deposits succeed but redemptions fail will cause issues with calculations of how much to deposit and redeem if the contract is attempting to go from one money market (which fails redemption) and another money market (where we try to deposit funds that were never redeemed). We decided the added complexity here was not worth the additional benefit.

I agree that this is the correct and is the most safe approach. Consider the case where the desired portions of MMs is (AAVE, Compound, Morpho) at (30%, 30%, 40%) and it is currently at (40%, 40%, 20%). 20% would be redeemed from Morpho

and 10% would be deposited into AAVE and Compound. Handling redemptions failures by skipping the deposit step makes sense to keep underlying funds available and "safe".

However, we are not actually adhering to this idea of safety where in the example provided, a Morpho MM redemption succeeds but we are unable to deposit 10% of funds into AAVE. Ultimately, we should silently handle a failed MM deposit as we are effectively keeping funds available that way.

0xleastwood

Otherwise, we are keeping the same dependency on protocols being able to DoS the rebalancing process. Priority should be made to keep funds available by allowing MMs to always redeem but preventing any single MM from DoS'ing via deposit.

0xleastwood

Circling back to this, it appears that reverts are already handled correctly. The comment on this line made it seem that this was not the case.

https://github.com/notional-finance/contracts-v2/pull/117/files#diff-293f4ba7dc1c8e8b05afa4825c463d18cacec0625a52bb897a38bd1046c18c31R336

0xleastwood

@0xleastwood + @xiaoming9090: Verified. Fixed in https://github.com/notional-finance/contracts-v2/pull/117

Issue M-8: Inadequate slippage control

Source: https://github.com/sherlock-audit/2023-03-notional-judging/issues/177

Found by

xiaoming90 + 0xleastwood

Summary

The current slippage control mechanism checks a user's acceptable interest rate limit against the post-trade rate, which could result in trades proceeding at rates exceeding the user's defined limit.

Vulnerability Detail

https://github.com/sherlock-audit/2023-03-notional/blob/main/contracts-v2/contracts/internal/markets/InterestRateCurve.sol#L421

```
File: InterestRateCurve.sol
421:     function _getNetCashAmountsUnderlying(
422:         InterestRateParameters memory irParams,
423:         MarketParameters memory market,
424:         CashGroupParameters memory cashGroup,
425:         int256 totalCashUnderlying,
426:         int256 fCashToAccount,
427:         uint256 timeToMaturity
428:     ) private pure returns (int256 postFeeCashToAccount, int256
 ↪  netUnderlyingToMarket, int256 cashToReserve) {
429:         uint256 utilization = getfCashUtilization(fCashToAccount,
 ↪  market.totalfCash, totalCashUnderlying);
430:         // Do not allow utilization to go above 100 on trading
431:         if (utilization > uint256(Constants.RATE_PRECISION)) return (0, 0,
 ↪  0);
432:         uint256 preFeeInterestRate = getInterestRate(irParams, utilization);
433:
434:         int256 preFeeCashToAccount = fCashToAccount.divInRatePrecision(
435:             getfCashExchangeRate(preFeeInterestRate, timeToMaturity)
436:         ).neg();
437:
438:         uint256 postFeeInterestRate = getPostFeeInterestRate(irParams,
 ↪  preFeeInterestRate, fCashToAccount < 0);
439:         postFeeCashToAccount = fCashToAccount.divInRatePrecision(
440:             getfCashExchangeRate(postFeeInterestRate, timeToMaturity)
441:         ).neg();
```

When executing a fCash trade, the interest rate is computed based on the utilization of the current market (Refer to Line 432). The `postFeeInterestRate` is then computed based on the `preFeeCashToAccount` and trading fee, and this rate will be used to derive the exchange rate needed to convert `fCashToAccount` to the net prime cash (`postFeeCashToAccount`).

Note that the interest rate used for the trade is `postFeeInterestRate`, and `postFeeCashToAccount` is the amount of cash credit or debit to an account.

If there is any slippage control in place, the slippage should be checked against the `postFeeInterestRate` or `postFeeCashToAccount`. As such, there are two approaches to implementing slippage controls:

- 1st Approach - The current interest rate is `2%`. User sets their acceptable interest rate limit at 3% when the user submits the trade transaction. The user's tolerance is `1%`. From the time the trade is initiated to when it's executed, the rate (`postFeeInterestRate`) rises to 5%, the transaction should revert due to the increased slippage beyond the user's tolerance.
- 2nd Approach - If a user sets the minimum trade return of 1000 cash, but the return is only 900 cash (`postFeeCashToAccount`) when the trade is executed, the transaction should revert as it exceeded the user's slippage tolerance

Note: When users submit a trade transaction, the transaction is held in the mempool for a period of time before executing, and thus the market condition and interest rate might change during this period, and slippage control is used to protect users from these fluctuations.

However, within the codebase, it was observed that the slippage was not checked against the `postFeeInterestRate` or `postFeeCashToAccount`.

https://github.com/sherlock-audit/2023-03-notional/blob/main/contracts-v2/contracts/internal/markets/InterestRateCurve.sol#L405

```
File: InterestRateCurve.sol
371:         // returns the net cash amounts to apply to each of the three
 ↪  relevant balances.
372:         (
373:             int256 netUnderlyingToAccount,
374:             int256 netUnderlyingToMarket,
375:             int256 netUnderlyingToReserve
376:         ) = _getNetCashAmountsUnderlying(
377:             irParams,
378:             market,
379:             cashGroup,
380:             totalCashUnderlying,
381:             fCashToAccount,
382:             timeToMaturity
383:         );
```

```
..SNIP..
388:         {
389:             // Do not allow utilization to go above 100 on trading,
     calculate the utilization after
390:             // the trade has taken effect, meaning that fCash changes and
     cash changes are applied to
391:             // the market totals.
392:             market.totalfCash = market.totalfCash.subNoNeg(fCashToAccount);
393:             totalCashUnderlying =
     totalCashUnderlying.add(netUnderlyingToMarket);
394:
395:             uint256 utilization = getfCashUtilization(0, market.totalfCash,
     totalCashUnderlying);
396:             if (utilization > uint256(Constants.RATE_PRECISION)) return (0,
     0);
397:
398:             uint256 newPreFeeImpliedRate = getInterestRate(irParams,
     utilization);
..SNIP..
404:             // Saves the preFeeInterestRate and fCash
405:             market.lastImpliedRate = newPreFeeImpliedRate;
406:         }
```

After computing the net prime cash (`postFeeCashToAccount == netUnderlyingToAccount`) at Line 373 above, it updates the `market.totalfCash` and `totalCashUnderlying`. Line 395 computes the `utilization` after the trade happens, and uses the latest `utilization` to compute the new interest rate after the trade and save it within the `market.lastImpliedRate`

https://github.com/sherlock-audit/2023-03-notional/blob/main/contracts-v2/contracts/external/actions/TradingAction.sol#L268

```
File: TradingAction.sol
234:     function _executeLendBorrowTrade(
..SNIP..
256:         cashAmount = market.executeTrade(
257:             account,
258:             cashGroup,
259:             fCashAmount,
260:             market.maturity.sub(blockTime),
261:             marketIndex
262:         );
263:
264:         uint256 rateLimit = uint256(uint32(bytes4(trade << 104)));
265:         if (rateLimit != 0) {
266:             if (tradeType == TradeActionType.Borrow) {
267:                 // Do not allow borrows over the rate limit
```

```
268:                 require(market.lastImpliedRate <= rateLimit, "Trade failed,
↪  slippage");
269:             } else {
270:                 // Do not allow lends under the rate limit
271:                 require(market.lastImpliedRate >= rateLimit, "Trade failed,
↪  slippage");
272:             }
273:         }
274:     }
```

The trade is executed at Line 256 above. After the trade is executed, it will check for the slippage at Line 264-273 above.

Let IR_1 be the interest rate used during the trade (`postFeeInterestRate`), IR_2 be the interest rate after the trade (`market.lastImpliedRate`), and IR_U be the user's acceptable interest rate limit (`rateLimit`).

Based on the current slippage control implementation, IR_U is checked against IR_2. Since the purpose of having slippage control in DeFi trade is to protect users from unexpected and unfavorable price changes **during** the execution of a trade, IR_1 should be used instead.

