



SHERLOCK

SHERLOCK SECURITY REVIEW FOR



Prepared for:

UXD

Prepared by:

Sherlock

Lead Security Expert:

0x52

Dates Audited:

January 2 - January 16, 2023

Prepared on:

February 7, 2023

Introduction

UXD Protocol is a fully collateralized decentralized stablecoin backed by delta-neutral position using derivatives.

Scope

- files under `contracts/`, except the tests `contracts/tests`

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
12	7

Issues not fixed or acknowledged

Medium	High
0	0

Security experts who found valid issues

[0x52](#)
[csanuragjain](#)
[ctf_sec](#)
[berndartmueller](#)
[hansfrieze](#)
[DecorativePineapple](#)
[rvierdiiev](#)
[HonorLt](#)
[0xNazgul](#)
[clems4ever](#)

[HollaDieWaldfee](#)
[cccZ](#)
[peanuts](#)
[OKage](#)
[GimelSec](#)
[yixxas](#)
[Bahurum](#)
[hl_](#)
[koxuan](#)
[keccak123](#)

[ck](#)
[jonatascm](#)
[minhtrng](#)
[zeroknots](#)
[wagmi](#)
[Jeiwan](#)
[duc](#)
[CRYP70](#)
[jprod15](#)
[Zarf](#)



Ruhum
unforgiven

kankodu
chiranz

dipp
carrot



Issue H-1: PerpDespository#reblance and rebalanceLite can be called to drain funds from anyone who has approved PerpDepository

Source: <https://github.com/sherlock-audit/2023-01-uxd-judging/issues/288>

Found by

clems4ever, dipp, 0xNazgul, DecorativePineapple, ctf_sec, unforgiven, 0x52, ck, HollaDieWaldfee, GimelSec, chiranz, berndartmueller, yixxas, Zarf, carrot, koxuan, hl_, Ruhum, kankodu, Bahurum

Summary

PerpDespository#reblance and rebalanceLite allows anyone to specify the account that pays the quote token. These functions allow a malicious user to abuse any allowance provided to PerpDirectory. rebalance is the worst of the two because the malicious user could sandwich attack the rebalance to steal all the funds and force the unsuspecting user to pay the shortfall.

Vulnerability Detail

```
function rebalance(
    uint256 amount,
    uint256 amountOutMinimum,
    uint160 sqrtPriceLimitX96,
    uint24 swapPoolFee,
    int8 polarity,
    address account // @audit user specified payer
) external nonReentrant returns (uint256, uint256) {
    if (polarity == -1) {
        return
            _rebalanceNegativePnlWithSwap(
                amount,
                amountOutMinimum,
                sqrtPriceLimitX96,
                swapPoolFee,
                account // @audit user address passed directly
            );
    } else if (polarity == 1) {
        // disable rebalancing positive PnL
        revert PositivePnlRebalanceDisabled(msg.sender);
        // return _rebalancePositivePnlWithSwap(amount, amountOutMinimum,
        ↪ sqrtPriceLimitX96, swapPoolFee, account);
    } else {
```



```

        revert InvalidRebalance(polarity);
    }
}

```

rebalance is an unpermissioned function that allows anyone to call and rebalance the PNL of the depository. It allows the caller to specify the an account that passes directly through to `_rebalanceNegativePnlWithSwap`

```

function _rebalanceNegativePnlWithSwap(
    uint256 amount,
    uint256 amountOutMinimum,
    uint160 sqrtPriceLimitX96,
    uint24 swapPoolFee,
    address account
) private returns (uint256, uint256) {
    ...
    // @audit this uses user supplied swap parameters which can be malicious
    SwapParams memory params = SwapParams({
        tokenIn: assetToken,
        tokenOut: quoteToken,
        amountIn: baseAmount,
        amountOutMinimum: amountOutMinimum,
        sqrtPriceLimitX96: sqrtPriceLimitX96,
        poolFee: swapPoolFee
    });
    uint256 quoteAmountOut = spotSwapper.swapExactInput(params);
    int256 shortFall = int256(
        quoteAmount.fromDecimalToDecimal(18, ERC20(quoteToken).decimals())
    ) - int256(quoteAmountOut);
    if (shortFall > 0) {
        // @audit shortfall is taken from account specified by user
        IERC20(quoteToken).transferFrom(
            account,
            address(this),
            uint256(shortFall)
        );
    } else if (shortFall < 0) {
        ...
    }
    vault.deposit(quoteToken, quoteAmount);

    emit Rebalanced(baseAmount, quoteAmount, shortFall);
    return (baseAmount, quoteAmount);
}

```

`_rebalanceNegativePnlWithSwap` uses both user specified swap parameters and takes the shortfall from the account specified by the user. This is where the



function can be abused to steal funds from any user that sets an allowance for this contract. A malicious user can sandwich attack the swap and specify malicious swap parameters to allow them to steal the entire rebalance. This creates a large shortfall which will be taken from the account that they specify, effectively stealing the funds from the user.

Example: Any account that gives the depository allowance can be stolen from. Imagine the following scenario. The multisig is going to rebalance the contract for 15000 USDC worth of ETH and based on current market conditions they are estimating that there will be a 1000 USDC shortfall because of the difference between the perpetual and spot prices (divergences between spot and perpetual price are common in trending markets). They first approve the depository for 1000 USDC. A malicious user sees this approval and immediately submits a transaction of their own. They request to rebalance only 1000 USDC worth of ETH and sandwich attack the swap to steal the rebalance. They specify the multisig as `account` and force it to pay the 1000 USDC shortfall and burn their entire allowance, stealing the USDC.

Impact

Anyone that gives the depository allowance can easily have their entire allowance stolen

Code Snippet

<https://github.com/sherlock-audit/2023-01-uxd/blob/main/contracts/integrations/perp/PerpDepository.sol#L478-L528>

Tool used

Manual Review

Recommendation

`PerpDespository#reblance` and `rebalanceLite` should use `msg.sender` instead of `account`:

```
function rebalance(
    uint256 amount,
    uint256 amountOutMinimum,
    uint160 sqrtPriceLimitX96,
    uint24 swapPoolFee,
    int8 polarity,
-   address account
) external nonReentrant returns (uint256, uint256) {
    if (polarity == -1) {
```



```

        return
            _rebalanceNegativePnlWithSwap(
                amount,
                amountOutMinimum,
                sqrtPriceLimitX96,
                swapPoolFee,
                account
            );
    } else if (polarity == 1) {
        // disable rebalancing positive PnL
        revert PositivePnlRebalanceDisabled(msg.sender);
        // return _rebalancePositivePnlWithSwap(amount, amountOutMinimum,
        //     sqrtPriceLimitX96, swapPoolFee, account);
    } else {
        revert InvalidRebalance(polarity);
    }
}

```



Issue H-2: Malicious user can use an excessively large `_toAddress` in `OFTCore#sendFrom` to break layerZero communication

Source: <https://github.com/sherlock-audit/2023-01-uxd-judging/issues/270>

Found by

0x52

Summary

By default layerZero implements a blocking behavior, that is, that each message must be processed and succeed in the order that it was sent. In order to circumvent this behavior the receiver must implement their own try-catch pattern. If the try-catch pattern in the receiving app ever fails then it will revert to its blocking behavior. The `_toAddress` input to `OFTCore#sendFrom` is calldata of any arbitrary length. An attacker can abuse this and submit a send request with an excessively large `_toAddress` to break communication between network with different gas limits.

Vulnerability Detail

```
function sendFrom(address _from, uint16 _dstChainId, bytes calldata _toAddress,
    ↪ uint _amount, address payable _refundAddress, address _zroPaymentAddress,
    ↪ bytes calldata _adapterParams) public payable virtual override {
    _send(_from, _dstChainId, _toAddress, _amount, _refundAddress,
        ↪ _zroPaymentAddress, _adapterParams);
}
```

The `_toAddress` input to `OFTCore#sendFrom` is a bytes calldata of any arbitrary size. This can be used as follows to break communication between chains that have different block gas limits.

Example: Let's say that an attacker wishes to permanently block the channel Arbitrum -> Optimism. Arbitrum has a massive gas block limit, much higher than Optimism's 20M block gas limit. The attacker would call `sendFrom` on the Arbitrum chain with the Optimism chain as the destination. For the `_toAddress` input they would use an absolutely massive amount of bytes. This would be packed into the payload which would be called on Optimism. Since Arbitrum has a huge gas limit the transaction would send from the Arbitrum side but it would be so big that the transaction could never succeed on the Optimism side due to gas constraints. Since that nonce can never succeed the communication channel will be permanently blocked at the Optimism endpoint, bypassing the nonblocking



behavior implemented in the OFT design and reverting to the default blocking behavior of layerZero.

Users can still send messages and burn their tokens from Arbitrum -> Optimism but the messages can never be received. This could be done between any two chain in which one has a higher block gas limit. This would cause massive loss of funds and completely cripple the entire protocol.

