SHERLOCK

SHERLOCK SECURITY REVIEW FOR

Prepared for:	Opyn
Prepared by:	Sherlock
Lead Security Expert:	thec00n
Dates Audited:	November 23 - November 30, 2022
Prepared on:	December 9, 2022

Introduction

Opyn is building DeFi-native derivatives and options infrastructure in DeFi. Known for developing Squeeth, perpetual exposure to ETH squared.

Scope

The following contracts are in scope

1. CrabNetting.sol (its interfaces) and its dependencies CrabStrategyV2.sol, Controller.sol

You can find the dependencies inside this repo

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
4	4

Issues not fixed or acknowledged

Medium	High
0	0

Security experts who found valid issues

hyh
keccak123
yixxas
KingNFT
cccz
Jeiwan
bin2chen
thec00n
joestakey
libratus
Zarf
indijanc
CRYP70
rotcivegaf
adriro
Met
chainNue
Deivitto

dipp
0x52
rvierdiiev
HonorLt
reassor
Haruxe
minhtrng
Atarpara
jonatascm
John
aviggiano
__141345__
csanuragjain
zapaz
Trumpero
kaliberpoziomka
imare
caventa
seyni
hansfriese
zimu
ctf_sec

Issue H-1: debtToMint incorrectly treats feeAdjustment decimals

Source: https://github.com/sherlock-audit/2022-11-opyn-judging/issues/236

Found by

keccak123, hyh

Summary

_debtToMint() will return `0` decimals amounts and `sqthToSell` in depositAuction() will be insignificant, leading to ignoring the market orders used and depositing auction to be void as no external funding will be brought in.

Vulnerability Detail

`feeAdjustment=_calcFeeAdjustment()` is `(squeethEthPrice*feeRate)/10000` and have `18` decimals.

`wSqueethToMint=(_amount*debt)/(collateral+(debt*feeAdjustment))` will have `36` decimals in numerator and the same `36` in denominator, yielding `0` decimals figure. That figure is `sqthToSell`, so no market buying orders will be ever filled.

Impact

depositAuction() will malfunction all the time, either reverting or producing less WETH and less CRAB than desired, i.e. there will be no deposit auction as market order part is needed to bring in the liquidity to be distributed.

Setting the severity to be high as this is system malfunction with material impact and no prerequisites.

Code Snippet

feeAdjustment is treated as if it has no decimals:

https://github.com/sherlock-audit/2022-11-opyn/blob/main/crab-netting/src/CrabNetting.sol#L476-L485

```
/**
 * @dev calculates wSqueeth minted when amount is deposited
 * @param _amount to deposit into crab
 */
function _debtToMint(uint256 _amount) internal view returns (uint256) {
    uint256 feeAdjustment = _calcFeeAdjustment();
    (,, uint256 collateral, uint256 debt) =
↪   ICrabStrategyV2(crab).getVaultDetails();
    uint256 wSqueethToMint = (_amount * debt) / (collateral + (debt *
↪   feeAdjustment));
    return wSqueethToMint;
}
```

while it has 18 decimals:

https://github.com/sherlock-audit/2022-11-opyn/blob/main/crab-netting/src/CrabNetting.sol#L795-L800

```
function _calcFeeAdjustment() internal view returns (uint256) {
    uint256 feeRate = IController(sqthController).feeRate();
    if (feeRate == 0) return 0;
    uint256 squeethEthPrice = IOracle(oracle).getTwap(ethSqueethPool, sqth, weth,
↪   sqthTwapPeriod, true);
    return (squeethEthPrice * feeRate) / 10000;
}
```

As `sqthToSell` to be insignificant, there will be no Squeeth selling at all:

https://github.com/sherlock-audit/2022-11-opyn/blob/main/crab-netting/src/CrabNetting.sol#L491-L504

```
function depositAuction(DepositAuctionParams calldata _p) external onlyOwner {
    _checkOTCPrice(_p.clearingPrice, false);
    /**
     * step 1: get eth from mm
     *     step 2: get eth from deposit usdc
     *     step 3: crab deposit
     *     step 4: flash deposit
     *     step 5: send sqth to mms
     *     step 6: send crab to depositors
     */
    uint256 initCrabBalance = IERC20(crab).balanceOf(address(this));
    uint256 initEthBalance = address(this).balance;

    uint256 sqthToSell = _debtToMint(_p.totalDeposit);
```