Assume that at the time of executing a trade (`TradeActionType.Borrow`), IR_1 spikes up and exceeds IR_U. However, since the slippage control checks IR_U against IR_2, which may have resettled to IR_U or lower, the transaction proceeds despite exceeding the user's acceptable rate limit. So, the transaction succeeds without a revert.

This issue will exacerbate when executing large trades relative to pool liquidity.

Impact

The existing slippage control does not provide the desired protection against unexpected interest rate fluctuations during the transaction. As a result, users might be borrowing at a higher cost or lending at a lower return than they intended, leading to losses.

Code Snippet

https://github.com/sherlock-audit/2023-03-notional/blob/main/contracts-v2/contracts/internal/markets/InterestRateCurve.sol#L421

https://github.com/sherlock-audit/2023-03-notional/blob/main/contracts-v2/contracts/internal/markets/InterestRateCurve.sol#L405

https://github.com/sherlock-audit/2023-03-notional/blob/main/contracts-v2/contracts/external/actions/TradingAction.sol#L268

Tool used

Manual Review

Recommendation

Consider updating the slippage control to compare the user's acceptable interest rate limit (`rateLimit`) against the interest rate used during the trade execution (`postFeeInterestRate`).

Discussion

jeffywu

Fixed: https://github.com/notional-finance/contracts-v2/pull/126

xiaoming9090

@0xleastwood + @xiaoming9090 : Verified. Fixed in PR https://github.com/notional-finance/contracts-v2/pull/126

Issue M-9: Inconsistent use of `VAULT_ACCOUNT_MIN_TIME` in vault implementation

Source: https://github.com/sherlock-audit/2023-03-notional-judging/issues/179

Found by

xiaoming90 + 0xleastwood

Summary

There is a considerable difference in implementation behaviour when a vault has yet to mature compared to after vault settlement.

Vulnerability Detail

There is some questionable functionality with the following `require` statement:

https://github.com/sherlock-audit/2023-03-notional-0xleastwood/blob/main/contracts-v2/contracts/external/actions/VaultAccountAction.sol#L242

```
File: VaultAccountAction.sol
242:     require(vaultAccount.lastUpdateBlockTime +
↪  Constants.VAULT_ACCOUNT_MIN_TIME <= block.timestamp)
```

The `lastUpdateBlockTime` variable is updated in *two* cases:

- A user enters a vault position, updating the vault state; including `lastUpdateBlockTime`. This is a proactive measure to prevent users from quickly entering and exiting the vault.
- The vault has matured and as a result, each time vault fees are assessed for a given vault account, `lastUpdateBlockTime` is updated to `block.timestamp` after calculating the pro-rated fee for the prime cash vault.

Therefore, before a vault has matured, it is not possible to quickly enter and exit a vault. But after `Constants.VAULT_ACCOUNT_MIN_TIME` has passed, the user can exit the vault as many times as they like. However, the same does not hold true once a vault has matured. Each time a user exits the vault, they must wait `Constants.VAULT_ACCOUNT_MIN_TIME` time again to re-exit. This seems like inconsistent behaviour.

Impact

The `exitVault()` function will ultimately affect prime and non-prime vault users differently. It makes sense for the codebase to be written in such a way that

functions execute in-line with user expectations.

Code Snippet

https://github.com/sherlock-audit/2023-03-notional-0xleastwood/blob/main/contracts-v2/contracts/external/actions/VaultAccountAction.sol#L242

https://github.com/sherlock-audit/2023-03-notional-0xleastwood/blob/main/contracts-v2/contracts/internal/vaults/VaultAccount.sol#L487-L506

https://github.com/sherlock-audit/2023-03-notional-0xleastwood/blob/main/contracts-v2/contracts/internal/vaults/VaultConfiguration.sol#L284

https://github.com/sherlock-audit/2023-03-notional-0xleastwood/blob/main/contracts-v2/contracts/internal/vaults/VaultConfiguration.sol#L257

Tool used

Manual Review

Recommendation

It might be worth adding an exception to `VaultConfiguration.settleAccountOrAccruePrimeCashFees()` so that when vault fees are calculated, `lastUpdatedBlockTime` is not updated to `block.timestamp`.

Discussion

jeffywu

Given how storage is structured, if we do not set lastUpdatedBlockTime then the prime vault account will end up paying fees on the same time portion twice. This is probably worse than not being able to exit the position twice in succession.

Although this issue is correct, the exit time is 1 minute. Will mark this as Won't Fix as the change will probably more complex than the benefit to UX.

Issue M-10: Return data from the external call not verified during deposit and redemption

Source: https://github.com/sherlock-audit/2023-03-notional-judging/issues/181

Found by

ShadowForce, xiaoming90 + 0xleastwood

Summary

The deposit and redemption functions did not verify the return data from the external call, which might cause the contract to wrongly assume that the deposit/redemption went well although the action has actually failed in the background.

Vulnerability Detail

https://github.com/sherlock-audit/2023-03-notional/blob/main/contracts-v2/contracts/internal/balances/protocols/GenericToken.sol#L63

```
File: GenericToken.sol
63:     function executeLowLevelCall(
64:         address target,
65:         uint256 msgValue,
66:         bytes memory callData
67:     ) internal {
68:         (bool status, bytes memory returnData) = target.call{value:
 ↪  msgValue}(callData);
69:         require(status, checkRevertMessage(returnData));
70:     }
```

When the external call within the `GenericToken.executeLowLevelCall` function reverts, the `status` returned from the `.call` will be `false`. In this case, Line 69 above will revert.

https://github.com/sherlock-audit/2023-03-notional/blob/main/contracts-v2/contracts/internal/balances/TokenHandler.sol#L375

```
File: TreasuryAction.sol
316:             for (uint256 j; j < depositData.targets.length; ++j) {
317:                 // This will revert if the individual call reverts.
318:                 GenericToken.executeLowLevelCall(
319:                     depositData.targets[j],
320:                     depositData.msgValue[j],
321:                     depositData.callData[j]
```

```
322:                 );
323:             }
```

For deposit and redeem, Notional assumes that all money markets will revert if the deposit/mint and redeem/burn has an error. Thus, it does not verify the return data from the external call. Refer to the comment in Line 317 above.

However, this is not always true due to the following reasons:

- Some money markets might not revert when errors occur but instead return `false (0)`. In this case, the current codebase will wrongly assume that the deposit/redemption went well although the action has failed.
- Compound might upgrade its contracts to return errors instead of reverting in the future.

Impact

The gist of prime cash is to integrate with multiple markets. Thus, the codebase should be written in a manner that can handle multiple markets. Otherwise, the contract will wrongly assume that the deposit/redemption went well although the action has actually failed in the background, which might potentially lead to some edge cases where assets are sent to the users even though the redemption fails.

Code Snippet

https://github.com/sherlock-audit/2023-03-notional/blob/main/contracts-v2/contracts/internal/balances/TokenHandler.sol#L375

https://github.com/sherlock-audit/2023-03-notional/blob/main/contracts-v2/contracts/external/actions/TreasuryAction.sol#L318

Tool used

Manual Review

Recommendation

Consider checking the `returnData` to ensure that the external money market returns a successful response after deposit and redemption.

Note that the successful response returned from various money markets might be different. Some protocols return `1` on a successful action, while Compound return zero (`NO_ERROR`).

Discussion

jeffywu

Since Compound V2 will not be used, this is a bit of a hypothetical. Also, if the external money market is trusted it should not "eat" the funds without reverting - it would be expected to return the funds which would cause the surrounding deposit amount checks to fail and the Notional transaction would revert.

I believe that this issue should be medium or low as suggested by the auditor.

Trumpero

I agree that the severity of this issue should be lower. Labeled it as a medium.

Issue M-11: Treasury rebalance will fail due to interest accrual

Source: https://github.com/sherlock-audit/2023-03-notional-judging/issues/189

Found by

xiaoming90 + 0xleastwood

Summary

If Compound has updated their interest rate model, then Notional will calculate the before total underlying token balance without accruing interest. If this exceeds `Constants.REBALANCING_UNDERLYING_DELTA`, then rebalance execution will revert.