Impact

Massive loss of user funds and protocol completely crippled

Code Snippet

<https://github.com/sherlock-audit/2023-01-uxd/blob/main/contracts/external/layer-zero/token/oft/OFTCore.sol#L31-L33>

Tool used

Manual Review

Recommendation

Limit the length of `_toAddress` to some amount (i.e. 256 bytes) as of right now EVM uses 20 bytes address and Sol/Aptos use 32 bytes address, so for right now it could be limited to 32 bytes.

```
function sendFrom(address _from, uint16 _dstChainId, bytes calldata
    ↪ _toAddress, uint _amount, address payable _refundAddress, address
    ↪ _zroPaymentAddress, bytes calldata _adapterParams) public payable
    ↪ virtual override {
+   require(_toAddress.length <= maxAddressLength);
    _send(_from, _dstChainId, _toAddress, _amount, _refundAddress,
        ↪ _zroPaymentAddress, _adapterParams);
}
```

Discussion

WarTech9

We are using a non blocking receiver with both UXP and UXD tokens inheriting from the `OFTV2Core` base contract. Thus when a message is received `nonBlockingLzReceive` function is called which does not block the channel between 2 chains if an error occurs.

Please provide more details if you feel otherwise.



0x00052

Correct you do use a non-blocking but using a huge `_toAddress` will cause the function to revert before the non-blocking error handling.

```
function receivePayload(uint16 _srcChainId, bytes calldata _srcAddress, address
↳ _dstAddress, uint64 _nonce, uint _gasLimit, bytes calldata _payload)
↳ external override receiveNonReentrant {
    // assert and increment the nonce. no message shuffling
    require(_nonce == ++inboundNonce[_srcChainId][_srcAddress], "LayerZero:
↳ wrong nonce");

    LibraryConfig storage uaConfig = uaConfigLookup[_dstAddress];

    // authentication to prevent cross-version message validation
    // protects against a malicious library from passing arbitrary data
    if (uaConfig.receiveVersion == DEFAULT_VERSION) {
        require(defaultReceiveLibraryAddress == msg.sender, "LayerZero: invalid
↳ default library");
    } else {
        require(uaConfig.receiveLibraryAddress == msg.sender, "LayerZero:
↳ invalid library");
    }

    // block if any message blocking
    StoredPayload storage sp = storedPayload[_srcChainId][_srcAddress];
    require(sp.payloadHash == bytes32(0), "LayerZero: in message blocking");

    try ILayerZeroReceiver(_dstAddress).lzReceive{gas: _gasLimit}(_srcChainId,
↳ _srcAddress, _nonce, _payload) {
        // success, do nothing, end of the message delivery
    } catch (bytes memory reason) {
        // revert nonce if any uncaught errors/exceptions if the ua chooses the
↳ blocking mode
        storedPayload[_srcChainId][_srcAddress] =
↳ StoredPayload(uint64(_payload.length), _dstAddress,
↳ keccak256(_payload));
        emit PayloadStored(_srcChainId, _srcAddress, _dstAddress, _nonce,
↳ _payload, reason);
    }
}
```

Above is the code run on the endpoint. It will use the try statement to call the `lzReceive` on the non-blocking app. Due to the huge amount of calldata the try function will immediately revert because it will run out of gas. This bypasses all the logic of the app and causes the checkpoint to store the payload, blocking the channel



hrishibhat

@WarTech9

WarTech9

The message is only stored if the `catch` block is executed. In your example, if a huge amount of data is passed in the receive function would run out of gas before that gets executed. Secondly, the transaction would revert within the `catch` block itself if the payload is too huge as a result of a huge `_dstAddress` being passed in, thus, message would not be stored and channel would not be blocked.

WarTech9

There could be some value in adding input size validation at source as recommended, since the size could be arbitrary. But this issue is medium/low risk

0x00052

```
// assert and increment the nonce. no message shuffling
require(_nonce == ++inboundNonce[_srcChainId][_srcAddress], "LayerZero: wrong
↳ nonce");
```

This line in endpoint requires that the nonce is sequential so if the message is too big to be executed at all then it will also block the channel even without being stored. As an example if the message with `nonce == 1` cannot be executed at all due to gas limits then the channel would also be blocked because trying to execute `nonce == 2` would always revert at that statement.

```
function send(uint16 _dstChainId, bytes calldata _destination, bytes calldata
↳ _payload, address payable _refundAddress, address _zroPaymentAddress, bytes
↳ calldata _adapterParams) external payable override sendNonReentrant {
    LibraryConfig storage uaConfig = uaConfigLookup[msg.sender];
    uint64 nonce = ++outboundNonce[_dstChainId][msg.sender];
    _getSendLibrary(uaConfig).send{value: msg.value}(msg.sender, nonce,
↳ _dstChainId, _destination, _payload, _refundAddress, _zroPaymentAddress,
↳ _adapterParams);
}
```

The nonce of the message is set on the sending end of the endpoint so it's impossible to work around a nonce that can't execute.

Secondly, the transaction would revert within the `catch` block itself if the payload is too huge as a result of a huge `_dstAddress` being passed in, thus, message would not be stored and channel would not be blocked.

This is not true either because the `_dstAddress` is not the same as `_toAddress` passed into the `OFTCore#sendFrom`. `_toAddress` is packed into the payload of the message and the `dstAddress` is set by `_lzSend`:



```
function _lzSend(uint16 _dstChainId, bytes memory _payload, address payable
↳ _refundAddress, address _zroPaymentAddress, bytes memory _adapterParams,
↳ uint _nativeFee) internal virtual {
    bytes memory trustedRemote = trustedRemoteLookup[_dstChainId];
    require(trustedRemote.length != 0, "LzApp: destination chain is not a
↳ trusted source");
    lzEndpoint.send{value: _nativeFee}(_dstChainId, trustedRemote //@audit
↳ _dstAddress, _payload, _refundAddress, _zroPaymentAddress,
↳ _adapterParams);
}
```

The destination address would be the address of the receiving contract (normal sized). Additionally when the message is stored, it only stores the hash of the message so it doesn't require a huge amount of gas to store.

hrishibhat

Considering this a valid high issue based on the above comments.

Oxlryna

LayerZero team is here. Thank you for digging into this! We have a max payload size (10000) set in our RelayerV2 contract which prevents the described situation (the contract isn't verified, but you can fork a chain and test). If a payload exceeds the specified limit the transaction will revert on the source. If a client application is configured to use a non-default Relayer it must set the payload limit itself. Generally, we believe it's a protocol's responsibility to enforce the max payload size.

berndartmueller

Escalate for 10 USDC

Consider the response from the LayerZero team above to reconsider the severity and validity of this issue. If UXD uses the RelayerV2 contract provided by LayerZero, this is a non-issue.

sherlock-admin

Escalate for 10 USDC

Consider the response from the LayerZero team above to reconsider the severity and validity of this issue. If UXD uses the RelayerV2 contract provided by LayerZero, this is a non-issue.

You've created a valid escalation for 10 USDC!

To remove the escalation from consideration: Delete your comment. To change the amount you've staked on this escalation: Edit your comment **(do not create a new comment)**.



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

rvierdiyev

Escalate for 15 usdc. Increased amount just to make sure this will be rechecked.

sherlock-admin

Escalate for 15 usdc. Increased amount just to make sure this will be rechecked.

You've created a valid escalation for 15 USDC!

To remove the escalation from consideration: Delete your comment. To change the amount you've staked on this escalation: Edit your comment **(do not create a new comment)**.

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

hrishibhat

Escalation rejected

Comment from Layer Zero shows that there is a valid risk from payload which is now fixed: <https://github.com/LayerZero-Labs/LayerZero/pull/24/commits> The relayer did not seem have a check for max payload size previously.

Additionally as suggested by the LZ

we believe it's a protocol's responsibility to enforce the max payload size.

sherlock-admin

Escalation rejected

Comment from Layer Zero shows that there is a valid risk from payload which is now fixed:

<https://github.com/LayerZero-Labs/LayerZero/pull/24/commits> The relayer did not seem have a check for max payload size previously.

Additionally as suggested by the LZ

we believe it's a protocol's responsibility to enforce the max payload size.

This issue's escalations have been rejected!

Watsons who escalated this issue will have their escalation amount deducted from their next payout.



Issue H-3: RageTrade senior vault USDC deposits are subject to utilization caps which can lock deposits for long periods of time leading to UXD instability

Source: <https://github.com/sherlock-audit/2023-01-uxd-judging/issues/253>

Found by

clems4ever, ctf_sec, 0x52, 0xNazgul

Summary

RageTrade senior vault requires that it maintains deposits above and beyond the current amount loaned to the junior vault. Currently this is set at 90%, that is the vault must maintain at least 10% more deposits than loans. Currently the junior vault is in high demand and very little can be withdrawn from the senior vault. A situation like this is far from ideal because in the even that there is a strong depeg of UXD a large portion of the collateral could be locked in the vault unable to be withdrawn.

Vulnerability Detail

DnGmxSeniorVault.sol

```
function beforeWithdraw(
    uint256 assets,
    uint256,
    address
) internal override {
    /// @dev withdrawal will fail if the utilization goes above maxUtilization
    ↳ value due to a withdrawal
    // totalUsdcBorrowed will reduce when borrower (junior vault) repays
    if (totalUsdcBorrowed() > ((totalAssets() - assets) * maxUtilizationBps) /
        ↳ MAX_BPS)
        revert MaxUtilizationBreached();

    // take out required assets from aave lending pool
    pool.withdraw(address(asset), assets, address(this));
}
```

DnGmxSeniorVault.sol#beforeWithdraw is called before each withdraw and will revert if the withdraw lowers the utilization of the vault below a certain threshold. This is problematic in the event that large deposits are required to maintain the stability of UXD.



