Gains Network Security Review

Pashov Audit Group

Conducted by: ast3ros, Peakbolt, Said, sashik-eth

February 23th 2024 - March 29th 2024

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of multiple teams of some of the best smart contract security researchers in the space. Having a combined reported security vulnerabilities count of over 1000, the group strives to create the absolute very best audit journey possible - although 100% security can never be guaranteed, we do guarantee the best efforts of our experienced researchers for your blockchain protocol. Check our previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Introduction

A time-boxed security review of the **gTrade-contracts** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About Gains Network

Gains Network is a liquidity-efficient decentralized leveraged trading platform. Trades are opened with DAI, USDC or WETH collateral, regardless of the trading pair. The leverage is synthetic and backed by the respective gToken vault, and the GNS token. Trader profit is taken from the vaults to pay the traders PnL (if positive), or receives trader losses from trades if their PnL was negative.

5. Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

5.1. Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- Low - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

5.2. Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

5.3. Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

6. Security Assessment Summary

review commit hash - 725b997cfc632361a0e01b3d996f63bc09783e3c

fixes review commit hash - 7de396938464d6d5e9e3d92ffa717d574e73be33

Scope

The following smart contracts were in scope of the audit:

```
- GNSAddressStore
- GNSDiamondCut
- GNSDiamondLoupe
- GNSDiamondStorage
- GNSBorrowingFees
- GNSFeeTiers
- GNSPairsStorage
- GNSPriceAggregator
- GNSPriceImpact
- GNSReferrals
- GNSTrading
- GNSTradingCallbacks
- GNSTradingStorage
- GNSTriggerRewards
- GNSMultiCollatDiamond
- AddressStoreUtils
- BorrowingFeesUtils
- ChainUtils
- ChainlinkClientUtils
- CollateralUtils
- DiamondUtils
- FeeTiersUtils
- PackingUtils
- PairsStorageUtils
- PriceAggregatorUtils
- PriceImpactUtils
- ReferralsUtils
- StorageUtils
- TradingCallbacksUtils
- TradingStorageUtils
- TradingUtils
- TriggerRewardsUtils
```

7. Executive Summary

Over the course of the security review, ast3ros, Peakbolt, Said, sashik-eth engaged with Gains Network to review Gains Network. In this period of time a total of **24** issues were uncovered.

Protocol Summary

Protocol Name	Gains Network
Repository	https://github.com/GainsNetwork-org/gTrade-contracts
Date	February 23th 2024 - March 29th 2024
Protocol Type	Leveraged trading platform

Findings Count

Severity	**Amount**
Critical	2
High	8
Medium	8
Low	6
Total Findings	**24**

Summary of Findings

ID	Title	Severity	Status
[C-01]	Liquidations prevented for non-18 decimal collaterals	Critical	Resolved
[C-02]	updateTp() and updateSl() can be abused to obtain risk-free trade	Critical	Resolved
[H-01]	Lost roles after the proxy upgrade	High	Resolved
[H-02]	ReferralsUtils.updateStartReferrerFeeP() implements wrong check	High	Resolved
[H-03]	ReferralsUtils.updateReferralsOpenFeeP() implements wrong check	High	Resolved
[H-04]	Collateral not approved after updating of GToken address	High	Resolved
[H-05]	closeTradeMarket providing wrong value when triggering getPrice	High	Resolved
[H-06]	Wrong validation in toggleCollateralActiveState()	High	Resolved
[H-07]	Cancelled MARKET_CLOSE order will affect TP/SL trigger	High	Resolved
[H-08]	Vulnerability to DOS attacks during market closures	High	Acknowledged
[M-01]	getCollaterals() does not return the last collateral	Medium	Resolved
[M-02]	Incorrect storage slot assignments	Medium	Resolved
[M-03]	Inaccurate calculation of v.positionSizeCollateralAfterReferralFees	Medium	Resolved
[M-04]	Inaccurate calculation of referral fees for referrers and allies	Medium	Acknowledged

[M-05]	Reorg incident could affect fulfilling orders	Medium	Acknowledged
[M-06]	Profitable trade positions risk unwarranted liquidation and loss of profit	Medium	Acknowledged
[M-07]	Liquidation failure for traders on USDC blacklist	Medium	Resolved
[M-08]	Trader can evade loss when Stop order is not fulfilled in time	Medium	Acknowledged
[L-01]	Unnecessary _transferCollateralToAddress() used	Low	Resolved
[L-02]	Chainlink feeds could have 18 decimals	Low	Resolved
[L-03]	s.traders will contain duplicates and affect getTraders()	Low	Resolved
[L-04]	_correctTp and _correctSl should be called when updateTrade is called	Low	Resolved
[L-05]	Extra dust amount of GNS minted	Low	Acknowledged
[L-06]	getAllTrades and getAllTradeInfos should exit the loop earlier	Low	Acknowledged

