SHERLOCK

SHERLOCK SECURITY REVIEW FOR

Contest type:	Public
Prepared for:	Notional
Prepared by:	Sherlock
Lead Security Expert:	xiaoming90
Dates Audited:	June 26 - July 3, 2024
Prepared on:	September 6, 2024

Introduction

This update creates a Leveraged Vault integration with Pendle where Notional users can take leverage to buy PT tokens. It also includes an update to existing vaults that allows incentives to be more flexibly managed.

Scope

Repository: notional-finance/leveraged-vaults

Branch: feat/vault-rewarder-staking-vaults

Commit: 9d0df3326f085e4f732e03fea65e0dee4b7dbf04

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
6	13

Issues not fixed or acknowledged

Medium	High
0	0

Security experts who found valid issues

xiaoming90
ZeroTrust
novaman33
nirohgo
lemonmon
DenTonylifer

blackhole
BiasedMerc
aman
eeyore
denzi_
Ironsidesec

chaduke
brgltd
pseudoArtist
yotov721
0xrobsol
TopStar

Issue H-1: `_splitWithdrawRequest` will make invalid withdraw requests in an edge case

Source: https://github.com/sherlock-audit/2024-06-leveraged-vaults-judging/issues/6

Found by

ZeroTrust, aman, novaman33

Summary

When an account is deleveraged, _splitWithdrawRequest is called so that pending withdraw requests of the account that is being liquidated are split between them and the liquidator. However when the account is being fully liquidated, the old withdraw request is deleted which creates an invalid Id for the liquidator's withdrawRequest.

Vulnerability Detail

In `_splitWithdrawRequest` the request of the from address is being read by storage:

```
WithdrawRequest storage w = VaultStorage.getAccountWithdrawRequest()[_from];
```

Then the following check is made to delete the withdraw request of the from account if all the vault tokens are being taken from him:

```
if (w.vaultShares == vaultShares) {
            // If the resulting vault shares is zero, then delete the request.
 ↪  The _from account's
            // withdraw request is fully transferred to _to
            delete VaultStorage.getAccountWithdrawRequest()[_from];
        }
```

Here the delete keyword is used to reset the withdraw request of the from account. However the `w` variable is still a pointer to this place in storage meaning that resetting VaultStorage.getAccountWithdrawRequest()[_from] will also be resetting `w`. As a result `w.requestId=0` and

```
toWithdraw.requestId = w.requestId;
```

Here the new requestId is equal to 0 which is the default value meaning that this withdraw request will not be recognized by `_finalizeWithdrawsManual` and the other finalize withdraw functions and all these vault shares will be lost. Also if

`initiateWithdraw` is called the old vaultShares will be wiped out for the new shares to be withdrawn.

Impact

Loss of vaut tokens for the liquidator.

Code Snippet

https://github.com/sherlock-audit/2024-06-leveraged-vaults/blob/14d3eaf0445c251c52c86ce88a84a3f5b9dfad94/leveraged-vaults-private/contracts/vaults/common/WithdrawRequestBase.sol#L221

Tool used

Manual Review

Recommendation

Store the requestId of the from address in another memory variable.

Discussion

sherlock-admin2

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

0xmystery commented:

> w has been assigned and will not be reset

novaman33

Escalate, This is a valid issue. w is a pointer and is being reset when mapping is deleted.

sherlock-admin3

> Escalate, This is a valid issue. w is a pointer and is being reset when mapping is deleted.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

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

amankakar

This Issue is dup of #89

You can observe that there is no return or halt in the flow when a request is deleted. The same `requestID`, which has been deleted, retains its storage scope. Consequently, the `requestID` assigned to the new withdrawal request for the `liquidator` will always be zero in this scenario. This is a significant issue that could result in asset loss and needs to be reconsidered. For proof, I have also added simple code demonstrating that the `requestID` will be 0 after deletion.

mystery0x

> This Issue is dup of #89
>
> You can observe that there is no return or halt in the flow when a request is deleted. The same `requestID`, which has been deleted, retains its storage scope. Consequently, the `requestID` assigned to the new withdrawal request for the `liquidator` will always be zero in this scenario. This is a significant issue that could result in asset loss and needs to be reconsidered. For proof, I have also added simple code demonstrating that the `requestID` will be 0 after deletion.

Please provide a coded POC alleging your claim. I will also request the sponsors looking into your report.

novaman33

@mystery0x

```
// SPDX-License-Identifier: UNLICENSED

pragma solidity 0.8.22;

import "forge-std/Test.sol";

struct WithdrawRequest {
    uint256 requestId;
    uint256 vaultShares;
    bool hasSplit;
}
contract TestRequestID is Test {
    address alice = makeAddr("aLice");
    address bob = makeAddr("bob");

    struct SplitWithdrawRequest {
        uint256 totalVaultShares; // uint64
        uint256 totalWithdraw; // uint184?
        bool finalized;
    }
```

```
    function setUp() external {}

    uint256 public constant DISTRIBUTION_DIVISOR = 100 ether;

    function testRequestId() external {
        _createRequest();
        _splitWithdrawRequest(address(alice), address(bob), 100);
        WithdrawRequest storage w = VaultStorage.getAccountWithdrawRequest()[
            bob
        ];
        assert(w.requestId == 0);
    }

    function _splitWithdrawRequest(
        address _from,
        address _to,
        uint256 vaultShares
    ) internal {
        WithdrawRequest storage w = VaultStorage.getAccountWithdrawRequest()[
            _from
        ];
        if (w.requestId == 0) return;
        // @audit : Ingore this code as it does not have any impact in our case
        // Create a new split withdraw request
        // if (!w.hasSplit) {
        //     SplitWithdrawRequest memory s = VaultStorage
        //         .getSplitWithdrawRequest()[w.requestId];
        //     // Safety check to ensure that the split withdraw request is not
↪   active, split withdraw
        //     // requests are never deleted. This presumes that all withdraw
↪   request ids are unique.
        //     require(s.finalized == false && s.totalVaultShares == 0);
        //     VaultStorage
        //     .getSplitWithdrawRequest()[w.requestId].totalVaultShares = w
        //         .vaultShares;
        // }

        console.log("Request ID is Before Delete :", w.requestId);

        if (w.vaultShares == vaultShares) {
            // If the resulting vault shares is zero, then delete the request.
↪   The _from account's
            // withdraw request is fully transferred to _to
            delete VaultStorage.getAccountWithdrawRequest()[_from];
        } else {
            // Otherwise deduct the vault shares
            w.vaultShares = w.vaultShares - vaultShares;
```


```
            w.hasSplit = true;
        }

        // Ensure that no withdraw request gets overridden, the _to account
↪   always receives their withdraw
        // request in the account withdraw slot.
        WithdrawRequest storage toWithdraw = VaultStorage
            .getAccountWithdrawRequest()[_to];
        require(
            toWithdraw.requestId == 0 || toWithdraw.requestId == w.requestId,
            "Existing Request"
        );

        console.log("Request ID is After Delete :", w.requestId);
        // Either the request gets set or it gets incremented here.
        toWithdraw.requestId = w.requestId; // @audit : the request id will be
↪   zero here
        toWithdraw.vaultShares = toWithdraw.vaultShares + vaultShares;
        toWithdraw.hasSplit = true;
    }

    function _createRequest() internal {
        WithdrawRequest storage accountWithdraw = VaultStorage
            .getAccountWithdrawRequest()[alice];

        accountWithdraw.requestId = uint256(uint160(address(alice)));
        accountWithdraw.vaultShares = 100;
        accountWithdraw.hasSplit = false;
    }
}

library VaultStorage {
    /// @notice Emitted when vault settings are updated
    // event StrategyVaultSettingsUpdated(StrategyVaultSettings settings);
    // Wrap timestamp in a struct so that it can be passed around as a storage
↪   pointer
    struct LastClaimTimestamp {
        uint256 value;
    }

    /// @notice account initiated WithdrawRequest
    uint256 private constant ACCOUNT_WITHDRAW_SLOT = 1000008;

    function getAccountWithdrawRequest()
        internal
        pure
        returns (mapping(address => WithdrawRequest) storage store)
```


```
    {
        assembly {
            store.slot := ACCOUNT_WITHDRAW_SLOT
        }
    }
}
```

```
Request ID is Before Delete : 964133639515395071211053928642046522704087555788
Request ID is After Delete : 0
```

WangSecurity

I agree with the escalation, it is a nice find. Planning to accept it and validate it with High severity since it will happen with every full liquidation. The duplicate is #89, @mystery0x are there other duplicates as well?

brakeless-wtp

#41

WangSecurity

Result: High Has duplicates

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- novaman33: accepted

mystery0x

> I agree with the escalation, it is a nice find. Planning to accept it and validate it with High severity since it will happen with every full liquidation. The duplicate is #89, @mystery0x are there other duplicates as well?

#6, #41, and #89 are the only three reports submitting this finding.

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/notional-finance/leveraged-vaults/pull/106

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue H-2: Lido withdraw limitation will brick the withdraw process in an edge case

Source: https://github.com/sherlock-audit/2024-06-leveraged-vaults-judging/issues/14

Found by

novaman33

Summary

Lido protocol has limitation regarding the requestWithdraw function. However some of these limitation have not been considered in the `_initiateWithdrawImpl` leading to users being unable to claim their vault shares even after the cooldown.

Vulnerability Detail

Lido has stated the following withdraw limitation in their docs:

1. withdrawals must not be paused
2. stETH balance of msg.sender must be greater than the sum of all _amounts
3. there must be approval from the msg.sender to this contract address for the overall amount of stETH token transfer
4. each amount in _amounts must be greater than MIN_STETH_WITHDRAWAL_AMOUNT and lower than MAX_STETH_WITHDRAWAL_AMOUNT (values that can be changed by the DAO) Extracted from here(https://docs.lido.fi/contracts/withdrawal-queue-erc721#requestwithdrawals)

Consider the following scenario:

1) A user who has shares representing stETH less than the MIN_STETH_WITHDRAWAL_AMOUNT or more than the MAX_STETH_WITHDRAWAL_AMOUNT calls `initiateWithdraw`. The withdraw will be initiated successfully and the rsETH to withdraw will be sent to the holder contract which is going to start the cooldown.
2) However after the cooldown has passed the user will call the `triggerExtraStep` function which will always result in revert because of the Lido requirements regarding the amount to be withdrawn(mentioned in point 4).

Impact

The user will experience a full DOS of the protocol. They will have a pending withdraw that will never finish, which will result in their funds being locked forever. They will not be able to liquidate or deposit because of the pending withdraw. The function `triggerExtraStep` will always revert and the tokens from Kelp will never be claimed, because of Lido's limitation. - High

Code Snippet

https://github.com/sherlock-audit/2024-06-leveraged-vaults/blob/14d3eaf0445c251c52c86ce88a84a3f5b9dfad94/leveraged-vaults-private/contracts/vaults/staking/protocols/Kelp.sol#L83

Tool used

Manual Review

Recommendation

Consider enforcing withdraw limitations so that if a user has more than the MAX_STETH_WITHDRAWAL_AMOUNT split it on two requests, or create deposit limitations.

Discussion

sherlock-admin2

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

0xmystery commented:

> Calls made by the Notional proxy (i.e. depositFromNotional and redeemFromNotional) are restricted by deposit sizes and maximum leverage ratios enforced by the main Notional contract

novaman33

Escalate, This is a valid issue that will result in permanent lock of funds. The way I understand your comment is that there can be deposit limitations, however I do not consider them to be relevant here as a user could simply deposit more than once or gain funds by liquidating others. Also these min and max values by the Lido protocol are changable by the DAO. The issue is that in the same function funds are pulled from Kelp and sent to Lido and these limitations will cause a revert of the whole function. I did review the code and I did not see any restrictions as you have stated. In case I am missing something could you provide a reference to the check you have considered in order to invalidate the issue?

sherlock-admin3

> Escalate, This is a valid issue that will result in permanent lock of funds. The way I understand your comment is that there can be deposit limitations, however I do not consider them to be relevant here as a user could simply deposit more than once or gain funds by liquidating others. Also these min and max values by the Lido protocol are changable by the DAO. The issue is that in the same function funds are pulled from Kelp and sent to Lido and these limitations will cause a revert of the whole function. I did review the code and I did not see any restrictions as you have stated. In case I am missing something could you provide a reference to the check you have considered in order to invalidate the issue?

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

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

mystery0x

> Escalate, This is a valid issue that will result in permanent lock of funds. The way I understand your comment is that there can be deposit limitations, however I do not consider them to be relevant here as a user could simply deposit more than once or gain funds by liquidating others. Also these min and max values by the Lido protocol are changable by the DAO. The issue is that in the same function funds are pulled from Kelp and sent to Lido and these limitations will cause a revert of the whole function. I did review the code and I did not see any restrictions as you have stated. In case I am missing something could you provide a reference to the check you have considered in order to invalidate the issue?

It's found in the contest `Details` page. These issues are commonly known and will be low at most for the suggested mitigation you provided.

novaman33

@mystery0x There are no restriction regarding how much the user can deposit. The restrictions are so that the position is healthy and the leverage ratio is less than the maximum ratio. The issue is that if a user tries to withdraw less than 100 wei they will experience a full DOS since they will have a withdraw request that will be pending but will not be able to finalize it since the triggerExtraStep will always revert because of the lido limitation. The other case however is much more scary. If a whale staker comes, and they have more than 1000ETH(which is the MAX_STETH_WITHDRAWAL_AMOUNT), when trying to withdraw this amount of

money, triggerExtraStep will always revert, they will have a pending withdrawRequest meaning they will not be able to deposit or liquidate. While you did mention the contest page statement about the restrictions in deposit sizers I did review them and both cases are possible with these restrictions. I cannot agree that permanently locking funds(which will happen in both cases) and experiencing full DOS is a low severity case.

WangSecurity

This is the `holder` contract, correct? https://github.com/sherlock-audit/2024-06-leveraged-vaults/blob/14d3eaf0445c251c52c86ce88a84a3f5b9dfad94/leveraged-vaults-private/contracts/vaults/staking/protocols/ClonedCoolDownHolder.sol

It has a rescue function to retrieve the tokens. Also, can you send a link to `initiateWithdraw`, there are several functions with similar names, want to understand what you talk about specifically.

novaman33

@WangSecurity, So in order for a user to withdraw, they first call initiateWithdraw(BaseStakingVault.sol), which calls _initiateWithdraw(WithdrawRequestBase.sol), which than calls `_initiateWithdrawImpl`(this is from the Kelp.sol in this case). In `_initiateWithdrawImpl`(Kelp.sol), a holder contract is made and rsETH is transfered to the new holder address. Than holder.startColldown is called. `_startCooldown` transfers all the rsETH balance of the holder contract to Kelp and calls `WithdrawManager.initiateWithdrawal(stETH, balance);` The problem is in the `triggerExtraStep` function, where firstly the withdraw is completed from Kelp and then `LidoWithdraw.requestWithdrawals(amounts, address(this));` is called, which has the limitations mentioned in this report. The main issue is that the holder contract has no other way to take the stETH tokens from Kelp, but only triggerExtraStep which will result in a revert in the forementioned edge cases. The rescueTokens function will not be able to retrieve those tokens because they are not stuck in the holder contract but are stuck in Kelp with no way to be claimed. The restrictions mentioned by the judge are from the readMe but they are not applicable in this case because they do not prevent users from withdrawing less than 100wei or more than 1000ETH. There are only leverage restrictions which are to keep positions healthy. These however are irrelevant in this specific case.

WangSecurity

Thank you for that clarifications. For the scenario about minimum withdrawal I would consider low, since the loss if 100 wei. But for the maximum is completely viable. I believe the high severity is appropriate here, since it will affect every whale investor in Notional and causes complete loss of funds.

Planning to accept the escalation and validate with high severity. Are there any duplicates @novaman33 @mystery0x ?

novaman33

@WangSecurity believe it is solo.

WangSecurity

Result: High Unique

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- novaman33: accepted

jeffywu

This issue is resolved by removing the Lido stETH withdraw in favor of an ETH withdraw.

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/notional-finance/leveraged-vaults/pull/92

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue H-3: Selling sUSDe is vulnerable to sandwich attack when staked token is DAI

Source: https://github.com/sherlock-audit/2024-06-leveraged-vaults-judging/issues/18

Found by

0xrobsol, Ironsidesec, TopStar, ZeroTrust, aman, blackhole, chaduke, denzi_, lemonmon, yotov721

Summary

The protocol has functionality that makes a trade in 2 parts (legs). It only has slippage protection in the second part, but the second part is only executed in certain conditions, leaving the trade without slippage protection.

Vulnerability Detail

The protocol offers users the functionality to leverage stake and receive leveraged yield. This can be achieved by a **borrowed** token being wrapped or exchanged into a staking token that is pegged to the borrow token's value.

Note that the user may instead deposit the borrow token directly.

One of the tokens Notional uses is Ethena's `USDe` and `sUSDe`. The user would be receiving leveraged `USDe` yield. In the case of Ethena/Notional the borrowed token is DAI.

Once a user wants to exit `BaseStakingVault::_redeemFromNotional` is called. There are two options instant redemption or through the withdraw request functionality. If instant redemption is used `EthenaLib::_sellStakedUSDe` is called.

```
function _executeInstantRedemption(
    address /* account */,
    uint256 vaultShares,
    uint256 /* maturity */,
    RedeemParams memory params
) internal override returns (uint256 borrowedCurrencyAmount) {
    uint256 sUSDeToSell = getStakingTokensForVaultShare(vaultShares);

    // Selling sUSDe requires special handling since most of the liquidity
    // sits inside a sUSDe/sDAI pool on Curve.
    return EthenaLib._sellStakedUSDe(
        sUSDeToSell, BORROW_TOKEN, params.minPurchaseAmount,
↪   params.exchangeData, params.dexId
```

```
    );
}
```

The `_sellStakedUSDe` function has two trades. The first one swapping from `sUSDe` to `sDAI`. The second is only executed if the borrow token is NOT `DAI` as seen in the code snippet bellow.

```
    function _sellStakedUSDe(
        uint256 sUSDeAmount,
        address borrowToken,
        uint256 minPurchaseAmount,
        bytes memory exchangeData,
        uint16 dexId
    ) internal returns (uint256 borrowedCurrencyAmount) {
        Trade memory sDAITrade = Trade({
            tradeType: TradeType.EXACT_IN_SINGLE,
            sellToken: address(sUSDe),
            buyToken: address(sDAI),
            amount: sUSDeAmount,
            limit: 0, // NOTE: no slippage guard is set here, it is enforced in
 ↪  the second leg
                      // of the trade.
            deadline: block.timestamp,
            exchangeData: abi.encode(CurveV2Adapter.CurveV2SingleData({
                pool: 0x167478921b907422F8E88B43C4Af2B8BEa278d3A,
                fromIndex: 1, // sUSDe
                toIndex: 0 // sDAI
            }))
        });


        (/* */, uint256 sDAIAmount) =
 ↪  sDAITrade._executeTrade(uint16(DexId.CURVE_V2));


        // Unwraps the sDAI to DAI
        uint256 daiAmount = sDAI.redeem(sDAIAmount, address(this),
 ↪  address(this));

=>      if (borrowToken != address(DAI)) {
            Trade memory trade = Trade({
                tradeType: TradeType.EXACT_IN_SINGLE,
                sellToken: address(DAI),
                buyToken: borrowToken,
                amount: daiAmount,
                limit: minPurchaseAmount,
```

```
                deadline: block.timestamp,
                exchangeData: exchangeData
            });


            // Trades the unwrapped DAI back to the given token.
            (/* */, borrowedCurrencyAmount) = trade._executeTrade(dexId);
        } else {
            borrowedCurrencyAmount = daiAmount;
        }
    }
```

There is NO slippage protection on the first trade. The reason being that slippage is checked in the second trade. However the second trade is only executed in the borrow token is NOT `DAI`. This opens the possibility of the trade being sandwich attacked by MEV bots stealing large portions of user funds, if the borrowed token is DAI, because the second trade would NOT be executed. Hence no slippage at all would be enforced in the transaction.

Impact

Loss of funds due to sandwich attack

Code Snippet

https://github.com/sherlock-audit/2024-06-leveraged-vaults/blob/14d3eaf0445c251c52c86ce88a84a3f5b9dfad94/leveraged-vaults-private/contracts/vaults/staking/protocols/Ethena.sol#L124-L167

Tool used

Manual Review

Recommendation

Add a slippage parameter to the first trade as well or use the `minPurchaseAmount` parameter as a minAmountOut:

```
    function _sellStakedUSDe(
        uint256 sUSDeAmount,
        address borrowToken,
        uint256 minPurchaseAmount,
        bytes memory exchangeData,
        uint16 dexId
    ) {
```

```
...

        uint256 daiAmount = sDAI.redeem(sDAIAmount, address(this),
↪   address(this));

        if (borrowToken != address(DAI)) {
            Trade memory trade = Trade({
                tradeType: TradeType.EXACT_IN_SINGLE,
                sellToken: address(DAI),
                buyToken: borrowToken,
                amount: daiAmount,
                limit: minPurchaseAmount,
                deadline: block.timestamp,
                exchangeData: exchangeData
            });

            // Trades the unwrapped DAI back to the given token.
            (/* */, borrowedCurrencyAmount) = trade._executeTrade(dexId);
        } else {
            borrowedCurrencyAmount = daiAmount;
+           require(borrowedCurrencyAmount >= minPurchaseAmount,
↪   "NotEnoughtAmountOut");
        }
    }
```

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/notional-finance/leveraged-vaults/pull/93/files

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue H-4: `EtherFiLib::_initiateWithdrawImpl` will revert because rebase tokens transfer 1-2 less wei

Source: https://github.com/sherlock-audit/2024-06-leveraged-vaults-judging/issues/43

Found by

ZeroTrust

Summary

`EtherFiLib::_initiateWithdrawImpl` will revert because rebase tokens transfer 1-2 less wei

Vulnerability Detail

The protocol always assumes that the amount of tokens received is equal to the amount of tokens transferred. This is not the case for rebasing tokens, such as stETH and eETH, because internally they transfer shares which generally results in the received amount of tokens being lower than the requested one by a couple of wei because of roundings: transferring 1e18 eETH tokens from A to B, will may result in B receiving 0.99999e18 eETH tokens.

```
function _initiateWithdrawImpl(uint256 weETHToUnwrap) internal returns (uint256
↪  requestId) {
@>        uint256 eETHReceived = weETH.unwrap(weETHToUnwrap);
        eETH.approve(address(LiquidityPool), eETHReceived);
@>        return LiquidityPool.requestWithdraw(address(this), eETHReceived);
    }
```

- 1 Unwraps weETHToUnwrap weETH, meaning Transfers eETHReceived of eETH from the weETH contract to this contract itself
- 2 RequestWithdraw eETHReceived, which will attempt to transfer eETHReceived from the contract to the Etherfi protocol But the actual amount of eETH received may be eETHReceived - 1~2 wei.