Impact

UXD may become destabilized in the event that the senior vault has high utilization and the collateral is inaccessible

Code Snippet

<https://github.com/sherlock-audit/2023-01-uxd/blob/main/contracts/integrations/rage-trade/RageDnDepository.sol#L99-L115>

Tool used

Manual Review

Recommendation

I recommend three safeguards against this:

- 1) Monitor the current utilization of the senior vault and limit deposits if utilization is close to locking positions
- 2) Maintain a portion of the USDC deposits outside the vault (i.e. 10%) to avoid sudden potential liquidity crunches
- 3) Create functions to balance the proportions of USDC in and out of the vault to withdraw USDC from the vault in the event that utilization threatens to lock collateral

Discussion

WarTech9

Possible usecase for insurance fund.

acamill

This is the main downside of using not fully liquid strategies for the ALM model, upside being higher yield. We can mitigate this issue with buffers but that's always an issue, and adding buffers with either protocol funds or insurance fund is equivalent to using lower yield strategies, as such not an ideal solution either. (and it add complexity)

My personal opinion is to keep the cap on the illiquid strategy to be low enough relative to the total circulating UXD, that way keeping the high yield but reducing the liquidity crunch issue. That's what we are currently doing on Solana, working on smarter rebalancing and better risk management to keep these cap relevant.



Issue H-4: PerpDepository has no way to withdraw profits depriving stakers of profits owed

Source: <https://github.com/sherlock-audit/2023-01-uxd-judging/issues/251>

Found by

0x52

Summary

PerpDepository has no way to calculate or withdraw any profits made by the vault. By design stakes are entitled to a portion of the profits generated by the delta-neutral strategy. The issue is that the vault never implements a way to withdraw profits to stakers, resulting in loss of revenue for them.

Vulnerability Detail

See summary.

Impact

Profits owed stakers will be trapped in the contract and they will lose that portion of their revenue

Code Snippet

<https://github.com/sherlock-audit/2023-01-uxd/blob/main/contracts/integrations/perp/PerpDepository.sol#L25>

Tool used

Manual Review

Recommendation

Create a function to calculate and withdraw protocol profit to be awarded to stakers

Discussion

WarTech9

Profits on PerpDepository are currently locked in the depository and can be unlocked in future updates through positive PnL rebalancing. RageDepository



profits are locked in that contract and can be withdrawn by the contract owner (governance) through the `withdrawProfits()` function

rvierdiyev

Escalate for 11 USDC.

This is not a vulnerability. As @WarTech9 said,

RageDepository profits are locked in that contract and can be withdrawn by the contract owner (governance) through the `withdrawProfits()` function

This is design choice of protocol. I have discussed that during the contest with @WarTech9, so he explained me that and i didn't submit that as issue. As he said, that rewards are not lost and governance can withdraw it later to reuse.

Also author of report states in impact section that:

Profits owed stakers will be trapped in the contract and they will lose that portion of their revenue

But this is not true, as i described above, that value is not stucked as protocol can withdraw it.

sherlock-admin

Escalate for 11 USDC.

This is not a vulnerability. As @WarTech9 said,

RageDepository profits are locked in that contract and can be withdrawn by the contract owner (governance) through the `withdrawProfits()` function

This is design choice of protocol. I have discussed that during the contest with @WarTech9, so he explained me that and i didn't submit that as issue. As he said, that rewards are not lost and governance can withdraw it later to reuse.

Also author of report states in impact section that:

Profits owed stakers will be trapped in the contract and they will lose that portion of their revenue

But this is not true, as i described above, that value is not stucked as protocol can withdraw it.

You've created a valid escalation for 11 USDC!

To remove the escalation from consideration: Delete your comment. To change the amount you've staked on this escalation: Edit your comment **(do not create a new comment)**.



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

hrishibhat

Escalation rejected

This is in PerpDepository and not RageDepository. Profits in PerpDepository are stuck with no method to unlock them except while upgrading the contract. Although contract upgrades are to be used for feature updates and add changes to handle extreme situations, this issue is clearly a smart contract bug where funds are locked.

sherlock-admin

Escalation rejected

This is in PerpDepository and not RageDepository. Profits in PerpDepository are stuck with no method to unlock them except while upgrading the contract. Although contract upgrades are to be used for feature updates and add changes to handle extreme situations, this issue is clearly a smart contract bug where funds are locked.

This issue's escalations have been rejected!

Watsons who escalated this issue will have their escalation amount deducted from their next payout.



Issue H-5: USDC deposited to PerpDepository.sol are irretrievable and effectively causes UDX to become undercollateralized

Source: <https://github.com/sherlock-audit/2023-01-uxd-judging/issues/250>

Found by

csanuragjain, 0x52

Summary

PerpDepository rebalances negative PNL into USDC holdings. This preserves the delta neutrality of the system by exchanging base to quote. This is problematic though as once it is in the vault as USDC it can never be withdrawn. The effect is that the delta neutral position can never be liquidated but the USDC is inaccessible so UDX is effectively undercollateralized.

Vulnerability Detail

`_processQuoteMint`, `_rebalanceNegativePnlWithSwap` and `_rebalanceNegativePnlLite` all add USDC collateral to the system. There were originally two ways in which USDC could be removed from the system. The first was positive PNL rebalancing, which has now been deactivated. The second is for the owner to remove the USDC via `withdrawInsurance`.

```
function withdrawInsurance(uint256 amount, address to)
    external
    nonReentrant
    onlyOwner
{
    if (amount == 0) {
        revert ZeroAmount();
    }

    insuranceDeposited -= amount;

    vault.withdraw(insuranceToken(), amount);
    IERC20(insuranceToken()).transfer(to, amount);

    emit InsuranceWithdrawn(msg.sender, to, amount);
}
```

The issue is that `withdrawInsurance` cannot actually redeem any USDC. Since `insuranceDeposited` is a `uint256` and is decremented by the `withdraw`, it is



impossible for more USDC to be withdrawn than was originally deposited.

The result is that there is no way for the USDC to ever be redeemed and therefore over time will lead to the system becoming undercollateralized due to its inaccessibility.

Impact

UDX will become undercollateralized and the ecosystem will spiral out of control

Code Snippet

<https://github.com/sherlock-audit/2023-01-uxd/blob/main/contracts/integrations/PerpDepository.sol#L478-L528>

<https://github.com/sherlock-audit/2023-01-uxd/blob/main/contracts/integrations/PerpDepository.sol#L615-L644>

<https://github.com/sherlock-audit/2023-01-uxd/blob/main/contracts/integrations/PerpDepository.sol#L385-L397>

Tool used

Manual Review

Recommendation

Allow all USDC now deposited into the insurance fund to be redeemed 1:1



Issue H-6: PerpDepository#getPositionValue uses incorrect value for TWAP interval allowing more than intended funds to be extracted

Source: <https://github.com/sherlock-audit/2023-01-uxd-judging/issues/249>

Found by

HonorLt, 0x52, berndartmueller, OKage, cccz, DecorativePineapple, ctf_sec

Summary

PerpDepository#getPositionValue queries the exchange for the mark price to calculate the unrealized PNL. Mark price is defined as the 15 minute TWAP of the market. The issue is that it uses the 15 second TWAP instead of the 15 minute TWAP

Vulnerability Detail

As stated in the docs and as implemented in the ClearHouseConfig contract, the mark price is a 15 minute / 900 second TWAP.

```
function getPositionValue() public view returns (uint256) {
    uint256 markPrice = getMarkPriceTwap(15);
    int256 positionSize = IAccountBalance(clearingHouse.getAccountBalance())
        .getTakerPositionSize(address(this), market);
    return markPrice.mulWadUp(_abs(positionSize));
}

function getMarkPriceTwap(uint32 twapInterval)
    public
    view
    returns (uint256)
{
    IExchange exchange = IExchange(clearingHouse.getExchange());
    uint256 markPrice = exchange
        .getSqrtMarkTwapX96(market, twapInterval)
        .formatSqrtPriceX96ToPriceX96()
        .formatX96ToX10_18();
    return markPrice;
}
```

As seen in the code above getPositionValue uses 15 as the TWAP interval. This means it is pulling a 15 second TWAP rather than a 15 minute TWAP as intended.



Impact

The mark price and by extension the position value will frequently be different from true mark price of the market allowing for larger rebalances than should be possible.

Code Snippet

<https://github.com/sherlock-audit/2023-01-uxd/blob/main/contracts/integrations/erp/PerpDepository.sol#L708-L713>

Tool used

Manual Review

Recommendation

I recommend pulling the TWAP fresh each time from ClearingHouseConfig, because the TWAP can be changed at anytime. If it is desired to make it a constant then it should at least be changed from 15 to 900.



Issue H-7: User specified slippage allows frontrunning

Source: <https://github.com/sherlock-audit/2023-01-uxd-judging/issues/192>

Found by

HonorLt, minhtrng, jonatascm, koxuan, yixxas, wagmi, ck, HollaDieWaldfee, zeroknots, GimelSec, peanuts, keccak123

Summary

`rebalance` and `rebalanceLite` can be called by any user. Assets are taken from a user specified `account` address which has approved `PerpDepository`. If an address has a non-zero approval for `PerpDepository`, a frontrunner can use `rebalance` to transfer funds and profit by sandwiching the Uniswap pool swap.

Vulnerability Detail

When `mint` or `redeem` is called in `UXDController`, `msg.sender` is where the value is coming from. But `rebalance` allows for the caller to specify the account where funds are coming from. This means `msg.sender` can be any address. This allows for different scenarios where a frontrunner can profit with these steps.

