SHERLOCK

# SHERLOCK SECURITY REVIEW FOR

SHERLOCK

| | |
|---|---|
| **Prepared for:** | Rage Trade |
| **Prepared by:** | Sherlock |
| **Lead Security Expert:** | 0x52 |
| **Dates Audited:** | October 31 - November 14, 2022 |
| **Prepared on:** | November 23, 2022 |

# Introduction

Rage Trade is building the most liquid, composable, and only omnichain ETH perp (powered by UNI v3).

## Scope

The following contracts in the RageTrade/delta-neutral-gmx-vaults @ 8bea1afbe746387b1a66ea9357bd41fb1c74830b repo are in scope.

- `ERC4626/ERC4626Upgradeable.sol`
- `libraries/DnGmxJuniorVaultManager.sol`
- `libraries/FeeSplitStrategy.sol`
- `libraries/SafeCast.sol`
- `periphery/WithdrawPeriphery.sol`
- `vaults/DnGmxBatchingManager.sol`
- `vaults/DnGmxJuniorVault.sol`
- `vaults/DnGmxSeniorVault.sol`

## Findings

Each issue has an assigned severity:

- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- High issues are directly exploitable security vulnerabilities that need to be fixed.

## Issues found

| Medium | High |
|---|---|
| 6 | 2 |

## Issues not fixed or acknowledged

| Medium | High |
|---|---|
| 0 | 0 |

## Security experts who found valid issues

0x52
clems4ever
GimelSec
cccz
Nyx
simon135
ctf_sec
rvierdiiev
joestakey
__141345__
tives
peanuts

# Issue H-1: If a user approves junior vault tokens to WithdrawPeriphery, anyone can withdraw/redeem his/her token

Source: https://github.com/sherlock-audit/2022-10-rage-trade-judging/issues/79

## Found by

cccz, Nyx, simon135, clems4ever, GimelSec

## Summary

If users want to withdraw/redeem tokens by WithdrawPeriphery, they should approve token approval to WithdrawPeriphery, then call `withdrawToken()` or `redeemToken()`. But if users approve `dnGmxJuniorVault` to WithdrawPeriphery, anyone can withdraw/redeem his/her token.

## Vulnerability Detail

Users should approve `dnGmxJuniorVault` before calling `withdrawToken()` or `redeemToken()`:

```
    function withdrawToken(
        address from,
        address token,
        address receiver,
        uint256 sGlpAmount
    ) external returns (uint256 amountOut) {
        // user has approved periphery to use junior vault shares
        dnGmxJuniorVault.withdraw(sGlpAmount, address(this), from);
...

    function redeemToken(
        address from,
        address token,
        address receiver,
        uint256 sharesAmount
    ) external returns (uint256 amountOut) {
        // user has approved periphery to use junior vault shares
        dnGmxJuniorVault.redeem(sharesAmount, address(this), from);
...
```

For better user experience, we always use `approve(WithdrawPeriphery,type(uint256).max)`. It means that if Alice approves the max amount, anyone can withdraw/redeem her tokens anytime. Another scenario is that if Alice approves 30 amounts, she wants

to call `withdrawToken` to withdraw 30 tokens. But in this case Alice should send two transactions separately, then an attacker can frontrun `withdrawToken` transaction and withdraw Alice's token.

## Impact

Attackers can frontrun withdraw/redeem transactions and steal tokens. And some UI always approves max amount, which means that anyone can withdraw users tokens.

## Code Snippet

https://github.com/sherlock-audit/2022-10-rage-trade/blob/main/dn-gmx-vaults/contracts/periphery/WithdrawPeriphery.sol#L119-L120 https://github.com/sherlock-audit/2022-10-rage-trade/blob/main/dn-gmx-vaults/contracts/periphery/WithdrawPeriphery.sol#L139-L140

## Tool used

Manual Review

## Recommendation

Replace `from` parameter by `msg.sender`.

```
// user has approved periphery to use junior vault shares
dnGmxJuniorVault.withdraw(sGlpAmount, address(this), msg.sender);

// user has approved periphery to use junior vault shares
dnGmxJuniorVault.redeem(sharesAmount, address(this), msg.sender);
```

# Issue H-2: DnGmxJuniorVaultManager#_rebalanceBorrow logic is flawed and could result in vault liquidation

Source: https://github.com/sherlock-audit/2022-10-rage-trade-judging/issues/62

## Found by

0x52

## Summary

DnGmxJuniorVaultManager#_rebalanceBorrow fails to rebalance correctly if only one of the two assets needs a rebalance. In the case where one assets increases rapidly in price while the other stays constant, the vault may be liquidated.