Vulnerability Detail

The `TreasuryAction._executeRebalance()` function will revert on a specific edge case where `oracle.getTotalUnderlyingValueStateful()` does not accrue interest before calculating the value of the treasury's `cToken` holdings.

https://github.com/sherlock-audit/2023-03-notional-0xleastwood/blob/main/contracts-v2/contracts/external/actions/TreasuryAction.sol#L284-L302

```
File: TreasuryAction.sol
284:     function _executeRebalance(uint16 currencyId) private {
285:         IPrimeCashHoldingsOracle oracle =
 ↪  PrimeCashExchangeRate.getPrimeCashHoldingsOracle(currencyId);
286:         uint8[] memory rebalancingTargets =
 ↪  _getRebalancingTargets(currencyId, oracle.holdings());
287:         (RebalancingData memory data) =
 ↪  REBALANCING_STRATEGY.calculateRebalance(oracle, rebalancingTargets);
288:
289:         (/* */, uint256 totalUnderlyingValueBefore) =
 ↪  oracle.getTotalUnderlyingValueStateful();
290:
291:         // Process redemptions first
292:         Token memory underlyingToken =
 ↪  TokenHandler.getUnderlyingToken(currencyId);
293:         TokenHandler.executeMoneyMarketRedemptions(underlyingToken,
 ↪  data.redeemData);
294:
295:         // Process deposits
296:         _executeDeposits(underlyingToken, data.depositData);
297:
```

```
298:         (/* */, uint256 totalUnderlyingValueAfter) =
 ↪  oracle.getTotalUnderlyingValueStateful();
299:
300:         int256 underlyingDelta =
 ↪  totalUnderlyingValueBefore.toInt().sub(totalUnderlyingValueAfter.toInt());
301:         require(underlyingDelta.abs() <
 ↪  Constants.REBALANCING_UNDERLYING_DELTA);
302:     }
```

`cTokenAggregator.getExchangeRateView()` returns the exchange rate which is used to calculate the underlying value of `cToken` holdings in two ways:

- If the interest rate model is unchanged, then we correctly accrue interest by calculating it without mutating state.
- If the interest rate model HAS changed, then we query `cToken.exchangeRateStored()` which DOES NOT accrue interest.

```
File: cTokenAggregator.sol
092:     function getExchangeRateView() external view override returns (int256) {
093:         // Return stored exchange rate if interest rate model is updated.
094:         // This prevents the function from returning incorrect exchange
 ↪  rates
095:         uint256 exchangeRate = cToken.interestRateModel() ==
 ↪  INTEREST_RATE_MODEL
096:             ? _viewExchangeRate()
097:             : cToken.exchangeRateStored();
098:         _checkExchangeRate(exchangeRate);
099:
100:         return int256(exchangeRate);
101:     }
```

Therefore, if the interest rate model has changed, `totalUnderlyingValueBefore` will not include any accrued interest and `totalUnderlyingValueAfter` will include all accrued interest. As a result, it is likely that the delta between these two amounts will exceed `Constants.REBALANCING_UNDERLYING_DELTA`, causing the rebalance to ultimately revert.

It does not really make sense to not accrue interest if the interest rate model has changed unless we want to avoid any drastic changes to Notional's underlying protocol. Then we may want to explicitly revert here instead of allowing the rebalance function to still execute.

Impact

The treasury manager is unable to rebalance currencies across protocols and therefore it is likely that most funds become under-utilised as a result.

Code Snippet

https://github.com/sherlock-audit/2023-03-notional-0xleastwood/blob/main/contracts-v2/contracts/external/actions/TreasuryAction.sol#L284-L302

Tool used

Manual Review

Recommendation

Ensure this is well-understand and consider accruing interest under any circumstance. Alternatively, if we do not wish to accrue interest when the interest rate model has changed, then we need to make sure that `underlyingDelta` does not include this amount as `TreasuryAction._executeDeposits()` will ultimately update the vault's position in Compound.

Discussion

jeffywu

Valid issue, although I would disagree with the severity since interest rate models are unlikely to change and we have already deprecated Compound V2 support.

0xleastwood

Escalate for 10 USDC

While Compound V2 is intended to be deprecated, a substantial portion of Notional's codebase relied on extensively on this at contest time. Looking at severity guidelines, I think this one should be included as `medium` severity because the scenario is considered to be viable, although unlikely.

> Medium: There is a viable scenario (even if unlikely) that could cause the protocol to enter a state where a material amount of funds can be lost. The attack path is possible with assumptions that either mimic on-chain conditions or reflect conditions that have a reasonable chance of becoming true in the future. The more expensive the attack is for an attacker, the less likely it will be included as a Medium (holding all other factors constant). The vulnerability must be something that is not considered an acceptable risk by a reasonable protocol team.

sherlock-admin

> Escalate for 10 USDC
>
> While Compound V2 is intended to be deprecated, a substantial portion of Notional's codebase relied on extensively on this at contest time. Looking at severity guidelines, I think this one should be included as

`medium` severity because the scenario is considered to be viable, although unlikely.

> Medium: There is a viable scenario (even if unlikely) that could cause the protocol to enter a state where a material amount of funds can be lost. The attack path is possible with assumptions that either mimic on-chain conditions or reflect conditions that have a reasonable chance of becoming true in the future. The more expensive the attack is for an attacker, the less likely it will be included as a Medium (holding all other factors constant). The vulnerability must be something that is not considered an acceptable risk by a reasonable protocol team.

You've created a valid escalation for 10 USDC!

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.

hrishibhat

Result: Medium Unique Agree with the points raised in the escalation. Considering this a valid medium

sherlock-admin

Escalations have been resolved successfully!

Escalation status:

- 0xleastwood: accepted

Issue M-12: Debt cannot be repaid without redeeming vault share

Source: https://github.com/sherlock-audit/2023-03-notional-judging/issues/191

Found by

xiaoming90 + 0xleastwood

Summary

Debt cannot be repaid without redeeming the vault share. As such, users have to redeem a certain amount of vault shares/strategy tokens at the current market price to work around this issue, which deprives users of potential gains from their vault shares if they maintain ownership until the end.

Vulnerability Detail

https://github.com/sherlock-audit/2023-03-notional/blob/main/contracts-v2/contracts/external/actions/VaultAccountAction.sol#L277

```
File: VaultAccountAction.sol
224:     function exitVault(
225:         address account,
226:         address vault,
227:         address receiver,
228:         uint256 vaultSharesToRedeem,
229:         uint256 lendAmount,
230:         uint32 minLendRate,
231:         bytes calldata exitVaultData
232:     ) external payable override nonReentrant returns (uint256
 ↪  underlyingToReceiver) {
..SNIP..
261:         // If insufficient strategy tokens are redeemed (or if it is set to
 ↪  zero), then
262:         // redeem with debt repayment will recover the repayment from the
 ↪  account's wallet
263:         // directly.
264:         underlyingToReceiver =
 ↪  underlyingToReceiver.add(vaultConfig.redeemWithDebtRepayment(
265:             vaultAccount, receiver, vaultSharesToRedeem, exitVaultData
266:         ));
```

There is a valid scenario where users want to repay debt without redeeming their vault shares/strategy tokens (mentioned in the comments above "or if it is set to

zero" at Line 251-263). In this case, the users will call `exitVault` with `vaultSharesToRedeem` parameter set to zero. The entire debt to be repaid will then be recovered directly from the account's wallet.

Following is the function trace of the `VaultAccountAction.exitVault`:

```
VaultAccountAction.exitVault
VaultConfiguration.redeemWithDebtRepayment
  VaultConfiguration._redeem
    IStrategyVault.redeemFromNotional
      MetaStable2TokenAuraVault._redeemFromNotional
        MetaStable2TokenAuraHelper.redeem
          Balancer2TokenPoolUtils._redeem
            StrategyUtils._redeemStrategyTokens
```

https://github.com/notional-finance/leveraged-vaults/blob/c707f7781e36d7a1259214dde2221f892a81a9c1/contracts/vaults/common/internal/strategy/StrategyUtils.sol#L153

```
File: StrategyUtils.sol
147:     function _redeemStrategyTokens(
148:         StrategyContext memory strategyContext,
149:         uint256 strategyTokens
150:     ) internal returns (uint256 poolClaim) {
151:         poolClaim = _convertStrategyTokensToPoolClaim(strategyContext,
↪  strategyTokens);
152:
153:         if (poolClaim == 0) {
154:             revert Errors.ZeroPoolClaim();
155:         }
```

The problem is that if the vault shares/strategy tokens to be redeemed are zero, the `poolClaim` will be zero and cause a revert within the `StrategyUtils._redeemStrategyTokens` function call. Thus, users who want to repay debt without redeeming their vault shares/strategy tokens will be unable to do so.