This renders sqth buying orders block void, i.e. it will be always `_p.orders[0].quantity>=remainingToSell`:

https://github.com/sherlock-audit/2022-11-opyn/blob/main/crab-netting/src/CrabNetting.sol#L504-L524

```
uint256 sqthToSell = _debtToMint(_p.totalDeposit);
// step 1 get all the eth in
```

```
uint256 remainingToSell = sqthToSell;
for (uint256 i = 0; i < _p.orders.length; i++) {
    require(_p.orders[i].isBuying, "auction order not buying sqth");
    require(_p.orders[i].price >= _p.clearingPrice, "buy order price less than
↪   clearing");
    _checkOrder(_p.orders[i]);
    if (_p.orders[i].quantity >= remainingToSell) {
        IWETH(weth).transferFrom(
            _p.orders[i].trader, address(this), (remainingToSell *
↪   _p.clearingPrice) / 1e18
        );
        remainingToSell = 0;
        break;
    } else {
        IWETH(weth).transferFrom(
            _p.orders[i].trader, address(this), (_p.orders[i].quantity *
↪   _p.clearingPrice) / 1e18
        );
        remainingToSell -= _p.orders[i].quantity;
    }
}
require(remainingToSell == 0, "not enough buy orders for sqth");
```

Tool used

Manual Review

Recommendation

Consider adding decimals treatment, for example:

https://github.com/sherlock-audit/2022-11-opyn/blob/main/crab-netting/src/CrabNetting.sol#L476-L485

```
    /**
     * @dev calculates wSqueeth minted when amount is deposited
     * @param _amount to deposit into crab
     */
    function _debtToMint(uint256 _amount) internal view returns (uint256) {
        uint256 feeAdjustment = _calcFeeAdjustment();
        (,, uint256 collateral, uint256 debt) =
↪   ICrabStrategyV2(crab).getVaultDetails();
-       uint256 wSqueethToMint = (_amount * debt) / (collateral + (debt *
↪   feeAdjustment));
+       uint256 wSqueethToMint = (_amount * debt) / (collateral + (debt *
↪   feeAdjustment) / 1e18);
        return wSqueethToMint;
    }
```

Discussion

thec00n

Nice find. Fix lgtm.

Issue H-2: Netting and withdraw auction can be frozen permanently

Source: https://github.com/sherlock-audit/2022-11-opyn-judging/issues/219

Found by

cccz, hyh, bin2chen, yixxas, libratus, joestakey, Zarf, KingNFT

Summary

An attacker can permanently block the auctions by using a blocked address to fail USDC transfers, which are now required for the auction to proceed.

Vulnerability Detail

Say Bob knows that one of his addresses is blocked by USDC. He has/can obtain CRAB, which he can transfer to this address.

As withdraw queue requires each transfer call to be successful, this will permanently freezes the functionality, i.e. all future auctions will be blocked.

Knowing that, Bob will block the auctions when it's beneficial to him the most.

Impact

netAtPrice() and withdrawAuction() will be blocked as long as Bob's withdrawal is queued. There is no way for the owner to manually alter this state.

As auction timing can have material impact on the beneficiaries, the inability to perform netting and withdraw auction will lead to losses for them as Bob will choose the moment to execute the attack to benefit himself at the expense of the participants.

Setting the severity to be high as this is permanent freeze of the core functionality fully controllable by the attacker only.