## Vulnerability Detail

```
// If both eth and btc swap amounts are not beyond the threshold then no
↪  flashloan needs to be executed | case 1
if (btcAssetAmount == 0 && ethAssetAmount == 0) return;

if (repayDebtBtc && repayDebtEth) {
    // case where both the token assets are USDC
    // only one entry required which is combined asset amount for both tokens
    assets = new address[](1);
    amounts = new uint256[](1);

    assets[0] = address(state.usdc);
    amounts[0] = (btcAssetAmount + ethAssetAmount);
} else if (btcAssetAmount == 0 || ethAssetAmount == 0) {
    // Exactly one would be true since case-1 excluded (both false) | case-2
    // One token amount = 0 and other token amount > 0
    // only one entry required for the non-zero amount token
    assets = new address[](1);
    amounts = new uint256[](1);

    if (btcAssetAmount == 0) {
        assets[0] = (repayDebtBtc ? address(state.usdc) : address(state.wbtc));
        amounts[0] = btcAssetAmount;
    } else {
        assets[0] = (repayDebtEth ? address(state.usdc) : address(state.weth));
        amounts[0] = ethAssetAmount;
    }
```

The logic above is used to determine what assets to borrow using the flashloan. If

the rebalance amount is under a threshold then the assetAmount is set equal to zero. The first check `if(btcAssetAmount==0ethAssetAmount==0)return;` is a short circuit that returns if neither asset is above the threshold. The third check `elseif(btcAssetAmount==0||ethAssetAmount==0)` is the point of interest. Since we short circuit if both are zero then to meet this condition exactly one asset needs to be rebalanced. The logic that follows is where the error is. In the comments it indicates that it needs to enter with the non-zero amount token but the actual logic reflects the opposite. If `btcAssetAmount==0` it actually tries to enter with wBTC which would be the zero amount asset.

The result of this can be catastrophic for the vault. If one token increases in value rapidly while the other is constant the vault will only ever try to rebalance the one token but because of this logical error it will never actually complete the rebalance. If the token increase in value enough the vault would actually end up becoming liquidated.

## Impact

Vault is unable to rebalance correctly if only one asset needs to be rebalanced, which can lead to the vault being liquidated

## Code Snippet

https://github.com/sherlock-audit/2022-10-rage-trade/blob/main/dn-gmx-vaults/contracts/libraries/DnGmxJuniorVaultManager.sol#L353-L458

## Tool used

Manual Review

## Recommendation

Small change to reverse the logic and make it correct:

```
-       if (btcAssetAmount == 0) {
+       if (btcAssetAmount != 0) {
            assets[0] = (repayDebtBtc ? address(state.usdc) :
            ↪  address(state.wbtc));
            amounts[0] = btcAssetAmount;
        } else {
            assets[0] = (repayDebtEth ? address(state.usdc) :
            ↪  address(state.weth));
            amounts[0] = ethAssetAmount;
        }
```

# Issue M-1: DnGmxJuniorVaultManager#harvestFees can push junior vault borrowedUSDC above borrow cap and DOS vault

Source: https://github.com/sherlock-audit/2022-10-rage-trade-judging/issues/67

## Found by

0x52

## Summary

DnGmxJuniorVaultManager#harvestFees grants fees to the senior vault by converting the WETH to USDC and staking it directly. The result is that the senior vault gains value indirectly by increasing the debt of the junior vault. If the junior vault is already at it's borrow cap this will push it's total borrow over the borrow cap causing DnGmxSeniorVault#availableBorrow to underflow and revert. This is called each time a user deposits or withdraws from the junior vault meaning that the junior vault can no longer deposit or withdraw.