1. a frontrunner detects a call of `rebalance` transaction in the mempool for a certain account address
2. the frontrunner duplicates the transaction but increases the gas amount (to allow frontrunning the original transaction) and changes the `amountOutMinimum` value to zero
3. the frontrunner can profit by sandwiching the Uniswap swap which now has no slippage setting
4. The user will lose value

Impact

An account that is used in `rebalance` can lose value

Code Snippet

`rebalance` can be frontrun <https://github.com/sherlock-audit/2023-01-uxd/blob/main/contracts/integrations/perp/PerpDepository.sol#L446>

Tool used

Manual Review



Recommendation

`rebalance` and `rebalanceLite` should use `msg.sender` to replace the function argument account address.

Discussion

WarTech9

This is a duplicate of #288



Issue M-1: `rebalanceLite` should provide a slippage protection

Source: <https://github.com/sherlock-audit/2023-01-uxd-judging/issues/429>

Found by

HollaDieWaldfee, hansfrieze

Summary

Users can lose funds while rebalancing.

Vulnerability Detail

The protocol provides two kinds of rebalancing functions - `rebalance()` and `rebalanceLite()`. While the function `rebalance()` is protected from an unintended slippage because the caller can specify `amountOutMinimum`, `rebalanceLite()` does not have this protection. This makes the user vulnerable to unintended slippage due to various scenarios.

```
PerpDepository.sol
597:     function rebalanceLite(
598:         uint256 amount,
599:         int8 polarity,
600:         uint160 sqrtPriceLimitX96,
601:         address account
602:     ) external nonReentrant returns (uint256, uint256) {
603:         if (polarity == -1) {
604:             return
605:                 _rebalanceNegativePnlLite(amount, sqrtPriceLimitX96,
↳ account);
606:         } else if (polarity == 1) {
607:             // disable rebalancing positive PnL
608:             revert PositivePnlRebalanceDisabled(msg.sender);
609:             // return _rebalancePositivePnlLite(amount, sqrtPriceLimitX96,
↳ account);
610:         } else {
611:             revert InvalidRebalance(polarity);
612:         }
613:     }
614:
615:     function _rebalanceNegativePnlLite(
616:         uint256 amount,
617:         uint160 sqrtPriceLimitX96,
618:         address account
```



```

619:    ) private returns (uint256, uint256) {
620:        uint256 normalizedAmount = amount.fromDecimalToDecimal(
621:            ERC20(quoteToken).decimals(),
622:            18
623:        );
624:
625:        _checkNegativePnl(normalizedAmount);
626:        IERC20(quoteToken).transferFrom(account, address(this), amount);
627:        IERC20(quoteToken).approve(address(vault), amount);
628:        vault.deposit(quoteToken, amount);
629:
630:        bool isShort = false;
631:        bool amountIsInput = true;
632:        (uint256 baseAmount, uint256 quoteAmount) = _placePerpOrder(
633:            normalizedAmount,
634:            isShort,
635:            amountIsInput,
636:            sqrtPriceLimitX96
637:        );
638:        vault.withdraw(assetToken, baseAmount);
639:        IERC20(assetToken).transfer(account, baseAmount);
640:
641:        emit Rebalanced(baseAmount, quoteAmount, 0);
642:
643:        return (baseAmount, quoteAmount);
644:    }

```

Especially, according to the communication with the PERP dev team, it is possible for the Perp's ClearingHouse to fill the position partially when the price limit is specified (`sqrtPriceLimitX96`). It is also commented in the Perp contract comments [here](#).

```

63:    /// @param sqrtPriceLimitX96 tx will fill until it reaches this price
64:    ↪ but WON'T REVERT
65:    struct InternalOpenPositionParams {
66:        address trader;
67:        address baseToken;
68:        bool isBaseToQuote;
69:        bool isExactInput;
70:        bool isClose;
71:        uint256 amount;
72:        uint160 sqrtPriceLimitX96;
73:    }

```

So it is possible that the order is not placed to the full amount. As we can see in the #L626~#L628, the UXD protocol grabs the quote token of `amount` and deposits to the Perp's vault. And the unused amount will remain in the Perp vault while this is



supposed to be returned to the user who called this rebalance function.

Impact

Users can lose funds while lite rebalancing.

Code Snippet

<https://github.com/sherlock-audit/2023-01-uxd/blob/main/contracts/integrations/perp/PerpDepository.sol#L597>

Tool used

Manual Review

Recommendation

Add a protection parameter to the function `rebalanceLite()` so that the user can specify the minimum out amount.

Discussion

WarTech9

The caller can specify the target price using the `sqrtpPriceLimitX96` parameter to the `rebalanceLite` function. This offers slippage protection.

hansfriesse

Escalate for 10 USDC

I suggest the judge and the sponsor read this issue carefully again. The key problem is that the Perp protocol can partially fill the position, especially when the `sqrtpPriceLimitX96` is specified. (This is related to how Uniswap works, check [here](#)) So it is possible that the order is not placed to the full amount and the remaining amount should be returned to the user. I admit that my explanation sounded vague because I mentioned slippage. I mean, the protocol should return the remaining value or allow the user to explicitly specify the minimum output amount. Please check the screenshot of the chat I had with the Perp team. They confirmed it is possible that the order is not filled to the full amount when the `sqrtpPriceLimitX96` is specified.



<https://user-images.githubusercontent.com/4553148/216355394/37153565923174501c28536cbb>



sherlock-admin

Escalate for 10 USDC

I suggest the judge and the sponsor read this issue carefully again. The key problem is that the Perp protocol can partially fill the position, especially when the `sqrPriceLimitX96` is specified. (This is related to how Uniswap works, check [here](#)) So it is possible that the order is not placed to the full amount and the remaining amount should be returned to the user. I admit that my explanation sounded vague because I mentioned slippage. I mean, the protocol should return the remaining value or allow the user to explicitly specify the minimum output amount. Please check the screenshot of the chat I had with the Perp team. They confirmed it is possible that the order is not filled to the full amount when the `sqrPriceLimitX96` is specified.



You've created a valid escalation for 10 USDC!

To remove the escalation from consideration: Delete your comment. To change the amount you've staked on this escalation: Edit your comment **(do not create a new comment)**.

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

WarTech9

With the updated comments I agree there is a valid issue here. If the requested amount is not filled completely in the Perp order, we should only transfer from the user the amount returned from the `_placePerpOrder()` call.

hrishibhat

Escalation accepted.

As pointed out in the Escalation that there is the possibility of partial order fills when `sqrPriceLimitX96` is specified.

sherlock-admin

Escalation accepted.

As pointed out in the Escalation that there is the possibility of partial order fills when `sqrPriceLimitX96` is specified.

This issue's escalations have been accepted!



Contestants' payouts and scores will be updated according to the changes made on this issue.



Issue M-2: PerpDepository._rebalanceNegativePnlWithSwap() shouldn't use a sqrtPriceLimitX96 twice.

Source: <https://github.com/sherlock-audit/2023-01-uxd-judging/issues/425>

Found by

hansfries

Summary

PerpDepository._rebalanceNegativePnlWithSwap() shouldn't use a sqrtPriceLimitX96 twice.

Vulnerability Detail

Currently, _rebalanceNegativePnlWithSwap() uses a sqrtPriceLimitX96 param twice for placing a perp order and swapping.

```
function _rebalanceNegativePnlWithSwap(
    uint256 amount,
    uint256 amountOutMinimum,
    uint160 sqrtPriceLimitX96,
    uint24 swapPoolFee,
    address account
) private returns (uint256, uint256) {
    uint256 normalizedAmount = amount.fromDecimalToDecimal(
        ERC20(quoteToken).decimals(),
        18
    );
    _checkNegativePnl(normalizedAmount);
    bool isShort = false;
    bool amountIsInput = true;
    (uint256 baseAmount, uint256 quoteAmount) = _placePerpOrder(
        normalizedAmount,
        isShort,
        amountIsInput,
        sqrtPriceLimitX96
    );
    vault.withdraw(assetToken, baseAmount);
    SwapParams memory params = SwapParams({
        tokenIn: assetToken,
        tokenOut: quoteToken,
        amountIn: baseAmount,
        amountOutMinimum: amountOutMinimum,
        sqrtPriceLimitX96: sqrtPriceLimitX96, //@audit
```



```
        poolFee: swapPoolFee
    });
    uint256 quoteAmountOut = spotSwapper.swapExactInput(params);
```

In `_placePerpOrder()`, it uses the uniswap pool inside the perp protocol and uses a `spotSwapper` for the second swap which is for the uniswap as well.

But as we can see [here](#), Uniswap V3 introduces multiple pools for each token pair and 2 pools might be different and I think it's not good to use the same `sqrtPriceLimitX96` for different pools.

Also, I think it's not mandatory to check a `sqrtPriceLimitX96` as it checks `amountOutMinimum` already. (It checks `amountOutMinimum` only in `_openLong()` and `_openShort()`.)

Impact

`PerpDepository._rebalanceNegativePnlWithSwap()` might revert when it should work as it uses the same `sqrtPriceLimitX96` for different pools.

Code Snippet

<https://github.com/sherlock-audit/2023-01-uxd/blob/main/contracts/integrations/perp/PerpDepository.sol#L478>

Tool used

Manual Review

Recommendation

I think we can use the `sqrtPriceLimitX96` param for one pool only and it would be enough as there is an `amountOutMinimum` condition.

Issue M-3: Vulnerable GovernorVotesQuorumFraction version

Source: <https://github.com/sherlock-audit/2023-01-uxd-judging/issues/423>

Found by

ctf_sec, HonorLt

Summary

The protocol uses an OZ version of contracts that contain a known vulnerability in government contracts.