8. Findings

8.1. Critical Findings

[C-01] Liquidations prevented for non-18 decimal collaterals

Severity

Impact: High

Likelihood: High

Description

`getTradeLiquidationPrice` is called when `triggerOrder` processes a liquidation order or a pending order. It calculates the `liqPrice` and checks if the order needs to be updated to `SL_CLOSE` once the stop-loss price is reached. The `getTradeLiquidationPrice` requires the collateral amount provided to be in collateral precision, for calculating the borrowing fee and the liquidation price.

```
/**
   * @dev Pure function that returns the liquidation price for a trade
     (1e10 precision)
   * @param _openPrice trade open price (1e10 precision)
   * @param _long true if long, false if short
   * @param _collateral trade collateral (collateral precision)
   * @param _leverage trade leverage (1e3 precision)
   * @param _borrowingFeeCollateral borrowing fee amount (collateral precision)
   * @param _collateralPrecisionDelta collateral precision delta
     (10^18/10^decimals)
   */
  function _getTradeLiquidationPrice(
      uint256 _openPrice,
      bool _long,
      uint256 _collateral,
      uint256 _leverage,
      uint256 _borrowingFeeCollateral,
      uint128 _collateralPrecisionDelta
  ) internal pure returns (uint256) {
      uint256 precisionDeltaUint = uint256(_collateralPrecisionDelta);

      int256 openPriceInt = int256(_openPrice);
      // LIQ_THRESHOLD_P = 90; // -90% pnl
      int256 collateralLiqNegativePnlInt = int256(
      //(_collateral * LIQ_THRESHOLD_P * precisionDeltaUint * 1e3) / 100); // 1e18 * 1e3
      int256 borrowingFeeInt = int256
      //(_borrowingFeeCollateral * precisionDeltaUint * 1e3); // 1e18 * 1e3

      // PRECISION
      int256 liqPriceDistance = (openPriceInt *
      //(collateralLiqNegativePnlInt - borrowingFeeInt)) / // 1e10 * 1e18 * 1e3
          int256(_collateral) /
          int256(_leverage) /
          int256(precisionDeltaUint); // 1e10

      int256 liqPrice = _long ? openPriceInt - liqPriceDistance : openPriceInt +
      // liqPriceDistance; // 1e10

      return liqPrice > 0 ? uint256(liqPrice) : 0; // 1e10
  }
```

However, the collateral provided when `getTradeLiquidationPrice` from `triggerOrder` is not in collateral precision.

```
function triggerOrder(uint256 _packed) internal onlyTradingActivated(true) {
      (
        uint256_orderType,
        addresstrader,
        uint32index
      ) = _packed.unpackTriggerOrder(

      // ...

      if (orderType == ITradingStorage.PendingOrderType.LIQ_CLOSE) {
          if (t.sl > 0) {
              uint256 liqPrice = _getMultiCollatDiamond
                ().getTradeLiquidationPrice(
                  IBorrowingFees.LiqPriceInput(
                      t.collateralIndex,
                      t.user,
                      t.pairIndex,
                      t.index,
                      t.openPrice,
                      t.long,
>>>                   t.collateralAmount / collateralPrecisionDelta / PRECISION,
                      t.leverage
                  )
              );

              // If liq price not closer than SL, turn order into a SL order
              if ((t.long && liqPrice <= t.sl) ||
                (!t.long && liqPrice >= t.sl)) {
                  orderType = ITradingStorage.PendingOrderType.SL_CLOSE;
              }
          }
      } else {
          if
            (orderType == ITradingStorage.PendingOrderType.SL_CLOSE && t.sl == 0) revert IT
          if
            (orderType == ITradingStorage.PendingOrderType.TP_CLOSE && t.tp == 0) revert IT
      }

      // ...
  }
```

If the collateral is a non-18 decimal token, this could lead to a scenario where `triggerOrder` would revert, as `collateralAmount` / `precisionDelta` / `PRECISION` equals 0 in most cases, causing `getTradeLiquidationPrice` to revert due to attempting division by 0.

For USDC case :

decimals = 6 precisionDelta = 1e12 PRECISION = 1e10

So, collateral needs to return at least 1 is to be `collateralAmount = precisionDelta * PRECISION` = 1e10 x 1e12 = 1e22 or 100,000,000,000,000,000 USDC. For collateral that uses USDC and has stop loss configured and have less than that value, it will not be liquidatable.

In conclusion, trades that use USDC, WBTC, or other non-18 decimal assets as collateral and configure a stop loss will not be liquidatable in almost any scenario.

Recommendations

Provide `t.collateralAmount` instead of `t.collateralAmount / collateralPrecisionDelta / PRECISION` to `getTradeLiquidationPrice`.

[C-02] `updateTp()` and `updateSl()` can be abused to obtain risk-free trade

Severity

Impact: High

Likelihood: High

Description

When an open order is executed within `executeTriggerOpenOrderCallback()`, validation is performed in `_openTradePrep()` to cancel any invalid order. This includes checking whether the TP or SL has been met.

However, the trader is allowed to update the TP/SL as long as there are no pending `TP_CLOSE` or `SL_CLOSE` orders.

This means that the trader can abuse `updateTp()` or `updateSl()` to block the execution of the pending order even when the open price has been hit. The trader can block it till the price is favorable, which means the trader need not pay any fees as long as the execution is blocked.

When the price is favorable for the trader, the trader can update the TP/SL again to unblock the open order execution, which will then be executed at the price when the limit trade was created. The trader can then close the trade immediately for the profit.

Suppose the scenario,

1. At block `t1`, trader calls `openTrade()` to open a LIMIT order with the price X. This will trigger a `storeTrade()` where `createdBlock = t1`.
2. Trader also calls `updateTP()` at the same time, with a TP such that it will revert on `executeTriggerOpenOrderCallback()` to block all order execution. This prevents the trade from being registered even when the open price has hit.
3. When price is favorable at block `t2`, the trader calls `updateTP()` again so that it will not block execution anymore.
4. The trader can now call `triggerOrder()` again to trigger the trading callback with new oracle requests.

5. The LIMIT order will be executed and opened at the limit price, which would be met as the `triggerOrder()` will request the lookback prices based on `createdBlock = t1`.
6. Once the trade is registered, a trader can immediately close the trade to lock in the profit.