Code Snippet

netAtPrice() will be reverting at Bob's withdrawal:

https://github.com/sherlock-audit/2022-11-opyn/blob/main/crab-netting/src/CrabNetting.sol#L389-L419

```
    // process withdraws and send usdc
    i = withdrawsIndex;
    while (crabQuantity > 0) {
        Receipt memory withdraw = withdraws[i];
        if (withdraw.amount == 0) {
            i++;
            continue;
        }
        if (withdraw.amount <= crabQuantity) {
            crabQuantity = crabQuantity - withdraw.amount;
            crabBalance[withdraw.sender] -= withdraw.amount;
            amountToSend = (withdraw.amount * _price) / 1e18;
            IERC20(usdc).transfer(withdraw.sender, amountToSend);

            emit CrabWithdrawn(withdraw.sender, withdraw.amount, amountToSend,
 ↪  i);

            delete withdraws[i];
            i++;
        } else {
            withdraws[i].amount = withdraw.amount - crabQuantity;
            crabBalance[withdraw.sender] -= crabQuantity;
            amountToSend = (crabQuantity * _price) / 1e18;
            IERC20(usdc).transfer(withdraw.sender, amountToSend);

            emit CrabWithdrawn(withdraw.sender, withdraw.amount, amountToSend,
 ↪  i);

            crabQuantity = 0;
        }
    }
    withdrawsIndex = i;
}
```

withdrawAuction() similarly will fail on Bob's entry:

https://github.com/sherlock-audit/2022-11-opyn/blob/main/crab-netting/src/CrabNetting.sol#L687-L720

```
// step 5 pay all withdrawers and mark their withdraws as done
uint256 remainingWithdraws = _p.crabToWithdraw;
uint256 j = withdrawsIndex;
uint256 usdcAmount;
while (remainingWithdraws > 0) {
    Receipt memory withdraw = withdraws[j];
    if (withdraw.amount == 0) {
        j++;
        continue;
    }
    if (withdraw.amount <= remainingWithdraws) {
        // full usage
        remainingWithdraws -= withdraw.amount;
        crabBalance[withdraw.sender] -= withdraw.amount;

        // send proportional usdc
        usdcAmount = (((withdraw.amount * 1e18) / _p.crabToWithdraw) *
 ↪  usdcReceived) / 1e18;
        IERC20(usdc).transfer(withdraw.sender, usdcAmount);
        emit CrabWithdrawn(withdraw.sender, withdraw.amount, usdcAmount, j);
        delete withdraws[j];
        j++;
    } else {
        withdraws[j].amount -= remainingWithdraws;
        crabBalance[withdraw.sender] -= remainingWithdraws;

        // send proportional usdc
        usdcAmount = (((remainingWithdraws * 1e18) / _p.crabToWithdraw) *
 ↪  usdcReceived) / 1e18;
        IERC20(usdc).transfer(withdraw.sender, usdcAmount);
        emit CrabWithdrawn(withdraw.sender, remainingWithdraws, usdcAmount, j);

        remainingWithdraws = 0;
    }
}
withdrawsIndex = j;
```

netAtPrice() and withdrawAuction() unavailability and the whole withdrawal queue freeze will be permanent as `withdrawsIndex` can be changed only in netAtPrice() and withdrawAuction(), i.e. there is no way to skip any entry, including Bob's.

I.e. only Bob can unstuck the system by removing the withdrawal:

https://github.com/sherlock-audit/2022-11-opyn/blob/main/crab-netting/src/CrabNetting.sol#L319-L346

```
/**
 * @notice withdraw Crab from queue
 * @param _amount Crab amount to dequeue
 */
function dequeueCrab(uint256 _amount) external {
    require(!isAuctionLive, "auction is live");
    crabBalance[msg.sender] = crabBalance[msg.sender] - _amount;
    require(
        crabBalance[msg.sender] >= minCrabAmount || crabBalance[msg.sender] == 0,
        "remaining amount smaller than minimum, consider removing full balance"
    );
    // deQueue crab from the last, last in first out
    uint256 toRemove = _amount;
    uint256 lastIndexP1 = userWithdrawsIndex[msg.sender].length;
    for (uint256 i = lastIndexP1; i > 0; i--) {
        Receipt storage r = withdraws[userWithdrawsIndex[msg.sender][i - 1]];
        if (r.amount > toRemove) {
            r.amount -= toRemove;
            toRemove = 0;
            break;
        } else {
            toRemove -= r.amount;
            delete withdraws[userWithdrawsIndex[msg.sender][i - 1]];
        }
    }
    IERC20(crab).transfer(msg.sender, _amount);
    emit CrabDeQueued(msg.sender, _amount, crabBalance[msg.sender]);
}
```