## Vulnerability Detail

```
if (_seniorVaultWethRewards > state.wethConversionThreshold) {
    // converts senior tranche share of weth into usdc and deposit into AAVE
    // Deposit aave vault share to AAVE in usdc
    uint256 minUsdcAmount = _getTokenPriceInUsdc(state, state.weth).mulDivDown(
        _seniorVaultWethRewards * (MAX_BPS - state.slippageThresholdSwapEthBps),
        MAX_BPS * PRICE_PRECISION
    );
    // swaps weth into usdc
    (uint256 aaveUsdcAmount, ) = state._swapToken(
        address(state.weth),
        _seniorVaultWethRewards,
        minUsdcAmount
    );

    // supplies usdc into AAVE
    state._executeSupply(address(state.usdc), aaveUsdcAmount);

    // resets senior tranche rewards
    state.seniorVaultWethRewards = 0;
```

The above lines converts the WETH owed to the senior vault to USDC and deposits it into Aave. Increasing the aUSDC balance of the junior vault.

```
function getUsdcBorrowed() public view returns (uint256 usdcAmount) {
    return
        uint256(
            state.aUsdc.balanceOf(address(this)).toInt256() -
                state.dnUsdcDeposited -
                state.unhedgedGlpInUsdc.toInt256()
        );
}
```

The amount of USDC borrowed is calculated based on the amount of aUSDC that the junior vault has. By depositing the fees directly above, the junior vault has effectively "borrowed" more USDC. This can be problematic if the junior vault is already at it's borrow cap.

```
function availableBorrow(address borrower) public view returns (uint256
↪  availableAUsdc) {
    uint256 availableBasisCap = borrowCaps[borrower] -
    ↪  IBorrower(borrower).getUsdcBorrowed();
    uint256 availableBasisBalance = aUsdc.balanceOf(address(this));

    availableAUsdc = availableBasisCap < availableBasisBalance ?
    ↪  availableBasisCap : availableBasisBalance;
}
```

If the vault is already at it's borrow cap then the line calculating `availableBasisCap` will underflow and revert.

## Impact

availableBorrow will revert causing deposits/withdraws to revert

## Code Snippet

https://github.com/sherlock-audit/2022-10-rage-trade/blob/main/dn-gmx-vaults/contracts/vaults/DnGmxSeniorVault.sol#L350-L355

## Tool used

Manual Review

## Recommendation

Check if borrowed exceeds borrow cap and return zero to avoid underflow:

```
function availableBorrow(address borrower) public view returns (uint256
↪  availableAUsdc) {

+   uint256 borrowCap = borrowCaps[borrower];
+   uint256 borrowed = IBorrower(borrower).getUsdcBorrowed();

+   if (borrowed > borrowCap) return 0;

+   uint256 availableBasisCap = borrowCap - borrowed;

-   uint256 availableBasisCap = borrowCaps[borrower] -
↪  IBorrower(borrower).getUsdcBorrowed();
    uint256 availableBasisBalance = aUsdc.balanceOf(address(this));

    availableAUsdc = availableBasisCap < availableBasisBalance ?
    ↪  availableBasisCap : availableBasisBalance;
}
```

## Discussion

**0x00052**

Typo. Only meant to mention #52

# Issue M-2: Wrong price calculation in DnGmxJuniorVaultManager.sol

Source: https://github.com/sherlock-audit/2022-10-rage-trade-judging/issues/61

## Found by

clems4ever

## Summary

in `DnGmxJuniorVaultManager.sol` at line:647: `usdcPrice` should be on denominator and `MAX_PRECISION` on numerator (cf pricing in Vault: uint256 redemptionAmount = _usdgAmount.mul(PRICE_PRECISION).div(price);)

https://github.com/sherlock-audit/2022-10-rage-trade/blob/main/dn-gmx-vaults/contracts/libraries/DnGmxJuniorVaultManager.sol#L646

## Vulnerability Detail

## Impact

In the case usdcPrice is higher than 1$ (which already happened in reasonable market circumstances). Min amount expected will be higher than swap result under 0% slippage conditions. The call will revert, which will delay rebalances until usdcPrice comes back to 1$, and causing potential loss to the protocol.