POC

Add the following test to L311 of `executeTriggerOpenOrderCallback.test.js`.

```
describe('POC for C-01', () => {
    it('[C-01] `UpdateTp()` and `UpdateSL
      ()` can be abused to obtain risk-free trade', async () => {
      limitId = [accounts[0], 0];
      tradeId = [accounts[0], 3];

      console.log("-------------------------");

      let currentBlock = await d.block.getBlockNumber();
      console.log("currentBlock       : %d\n",currentBlock);


      requestTx = await d.diamond.triggerOrder(packNft(
        packNft

      ), { from: gov }
      orderId = getOrderId(requestTx);

      console.log("calling updateTp() to block order execution for fulfill
        (), by setting a TP that will cancel order due to TP_REACHED");
       await d.diamond.updateTp(0, 1 * 1e10);

      let td = await d.diamond.getTrade(accounts[0], 0);
      let tdInfo = await d.diamond.getTradeInfo(accounts[0], 0);
      console.log("trade.tp           : %d ", td.tp);
      console.log("tpLastUpdatedBlock : %d ",tdInfo.tpLastUpdatedBlock);
      console.log("createdBlock       : %d (used by getPrice
        () to retrieve historical price)",tdInfo.createdBlock);

      const priceData1 = pack(
        [1499*1e10,
        1505*1e10,
        1495*1e10,
        dummyTsSec].map
      ), [64, 64, 64, 64].map(BigInt
      await fulfillOracleRequests(requestTx, d.oracles.slice(0, 3), priceData1);

      td = await d.diamond.getTrade(accounts[0], 3);
      console.log("\n");
      console.log("trade is not registered due to cancellation.");
      console.log("trade.isOpen           : %s ", td.isOpen);

      console.log("-------------------------");

      await d.block.increase(10);
      currentBlock = await d.block.getBlockNumber();
      console.log("currentBlock       : %d\n",currentBlock);


      console.log("Once price is favorable, call updateTp
        () to unblock the cancellation.");
      await d.diamond.updateTp(0, 2175 * 1e10);

      td = await d.diamond.getTrade(accounts[0], 0);
      tdInfo = await d.diamond.getTradeInfo(accounts[0], 0);
      console.log("trade.tp           : %d ", td.tp);
      console.log("tpLastUpdatedBlock : %d ",tdInfo.tpLastUpdatedBlock);
      console.log("createdBlock       : %d (used by getPrice
        () to retrieve historical price)",tdInfo.createdBlock);

      console.log("call triggerOrder(
        "calltriggerOrder

      ) to request oracle to fulfill order, with price from original createdBlock."
      requestTx = await d.diamond.triggerOrder(packNft(
        packNft

      ), { from: gov }
      orderId = getOrderId(requestTx);

      // ignore price data, as this POC just shows that it is possible to delay
      // order execution till price is favorable
      const priceData2 = pack(
        [1499*1e10,
        1505*1e10,
        1495*1e10,
        dummyTsSec].map
      ), [64, 64, 64, 64].map(BigInt
      await fulfillOracleRequests(requestTx, d.oracles.slice(0, 3), priceData2);

      td = await d.diamond.getTrade(accounts[0], 3);
      console.log("\n");
      console.log
        ("trade is now registered as cancellation is unblocked by trader.");
      console.log("trade.isOpen            : %s ", td.isOpen);

      console.log("\n");
      console.log(
        "nowthatpriceisfavorable,
        wecanclosethisregisteredtradeimmediatelyforaprofit."
      );


    });
```

Recommendations

In general, do not allow users to be able to influence the cancellation of any pending open order.

To fix this issue, modify `updateTp()` and `updateSl()` such that they can only be used for registered trade and not open limit orders.

Users can still use `updateOpenOrder()` instead to update TP/SL for an open limit order. That will reset the `createdBlock` and validate TP/SL.

8.2. High Findings

[H-01] Lost roles after the proxy upgrade

Severity

Impact: Medium

Likelihood: High

Description

In the current implementation of the `GNSMultiCollatDiamond` contract, the `accessControl` mapping takes slot `2` in the proxy storage: Link

The first two slots are taken by variables from the `Initializable` contract and by struct `Addresses` which currently holds only one address - `gns`.

But in the new version of the `GNSMultiCollatDiamond` contract, struct `Addresses` would take 2 more storage slots:

```
File: IAddressStore.sol
    struct Addresses {
        address gns; // GNS token address
        address gnsStaking; // GNS staking address
        address linkErc677; // ERC677 LINK token address
    }

    struct AddressStore {
        Addresses globalAddresses;
        mapping(address => mapping(Role => bool)) accessControl;
        uint256[47] __gap;
    }
```

This would put the `accessControl` mapping at slot `4`. In Solidity mapping values are addressed based on the key and storage slot number that mapping takes, meaning addressing used in the current version of `GNSMultiCollatDiamond` would be changed in the new one. This would result in losing all current roles info since `hasRole` now would address `accessControl` mapping using storage slot `4` instead of `2`.

No funds would be at risk but all role-gated functionality would be inaccessible and a new upgrade of proxy that fixes the storage layout would be required.