Tool used

Manual Review

Recommendation

Consider trying to transfer and skipping if there is any malfunction, for example:

https://github.com/sherlock-audit/2022-11-opyn/blob/main/crab-netting/src/CrabNetting.sol#L389-L419

```
        // process withdraws and send usdc
        i = withdrawsIndex;
        while (crabQuantity > 0) {
            Receipt memory withdraw = withdraws[i];
            if (withdraw.amount == 0) {
                i++;
                continue;
            }
            if (withdraw.amount <= crabQuantity) {
                amountToSend = (withdraw.amount * _price) / 1e18;
-               IERC20(usdc).transfer(withdraw.sender, amountToSend);
+               try IERC20(usdc).transfer(withdraw.sender, amountToSend) {
                    crabQuantity = crabQuantity - withdraw.amount;
                    crabBalance[withdraw.sender] -= withdraw.amount;
                    emit CrabWithdrawn(withdraw.sender, withdraw.amount,
↪   amountToSend, i);

                    delete withdraws[i];
+               } catch {
+               }
```

```
                i++;
            } else {
                amountToSend = (crabQuantity * _price) / 1e18;
-               IERC20(usdc).transfer(withdraw.sender, amountToSend);
+               try IERC20(usdc).transfer(withdraw.sender, amountToSend) {
                    withdraws[i].amount = withdraw.amount - crabQuantity;
                    crabBalance[withdraw.sender] -= crabQuantity;
                    emit CrabWithdrawn(withdraw.sender, withdraw.amount,
 ↪   amountToSend, i);

                    crabQuantity = 0;
+               } catch {
+                   ++i;
+               }
            }
        }
        withdrawsIndex = i;
    }
```

This can be paired with introduction of the onlyOwner rescue function to handle the transfer manually, say for USDC ban case: auction operator transfers to self, swaps and return the funds to depositor in another form.

Notice that skipping the entry causes no harm for the withdrawer as dequeueCrab() can be run any time for it.

Discussion

sanandnarayan

https://github.com/opynfinance/squeeth-monorepo/pull/801

thec00n

The owner can repay Crabv2 to users and delete the withdraw in case the auction functions fail because of a blacklisted USDC address.

sanandnarayan

this is almost what the fix does. Instead of sending the crabV2 back, the blacklisted user can claim it / dequeue it themselves. And since we reject/delete the withdraw the auction can be resubmitted with a lesser amount / the next withdraw amounts could be used

thec00n

Ah yes indeed.

jacksanford1

Fix accepted

Issue H-3: Adverary can DOS contract by making a large number of deposits/withdraws then removing them all

Source: https://github.com/sherlock-audit/2022-11-opyn-judging/issues/148

Found by

rvierdiiev, hyh, Jeiwan, yixxas, 0x52, Met, indijanc, chainNue, rotcivegaf, adriro, KingNFT

Summary

When a user dequeues a withdraw or deposit it leaves a blank entry in the withdraw/deposit. This entry must be read from memory and skipped when processing the withdraws/deposits which uses gas for each blank entry. An adversary could exploit this to DOS the contract. By making a large number of these blank deposits they could make it impossible to process any auction.

Vulnerability Detail

```
while (_quantity > 0) {
    Receipt memory deposit = deposits[i];
    if (deposit.amount == 0) {
        i++;
        continue;
    }
    if (deposit.amount <= _quantity) {
        // deposit amount is lesser than quantity use it fully
        _quantity = _quantity - deposit.amount;
        usdBalance[deposit.sender] -= deposit.amount;
        amountToSend = (deposit.amount * 1e18) / _price;
        IERC20(crab).transfer(deposit.sender, amountToSend);
        emit USDCDeposited(deposit.sender, deposit.amount, amountToSend, i, 0);
        delete deposits[i];
        i++;
    } else {
        // deposit amount is greater than quantity; use it partially
        deposits[i].amount = deposit.amount - _quantity;
        usdBalance[deposit.sender] -= _quantity;
        amountToSend = (_quantity * 1e18) / _price;
        IERC20(crab).transfer(deposit.sender, amountToSend);
        emit USDCDeposited(deposit.sender, _quantity, amountToSend, i, 0);
        _quantity = 0;
    }
}
```

The code above processes deposits in the order they are submitted. An adversary can exploit this by withdrawing/depositing a large number of times then dequeuing them to create a larger number of blank deposits. Since these are all zero, it creates a fill or kill scenario. Either all of them are skipped or none. If the adversary makes the list long enough then it will be impossible to fill without going over block gas limit.