Vulnerability Detail

UXDGovernor contract inherits from GovernorVotesQuorumFraction:

```
contract UXDGovernor is
    ReentrancyGuard,
    Governor,
    GovernorVotes,
    GovernorVotesQuorumFraction,
    GovernorTimelockControl,
    GovernorCountingSimple,
    GovernorSettings
```

An OZ security recommendation has revealed a known vulnerability in this contract: <https://github.com/OpenZeppelin/openzeppelin-contracts/security/advisories/GHS-A-xrc4-737v-9q75>

It was patched in version 4.7.2, but this protocol uses an older version: "@openzeppelin/contracts": "^4.6.0"

Impact

The potential impact is described in the OZ advisory. This issue was assigned with a severity of High from OZ, so I am sticking with it in this submission.

Code Snippet

<https://github.com/sherlock-audit/2023-01-uxd/blob/main/contracts/governance/UXDGovernor.sol#L37>



Tool used

Manual Review

Recommendation

Update the OZ version of contracts to version $\geq 4.7.2$ or at least follow the workarounds of OZ if not possible otherwise.

Discussion

WarTech9

This was already fixed here: <https://github.com/UXDProtocol/uxd-evm/commit/dca0e857111f3f7946ee5c5a188dbb23ca80859>

hrishibhat

This requires certain scenario where the previous quorum should have failed & the quorum fraction has to be changed post which this issue could be valid.
Considering this issue a valid medium.



Issue M-4: Deposit and withdraw to the vault with the wrong decimals of amount in contract PerpDepository

Source: <https://github.com/sherlock-audit/2023-01-uxd-judging/issues/402>

Found by

berndartmueller, yixxas, HollaDieWaldfee, Bahurum, rvierdiiev, duc, peanuts

Summary

Function `vault.deposit` and `vault.withdraw` of vault in contract `PerpDepository` need to be passed with the amount in raw decimal of tokens (is different from 18 in case using USDC, WBTC, ... as base and quote tokens). But some calls miss the conversion of decimals from 18 to token's decimal, and pass wrong decimals into them.

Vulnerability Detail

- Function `vault.deposit` need to be passed the param amount in token's decimal (as same as `vault.withdraw`). You can see at function `_depositAsset` in contract `PerpDepository`.
- But there are some calls of `vault.deposit` and `vault.withdraw` that passed the amount in the wrong decimal (18 decimal). Let's see function `_rebalanceNegativePnlWithSwap` in contract `PerpDepository`:

Because function `_placePerpOrder` returns in decimal 18 (confirmed with sponsor WarTech), this calls pass `baseAmount` and `quoteAmount` in decimal 18, inconsistent with the above call. It leads to vault using the wrong decimal when depositing and withdrawing tokens.

- There is another case that use `vault.withdraw` with the wrong decimal (same as this case) in function `_rebalanceNegativePnlLite`:

Impact

Because of calling `vault.deposit` and `vault.withdraw` with the wrong decimal of the param amount, the protocol can lose a lot of funds. And some functionalities of the protocol can be broken cause it can revert by not enough allowance when calling these functions.

Code Snippet

<https://github.com/sherlock-audit/2023-01-uxd/blob/main/contracts/integrations/perp/PerpDepository.sol#L498> <https://github.com/sherlock-audit/2023-01-uxd/blob>



[/main/contracts/integrations/perp/PerpDepository.sol#L524](#)
<https://github.com/sherlock-audit/2023-01-uxd/blob/main/contracts/integrations/perp/PerpDepository.sol#L638>

Tool used

Manual Review

Recommendation

Should convert the param `amount` from token's decimal to decimal 18 before `vault.deposit` and `vault.withdraw`.

Discussion

WarTech9

`amount` is always in the unit of the token being deposited so no normalization required.

hrishibhat

@WarTech9 just confirming that your comment is applicable to all the duplicates tagged here. Right?

WarTech9

After further review this issue is valid. `quoteAmount` should be converted to `quoteDecimals` before `vault.deposit()` in `_rebalanceNegativePnlWithSwap()`

WarTech9

Issue exists in `_rebalanceNegativePnlWithSwap()` before `vault.deposit()`. Need to convert from 18 decimals to 6 for USDC. Issue does not exist in `_rebalanceNegativePnlLite` where amount returned by `_placePerpOrder()` is already in 18 decimals. So no conversion required before `vault.withdraw()`.



Issue M-5: PerpDepository#_rebalanceNegativePnlWithSwap fails to approve vault for quote deposit

Source: <https://github.com/sherlock-audit/2023-01-uxd-judging/issues/372>

Found by

HonorLt, 0x52, yixxas, Bahurum, rvierdiiev, GimelSec

Summary

Throughout the entirety of the contract it grants approval to the vault before depositing either quote or asset. In this case there is no approval which means that the deposit call will fail causing PerpDepository#_rebalanceNegativePnlWithSwap to always revert.

Vulnerability Detail

See summary.

Impact

PerpDepository#_rebalanceNegativePnlWithSwap won't function

Code Snippet

<https://github.com/sherlock-audit/2023-01-uxd/blob/main/contracts/integrations/perp/PerpDepository.sol#L478-L528>

Tool used

Manual Review

Recommendation

Add the missing approve call:

```
    } else if (shortFall < 0) {  
        // we got excess tokens in the spot swap. Send them to the account  
        → paying for rebalance  
        IERC20(quoteToken).transfer(  
            account,  
            _abs(shortFall)  
        );  
    }
```



```
+ IERC20(quoteToken).approve(address(vault), quoteAmount);  
  vault.deposit(quoteToken, quoteAmount);  
  
  emit Rebalanced(baseAmount, quoteAmount, shortFall);  
  return (baseAmount, quoteAmount);
```

Discussion

WarTech9

This is a duplicate of #339

0x00052

Two separate issues here. #339 is pointing out it's not approved for the swapper. This one is pointing out it's not approved for the vault. It should be approved for both

WarTech9

@0x00052 good catch. You're right. This is a separate issue from #339



Issue M-6: Inaccurate Perp debt calculation

Source: <https://github.com/sherlock-audit/2023-01-uxd-judging/issues/346>

Found by

peanuts, HollaDieWaldfee, hl_, berndartmueller

Summary

The anticipated Perp account debt value calculation via `PerpDepository.getDebtValue` is inaccurate and does not incorporate the (not yet settled) owed realized PnL `owedRealizedPnl`.

Vulnerability Detail

The `PerpDepository.getDebtValue` function calculates the account debt value by subtracting the pending funding payments and fees from the quote token balance and unrealized PnL. However, the owed realized PnL (`owedRealizedPnl`) is not considered in the calculation. The owed realized PnL is the realized PnL owed to the account but has **not yet been settled**.

Perp provides the `Vault.getSettlementTokenValue()` function to calculate the settlement token value of an account and uses it to determine the accounts' debt (if < 0 , see docs). For example, it is used to determine if an account is liquidable - see [Vault.isLiquidatable#L434](#)

Perps' specs define the value of an account as (see here for reference):

$$accountValue = \underbrace{collateral + owedRealizedPnl + pendingFundingPayment + pendingFee}_{totalCollateralValue} + \underbrace{\sum_{market} unrealizedPnl}_{totalUnrealizedPnl}$$

Impact

The Perp account debt calculation is inaccurate and deviates from the calculation by the Perp protocol itself. Even though the `PerpDepository.getDebtValue` function is `external`, it could lead to issues when querying from another contract or off-chain to use as decision criteria or manifest as a serious issue when used in an upgraded version of the contract.

Code Snippet

[integrations/perp/PerpDepository.sol#L773](#)



```

/// @notice Get the quote token balance of this user
/// @dev The total debt is computed as:
///      quote token balance + unrealized PnL - Pending fee - pending funding
↪ payments
/// @param account The account to return the debt for
/// @return debt The account debt, or zero if no debt.
function getDebtValue(address account) external view returns (uint256) {
    IAccountBalance perpAccountBalance = IAccountBalance(
        clearingHouse.getAccountBalance()
    );
    IExchange perpExchange = IExchange(clearingHouse.getExchange());
    int256 accountQuoteTokenBalance = vault.getBalance(account);
    if (accountQuoteTokenBalance < 0) {
        revert InvalidQuoteTokenBalance(accountQuoteTokenBalance);
    }
    int256 fundingPayment = perpExchange.getAllPendingFundingPayment(
        account
    );
    uint256 quoteTokenBalance = uint256(accountQuoteTokenBalance)
        .fromDecimalToDecimal(ERC20(quoteToken).decimals(), 18);
    (
        ↪ , // @audit-info `owedRealizedPnl` is omitted here and missing in the
        calculation below
        int256 perpUnrealizedPnl,
        uint256 perpPendingFee
    ) = perpAccountBalance.getPnlAndPendingFee(account);
    int256 debt = int256(quoteTokenBalance) +
        perpUnrealizedPnl -
        int256(perpPendingFee) -
        fundingPayment;
    return (debt > 0) ? 0 : _abs(debt);
}

```

Vault._getSettlementTokenValue

Vault._getSettlementTokenValue is called internally by Perp's public Vault.getSettlementTokenValue() function.

```

function _getSettlementTokenValue(address trader) internal view returns (int256
↪ settlementTokenValueX10_18) {
    (int256 settlementBalanceX10_18, int256 unrealizedPnlX10_18) =
        _getSettlementTokenBalanceAndUnrealizedPnl(trader);
    return settlementBalanceX10_18.add(unrealizedPnlX10_18);
}