Impact

Users cannot repay debt without redeeming their vault shares/strategy tokens. To do so, they have to redeem a certain amount of vault shares/strategy tokens at the current market price to work around this issue so that `poolClaim > 0`, which deprives users of potential gains from their vault shares if they maintain ownership until the end.

Code Snippet

https://github.com/sherlock-audit/2023-03-notional/blob/main/contracts-v2/contracts/external/actions/VaultAccountAction.sol#L277

Tool used

Manual Review

Recommendation

Within the `VaultConfiguration.redeemWithDebtRepayment` function, skip the vault share redemption if `vaultShares` is zero. In this case, the `amountTransferred` will be zero, and the subsequent code will attempt to recover the entire `underlyingExternalToRepay` amount directly from account's wallet.

```
function redeemWithDebtRepayment(
    VaultConfig memory vaultConfig,
    VaultAccount memory vaultAccount,
    address receiver,
    uint256 vaultShares,
    bytes calldata data
) internal returns (uint256 underlyingToReceiver) {
    uint256 amountTransferred;
    uint256 underlyingExternalToRepay;
    {
..SNIP..
+       if (vaultShares > 0) {
            // Repayment checks operate entirely on the
 ↪  underlyingExternalToRepay, the amount of
            // prime cash raised is irrelevant here since tempCashBalance is
 ↪  cleared to zero as
            // long as sufficient underlying has been returned to the protocol.
            (amountTransferred, underlyingToReceiver, /* primeCashRaised */) =
 ↪  _redeem(
                vaultConfig,
                underlyingToken,
                vaultAccount.account,
                receiver,
                vaultShares,
                vaultAccount.maturity,
                underlyingExternalToRepay,
                data
            );
+       }
..Recover any unpaid debt amount from the account directly..
..SNIP..
```

Alternatively, update the `StrategyUtils._redeemStrategyTokens` function to handle zero vault share appropriately. However, note that the revert at Line 154 is added as part of mitigation to the "minting zero-share" bug in the past audit. Therefore, any changes to this part of the code must ensure that the "minting zero-share" bug is not being re-introduced. Removing the code at 153-155 might result in the user's vault share being "burned" but no assets in return under certain conditions.

```
File: StrategyUtils.sol
147:     function _redeemStrategyTokens(
148:         StrategyContext memory strategyContext,
149:         uint256 strategyTokens
150:     ) internal returns (uint256 poolClaim) {
151:         poolClaim = _convertStrategyTokensToPoolClaim(strategyContext,
↪  strategyTokens);
152:
153:         if (poolClaim == 0) {
154:             revert Errors.ZeroPoolClaim();
155:         }
```

Discussion

jeffywu

Agree with the issue, however, the fix will be to remove the error from _redeemStrategyTokens. We should always call the vault in case there is something it needs to do during the vault action.

jeffywu

Fixed in: https://github.com/notional-finance/leveraged-vaults/pull/54

xiaoming9090

@0xleastwood + @xiaoming9090 : Verified. Fixed in the codebase of Notional's Leverage Vault in PR https://github.com/notional-finance/leveraged-vaults/pull/54

Issue M-13: Vault account might not be able to exit after liquidation

Source: https://github.com/sherlock-audit/2023-03-notional-judging/issues/192

Found by

xiaoming90 + 0xleastwood

Summary

The vault exit might fail after a liquidation event, leading to users being unable to main their positions.

Vulnerability Detail

Assume that a large portion of the vault account gets liquidated which results in a large amount of cash deposited into the vault account's cash balance. In addition, interest will also start accruing within the vault account's cash balance.

Let x be the `primaryCash` of a vault account after a liquidation event and interest accrual.

The owner of the vault account decided to exit the vault by calling `exitVault`. Within the `exitVault` function, the `vaultAccount.tempCashBalance` will be set to x.

Next, the `lendToExitVault` function is called. Assume that the cost in prime cash terms to lend an offsetting fCash position is $-y$ (`primeCashCostToLend`). The `updateAccountDebt` function will be called, and the `vaultAccount.tempCashBalance` will be updated to $x + (-y) \Rightarrow x - y$. If $x > y$, then the new `vaultAccount.tempCashBalance` will be more than zero.

Subsequently, the `redeemWithDebtRepayment` function will be called. However, since `vaultAccount.tempCashBalance` is larger than zero, the transaction will revert, and the owner cannot exit the vault.

https://github.com/sherlock-audit/2023-03-notional/blob/main/contracts-v2/contracts/internal/vaults/VaultConfiguration.sol#L429

```
File: VaultConfiguration.sol
424:             if (vaultAccount.tempCashBalance < 0) {
425:                 int256 x = vaultConfig.primeRate.convertToUnderlying(vaultA
↪  ccount.tempCashBalance).neg();
426:                 underlyingExternalToRepay =
↪  underlyingToken.convertToUnderlyingExternalWithAdjustment(x).toUint();
427:             } else {
428:                 // Otherwise require that cash balance is zero. Cannot have
↪  a positive cash balance in this method
```

```
429:                 require(vaultAccount.tempCashBalance == 0);
430:             }
```

Impact

The owner of the vault account would not be able to exit the vault to main their position. As such, their assets are stuck within the protocol.

Code Snippet

https://github.com/sherlock-audit/2023-03-notional/blob/main/contracts-v2/contracts/internal/vaults/VaultConfiguration.sol#L429

Tool used

Manual Review

Recommendation

Consider refunding the excess positive `vaultAccount.tempCashBalance` to the users so that `vaultAccount.tempCashBalance` will be cleared (set to zero) before calling the `redeemWithDebtRepayment` function.

Discussion

jeffywu

Fixed in: https://github.com/notional-finance/contracts-v2/pull/133

xiaoming9090

@0xleastwood + @xiaoming9090: Verified. Fixed in PR https://github.com/notional-finance/contracts-v2/pull/133

Issue M-14: Rebalance process reverts due to zero amount deposit and redemption

Source: https://github.com/sherlock-audit/2023-03-notional-judging/issues/195

Found by

xiaoming90 + 0xleastwood

Summary

Depositing or redeeming zero amount against certain external money markets will cause the rebalancing process to revert.

Vulnerability Detail

For a specific holding (e.g. cToken), the `redeemAmounts` and `depositAmounts` are mutually exclusive. So if the `redeemAmounts` for a specific holding is non-zero, the `depositAmounts` will be zero and vice-versa. This is because of the if-else block at Lines 48-56 below. Only `redeemAmounts` or `depositAmounts` of a specific holding can be initialized, but not both.

https://github.com/sherlock-audit/2023-03-notional/blob/main/contracts-v2/contracts/external/pCash/ProportionalRebalancingStrategy.sol#L48

```
File: ProportionalRebalancingStrategy.sol
40:         for (uint256 i; i < holdings.length;) {
41:             address holding = holdings[i];
42:             uint256 targetAmount = totalValue * rebalancingTargets[i] /
↪  uint256(Constants.PERCENTAGE_DECIMALS);
43:             uint256 currentAmount = values[i];
44:
45:             redeemHoldings[i] = holding;
46:             depositHoldings[i] = holding;
47:
48:             if (targetAmount < currentAmount) {
49:                 unchecked {
50:                     redeemAmounts[i] = currentAmount - targetAmount;
51:                 }
52:             } else if (currentAmount < targetAmount) {
53:                 unchecked {
54:                     depositAmounts[i] = targetAmount - currentAmount;
55:                 }
56:             }
57:
58:             unchecked {
```

```
59:                 ++i;
60:             }
61:         }
62:
```

For each holding, the following codes always deposit or redeem a zero value. For example, cETH holding, if the `redeemAmounts` is 100 ETH, the `depositAmounts` will be zero. (because of the if-else block). Therefore, `getDepositCalldataForRebalancing` function will be executed and attempt to deposit zero amount to Compound.

https://github.com/sherlock-audit/2023-03-notional/blob/main/contracts-v2/contracts/external/pCash/ProportionalRebalancingStrategy.sol#L63

```
File: ProportionalRebalancingStrategy.sol
63:         rebalancingData.redeemData =
↪   oracle.getRedemptionCalldataForRebalancing(redeemHoldings, redeemAmounts);
64:         rebalancingData.depositData =
↪   oracle.getDepositCalldataForRebalancing(depositHoldings, depositAmounts);
```

The problem is that the deposit/mint or redeem/burn function of certain external money markets will revert if the amount is zero. Notional is considering integrating with a few external monkey markets and one of them is AAVE.