Step 2 will fail, because the contract doesn't have enough eETH. The issue lies in attempting to transfer eETHReceived of eETH in step 2 instead of wrapping the actual amount of tokens received.

Impact

Contract functionality DoS

Code Snippet

https://github.com/sherlock-audit/2024-06-leveraged-vaults/blob/main/leveraged-vaults-private/contracts/vaults/staking/protocols/EtherFi.sol#L24

Tool used

Manual Review

Recommendation

```
function _initiateWithdrawImpl(uint256 weETHToUnwrap) internal returns (uint256
↪  requestId) {
+       uint256 balanceBefore = eETH.balanceOf(address(this));
        uint256 eETHReceived = weETH.unwrap(weETHToUnwrap);
+       uint256 balanceAfter = eETH.balanceOf(address(this));
-        eETH.approve(address(LiquidityPool), eETHReceived);
-        return LiquidityPool.requestWithdraw(address(this), eETHReceived);
+         eETH.approve(address(LiquidityPool), balanceAfter - balanceBefore);
+        return LiquidityPool.requestWithdraw(address(this), balanceAfter -
↪  balanceBefore);
    }
```

Discussion

sherlock-admin2

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

0xmystery commented:

> It concerns deposit(), NOT wrap/unwrap

ZeroTrust01

Escalate This should be considered a valid issue.

```
function unwrap(uint256 _weETHAmount) external returns (uint256) {
        require(_weETHAmount > 0, "Cannot unwrap a zero amount");
        uint256 eETHAmount = liquidityPool.amountForShare(_weETHAmount);
        _burn(msg.sender, _weETHAmount);
@>>         eETH.transfer(msg.sender, eETHAmount);
        return eETHAmount;
    }
```

The weETH::unwrap() function includes a transfer operation, so this involves the transfer of rebase tokens in this contract. In the LiquidityPool.requestWithdraw

function, rebase tokens are transferred out of this contract because internally they transfer shares, which generally results in the received amount of tokens being lower than the requested one by a couple of wei due to rounding, So it will revert.

This is exactly the same issue as the one confirmed in the contest a few weeks ago. https://github.com/sherlock-audit/2024-05-sophon-judging/issues/63 https://github.com/sherlock-audit/2024-05-sophon-judging/issues/119

sherlock-admin3

> Escalate This should be considered a valid issue.
>
> ```
> function unwrap(uint256 _weETHAmount) external returns (uint256) {
>         require(_weETHAmount > 0, "Cannot unwrap a zero amount");
>         uint256 eETHAmount = liquidityPool.amountForShare(_weETHAmount);
>         _burn(msg.sender, _weETHAmount);
> @>>         eETH.transfer(msg.sender, eETHAmount);
>         return eETHAmount;
>     }
> ```
>
> The weETH::unwrap() function includes a transfer operation, so this involves the transfer of rebase tokens in this contract. In the LiquidityPool.requestWithdraw function, rebase tokens are transferred out of this contract because internally they transfer shares, which generally results in the received amount of tokens being lower than the requested one by a couple of wei due to rounding, So it will revert.
>
> This is exactly the same issue as the one confirmed in the contest a few weeks ago.
> https://github.com/sherlock-audit/2024-05-sophon-judging/issues/63
> https://github.com/sherlock-audit/2024-05-sophon-judging/issues/119

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

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

mystery0x

> Escalate This should be considered a valid issue.
>
> ```
> function unwrap(uint256 _weETHAmount) external returns (uint256) {
>         require(_weETHAmount > 0, "Cannot unwrap a zero amount");
>         uint256 eETHAmount = liquidityPool.amountForShare(_weETHAmount);
>         _burn(msg.sender, _weETHAmount);
> @>>         eETH.transfer(msg.sender, eETHAmount);
> ```

```
        return eETHAmount;
    }
```

The weETH::unwrap() function includes a transfer operation, so this involves the transfer of rebase tokens in this contract. In the LiquidityPool.requestWithdraw function, rebase tokens are transferred out of this contract because internally they transfer shares, which generally results in the received amount of tokens being lower than the requested one by a couple of wei due to rounding, So it will revert.

This is exactly the same issue as the one confirmed in the contest a few weeks ago. sherlock-audit/2024-05-sophon-judging#63 sherlock-audit/2024-05-sophon-judging#119

Here's the selected report for Sophon contest that's related to your finding on eETH:

https://github.com/sherlock-audit/2024-05-sophon-judging/issues/4

Read through the report carefully, and you will notice wrap/unwrap is always 1:1. It's deposit() that's causing the 1-2 wei issue which does not apply to your report context.

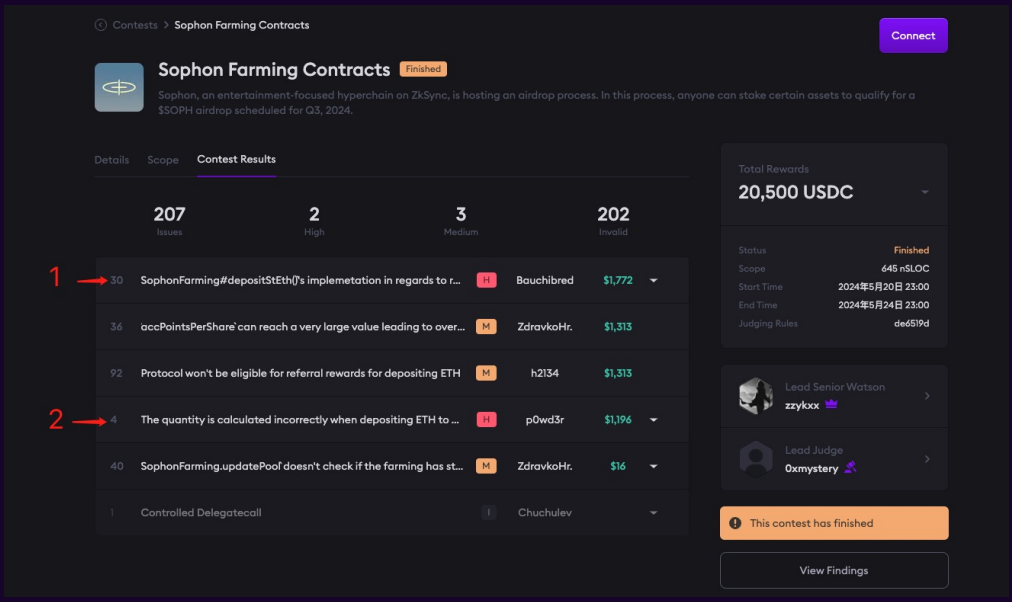
ZeroTrust01

Escalate This should be considered a valid issue.

```
function unwrap(uint256 _weETHAmount) external returns (uint256) {
        require(_weETHAmount > 0, "Cannot unwrap a zero amount");
        uint256 eETHAmount =
↪   liquidityPool.amountForShare(_weETHAmount);
        _burn(msg.sender, _weETHAmount);
@>>         eETH.transfer(msg.sender, eETHAmount);
        return eETHAmount;
    }
```

The weETH::unwrap() function includes a transfer operation, so this involves the transfer of rebase tokens in this contract. In the LiquidityPool.requestWithdraw function, rebase tokens are transferred out of this contract because internally they transfer shares, which generally results in the received amount of tokens being lower than the requested one by a couple of wei due to rounding, So it will revert. This is exactly the same issue as the one confirmed in the contest a few weeks ago. sherlock-audit/2024-05-sophon-judging#63 sherlock-audit/2024-05-sophon-judging#119

> Here's the selected report for Sophon contest that's related to your finding on eETH:
>
> sherlock-audit/2024-05-sophon-judging#4
>
> Read through the report carefully, and you will notice wrap/unwrap is always 1:1. It's deposit() that's causing the 1-2 wei issue which does not apply to your report context.



You mixed up the link I provided. I pointed out that this issue is exactly the same as **issue group 1** (in the picture), but your link is to issue group 2 (in the picture). I never said it was the same as issue group 2.

WangSecurity

@ZeroTrust01 could you provide the link to the eETH contract please, so I can see how they've implemented their transfer function with fees.

ZeroTrust01

> @ZeroTrust01 could you provide the link to the eETH contract please, so I can see how they've implemented their transfer function with fees.

This is the address of the eETH contract.
https://etherscan.io/address/0x35fA164735182de50811E8e2E824cFb9B6118ac2

This issue is not about the transfer function with fees; **it is about rebase token transfer using shares.** If the transfer amount is 1e18, the received amount might be (1e18 - 1 wei).

I learned this from a report where the issue was exactly the same as this one.
https://github.com/sherlock-audit/2024-05-sophon-judging/issues/119

WangSecurity

Thank you. For future reference, historical decisions are not sources of truth and each issue is different, so I would like to ask you instead of saying "this issue is the exact same as in the previous contest", prove that your issue is valid, regardless of other contests and decisions. Often it's the case that the same issue is valid in one contest, but invalid in an another.

About the issue. Even though, Notional's code fetches the `eEthReceived` from `weEth.unwrap` the `eEthReceived` represents the amount before `eEth.transfer`. `eETH.transfer` makes a conversion from input amount to token shares and sends the amount of shares received, which can be lower due to precision loss. Later, this will revert since the contract tries to send `eEthreceived`.

Planning to accept the escalation and validate the report with high severity since it can happen on every withdrawal from EtherFi. @mystery0x @ZeroTrust01 are there any duplicates?

brakeless-wtp

I believe it is unique.

WangSecurity

Result: High Unique

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- ZeroTrust01: accepted

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/notional-finance/leveraged-vaults/pull/94/files

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue H-5: Incorrect valuation of vault share

Source: https://github.com/sherlock-audit/2024-06-leveraged-vaults-judging/issues/60

Found by

blackhole, nirohgo, xiaoming90

Summary

The code for computing the valuation of the vault shares was found to be incorrect. As a result, the account's collateral will be overinflated, allowing malicious users to borrow significantly more than the actual collateral value, draining assets from the protocol.

Vulnerability Detail

Let BT be the borrowed token with 6 decimal precision, and RT be the redemption token with 18 decimal precision.

When a withdraw request has split and is finalized, the following `_getValueOfSplitFinalizedWithdrawRequest` function will be used to calculate the value of a withdraw request in terms of the borrowed token (BT).

In Line 77 below, the `Deployments.TRADING_MODULE.getOraclePrice` function will be called to fetch the exchange rate of the redemption token (RT) and borrowed token (BT).

https://github.com/sherlock-audit/2024-06-leveraged-vaults/blob/main/leveraged-vaults-private/contracts/vaults/common/WithdrawRequestBase.sol#L77

```
File: WithdrawRequestBase.sol
65:     function _getValueOfSplitFinalizedWithdrawRequest(
66:         WithdrawRequest memory w,
67:         SplitWithdrawRequest memory s,
68:         address borrowToken,
69:         address redeemToken
70:     ) internal virtual view returns (uint256) {
71:         // If the borrow token and the withdraw token match, then there is
 ↪  no need to apply
72:         // an exchange rate at this point.
73:         if (borrowToken == redeemToken) {
74:             return (s.totalWithdraw * w.vaultShares) / s.totalVaultShares;
75:         } else {
76:             // Otherwise, apply the proper exchange rate
```

```
77:             (int256 rate, /* */) =
↪  Deployments.TRADING_MODULE.getOraclePrice(redeemToken, borrowToken);
78:
79:             return (s.totalWithdraw * rate.toUint() * w.vaultShares) /
80:                 (s.totalVaultShares * Constants.EXCHANGE_RATE_PRECISION);
81:         }
82:     }
```

Within the `Deployments.TRADING_MODULE.getOraclePrice` function, chainlink oracle will be used. Refer to the source code's comment on Lines 255-257 below for more details. Note that the RT is the base, while the BT is the quote here.

Assume that one BT is worth 1 USD, so the `quotePrice` will be 1e8. Assume that one RT is worth 10 USD, so the `basePrice` will be 10e18. Note that Chainlink oracle's price is always denominated in 8 decimals for USD price feed.

This function will always return the exchange rate is `RATE_DECIMALS` (18 decimals - Hardcoded). Thus, based on the calculation in Lines 283-285, the exchange rate returned will be 10e18, which is equivalent to one unit of RT is worth 10 units of BT.

https://github.com/sherlock-audit/2024-06-leveraged-vaults/blob/main/leveraged-vaults-private/contracts/trading/TradingModule.sol#L283

```
File: TradingModule.sol
255:     /// @notice Returns the Chainlink oracle price between the baseToken
↪  and the quoteToken, the
256:     /// Chainlink oracles. The quote currency between the oracles must
↪  match or the conversion
257:     /// in this method does not work. Most Chainlink oracles are
↪  baseToken/USD pairs.
258:     /// @param baseToken address of the first token in the pair, i.e. USDC
↪  in USDC/DAI
259:     /// @param quoteToken address of the second token in the pair, i.e. DAI
↪  in USDC/DAI
260:     /// @return answer exchange rate in rate decimals
261:     /// @return decimals number of decimals in the rate, currently
↪  hardcoded to 1e18
262:     function getOraclePrice(address baseToken, address quoteToken)
263:         public
264:         view
265:         override
266:         returns (int256 answer, int256 decimals)
267:     {
268:         _checkSequencer();
269:         PriceOracle memory baseOracle = priceOracles[baseToken];
270:         PriceOracle memory quoteOracle = priceOracles[quoteToken];
271:
```

```
272:         int256 baseDecimals = int256(10**baseOracle.rateDecimals);
273:         int256 quoteDecimals = int256(10**quoteOracle.rateDecimals);
274:
275:         (/* */, int256 basePrice, /* */, uint256 bpUpdatedAt, /* */) =
 ↪  baseOracle.oracle.latestRoundData();
276:         require(block.timestamp - bpUpdatedAt <=
 ↪  maxOracleFreshnessInSeconds);
277:         require(basePrice > 0); /// @dev: Chainlink Rate Error
278:
279:         (/* */, int256 quotePrice, /* */, uint256 qpUpdatedAt, /* */) =
 ↪  quoteOracle.oracle.latestRoundData();
280:         require(block.timestamp - qpUpdatedAt <=
 ↪  maxOracleFreshnessInSeconds);
281:         require(quotePrice > 0); /// @dev: Chainlink Rate Error
282:
283:         answer =
284:             (basePrice * quoteDecimals * RATE_DECIMALS) /
285:             (quotePrice * baseDecimals);
286:         decimals = RATE_DECIMALS;
287:     }
```

The `rate` at Line 77 below will be 10e18 based on our earlier calculation. Note the following:

- `s.totalWithdraw` is the total RT claimed and is denominated in 18 decimals (Token's native precision). Assume that `s.totalWithdraw=100e18 RT` was claimed.
- `w.vaultShares` and `s.totalVaultShares` are the number of vault shares and is denominated in 8 decimals (`INTERNAL_TOKEN_PRECISION`). Assume that `w.vaultShares=5e8` and `s.totalVaultShares=10e8`
- `Constants.EXCHANGE_RATE_PRECISION` is 1e18

Intuitively, the split's total withdraw (`s.totalWithdraw`) is 100 units of RT. In terms of the borrowed token (BT), it will be 1000 units of BT since the price is (1:10). Since the withdraw request owns 50% of the vault shares in the split withdraw request, it is entitled to 500 units of BT.

https://github.com/sherlock-audit/2024-06-leveraged-vaults/blob/main/leveraged-vaults-private/contracts/vaults/common/WithdrawRequestBase.sol#L77

```
File: WithdrawRequestBase.sol
65:     function _getValueOfSplitFinalizedWithdrawRequest(
66:         WithdrawRequest memory w,
67:         SplitWithdrawRequest memory s,
68:         address borrowToken,
69:         address redeemToken
```

```
70:     ) internal virtual view returns (uint256) {
71:         // If the borrow token and the withdraw token match, then there is
    ↪  no need to apply
72:         // an exchange rate at this point.
73:         if (borrowToken == redeemToken) {
74:             return (s.totalWithdraw * w.vaultShares) / s.totalVaultShares;
75:         } else {
76:             // Otherwise, apply the proper exchange rate
77:             (int256 rate, /* */) =
    ↪  Deployments.TRADING_MODULE.getOraclePrice(redeemToken, borrowToken);
78:
79:             return (s.totalWithdraw * rate.toUint() * w.vaultShares) /
80:                 (s.totalVaultShares * Constants.EXCHANGE_RATE_PRECISION);
81:         }
82:     }
```

To calculate the value of a withdraw request in terms of the borrowed token (BT), the following formula at Line 79 above will be used:

```
(s.totalWithdraw * rate * w.vaultShares) / (s.totalVaultShares *
↪  Constants.EXCHANGE_RATE_PRECISION)
(100e18 * 10e18 * 5e8) / (10e8 * 1e18)
500000000000000000000
500000000000000e6
```

However, the code above indicates that the withdraw request is entitled to 500000000000000 units of BT instead of 500 units of BT, which is overly inflated. As a result, the account's collateral will be overly inflated.

Impact

The account's collateral will be overinflated, allowing malicious users to borrow significantly more than the actual collateral value, stealing assets from the protocol.

Code Snippet

https://github.com/sherlock-audit/2024-06-leveraged-vaults/blob/main/leveraged-vaults-private/contracts/vaults/common/WithdrawRequestBase.sol#L77

https://github.com/sherlock-audit/2024-06-leveraged-vaults/blob/main/leveraged-vaults-private/contracts/trading/TradingModule.sol#L283

Tool used

Manual Review

Recommendation

Update the formula to as follows:

```
- (s.totalWithdraw * rate.toUint() * w.vaultShares) / (s.totalVaultShares *
↪  Constants.EXCHANGE_RATE_PRECISION);
+ (s.totalWithdraw * rate.toUint() * w.vaultShares * BORROW_PRECISION) /
↪  (s.totalVaultShares * Constants.EXCHANGE_RATE_PRECISION *
↪  REDEMPTION_PRECISION);
```

`BORROW_PRECISION` = 1e6 and `REDEMPTION_PRECISION` = 1e18.

Let's redo the calculation to verify that the new formula works as intended:

```
(s.totalWithdraw * rate.toUint() * w.vaultShares * BORROW_PRECISION) /
↪  (s.totalVaultShares * Constants.EXCHANGE_RATE_PRECISION *
↪  REDEMPTION_PRECISION);
(100e18 * 10e18 * 5e8 * 1e6) / (10e8 * 1e18 * 1e18)
500000000
500e6
```

The new formula returned 500 units of BT, which is correct.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/notional-finance/leveraged-vaults/pull/102

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue H-6: Loss of rewards due to continuous griefing attacks on L2 environment

Source: https://github.com/sherlock-audit/2024-06-leveraged-vaults-judging/issues/61

The protocol has acknowledged this issue.

Found by

nirohgo, xiaoming90

Summary

On the L2 environment (e.g., Arbitrum), due to low transaction fees, it is possible for malicious users to perform griefing attacks against the reward features, leading to a loss of rewards.

Vulnerability Detail

Instance 1 Line 174 (`balanceAfter - balancesBefore[i]`) attempts to compute the number of reward tokens claimed.

Malicious users could call the permissionless `claimRewardTokens` function on every new block, which in turn calls the internal `_claimVaultRewards` function. This is feasible due to the low transaction fees on L2 (Arbitrum), leading to a small number of reward tokens being claimed each time.

https://github.com/sherlock-audit/2024-06-leveraged-vaults/blob/main/leveraged-vaults-private/contracts/vaults/common/VaultRewarderLib.sol#L174

```
File: VaultRewarderLib.sol
148:     function _claimVaultRewards(
149:         uint256 totalVaultSharesBefore,
150:         VaultRewardState[] memory state,
151:         RewardPoolStorage memory rewardPool
152:     ) internal {
153:         uint256[] memory balancesBefore = new uint256[](state.length);
154:         // Run a generic call against the reward pool and then do a balance
155:         // before and after check.
156:         for (uint256 i; i < state.length; i++) {
157:             // Presumes that ETH will never be given out as a reward token.
158:             balancesBefore[i] =
↪   IERC20(state[i].rewardToken).balanceOf(address(this));
159:         }
160:
```

```
161:         _executeClaim(rewardPool);
162:
163:         rewardPool.lastClaimTimestamp = uint32(block.timestamp);
164:         VaultStorage.setRewardPoolStorage(rewardPool);
165:
166:         // This only accumulates rewards claimed, it does not accumulate
 ↪  any secondary emissions
167:         // that are streamed to vault users.
168:         for (uint256 i; i < state.length; i++) {
169:             uint256 balanceAfter =
 ↪  IERC20(state[i].rewardToken).balanceOf(address(this));
170:             _accumulateSecondaryRewardViaClaim(
171:                 i,
172:                 state[i],
173:                 // balanceAfter should never be less than balanceBefore
174:                 balanceAfter - balancesBefore[i],
175:                 totalVaultSharesBefore
176:             );
177:         }
178:     }
```

The `tokensClaimed` will be a very small value, as mentioned earlier. Assume that the precision of the reward token (RT) claimed is 6 decimals precision. The vault shares are denominated in 8 decimals precision (`Constants.INTERNAL_TOKEN_PRECISION`) in Notional.