Impact

Contract can be permanently DOS'd

Code Snippet

https://github.com/sherlock-audit/2022-11-opyn/blob/main/crab-netting/src/CrabNetting.sol#L362-L386

Tool used

Manual Review

Recommendation

Two potential solutions. The first would be to limit the number of deposits/withdraws that can be processed in a single netting. The second would be to allow the owner to manually skip withdraws/deposits by calling an function that increments depositsIndex and withdrawsIndex.

Discussion

sanandnarayan

fix: https://github.com/opynfinance/squeeth-monorepo/pull/805

thec00n

Allows the owner to set the `withdrawsIndex` and `depositIndex` and so to skip the queue index forward if there is some issue with the queue elements. At least this should allow the auction functions to continue to work.

jacksanford1

FIx accepted

Issue H-4: Orders from other market makers can be invalidated

Source: https://github.com/sherlock-audit/2022-11-opyn-judging/issues/6

Found by

thec00n, cccz, John, hyh, Atarpara, aviggiano, HonorLt, jonatascm, indijanc, reassor, rotcivegaf, Haruxe, minhtrng, adriro

Summary

The `checkOrder()` function performs verification of pre-signed orders. This function allows anyone to set the status of an order as used by storing the nonce contained in the order. Orders and their respective nonce can only be used once.

Vulnerability Detail

The `_useNonce()` function is called as called as part of the `checkOrder()` function. It checks that the nonce of a trader has not already been used, marks the nonce as used and performs other order verification checks. Orders and their respective nonce are also checked by the same implementation as part of the auction functions `withdrawAuction()` and `depositAuction()`. An order that has been invalidated once can not be used anymore and by calling `checkOrder()` any user can invalidate existing orders.

Impact

A malicious user could perform a grieving attack and invalidate any presigned orders by monitoring the mempool and front run any orders that are submitted to `withdrawAuction()` and `depositAuction()` and send them to `checkOrder()`. One invalidated order can cause the auction functions to fail.

Code Snippet

https://github.com/sherlock-audit/2022-11-opyn/blob/main/crab-netting/src/CrabNetting.sol#L447-L476

https://github.com/sherlock-audit/2022-11-opyn/blob/main/crab-netting/src/CrabNetting.sol#L756-L759

Tool used

Manual Review

Recommendation

1.) Change the `checkOrder()` and `_checkOrder()` to a view function 2.) Remove `_useNonce()` from `_checkOrder()` 3.) Use `_useNonce()` and `_checkOrder()` in `withdrawAuction()` and `depositAuction()`

Discussion

sanandnarayan

fix https://github.com/opynfinance/squeeth-monorepo/pull/806

thec00n

Fix lgtm.

Issue M-1: Precision is lost in depositAuction and withdrawAuction user amount due calculations

Source: https://github.com/sherlock-audit/2022-11-opyn-judging/issues/201

Found by

yixxas, CRYP70, hyh

Summary

Formulas for `usdcAmount`, `portion.crab`, `portion.eth` used in depositAuction() and withdrawAuction() for queued distributions perform division first, which lead to truncation and fund loss in the numerical corner cases.

Vulnerability Detail

depositAuction() and withdrawAuction() use the same approach for USDC and crab amount calculation. Let's focus on withdrawAuction(), there it is `usdcAmount=(((withdraw.amount*1e18)/_p.crabToWithdraw)*usdcReceived)/1e18`.

When `_p.crabToWithdraw` is big compared to `withdraw.amount`, the `((withdraw.amount*1e18)/_p.crabToWithdraw)` can become zero as result of integer division.

As an example there can be an ordinary user and a whale situation, for the user it can be `withdraw.amount=900`, while `_p.crabToWithdraw=1000e18`, `usdcReceived=2e18`, then `usdcAmount=(((withdraw.amount*1e18)/_p.crabToWithdraw)*usdcReceived)/1e18=0`, while it should be `usdcAmount=(withdraw.amount*usdcReceived)/_p.crabToWithdraw=(900*2e18)/1000e18=1`.