In this case, when Notional deposit zero amount to AAVE or redeem zero amount from AAVE, it causes the rebalancing process to revert because of the onlyAmountGreaterThanZero modifier on the AAVE's `deposit` and `redeem` function.

https://github.com/aave/aave-protocol/blob/4b4545fb583fd4f400507b10f3c3114f45b8a037/contracts/lendingpool/LendingPool.sol#L305

```
function deposit(address _reserve, uint256 _amount, uint16 _referralCode)
    external
    payable
    nonReentrant
    onlyActiveReserve(_reserve)
    onlyUnfreezedReserve(_reserve)
    onlyAmountGreaterThanZero(_amount)
{
```

https://github.com/aave/aave-protocol/blob/4b4545fb583fd4f400507b10f3c3114f45b8a037/contracts/lendingpool/LendingPool.sol#LL331C1-L342C6

```
function redeemUnderlying(
    address _reserve,
    address payable _user,
    uint256 _amount,
    uint256 _aTokenBalanceAfterRedeem
```

```
)
    external
    nonReentrant
    onlyOverlyingAToken(_reserve)
    onlyActiveReserve(_reserve)
    onlyAmountGreaterThanZero(_amount)
{
```

The above issue is not only limited to AAVE and might also happen in other external markets.

Even if the external money market does not revert when minting or burning zero amount, there is a small possibility that the supported underlying token might revert on zero value transfer (https://github.com/d-xo/weird-erc20#revert-on-zero-value-transfers). Because mint will do a `transferOut` and burn will do a `transferIn` against zero value

Impact

Notional would not be able to rebalance its underlying holding. The key feature of Notional V3 is to allow its Treasury Manager to rebalance underlying holdings into various other money market protocols.

This makes Notional more resilient to issues in external protocols and future-proofs the protocol. If rebalancing does not work, Notional will be unable to move its fund out of a vulnerable external market, potentially draining protocol funds if this is not mitigated.

Another purpose of rebalancing is to allow Notional to allocate Notional V3's capital to new opportunities or protocols that provide a good return. If rebalancing does not work, the protocol and its users will lose out on the gain from the investment.

On the other hand, if an external monkey market that Notional invested in is consistently underperforming or yielding negative returns, Notional will perform a rebalance to reallocate its funds to a better market. However, if rebalancing does not work, they will be stuck with a suboptimal asset allocation, and the protocol and its users will incur losses.

Code Snippet

https://github.com/sherlock-audit/2023-03-notional/blob/main/contracts-v2/contracts/external/pCash/ProportionalRebalancingStrategy.sol#L48

https://github.com/sherlock-audit/2023-03-notional/blob/main/contracts-v2/contracts/external/pCash/ProportionalRebalancingStrategy.sol#L63

Tool used

Manual Review

Recommendation

Consider implementing validation to ensure the contract does not deposit zero amount to or redeem zero amount from the external market.

Following is the pseudocode for the potential fixes that could be implemented within the `_getDepositCalldataForRebalancing` of the holding contract to mitigate this issue. The same should be done for redemption.

```
function _getDepositCalldataForRebalancing(
    address[] calldata holdings,
    uint256[] calldata depositAmounts
) internal view virtual override returns (
    DepositData[] memory depositData
) {
    require(holdings.length == NUM_ASSET_TOKENS);
    for (int i = 0; i < holdings.length; i++) {
        if (depositAmounts[i] > 0) {
            // populate the depositData[i] with the deposit calldata to external
 ↪  money market>
        }
    }
}
```

The above solution will return an empty calldata if the deposit amount is zero for a specific holding.

Within the `_executeDeposits` function, skip the `depositData` if it has not been initialized.

```
function _executeDeposits(Token memory underlyingToken, DepositData[] memory
 ↪  deposits) private {
    uint256 totalUnderlyingDepositAmount;

    for (uint256 i; i < deposits.length; i++) {
        DepositData memory depositData = deposits[i];
        // if depositData is not initialized, skip to the next one
```

Discussion

jeffywu

Fixed in: https://github.com/notional-finance/contracts-v2/pull/120

xiaoming9090

@0xleastwood + @xiaoming9090 : Verified. Fixed in PR https://github.com/notional-finance/contracts-v2/pull/120

Issue M-15: Inaccurate settlement reserve accounting

Source: https://github.com/sherlock-audit/2023-03-notional-judging/issues/196

Found by

xiaoming90 + 0xleastwood

Summary

The off-chain accounting of fCash debt or prime cash in the settlement reserve will be inaccurate due to an error when handling the conversion between signed and unsigned integers.

Vulnerability Detail

Events will be emitted to reconcile off-chain accounting for the edge condition when leveraged vaults lend at zero interest. This event will be emitted if there is fCash debt or prime cash in the settlement reserve.

In an event where `s.fCashDebtHeldInSettlementReserve > 0` and `s.primeCashHeldInSettlementReserve <= 0`, no event will be emitted. As a result, the off-chain accounting of fCash debt or prime cash in the settlement reserve will be off.

The reason is that since `fCashDebtInReserve` is the negation of `s.fCashDebtHeldInSettlementReserve`, which is an unsigned integer, `fCashDebtInReserve` will always be less than or equal to 0. Therefore, `fCashDebtInReserve > 0` will always be false and is an unsatisfiable condition.

https://github.com/sherlock-audit/2023-03-notional/blob/main/contracts-v2/contracts/internal/pCash/PrimeRateLib.sol#L459

```
File: PrimeRateLib.sol
457:         // This is purely done to fully reconcile off chain accounting with
 ↪  the edge condition where
458:         // leveraged vaults lend at zero interest.
459:         int256 fCashDebtInReserve =
 ↪  -int256(s.fCashDebtHeldInSettlementReserve);
460:         int256 primeCashInReserve =
 ↪  int256(s.primeCashHeldInSettlementReserve);
461:         if (fCashDebtInReserve > 0 || primeCashInReserve > 0) {
462:             int256 settledPrimeCash = convertFromUnderlying(settlementRate,
 ↪  fCashDebtInReserve);
463:             int256 excessCash;
464:             if (primeCashInReserve > settledPrimeCash) {
465:                 excessCash = primeCashInReserve - settledPrimeCash;
```

```
466:                 BalanceHandler.incrementFeeToReserve(currencyId,
 ↪  excessCash);
467:             }
468:
469:             Emitter.emitSettlefCashDebtInReserve(
470:                 currencyId, maturity, fCashDebtInReserve, settledPrimeCash,
 ↪  excessCash
471:             );
472:         }
```

Impact

The off-chain accounting of fCash debt or prime cash in the settlement reserve will be inaccurate. Users who rely on inaccurate accounting information to conduct any form of financial transaction will expose themselves to unintended financial risks and make ill-informed decisions.

Code Snippet

https://github.com/sherlock-audit/2023-03-notional/blob/main/contracts-v2/contracts/internal/pCash/PrimeRateLib.sol#L459

Tool used

Manual Review

Recommendation

It is recommended to implement the following fix:

```
- int256 fCashDebtInReserve = -int256(s.fCashDebtHeldInSettlementReserve);
+ int256 fCashDebtInReserve = int256(s.fCashDebtHeldInSettlementReserve);
```

Discussion

jeffywu

Fixed: https://github.com/notional-finance/contracts-v2/pull/139

xiaoming9090

@0xleastwood + @xiaoming9090 : Verified. Fixed in PR https://github.com/notional-finance/contracts-v2/pull/139

Issue M-16: Rebalance stops working when more holdings are added

Source: https://github.com/sherlock-audit/2023-03-notional-judging/issues/198

Found by

xiaoming90 + 0xleastwood

Summary

Notional would not be able to rebalance its underlying holding when more holdings are added.