## Code Snippet

## Tool used

Manual Review

## Recommendation

Recommendation in summary

## Discussion

**0xDosa**

The suggested fix seems to be incorrect. Since we already are passing the value in usdc amount hence neither multiplication nor division with usdcPrice should be required.

**0x00052**

Agreed, suggested fix is incorrect. It should only adjust for slippage

# Issue M-3: WithdrawPeriphery#_convertToToken slippage control is broken for any token other than USDC

Source: https://github.com/sherlock-audit/2022-10-rage-trade-judging/issues/55

## Found by

0x52

## Summary

WithdrawPeriphery allows the user to redeem junior share vaults to any token available on GMX, applying a fixed slippage threshold to all redeems. The slippage calculation always returns the number of tokens to 6 decimals. This works fine for USDC but for other tokens like WETH or WBTC that are 18 decimals the slippage protection is completely ineffective and can lead to loss of funds for users that are withdrawing.

## Vulnerability Detail

```
function _convertToToken(address token, address receiver) internal returns
↪  (uint256 amountOut) {
    // this value should be whatever glp is received by calling withdraw/redeem
    ↪  to junior vault
    uint256 outputGlp = fsGlp.balanceOf(address(this));

    // using min price of glp because giving in glp
    uint256 glpPrice = _getGlpPrice(false);

    // using max price of token because taking token out of gmx
    uint256 tokenPrice = gmxVault.getMaxPrice(token);

    // apply slippage threshold on top of estimated output amount
    uint256 minTokenOut = outputGlp.mulDiv(glpPrice * (MAX_BPS -
    ↪  slippageThreshold), tokenPrice * MAX_BPS);

    // will revert if atleast minTokenOut is not received
    amountOut = rewardRouter.unstakeAndRedeemGlp(address(token), outputGlp,
    ↪  minTokenOut, receiver);
}
```

WithdrawPeriphery allows the user to redeem junior share vaults to any token available on GMX. To prevent users from losing large amounts of value to MEV the contract applies a fixed percentage slippage. minToken out is returned to 6 decimals regardless of the token being requested. This works for tokens with 6 decimals like USDC, but is completely ineffective for the majority of tokens that aren't.

## Impact

Users withdrawing tokens other than USDC can suffer huge loss of funds due to virtually no slippage protection

## Code Snippet

https://github.com/sherlock-audit/2022-10-rage-trade/blob/main/dn-gmx-vaults/contracts/periphery/WithdrawPeriphery.sol#L147-L161

## Tool used

Manual Review

## Recommendation

Adjust minTokenOut to match the decimals of the token:

```
    uint256 minTokenOut = outputGlp.mulDiv(glpPrice * (MAX_BPS -
    ↪  slippageThreshold), tokenPrice * MAX_BPS);
+   minTokenOut = minTokenOut * 10 ** (token.decimals() - 6);
```

## Discussion

**0xDosa**

Agreed on the issue but the severity level should be medium since loss of funds is not possible. While swapping on GMX, there is min-max spread and fees but no slippage due to them using chainlink oracles for pricing the tokens, so a direct sandwich attack would not work.

**Evert0x**

Downgrading to medium

# Issue M-4: WithdrawPeriphery uses incorrect value for MAX_BPS which will allow much higher slippage than intended

Source: https://github.com/sherlock-audit/2022-10-rage-trade-judging/issues/39

## Found by

0x52

## Summary

WithdrawPeriphery accidentally uses an incorrect value for MAX_BPS which will allow for much higher slippage than intended.

## Vulnerability Detail

```
uint256 internal constant MAX_BPS = 1000;
```

BPS is typically 10,000 and using 1000 is inconsistent with the rest of the ecosystem contracts and tests. The result is that slippage values will be 10x higher than intended.

## Impact

Unexpected slippage resulting in loss of user funds, likely due to MEV

## Code Snippet

https://github.com/sherlock-audit/2022-10-rage-trade/blob/main/dn-gmx-vaults/contracts/periphery/WithdrawPeriphery.sol#L47

## Tool used

Manual Review

## Recommendation

Correct MAX_BPS:

```
-   uint256 internal constant MAX_BPS = 1000;
+   uint256 internal constant MAX_BPS = 10_000;
```

# Issue M-5: Early depositors to DnGmxSeniorVault can manipulate exchange rates to steal funds from later depositors

Source: https://github.com/sherlock-audit/2022-10-rage-trade-judging/issues/37

## Found by

ctf_sec, cccz, __141345__, tives, peanuts, joestakey, clems4ever, rvierdiiev, GimelSec, 0x52

## Summary

To calculate the exchange rate for shares in DnGmxSeniorVault it divides the total supply of shares by the totalAssets of the vault. The first deposit can mint a very small number of shares then donate aUSDC to the vault to grossly manipulate the share price. When later depositor deposit into the vault they will lose value due to precision loss and the adversary will profit.