Next foundry test could demonstrate the described issue:

```
pragma solidity 0.8.23;

import "lib/forge-std/src/Test.sol";
import "contracts/core/GNSMultiCollatDiamond.sol";
import
   "node_modules/@openzeppelin/contracts/proxy/transparent/TransparentUpgradeableProxy.sol"

contract Unit is Test {
    function setUp() public {}

    function test_UpgradeBrokeRoles() public {
        uint256 mainnetFork = vm.createFork("https://arb1.arbitrum.io/rpc");
        vm.selectFork(mainnetFork);

        // Fetch current proxy address
        ITransparentUpgradeableProxy proxyDiamond;
        proxyDiamond = ITransparentUpgradeableProxy(
            payable(0xFF162c694eAA571f685030649814282eA457f169)
            );

        // Check that address has manager role before upgrade
        bool _hasRole = GNSAddressStore(payable(address(proxyDiamond)))
            .hasRole(
              0x1632C38cB208df8409753729dBfbA5c58626F637,
              IAddressStore.Role.ROLES_MANAGER
            );

        GNSMultiCollatDiamond newDiamondImplementation;
        newDiamondImplementation = new GNSMultiCollatDiamond();
        // Upgrading diamond with new implementation
        vm.prank(0xe18be0113c38c91b3B429d04fDeb84359fBCb2eB);
        proxyDiamond.upgradeTo(address(newDiamondImplementation));

        // Check that address has manager role after upgrade
        bool hasRole_ = GNSAddressStore(payable(address(proxyDiamond)))
            .hasRole(
              0x1632C38cB208df8409753729dBfbA5c58626F637,
              IAddressStore.Role.ROLES_MANAGER
            );

        assertTrue
          (_hasRole == hasRole_, "Address should not lose role after upgrade");
    }
}
```

Recommendations

Consider updating storage in a way that would not change the current variables' location, for example, `gnsStaking` and `linkErc677` could be added after the `accessControl` mapping.

[H-02] `ReferralsUtils.updateStartReferrerFeeP()` implements wrong check

Severity

Impact: High

Likelihood: Medium

Description

At L64 in the `updateStartReferrerFeeP` function comparing sign is wrongly set to `<=`, while it should be `>`, resulting in an inability to set `startReferrerFeeP` to value in the correct range:

```
File: ReferralsUtils.sol
63:     function updateStartReferrerFeeP(uint256 _value) internal {
64:         if
  (_value <= MAX_START_REFERRER_FEE_P) revert IGeneralErrors.AboveMax();
65:
66:         _getStorage().startReferrerFeeP = _value;
67:
68:         emit IReferralsUtils.UpdatedStartReferrerFeeP(_value);
69:     }
```

Recommendations

Consider updating the sign in the comparing to `>`.

[H-03] `ReferralsUtils.updateReferralsOpenFeeP()` implements wrong check

Severity

Impact: High

Likelihood: Medium

Description

At L75 in the `updateReferralsOpenFeeP` function comparing sign is wrongly set to <=, while it should be >, resulting in an inability to set openFeeP to value in the correct range:

```
File: ReferralsUtils.sol
    function updateReferralsOpenFeeP(uint256 _value) internal {
        if (_value <= MAX_OPEN_FEE_P) revert IGeneralErrors.AboveMax();

        _getStorage().openFeeP = _value;

        emit IReferralsUtils.UpdatedOpenFeeP(_value);
    }
```

Recommendations

Consider updating the sign in the comparing to >.

[H-04] Collateral not approved after updating of GToken address

Severity

Impact: High

Likelihood: Medium

Description

During adding new collateral tokens to the protocol, it's approved for corresponding `_gToken` contract address and GNS `staking` addresses:

```
File: TradingStorageUtils.sol
    function addCollateral(address _collateral, address _gToken) internal {
...
        // Setup Staking and GToken approvals
        IERC20 collateral = IERC20(_collateral);
        collateral.approve(_gToken, type(uint256).max);
        collateral.approve(staking, type(uint256).max);

        emit ITradingStorageUtils.CollateralAdded(_collateral, index, _gToken);
    }
```

Later, the `_gToken` address could be updated using the `TradingStorageUtils#updateGToken` function:

```
File: TradingStorageUtils.sol
    function updateGToken(address _collateral, address _gToken) internal {
        ITradingStorage.TradingStorage storage s = _getStorage();

        uint8 index = s.collateralIndex[_collateral];

        if (index == 0) {
            revert IGeneralErrors.DoesntExist();
        }

        if (_gToken == address(0)) {
            revert IGeneralErrors.ZeroAddress();
        }

        s.gTokens[index] = _gToken;

        emit ITradingStorageUtils.GTokenUpdated
  (_collateral, index, _gToken);
    }
```

However, this function misses the approval call to the collateral token, resulting in an inability of the new GToken contract to transfer collateral from the `GNSMultiCollatDiamond` address.

Recommendations

Consider adding an `approve` call to the collateral contract in the `TradingStorageUtils.updateGToken()` function.

[H-05] `closeTradeMarket` providing wrong value when triggering `getPrice`

Severity

Impact: Medium

Likelihood: High

Description

When `getPrice` is triggered, it will construct a Chainlink request, providing information based on the processed order type and data. Then, it calculates the `linkFeePerNode` using the provided `positionSizeCollateral` and eventually sends the request to the oracles, including the link request and `linkFeePerNode`. When calculating `linkFeePerNode`, it uses the `getLinkFee` function, expecting `_positionSizeCollateral` in its collateral precision.

```
function getLinkFee(
        uint8 _collateralIndex,
        uint16 _pairIndex,
        uint256 _positionSizeCollateral // collateral precision
    ) internal view returns (uint256) {
        (, int256 linkPriceUsd, , , ) = _getStorage
          ().linkUsdPriceFeed.latestRoundData();
        // NOTE: all [token / USD] feeds are 8 decimals
        return
            (getUsdNormalizedValue(
                _collateralIndex,
                _getMultiCollatDiamond().pairOracleFeeP
                  (_pairIndex) * _positionSizeCollateral
            ) * 1e8) /
            uint256(linkPriceUsd) /
            PRECISION /
            100;
    }
```