Impact

When truncation occurs the corresponding depositor or withdrawer will experience the loss as less funds to be distributed to them.

Setting the severity to medium as this have material impact in a numerical corner cases only.

Code Snippet

withdrawAuction() use `usdcAmount=(((withdraw.amount*1e18)/_p.crabToWithdraw)*usdcReceived)/1e18`:

https://github.com/sherlock-audit/2022-11-opyn/blob/main/crab-netting/src/CrabNetting.sol#L697-L718

```
if (withdraw.amount <= remainingWithdraws) {
    // full usage
    remainingWithdraws -= withdraw.amount;
    crabBalance[withdraw.sender] -= withdraw.amount;

    // send proportional usdc
    usdcAmount = (((withdraw.amount * 1e18) / _p.crabToWithdraw) * usdcReceived)
↪   / 1e18;
    IERC20(usdc).transfer(withdraw.sender, usdcAmount);
    emit CrabWithdrawn(withdraw.sender, withdraw.amount, usdcAmount, j);
    delete withdraws[j];
    j++;
} else {
    withdraws[j].amount -= remainingWithdraws;
    crabBalance[withdraw.sender] -= remainingWithdraws;

    // send proportional usdc
    usdcAmount = (((remainingWithdraws * 1e18) / _p.crabToWithdraw) *
↪   usdcReceived) / 1e18;
    IERC20(usdc).transfer(withdraw.sender, usdcAmount);
    emit CrabWithdrawn(withdraw.sender, remainingWithdraws, usdcAmount, j);

    remainingWithdraws = 0;
}
```

depositAuction() use the same approach for `portion.crab` and `portion.eth`:

https://github.com/sherlock-audit/2022-11-opyn/blob/main/crab-netting/src/CrabNetting.sol#L584-L618

```
if (queuedAmount <= remainingDeposits) {
    remainingDeposits = remainingDeposits - queuedAmount;
    usdBalance[deposits[k].sender] -= queuedAmount;

    portion.crab = (((queuedAmount * 1e18) / _p.depositsQueued) * to_send.crab) /
↪   1e18;

    IERC20(crab).transfer(deposits[k].sender, portion.crab);

    portion.eth = (((queuedAmount * 1e18) / _p.depositsQueued) * to_send.eth) /
↪   1e18;
    if (portion.eth > 1e12) {
        IWETH(weth).transfer(deposits[k].sender, portion.eth);
    } else {
        portion.eth = 0;
    }
    emit USDCDeposited(deposits[k].sender, queuedAmount, portion.crab, k,
 ↪  portion.eth);

    delete deposits[k];
    k++;
} else {
    usdBalance[deposits[k].sender] -= remainingDeposits;

    portion.crab = (((remainingDeposits * 1e18) / _p.depositsQueued) *
 ↪  to_send.crab) / 1e18;
    IERC20(crab).transfer(deposits[k].sender, portion.crab);

    portion.eth = (((remainingDeposits * 1e18) / _p.depositsQueued) *
 ↪  to_send.eth) / 1e18;
    if (portion.eth > 1e12) {
        IWETH(weth).transfer(deposits[k].sender, portion.eth);
    } else {
        portion.eth = 0;
    }
    emit USDCDeposited(deposits[k].sender, remainingDeposits, portion.crab, k,
 ↪  portion.eth);

    deposits[k].amount -= remainingDeposits;
    remainingDeposits = 0;
}
```