## Vulnerability Detail

```
function convertToShares(uint256 assets) public view virtual returns (uint256) {
    uint256 supply = totalSupply(); // Saves an extra SLOAD if totalSupply is
    ↪  non-zero.

    return supply == 0 ? assets : assets.mulDivDown(supply, totalAssets());
}
```

Share exchange rate is calculated using the total supply of shares and the totalAsset. This can lead to exchange rate manipulation. As an example, an adversary can mint a single share, then donate 1e8 aUSDC. Minting the first share established a 1:1 ratio but then donating 1e8 changed the ratio to 1:1e8. Now any deposit lower than 1e8 (100 aUSDC) will suffer from precision loss and the attackers share will benefit from it.

This same vector is present in DnGmxJuniorVault.

## Impact

Adversary can effectively steal funds from later users

## Code Snippet

https://github.com/sherlock-audit/2022-10-rage-trade/blob/main/dn-gmx-vaults/contracts/vaults/DnGmxSeniorVault.sol#L211-L221

## Tool used

Manual Review

## Recommendation

Initialize should include a small deposit, such as 1e6 aUSDC that mints the share to a dead address to permanently lock the exchange rate:

```
    aUsdc.approve(address(pool), type(uint256).max);
    IERC20(asset).approve(address(pool), type(uint256).max);

+   deposit(1e6, DEAD_ADDRESS);
```

## Discussion

**0xDosa**

We will ensure a guarded launch process that safeguards the first deposit to avoid being manipulated.

**Evert0x**

We are still considering it a valid issue as the guarded launch process is out of scope.

# Issue M-6: DnGmxJuniorVaultManager#_totalAssets current implementation doesn't properly maximize or minimize

Source: https://github.com/sherlock-audit/2022-10-rage-trade-judging/issues/36

## Found by

0x52

## Summary

The maximize input to DnGmxJuniorVaultManager#_totalAssets indicates whether to either maximize or minimize the NAV. Internal logic of the function doesn't accurately reflect that because under some circumstances, maximize = true actually returns a lower value than maximize = false.

## Vulnerability Detail

```
uint256 unhedgedGlp = (state.unhedgedGlpInUsdc + dnUsdcDepositedPos).mulDivDown(
    PRICE_PRECISION,
    _getGlpPrice(state, !maximize)
);

// calculate current borrow amounts
(uint256 currentBtc, uint256 currentEth) = _getCurrentBorrows(state);
uint256 totalCurrentBorrowValue = _getBorrowValue(state, currentBtc, currentEth);

// add negative part to current borrow value which will be subtracted at the end
// convert usdc amount into glp amount
uint256 borrowValueGlp = (totalCurrentBorrowValue +
↪  dnUsdcDepositedNeg).mulDivDown(
    PRICE_PRECISION,
    _getGlpPrice(state, !maximize)
);

// if we need to minimize then add additional slippage
if (!maximize) unhedgedGlp = unhedgedGlp.mulDivDown(MAX_BPS -
↪  state.slippageThresholdGmxBps, MAX_BPS);
if (!maximize) borrowValueGlp = borrowValueGlp.mulDivDown(MAX_BPS -
↪  state.slippageThresholdGmxBps, MAX_BPS);
```

To maximize the estimate for the NAV of the vault underlying debt should minimized and value of held assets should be maximized. Under the current settings there is

a mix of both of those and the function doesn't consistently minimize or maximize. Consider when NAV is "maxmized". Under this scenario the value of when estimated the GlpPrice is minimized. This minimizes the value of both the borrowedGlp (debt) and of the unhedgedGlp (assets). The result is that the NAV is not maximized because the value of the assets are also minimized. In this scenario the GlpPrice should be maximized when calculating the assets and minimized when calculating the debt. The reverse should be true when minimizing the NAV. Slippage requirements are also applied incorrectly when adjusting borrowValueGlp. The current implementation implies that if the debt were to be paid back that the vault would repay their debt for less than expected. When paying back debt the slippage should imply paying more than expected rather than less, therefore the slippage should be added rather than subtracted.