```

Vault._getSettlementTokenBalanceAndUnrealizedPnl



```

/// @notice Get the specified trader's settlement token balance, including
↳ pending fee, funding payment,
///      owed realized PnL, but without unrealized PnL)
/// @dev Note the difference between the return
↳ argument `settlementTokenBalanceX10_18` and
///      the return value of `getSettlementTokenValue()`.
///      The first one is settlement token balance with pending fee, funding
↳ payment, owed realized PnL;
///      The second one is the first one plus unrealized PnL.
/// @return settlementTokenBalanceX10_18 Settlement amount in 18 decimals
/// @return unrealizedPnlX10_18 Unrealized PnL in 18 decimals
function _getSettlementTokenBalanceAndUnrealizedPnl(address trader)
    internal
    view
    returns (int256 settlementTokenBalanceX10_18, int256 unrealizedPnlX10_18)
{
    int256 fundingPaymentX10_18 =
↳ IExchange(_exchange).getAllPendingFundingPayment(trader);

    int256 owedRealizedPnlX10_18;
    uint256 pendingFeeX10_18;
    (owedRealizedPnlX10_18, unrealizedPnlX10_18, pendingFeeX10_18) =
↳ IAccountBalance(_accountBalance)
        .getPnlAndPendingFee(trader);

    settlementTokenBalanceX10_18 =
↳ getBalance(trader).parseSettlementToken(_decimals).add(
        pendingFeeX10_18.toInt256().sub(fundingPaymentX10_18).add(owedRealizedPn
↳ lX10_18) // @audit-info owed realized PnL is added here
    );

    return (settlementTokenBalanceX10_18, unrealizedPnlX10_18);
}

```

Tool used

Manual Review

Recommendation

Consider using the `Vault.getSettlementTokenValue()` function to determine the accounts' debt ([see docs](#)).



Issue M-7: Rebalancing a negative Perp PnL via a Uniswap V3 token swap is broken due to the lack of token spending allowance

Source: <https://github.com/sherlock-audit/2023-01-uxd-judging/issues/339>

Found by

0x52, Jeiwan, berndartmueller, koxuan, jprod15, Bahurum, cccz, CRYPT70, rvierdiev, GimelSec

Summary

The ISwapper spotSwapper (i.e., Uniswapper) helper contract, used by the PerpDepository._rebalanceNegativePnlWithSwap function to perform the actual Uniswap V3 token swap, is missing the required assetToken spending allowance due to a lack of calling the assetToken.approve function.

Vulnerability Detail

Rebalancing a negative Perp PnL with the PerpDepository.rebalance function calls the _rebalanceNegativePnlWithSwap function, which performs a Uniswap swap. However, the required assetToken spending allowance for the ISwapper spotSwapper (i.e. Uniswapper) helper contract is missing. This leads to a revert due to insufficient allowance.

Impact

Rebalancing a negative Perp PnL via a Uniswap swap is missing the token approval and leads to a revert.

Code Snippet

[integrations/perp/PerpDepository.sol#L507](#)

```
function _rebalanceNegativePnlWithSwap(
    uint256 amount,
    uint256 amountOutMinimum,
    uint160 sqrtPriceLimitX96,
    uint24 swapPoolFee,
    address account
) private returns (uint256, uint256) {
    uint256 normalizedAmount = amount.fromDecimalToDecimal(
        ERC20(quoteToken).decimals(),
```



```

        18
    );
    _checkNegativePnl(normalizedAmount);
    bool isShort = false;
    bool amountIsInput = true;
    (uint256 baseAmount, uint256 quoteAmount) = _placePerpOrder(
        normalizedAmount,
        isShort,
        amountIsInput,
        sqrtPriceLimitX96
    );
    vault.withdraw(assetToken, baseAmount);
    SwapParams memory params = SwapParams({
        tokenIn: assetToken,
        tokenOut: quoteToken,
        amountIn: baseAmount,
        amountOutMinimum: amountOutMinimum,
        sqrtPriceLimitX96: sqrtPriceLimitX96,
        poolFee: swapPoolFee
    });
    uint256 quoteAmountOut = spotSwapper.swapExactInput(params); // @audit-info
    ↪ missing token approval

    // [...]
}

```

Tool used

Manual Review

Recommendation

Consider adding the appropriate token approval before the swap in L507.



Issue M-8: Redeeming all UXD tokens is not possible if some have been minted via Perp quote minting

Source: <https://github.com/sherlock-audit/2023-01-uxd-judging/issues/338>

Found by

berndartmueller

Summary

If some UXD tokens have been minted via Perp quote minting (i.e. USDC - due to negative PnL), rather than asset minting (i.e. WETH), redeeming all UXD tokens is not possible anymore due to not accounting for quote minting in the `netAssetDeposits` variable.

Vulnerability Detail

The Perp integration allows users to deposit assets and return the redeemable (UXD) amount that can be minted. Assets can be deposited in two ways: either `assetToken` or `quoteToken`. `quoteToken` (i.e., USDC) can only be deposited if there's a negative `PnL > amount` to pay off the negative PnL.

Depositing `assetToken` (i.e., WETH) increases the internal accounting variable `netAssetDeposits`, which is the amount of `assetToken` deposited minus the amount of `assetToken` withdrawn. Withdrawing `assetToken` decreases `netAssetDeposits` by the amount withdrawn.

However, depositing `quoteToken` (if available) does not increase the `netAssetDeposits` variable. This leads to more UXD tokens being minted than `assetToken` deposited.

Thus, redeeming all UXD tokens is not possible anymore if some UXD tokens have been minted via quote minting. Users who are left last with redeeming their UXD tokens cannot do so.

Consider the following example:

- `assetToken` = WETH
- `quoteToken` = USDC
- 1 ether of WETH = 1_000e6 USDC = 1_000e18 UXD
- Negative PnL occurs during deposits because of market volatility



	() Time	Action	netAssetDeposits	UXD supply	
			()		
T0		Alice deposits 10 ether of assetToken	10 ether	10 ether	10_000e18
T1		Bob deposits 10 ether of assetToken	20 ether	20 ether	20_000e18
T2		Caroline deposits 1_000e6 of quoteToken (due to negative PnL)	20 ether	21 ether	21_000e18
T10		Caroline redeems 1_000e18 UXD via 1 ether assetToken	19 ether	20 ether	20_000e18
T11		Bob redeems 10_000e18 UXD via 10 ether assetToken	9 ether	10 ether	10_000e18
T12		Alice redeems 10_000e18 UXD via 10 ether assetToken -> fails	9 ether	10 ether	10_000e18
			()		

Alice is not able to redeem her balance of 10_000e18 UXD because netAssetDeposits = 9 ether is insufficient to redeem 10_000e18 UXD = 10 ether.

Impact

UXD tokens can not be fully redeemed via the Perp depository if some have been minted via quote minting.

Code Snippet

[integrations/perp/PerpDepository.sol#L284](#)

Asset token deposits increase the internal accounting variable netAssetDeposits by the amount deposited.

```

/// @notice Deposits collateral to back the delta-neutral position
/// @dev Only called by the controller
/// @param amount The amount to deposit
function _depositAsset(uint256 amount) private {
    netAssetDeposits += amount;

    IERC20(assetToken).approve(address(vault), amount);
    vault.deposit(assetToken, amount);
}

```

[integrations/perp/PerpDepository.sol#L298](#)

Withdrawing asset tokens (i.e. WETH) decreases the internal accounting variable netAssetDeposits by the amount withdrawn and thus limits the amount of redeemable UXD tokens to netAssetDeposits.



```

/// @notice Withdraws collateral to used in the delta-neutral position.
/// @dev This should only happen when redeeming UXD for collateral.
/// Only called by the controller.
/// @param amount The amount to deposit
function _withdrawAsset(uint256 amount, address to) private {
    if (amount > netAssetDeposits) {
        revert InsufficientAssetDeposits(netAssetDeposits, amount);
    }
    netAssetDeposits -= amount;

    vault.withdraw(address(assetToken), amount);
    IERC20(assetToken).transfer(to, amount);
}

```

integrations/perp/PerpDepository.redeem

Redeeming via quote tokens is disabled. Only redeeming via `assetToken` is possible. Thus, leaving no other option than to redeem via `assetToken`.

```

function redeem(
    address asset,
    uint256 amount
) external onlyController returns (uint256) {
    if (asset == assetToken) {
        (uint256 base, ) = _openLong(amount);
        _withdrawAsset(base, address(controller));
        return base;
    } else if (asset == quoteToken) {
        revert QuoteRedeemDisabled(msg.sender);
        // return _processQuoteRedeem(amount);
    } else {
        revert UnsupportedAsset(asset);
    }
}

```

Tool used

Manual Review

Recommendation

Consider changing the internal accounting of `netAssetDeposits` to include `quoteToken` deposits.



Discussion

berndartmueller

Escalate for 25 USDC

I think this issue is wrongly duped. The duped issues mention a similar issue in the context of the **rebalancing** functions. This issue demonstrated here is about **quote minting** in the case of a negative PnL.

I think it's justified to leave this issue on its own without the duplicate issues. The duped issues should be left as a separate issue (Please note that #434 does not mention the real impact of not being able to redeem fully).