Tool used

Manual Review

Recommendation

Consider performing multiplication first in all the case, for example for withdrawAuction():

https://github.com/sherlock-audit/2022-11-opyn/blob/main/crab-netting/src/CrabNetting.sol#L697-L718

```
            if (withdraw.amount <= remainingWithdraws) {
                // full usage
                remainingWithdraws -= withdraw.amount;
                crabBalance[withdraw.sender] -= withdraw.amount;

                // send proportional usdc
-               usdcAmount = (((withdraw.amount * 1e18) / _p.crabToWithdraw) *
 ↪  usdcReceived) / 1e18;
+               usdcAmount = (withdraw.amount * usdcReceived) /
 ↪  _p.crabToWithdraw;
                IERC20(usdc).transfer(withdraw.sender, usdcAmount);
                emit CrabWithdrawn(withdraw.sender, withdraw.amount, usdcAmount,
 ↪  j);
                delete withdraws[j];
                j++;
            } else {
                withdraws[j].amount -= remainingWithdraws;
                crabBalance[withdraw.sender] -= remainingWithdraws;

                // send proportional usdc
-               usdcAmount = (((remainingWithdraws * 1e18) / _p.crabToWithdraw) *
 ↪  usdcReceived) / 1e18;
+               usdcAmount = (remainingWithdraws * usdcReceived) /
 ↪  _p.crabToWithdraw;
                IERC20(usdc).transfer(withdraw.sender, usdcAmount);
                emit CrabWithdrawn(withdraw.sender, remainingWithdraws,
 ↪  usdcAmount, j);

                remainingWithdraws = 0;
            }
```

`withdraw.amount` and `_p.crabToWithdraw` have 18 decimals here, `usdcReceived` and resulting `usdcAmount` have 6 decimals.

Discussion

sanandnarayan

fix : https://github.com/opynfinance/squeeth-monorepo/pull/804

thec00n

Fix lgtm.

I think this should be set to medium severity, as the author suggests.

Issue M-2: Denial of Service - userDepositIndex and userWithdrawIndex growing indefinitely

Source: https://github.com/sherlock-audit/2022-11-opyn-judging/issues/156

Found by

Trumpero, HonorLt, __141345__, csanuragjain, zapaz, indijanc, reassor, Met, KingNFT

Summary

State variables `userDepositIndex` and `userWithdrawIndex` are growing indefinitely, this might lead to expensive transactions and effectively denial of service for the user when calling `withdrawUSCD` and `dequeueCrabs` functions that require iterations over the whole arrays.

Vulnerability Detail

The contract is using `userDepositsIndex` and `userWithdrawsIndex` to track deposits and withdraw for users. This helps with parsing state variables `deposits` and `withdraws` that contain data for all users. `userDepositsIndex` is expanded by `depositUSDC` function and the elements are deleted by `withdrawUSDC`, however the delete is only setting data located at the given index to zero. This make all element re-parsed every time the function is called, making the user consuming more gas. The very same logic is present within the function `queueCrabForWithdrawal` and `dequeueCrabs`. The first one will make the `userWithdrawsIndex` grow and the second will just zero-out the elements but keep parsing them every time the user call the function.

Impact

Denial of service/very expensive transactions for users of the protocol.

Code Snippet

- https://github.com/sherlock-audit/2022-11-opyn/blob/main/crab-netting/src/CrabNetting.sol#L288-L300
- https://github.com/sherlock-audit/2022-11-opyn/blob/main/crab-netting/src/CrabNetting.sol#L331-L343

Tool used

Manual Review

Recommendation

It is recommended to remove elements from the arrays `userDepositsIndex`/`userWithdrawsIndex` using `pop()` function when deleting deposits. This should be easy to implement since the iteration starts from last item and goes down until first element.

Discussion

sanandnarayan

fix: https://github.com/opynfinance/squeeth-monorepo/pull/799

thec00n

Fix LGTM.

Issue M-3: User withdrawals are dependent on admin actions

Source: https://github.com/sherlock-audit/2022-11-opyn-judging/issues/83

Found by

thec00n, Jeiwan, dipp, Deivitto

Summary

Users can deposit USDC and Crabv2 tokens at any time, but there are limitations around withdrawals. Users could have permanently locked up their funds if specific owner actions are not triggered.

Vulnerability Detail

The owner can call `toggleAuctionLive()` and prevent any withdrawals from occurring. User withdrawals are only enabled again when the owner calls `withdrawAuction()` or `depositAuction()`. If the owner loses their key or becomes malicious and never calls these functions, then the users have no way of withdrawing their funds.

Impact

Users could get their funds locked up in the `Netting` contract without a way to withdraw them again.