Assume that the `totalVaultSharesBefore` is 10000000 shares, which is 10000000e8. If the number of reward tokens claimed is less than 10000000 (e.g., 9999999), it will round down to zero. As a result, the reward tokens claimed will be lost.

$$\frac{10000000e8}{1e8} > tokensClaimed$$
$$10000000 > tokensClaimed$$

The issue will get worse when the TVL of the vault grows, and the protocol attracts more liquidity, leading to bigger `totalVaultSharesBefore`, causing the rounding to zero error to be triggered more easily.

https://github.com/sherlock-audit/2024-06-leveraged-vaults/blob/main/leveraged-vaults-private/contracts/vaults/common/VaultRewarderLib.sol#L340

```
File: VaultRewarderLib.sol
332:     function _accumulateSecondaryRewardViaClaim(
333:         uint256 index,
334:         VaultRewardState memory state,
```

```
335:         uint256 tokensClaimed,
336:         uint256 totalVaultSharesBefore
337:     ) private {
338:         if (tokensClaimed == 0) return;
339:
340:         state.accumulatedRewardPerVaultShare += (
341:             (tokensClaimed * uint256(Constants.INTERNAL_TOKEN_PRECISION)) /
 ↪  totalVaultSharesBefore
342:         ).toUint128();
343:
344:         VaultStorage.getVaultRewardState()[index] = state;
345:     }
```

Instance 2 Similarly, the issue mentioned in Issue 1 is also applicable to claiming the reward from emission.

Due to the low transaction fees on L2 (Arbitrum), malicious users could trigger the `_getAccumulatedRewardViaEmissionRate` function on every new block. As a result, the `timeSinceLastAccumulation` in Line 373 below will be a very small value.

The math that leads to rounding zero error is similar to Issue 1. When it round to zero, it will lead to a loss of emission rewards.

https://github.com/sherlock-audit/2024-06-leveraged-vaults/blob/main/leveraged-vaults-private/contracts/vaults/common/VaultRewarderLib.sol#L373

```
File: VaultRewarderLib.sol
347:     function _accumulateSecondaryRewardViaEmissionRate(
348:         uint256 index,
349:         VaultRewardState memory state,
350:         uint256 totalVaultSharesBefore
351:     ) private {
352:         state.accumulatedRewardPerVaultShare =
 ↪  _getAccumulatedRewardViaEmissionRate(
353:             state, totalVaultSharesBefore, block.timestamp
354:         ).toUint128();
355:         state.lastAccumulatedTime = uint32(block.timestamp);
356:
357:         VaultStorage.getVaultRewardState()[index] = state;
358:     }
359:
360:     function _getAccumulatedRewardViaEmissionRate(
361:         VaultRewardState memory state,
362:         uint256 totalVaultSharesBefore,
363:         uint256 blockTime
364:     ) private pure returns (uint256) {
365:         // Short circuit the method with no emission rate
```

```
366:         if (state.emissionRatePerYear == 0) return
 ↪  state.accumulatedRewardPerVaultShare;
367:         require(0 < state.endTime);
368:         uint256 time = blockTime < state.endTime ? blockTime :
 ↪  state.endTime;
369:
370:         uint256 additionalIncentiveAccumulatedPerVaultShare;
371:         if (state.lastAccumulatedTime < time && 0 < totalVaultSharesBefore)
 ↪  {
372:             // NOTE: no underflow, checked in if statement
373:             uint256 timeSinceLastAccumulation = time -
 ↪  state.lastAccumulatedTime;
374:             // Precision here is:
375:             //  timeSinceLastAccumulation (SECONDS)
376:             //  emissionRatePerYear (REWARD_TOKEN_PRECISION)
377:             //  INTERNAL_TOKEN_PRECISION (1e8)
378:             // DIVIDE BY
379:             //  YEAR (SECONDS)
380:             //  INTERNAL_TOKEN_PRECISION (1e8)
381:             // => Precision = REWARD_TOKEN_PRECISION *
 ↪  INTERNAL_TOKEN_PRECISION / INTERNAL_TOKEN_PRECISION
382:             // => rewardTokenPrecision
383:             additionalIncentiveAccumulatedPerVaultShare =
384:                 (timeSinceLastAccumulation
385:                     * uint256(Constants.INTERNAL_TOKEN_PRECISION)
386:                     * state.emissionRatePerYear)
387:                 / (Constants.YEAR * totalVaultSharesBefore);
388:         }
389:
390:         return state.accumulatedRewardPerVaultShare +
 ↪  additionalIncentiveAccumulatedPerVaultShare;
391:     }
```

Impact

Loss of reward tokens, as shown in the scenarios above.

Code Snippet

https://github.com/sherlock-audit/2024-06-leveraged-vaults/blob/main/leveraged-vaults-private/contracts/vaults/common/VaultRewarderLib.sol#L174

https://github.com/sherlock-audit/2024-06-leveraged-vaults/blob/main/leveraged-vaults-private/contracts/vaults/common/VaultRewarderLib.sol#L340

https://github.com/sherlock-audit/2024-06-leveraged-vaults/blob/main/leveraged-vaults-private/contracts/vaults/common/VaultRewarderLib.sol#L373

Tool used

Manual Review

Recommendation

To reduce the impact of this issue, consider making the `claimRewardTokens` function permissioned and only allowing whitelisted bots to trigger it periodically.

Discussion

sherlock-admin3

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

0xmystery commented:

> Low/QA at most due to dust amount loss

xiaoming9090

Escalate.

This should be a valid issue. If Notional is deployed only on the Mainnet, this issue does not pose a significant risk. However, Notional is also deployed on L2 (Arbitrum), thus the risk cannot be ignored. As mentioned in the report, this attack is feasible due to low transaction fees on L2.

In addition, the loss is not a dust amount, as mentioned in the Judging comment. Under the right conditions (e.g., Vault with large TVL), and if malicious actors frequently perform griefing attacks against the reward features, the entire reward amount could be lost.

sherlock-admin3

> Escalate.
>
> This should be a valid issue. If Notional is deployed only on the Mainnet, this issue does not pose a significant risk. However, Notional is also deployed on L2 (Arbitrum), thus the risk cannot be ignored. As mentioned in the report, this attack is feasible due to low transaction fees on L2.
>
> In addition, the loss is not a dust amount, as mentioned in the Judging comment. Under the right conditions (e.g., Vault with large TVL), and if

> malicious actors frequently perform griefing attacks against the reward features, the entire reward amount could be lost.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

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

mystery0x

> Escalate.
>
> This should be a valid issue. If Notional is deployed only on the Mainnet, this issue does not pose a significant risk. However, Notional is also deployed on L2 (Arbitrum), thus the risk cannot be ignored. As mentioned in the report, this attack is feasible due to low transaction fees on L2.
>
> In addition, the loss is not a dust amount, as mentioned in the Judging comment. Under the right conditions (e.g., Vault with large TVL), and if malicious actors frequently perform griefing attacks against the reward features, the entire reward amount could be lost.

Will have sponsors look into this finding... but will `balanceAfter - balancesBefore[i]` not have the exploit circumvented when rolling over?

WangSecurity

Moreover, as I understand a completely possible scenario that there will be regular users claiming the rewards (e.g. Block 1 - Alice, block 2 - Bob, block 3 - Jack, etc.), so it even removes the need for the attack to be on L2 I believe.

Secondly, yes, the loss each time individually is very small, but if we take a broader picture, if the attacker repeats this attack for just one day, all the rewards for this day will be lost, which scales up, taking large TVL into account.

Planning to accept the escalation and validate the issue with high severity, since all the rewards can be lost. Yes, the attacker doesn't gain anything, but I believe there are no extensive constraints and a definite loss of funds.

WangSecurity

Are there additional duplicates except #90? @mystery0x

0502lian

I can’t agree that this issue is high, based on three points:

- (1) I disagree with the example in the report where the precision of the Reward Token is 6(both scenarios 1 and 2). Based on the protocols mentioned (arb,

cvx), it should be 18 and After Constants.INTERNAL_TOKEN_PRECISION is used by dev to expand to 1e8 before calculating shares. For example, if the Reward Token precision is 18, even if the reward per block is only one token, i.e., 1e18 arb, then to round to 0, the required VaultShare would be 1e18 * 1e8 (not the 10000000e8 in reports), which is almost impossible to achieve.

Therefore, for a Reward Token with a precision of 18, causing the rounding to zero error is almost impossible.

- (2) The attacker does not profit and actually incurs a gas loss.

I suggest considering the sponsor's opinion when judging whether this issue is high. @WangSecurity @T-Woodward @jeffywu

xiaoming9090

I can't agree that this issue is high, based on three points:

- (1) I disagree with the example in the report where the precision of the Reward Token is 6(both scenarios 1 and 2). Based on the protocols mentioned (arb, cvx), it should be 18 and After Constants.INTERNAL_TOKEN_PRECISION is used by dev to expand to 1e8 before calculating shares. For example, if the Reward Token precision is 18, even if the reward per block is only one token, i.e., 1e18 arb, then to round to 0, the required VaultShare would be 1e18 * 1e8 (not the 10000000e8 in reports), which is almost impossible to achieve. **Therefore, for a Reward Token with a precision of 18, causing the rounding to zero error is almost impossible.**
- (2) The attacker does not profit and actually incurs a gas loss.

I suggest considering the sponsor's opinion when judging whether this issue is high. @WangSecurity @T-Woodward @jeffywu

1) There is no part in the contest README stating that only ARB/CVX or reward tokens with 18 decimals will be accepted. Thus, it will be considered that reward tokens with 6 decimals (e.g., USDC, USDT) will also be accepted as reward tokens for the context of this audit contest. Also, the new contracts are meant to handle any reward tokens Notional might receive and would like to distribute to its users.
2) Gas in L2 is insignificant compared to the loss of the reward tokens.

jeffywu

Looks like this issue is also a duplicate:
https://github.com/sherlock-audit/2024-06-leveraged-vaults-judging/issues/115

Reward tokens with 6 decimals can be used so that part is valid.

I think there are some similar issues in severity where we would consider this valid and fix it, but in reality someone doing this seems unlikely without some sort of profit motivation. I'd consider this a valid medium severity issue.

novaman33

I also think medium is more appropriate as it is a costly attack with no incentive for the attacker. However from the readMe tokens with 6 decimals were not in scope in this audit so it might be invalid.

xiaoming9090

> I also think medium is more appropriate as it is a costly attack with no incentive for the attacker.

In L2 environment, the cost of attack is significantly lower than on Ethereum. Thus, as stated earlier, this type of griefing attack is only relevant in the L2 environment in which Notional resides. The maximum impact is that all reward tokens will be lost.

Note that all DOS/griefing issues, by nature, do not directly benefit the attackers but cause a loss of assets for the victim (users or protocols). However, these issues have been consistently judged as High in Sherlock as long as the issues demonstrate a definite loss of assets to the victims with no extensive constraints.

> However from the readMe tokens with 6 decimals were not in scope in this audit so it might be invalid.

The contest's README does not state that tokens with 6 decimals are not in scope. As a whole, Notional protocol only explicitly disallows tokens that are less than 6 decimals or more than 18 decimals.

0502lian

I agree with the sponsor's opinion.

Assuming on Arbitrum, with an average of 0.25s per block, considering the complexity of the claimRewardTokens function, the gas cost for each transaction is reasonably estimated to be 0.01 USD - 0.1 USD. The estimated number of transactions per day is 4 * 3600 * 24 = 345600. If it continues for one day, the attacker's cost is: 345600 * (0.01 USD ~ 0.1 USD) = 3456 USD ~ 34560 USD.

And the attacker's profit is 0.

Considering that Notional has many staking vaults, assuming the value of one vault is 1M USD and the annual reward rate is 10%, which is a reasonable estimate, then the daily loss of funds from the vault is: 1000000 * 10% / 365 = 273.97 USD.

notN000B

```
additionalIncentiveAccumulatedPerVaultShare  = (timeSinceLastAccumulation * 1e8
↪  * emissionRate) / (31536000 * totalVaultSharesBefore)
```

The `additionalIncentiveAccumulatedPerVaultShare` will be zero if `(t * 1e8 * rate) < 31536000 * totalSharesBefore`, which is achievable.

For instance, with `totalSharesBefore` being 1,000,000,000 shares, this equates to `1,000,000,000 * 1e8`. To prevent reward accumulation, an attacker needs to ensure `t * rate *1e8 < 31536000 * 1,000,000,000 * 1e8`. Even with a high emission rate of 1e6 reward tokens (which is very unlikely), the attacker would need to call functions just before `t < 31536 seconds (approximately 8.76 minutes)`.

With such a high emission rate, there would be roughly 9 minutes to DOS the function, even on the mainnet. As totalShares increase and the emission rate decreases(Which will be the case in reality), DOSing the function becomes even easier

Issuse should be HIGH. This was my first contest I can't escalate my issue but This issue seems sames as mine
https://github.com/sherlock-audit/2024-06-leveraged-vaults-judging/issues/115

0502lian

For instance, with totalSharesBefore being 1,000,000,000 shares, this equates to 1,000,000,000 * 1e8. ---------That means you need 1 billion ETH in a vault(eg PendlePTEtherFiVault). The total circulating supply of Ethereum (ETH) is currently approximately 0.12 billion ETH

WangSecurity

Firstly, about the reward tokens. The reason why they weren't specified in the README is because they're not set by the admins of Notional. The reward tokens are the tokens that other protocols, which Notional integrates with, use to pay out the rewards/grants. Hence, I believe it's enough contextual evidence that the protocol indeed needs to work with any type of reward tokens.

Secondly, I see that there is no impact for the attacker and on mainnet the cost is significant. As I've said in my previous comment, it can happen without a malicious intent, when different users will claim rewards in every block. That is why I believe it qualifies for High severity:

> Definite loss of funds without (extensive) limitations of external conditions

Planning to accept the escalation and validate the issue with high severity. Duplicates are #90. #115 is not a duplicate, check my comment under it.

WangSecurity

Result: High Has duplicates

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- xiaoming9090: accepted

jeffywu

Marking this as won't fix, I don't see a reason why this would happen. If it does we would perhaps consider guarding this claim method.

Issue H-7: Users can deny the vault from claiming reward tokens

Source: https://github.com/sherlock-audit/2024-06-leveraged-vaults-judging/issues/63

The protocol has acknowledged this issue.

Found by

DenTonylifer, xiaoming90

Summary

Users can deny the vault from claiming reward tokens by front-running the `_claimVaultRewards` function.

Vulnerability Detail

The `_claimVaultRewards` function will call the `_executeClaim` function to retrieve the reward tokens from the external protocols (e.g., Convex or Aura). The reward tokens will be transferred directly to the vault contract. The vault computes the number of reward tokens claimed by taking the difference of the before and after balance.

https://github.com/sherlock-audit/2024-06-leveraged-vaults/blob/main/leveraged-vaults-private/contracts/vaults/common/VaultRewarderLib.sol#L174

```
File: VaultRewarderLib.sol
148:     function _claimVaultRewards(
..SNIP..
152:     ) internal {
153:         uint256[] memory balancesBefore = new uint256[](state.length);
154:         // Run a generic call against the reward pool and then do a balance
155:         // before and after check.
156:         for (uint256 i; i < state.length; i++) {
157:             // Presumes that ETH will never be given out as a reward token.
158:             balancesBefore[i] =
↪   IERC20(state[i].rewardToken).balanceOf(address(this));
159:         }
160:
161:         _executeClaim(rewardPool);
..SNIP..
168:         for (uint256 i; i < state.length; i++) {
169:             uint256 balanceAfter =
↪   IERC20(state[i].rewardToken).balanceOf(address(this));
```

```
170:                 _accumulateSecondaryRewardViaClaim(
171:                     i,
172:                     state[i],
173:                     // balanceAfter should never be less than balanceBefore
174:                     balanceAfter - balancesBefore[i],
175:                     totalVaultSharesBefore
176:                 );
177:             }
178:         }
```

https://github.com/sherlock-audit/2024-06-leveraged-vaults/blob/main/leveraged-vaults-private/contracts/vaults/common/VaultRewarderLib.sol#L181

```
File: VaultRewarderLib.sol
181:     function _executeClaim(RewardPoolStorage memory r) internal {
182:         if (r.poolType == RewardPoolType.AURA) {
183:             require(IAuraRewardPool(r.rewardPool).getReward(address(this),
↪  true));
184:         } else if (r.poolType == RewardPoolType.CONVEX_MAINNET) {
185:
↪  require(IConvexRewardPool(r.rewardPool).getReward(address(this), true));
186:         } else if (r.poolType == RewardPoolType.CONVEX_ARBITRUM) {
187:
↪  IConvexRewardPoolArbitrum(r.rewardPool).getReward(address(this));
188:         } else {
189:             revert();
190:         }
191:     }
```

However, the `getReward` function of the external protocols can be executed by anyone. Refer to Appendix A for the actual implementation of the `getReward` function.

As a result, malicious users can front-run the `_claimVaultRewards` transaction and trigger the `getReward` function of the external protocols directly, resulting in the reward tokens to be sent to the vault before the `_claimVaultRewards` is executed.

When the `_claimVaultRewards` function is executed, the before/after snapshot will ultimately claim the zero amount. The code `balanceAfter - balancesBefore[i]` at Line 174 above will always produce zero if the call to `_claimVaultRewards` is front-run.

As a result, reward tokens are forever lost in the contract.

Impact

High as this issue is the same this issue in the past Notional V3 contest.

Loss of assets as the reward tokens intended for Notional and its users are lost.

Code Snippet

https://github.com/sherlock-audit/2024-06-leveraged-vaults/blob/main/leveraged-vaults-private/contracts/vaults/common/VaultRewarderLib.sol#L174

https://github.com/sherlock-audit/2024-06-leveraged-vaults/blob/main/leveraged-vaults-private/contracts/vaults/common/VaultRewarderLib.sol#L181

Tool used

Manual Review

Recommendation

Consider using the entire balance instead of the difference between before and after balances.

Appendix A - `getReward` of Convex and Aura's reward pool contract

Aura's Reward Pool on Mainnet

https://etherscan.io/address/0x44D8FaB7CD8b7877D5F79974c2F501aF6E65AbBA#code#L980

```
 function getReward(address _account, bool _claimExtras) public
↪   updateReward(_account) returns(bool){
    uint256 reward = earned(_account);
    if (reward > 0) {
        rewards[_account] = 0;
        rewardToken.safeTransfer(_account, reward);
        IDeposit(operator).rewardClaimed(pid, _account, reward);
        emit RewardPaid(_account, reward);
    }

    //also get rewards from linked rewards
    if(_claimExtras){
        for(uint i=0; i < extraRewards.length; i++){
            IRewards(extraRewards[i]).getReward(_account);
        }
    }
    return true;
}
```

Aura's Reward Pool on Arbitrum

https://arbiscan.io/address/0x17F061160A167d4303d5a6D32C2AC693AC87375b#code#F15#L296

```
/**
   * @dev Gives a staker their rewards, with the option of claiming extra rewards
   * @param _account     Account for which to claim
   * @param _claimExtras Get the child rewards too?
   */
  function getReward(address _account, bool _claimExtras) public
↪  updateReward(_account) returns(bool){
      uint256 reward = earned(_account);
      if (reward > 0) {
          rewards[_account] = 0;
          rewardToken.safeTransfer(_account, reward);
          IDeposit(operator).rewardClaimed(pid, _account, reward);
          emit RewardPaid(_account, reward);
      }

      //also get rewards from linked rewards
      if(_claimExtras){
          for(uint i=0; i < extraRewards.length; i++){
              IRewards(extraRewards[i]).getReward(_account);
          }
      }
      return true;
  }
```

Convex's Reward Pool on Arbitrum

https://arbiscan.io/address/0x93729702Bf9E1687Ae2124e191B8fFbcC0C8A0B0#code#F1#L337

```
//claim reward for given account (unguarded)
function getReward(address _account) external {
    //check if there is a redirect address
    if(rewardRedirect[_account] != address(0)){
        _checkpoint(_account, rewardRedirect[_account]);
    }else{
        //claim directly in checkpoint logic to save a bit of gas
        _checkpoint(_account, _account);
    }
}
```

Convex for Mainnet https://etherscan.io/address/0xD1DdB0a0815fD28932fBb194C84003683AF8a824#code#L980

```
function getReward(address _account, bool _claimExtras) public
↪  updateReward(_account) returns(bool){
    uint256 reward = earned(_account);
    if (reward > 0) {
        rewards[_account] = 0;
        rewardToken.safeTransfer(_account, reward);
        IDeposit(operator).rewardClaimed(pid, _account, reward);
        emit RewardPaid(_account, reward);
    }

    //also get rewards from linked rewards
    if(_claimExtras){
        for(uint i=0; i < extraRewards.length; i++){
            IRewards(extraRewards[i]).getReward(_account);
        }
    }
    return true;
}
```

Discussion

DenTonylifer

Disagree with severity

This should be high issue according to Sherlock rules:

- there is no limitations of external conditions - attack can be done for free by anyone and anytime
- loss of rewards for users even up to 100% of reward amount
- loss for protocol as there is no ability to resque stuck tokens

Also look at similar high-severity issues from past audits:
https://github.com/sherlock-audit/2023-06-tokemak-judging/issues/738
https://github.com/sherlock-audit/2023-03-notional-judging/issues/168

xiaoming9090

Escalate.