However, when `getPrice` is triggered from `closeTradeMarket`, it providing `collateralAmount` multiplied by the leverage and divided by precision delta and `PRECISION`.

```
function closeTradeMarket(uint32 _index) internal onlyTradingActivated
      (true) {
        address sender = _msgSender();

        ITradingStorage.Trade memory t = _getMultiCollatDiamond().getTrade
          (sender, _index);
        if (
            _getMultiCollatDiamond().getTradePendingOrderBlock(
                ITradingStorage.Id({user: t.user, index: t.index}),
                ITradingStorage.PendingOrderType.MARKET_CLOSE
            ) > 0
        ) revert ITradingUtils.AlreadyBeingMarketClosed();

        if (!t.isOpen) revert ITradingUtils.NoTrade();

        if
          (t.tradeType != ITradingStorage.TradeType.TRADE) revert ITradingUtils.WrongTradeT



        ITradingStorage.PendingOrder memory pendingOrder;
        pendingOrder.trade.user = t.user;
        pendingOrder.trade.index = t.index;
        pendingOrder.trade.pairIndex = t.pairIndex;
        pendingOrder.user = sender;
        pendingOrder.orderType = ITradingStorage.PendingOrderType.MARKET_CLOSE;

        pendingOrder = _getMultiCollatDiamond().storePendingOrder(pendingOrder);
        ITradingStorage.Id memory orderId = ITradingStorage.Id
          ({user: pendingOrder.user, index: pendingOrder.index});

        _getMultiCollatDiamond().getPrice(
            t.collateralIndex,
            t.pairIndex,
            orderId,
            pendingOrder.orderType,
>>>         (
  t.collateralAmount*t.leverage
) / 1e3 / collateralConfig.precisionDelta / PRECISION,
            ChainUtils.getBlockNumber()
        );

        emit ITradingUtils.MarketOrderInitiated
          (orderId, sender, t.pairIndex, false);
    }
```

This will result in the wrong value when calculating the fee that needs to be sent to the oracles.

Recommendations

Provide `(_trade.collateralAmount * _trade.leverage) / 1e3` instead, to maintain the required collateral precision.

[H-06] Wrong validation in `toggleCollateralActiveState()`

Severity

Impact: Medium

Likelihood: High

Description

The active state of the collateral can be toggled using `toggleCollateralActiveState()`, which will set `isActive` for the specified collateral index. This allows the governance to disable specified collateral when required and prevent opening of new trades using that collateral.

However, `toggleCollateralActiveState()` has an error in the validation check. It reverts when `collateral.precision > 0`, which would occur for all existing collateral as precision is set upon added.

This will prevent governance from disabling the collateral for trading when required in an emergency situation such as depegging of the stablecoin collateral.

```
function toggleCollateralActiveState(uint8 _collateralIndex) internal {
        ITradingStorage.TradingStorage storage s = _getStorage();

                 ITradingStorage.Collateral storage collateral = s.collaterals[_collateralI

        //@audit this will revert as precision is > 0 for existing collaterals
        if (collateral.precision > 0) {
            revert IGeneralErrors.DoesntExist();
        }

        bool toggled = !collateral.isActive;
        collateral.isActive = toggled;

        emit ITradingStorageUtils.CollateralUpdated(_collateralIndex, toggled);
    }
```

Recommendations

Change the check to `collateral.precision == 0`.

[H-07] Cancelled `MARKET_CLOSE` order will affect TP/SL trigger

Severity

Impact: High

Likelihood: Medium

Description

When the TP / SL are set in a trade, `tpLastUpdatedBlock` / `slLastUpdatedBlock` will be set to the current block number. This is to ensure that triggering of the TP and SL will be guaranteed as the callback handling will process the TP_CLOSE / SL_CLOSE order based on the TP/SL last updated block number.

However, a canceled `MARKET_CLOSE` will incorrectly reset both `tpLastUpdatedBlock` and `slLastUpdatedBlock` to the current block number, even when the TP / SL were set in the past.

That will occur when a `MARKET_CLOSE` order was placed but unsuccessfully executed, causing the order to be canceled. During the cancellation, the `collateralAmount` is deducted to pay for the oracle cost, requiring an update to `collateralAmount` via `updateTrade()`. The call to `updateTrade()` will reset both `tpLastUpdatedBlock` and `slLastUpdatedBlock` to the current block number.

This will be problematic if the TP/SL was already configured as any triggering of the TP/SL will be based on the incorrectly reset block number caused by the canceled `MARKET_CLOSE` order.

~~~
function closeTradeMarketCallback(
        ITradingCallbacks.AggregatorAnswer memory _a
    ) internal onlyTradingActivated(true) {
        ...
            if (cancelReason == ITradingCallbacks.CancelReason.NONE) {
            ...
            } else {

                t.collateralAmount -= uint120(govFee);
                _getMultiCollatDiamond().updateTrade(t, 0);
    }

    function updateTrade
      (ITradingStorage.Trade memory _t, uint16 _maxSlippageP) internal {
        ```
        i.createdBlock = uint32(ChainUtils.getBlockNumber());
        //@audit both TP last updated block will be reset to current block
        // number  when MARKET_CLOSE is cancelled
        i.tpLastUpdatedBlock = i.createdBlock;
        i.slLastUpdatedBlock = i.createdBlock;
    }
~~~

Recommendations

In the cancellation logic of `closeTradeMarketCallback()` replace `updateTrade()` with a new function that only updates the `collateralAmount` and not reset `tpLastUpdatedBlock` and `slLastUpdatedBlock`.