Vulnerability Detail

https://github.com/sherlock-audit/2023-03-notional/blob/main/contracts-v2/contracts/external/actions/TreasuryAction.sol#L301

```
File: TreasuryAction.sol
284:     function _executeRebalance(uint16 currencyId) private {
285:         IPrimeCashHoldingsOracle oracle =
↪  PrimeCashExchangeRate.getPrimeCashHoldingsOracle(currencyId);
286:         uint8[] memory rebalancingTargets =
↪  _getRebalancingTargets(currencyId, oracle.holdings());
287:         (RebalancingData memory data) =
↪  REBALANCING_STRATEGY.calculateRebalance(oracle, rebalancingTargets);
288:
289:         (/* */, uint256 totalUnderlyingValueBefore) =
↪  oracle.getTotalUnderlyingValueStateful();
290:
291:         // Process redemptions first
292:         Token memory underlyingToken =
↪  TokenHandler.getUnderlyingToken(currencyId);
293:         TokenHandler.executeMoneyMarketRedemptions(underlyingToken,
↪  data.redeemData);
294:
295:         // Process deposits
296:         _executeDeposits(underlyingToken, data.depositData);
297:
298:         (/* */, uint256 totalUnderlyingValueAfter) =
↪  oracle.getTotalUnderlyingValueStateful();
299:
300:         int256 underlyingDelta =
↪  totalUnderlyingValueBefore.toInt().sub(totalUnderlyingValueAfter.toInt());
```

```
301:         require(underlyingDelta.abs() <
↪  Constants.REBALANCING_UNDERLYING_DELTA);
302:     }
```

If the underlying delta is equal to or larger than the acceptable delta, the rebalancing process will fail and revert as per Line 301 above.

`Constants.REBALANCING_UNDERLYING_DELTA` is currently hardcoded to 0.0001. There is only 1 holding (cToken) in the current code base, so 0.0001 might be the optimal acceptable delta.

Let c be the underlying delta for cToken holding. Then, $0 <= c < 0.0001$.

However, as more external markets are added to Notional, the number of holdings will increase, and the rounding errors could accumulate. Let a and m be the underlying delta for aToken and morpho token respectively. Then $0 <= (c + a + m) < 0.0001$.

The accumulated rounding error or underlying delta $(c + a + m)$ could be equal to or larger than 0.0001 and cause the `_executeRebalance` function always to revert. As a result, Notional would not be able to rebalance its underlying holding.

Impact

Notional would not be able to rebalance its underlying holding. The key feature of Notional V3 is to allow its Treasury Manager to rebalance underlying holdings into various other money market protocols.

This makes Notional more resilient to issues in external protocols and future-proofs the protocol. If rebalancing does not work, Notional will be unable to move its fund out of a vulnerable external market, potentially draining protocol funds if this is not mitigated.

Another purpose of rebalancing is to allow Notional to allocate Notional V3's capital to new opportunities or protocols that provide a good return. If rebalancing does not work, the protocol and its users will lose out on the gain from the investment.

On the other hand, if an external monkey market that Notional invested in is consistently underperforming or yielding negative returns, Notional will perform a rebalance to reallocate its funds to a better market. However, if rebalancing does not work, they will be stuck with a suboptimal asset allocation, and the protocol and its users will incur losses.

Code Snippet

https://github.com/sherlock-audit/2023-03-notional/blob/main/contracts-v2/contracts/external/actions/TreasuryAction.sol#L301

Tool used

Manual Review

Recommendation

If the acceptable underlying delta for one holding (cToken) is ≈ 0.0001, the acceptable underlying delta for three holdings should be ≈ 0.0003 to factor in the accumulated rounding error or underlying delta.

Instead of hardcoding the `REBALANCING_UNDERLYING_DELTA`, consider allowing the governance to adjust this acceptable underlying delta to accommodate more holdings in the future and to adapt to potential changes in market conditions.

Discussion

jeffywu

Fixed in this PR: https://github.com/notional-finance/contracts-v2/pull/118

xiaoming9090

@0xleastwood + @xiaoming9090: Verified. Fixed in PR https://github.com/notional-finance/contracts-v2/pull/118. The issue of unnecessary scaling in PR 118 has been resolved in this commit (https://github.com/notional-finance/contracts-v2/commit/a200877c2b1d7c9f4a13b1fd84c8ecd3dae005a9).

Issue M-17: Liquidation frontrunning can prevent debt repayment upon unpausing (restoring full router)

Source: https://github.com/sherlock-audit/2023-03-notional-judging/issues/203

Found by

0x00ffDa

Summary

During a period of time that the PauseRouter is in use, the valuations of a user's debt and collateral may make them subject to liquidation. But, in the first block after the normal Router is restored, MEV bots can preempt any transactions that could prevent the liquidation (e.g. repayment or adding collateral).

Vulnerability Detail

Pausing and unpausing the router is performed via a call to the active router's inherited UUPSUpgradeable.upgradeTo() (or `upgradeToAndCall()`) function and supplying the address of the next router contract, which calls `GovernanceAction._authorizeUpgrade()` and then `ERC1967Upgrade._upgradeToAndCallSecure()` which ends with ERC1967Upgrade._upgradeTo() switching the implementation address to realize the desired change. Pausing may be performed by either the owner or the pause guardian role. Only the owner can unpause the system.

Notional has forked governance logic from Compound. In Compound, "*Importantly, the pauseGuardian does not have the ability to prevent users from exiting their positions by calling redeem or repayBorrow*". However, in Notional this is not true. The PauseRouter does not delegate account action functions that would allow debt repayment. As such, without recourse to avoid it, user debt positions may become liquidatable during a pause.

MEV bots are able to use view functions to monitor account health during a pause. So, while they may actively detect and frontrun debt repayment transactions upon system unpausing, it is not required. The normal operation of liquidators bots can have the same effect.

Note that liquidations may be enabled during a pause as well. (That is determined by system configuration at the discretion of the owner or pause guardian and enabling it would pose additional liquidation risk to users.) The frontrunning vulnerability is present even if liquidations are not enabled during a pause.

Ref: audit finding

Impact

By frontrunning any debt repayment (or collateral deposit) attempts after unpausing the router, MEV bots can unfairly liquidate all debt positions that became eligible for liquidation during the pause. This causes loss of funds to all affected users.

Code Snippet

Tool used

Manual Review

Recommendation

If liquidation is not allowed during a pause, add a grace period after unpausing during which liquidation remains blocked to allow users to avoid unfair liquidation by repaying debt or supplying additional collateral.

Discussion

0x00ffDa

Escalate for 10 USDC

I believe this finding should be reclassified as "High" severity because it meets all the current Sherlock criteria for "High" issues:

> High: This vulnerability would result in a material loss of funds, and the cost of the attack is low (relative to the amount of funds lost). The attack path is possible with reasonable assumptions that mimic on-chain conditions. The vulnerability must be something that is not considered an acceptable risk by a reasonable protocol team.

Specifically ... unfair liquidation is a material loss of user funds and can occur with low cost by any liquidator. Use of the `PauseRouter` is expected in any upgrade and it may also be used in other circumstances, so this is not "stars must align" scenario. The sponsor has already confirmed this is a valid finding.

sherlock-admin

> Escalate for 10 USDC
>
> I believe this finding should be reclassified as "High" severity because it meets all the current Sherlock criteria for "High" issues:
>
> > High: This vulnerability would result in a material loss of funds, and the cost of the attack is low (relative to the amount of funds lost). The attack path is possible with reasonable assumptions that mimic on-chain conditions. The vulnerability

> > must be something that is not considered an acceptable risk by a reasonable protocol team.
>
> Specifically ... unfair liquidation is a material loss of user funds and can occur with low cost by any liquidator. Use of the `PauseRouter` is expected in any upgrade and it may also be used in other circumstances, so this is not "stars must align" scenario. The sponsor has already confirmed this is a valid finding.

You've created a valid escalation for 10 USDC!

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.