This issue should be a High instead of Medium.

Malicious users can easily cause Notional and its users to lose rewards by triggering the `getReward` function of the external protocols as described in my report.

Note that this issue is exactly the same as the issue (https://github.com/sherlock-audit/2023-03-notional-judging/issues/200) found in the past Notional contest,

which was judged as a High finding. Thus, it should be consistently applied here.

sherlock-admin3

> Escalate.
>
> This issue should be a High instead of Medium.
>
> Malicious users can easily cause Notional and its users to lose rewards by triggering the `getReward` function of the external protocols as described in my report.
>
> Note that this issue is exactly the same as the issue (https://github.com/sherlock-audit/2023-03-notional-judging/issues/200) found in the past Notional contest, which was judged as a High finding. Thus, it should be consistently applied here.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

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

WangSecurity

I agree the severity should be high, don't see any extensive constraints here, even though the attack is griefing. Planning to accept the escalation and upgrade the severity.

WangSecurity

Result: High Has duplicates

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- xiaoming9090: accepted

jeffywu

Similar to #61, won't fix unless we ever see this becoming an issue. I think this is more of a hypothetical attack vector than a real one. If we had to fix this we would have to create some sort of manual override to the reward calculation.

Issue H-8: Malicious users can steal reward tokens via re-entrancy attack

Source: https://github.com/sherlock-audit/2024-06-leveraged-vaults-judging/issues/64

Found by

xiaoming90

Summary

Malicious users can steal reward tokens via re-entrancy attack.

Vulnerability Detail

During the redemption of vault shares, the `_updateAccountRewards` function will be triggered at the end of the function to update the account rewards.

https://github.com/sherlock-audit/2024-06-leveraged-vaults/blob/main/leveraged-vaults-private/contracts/vaults/common/SingleSidedLPVaultBase.sol#L282

```
File: SingleSidedLPVaultBase.sol
282:     function _redeemFromNotional(
283:         address account, uint256 vaultShares, uint256 /* maturity */, bytes
 ↪  calldata data
284:     ) internal override virtual whenNotLocked returns (uint256
 ↪  finalPrimaryBalance) {
..SNIP..
316:         _updateAccountRewards({
317:             account: account,
318:             vaultShares: vaultShares,
319:             totalVaultSharesBefore: totalVaultSharesBefore,
320:             isMint: false
321:         });
322:     }
```

Assume that at T1

- Bob has 100 vault shares
- Current `rewardsPerVaultShare` is 1.0
- Bob's debt (`VaultStorage.getAccountRewardDebt()[rewardToken][Bob]`) is 100 (`100 shares * 1.0`)

Assume that at T2:

- Bob attempts to redeem 90 vault shares
- Current `rewardsPerVaultShare` is 2.0

When Line 211 below is executed, the `vaultSharesBefore` will be set to 100 vault shares. The `_claimAccountRewards` function will be executed in Line 212, and it will execute the `_claimRewardToken` function internally.

https://github.com/sherlock-audit/2024-06-leveraged-vaults/blob/main/leveraged-vaults-private/contracts/vaults/common/VaultRewarderLib.sol#L211

```
File: VaultRewarderLib.sol
202:     /// @notice Called by the vault during enter and exit vault to update
 ↪  the account reward claims.
203:     function updateAccountRewards(
204:         address account,
205:         uint256 vaultShares,
206:         uint256 totalVaultSharesBefore,
207:         bool isMint
208:     ) external {
209:         // Can only be called via enter or exit vault
210:         require(msg.sender == address(Deployments.NOTIONAL));
211:         uint256 vaultSharesBefore = _getVaultSharesBefore(account);
212:         _claimAccountRewards(
213:             account,
214:             totalVaultSharesBefore,
215:             vaultSharesBefore,
216:             isMint ? vaultSharesBefore + vaultShares : vaultSharesBefore -
 ↪  vaultShares
217:         );
218:     }
```

Within the `_claimRewardToken` function, the `_getRewardsToClaim` function will be executed to compute the number of reward tokens that Bob is entitled to. Based on the formula within the `_getRewardsToClaim` function, Bob is entitled 100 reward tokens.

```
rewardToClaim = (vaultSharesBefore * rewardsPerVaultShare) - Bob's debt
rewardToClaim = (100 shares * 2.0) - 100 = 100
```

In Line 306 below, Bob's debt (`VaultStorage.getAccountRewardDebt()[rewardToken][Bob]`) will be updated to 20 (`vaultSharesAfter * rewardsPerVaultShare = 10 shares * 2.0 = 20`). Note that `vaultSharesAfter` is 10 shares because Bob withdraws 90 shares from his initial 100 shares.

In Line 316 below, 100 reward tokens will be transferred to Bob.

https://github.com/sherlock-audit/2024-06-leveraged-vaults/blob/main/leveraged-vaults-private/contracts/vaults/common/VaultRewarderLib.sol#L316

```
File: VaultRewarderLib.sol
295:     function _claimRewardToken(
296:         address rewardToken,
297:         address account,
298:         uint256 vaultSharesBefore,
299:         uint256 vaultSharesAfter,
300:         uint256 rewardsPerVaultShare
301:     ) internal returns (uint256 rewardToClaim) {
302:         rewardToClaim = _getRewardsToClaim(
303:             rewardToken, account, vaultSharesBefore, rewardsPerVaultShare
304:         );
305: 
306:         VaultStorage.getAccountRewardDebt()[rewardToken][account] = (
307:             (vaultSharesAfter * rewardsPerVaultShare) /
308:                 uint256(Constants.INTERNAL_TOKEN_PRECISION)
309:         );
310: 
311:         if (0 < rewardToClaim) {
312:             // Ignore transfer errors here so that any strange failures
 ↪  here do not
313:             // prevent normal vault operations from working. Failures may
 ↪  include a
314:             // lack of balances or some sort of blacklist that prevents an
 ↪  account
315:             // from receiving tokens.
316:             try IEIP20NonStandard(rewardToken).transfer(account,
 ↪  rewardToClaim) {
317:                 bool success = TokenUtils.checkReturnCode();
318:                 if (success) {
319:                     emit VaultRewardTransfer(rewardToken, account,
 ↪  rewardToClaim);
320:                 } else {
321:                     emit VaultRewardTransfer(rewardToken, account, 0);
322:                 }
323:             // Emits zero tokens transferred if the transfer fails.
324:             } catch {
325:                 emit VaultRewardTransfer(rewardToken, account, 0);
326:             }
327:         }
```

Assume that the reward token contains a hook or callback. As a result, the control will be passed back to Bob. Note that there are no restrictions on the type of reward tokens in the context of this audit.

Bob can re-enter the vault and execute the `claimAccountRewards` function, which is not guarded against re-entrancy. When Line 197 is executed, the `totalVaultSharesBefore` will still remain 100 vault shares because the execution `_redeemFromNotional` function has not been completed yet. Thus, the number of vault shares has not been updated on Notional side yet. The `_claimRewardToken` function, followed by `_getRewardsToClaim` will be executed again internally.

https://github.com/sherlock-audit/2024-06-leveraged-vaults/blob/main/leveraged-vaults-private/contracts/vaults/common/VaultRewarderLib.sol#L197

```
File: VaultRewarderLib.sol
193:     /// @notice Callable by an account to claim their own rewards, we know
 ↪  that the vault shares have
194:     /// not changed in this transaction because the contract has not been
 ↪  called by Notional
195:     function claimAccountRewards(address account) external override {
196:         require(msg.sender == account);
197:         uint256 totalVaultSharesBefore =
 ↪  VaultStorage.getStrategyVaultState().totalVaultSharesGlobal;
198:         uint256 vaultSharesBefore = _getVaultSharesBefore(account);
199:         _claimAccountRewards(account, totalVaultSharesBefore,
 ↪  vaultSharesBefore, vaultSharesBefore);
200:     }
```

Based on the formula within the `_getRewardsToClaim` function, Bob is entitled 180 reward tokens.

```
rewardToClaim = (vaultSharesBefore * rewardsPerVaultShare) - Bob's debt
rewardToClaim = (100 shares * 2.0) - 20 = 180
```

The vault will transfer an additional 180 reward tokens to Bob again, which is incorrect. In this case, Bob has stolen 180 reward tokens from the vault and other shareholders.

Instance 2
https://github.com/sherlock-audit/2024-06-leveraged-vaults/blob/main/leveraged-vaults-private/contracts/vaults/common/WithdrawRequestBase.sol#L110

```
File: WithdrawRequestBase.sol
109:     function _initiateWithdraw(address account, bool isForced) internal {
110:         uint256 vaultShares = Deployments.NOTIONAL.getVaultAccount(account,
 ↪  address(this)).vaultShares;
111:         require(0 < vaultShares);
```

Attackers can also call this function. Because Line 110 will still read the outdated

vault share info, it will be the higher than expected number.

Impact

Reward tokens can be stolen by malicious users.

Code Snippet

https://github.com/sherlock-audit/2024-06-leveraged-vaults/blob/main/leveraged-vaults-private/contracts/vaults/common/SingleSidedLPVaultBase.sol#L282

https://github.com/sherlock-audit/2024-06-leveraged-vaults/blob/main/leveraged-vaults-private/contracts/vaults/common/VaultRewarderLib.sol#L211

https://github.com/sherlock-audit/2024-06-leveraged-vaults/blob/main/leveraged-vaults-private/contracts/vaults/common/VaultRewarderLib.sol#L316

https://github.com/sherlock-audit/2024-06-leveraged-vaults/blob/main/leveraged-vaults-private/contracts/vaults/common/VaultRewarderLib.sol#L197

https://github.com/sherlock-audit/2024-06-leveraged-vaults/blob/main/leveraged-vaults-private/contracts/vaults/common/WithdrawRequestBase.sol#L110

Tool used

Manual Review

Recommendation

Add re-entrancy guard on the `claimAccountRewards` function to prevent anyone from re-entering the vault under any circumstance.

Discussion

sherlock-admin2

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

0xmystery commented:

> Devoid of coded POC to substantiate exploit

xiaoming9090

Escalate.

This issue should be a valid High.

The lead judge mentioned that the issue was "Devoid of coded POC to substantiate exploit" and marked it as invalid. However, the POC in the report is already

sufficient to demonstrate that the vulnerability mentioned in the report could lead to a loss of assets. Thus, it should be valid.

sherlock-admin3

> Escalate.
>
> This issue should be a valid High.
>
> The lead judge mentioned that the issue was "Devoid of coded POC to substantiate exploit" and marked it as invalid. However, the POC in the report is already sufficient to demonstrate that the vulnerability mentioned in the report could lead to a loss of assets. Thus, it should be valid.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

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

mystery0x

> Escalate.
>
> This issue should be a valid High.
>
> The lead judge mentioned that the issue was "Devoid of coded POC to substantiate exploit" and marked it as invalid. However, the POC in the report is already sufficient to demonstrate that the vulnerability mentioned in the report could lead to a loss of assets. Thus, it should be valid.

Will have the sponsors look into this finding to decide whether or not their reward tokens will have hook entailed. It seems to me the sponsors would have specified any of these weird tokens requiring non-reentrant visibility in the 2nd question of the contest readme `Details` if they had been adopted.

WangSecurity

The protocol didn't specify which tokens will be used as rewards, so we have to assume only the standard tokens without any weird traits will be used. Hence, we should assume tokens with hooks or callbacks allowing for reentrancy won't be used. Planning to reject the escalation and leave the issue as it is.

xiaoming9090

> The protocol didn't specify which tokens will be used as rewards, so we have to assume only the standard tokens without any weird traits will be used. Hence, we should assume tokens with hooks or callbacks allowing

for reentrancy won't be used. Planning to reject the escalation and leave the issue as it is.

@WangSecurity The contracts are meant to handle tokens that Notional protocol will receive from other protocols or projects (e.g., grants). Thus, it is not possible for Notional to predict what kind of tokens they will receive in the future. To be on the safe side, we should assume the worst-case scenario where it might be possible that some of the reward tokens might contain hook or callback, and necessary measures should be implemented to guard against potential re-entrancy attack.

@jeffywu You might want to have a look at this. Thanks.

jeffywu

I agree with @xiaoming9090's assessment, this is a valid issue. Some reward tokens may hold callback hooks that we are unaware of.

0502lian

In my opinion, even if there is reentrancy, there won't be any loss. If my understanding is incorrect, please correct me. @xiaoming9090 @jeffywu Thank you!

The loss due to reentrancy is based on the description by @xiaoming9090 , 'When Line 197 is executed, the totalVaultSharesBefore will still remain 100 vault shares because the execution of the _redeemFromNotional function has not been completed yet. Thus, the number of vault shares has not been updated on the Notional side yet.'

However, after research, I found that totalVaultSharesBefore is updated first and then _redeemFromNotional() is called.

https://github.com/notional-finance/contracts-v3/blob/b664c157051d256ce583ba19da3e26c6cf5061ac/contracts/external/actions/VaultAccountAction.sol#L215

```
function exitVault(
        address account,
        address vault,
        address receiver,
        uint256 vaultSharesToRedeem,
        uint256 lendAmount,
        uint32 minLendRate,
        bytes calldata exitVaultData
    ) external payable override nonReentrant returns (uint256
↪   underlyingToReceiver) {
       //---------skip---------
          //update here
    @>>    vaultState.exitMaturity(vaultAccount, vaultConfig,
↪   vaultSharesToRedeem);
```

```
        if (vaultAccount.tempCashBalance > 0) {
            Emitter.emitTransferPrimeCash(
                vault, receiver, vaultConfig.borrowCurrencyId,
↪   vaultAccount.tempCashBalance
            );

            underlyingToReceiver = VaultConfiguration.transferFromNotional(
                receiver, vaultConfig.borrowCurrencyId,
↪   vaultAccount.tempCashBalance, vaultConfig.primeRate, false
            );

            vaultAccount.tempCashBalance = 0;
        }

        // If insufficient strategy tokens are redeemed (or if it is set to
↪   zero), then
        // redeem with debt repayment will recover the repayment from the
↪   account's wallet
        // directly.
         // call _redeemFromNotional() in redeemWithDebtRepayment()
    @>>  underlyingToReceiver =
↪   underlyingToReceiver.add(vaultConfig.redeemWithDebtRepayment(
            vaultAccount, receiver, vaultSharesToRedeem, exitVaultData
        ));

        // Set the vault state after redemption completes
        vaultState.setVaultState(vaultConfig);

       //---skip----------
    }
```

update storage

```
function exitMaturity(
        VaultState memory vaultState,
        VaultAccount memory vaultAccount,
        VaultConfig memory vaultConfig,
        uint256 vaultSharesToRedeem
    ) internal {
        require(vaultAccount.maturity == vaultState.maturity);
        mapping(address => mapping(uint256 => VaultStateStorage)) storage store
↪   = LibStorage.getVaultState();
        VaultStateStorage storage s =
↪   store[vaultConfig.vault][vaultState.maturity];
```

```
        // Update the values in memory
        vaultState.totalVaultShares =
↪   vaultState.totalVaultShares.sub(vaultSharesToRedeem);
        vaultAccount.vaultShares =
↪   vaultAccount.vaultShares.sub(vaultSharesToRedeem);

        // Update the global value in storage
@>>         s.totalVaultShares = vaultState.totalVaultShares.toUint80();
    }
```

redeemWithDebtRepayment calls _redeem, then calls redeemFromNotional, then calls _redeemFromNotional

```
/// @notice Redeems without any debt repayment and sends profits back to the
↪   receiver
    function redeemWithDebtRepayment(
        VaultConfig memory vaultConfig,
        VaultAccount memory vaultAccount,
        address receiver,
        uint256 vaultShares,
        bytes calldata data
    ) internal returns (uint256 underlyingToReceiver) {
        uint256 amountTransferred;
        uint256 underlyingExternalToRepay;
        {
            //-----------skip-------------
            (amountTransferred, underlyingToReceiver, /* primeCashRaised */) =
↪   _redeem(
                vaultConfig,
                underlyingToken,
                vaultAccount.account,
                receiver,
                vaultShares,
                vaultAccount.maturity,
                underlyingExternalToRepay,
                data
            );
        }

      // ------skip--------
    }
```

```
function _redeem(
        VaultConfig memory vaultConfig,
```

```
        Token memory underlyingToken,
        address account,
        address receiver,
        uint256 vaultShares,
        uint256 maturity,
        uint256 underlyingExternalToRepay,
        bytes calldata data
    ) private returns (
        uint256 amountTransferred,
        uint256 underlyingToReceiver,
        int256 primeCashRaised
    ) {
            //-----skip-----
        {
            uint256 balanceBefore = underlyingToken.balanceOf(address(this));
   @>>          underlyingToReceiver =
↪   IStrategyVault(vaultConfig.vault).redeemFromNotional(
                account, receiver, vaultShares, maturity,
↪   underlyingExternalToRepay, data
            );
    }

//--------skip------
```

Because all data is updated before token transfer, There is nothing can be done in re-entrancy.

xiaoming9090

During the re-entrancy, Bob's vault shares should have decreased from 100 vault shares to 10 vault shares since he withdraws 90 vault shares. However, it remains at 100 vault shares. This is because vault account shares are not updated in storage until the vault complete its exit here. Thus, the reward calculation in the re-entrancy will be incorrect.

There is a minor typo in this paragraph. Refer to the update below. Apart from that, the rest, including the math calculation, is correct.

```
- Line 197 is executed, the totalVaultSharesBefore will still remain 100 vault
↪   shares because the execution _redeemFromNotional function has not been
↪   completed yet.
+ Line 198 is executed, the vaultSharesBefore will still remain 100 vault shares
↪   because the execution _redeemFromNotional function has not been completed
↪   yet.
```

0502lian

> During the re-entrancy, Bob's vault shares should have decreased from 100 vault shares to 10 vault shares since he withdraws 90 vault shares. However, it remains at 100 vault shares. This is because vault account shares are not updated in storage until the vault complete its exit here. Thus, the reward calculation in the re-entrancy will be incorrect.
>
> There is a minor typo in this paragraph. Refer to the update below. Apart from that, the rest, including the math calculation, is correct.

```
- Line 197 is executed, the totalVaultSharesBefore will still remain 100
↪  vault shares because the execution _redeemFromNotional function has not
↪  been completed yet.
+ Line 198 is executed, the vaultSharesBefore will still remain 100 vault
↪  shares because the execution _redeemFromNotional function has not been
↪  completed yet.
```

Then I think you are right. User's vault account shares indeed updates after _redeemFromNotional(). If the reward token is ERC777 , it could be a re-entrancy attack.

novaman33

How are these in scope since the answer of the question in readMe is: ‘ If you are integrating tokens, are you allowing only whitelisted tokens to work with the codebase or any complying with the standard? Are they assumed to have certain properties, e.g. be non-reentrant? Are there any types of weird tokens you want to integrate?

EtherFi: weETH, eETH Ethena: USDe, sUSDe Pendle: PT tokens Kelp: rsETH ‘

xiaoming9090

> How are these in scope since the answer of the question in readMe is: ‘ If you are integrating tokens, are you allowing only whitelisted tokens to work with the codebase or any complying with the standard? Are they assumed to have certain properties, e.g. be non-reentrant? Are there any types of weird tokens you want to integrate?
>
> EtherFi: weETH, eETH Ethena: USDe, sUSDe Pendle: PT tokens Kelp: rsETH ‘

Note that there are two areas where ERC20 tokens will be used:

1) Vault's assets
2) Reward tokens

The above-listed tokens (weETH, USDe, sUSDe, rsETH) are the vault's asset tokens. Anyone who reviews the vault code will clearly understand it. Thus, it is

understandable from the sponsor's point of view that this information refers only to the vault's asset tokens, and not the reward tokens.

As mentioned in my earlier comments, the reward-related contracts meant to handle tokens that Notional protocol will receive from other protocols or projects (e.g., grants). Thus, it is not possible for Notional to predict what kind of tokens they will receive in the future. Thus, it makes sense for the protocol not to narrow down the reward tokens accepted at this point.

In addition, in the sponsor's earlier comment, they already shared the view that some reward tokens might contain callback hooks that they are unaware of.

novaman33

The question is regarding any strange ERC20 tokens. Their answer is a list of tokens that does include some rebasing tokens. I agree that there are two areas where ERC20 tokens are used, but I do not agree that by any means this means that all the other tokens are supported(from this answer). Given Sherlock's hierarchy of truth, both this and #61 should be invalid, as sponsors comments in judging cannot change the scope.

WangSecurity

About the reward tokens. The reason why they weren't specified in the README is because they're not set by the admins of Notional. The reward tokens are the tokens that other protocols, which Notional integrates with, use to pay out the rewards/grants. Hence, I believe it's enough contextual evidence that the protocol indeed needs to work with any type of reward tokens.

Therefore, I believe this issue is valid. Even though it's only possible with tokens with hooks, the report shows how the attacker gets almost 200% more shares than they should've got. Hence, I believe high severity is appropriate, planning to accept the escalation.

WangSecurity

Result: High Unique

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- xiaoming9090: accepted

WangSecurity

@xiaoming9090 @mystery0x @brakeless-wtp are there any duplicates?

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/notional-finance/leveraged-vaults/pull/96

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue H-9: Wrong decimal precision resulted in the price being inflated

Source: https://github.com/sherlock-audit/2024-06-leveraged-vaults-judging/issues/66

Found by

nirohgo, xiaoming90

Summary

The wrong decimal precision inflated the price returned from the oracle. As a result, the account's collateral will be overinflated, allowing malicious users to borrow significantly more than the actual collateral value, stealing assets from the protocol.