[H-08] Vulnerability to DOS attacks during market closures

Severity

Impact: Medium - The oracle system is susceptible to Denial of Service (DOS) attacks during market closures, potentially delaying the processing of legitimate transactions in other active markets such as crypto.

Likelihood: High - The vulnerability can be exploited whenever traditional markets are closed.

Description

When a trader attempts to close a position and the oracle returns `_a.price == 0`, it indicates that the market is closed, and the trade cannot be executed. This scenario triggers a fee (`govFee`) to cover the oracle service costs.

```
function closeTradeMarketCallback(
        ITradingCallbacks.AggregatorAnswer memory _a
    ) internal onlyTradingActivated(true) {

        ...

        ITradingCallbacks.CancelReason cancelReason = !t.isOpen
            ? ITradingCallbacks.CancelReason.NO_TRADE
            :
              (_a.price == 0 ? ITradingCallbacks.CancelReason.MARKET_CLOSED : ITradingCallb
            ...

            } else { // MARKET_CLOSED
                // Gov fee to pay for oracle cost
                _updateTraderPoints(t.collateralIndex, t.user, 0, t.pairIndex);
                uint256 govFee = _handleGovFees(
                    t.collateralIndex,
                    t.user,
                    t.pairIndex,
                    v.positionSizeCollateral,
                    _getMultiCollatDiamond().isTradeCollateralInStorage(
                        ITradingStorage.Id({user: t.user, index: t.index})
                    )
                );
                t.collateralAmount -= uint120(govFee);
                _getMultiCollatDiamond().updateTrade(t, 0);
                ...
            }
        }

        ...
    }
```

Given that markets for assets like forex and commodities may close for extended periods, ranging from a few hours to an entire day, and the system permits extremely high leverage (up to 250x for commodities and 1000x for forex), a loophole exists for potential DOS attacks at minimal cost.

Let see an attack scenario:

- Assume `openFeeP` is 3%.
- A trader opens a forex trading position with a $1.5 collateral and 1000x leverage, equaling a $1500 position size, to satisfy the minimum requirement.
- When the market of the trading pair closes, the trader initiates a `closeTradeMarket` request.
- Because the market is closed, all oracle nodes return 0 and `closeTradeMarketCallback` is triggered. The trader is charged 3% of the collateral amount or $0.045($1.5*3%=$0.045).
- By repeatedly issuing `closeTradeMarket` requests, a trader can bog down the oracle system with minimal cost per attempt (3% of the reduced collateral amount).
- To increase the scale of the attack, the malicious trader can open a position for every trading pair and spam all of them at the same time when the market is closed.
- The attack only ends when the market is reopened.

Recommendations

Consider the following option:

- Restricting traders from requesting `closeTradeMarket` more than once during market closures.
- Introducing a fixed minimum fee for `closeTradeMarket` requests when markets are closed, deterring spamming behavior by increasing the cost of attacks.
- Increasing min position size, which effectively increases attack cost.

8.3. Medium Findings

[M-01] `getCollaterals()` does not return the last collateral

Severity

Impact: Low

Likelihood: High

Description

The `getCollaterals()` function is used to retrieve all the existing collaterals in the protocol. And as evident in `addCollateral()`, the `collateralIndex` for existing collateral starts from 1, while 0 is not used for any collateral.

However, `getCollateral()` accesses the `Collateral[]` incorrectly and retrieves the collaterals from index 0. This will cause it to return the empty collateral at index 0 and also fail to return the last collateral.

```
function getCollaterals() internal view returns
      (ITradingStorage.Collateral[] memory) {
        ITradingStorage.TradingStorage storage s = _getStorage();



         //@audit it should be  for
         //(uint8 i = 1; i < s.lastCollateralIndex +1; ++i) { instead
        for (uint8 i; i < s.lastCollateralIndex; ++i) {
            collaterals[i] = s.collaterals[i];
        }

        return collaterals;
    }
```

Recommendations

Change from

```
for (uint8 i; i < s.lastCollateralIndex; ++i) {
```

to

```
for (uint8 i = 1; i < s.lastCollateralIndex + 1; ++i) {
```

[M-02] Incorrect storage slot assignments

Severity

Impact: High

Likelihood: Low

Description

We have the extracted storage layout of the GNSMultiCollatDiamond contract.

```
|
  Name                      | Type                         | Slot | Offset | Bytes |

|
  _initialized              | uint8                        | 0    | 0      | 1     |
|
  _initializing             | bool                         | 0    | 1      | 1     |
|
  addressStore              | AddressStore                 | 1    | 0      | 1632  |
|
  pairsStorage              | PairsStorage                 | 52   | 0      | 1600  |
|
  referralsStorage          | ReferralsStorage             | 102  | 0      | 1600  |
|
  feeTiersStorage           | FeeTiersStorage              | 152  | 0      | 1600  |
|
  priceImpactStorage        | PriceImpactStorage           | 202  | 0      | 1600  |
|
  diamondStorage            | DiamondStorage               | 252  | 0      | 1600  |
|
  tradingStorage            | TradingStorage               | 302  | 0      | 1600  |
|
  triggerRewardsStorage     | TriggerRewardsStorage        | 352  | 0      | 1568  |
|
  tradingInteractionsStorage | TradingInteractionsStorage  | 401  | 0      | 1600  |
|
  tradingCallbacksStorage   | TradingCallbacksStorage      | 451  | 0      | 1600  |
|
  borrowingFeesStorage      | BorrowingFeesStorage         | 501  | 0      | 1600  |
|
  priceAggregatorStorage    | PriceAggregatorStorage       | 551  | 0      | 1696  |
```