## Impact

DnGmxJuniorVaultManager#_totalAssets doesn't accurately reflect NAV. Since this is used when determining critical parameters it may lead to inaccuracies.

## Code Snippet

https://github.com/sherlock-audit/2022-10-rage-trade/blob/main/dn-gmx-vaults/contracts/libraries/DnGmxJuniorVaultManager.sol#L1013-L1052

## Tool used

Manual Review

## Recommendation

To properly maximize the it should assume the best possible rate for exchanging it's assets. Likewise to minimize it should assume it's debt is a large as possible and this it encounters maximum possible slippage when repaying it's debt. I recommend the following changes:

```
    uint256 unhedgedGlp = (state.unhedgedGlpInUsdc +
    ↪  dnUsdcDepositedPos).mulDivDown(
        PRICE_PRECISION,
-       _getGlpPrice(state, !maximize)
+       _getGlpPrice(state, maximize)
    );

    // calculate current borrow amounts
    (uint256 currentBtc, uint256 currentEth) = _getCurrentBorrows(state);
    uint256 totalCurrentBorrowValue = _getBorrowValue(state, currentBtc,
    ↪  currentEth);
```

```
    // add negative part to current borrow value which will be subtracted at the
    ↪  end
    // convert usdc amount into glp amount
    uint256 borrowValueGlp = (totalCurrentBorrowValue +
    ↪  dnUsdcDepositedNeg).mulDivDown(
        PRICE_PRECISION,
        _getGlpPrice(state, !maximize)
    );

    // if we need to minimize then add additional slippage
    if (!maximize) unhedgedGlp = unhedgedGlp.mulDivDown(MAX_BPS -
    ↪  state.slippageThresholdGmxBps, MAX_BPS);
-   if (!maximize) borrowValueGlp = borrowValueGlp.mulDivDown(MAX_BPS -
↪   state.slippageThresholdGmxBps, MAX_BPS);
+   if (!maximize) borrowValueGlp = borrowValueGlp.mulDivDown(MAX_BPS +
↪   state.slippageThresholdGmxBps, MAX_BPS);
```

## Discussion

### 0xDosa

Dividing with minimum price would maximize the asset/borrow amount and vice versa. So the correct fix should be this. @0x00052 could you confirm?

```
     uint256 unhedgedGlp = (state.unhedgedGlpInUsdc +
     ↪  dnUsdcDepositedPos).mulDivDown(
        PRICE_PRECISION,
       _getGlpPrice(state, !maximize)
    );

    // calculate current borrow amounts
    (uint256 currentBtc, uint256 currentEth) = _getCurrentBorrows(state);
    uint256 totalCurrentBorrowValue = _getBorrowValue(state, currentBtc,
    ↪  currentEth);

    // add negative part to current borrow value which will be subtracted at the
    ↪  end
    // convert usdc amount into glp amount
    uint256 borrowValueGlp = (totalCurrentBorrowValue +
    ↪  dnUsdcDepositedNeg).mulDivDown(
        PRICE_PRECISION,
-       _getGlpPrice(state, !maximize)
+       _getGlpPrice(state, maximize)

    );

    // if we need to minimize then add additional slippage
```

```
    if (!maximize) unhedgedGlp = unhedgedGlp.mulDivDown(MAX_BPS -
    ↪  state.slippageThresholdGmxBps, MAX_BPS);
-   if (!maximize) borrowValueGlp = borrowValueGlp.mulDivDown(MAX_BPS -
↪   state.slippageThresholdGmxBps, MAX_BPS);
+   if (!maximize) borrowValueGlp = borrowValueGlp.mulDivDown(MAX_BPS +
↪   state.slippageThresholdGmxBps, MAX_BPS);
```

**0x00052**

Good catch! You're right, I got that backwards.