Vulnerability Detail

When Notional's `PendlePTOracle` is deployed, the `ptDecimals` is set to the decimals of the PT token, as shown below.

https://github.com/sherlock-audit/2024-06-leveraged-vaults/blob/main/leveraged-vaults-private/contracts/oracles/PendlePTOracle.sol#L58

```
File: PendlePTOracle.sol
32:     constructor (
..SNIP..
50:         uint8 _baseDecimals = baseToUSDOracle_.decimals();
51:         (/* */, address pt, /* */) = IPMarket(pendleMarket_).readTokens();
52:         uint8 _ptDecimals = IERC20(pt).decimals();
..SNIP..
57:         baseToUSDDecimals = int256(10**_baseDecimals);
58:         ptDecimals = int256(10**_ptDecimals);
```

The `_getPTRate` function below will return:

- If `useSyOracleRate` is true, the Pendle's `getPtToSyRate` function will be called to return how many SY tokens one unit of PT is worth
- If `useSyOracleRate` is false, the Pendle's `getPtToAssetRate` function will be called to return how many underlying asset tokens one unit of PT is worth

https://github.com/sherlock-audit/2024-06-leveraged-vaults/blob/main/leveraged-vaults-private/contracts/oracles/PendlePTOracle.sol#L85

```
File: PendlePTOracle.sol
85:     function _getPTRate() internal view returns (int256) {
86:         uint256 ptRate = useSyOracleRate ?
87:             Deployments.PENDLE_ORACLE.getPtToSyRate(pendleMarket,
 ↪  twapDuration) :
88:             Deployments.PENDLE_ORACLE.getPtToAssetRate(pendleMarket,
 ↪  twapDuration);
89:         return ptRate.toInt();
90:     }
```

Using PT fUSDC 26DEC2024 as an example to illustrate the issue. Note that this issue will also occur on other PTs due to wrong math.

Assume that the `PendlePTOracle` provides the price of PT fUSDC 26DEC2024. For PT fUSDC 26DEC2024, the details are as follows:

> YT fUSDC 26DEC2024 (YT-fUSDC-...) = 6 decimals
>
> PT fUSDC 26DEC2024 (PT-fUSDC-...) = 6 decimals
>
> SY fUSDC (SY-fUSDC) = 8 decimals
>
> Underlying Asset = Flux USDC (fUSDC) = 8 decimals
>
> Market = 0xcb71c2a73fd7588e1599df90b88de2316585a860
>
> Pendle's Market Page: https://app.pendle.finance/trade/markets/0xcb71c2a73fd7588e1599df90b88de2316585a860/swap?view=pt&chain=ethereum&py=output

In this case, the `ptDecimals` will be 1e6. The `rateDecimals` is always hardcoded to `1e18`.

Assume that the `baseToUSD` provides the price of fUSDC in terms of US dollars, and `baseToUSDDecimals` is 1e8. The price returned is `99990557`, close to 1 US Dollar (USD).

Assume that `useSyOracleRate` is set to `False`, the Pendle's `getPtToAssetRate` function will be called, and the price (`ptRate`) returned will be `978197897539187120`, as shown below.

https://etherscan.io/address/0x9a9fa8338dd5e5b2188006f1cd2ef26d921650c2#readProxyContract

The above-returned price (`978197897539187120`) is in 18 decimals. This means 1 PT is worth around 1 fUSDC (`978197897539187120 / 1e18`), which makes sense.

However, based on the formula at Line 117 below, the price of the PT fUSDC 26DEC2024 will be `9.781055263e29`, which is incorrect and is an extremely large

number (inflated by 12 orders of magnitude). This is far from the intended price of PT fUSDC 26DEC2024, which should hover around 1 USD.

```
answer = (ptRate * baseToUSD * rateDecimals) /(baseToUSDDecimals * ptDecimals)
answer = (978197897539187120 * baseToUSD * rateDecimals) /(baseToUSDDecimals *
↪  1e6)
answer = (978197897539187120 * 99990557 * rateDecimals) /(1e8 * 1e6)
answer = (978197897539187120 * 99990557 * 1e18) /(1e8 * 1e6)
answer = 9.781055263e29
```

https://github.com/sherlock-audit/2024-06-leveraged-vaults/blob/main/leveraged-vaults-private/contracts/oracles/PendlePTOracle.sol#L117

```
File: PendlePTOracle.sol
092:     function _calculateBaseToQuote() internal view returns (
093:         uint80 roundId,
094:         int256 answer,
095:         uint256 startedAt,
096:         uint256 updatedAt,
097:         uint80 answeredInRound
098:     ) {
099:         _checkSequencer();
100:
101:         int256 baseToUSD;
102:         (
103:             roundId,
104:             baseToUSD,
105:             startedAt,
106:             updatedAt,
107:             answeredInRound
108:         ) = baseToUSDOracle.latestRoundData();
109:         require(baseToUSD > 0, "Chainlink Rate Error");
110:         // Overflow and div by zero not possible
111:         if (invertBase) baseToUSD = (baseToUSDDecimals * baseToUSDDecimals)
↪  / baseToUSD;
112:
113:         // Past expiration, hardcode the PT oracle price to 1. It is no
↪  longer tradable and
114:         // is worth 1 unit of the underlying SY at expiration.
115:         int256 ptRate = expiry <= block.timestamp ? ptDecimals :
↪  _getPTRate();
116:
117:         answer = (ptRate * baseToUSD * rateDecimals) /
118:             (baseToUSDDecimals * ptDecimals);
119:     }
```

The root cause is that the code wrongly assumes that the price returned from Pendle's `getPtToAssetRate` function is denominated in PT's decimals. However, it is, in fact, denominated in 18 decimals. Thus, when the PT is not 18 decimals, such as the one in our example (6 decimals), the price returned from Notional's `PendlePTOracle` will be overly inflated.

Impact

The price returned from the Notional's `PendlePTOracle` contract will be inflated. As a result, the account's collateral will be overinflated, allowing malicious users to borrow significantly more than the actual collateral value, stealing assets from the protocol.

Code Snippet

https://github.com/sherlock-audit/2024-06-leveraged-vaults/blob/main/leveraged-vaults-private/contracts/oracles/PendlePTOracle.sol#L58

https://github.com/sherlock-audit/2024-06-leveraged-vaults/blob/main/leveraged-vaults-private/contracts/oracles/PendlePTOracle.sol#L85

https://github.com/sherlock-audit/2024-06-leveraged-vaults/blob/main/leveraged-vaults-private/contracts/oracles/PendlePTOracle.sol#L117

Tool used

Manual Review

Recommendation

Update the formula to ensure that the PT rate is divided by the rate decimals of Pendle's PT Oracle.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/notional-finance/leveraged-vaults/pull/97

xiaoming9090

Additional changes made in https://github.com/notional-finance/leveraged-vaults/pull/86/commits/3c5e13050e6bc7d278b401118c03bfed3a787d87

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue H-10: Incorrect assumption that PT rate is 1.0 post-expiry

Source: https://github.com/sherlock-audit/2024-06-leveraged-vaults-judging/issues/69

Found by

lemonmon, xiaoming90

Summary

PT rate will be hardcoded to 1.0 post-expiry, which is incorrect. The price returned from the Notional's `PendlePTOracle` contract will be inflated. As a result, the account's collateral will be overinflated, allowing malicious users to borrow significantly more than the actual collateral value, stealing assets from the protocol.

Vulnerability Detail

The following are the configuration files for the PT weETH 27JUN2024 taken from the test files of the audit contest's repository.

For PT weETH 27JUN2024, note that the `useSyOracleRate` is set to `True`, as per Line 59 below.

https://github.com/sherlock-audit/2024-06-leveraged-vaults/blob/main/leveraged-vaults-private/tests/Staking/PendlePTTests.yml#L51

```
File: PendlePTTests.yml
51:   - stakeSymbol: weETH
52:     forkBlock: 221089505
53:     expiry: 27JUN2024
54:     primaryBorrowCurrency: ETH
55:     contractName: PendlePTGeneric
56:     oracles: [ETH]
57:     marketAddress: "0x952083cde7aaa11AB8449057F7de23A970AA8472"
58:     ptAddress: "0x1c27Ad8a19Ba026ADaBD615F6Bc77158130cfBE4"
59:     useSyOracleRate: 'true'
60:     tradeOnEntry: true
61:     primaryDex: UniswapV3
```

For PT weETH 27JUN2024, note that the `useSyOracleRate` is set to `True`, as per Line 118 below.

https://github.com/sherlock-audit/2024-06-leveraged-vaults/blob/main/leveraged-vaults-private/tests/generated/arbitrum/PendlePT_weETH_ETH.t.sol#L115

```
File: PendlePT_weETH_ETH.t.sol
115:         marketAddress = 0x952083cde7aaa11AB8449057F7de23A970AA8472;
116:         ptAddress = 0x1c27Ad8a19Ba026ADaBD615F6Bc77158130cfBE4;
117:         twapDuration = 15 minutes; // recommended 15 - 30 min
118:         useSyOracleRate = true;
119:         baseToUSDOracle = 0x9414609789C179e1295E9a0559d629bF832b3c04;
120:
121:         tokenInSy = 0x35751007a407ca6FEFfE80b3cB397736D2cf4dbe;
122:         borrowToken = 0x0000000000000000000000000000000000000000;
123:         tokenOutSy = 0x35751007a407ca6FEFfE80b3cB397736D2cf4dbe;
124:         redemptionToken = 0x35751007a407ca6FEFfE80b3cB397736D2cf4dbe;
```

In this case, the `useSyOracleRate` is set to `True` in the `PendlePTOracle` contract.

In Line 115 below, the PT rate will be hardcoded to 1.0 post-expiry. Per the comment at Lines 113-114, it assumes that 1 unit of PT is worth 1 unit of the underlying SY at expiration. However, this assumption is incorrect.

https://github.com/sherlock-audit/2024-06-leveraged-vaults/blob/main/leveraged-vaults-private/contracts/oracles/PendlePTOracle.sol#L113

```
File: PendlePTOracle.sol
092:     function _calculateBaseToQuote() internal view returns (
..SNIP..
113:         // Past expiration, hardcode the PT oracle price to 1. It is no
↪  longer tradable and
114:         // is worth 1 unit of the underlying SY at expiration.
115:         int256 ptRate = expiry <= block.timestamp ? ptDecimals :
↪  _getPTRate();
116:
117:         answer = (ptRate * baseToUSD * rateDecimals) /
118:             (baseToUSDDecimals * ptDecimals);
119:     }
```

Per the Pendle documentation:

> In the case of reward-bearing assets, it's particularly important to note that PT is redeemable 1:1 for the accounting asset, *NOT* the **underlying asset.
>
> For example, the value of Renzo ezETH increases overtime relative to ETH as staking and restaking rewards are accrued. For every 1 PT-ezETH you own, you'll be able to redeem 1 ETH worth of ezETH upon maturity, *NOT* 1 ezETH which has a higher value**.**

Using PT weETH 27JUN2024, which is used within the ether.fi vault as an example:

- Underlying assets = weETH (Wrapped eETH)
- SY = SY weETH
- PT = PT weETH 27JUN2024
- Market = https://arbiscan.io/address/0x952083cde7aaa11AB8449057F7de23A970AA8472 (isExpired = true)
- Accounting assets = eETH = ETH

Per the Pendle's eETH market page, it has stated that 1 PT eETH is equal to 1 eETH (also equal to 1 ETH) at maturity.

However, as noted earlier, the code assumes that one unit of PT is worth one unit of weETH instead of one unit of PT being worth one unit of eETH, which is incorrect.

On 1 July, the price of weETH was 3590 USD, while the price of eETH was 3438 USD. This is a difference of 152 USD.

As a result, the price returned from the Notional's `PendlePTOracle` contract will be inflated.

Additional Information The PT weETH 27JUN2024 has already expired as of 1 July 2024.

Let's verify if the PT rate is 1.0 after maturity by inspecting the rate returned from Pendle PT Oracle's 'getPtToSyRate' function.

As shown above, the PT rate at maturity is `0.9598002817` instead of `1.0`. Thus, it is incorrect to assume that the PT rate is 1.0 post-expiry.

Impact

The price returned from the Notional's `PendlePTOracle` contract will be inflated. As a result, the account's collateral will be overinflated, allowing malicious users to borrow significantly more than the actual collateral value, stealing assets from the protocol.

Code Snippet

https://github.com/sherlock-audit/2024-06-leveraged-vaults/blob/main/leveraged-vaults-private/contracts/oracles/PendlePTOracle.sol#L113

Tool used

Manual Review

Recommendation

Ensure that the correct PT rate is used post-expiry.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/notional-finance/leveraged-vaults/pull/98

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue H-11: Lack of slippage control on `_redeemPT` function

Source:
https://github.com/sherlock-audit/2024-06-leveraged-vaults-judging/issues/70

Found by

BiasedMerc, Ironsidesec, ZeroTrust, blackhole, brgltd, denzi_, lemonmon, pseudoArtist, xiaoming90

Summary

The slippage control on the `_redeemPT` function has been disabled. As a result, it can lead to a loss of assets. Slippage can occur naturally due to on-chain trading activities or the victim being sandwiched by malicious users/MEV.

Vulnerability Detail

In Line 137 of the `_redeemPT` function, the `minTokenOut` is set to `0`, which disables the slippage control. Note that redeeming one `TOKEN_OUT_SY` does not always give you one `netTokenOut`. Not all SY contracts will burn one share and return 1 yield token back. Inspecting the Pendle's source code will reveal that for some SY contracts, some redemption will involve withdrawing/redemption from external staking protocol or performing some swaps, which might suffer from some slippage.

https://github.com/sherlock-audit/2024-06-leveraged-vaults/blob/main/leveraged-vaults-private/contracts/vaults/staking/protocols/PendlePrincipalToken.sol#L137

```
File: PendlePrincipalToken.sol
123:     /// @notice Handles PT redemption whether it is expired or not
124:     function _redeemPT(uint256 vaultShares) internal returns (uint256
 ↪  netTokenOut) {
125:         uint256 netPtIn = getStakingTokensForVaultShare(vaultShares);
126:         uint256 netSyOut;
127:
128:         // PT tokens are known to be ERC20 compatible
129:         if (PT.isExpired()) {
130:             PT.transfer(address(YT), netPtIn);
131:             netSyOut = YT.redeemPY(address(SY));
132:         } else {
133:             PT.transfer(address(MARKET), netPtIn);
134:             (netSyOut, ) = MARKET.swapExactPtForSy(address(SY), netPtIn,
 ↪  "");
135:         }
136:
```

```
137:             netTokenOut = SY.redeem(address(this), netSyOut, TOKEN_OUT_SY, 0,
↪  true);
138:         }
```

The `_redeemPT` function is being used in two places:

Instance 1 - Within `_executeInstantRedemption` function If `TOKEN_OUT_SY == BORROW_TOKEN`, the code will accept any `netTokenOut` redeemed, even if it is fewer than expected due to slippage.

https://github.com/sherlock-audit/2024-06-leveraged-vaults/blob/main/leveraged-vaults-private/contracts/vaults/staking/protocols/PendlePrincipalToken.sol#L140

```
File: PendlePrincipalToken.sol
140:     function _executeInstantRedemption(
141:         address /* account */,
142:         uint256 vaultShares,
143:         uint256 /* maturity */,
144:         RedeemParams memory params
145:     ) internal override returns (uint256 borrowedCurrencyAmount) {
146:         uint256 netTokenOut = _redeemPT(vaultShares);
147: 
148:         if (TOKEN_OUT_SY != BORROW_TOKEN) {
149:             Trade memory trade = Trade({
150:                 tradeType: TradeType.EXACT_IN_SINGLE,
151:                 sellToken: TOKEN_OUT_SY,
152:                 buyToken: BORROW_TOKEN,
153:                 amount: netTokenOut,
154:                 limit: params.minPurchaseAmount,
155:                 deadline: block.timestamp,
156:                 exchangeData: params.exchangeData
157:             });
158: 
159:             // Executes a trade on the given Dex, the vault must have
↪  permissions set for
160:             // each dex and token it wants to sell.
161:             (/* */, borrowedCurrencyAmount) = _executeTrade(params.dexId,
↪  trade);
162:         } else {
163:             borrowedCurrencyAmount = netTokenOut;
164:         }
```

Instance 2 - Within `_initiateWithdrawImpl` function The code will accept any `tokenOutSy` redeemed, even if it is fewer than expected due to slippage, and proceed to withdraw them from external protocols.

https://github.com/sherlock-audit/2024-06-leveraged-vaults/blob/main/leveraged-vaults-private/contracts/vaults/staking/protocols/PendlePrincipalToken.sol#L171

```
File: PendlePrincipalToken.sol
171:     function _initiateWithdrawImpl(
172:         address account, uint256 vaultSharesToRedeem, bool isForced
173:     ) internal override returns (uint256 requestId) {
174:         // When doing a direct withdraw for PTs, we first redeem or trade
 ↪  out of the PT
175:         // and then initiate a withdraw on the TOKEN_OUT_SY. Since the
 ↪  vault shares are
176:         // stored in PT terms, we pass tokenOutSy terms (i.e. weETH or
 ↪  sUSDe) to the withdraw
177:         // implementation.
178:         uint256 tokenOutSy = _redeemPT(vaultSharesToRedeem);
179:         requestId = _initiateSYWithdraw(account, tokenOutSy, isForced);
180:         // Store the tokenOutSy here for later when we do a valuation check
 ↪  against the position
181:         VaultStorage.getWithdrawRequestData()[requestId] =
 ↪  abi.encode(tokenOutSy);
182:     }
```

Impact

Loss of assets due to lack of slippage control. Slippage can occur naturally due to on-chain trading activities or the victim being sandwiched by malicious users/MEV.

Code Snippet

https://github.com/sherlock-audit/2024-06-leveraged-vaults/blob/main/leveraged-vaults-private/contracts/vaults/staking/protocols/PendlePrincipalToken.sol#L137

Tool used

Manual Review

Recommendation

Consider implementing the required slippage control.

Discussion

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/notional-finance/leveraged-vaults/pull/99

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue H-12: The withdrawValue calculation in _calculateValueOfWithdrawRequest is incorrect.

Source: https://github.com/sherlock-audit/2024-06-leveraged-vaults-judging/issues/78

Found by

ZeroTrust The withdrawValue calculation in _calculateValueOfWithdrawRequest is incorrect.

Summary

The withdrawValue calculation in _calculateValueOfWithdrawRequest is incorrect.

Vulnerability Detail

```
function _calculateValueOfWithdrawRequest(
        WithdrawRequest memory w,
        uint256 stakeAssetPrice,
        address borrowToken,
        address redeemToken
    ) internal view returns (uint256 borrowTokenValue) {
        if (w.requestId == 0) return 0;

        // If a withdraw request has split and is finalized, we know the fully
↪   realized value of
        // the withdraw request as a share of the total realized value.
        if (w.hasSplit) {
            SplitWithdrawRequest memory s =
↪   VaultStorage.getSplitWithdrawRequest()[w.requestId];
            if (s.finalized) {
                return _getValueOfSplitFinalizedWithdrawRequest(w, s,
↪   borrowToken, redeemToken);
            }
        }

        // In every other case, including the case when the withdraw request has
↪   split, the vault shares
        // in the withdraw request (`w`) are marked at the amount of vault
↪   shares the account holds.
@>>         return _getValueOfWithdrawRequest(w, stakeAssetPrice);
    }
```

We can see that for cases without hasSplit or with hasSplit but not finalized, the value is calculated using _getValueOfWithdrawRequest. Let's take a look at `PendlePTEtherFiVault::_getValueOfWithdrawRequest()`.

```
function _getValueOfWithdrawRequest(
        WithdrawRequest memory w, uint256 /* */
    ) internal override view returns (uint256) {
@>>         uint256 tokenOutSY = getTokenOutSYForWithdrawRequest(w.requestId);
        // NOTE: in this vault the tokenOutSy is known to be weETH.
        (int256 weETHPrice, /* */) = TRADING_MODULE.getOraclePrice(TOKEN_OUT_SY,
 ↪  BORROW_TOKEN);
        return (tokenOutSY * weETHPrice.toUint() * BORROW_PRECISION) /
            (WETH_PRECISION * Constants.EXCHANGE_RATE_PRECISION);
    }
```

Here, tokenOutSY represents the total amount of WETH requested for withdrawal, not just the portion represented by the possibly split w.vaultShares. Therefore, _getValueOfWithdrawRequest returns the entire withdrawal value. If a WithdrawRequest has been split but not finalized, this results in an incorrect calculation.