(<https://github.com/sherlock-audit/2023-01-uxd-judging/issues/97> has also been considered a separate issue)

sherlock-admin

Escalate for 25 USDC

I think this issue is wrongly duped. The duped issues mention a similar issue in the context of the **rebalancing** functions. This issue demonstrated here is about **quote minting** in the case of a negative PnL.

I think it's justified to leave this issue on its own without the duplicate issues. The duped issues should be left as a separate issue (Please note that #434 does not mention the real impact of not being able to redeem fully).

(<https://github.com/sherlock-audit/2023-01-uxd-judging/issues/97> has also been considered a separate issue)

You've created a valid escalation for 25 USDC!

To remove the escalation from consideration: Delete your comment. To change the amount you've staked on this escalation: Edit your comment (**do not create a new comment**).

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

hrishibhat

Escalation accepted

Agreed that the issues is different from its dupes. Separating from the rest of the issues #178, #262, #141 as they are considered as low together, since not updating `netAssetDeposits` on `rebalance` does not cause any real issues. Also, issue #434 touches upon the issue partially, but does not fully describe the complete issue & its impact. Hence considering it a low.

sherlock-admin



Escalation accepted

Agreed that the issues is different from its dupes. Separating from the rest of the issues #178, #262, #141 as they are considered as low together, since not updating `netAssetDeposits` on rebalance does not cause any real issues. Also, issue #434 touches upon the issue partially, but does not fully describe the complete issue & its impact. Hence considering it a low.

This issue's escalations have been accepted!

Contestants' payouts and scores will be updated according to the changes made on this issue.



Issue M-9: Price disparities between spot and perpetual pricing can heavily destabilize UXD

Source: <https://github.com/sherlock-audit/2023-01-uxd-judging/issues/305>

Found by

0x52

Summary

When minting UXD using PerpDepository.sol the amount of UXD minted corresponds to the amount of vUSD gained from selling the deposited ETH. This is problematic given that Perp Protocol is a derivative rather than a spot market, which means that price differences cannot be directly arbitrated with spot markets. The result is that derivative markets frequently trade at a price higher or lower than the spot price. The result of this is that UXD is actually pegged to vUSD rather than USD. This key difference can cause huge strain on a USD peg and likely depegging.

Vulnerability Detail

```
function deposit(
    address asset,
    uint256 amount
) external onlyController returns (uint256) {
    if (asset == assetToken) {
        _depositAsset(amount);
        (, uint256 quoteAmount) = _openShort(amount);
        return quoteAmount; // @audit this mint UXD equivalent to the amount of
        ↪ vUSD gained
    } else if (asset == quoteToken) {
        return _processQuoteMint(amount);
    } else {
        revert UnsupportedAsset(asset);
    }
}
```

PerpDepository#deposit shorts the deposit amount and returns the amount of vUSD resulting from the swap, which effectively pegs it to vUSD rather than USD. When the perpetual is trading at a premium arbitrage will begin happening between the spot and perpetual asset and the profit will be taken at the expense of the UXD peg.

Example: Imagine markets are heavily trending with a spot price of \$1500 and a perpetual price of \$1530. A user can now buy 1 ETH for \$1500 and deposit it to



mint 1530 UXD. They can then swap the UXD for 1530 USDC (or other stablecoin) for a profit of \$30. The user can continue to do this until either the perpetual price is arbitrated down to \$1500 or the price of UXD is \$0.98.

Impact

UXD is pegged to vUSD rather than USD which can cause instability and loss of peg

Code Snippet

<https://github.com/sherlock-audit/2023-01-uxd/blob/main/contracts/integrations/PerpDepository.sol#L240-L253>

Tool used

Manual Review

Recommendation

I recommend integrating with a chainlink oracle and using its price to determine the true spot price of ETH. When a user mints make sure that the amount minted is never greater than the spot price of ETH which will prevent the negative pressure on the peg:

```
function deposit(
    address asset,
    uint256 amount
) external onlyController returns (uint256) {
    if (asset == assetToken) {
        _depositAsset(amount);
        (, uint256 quoteAmount) = _openShort(amount);

+       spotPrice = assetOracle.getPrice();
+       assetSpotValue = amount.mulwad(spotPrice);

-       return quoteAmount;
+       return quoteAmount <= assetSpotValue ? quoteAmount: assetSpotValue;
    } else if (asset == quoteToken) {
        return _processQuoteMint(amount);
    } else {
        revert UnsupportedAsset(asset);
    }
}
```



Discussion

WarTech9

It's a design decision to use the PERP price to determine mint/redeem amounts. We can add oracle pricing in the future to be more robust, but that is not a priority at this moment.

acamill

This is a known issue on Solana implementation too. Over the course of the year this offset due to the spread between spot and perp prices always went back to 0. But this is something that we have in mind for later

hrishibhat

This is a valid issue in case of certain market conditions or manipulations for the vUSD to trade away from the peg. Considering this issue as a valid medium.



Issue M-10: The `FullMath` library is unable to handle intermediate overflows due to overflow that's desired but never reached

Source: <https://github.com/sherlock-audit/2023-01-uxd-judging/issues/273>

Found by

DecorativePineapple

Summary

The `FullMath` library doesn't correctly handle the case when an intermediate value overflows 256 bits. This happens because an overflow is desired in this case but it's never reached.

Vulnerability Detail

The `FullMath` library was taken from `Uniswap v3-core`. However, the original solidity version that was used was `< 0.8.0`, meaning that the execution didn't revert when an overflow was reached. This effectively means that when a phantom overflow (a multiplication and division where an intermediate value overflows 256 bits) occurs the execution will revert and the correct result won't be returned. The original library was designed in a way that could handle intermediate overflows. The `FullMath` library is used in the `MathLiblibrary` in order to format an 18-decimal number to a FixedPoint 96.Q96 number.

Impact

The correct result isn't returned in this case and the execution gets reverted when a phantom overflows occurs.

Code Snippet

The `FullMath` library which doesn't use an unchecked block:

```
library FullMath {
    /// @notice Calculates floor(abdenominator) with full precision. Throws if
    ↪ result overflows a uint256 or denominator == 0
    /// @param a The multiplicand
    /// @param b The multiplier
    /// @param denominator The divisor
    /// @return result The 256-bit result
    /// @dev Credit to Remco Bloemen under MIT license
    ↪ https://xn--2-umb.com/21/muldiv
```



```

function mulDiv(
    uint256 a,
    uint256 b,
    uint256 denominator
) internal pure returns (uint256 result) {
    // 512-bit multiply [prod1 prod0] = a * b
    // Compute the product mod 2**256 and mod 2**256 - 1
    // then use the Chinese Remainder Theorem to reconstruct
    // the 512 bit result. The result is stored in two 256
    // variables such that product = prod1 * 2**256 + prod0
    uint256 prod0; // Least significant 256 bits of the product
    ...

    result = prod0 * inv;
    return result;
}

function mulDivRoundingUp(
    uint256 a,
    uint256 b,
    uint256 denominator
) internal pure returns (uint256 result) {
    result = mulDiv(a, b, denominator);
    if (mulmod(a, b, denominator) > 0) {
        require(result < type(uint256).max);
        result++;
    }
}
}

```

Tool used

Manual Code Review

Recommendation

It is advised to put the entire function bodies of `mulDiv` and `mulDivRoundingUp` in an unchecked block. A modified version of the original `Fullmath` library that uses unchecked blocks to handle the overflow, can be found in the 0.8 branch of the [Uniswap v3-core repo](#).

Discussion

WarTech9



Unchecked doesn't handle the error, any overflow/underflow would fail silently. upgrading to the newer version of Fullmath should would be good though but is low priority.

hrishibhat

Agree with sponsor comment. Considering the issue as informational

DecorativePineapple

Escalate for 15 USDC

As noted in the FullMath library, it was designed in order to handle "phantom overflow" i.e., allows multiplication and division where an intermediate value overflows 256 bits. This means that the **original** library can handle statements like : `FullMath.mulDiv(type(uint256).max / 2, 3, 111))` . This a feature of the library, as statements like this cannot be handled by solidity >0.8 directly as the execution will revert due to an overflow. Because the original library was created with solidity version <0.8.0(which doesn't revert on overflows) this behaviour was allowed as the **expected** intermediary overflow could be reached.

However, due to the fact that the FullMath Library was ported to solidity version 0.8.9, which is >0.8, this operation would revert as the intermediate calculations would overflow, meaning that it can't handle those multiplication and division where an intermediate value overflows the 256 bits. The fix is to mark the full body in an unchecked block, in order to leverage the fact that the original version was designed in order to allow the intermediate overflow.

I think that the comment left by the sponsor: `Unchecked doesn't handle the error`, doesn't reflect the issue, as by using the unchecked block the desired behaviour could be reached.

A modified version of the original Fullmath library that uses unchecked blocks to handle the **intended** overflow, can be found in the 0.8 branch of the Uniswap v3-core repo.

sherlock-admin

Escalate for 15 USDC

As noted in the FullMath library, it was designed in order to handle "phantom overflow" i.e., allows multiplication and division where an intermediate value overflows 256 bits. This means that the **original** library can handle statements like : `FullMath.mulDiv(type(uint256).max / 2, 3, 111))` . This a feature of the library, as statements like this cannot be handled by solidity >0.8 directly as the execution will revert due to an overflow. Because the original library was created with solidity version <0.8.0(which doesn't revert on overflows) this behaviour was allowed as the **expected** intermediary overflow could be reached.