Here is the storage slot assigned to use to retrieve the storage variables

```
library StorageUtils {
        uint256 internal constant GLOBAL_ADDRESSES_SLOT = 1;
        uint256 internal constant GLOBAL_PAIRS_STORAGE_SLOT = 51;
        uint256 internal constant GLOBAL_REFERRALS_SLOT = 101;
        uint256 internal constant GLOBAL_FEE_TIERS_SLOT = 151;
        uint256 internal constant GLOBAL_PRICE_IMPACT_SLOT = 201;
        uint256 internal constant GLOBAL_DIAMOND_SLOT = 251;
        uint256 internal constant GLOBAL_TRADING_STORAGE_SLOT = 301;
        uint256 internal constant GLOBAL_TRIGGER_REWARDS_SLOT = 351;
        uint256 internal constant GLOBAL_TRADING_SLOT = 401;
        uint256 internal constant GLOBAL_TRADING_CALLBACKS_SLOT = 451;
        uint256 internal constant GLOBAL_BORROWING_FEES_SLOT = 501;
        uint256 internal constant GLOBAL_PRICE_AGGREGATOR_SLOT = 551;
    }
```

link

We can see that the storage slots of `PairsStorage`, `ReferralsStorage`, `FeeTiersStorage`, `PriceImpactStorage`, `DiamondStorage`, `TradingStorage` and `TriggerRewardsStorage` are wrongly assigned.

It can lead to storage collision and break the `GNSMultiCollatDiamond` contract.

Recommendations

Update the constants in `StorageUtils` to reflect the correct storage slot.

[M-03] Inaccurate calculation of `v.positionSizeCollateralAfterReferralFees`

Severity

Impact: Medium

Likelihood: Medium

Description

When calculating v.positionSizeCollateralAfterReferralFees and reward1 (the actual referral reward), the following calculation will be used:

```
// ...
    if (_getMultiCollatDiamond().getTraderActiveReferrer
      (_trade.user) != address(0)) {
        // Use this variable to store position size for dev/gov fees after
        // referral fees
        // and before volumeReferredUsd increases
>>>     v.positionSizeCollateralAfterReferralFees =
            (v.positionSizeCollateral *
                (100 *
                    PRECISION -
                    _getMultiCollatDiamond().calculateFeeAmount(
                        _trade.user,
                        _getMultiCollatDiamond().getReferralsPercentOfOpenFeeP
                          (_trade.user)
                    ))) /
            100 /
            PRECISION;

>>>     v.reward1 = _distributeReferralReward(
            _trade.collateralIndex,
            _trade.user,
            _getMultiCollatDiamond().calculateFeeAmount
            //(_trade.user, v.positionSizeCollateral), // apply fee tiers here to v.positio
            _getMultiCollatDiamond().pairOpenFeeP(_trade.pairIndex),
            v.gnsPriceCollateral
        );

        _sendToVault(_trade.collateralIndex, v.reward1, _trade.user);

        _trade.collateralAmount -= uint120(v.reward1);

        emit ITradingCallbacksUtils.ReferralFeeCharged
          (_trade.user, _trade.collateralIndex, v.reward1);
    }
    // ...
```

Where the percentage of referral open fee is from `getReferralsPercentOfOpenFeeP` and using this formula :

```
function getPercentOfOpenFeeP_calc
      (uint256 _volumeReferredUsd) internal view returns (uint256 resultP) {
        IReferralsUtils.ReferralsStorage storage s = _getStorage();
        uint startReferrerFeeP = s.startReferrerFeeP;
        uint openFeeP = s.openFeeP;
        resultP =
            (openFeeP *
                (startReferrerFeeP *
                    PRECISION +
                    (_volumeReferredUsd * PRECISION *
                      (100 - startReferrerFeeP)) /
                    1e18 /
                    s.targetVolumeUsd)) /
            100;


                 resultP = resultP > openFeeP * PRECISION ? openFeeP * PRECISION : resultP;
    }
```

However, when calculating `reward1`, it will use `getReferrerFeeP`, which has a different formula to calculate the referrer fee :

```
function getReferrerFeeP(
      uint256_pairOpenFeeP,
      uint256_volumeReferredUsd
    ) internal view returns (uint256
        IReferralsUtils.ReferralsStorage storage s = _getStorage();

        uint256 maxReferrerFeeP = (_pairOpenFeeP * 2 * s.openFeeP) / 100;

        uint256 minFeeP = (maxReferrerFeeP * s.startReferrerFeeP) / 100;

        uint256 feeP = minFeeP + (
          (maxReferrerFeeP - minFeeP) * _volumeReferredUsd) / 1e18 / s.targetVolumeUsd;

        return feeP > maxReferrerFeeP ? maxReferrerFeeP : feeP;
    }
```

This will result in `v.positionSizeCollateralAfterReferralFees` not being based on the actual referral fee. Consequently, when `v.positionSizeCollateralAfterReferralFees` is passed to `_handleGovFees` for calculating `govFee`, it will process the wrong value.