Code Snippet

https://github.com/sherlock-audit/2022-11-opyn/blob/main/crab-netting/src/CrabNetting.sol#L276-L283

https://github.com/sherlock-audit/2022-11-opyn/blob/main/crab-netting/src/CrabNetting.sol#L321-L327

https://github.com/sherlock-audit/2022-11-opyn/blob/main/crab-netting/src/CrabNetting.sol#L223-L226

Tool used

Manual Review

Recommendation

Lock up times are necessary for the system to work but users should always be able to withdraw their funds eventually without any dependecy of the owner. When users deposit tokens, a meaningful expiry timestamp should be set by the contract. Before the expiry deposits are locked and the funds can be used during auctions. After expiry deposits are skipped and users can withdraw them at any time.

Discussion

sanandnarayan

fix: https://github.com/opynfinance/squeeth-monorepo/pull/807

thec00n

It covers at least the case where the key or signing powers are lost. Owner could still push out the `turnOffAuctionLive()` time threshold by calling `toggleAuctionLive()` twice just before the auction can be turned off by users. Just saying ...

sanandnarayan

alternate fix: https://github.com/opynfinance/squeeth-monorepo/pull/809

jacksanford1

Comment for report: thec00n accepts the alternate fix.

Issue M-4: Used orders or revoked token authorizations can cause `withdrawAuction` and `depositAuction` to fail

Source: https://github.com/sherlock-audit/2022-11-opyn-judging/issues/60

Found by

thec00n, kaliberpoziomka, hansfriese, bin2chen, ctf_sec, seyni, __141345__, zimu, chainNue, Haruxe, imare, caventa

Summary

The owner must ensure that all orders are valid before submitting an auction, as a single order failure can revert an entire auction. The current implementation allows a market maker to invalidate their order by front-running an auction transaction, causing the auction to fail. Other ways to cause the auction functions to fail are listed below.

Vulnerability Detail

A market maker can invalidate their order when `withdrawAuction()` and `depositAuction()` is submitted from the owner by:

- setting the nonce of their order as used by calling `setNonceTrue()` or by calling `checkOrder()` and setting the nonce of orders as used (see https://github.com/sherlock-audit/2022-11-opyn-thec00n/issues/1).
- By revoking permissions to transfer tokens for the `CrabNetting` contract or transferring required tokens from the trading account so that the transfer fails.

Large user withdrawals could also occur right before the auction is submitted which could could cause the auction functions to fail.

Impact

A malicious market maker or user could perform a griefing attack and repeatedly cause auctions to fail.

Code Snippet

https://github.com/sherlock-audit/2022-11-opyn/blob/main/crab-netting/src/CrabNetting.sol#L756-L759

https://github.com/sherlock-audit/2022-11-opyn/blob/main/crab-netting/src/CrabNetting.sol#L507-L524

https://github.com/sherlock-audit/2022-11-opyn/blob/main/crab-netting/src/CrabNetting.sol#L645-L656

Tool used

Manual Review

Recommendation

- The function `_useNonce()` should not fail when the nonce has been used. Instead, return a flag with the status of the nonce update. Orders that already have used nonces should be skipped from processing in `withdrawAuction()` and `depositAuction()`.
- The auction functions should check if the market maker has sufficient balance and if the `CrabNetting` contract is authorized to successfully perform the transfer from a related order. Insufficient permissions or funds from users that sign orders should also be skipped.
- Require that auction `isAuctionLive` is set to true for `withdrawAuction()` and `depositAuction()` so withdrawals can not occur during auctions.

Discussion

sanandnarayan

So , before calling the auction function we actually check if the market makers have adequate approval and balance. if a market maker cancels approval or reduces balance or cancels nonce. Then we remove that order and resubmit the auction

sanandnarayan

So , before calling the auction function we actually check if the market makers have adequate approval and balance. if a market maker cancels approval or reduces balance or cancels nonce. Then we remove that order and resubmit the auction transaction

thec00n

Yes but it might lead to failed transactions. It's better to check anything that could fail onchain and then skip the order if it's not valid.

I also think that this issue should be medium not high.

sanandnarayan

Agree regarding the technicality. Practically, market makers / auction participants don't grieve , so we are okay with current implementation. I acknowledge the issue