This issue also occurs in `PendlePTKelpVault::_getValueOfWithdrawRequest()`.

```
function _getValueOfWithdrawRequest(
        WithdrawRequest memory w, uint256 /* */
    ) internal override view returns (uint256) {
        return KelpLib._getValueOfWithdrawRequest(w, BORROW_TOKEN,
 ↪  BORROW_PRECISION);
    }
//KelpLib._getValueOfWithdrawRequest) function
function _getValueOfWithdrawRequest(
        WithdrawRequest memory w,
        address borrowToken,
        uint256 borrowPrecision
    ) internal view returns (uint256) {
        address holder = address(uint160(w.requestId));

        uint256 expectedStETHAmount;
        if (KelpCooldownHolder(payable(holder)).triggered()) {
            uint256[] memory requestIds =
 ↪  LidoWithdraw.getWithdrawalRequests(holder);
            ILidoWithdraw.WithdrawalRequestStatus[] memory withdrawsStatus =
 ↪  LidoWithdraw.getWithdrawalStatus(requestIds);

            expectedStETHAmount = withdrawsStatus[0].amountOfStETH;
        } else {
```

```
            (/* */, expectedStETHAmount, /* */, /* */) =
↪   WithdrawManager.getUserWithdrawalRequest(stETH, holder, 0);

        }

        (int256 stETHToBorrowRate, /* */) =
↪   Deployments.TRADING_MODULE.getOraclePrice(
            address(stETH), borrowToken
        );
            //@audit not the w.vaulteShares
@>>         return (expectedStETHAmount * stETHToBorrowRate.toUint() *
↪   borrowPrecision) /
            (Constants.EXCHANGE_RATE_PRECISION * stETH_PRECISION);
    }
```

This issue also occurs in `PendlePTStakedUSDeVault::_getValueOfWithdrawRequest()`.

```
function _getValueOfWithdrawRequest(
        WithdrawRequest memory w, uint256 /* */
    ) internal override view returns (uint256) {
        // NOTE: This withdraw valuation is not based on the vault shares value
↪   so we do not
        // need to use the PendlePT metadata conversion.
        return EthenaLib._getValueOfWithdrawRequest(w, BORROW_TOKEN,
↪   BORROW_PRECISION);
    }
//EthenaLib._getValueOfWithdrawRequest()
function _getValueOfWithdrawRequest(
        WithdrawRequest memory w,
        address borrowToken,
        uint256 borrowPrecision
    ) internal view returns (uint256) {
        address holder = address(uint160(w.requestId));
        // This valuation is the amount of USDe the account will receive at
↪   cooldown, once
        // a cooldown is initiated the account is no longer receiving sUSDe
↪   yield. This balance
        // of USDe is transferred to a Silo contract and guaranteed to be
↪   available once the
        // cooldown has passed.
        IsUSDe.UserCooldown memory userCooldown = sUSDe.cooldowns(holder);

        int256 usdeToBorrowRate;
        if (borrowToken == address(USDe)) {
            usdeToBorrowRate = int256(Constants.EXCHANGE_RATE_PRECISION);
        } else {
```

```
            // If not borrowing USDe, convert to the borrowed token
            (usdeToBorrowRate, /* */) =
↪   Deployments.TRADING_MODULE.getOraclePrice(
                address(USDe), borrowToken
            );
        }
        //@audit not the w.vaulteShares
@>>         return (userCooldown.underlyingAmount * usdeToBorrowRate.toUint() *
↪   borrowPrecision) /
            (Constants.EXCHANGE_RATE_PRECISION * USDE_PRECISION);
    }
```

The calculation is correct only in `EtherFiVault::_getValueOfWithdrawRequest()`.

```
function _getValueOfWithdrawRequest(
        WithdrawRequest memory w, uint256 weETHPrice
    ) internal override view returns (uint256) {
        return EtherFiLib._getValueOfWithdrawRequest(w, weETHPrice,
↪   BORROW_PRECISION);
    }
//EtherFiLib._getValueOfWithdrawRequest() function
function _getValueOfWithdrawRequest(
        WithdrawRequest memory w,
        uint256 weETHPrice,
        uint256 borrowPrecision
    ) internal pure returns (uint256) {
            //@audit this is correct, using vaultShares
@>>         return (w.vaultShares * weETHPrice * borrowPrecision) /
            (uint256(Constants.INTERNAL_TOKEN_PRECISION) *
↪   Constants.EXCHANGE_RATE_PRECISION);
    }
```

Impact

Overestimation of user assets can lead to scenarios where users should have been liquidated but were not, resulting in losses for the protocol.

Code Snippet

https://github.com/sherlock-audit/2024-06-leveraged-vaults/blob/14d3eaf0445c251c52c86ce88a84a3f5b9dfad94/leveraged-vaults-private/contracts/vaults/common/WithdrawRequestBase.sol#L86

https://github.com/sherlock-audit/2024-06-leveraged-vaults/blob/14d3eaf0445c251c52c86ce88a84a3f5b9dfad94/leveraged-vaults-private/contracts/vaults/stakin

g/PendlePTEtherFiVault.sol#L46

https://github.com/sherlock-audit/2024-06-leveraged-vaults/blob/main/leveraged-vaults-private/contracts/vaults/staking/protocols/Kelp.sol#L114

https://github.com/sherlock-audit/2024-06-leveraged-vaults/blob/main/leveraged-vaults-private/contracts/vaults/staking/protocols/Ethena.sol#L77

Tool used

Manual Review

Recommendation

Modify the relevant calculation formulas.

Discussion

0502lian

Escalate

This should be considered a High issue.

Although _calculateValueOfWithdrawRequest is just a view function, it is used by BaseStakingVault::convertStrategyToUnderlying to evaluate the value of a user's shares in the staking Vault.

```
 function convertStrategyToUnderlying(
        address account,
        uint256 vaultShares,
        uint256 /* maturity */
    ) public virtual override view returns (int256 underlyingValue) {
        uint256 stakeAssetPrice = uint256(getExchangeRate(0));

        WithdrawRequest memory w = getWithdrawRequest(account);
@>        uint256 withdrawValue = _calculateValueOfWithdrawRequest(
            w, stakeAssetPrice, BORROW_TOKEN, REDEMPTION_TOKEN
        );
        // This should always be zero if there is a withdraw request.
        uint256 vaultSharesNotInWithdrawQueue = (vaultShares - w.vaultShares);

        uint256 vaultSharesValue = (vaultSharesNotInWithdrawQueue *
↪   stakeAssetPrice * BORROW_PRECISION) /
            (uint256(Constants.INTERNAL_TOKEN_PRECISION) *
↪   Constants.EXCHANGE_RATE_PRECISION);
```

```
@>          return (withdrawValue + vaultSharesValue).toInt();
    }
```

In Notional V3, the function calculateAccountHealthFactors calculates a user's health condition by calling getPrimaryUnderlyingValueOfShare, which in turn calls the vault's convertStrategyToUnderlying.

https://github.com/notional-finance/contracts-v3/blob/b664c157051d256ce583ba19da3e26c6cf5061ac/contracts/internal/vaults/VaultValuation.sol#L173

```
/// @notice Calculates account health factors for liquidation.
    function calculateAccountHealthFactors(
        VaultConfig memory vaultConfig,
        VaultAccount memory vaultAccount,
        VaultState memory vaultState,
        PrimeRate[2] memory primeRates
    ) internal view returns (
        VaultAccountHealthFactors memory h,
        VaultSecondaryBorrow.SecondaryExchangeRates memory er
    ) {
@>>         h.vaultShareValueUnderlying = getPrimaryUnderlyingValueOfShare(
             vaultState, vaultConfig, vaultAccount.account,
↪   vaultAccount.vaultShares
        );

       //-------skip-----
    }
```

https://github.com/notional-finance/contracts-v3/blob/b664c157051d256ce583ba19da3e26c6cf5061ac/contracts/internal/vaults/VaultValuation.sol#L54

```
/// @notice Returns the value in underlying of the primary borrow currency
↪   portion of vault shares.
    function getPrimaryUnderlyingValueOfShare(
        VaultState memory vaultState,
        VaultConfig memory vaultConfig,
        address account,
        uint256 vaultShares
    ) internal view returns (int256) {
        if (vaultShares == 0) return 0;

        Token memory token =
↪   TokenHandler.getUnderlyingToken(vaultConfig.borrowCurrencyId);
        return token.convertToInternal(
```

```
@>>
↪  IStrategyVault(vaultConfig.vault).convertStrategyToUnderlying(account,
↪  vaultShares, vaultState.maturity)
        );
    }
```

Whether a user's position should be liquidated is determined based on calculateAccountHealthFactors.

Therefore, as stated in the scope of impact in my report, "Overestimation of user assets can lead to scenarios where users should have been liquidated but were not, resulting in losses for the protocol."

Also considering the widespread nature of the error, in this audit, the calculation in PendlePTEtherFiVault, PendlePTKelpVault, PendlePTStakedUSDeVault, and EthenaVault is incorrect.

The impact of this error is High, and the likelihood of occurrence is between medium and high, according to Sherlock's rules on how to identify a high issue: "Definite loss of funds without (extensive) limitations of external conditions." Therefore, it should be judged as High.

sherlock-admin3

> Escalate
>
> This should be considered a High issue.
>
> Although _calculateValueOfWithdrawRequest is just a view function, it is used by BaseStakingVault::convertStrategyToUnderlying to evaluate the value of a user's shares in the staking Vault.

```
 function convertStrategyToUnderlying(
        address account,
        uint256 vaultShares,
        uint256 /* maturity */
    ) public virtual override view returns (int256 underlyingValue) {
        uint256 stakeAssetPrice = uint256(getExchangeRate(0));

        WithdrawRequest memory w = getWithdrawRequest(account);
@>        uint256 withdrawValue = _calculateValueOfWithdrawRequest(
            w, stakeAssetPrice, BORROW_TOKEN, REDEMPTION_TOKEN
        );
        // This should always be zero if there is a withdraw request.
        uint256 vaultSharesNotInWithdrawQueue = (vaultShares -
↪   w.vaultShares);

        uint256 vaultSharesValue = (vaultSharesNotInWithdrawQueue *
↪   stakeAssetPrice * BORROW_PRECISION) /
```


```
            (uint256(Constants.INTERNAL_TOKEN_PRECISION) *
↪   Constants.EXCHANGE_RATE_PRECISION);
@>          return (withdrawValue + vaultSharesValue).toInt();
    }
```

In Notional V3, the function calculateAccountHealthFactors calculates a user's health condition by calling getPrimaryUnderlyingValueOfShare, which in turn calls the vault's convertStrategyToUnderlying.

https://github.com/notional-finance/contracts-v3/blob/b664c157051d256ce583ba19da3e26c6cf5061ac/contracts/internal/vaults/VaultValuation.sol#L173

```
/// @notice Calculates account health factors for liquidation.
    function calculateAccountHealthFactors(
        VaultConfig memory vaultConfig,
        VaultAccount memory vaultAccount,
        VaultState memory vaultState,
        PrimeRate[2] memory primeRates
    ) internal view returns (
        VaultAccountHealthFactors memory h,
        VaultSecondaryBorrow.SecondaryExchangeRates memory er
    ) {
@>>         h.vaultShareValueUnderlying = getPrimaryUnderlyingValueOfShare(
             vaultState, vaultConfig, vaultAccount.account,
↪   vaultAccount.vaultShares
        );

       //-------skip-----
    }
```

https://github.com/notional-finance/contracts-v3/blob/b664c157051d256ce583ba19da3e26c6cf5061ac/contracts/internal/vaults/VaultValuation.sol#L54

```
/// @notice Returns the value in underlying of the primary borrow currency
↪   portion of vault shares.
    function getPrimaryUnderlyingValueOfShare(
        VaultState memory vaultState,
        VaultConfig memory vaultConfig,
        address account,
        uint256 vaultShares
    ) internal view returns (int256) {
        if (vaultShares == 0) return 0;
```

```
        Token memory token =
↪   TokenHandler.getUnderlyingToken(vaultConfig.borrowCurrencyId);
        return token.convertToInternal(
@>>
↪   IStrategyVault(vaultConfig.vault).convertStrategyToUnderlying(account,
↪   vaultShares, vaultState.maturity)
        );
    }
```

Whether a user's position should be liquidated is determined based on calculateAccountHealthFactors.

Therefore, as stated in the scope of impact in my report, "Overestimation of user assets can lead to scenarios where users should have been liquidated but were not, resulting in losses for the protocol."

Also considering the widespread nature of the error, in this audit, the calculation in PendlePTEtherFiVault, PendlePTKelpVault, PendlePTStakedUSDeVault, and EthenaVault is incorrect.

The impact of this error is High, and the likelihood of occurrence is between medium and high, according to Sherlock's rules on how to identify a high issue: "Definite loss of funds without (extensive) limitations of external conditions." Therefore, it should be judged as High.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

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

0502lian

Additional information:

This issue has the same impact as issue 60 https://github.com/sherlock-audit/2024-06-leveraged-vaults-judging/issues/60, although the error occurs in slightly different ways; issue 60 is caused by token precision errors, while this issue is due to incorrect share referencing. There is no reason why issue 60 is high while this issue is considered medium.

T-Woodward

Yes I think this is correct. Definitely should be a high severity issue

mystery0x

I still think medium rating will be more appropriate but will let the Sherlock judge decide on it:

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

WangSecurity

@0502lian I see that the functions mentioned in the escalation, which in the end call the`calculateValueOfWithdrawRequest`, are all view functions, but some of them internal, so I assume there are non-view functions that interact with them. Can you please forward me to them?

0502lian

1 About the Call Chain

Notional is a very complex protocol.
https://github.com/notional-finance/contracts-v3/blob/b664c157051d256ce583ba19da3e26c6cf5061ac/contracts/external/actions/VaultLiquidationAction.sol#L58

```
  function deleverageAccount(
        address account,
        address vault,
        address liquidator,
        uint16 currencyIndex,
        int256 depositUnderlyingInternal
    ) external payable nonReentrant override returns (
        uint256 vaultSharesToLiquidator,
        int256 depositAmountPrimeCash
    ) {
    //......skip.........
        // Currency Index is validated in this method
        (
            depositUnderlyingInternal,
            vaultSharesToLiquidator,
            pr
@>>         ) =
 ↪  IVaultAccountHealth(address(this)).calculateDepositAmountInDeleverage(
            currencyIndex, vaultAccount, vaultConfig, vaultState,
 ↪  depositUnderlyingInternal
        );
        //......skip.........
    }
```

You can see that the liquidation function deleverageAccount calls calculateDepositAmountInDeleverage().

https://github.com/notional-finance/contracts-v3/blob/b664c157051d256ce583ba19da3e26c6cf5061ac/contracts/external/actions/VaultAccountHealth.sol#L252

```
function calculateDepositAmountInDeleverage(
        uint256 currencyIndex,
        VaultAccount memory vaultAccount,
        VaultConfig memory vaultConfig,
        VaultState memory vaultState,
        int256 depositUnderlyingInternal
    ) external override returns (int256, uint256, PrimeRate memory) {
        // This method is only intended for use during deleverage account
   //.............skip...........

@>>             (h, er) =
 ↪  VaultValuation.calculateAccountHealthFactors(vaultConfig, vaultAccount,
 ↪  vaultState, primeRates);
        //.............skip...........
```

calculateDepositAmountInDeleverage() calls calculateAccountHealthFactors(), which I listed in my last comment. Finally, it calls the problematic function.

2 About the Sponsor

The sponsor has supported this issue as high priority in the above comments. “Yes I think this is correct. Definitely should be a high severity issue”

3 About Severity Comparison

As I have already expressed above, issue 60 also affects the _getValueOfSplitFinalizedWithdrawRequest view function(called by _calculateValueOfWithdrawRequest()), but its severity is rated as High.

The evidence is so obvious. I understand that the Sherlock judge wants to get to the truth, but I don’t understand why the first judge ignored this evidence and still considered it to be of medium severity.

@WangSecurity

WangSecurity

Based on the comment above, I agree high severity is indeed appropriate here. Planning to accept the escalation and upgrade the severity.

> The evidence is so obvious. I understand that the Sherlock judge wants to get to the truth, but I don’t understand why the first judge ignored this evidence and still considered it to be of medium severity

The problem here is that all the functions mentioned in the report and messages above you just say where the formula is incorrect. From judge's point of view it would be easier if you would make vulnerability path which function would be called and how this sequence of calls would lead to the impact. Just as an advice try to write your reports in a different way so it's easier to judge it correctly initially. Also, try new report templates, I think with them it will be a lot easier for both you and judges :)

WangSecurity

Result: High Unique

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- 0502lian: accepted

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/notional-finance/leveraged-vaults/pull/90/commits

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue H-13: `Kelp:_finalizeCooldown` cannot claim the withdrawal if adversary would requestWithdrawals with dust amount for the holder

Source: https://github.com/sherlock-audit/2024-06-leveraged-vaults-judging/issues/105

Found by

BiasedMerc, lemonmon, xiaoming90

Summary

If an adversary calls `LidoWithdraw.requestWithdrawals` with some dust stETH amount and the `KelpCooldownHolder`'s address, the withdrawal will be locked. It cannot be rescued via `rescueTokens`, since it does not have to logic to `claimWithdrawal`, therefore the withdrawal will be locked permanently.

Vulnerability Detail

The `KelpCooldownHolder` is responsible to withdraw from rsETH to stETH via `WithdraManager`, and then withdraw stETH to ETH via `LidoWithdraw`. Since it is two step process, the `KelpCooldownHolder` implements `triggerExtraStep`. After the first withdrawal request is finalized, the `triggerExtraStep` should be called to initiate the second withdrawal request.
https://github.com/sherlock-audit/2024-06-leveraged-vaults/blob/main/leveraged-vaults-private/contracts/vaults/staking/protocols/Kelp.sol#L83

The `KelpCooldownHolder._finalizeCooldown` will be called when the `exitVault` is called to finalize the process by `LidoWithdraw.claimWithdrawal` and send the claimed ETH back to the vault.

https://github.com/sherlock-audit/2024-06-leveraged-vaults/blob/main/leveraged-vaults-private/contracts/vaults/staking/protocols/Kelp.sol#L100

This `KelpCooldownHolder._finalizeCooldown` is, however, hardcoded to claim the 0-th request from the `LidoWithdraw.getWithdrawalRequests`, thus if there are more than 1 withdrawal requests for this `KelpCooldownHolder`, all the requests but the 0-th request will be ignored.

An adversary can abuse this fact by request withdrawal for the `KelpCooldownHolder` before the `triggerExtraStep` is called. In that case the real withdrawal request will not be the 0-th, and will be ignored.

After the finalize on the dust withdrawal is done, the accountWithdrawRequest on the PendlePTKelpVault will be deleted and this `finalizeCooldown` on the holder cannot be called again since it has onlyVault modifier. The `rescueTokens` will not help, as the stETH is already transferred to LidoWithdraw.

PoC

Here is a proof of concept demonstrating that a third party can call `LidoWithdrawals.requestWithdrawals`. And the requestId is in the order of request:

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.24;

import "forge-std/Test.sol";

import {IERC20} from "@interfaces/IERC20.sol";

interface ILidoWithdraw {
    struct WithdrawalRequestStatus {
        uint256 amountOfStETH;
        uint256 amountOfShares;
        address owner;
        uint256 timestamp;
        bool isFinalized;
        bool isClaimed;
    }

    function requestWithdrawals(uint256[] memory _amounts, address _owner)
↪   external returns (uint256[] memory requestIds);
    function getWithdrawalRequests(address _owner) external view returns
↪   (uint256[] memory requestsIds);
    function getWithdrawalStatus(uint256[] memory _requestIds) external view
↪   returns (WithdrawalRequestStatus[] memory statuses);
}

IERC20 constant rsETH = IERC20(0xA1290d69c65A6Fe4DF752f95823fae25cB99e5A7);
address constant stETH = 0xae7ab96520DE3A18E5e111B5EaAb095312D7fE84;
ILidoWithdraw constant LidoWithdraw =
↪   ILidoWithdraw(0x889edC2eDab5f40e902b864aD4d7AdE8E412F9B1);


contract KelpTestLido is Test {
    address stETHWhale;
    uint tokensClaimed;

    function setUp() public {
```

```
        stETHWhale = 0x804a7934bD8Cd166D35D8Fb5A1eb1035C8ee05ce;
        tokensClaimed = 10e18;
        vm.startPrank(stETHWhale);
        IERC20(stETH).transfer(address(this), tokensClaimed);
        vm.stopPrank();
    }

    function test_Lido_requestId() public {
        uint256[] memory amounts = new uint256[](1);
        amounts[0] = 100;
        // Adversary calls requestWithdrawals for the holder with dust amount
        vm.startPrank(stETHWhale);
        IERC20(stETH).approve(address(LidoWithdraw), amounts[0]);
        LidoWithdraw.requestWithdrawals(amounts, address(this));
        vm.stopPrank();

        // The real withdrawal request is done after
        amounts[0] = tokensClaimed;
        IERC20(stETH).approve(address(LidoWithdraw), amounts[0]);
        uint256[] memory requestIds = LidoWithdraw.requestWithdrawals(amounts,
↪   address(this));
        uint real_requestId = requestIds[0];
        for(uint i=0; i< requestIds.length; i++) {
          emit log_named_uint("request id for real withdrawal request",
↪   requestIds[i]);
        }

        requestIds = LidoWithdraw.getWithdrawalRequests(address(this));
        for(uint i=0; i< requestIds.length; i++) {
          console.log("id for %s: %s",i, requestIds[i]);
        }

        ILidoWithdraw.WithdrawalRequestStatus[] memory withdrawsStatus =
↪   LidoWithdraw.getWithdrawalStatus(requestIds);
        for(uint i=0; i< requestIds.length; i++) {
          console.log("stETH amount for %s: %s",i,
↪   withdrawsStatus[i].amountOfStETH);
        }

        require(real_requestId != requestIds[0]);
    }
}
```

The Result is below: note that the 0-th request only has the dust amount of stETH request (100)