0xleastwood

> Escalate for 10 USDC
>
> I believe this finding should be reclassified as "High" severity because it meets all the current Sherlock criteria for "High" issues:
>
> > High: This vulnerability would result in a material loss of funds, and the cost of the attack is low (relative to the amount of funds lost). The attack path is possible with reasonable assumptions that mimic on-chain conditions. The vulnerability must be something that is not considered an acceptable risk by a reasonable protocol team.
>
> Specifically ... unfair liquidation is a material loss of user funds and can occur with low cost by any liquidator. Use of the `PauseRouter` is expected in any upgrade and it may also be used in other circumstances, so this is not "stars must align" scenario. The sponsor has already confirmed this is a valid finding.

The pause router is not expected in any upgrade to the Notional protocol. It is only used in an emergency situation, so by definition, this is already an unlikely scenario to occur. Additionally, it would be safer to allow for liquidations after the protocol has been unpaused because this is how the protocol is designed to stay solvent. Adding an arbitrary grace period only adds to the risk that a vault position may become undercollateralised. If anything, a better approach would be to allow for vault accounts to be liquidated from within the pause router because this functionality isn't currently enabled.

0xleastwood

I do think Notional prioritises protocol solvency in this emergency situations over user experience and I think this is the correct approach.

0x00ffDa

> The pause router is not expected in any upgrade to the Notional protocol.

From my reading of the V3 upgrade script, it does plan to use the PauseRouter.

0xleastwood

> > The pause router is not expected in any upgrade to the Notional protocol.
>
> From my reading of the V3 upgrade script, it does plan to use the PauseRouter.

I think this is unique to V3 migration because it takes a bit of time and requires multiple steps to perform. But generally speaking, most patch fixes can be done atomically.

jeffywu

> The vulnerability must be something that is not considered an acceptable risk by a reasonable protocol team.

We will take this risk during the upgrade. So if that is the definition of a high priority issue then this is not a high priority issue.

hrishibhat

Result: Medium Unique Considering this issue a valid medium given the circumstances under which this would occur and as mentioned by the Sponsor this can be considered an accepted risk.

sherlock-admin

Escalations have been resolved successfully!

Escalation status:

- 0x00ffDa: rejected

Issue M-18: Underlying delta is calculated on internal token balance

Source: https://github.com/sherlock-audit/2023-03-notional-judging/issues/205

Found by

xiaoming90 + 0xleastwood

Summary

The underlying delta is calculated on the internal token balance, which might cause inconsistency with tokens of varying decimals.

Vulnerability Detail

https://github.com/sherlock-audit/2023-03-notional-0xleastwood/blob/main/contracts-v2/contracts/external/actions/TreasuryAction.sol#L301

```
File: TreasuryAction.sol
284:     function _executeRebalance(uint16 currencyId) private {
285:         IPrimeCashHoldingsOracle oracle =
↪  PrimeCashExchangeRate.getPrimeCashHoldingsOracle(currencyId);
286:         uint8[] memory rebalancingTargets =
↪  _getRebalancingTargets(currencyId, oracle.holdings());
287:         (RebalancingData memory data) =
↪  REBALANCING_STRATEGY.calculateRebalance(oracle, rebalancingTargets);
288:
289:         (/* */, uint256 totalUnderlyingValueBefore) =
↪  oracle.getTotalUnderlyingValueStateful();
290:
291:         // Process redemptions first
292:         Token memory underlyingToken =
↪  TokenHandler.getUnderlyingToken(currencyId);
293:         TokenHandler.executeMoneyMarketRedemptions(underlyingToken,
↪  data.redeemData);
294:
295:         // Process deposits
296:         _executeDeposits(underlyingToken, data.depositData);
297:
298:         (/* */, uint256 totalUnderlyingValueAfter) =
↪  oracle.getTotalUnderlyingValueStateful();
299:
300:         int256 underlyingDelta =
↪  totalUnderlyingValueBefore.toInt().sub(totalUnderlyingValueAfter.toInt());
```

```
301:         require(underlyingDelta.abs() <
↪  Constants.REBALANCING_UNDERLYING_DELTA);
302:     }
```

The `underlyingDelta` is denominated in internal token precision (1e8) and is computed by taking the difference between `totalUnderlyingValueBefore` and `totalUnderlyingValueAfter` in Line 300 above.

Next, the `underlyingDelta` is compared against the `Constants.REBALANCING_UNDERLYING_DELTA` (10_000=0.0001) to ensure that the rebalance did not exceed the acceptable delta threshold.

However, the same `Constants.REBALANCING_UNDERLYING_DELTA` is used across all tokens such as ETH, DAI, and USDC. As a result, the delta will not be consistent with tokens of varying decimals.

Impact

Using the internal token precision (1e8) might result in an over-sensitive trigger for tokens with fewer decimals (e.g. 1e6) as they are scaled up and an under-sensitive one for tokens with more decimals (e.g. 1e18) as they are scaled down, leading to inconsistency across different tokens when checking against the `Constants.REBALANCING_UNDERLYING_DELTA`.

This also means that the over-sensitive one will trigger a revert more easily and vice versa.

Code Snippet

https://github.com/sherlock-audit/2023-03-notional-0xleastwood/blob/main/contracts-v2/contracts/external/actions/TreasuryAction.sol#L301

Tool used

Manual Review

Recommendation

Consider using the external token balance and scale `Constants.REBALANCING_UNDERLYING_DELTA` to the token's decimals.

Discussion

jeffywu

Valid suggestion

jeffywu

Fixed: https://github.com/notional-finance/contracts-v2/pull/118

0xleastwood

@0xleastwood + @xiaoming9090: Verified. Fixed in PR https://github.com/notional-finance/contracts-v2/pull/118. The issue of unnecessary scaling in PR 118 has been resolved in this commit (https://github.com/notional-finance/contracts-v2/commit/a200877c2b1d7c9f4a13b1fd84c8ecd3dae005a9).

Additional Suggestion: Small improvement could be made where the delta is only checked if the change moves in the negative direction. This would protect the protocol a bit more from DoS-style of attacks via donation.

jeffywu

@0xleastwood, the `getTotalUnderlyingValueStateful` method reads from an internally tracked contract balance so it should not be affected by any donations to the protocol. However, agree that only checking to the downside would be slightly more efficient here.

0xleastwood

Good point, I forgot that was the case.

0xleastwood

@0xleastwood + @xiaoming9090: Verified. Fixed in https://github.com/notional-finance/contracts-v2/pull/118

Issue M-19: Secondary debt dust balances are not truncated

Source: https://github.com/sherlock-audit/2023-03-notional-judging/issues/210

Found by

xiaoming90 + 0xleastwood

Summary

Dust balances in primary debt are truncated toward zero. However, this truncation was not performed against secondary debts.

Vulnerability Detail

https://github.com/sherlock-audit/2023-03-notional-0xleastwood/blob/main/contracts-v2/contracts/internal/vaults/VaultAccount.sol#L231

```
File: VaultAccount.sol
212:     function updateAccountDebt(
..SNIP..
230:         // Truncate dust balances towards zero
231:         if (0 < vaultState.totalDebtUnderlying &&
↪   vaultState.totalDebtUnderlying < 10) vaultState.totalDebtUnderlying = 0;
..SNIP..
233:     }
```

`vaultState.totalDebtUnderlying` is primarily used to track the total debt of primary currency. Within the `updateAccountDebt` function, any dust balance in the `vaultState.totalDebtUnderlying` is truncated towards zero at the end of the function as shown above.

https://github.com/sherlock-audit/2023-03-notional-0xleastwood/blob/main/contracts-v2/contracts/internal/vaults/VaultSecondaryBorrow.sol#L304

```
File: VaultSecondaryBorrow.sol
304:     function _updateTotalSecondaryDebt(
305:         VaultConfig memory vaultConfig,
306:         address account,
307:         uint16 currencyId,
308:         uint256 maturity,
309:         int256 netUnderlyingDebt,
310:         PrimeRate memory pr
311:     ) private {
```

```
312:         VaultStateStorage storage balance =
↪  LibStorage.getVaultSecondaryBorrow()
313:             [vaultConfig.vault][maturity][currencyId];
314:         int256 totalDebtUnderlying =
↪  VaultStateLib.readDebtStorageToUnderlying(pr, maturity, balance.totalDebt);
315:
316:         // Set the new debt underlying to storage
317:         totalDebtUnderlying = totalDebtUnderlying.add(netUnderlyingDebt);
318:         VaultStateLib.setTotalDebtStorage(
319:             balance, pr, vaultConfig, currencyId, maturity,
↪  totalDebtUnderlying, false // not settled
320:         );
```

However, this approach was not consistently applied when handling dust balance in secondary debt within the `_updateTotalSecondaryDebt` function. Within the `_updateTotalSecondaryDebt` function, the dust balance in secondary debts is not truncated.