However, due to the fact that the FullMath Library was ported to solidity version 0.8.9, which is >0.8 , this operation would revert as the intermediate calculations would overflow, meaning that it can't handle those multiplication and division where an intermediate value overflows the 256 bits. The fix is to mark the full body in an unchecked block, in order to leverage the fact that the original version was designed in order to allow the intermediate overflow.

I think that the comment left by the sponsor: `Unchecked doesn't handle the error`, , doesn't reflect the issue, as by using the unchecked block the desired behaviour could be reached.

A modified version of the original Fullmath library that uses unchecked blocks to handle the **intended** overflow, can be found in the 0.8 branch of the Uniswap v3-core repo.

You've created a valid escalation for 15 USDC!

To remove the escalation from consideration: Delete your comment. To change the amount you've staked on this escalation: Edit your comment (**do not create a new comment**).

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

hrishibhat

Escalation accepted

Agree that there could a potential overflow, & it must be updated to latest FullMath version. Considering this a valid medium issue.

sherlock-admin

Escalation accepted

Agree that there could a potential overflow, & it must be updated to latest FullMath version. Considering this a valid medium issue

This issue's escalations have been accepted!

Contestants' payouts and scores will be updated according to the changes made on this issue.

sherlock-admin

Escalation accepted

Agree that there could a potential overflow, & it must be updated to latest FullMath version. Considering this a valid medium issue.

This issue's escalations have been accepted!



Contestants' payouts and scores will be updated according to the changes made on this issue.



Issue M-11: PerpDepository#_placePerpOrder miscalculates fees paid when shorting

Source: <https://github.com/sherlock-audit/2023-01-uxd-judging/issues/271>

Found by

Jeiwan, 0x52, berndartmueller, cccz, rvierdiev, GimelSec, peanuts, keccak123

Summary

PerpDepository#_placePerpOrder calculates the fee as a percentage of the quoteToken received. The issue is that this amount already has the fees taken so the fee percentage is being applied incorrectly.

Vulnerability Detail

```
function _placePerpOrder(
    uint256 amount,
    bool isShort,
    bool amountIsInput,
    uint160 sqrtPriceLimit
) private returns (uint256, uint256) {
    uint256 upperBound = 0; // 0 = no limit, limit set by sqrtPriceLimit

    IClearingHouse.OpenPositionParams memory params = IClearingHouse
        .OpenPositionParams({
            baseToken: market,
            isBaseToQuote: isShort, // true for short
            isExactInput: amountIsInput, // we specify exact input amount
            amount: amount, // collateral amount - fees
            oppositeAmountBound: upperBound, // output upper bound
            // solhint-disable-next-line not-rely-on-time
            deadline: block.timestamp,
            sqrtPriceLimitX96: sqrtPriceLimit, // max slippage
            referralCode: 0x0
        });

    (uint256 baseAmount, uint256 quoteAmount) = clearingHouse.openPosition(
        params
    );

    uint256 feeAmount = _calculatePerpOrderFeeAmount(quoteAmount);
    totalFeesPaid += feeAmount;
```




```

        emit PositionOpened(isShort, amount, amountIsInput, sqrtPriceLimit);
        return (baseAmount, quoteAmount);
    }

    function _calculatePerpOrderFeeAmount(uint256 amount)
        internal
        view
        returns (uint256)
    {
        return amount.mulWadUp(getExchangeFeeWad());
    }

```

When calculating fees, `PerpDepository#_placePerpOrder` use the quote amount returned when opening the new position. It always uses `exactIn` which means that for shorts the amount of baseAsset being sold is specified. The result is that quote amount returned is already less the fees. If we look at how the fee is calculated we can see that it is incorrect.

Example: Imagine the market price of ETH is \$1000 and there is a market fee of 1%. The 1 ETH is sold and the contract receives 990 USD. Using the math above it would calculate the fee as \$99 ($990 * 1\%$) but actually the fee is \$100.

I have submitted this as a medium because it is not clear from the given contracts what the fee totals are used for and I cannot fully assess the implications of the fee value being incorrect.

Impact

`totalFeesPaid` will be inaccurate which could lead to disparities in other contracts depending on how it is used

Code Snippet

<https://github.com/sherlock-audit/2023-01-uxd/blob/main/contracts/integrations/perp/PerpDepository.sol#L804-L810>

Tool used

Manual Review

Recommendation

Rewrite `_calculatePerpOrderFeeAmount` to correctly calculate the fees paid:

```

-   function _calculatePerpOrderFeeAmount(uint256 amount)
+   function _calculatePerpOrderFeeAmount(uint256 amount, bool isShort)
        internal

```



```
        view
        returns (uint256)
    {
+       if (isShort) {
+           return amount.divWadDown(WAD - getExchangeFeeWad()) - amount;
+       } else {
+           return amount.mulWadUp(getExchangeFeeWad());
+       }
    }
```



Issue M-12: PerpDepository.netAssetDeposits variable can prevent users to withdraw with underflow error

Source: <https://github.com/sherlock-audit/2023-01-uxd-judging/issues/97>

Found by

rvierdiiev

Summary

PerpDepository.netAssetDeposits variable can prevent users to withdraw with underflow error

Vulnerability Detail

When user deposits using PerpDepository, then `netAssetDeposits` variable is increased with the base assets amount, provided by depositor.

<https://github.com/sherlock-audit/2023-01-uxd/blob/main/contracts/integrations/perp/PerpDepository.sol#L283-L288>

```
function _depositAsset(uint256 amount) private {
    netAssetDeposits += amount;

    IERC20(assetToken).approve(address(vault), amount);
    vault.deposit(assetToken, amount);
}
```

Also when user withdraws, this `netAssetDeposits` variable is decreased with base amount that user has received for redeeming his UXD tokens.

<https://github.com/sherlock-audit/2023-01-uxd/blob/main/contracts/integrations/perp/PerpDepository.sol#L294-L302>

```
function _withdrawAsset(uint256 amount, address to) private {
    if (amount > netAssetDeposits) {
        revert InsufficientAssetDeposits(netAssetDeposits, amount);
    }
    netAssetDeposits -= amount;

    vault.withdraw(address(assetToken), amount);
    IERC20(assetToken).transfer(to, amount);
}
```



The problem here is that when user deposits X assets, then he receives Y UXD tokens. And when later he redeems his Y UXD tokens he can receive more or less than X assets. This can lead to situation when `netAssetDeposits` variable will be seting to negative value which will revert tx.

Example. 1.User deposits 1 WETH when it costs 1200\$. As result 1200 UXD tokens were minted and `netAssetDeposits` was set to 1. 2.Price of WETH has decreased and now it costs 1100. 3.User redeem his 1200 UXD tokens and receives from perp protocol $1200/1100=1.09$ WETH. But because `netAssetDeposits` is 1, then transaction will revert inside `_withdrawAsset` function with underflow error.

Impact

User can't redeem all his UXD tokens.

Code Snippet

<https://github.com/sherlock-audit/2023-01-uxd/blob/main/contracts/integrations/perp/PerpDepository.sol#L264-L278> <https://github.com/sherlock-audit/2023-01-uxd/blob/main/contracts/integrations/perp/PerpDepository.sol#L294-L302>
<https://github.com/sherlock-audit/2023-01-uxd/blob/main/contracts/integrations/perp/PerpDepository.sol#L240-L253> <https://github.com/sherlock-audit/2023-01-uxd/blob/main/contracts/integrations/perp/PerpDepository.sol#L283-L288>

Tool used

Manual Review

Recommendation

As you don't use this variable anywhere else, you can remove it. Otherwise you need to have 2 variables instead: `totalDeposited` and `totalWithdrawn`.

Discussion

WarTech9

One fix: `netAssetDeposits` should be updated during rebalancing.

WarTech9

@acamill This *can only be partially fixed* by updating `netAssetsDeposits` while rebalancing but that's only resolves the issue if rebalancing has occurred. It would still be possible to run into this if rebalancing has not yet occurred so its not a full fix. We could use 2 variables as suggested but due to changes in asset values between mints and redeems, those would diverge and would be meaningless. We



already have the position size which tells us this information, thus removing this field is the better option.

0x00052

Escalate for 25 USDC

This should only be medium severity because it is an edge case for the following reasons:

- 1) It can only occur if the average withdraw price is chronically under the average deposit price.
- 2) It only affects the tail end of withdraws, requiring a majority of the depository to be withdrawn
- 3) This state is not permanent because later deposits/withdraws can function in reverse (i.e. deposited at 1100 and withdraw at 1200) to cause netAssetsDeposits to go back up and free stuck assets

sherlock-admin

Escalate for 25 USDC

This should only be medium severity because it is an edge case for the following reasons:

- 1) It can only occur if the average withdraw price is chronically under the average deposit price.
- 2) It only affects the tail end of withdraws, requiring a majority of the depository to be withdrawn
- 3) This state is not permanent because later deposits/withdraws can function in reverse (i.e. deposited at 1100 and withdraw at 1200) to cause netAssetsDeposits to go back up and free stuck assets

You've created a valid escalation for 25 USDC!

To remove the escalation from consideration: Delete your comment. To change the amount you've staked on this escalation: Edit your comment (**do not create a new comment**).

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

hrishibhat

Escalation accepted

Given the certain specific requirements for the issue to occur, also note that there the condition can correct itself to free stuck assets. Considering this issue as medium



sherlock-admin

Escalation accepted

Given the certain specific requirements for the issue to occur, also note that there the condition can correct itself to free stuck assets.

Considering this issue as medium

This issue's escalations have been accepted!

Contestants' payouts and scores will be updated according to the changes made on this issue.