```
[PASS] test_Lido_requestId() (gas: 370501)
Logs:
  request id for real withdrawal request: 44537
  id for 0: 44536
  id for 1: 44537
  stETH amount for 0: 100
  stETH amount for 1: 10000000000000000000
```

The addresses are taken from the mainnet, so should fork from the mainnet to test.

Impact

A malicious actor can use dust amount of stETH to freeze withdrawal from the `KelpPTKelpVault`. The frozen withdrawal will be locked permanently,

Code Snippet

https://github.com/sherlock-audit/2024-06-leveraged-vaults/blob/main/leveraged-vaults-private/contracts/vaults/staking/protocols/Kelp.sol#L83

https://github.com/sherlock-audit/2024-06-leveraged-vaults/blob/main/leveraged-vaults-private/contracts/vaults/staking/protocols/Kelp.sol#L100

Tool used

Manual Review

Recommendation

Consider claiming all the withdrawals from LidoWithdraw. However, an adversary can request withdrawals for the holder multiple times with dust stETH, attempting another DoS factor. Alternatively consider storing the requestId in the triggerExtraStep, and claim only the stored request in the finalize step.

Discussion

mystery0x

The severity of this finding being medium is due to the fact it requires an externally malicious factor to incur the exploit.

xiaoming9090

Escalate

This issue should be High instead of Medium.

Anyone can easily execute this attack by injecting malicious withdrawal requests into the withdrawal queue, causing harm to the victim. The attack could lead to the following impacts:

- Loss of assets
- Account's collateral to be undervalued, leading to unexpected liquidation

Refer to my report for more details (https://github.com/sherlock-audit/2024-06-leveraged-vaults-judging/issues/62)

The lead judge downgraded this issue from High to Medium based on the following rationale, which is incorrect:

> The severity of this finding being medium is due to the fact it requires an externally malicious factor to incur the exploit

Anyone can execute this attack anytime, so no externally malicious factor is required here.

sherlock-admin3

> Escalate
>
> This issue should be High instead of Medium.
>
> Anyone can easily execute this attack by injecting malicious withdrawal requests into the withdrawal queue, causing harm to the victim. The attack could lead to the following impacts:
>
> - Loss of assets
> - Account's collateral to be undervalued, leading to unexpected liquidation
>
> Refer to my report for more details (https://github.com/sherlock-audit/2024-06-leveraged-vaults-judging/issues/62)
>
> The lead judge downgraded this issue from High to Medium based on the following rationale, which is incorrect:
>
> > The severity of this finding being medium is due to the fact it requires an externally malicious factor to incur the exploit
>
> Anyone can execute this attack anytime, so no externally malicious factor is required here.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

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

mystery0x

Please see my comment on #62

WangSecurity

I agree with the escalation, I don't see any malicious factors that are needed here and the attack is easily repeatable. Even though the attacker doesn't get anything, it's a clear loss of the entire withdrawal without external limitations.

Planning to accept the escalation and leave the issue as it is.

WangSecurity

Result: High Has duplicates

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- xiaoming9090: accepted

jeffywu

As a fix for this issue, we are switching to direct ETH withdraws from Kelp instead of stETH. This was not supported at the time we wrote the code but is now supported.

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/notional-finance/leveraged-vaults/pull/92

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-1: _claimRewardToken() will update accountRewardDebt even when there is a failure during reward claiming, as a result, a user might lose rewards.

Source: https://github.com/sherlock-audit/2024-06-leveraged-vaults-judging/issues/1

The protocol has acknowledged this issue.

Found by

BiasedMerc, ZeroTrust, chaduke, eeyore, nirohgo, xiaoming90

Summary

`_claimRewardToken()` will update accountRewardDebt even when there is a failure during reward claiming, for example, when there is a lack of balances or a temporary blacklist that prevents an account from receiving tokens for the moment. As a result, a user might lose rewards.

Vulnerability Detail

_claimRewardToken() will be called when a user needs to claim rewards, for example, via claimAccountRewards() -> _claimAccountRewards() -> _claimRewardToken().

However, the problem is that `_claimRewardToken()` will update accountRewardDebt even when there is a failure during reward claiming, for example, when there is a lack of balances or a temporary blacklist that prevents an account from receiving tokens for the moment.

https://github.com/sherlock-audit/2024-06-leveraged-vaults/blob/14d3eaf0445c251c52c86ce88a84a3f5b9dfad94/leveraged-vaults-private/contracts/vaults/common/VaultRewarderLib.sol#L295-L328

The following code will be executed to update `accountRewardDebt`:

```
VaultStorage.getAccountRewardDebt()[rewardToken][account] = (
            (vaultSharesAfter * rewardsPerVaultShare) /
                uint256(Constants.INTERNAL_TOKEN_PRECISION)
        );
```

Meanwhile, the try-catch block will succeed without reverting even there is a failure: for example, when there is a lack of balances or a temporary blacklist that prevents an account from receiving tokens for the moment.

As a result, a user will lost rewards since `accountRewardDebt` has been updated even though he has not received the rewards.

Impact

_claimRewardToken() will update accountRewardDebt even when there is a failure during reward claiming, as a result, a user might lose rewards.

Code Snippet

Tool used

Manual reading and foundry

Manual Review

Recommendation

We should only update `accountRewardDebt` when the claim is successful.

```
function _claimRewardToken(
        address rewardToken,
        address account,
        uint256 vaultSharesBefore,
        uint256 vaultSharesAfter,
        uint256 rewardsPerVaultShare
    ) internal returns (uint256 rewardToClaim) {
        rewardToClaim = _getRewardsToClaim(
            rewardToken, account, vaultSharesBefore, rewardsPerVaultShare
        );

-        VaultStorage.getAccountRewardDebt()[rewardToken][account] = (
-            (vaultSharesAfter * rewardsPerVaultShare) /
-                uint256(Constants.INTERNAL_TOKEN_PRECISION)
-        );

        if (0 < rewardToClaim) {
            // Ignore transfer errors here so that any strange failures here do
↪   not
            // prevent normal vault operations from working. Failures may
↪   include a
            // lack of balances or some sort of blacklist that prevents an
↪   account
            // from receiving tokens.
            try IEIP20NonStandard(rewardToken).transfer(account, rewardToClaim) {
                bool success = TokenUtils.checkReturnCode();
```

```
                if (success) {
+                   VaultStorage.getAccountRewardDebt()[rewardToken][account] = (
+                   (vaultSharesAfter * rewardsPerVaultShare) /
+                   uint256(Constants.INTERNAL_TOKEN_PRECISION)
+                   );
                    emit VaultRewardTransfer(rewardToken, account,
↪   rewardToClaim);
                } else {
                    emit VaultRewardTransfer(rewardToken, account, 0);
                }
            // Emits zero tokens transferred if the transfer fails.
            } catch {
                emit VaultRewardTransfer(rewardToken, account, 0);
            }
        }
    }
```

Discussion

jeffywu

While this is a valid, it's not clear what an alternative behavior would be. In the event that the transfer fails, there is no clear path to getting the token accounting back to a proper amount. We also do not want to allow the transaction to revert or this would block liquidations from being processed.

In the case where token receivers are blacklisted, not receiving rewards is probably the least bad of all potential outcomes.

jeffywu

Confirming that we will not fix this issue. If users are blacklisted or unable to receive rewards, those rewards will still be on the contract and may be transferrable via some other route after an upgrade. Any other solution here may cause the system to revert which would break liquidations.

In my opinion, some reward token becoming untransferrable to an account is not very likely.

Issue M-2: After a liquidator liquidates someone else's position, it could cause a Denial of Service (DoS) when their own position also needs to be liquidated.

Source: https://github.com/sherlock-audit/2024-06-leveraged-vaults-judging/issues/44

Found by

ZeroTrust

Summary

After a liquidator liquidates someone else's position, it could cause a Denial of Service (DoS) when their own position also needs to be liquidated.

Vulnerability Detail

```
function _splitWithdrawRequest(address _from, address _to, uint256 vaultShares)
↪  internal {
        WithdrawRequest storage w =
↪  VaultStorage.getAccountWithdrawRequest()[_from];
        if (w.requestId == 0) return;

        // Create a new split withdraw request
        if (!w.hasSplit) {
            SplitWithdrawRequest memory s =
↪  VaultStorage.getSplitWithdrawRequest()[w.requestId];
            // Safety check to ensure that the split withdraw request is not
↪  active, split withdraw
            // requests are never deleted. This presumes that all withdraw
↪  request ids are unique.
            require(s.finalized == false && s.totalVaultShares == 0);
            VaultStorage.getSplitWithdrawRequest()[w.requestId].totalVaultShares
↪  = w.vaultShares;
        }

        if (w.vaultShares == vaultShares) {
            // If the resulting vault shares is zero, then delete the request.
↪  The _from account's
            // withdraw request is fully transferred to _to
            delete VaultStorage.getAccountWithdrawRequest()[_from];
        } else {
            // Otherwise deduct the vault shares
```

```
@>>             w.vaultShares = w.vaultShares - vaultShares;
            w.hasSplit = true;
        }

        // Ensure that no withdraw request gets overridden, the _to account
↪   always receives their withdraw
        // request in the account withdraw slot.
        WithdrawRequest storage toWithdraw =
↪   VaultStorage.getAccountWithdrawRequest()[_to];
        require(toWithdraw.requestId == 0 || toWithdraw.requestId == w.requestId
↪   , "Existing Request");

        // Either the request gets set or it gets incremented here.
         toWithdraw.requestId = w.requestId;
        toWithdraw.vaultShares = toWithdraw.vaultShares + vaultShares;
        toWithdraw.hasSplit = true;
    }
```

Here is an assumption: the requested withdrawal vaultShares are always greater than or equal to vaultSharesFromLiquidation. This assumption holds true for regular users because the requested withdrawal vaultShares represent the user's entire vaultShares. However, this assumption does not hold for liquidators. POC The liquidatorvaultShares 10e8 A user need to be liquidated: vaultShares 1e8, withdrawRequest.vaultShares 1e8 the vaultSharesFromLiquidation is 1e8.

- After liquidation: The liquidatorvaultShares 11e8, withdrawRequest.vaultShares 1e8
- After sometime,The liquidator's position need to be liquidated the vaultSharesFromLiquidation is 2e8. Then `w.vaultShares = w.vaultShares - vaultShares;` will revert() This results in the liquidator's position not being liquidated, leading to a loss of funds for the protocol.

Impact

This results in the liquidator's position not being liquidated, leading to a loss of funds for the protocol.

Code Snippet

https://github.com/sherlock-audit/2024-06-leveraged-vaults/blob/14d3eaf0445c251c52c86ce88a84a3f5b9dfad94/leveraged-vaults-private/contracts/vaults/common/WithdrawRequestBase.sol#L205

Tool used

Manual Review

Recommendation

Consider the case where w.vaultShares is less than vaultShares.

Discussion

sherlock-admin4

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

0xmystery commented:

> w.vaultShares = w.vaultShares - vaultShares would just revert

0502lian

Escalate This should be considered a valid issue. The report points out the specific error, the conditions under which it occurs, and the impact it has when it does.

sherlock-admin3

> Escalate This should be considered a valid issue. The report points out the specific error, the conditions under which it occurs, and the impact it has when it does.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

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

mystery0x

> Escalate This should be considered a valid issue. The report points out the specific error, the conditions under which it occurs, and the impact it has when it does.

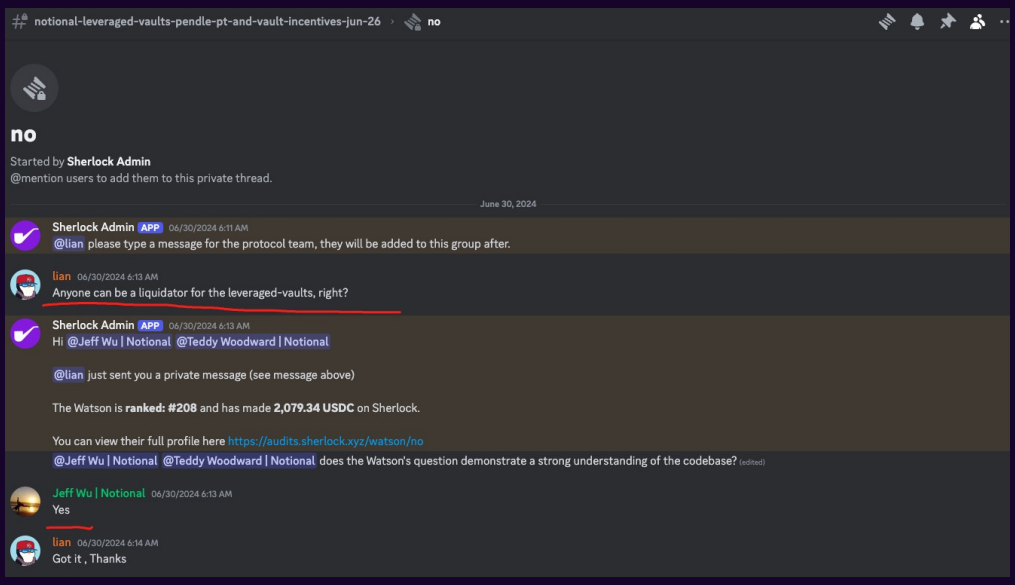
It's intended to deny the call if w.vaultShares < vaultShares. What's your suggested mitigation?

0502lian

> Escalate This should be considered a valid issue. The report points out the specific error, the conditions under which it occurs, and the impact it has when it does.

> It's intended to deny the call if w.vaultShares < vaultShares. What's your suggested mitigation?

The Dev team did not consider the scenario where w.vaultShares < vaultShares, which is very likely to occur. It has been confirmed with the sponsor that anyone can be a liquidator.



As long as the liquidator has their own position and liquidates someone else's position, this scenario will occur.

After this scenario occurs, the Dev team needs to consider more handling logic, which might not be solvable with just a few lines of code. So I currently don't have a particularly good suggested mitigation

WangSecurity

Please elaborate on how this can occur:

> After sometime,The liquidator's position need to be liquidated the vaultSharesFromLiquidation is 2e8. Then w.vaultShares = w.vaultShares - vaultShares; will revert() This results in the liquidator's position not being liquidated, leading to a loss of funds for the protocol

the `vaultShares` is input data, so how this situation may occur, the report just says it happens, without a sufficient explanation it is indeed possible.

0502lian

> Please elaborate on how this can occur:
>
> > After sometime,The liquidator's position need to be liquidated the vaultSharesFromLiquidation is 2e8. Then w.vaultShares = w.vaultShares - vaultShares; will revert() This results in the

liquidator's position not being liquidated, leading to a loss of funds for the protocol

the `vaultShares` is input data, so how this situation may occur, the report just says it happens, without a sufficient explanation it is indeed possible.

vaultShares is an input parameter for _splitWithdrawRequest, but it is not provided by the user. Instead, it is determined by the Notional protocol when the position needs to be liquidated, deciding how many vaultShares need to be liquidated to return to a healthy state.

In fact, most of the time, the entire vaultShares of a position need to be liquidated, and w.vaultShares is always less than the entire vaultShares (for liquidator's position)

https://github.com/sherlock-audit/2024-06-leveraged-vaults/blob/14d3eaf0445c251c52c86ce88a84a3f5b9dfad94/leveraged-vaults-private/contracts/vaults/staking/BaseStakingVault.sol#L208

T-Woodward

I think this is a valid finding, although I would ask @jeffywu to confirm in case I am missing anything.

The watson is correct that we did not anticipate an account having more vaultShares than their withdrawal request. The code is designed such that accounts should either only have a withdrawal request or have no withdrawal request at all. And he is also correct that this situation could cause an inability to liquidate.

However, I'm not sure this deserves a high severity. An account can't force their way into this situation. And I don't think the watson's assertion that "this is very likely to occur" is substantiated.

Furthermore, even if they are in the situation where they have a withdrawal request + additional vaultShares I don't think there's any way that they can take advantage of it. Additional deposits will fail, and if they try to redeem the vault will only allow them to redeem their full withdraw request which would put them back in the situation they were in before they did the liquidation.

So there's no way that an account could borrow against their position and take cash out of the system knowing that they couldn't get liquidated. They would just kind of be stuck.

WangSecurity

Thank you very much, with that I agree it's medium severity. The reason is that, firstly, the attacker cannot force them into this situation, secondly, there's not real benefit for them, except not being liqudiatable when they should.

Moreover, as I understand, the regular user can end up in this situation, which would be problem for them, based on the comment above.

Planning to accept the escalation and validate with medium severity. @mystery0x @0502lian are there any additional duplicates?

brakeless-wtp

I believe it is unique.

WangSecurity

Result: Medium Unique

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- 0502lian: accepted

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits: https://github.com/notional-finance/leveraged-vaults/pull/88

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-3: Premature collateralization check in the BaseStakingVault.initiateWithdraw() function can leave accounts undercollateralized

Source: https://github.com/sherlock-audit/2024-06-leveraged-vaults-judging/issues/56

Found by

eeyore

Summary

The collateralization check is currently performed before the user action that impacts the account's collateralization.

Vulnerability Detail

The `initiateWithdraw()` function can affect the solvency of the account. During this process, tokens may be unwrapped and new tokens pushed into the withdrawal queue, altering the underlying tokens for which collateralization was initially checked. This can result in a different collateralization level than initially assessed.

Additionally, this contradicts how Notional core contracts perform such checks, where they are always conducted as the final step in any user interaction.

Impact

The account may become undercollateralized or insolvent following the user action.

Code Snippet

https://github.com/sherlock-audit/2024-06-leveraged-vaults/blob/main/leveraged-vaults-private/contracts/vaults/staking/BaseStakingVault.sol#L250-L255

Tool used

Manual Review

Recommendation

Perform the account collateralization check after the `_initiateWithdraw()` function call:

```
function initiateWithdraw() external {
+       _initiateWithdraw({account: msg.sender, isForced: false});

        (VaultAccountHealthFactors memory health, /* */, /* */) =
 ↪  NOTIONAL.getVaultAccountHealthFactors(
            msg.sender, address(this)
        );
        VaultConfig memory config = NOTIONAL.getVaultConfig(address(this));
        // Require that the account is collateralized
        require(config.minCollateralRatio <= health.collateralRatio,
 ↪  "Insufficient Collateral");

-       _initiateWithdraw({account: msg.sender, isForced: false});
    }
}
```

Discussion

sherlock-admin4

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

0xmystery commented:

> Lacks proof to substantiate the bug on an intended design

Hash01011122 commented:

> Invalid, If this were to occur it would be admin error to add any other underlying asset

T-Woodward

I think this is a valid finding. Medium is a reasonable severity imo.

@jeffywu this reinforces the need to value withdraw requests as if they were still the staked asset except in the very strict case where we know exactly what we will get upon unstaking and exactly when we will get it

0xklapouchy

@mystery0x

To simplify the issue described, let's illustrate it using the `PendlePTKelpVault`.

Before initiating a withdrawal, the vault shares' value is calculated based on the rsETH/WETH price. After a withdrawal request, the calculation is based on the stETH/WETH price.

In a situation where, for some reason, the stETH price drops compared to rsETH, the user's shares' value will also drop after initiating the withdrawal request, leading to the position becoming unhealthy for both the user and the protocol.

In such a case, it is better to block the initiation of the withdrawal. The better operation for the user or the Notional protocol in such cases is to close the position via `_executeInstantRedemption()`, which will redeem rsETH to WETH via TRADING_MODULE at a better price.

0xklapouchy

Escalate.

This is a valid medium issue.

sherlock-admin3

> Escalate.
>
> This is a valid medium issue.

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

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

xiaoming9090

Escalate.

Escalating this issue on behalf of the submitter. Please review the above comments. Thanks.

sherlock-admin3

> Escalate.
>
> Escalating this issue on behalf of the submitter. Please review the above comments. Thanks.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

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

WangSecurity

Agree with the escalation, planning to accept it and validate with medium severity. @mystery0x are there additional duplicates?

brakeless-wtp

I believe it is unique.

WangSecurity

Result: Medium Unique

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- xiaoming9090: accepted

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/notional-finance/leveraged-vaults/pull/95/files

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-4: `rescueTokens` feature is broken

Source: https://github.com/sherlock-audit/2024-06-leveraged-vaults-judging/issues/72

Found by

xiaoming90

Summary

The rescue function is broken, and tokens cannot be rescued when needed, leading to assets being stuck in the contract.

Vulnerability Detail

The `ClonedCoolDownHolder` contains a feature that allows the protocol to recover lost tokens, as per the comment in Line 22 below. This function is guarded by the `onlyVault` modifier. Thus, only the vault can call this function.

https://github.com/sherlock-audit/2024-06-leveraged-vaults/blob/main/leveraged-vaults-private/contracts/vaults/staking/protocols/ClonedCoolDownHolder.sol#L23

```
File: ClonedCoolDownHolder.sol
22:     /// @notice If anything ever goes wrong, allows the vault to recover
 ↪  lost tokens.
23:     function rescueTokens(IERC20 token, address receiver, uint256 amount)
 ↪  external onlyVault {
24:         token.checkTransfer(receiver, amount);
25:     }
26:
```

However, it was found that none of the vaults could call the `rescueTokens` function. Thus, this feature is broken.

Impact

Medium. The rescue function is broken, and tokens cannot be rescued when needed, leading to assets being stuck in the contract.

Code Snippet

https://github.com/sherlock-audit/2024-06-leveraged-vaults/blob/main/leveraged-vaults-private/contracts/vaults/staking/protocols/ClonedCoolDownHolder.sol#L2

3

Tool used

Manual Review

Recommendation

Consider allowing the protocol admin to call the `rescueTokens` function directly, or update the implementation of vaults to allow the vault to call the `rescueTokens` function.

Discussion

sherlock-admin2

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

0xmystery commented:

> Lacks proof to substantiate the bug

xiaoming9090

Escalate.

This should be a valid issue. From the codebase, one can already see that the rescue function is broken. Since the rescue function is broken, if the protocol wants to rescue some tokens, it would not be able to, leading to a loss of assets.

sherlock-admin3

> Escalate.
>
> This should be a valid issue. From the codebase, one can already see that the rescue function is broken. Since the rescue function is broken, if the protocol wants to rescue some tokens, it would not be able to, leading to a loss of assets.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

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

lemonmon1984

This issue should be valid. Due to the lack of implementation, it's challenging to provide a proof of concept, but the issue is clear from the codebase. Additionally, GitHub issue #100 reports the same problem and should be duped into this one.

mystery0x

> Escalate.
>
> This should be a valid issue. From the codebase, one can already see that the rescue function is broken. Since the rescue function is broken, if the protocol wants to rescue some tokens, it would not be able to, leading to a loss of assets.

Will have the sponsors look into this finding. But it seems to me this function belongs to an abstract contract that's meant to be inherited by contracts needing to use it as determined by the protocol. For example, it's being inherited by Kelp.sol for whoever `_vault` to use it:

https://github.com/sherlock-audit/2024-06-leveraged-vaults/blob/main/leveraged-vaults-private/contracts/vaults/staking/protocols/Kelp.sol#L52-L55