Impact

The inconsistency in handling dust balances in primary and secondary debt could potentially lead to discrepancies in debt accounting within the protocol, accumulation of dust, and result in unforeseen consequences.

Code Snippet

https://github.com/sherlock-audit/2023-03-notional-0xleastwood/blob/main/contracts-v2/contracts/internal/vaults/VaultSecondaryBorrow.sol#L304

Tool used

Manual Review

Recommendation

Consider truncating dust balance in secondary debt within the `_updateTotalSecondaryDebt` function similar to what has been done for primary debt.

Discussion

jeffywu

Valid, medium severity looks good

jeffywu

Fixed in: https://github.com/notional-finance/contracts-v2/pull/137

xiaoming9090

@0xleastwood + @xiaoming9090 : Understood from the team that the truncation of dust is no longer necessary. Thus, they have been removed from the codebase. Update made in PR https://github.com/notional-finance/contracts-v2/pull/137

jacksanford1

Sherlock note: Classifying this as fixed.

Issue M-20: No minimum borrow size check against secondary debts

Source: https://github.com/sherlock-audit/2023-03-notional-judging/issues/212

Found by

xiaoming90 + 0xleastwood

Summary

Secondary debts were not checked against the minimum borrow size during exit, which could lead to accounts with insufficient debt becoming insolvent and the protocol incurring bad debts.

Vulnerability Detail

https://github.com/sherlock-audit/2023-03-notional-0xleastwood/blob/main/contracts-v2/contracts/internal/vaults/VaultAccount.sol#L140

```
File: VaultAccount.sol
121:     function _setVaultAccount(
..SNIP..
130:         // An account must maintain a minimum borrow size in order to enter
 ↪  the vault. If the account
131:         // wants to exit under the minimum borrow size it must fully exit
 ↪  so that we do not have dust
132:         // accounts that become insolvent.
133:         if (
134:             vaultAccount.accountDebtUnderlying.neg() <
 ↪  vaultConfig.minAccountBorrowSize &&
135:             // During local currency liquidation and settlement, the min
 ↪  borrow check is skipped
136:             checkMinBorrow
137:         ) {
138:             // NOTE: use 1 to represent the minimum amount of vault shares
 ↪  due to rounding in the
139:             // vaultSharesToLiquidator calculation
140:             require(vaultAccount.accountDebtUnderlying == 0 ||
 ↪  vaultAccount.vaultShares <= 1, "Min Borrow");
141:         }
```

A vault account has one primary debt (`accountDebtUnderlying`) and one or more secondary debts (`accountDebtOne` and `accountDebtTwo`).

When a vault account exits the vault, Notional will check that its primary debt (`accountDebtUnderlying`) meets the minimum borrow size requirement. If a vault account wants to exit under the minimum borrow size it must fully exit so that we do not have dust accounts that become insolvent. This check is being performed in Line 140 above.

However, this check is not performed against the secondary debts. As a result, it is possible that the secondary debts fall below the minimum borrow size after exiting.

Impact

Vault accounts with debt below the minimum borrow size are at risk of becoming insolvent, leaving the protocol with bad debts.

Code Snippet

https://github.com/sherlock-audit/2023-03-notional-0xleastwood/blob/main/contracts-v2/contracts/internal/vaults/VaultAccount.sol#L140

Tool used

Manual Review

Recommendation

Consider performing a similar check against the secondary debts (`accountDebtOne` and `accountDebtTwo`) within the `_setVaultAccount` function to ensure they do not fall below the minimum borrow size.

Discussion

jeffywu

Valid issue

jeffywu

Upon further review, it seems like the issue is invalid. Minimum borrow checks are applied every time debt changes.

Issue M-21: It may be possible to liquidate on behalf of another account

Source: https://github.com/sherlock-audit/2023-03-notional-judging/issues/215

Found by

ShadowForce, xiaoming90 + 0xleastwood

Summary

If the caller of any liquidation action is the vault itself, there is no validation of the `liquidator` parameter and therefore, any arbitrary account may act as the liquidator if they have approved any amount of funds for the `VaultLiquidationAction.sol` contract.

Vulnerability Detail

While the vault implementation itself should most likely handle proper validation of the parameters provided to actions enabled by the vault, the majority of important validation should be done within the Notional protocol. The base implementation for vaults does not seem to sanitise `liquidator` and hence users could deleverage accounts on behalf of a liquidator which has approved Notional's contracts.

https://github.com/sherlock-audit/2023-03-notional-0xleastwood/blob/main/contracts-v2/contracts/external/actions/VaultLiquidationAction.sol#L197-L237

```
File: VaultLiquidationAction.sol
197:     function _authenticateDeleverage(
198:         address account,
199:         address vault,
200:         address liquidator
201:     ) private returns (
202:         VaultConfig memory vaultConfig,
203:         VaultAccount memory vaultAccount,
204:         VaultState memory vaultState
205:     ) {
206:         // Do not allow invalid accounts to liquidate
207:         requireValidAccount(liquidator);
208:         require(liquidator != vault);
209:
210:         // Cannot liquidate self, if a vault needs to deleverage itself as
↪  a whole it has other methods
211:         // in VaultAction to do so.
212:         require(account != msg.sender);
213:         require(account != liquidator);
```

```
214:
215:         vaultConfig = VaultConfiguration.getVaultConfigStateful(vault);
216:         require(vaultConfig.getFlag(VaultConfiguration.DISABLE_DELEVERAGE)
 ↪  == false);
217:
218:         // Authorization rules for deleveraging
219:         if (vaultConfig.getFlag(VaultConfiguration.ONLY_VAULT_DELEVERAGE)) {
220:             require(msg.sender == vault);
221:         } else {
222:             require(msg.sender == liquidator);
223:         }
224:
225:         vaultAccount = VaultAccountLib.getVaultAccount(account,
 ↪  vaultConfig);
226:
227:         // Vault accounts that are not settled must be settled first by
 ↪  calling settleVaultAccount
228:         // before liquidation. settleVaultAccount is not permissioned so
 ↪  anyone may settle the account.
229:         require(block.timestamp < vaultAccount.maturity, "Must Settle");
230:
231:         if (vaultAccount.maturity == Constants.PRIME_CASH_VAULT_MATURITY) {
232:             // Returns the updated prime vault state
233:             vaultState =
 ↪  vaultAccount.accruePrimeCashFeesToDebtInLiquidation(vaultConfig);
234:         } else {
235:             vaultState = VaultStateLib.getVaultState(vaultConfig,
 ↪  vaultAccount.maturity);
236:         }
237:     }
```

Impact

A user may be forced to liquidate an account they do not wish to purchase vault shares for.

Code Snippet

https://github.com/notional-finance/leveraged-vaults/blob/master/contracts/vaults/BaseStrategyVault.sol#L204-L216

Tool used

Manual Review

Recommendation

Make the necessary changes to `BaseStrategyVault.sol` or `_authenticateDeleverage()`, whichever is preferred.

Discussion

jeffywu

Valid issue, the fix will be made in BaseStrategyVault. It's unclear how this would be done in `_authenticateDeleverage()`, but I am open to suggestions from the auditors there.

Jiaren-tang

Escalate for 10 USDC. Severity is high because this force liquidator to liquidate the position they are not willing to and make the liquiditator lose fund

sherlock-admin

> Escalate for 10 USDC. Severity is high because this force liquidator to liquidate the position they are not willing to and make the liquiditator lose fund

You've created a valid escalation for 10 USDC!

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.

hrishibhat

Result: Medium Has duplicates Given there preconditions for the issue that results in an undesirable outcome, there is no clear loss of funds to consider this a high issue. This is a valid medium.

sherlock-admin

Escalations have been resolved successfully!

Escalation status:

- ShadowForce: rejected

jeffywu

Fixed in: https://github.com/notional-finance/leveraged-vaults/pull/55

xiaoming9090

@0xleastwood + @xiaoming9090 : Verified. Fixed in PR https://github.com/notional-finance/leveraged-vaults/pull/55