```
contract KelpCooldownHolder is ClonedCoolDownHolder {
    bool public triggered = false;

    constructor(address _vault) ClonedCoolDownHolder(_vault) { }
```

Additionally, finding of this nature is rated low given that it's optionally needed. Also, all vaults are deemed out of scope for this contest, and hence should be informational IMO.

WangSecurity

> Also, all vaults are deemed out of scope for this contest

As I see not all vaults are OOS:

1. BaseStakingVault.sol
2. PendlePTEtherFiVault.sol
3. PendlePTKelpVault.sol
4. PendlePTStakedUSDeVault.sol

Hence, I would agree this report identifies an issue that breaks core contract functionality leading to a loss of funds. Planning to accept the escalation and validate with medium severity.

WangSecurity

Result: Medium Unique

sherlock-admin4

Escalations have been resolved successfully!

Escalation status:

- xiaoming9090: accepted

WangSecurity

@mystery0x I see that 100 is a duplicate, are there other duplicates?

mystery0x

> Lacks proof to substantiate the bug

Nope. #72 and #100 are the only two reports submitting this finding.

lemonmon1984

@mystery0x the labels on #100 hasn't been updated yet.

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/notional-finance/leveraged-vaults/pull/100

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-5: Protocol could be DOS by transfer error due to lack of code length check

Source: https://github.com/sherlock-audit/2024-06-leveraged-vaults-judging/issues/73

Found by

xiaoming90

Summary

The protocol could be DOS due to inadequate handling of transfer errors as it does not perform code length check. As a result, many critical features such as deposit, redemption, and liquidation will be broken, leading to assets being stuck or lost of assets.

Vulnerability Detail

Per the comment at Lines 312-315 below, the transfer error must be ignored to ensure that the error does not prevent normal vault operations from working. This is crucial because many critical features such as deposit, redemption, and liquidation will revert if the error is not handled properly, which could lead to assets being stuck or loss of assets.

https://github.com/sherlock-audit/2024-06-leveraged-vaults/blob/main/leveraged-vaults-private/contracts/vaults/common/VaultRewarderLib.sol#L316

```
File: VaultRewarderLib.sol
311:         if (0 < rewardToClaim) {
312:             // Ignore transfer errors here so that any strange failures
 ↪  here do not
313:             // prevent normal vault operations from working. Failures may
 ↪  include a
314:             // lack of balances or some sort of blacklist that prevents an
 ↪  account
315:             // from receiving tokens.
316:             try IEIP20NonStandard(rewardToken).transfer(account,
 ↪  rewardToClaim) {
317:                 bool success = TokenUtils.checkReturnCode();
318:                 if (success) {
319:                     emit VaultRewardTransfer(rewardToken, account,
 ↪  rewardToClaim);
320:                 } else {
321:                     emit VaultRewardTransfer(rewardToken, account, 0);
```

```
322:                 }
323:             // Emits zero tokens transferred if the transfer fails.
324:             } catch {
325:                 emit VaultRewardTransfer(rewardToken, account, 0);
326:             }
327:         }
```

Line 326 above attempts to mitigate the transfer error by "wrapping" the transfer call within the try-catch block. If the `transfer` function reverts, it will not revert the entire transaction and break the critical features of the protocol.

However, this approach was found to be insufficient to mitigate all cases of transfer error. There is still an edge case where an error could occur during transfer, reverting the entire transaction.

If the `rewardToken` points to an address that does not contain any code (codesize == 0), the transaction will revert instead of going into the try-catch block due to how Solidity works. It is possible that some reward tokens may contain self-destruct feature for certain reasons, resulting in the codesize becoming zero at some point in time.

Impact

If the edge case occurs, many critical features such as deposit, redemption, and liquidation will be broken, leading to assets being stuck or lost of assets.

Code Snippet

https://github.com/sherlock-audit/2024-06-leveraged-vaults/blob/main/leveraged-vaults-private/contracts/vaults/common/VaultRewarderLib.sol#L316

Tool used

Manual Review

Recommendation

Ensure that the transfer error does not revert the entire transaction under any circumstance. Consider implementing the following changes:

```
        if (0 < rewardToClaim) {
            // Ignore transfer errors here so that any strange failures here do
↪   not
            // prevent normal vault operations from working. Failures may
↪   include a
```

```
            // lack of balances or some sort of blacklist that prevents an
 ↪  account
            // from receiving tokens.
+           if (rewardToken.code.length > 0) {}
              try IEIP20NonStandard(rewardToken).transfer(account,
 ↪  rewardToClaim) {
                    bool success = TokenUtils.checkReturnCode();
                    if (success) {
                        emit VaultRewardTransfer(rewardToken, account,
 ↪  rewardToClaim);
                    } else {
                        emit VaultRewardTransfer(rewardToken, account, 0);
                    }
                // Emits zero tokens transferred if the transfer fails.
                } catch {
                    emit VaultRewardTransfer(rewardToken, account, 0);
                }
+           }
        }
```

Discussion

sherlock-admin4

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

0xmystery commented:

> Low/QA at most

xiaoming9090

Escalate.

This issue should be a Medium.

Although the probability of encountering such a reward token is low, it is still possible for such a scenario to occur (edge case). If it happens, the impact will be serious, causing many critical features such as deposit, redemption, and liquidation to be broken, leading to assets being stuck or lost of assets.

Per Sherlock's judging rules, issues that require certain external conditions or specific states that could lead to a loss of assets should be judged as Medium, which should be the case here:

1. Causes a loss of funds but **requires certain external conditions or specific states**, or a loss is highly constrained. The losses must exceed small, finite amount of funds, and any amount relevant based on the precision or significance of the loss.

sherlock-admin3

> Escalate.
>
> This issue should be a Medium.
>
> Although the probability of encountering such a reward token is low, it is still possible for such a scenario to occur (edge case). If it happens, the impact will be serious, causing many critical features such as deposit, redemption, and liquidation to be broken, leading to assets being stuck or lost of assets.
>
> Per Sherlock's judging rules, issues that require certain external conditions or specific states that could lead to a loss of assets should be judged as Medium, which should be the case here:
>
> 1. Causes a loss of funds but **requires certain external conditions or specific states**, or a loss is highly constrained. The losses must exceed small, finite amount of funds, and any amount relevant based on the precision or significance of the loss.

You've created a valid escalation!

To remove the escalation from consideration: Delete your comment.

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

mystery0x

> Escalate.
>
> This issue should be a Medium.
>
> Although the probability of encountering such a reward token is low, it is still possible for such a scenario to occur (edge case). If it happens, the impact will be serious, causing many critical features such as deposit, redemption, and liquidation to be broken, leading to assets being stuck or lost of assets.
>
> Per Sherlock's judging rules, issues that require certain external conditions or specific states that could lead to a loss of assets should be judged as Medium, which should be the case here:
>
> 1. Causes a loss of funds but **requires certain external conditions or specific states**, or a loss is highly constrained. The losses must exceed small, finite amount of funds, and any amount relevant based on the precision or significance of the loss.

Will have the sponsors look into this finding... but it seems to me that making sure `rewardToken.code.length != 0` is a low issue equivalent to sanity check on zero value/address.

Additionally, `0 < rewardToClaim` signifies that `rewardToken` is already intact assuredly having `rewardToken.code.length > 0`:

https://github.com/sherlock-audit/2024-06-leveraged-vaults/blob/main/leveraged-vaults-private/contracts/vaults/common/VaultRewarderLib.sol#L302-L304

```
rewardToClaim = _getRewardsToClaim(
    rewardToken, account, vaultSharesBefore, rewardsPerVaultShare
);
```

WangSecurity

The first question is, who sets the reward tokens? Notional admins, correct?

Secondly, if the reward token's contract was self-destructed, it wouldn't be even possible to withdraw such rewards. So with or without the try/catch block it leads to a loss of funds?

Thirdly, this destructed token can be replaced, correct?

Fourthly, I see code comments saying that the code should ignore reverts, but need to remark, that breaking statements from the code comments, doesn't mean the issue automatically warrants medium severity.

WangSecurity

@xiaoming9090 looking deeper into this issue, I'm unsure the reward tokens can be replaced, which leads to users not being able to get the reward tokens at all, correct? But it's only possible if the reward token has the `selfdestruct` opcode?

xiaoming9090

@WangSecurity

> looking deeper into this issue, I'm unsure the reward tokens can be replaced, which leads to users not being able to get the reward tokens at all, correct? But it's only possible if the reward token has the selfdestruct opcode?

As stated in the comments here or you may refer to the implementation of the updateRewardToken function, once a reward token is set, the address of the reward token can never be changed. Only the emission rate can be updated.

However, the main concern that I'm trying to highlight in this report is not the loss of reward tokens, but DOS or the breaking of the protocol's core functionality.

The sponsor went to great lengths to ensure that the token transfer would never revert under any circumstance. One has to understand why the sponsor makes such a great effort to do so and is even willing to forgo the reward tokens to ensure the continuity of the protocol. The reason is that if the token transfer reverts, for whatever reason, the entire protocol will be brick because the deposit, redeem, and liquidation (core) features rely on this claim reward function.

Thus, naturally, one would review whether the sponsor's current approach will cover all the edge cases. It was found that there is an edge case if a reward token self-destructs. The reward token's code will be empty, and the token transfer will revert, resulting in the deposit, redemption and liquidation (core) features being broken.

```
if (0 < rewardToClaim) {
    // Ignore transfer errors here so that any strange failures here do not
    // prevent normal vault operations from working. Failures may include a
    // lack of balances or some sort of blacklist that prevents an account
    // from receiving tokens.
    try IEIP20NonStandard(rewardToken).transfer(account, rewardToClaim) {
        bool success = TokenUtils.checkReturnCode();
        if (success) {
            emit VaultRewardTransfer(rewardToken, account, rewardToClaim);
        } else {
            emit VaultRewardTransfer(rewardToken, account, 0);
        }
    // Emits zero tokens transferred if the transfer fails.
    } catch {
        emit VaultRewardTransfer(rewardToken, account, 0);
    }
}
```

WangSecurity

Thank you for that clarification.

Agree with the escalation, planning to accept it with medium severity.

WangSecurity

Result: Medium Unique

sherlock-admin2

Escalations have been resolved successfully!

Escalation status:

- xiaoming9090: accepted

WangSecurity

@xiaoming9090 @mystery0x @brakeless-wtp are there any duplicates?

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/notional-finance/leveraged-vaults/pull/101

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Issue M-6: The _getValueOfWithdrawRequest function uses different methods for selecting assets in various vaults.

Source: https://github.com/sherlock-audit/2024-06-leveraged-vaults-judging/issues/80

Found by

ZeroTrust

Summary

The _getValueOfWithdrawRequest function uses different methods for selecting assets in various vaults.

Vulnerability Detail

```
 function _getValueOfWithdrawRequest(
        WithdrawRequest memory w, uint256 /* */
    ) internal override view returns (uint256) {
        uint256 tokenOutSY = getTokenOutSYForWithdrawRequest(w.requestId);
        // NOTE: in this vault the tokenOutSy is known to be weETH.
        (int256 weETHPrice, /* */) = TRADING_MODULE.getOraclePrice(TOKEN_OUT_SY,
↪   BORROW_TOKEN);
@>>         return (tokenOutSY * weETHPrice.toUint() * BORROW_PRECISION) /
             (WETH_PRECISION * Constants.EXCHANGE_RATE_PRECISION);
    }
```

In PendlePTEtherFiVault, weETH is used to withdraw eETH from the EtherFi protocol. Since eETH is still in a waiting period, the value of the withdrawal request is calculated using the price and quantity of weETH.

```
function _getValueOfWithdrawRequest(
        WithdrawRequest memory w,
        address borrowToken,
        uint256 borrowPrecision
    ) internal view returns (uint256) {
        address holder = address(uint160(w.requestId));

        uint256 expectedStETHAmount;
        if (KelpCooldownHolder(payable(holder)).triggered()) {
            uint256[] memory requestIds =
↪   LidoWithdraw.getWithdrawalRequests(holder);
```

```
            ILidoWithdraw.WithdrawalRequestStatus[] memory withdrawsStatus =
↪   LidoWithdraw.getWithdrawalStatus(requestIds);

            expectedStETHAmount = withdrawsStatus[0].amountOfStETH;
        } else {
@>>             (/* */, expectedStETHAmount, /* */, /* */) =
↪   WithdrawManager.getUserWithdrawalRequest(stETH, holder, 0);

        }

        (int256 stETHToBorrowRate, /* */) =
↪   Deployments.TRADING_MODULE.getOraclePrice(
            address(stETH), borrowToken
        );

        return (expectedStETHAmount * stETHToBorrowRate.toUint() *
↪   borrowPrecision) /
            (Constants.EXCHANGE_RATE_PRECISION * stETH_PRECISION);
    }
```

However, in PendlePTKelpVault, rsETH is used to withdraw stETH from the Kelp protocol. Similarly Since stETH is still in a waiting period, But the value of the withdrawal request is calculated using the expected amount and price of stETH(not rsETH).

```
   function _getValueOfWithdrawRequest(
        WithdrawRequest memory w,
        address borrowToken,
        uint256 borrowPrecision
    ) internal view returns (uint256) {
        address holder = address(uint160(w.requestId));
        // This valuation is the amount of USDe the account will receive at
↪   cooldown, once
        // a cooldown is initiated the account is no longer receiving sUSDe
↪   yield. This balance
        // of USDe is transferred to a Silo contract and guaranteed to be
↪   available once the
        // cooldown has passed.
        IsUSDe.UserCooldown memory userCooldown = sUSDe.cooldowns(holder);

        int256 usdeToBorrowRate;
        if (borrowToken == address(USDe)) {
            usdeToBorrowRate = int256(Constants.EXCHANGE_RATE_PRECISION);
        } else {
            // If not borrowing USDe, convert to the borrowed token
```

```
            (usdeToBorrowRate, /* */) =
↪   Deployments.TRADING_MODULE.getOraclePrice(
                address(USDe), borrowToken
            );
        }

@>>         return (userCooldown.underlyingAmount * usdeToBorrowRate.toUint() *
↪   borrowPrecision) /
            (Constants.EXCHANGE_RATE_PRECISION * USDE_PRECISION);
    }
```

Similarly, in PendlePTStakedUSDeVault, sUSDe is used to withdraw USDe from the Ethena protocol. Since USDe is still in a waiting period, the value of the withdrawal request is calculated using the expected amount and price of USDe(not sUSDe).

To summarize, PendlePTEtherFiVault uses the asset's value before redemption for the calculation, while the other two vaults use the expected asset value after redemption. One of these methods is incorrect. The incorrect calculation method affects the asset value assessment of users in the fund, which in turn impacts whether a user should be liquidated.

Impact

The incorrect calculation method affects the asset value assessment of users in the fund, which in turn impacts whether a user should be liquidated.

Code Snippet

https://github.com/sherlock-audit/2024-06-leveraged-vaults/blob/14d3eaf0445c251c52c86ce88a84a3f5b9dfad94/leveraged-vaults-private/contracts/vaults/staking/PendlePTEtherFiVault.sol#L46

https://github.com/sherlock-audit/2024-06-leveraged-vaults/blob/main/leveraged-vaults-private/contracts/vaults/staking/protocols/Kelp.sol#L114

https://github.com/sherlock-audit/2024-06-leveraged-vaults/blob/main/leveraged-vaults-private/contracts/vaults/staking/protocols/Ethena.sol#L77

Tool used

Manual Review

Recommendation

Choose a consistent approach for value assessment, either using the token before withdrawal or the token received after withdrawal.

Discussion

T-Woodward

The watson has correctly pointed out an inconsistency in the valuation methodology for withdrawal requests in these different vaults, but that inconsistency in and of itself is not a vulnerability.

He asserts that because we use two different methodologies, one must be wrong and one must be right. Therefore, he has shown that we have done something wrong with a critical part of the code and he deserves a bounty.

This line of reasoning is flawed. Neither approach is right, and neither is wrong. There is not an objectively correct way to value these withdrawal requests. For this to be a valid finding, we would need to see evidence that one of these valuation methodologies is actually exploitable / results in negative consequences. The watson has not shown that.

While consistency in this valuation methodology would be generally preferable, these are all different vaults with different assets that work different ways. The valuation methodology should match the problem at hand and should not just be consistent for the sake of consistency.

HOWEVER, having said all this, I looked further into the rsETH withdrawal request valuation and have concluded that we should change it because rsETH is slashable before it resolves to stETH, just like weETH. So in this case, we should switch the valuation methodology to match weETH. The sUSDe methodology still makes sense imo though because once the cooldown has started you know exactly how many USDe you will get and that won't change.

I'm not sure whether the watson deserves credit here. If we hadn't found the rsETH thing, I would have said no. But we did find the rsETH thing because of this issue report even though the watson didn't find it himself. Will let the judge decide.

T-Woodward

Think that if the watson does get a valid finding out of this, should be a medium severity at most. Once an account is in the withdrawal queue their account is locked until the withdrawal finalizes, so there's no way they could take advantage of a mispricing anyway.

mystery0x

Changing it to medium severity in light of the sponsor's detailed reasonings.

Hash01011122

@mystery0x isn't this dup of #60 as root cause is same? Correct me if I am wrong

mystery0x

> @mystery0x isn't this dup of #60 as root cause is same? Correct me if I am wrong

I think they are two different flaws/exploit stemming from different root cause. This one talking about inconsistencies of withdrawal request whereas #60 deals with decimals normalization/scaling issue.

Hash01011122

As far as I understand, this issue points out:

> PendlePTEtherFiVault uses the asset's value before redemption for the calculation, while the other two vaults use the expected asset value after redemption.
>
> > PendlePTStakedUSDeVault, sUSDe is used to withdraw USDe from the Ethena protocol. Since USDe is still in a waiting period, the value of the withdrawal request is calculated using the expected amount and price of USDe(not sUSDe).

Which is regarding different assets consideration while calculating the withdrawal.

While 60th issue is:

> `BT` be the borrowed token with 6 decimal precision, and `RT` be the redemption token with 18 decimal precision

Which is too about lack of consideration by protocol of using different assest tokens which lead to improper precision while calculating splitwithdrawal request.

Commonality: Incorrect implementation of before and after asset tokens.

I might be wrong here, but it would be better to recheck this two issues. @mystery0x, let me know your thoughts.

T-Woodward

In my opinion these are totally different issues

0502lian

> The watson has correctly pointed out an inconsistency in the valuation methodology for withdrawal requests in these different vaults, but that inconsistency in and of itself is not a vulnerability.
>
> He asserts that because we use two different methodologies, one must be wrong and one must be right. Therefore, he has shown that we have done something wrong with a critical part of the code and he deserves a bounty.
>
> This line of reasoning is flawed. Neither approach is right, and neither is wrong. There is not an objectively correct way to value these withdrawal

> requests. For this to be a valid finding, we would need to see evidence that one of these valuation methodologies is actually exploitable / results in negative consequences. The watson has not shown that.
>
> While consistency in this valuation methodology would be generally preferable, these are all different vaults with different assets that work different ways. The valuation methodology should match the problem at hand and should not just be consistent for the sake of consistency.
>
> HOWEVER, having said all this, I looked further into the rsETH withdrawal request valuation and have concluded that we should change it because rsETH is slashable before it resolves to stETH, just like weETH. So in this case, we should switch the valuation methodology to match weETH. The sUSDe methodology still makes sense imo though because once the cooldown has started you know exactly how many USDe you will get and that won't change.
>
> I'm not sure whether the watson deserves credit here. If we hadn't found the rsETH thing, I would have said no. But we did find the rsETH thing because of this issue report even though the watson didn't find it himself. Will let the judge decide.

Thanks to the sponsor for such a detailed explanation; what he said is factual. I would like to add:

- (1) I am indeed aware that the value of LSTs/LRTs tokens (including rsETH) can fluctuate, sometimes even dramatically, which is why I raised an issue113 about the fluctuation in the value of LSTs/LRTs tokens. The expectedAssetAmount is merely an expected value calculated based on the value of rsETH at the time of initiating a withdrawal, not the actual number of stETH finally withdrawn. It can not be used for the value of withdraw request. At that time, I should have included more detailed impacts of rsETH value fluctuations in this issue. Indeed, the Impact section in the report needs further enhancement.
- (2) Usually, during the Judge Contest phase, the Lead Judge for less clear issues will ask The Watson to provide a more detailed PoC. I don’t know why this Contest did not ask any Watson to provide one. If there had been such an opportunity, The Watson could have had the chance to add what they know.

I agree to let the Sherlock judge make the final decision.

sherlock-admin2

The protocol team fixed this issue in the following PRs/commits:
https://github.com/notional-finance/leveraged-vaults/pull/91

xiaoming9090

Additional change made https://github.com/notional-finance/leveraged-vaults/commit/de862c8fdb64d7e5f5b0eccc6807d6732f22fc99 and https://github.com/notional-finance/leveraged-vaults/pull/86/commits/3c5e13050e6bc7d278b401118c03bfed3a787d87

sherlock-admin2

The Lead Senior Watson signed off on the fix.

Disclaimers

Sherlock does not provide guarantees nor warranties relating to the security of the project.

Usage of all smart contract software is at the respective users' sole risk and is the users' responsibility.