Recommendations

Use the actual `reward1` instead when calculating `v.positionSizeCollateralAfterReferralFees`:

```
v.positionSizeCollateralAfterReferralFees
   = v.positionSizeCollateral - v.reward1
```

[M-04] Inaccurate calculation of referral fees for referrers and allies

Severity

Impact: Low

Likelihood: High

Description

When calculating the referral reward, the fee tiers are applied immediately to the `positionSizeCollateral`, so the position size is already adjusted by the `feeMultiplierCache` of the trader.

```
function _registerTrade(
        ITradingStorage.Trade memory _trade,
        ITradingStorage.PendingOrder memory _pendingOrder
    ) internal returns (ITradingStorage.Trade memory, uint256) {
        ...
            v.reward1 = _distributeReferralReward(
                _trade.collateralIndex,
                _trade.user,
                _getMultiCollatDiamond().calculateFeeAmount
                //(_trade.user, v.positionSizeCollateral), // apply fee tiers here to v.pos
                _getMultiCollatDiamond().pairOpenFeeP(_trade.pairIndex),
                v.gnsPriceCollateral
            );
        ...
    }
```

Using the adjusted `positionSizeCollateral` is correct in calculating the referrer fee. However, it is an incorrect calculation for the `volumeReferredUsd` of the referrers because the volume is reduced by `(1e3 - feeMultiplierCache) / FEE_MULTIPLIER_SCALE` percentage.

```
function distributeReferralReward(
        address _trader,
        uint256 _volumeUsd, // 1e18
        uint256 _pairOpenFeeP,
        uint256 _gnsPriceUsd // PRECISION (1e10)
    ) internal returns (uint256) {
        ...
        if (a.active) {
            ...
            a.volumeReferredUsd += _volumeUsd;
            ...
        }

        r.volumeReferredUsd += _volumeUsd;
        ...
    }
```

So it leads to the wrong referrer fee of the referrer since the `_volumeReferredUsd` is used to calculate the `feeP`.

```
function getReferrerFeeP(
      uint256_pairOpenFeeP,
      uint256_volumeReferredUsd
    ) internal view returns (uint256
        IReferralsUtils.ReferralsStorage storage s = _getStorage();

        uint256 maxReferrerFeeP = (_pairOpenFeeP * 2 * s.openFeeP) / 100;
        uint256 minFeeP = (maxReferrerFeeP * s.startReferrerFeeP) / 100;

        uint256 feeP = minFeeP + (
          (maxReferrerFeeP - minFeeP) * _volumeReferredUsd) / 1e18 / s.targetVolumeUsd;
        return feeP > maxReferrerFeeP ? maxReferrerFeeP : feeP;
    }
```

Recommendations

Use the `positionSizeCollateral` instead of `calculateFeeAmount(_trade.user, v.positionSizeCollateral)` when calculating `volumeReferredUsd` for referrers and allies.

[M-05] Reorg incident could affect fulfilling orders

Severity

Impact: High

Likelihood: Low

Description

Oracle nodes fulfill price data using the `requestId` parameter for identifying each request. This parameter is generated based on the diamond address and its current request count(nonce) at L190:

```
File: ChainlinkClientUtils.sol
183:     function _rawRequest(
184:         address oracleAddress,
185:         uint256 nonce,
186:         uint256 payment,
187:         bytes memory encodedRequest
188:     ) private returns (bytes32 requestId) {
189:         IPriceAggregator.PriceAggregatorStorage storage s = _getStorage();
190:         requestId = keccak256(abi.encodePacked(this, nonce));
191:         s.pendingRequests[requestId] = oracleAddress;
192:         emit ChainlinkRequested(requestId);
193:         if (!s.link.transferAndCall
  (oracleAddress, payment, encodedRequest))
194:             revert IPriceAggregatorUtils.TransferAndCallToOracleFailed();
195:     }
```

In case of a reorg incident, this could create a situation when orders would be fulfilled with price data that was requested for other orders. This would result in executing orders with completely wrong prices, generating losses/gains for users that shouldn't take place.

Consider the next scenario (for simplicity let's assume only one oracle exists in the system):

1. Alice opens a trade, `requestId` for her order is based on the current nonce "1" at the diamond storage - "abc123".
2. Bob opens a trade, `requestId` for his order is based on the current nonce "2" at the diamond storage - "def456".

3. Oracle sends 2 fulfilling txs with the correct prices for each order based on their `requestId`'s as expected.
4. Reorg incident happens, affecting the order of execution tx 1 and 2. Now Alice's order has `requestId` based on nonce "2" - "def456", however since fulfilling tx is already sent by oracle for this `requestId` with prices for Bob's order - Alice's trade is executed with fully wrong data, same with Bob.

Recommendations

Consider mixing the `orderId` values into the `requestId` generation. This would guarantee that each request has its unique identification.

[M-06] Profitable trade positions risk unwarranted liquidation and loss of profit

Severity

Impact: High

Likelihood: Low

Description

When the liquidator calls `triggerOrder` to liquidate a trade, the `_fromBlock` parameter is set to `tradeInfo.createdBlock`. It is the block where the trade is executed and stored in the system.

```
function _getPriceTriggerOrder(
        ITradingStorage.Trade memory _trade,
        ITradingStorage.Id memory _orderId,
        ITradingStorage.PendingOrderType _orderType,
        uint256 _positionSizeCollateral // collateral precision
    ) internal {
        ITradingStorage.TradeInfo memory tradeInfo = _getMultiCollatDiamond
          ().getTradeInfo(_trade.user, _trade.index);

        _getMultiCollatDiamond().getPrice(
            _trade.collateralIndex,
            _trade.pairIndex,
            _orderId,
            _orderType,
            _positionSizeCollateral,
            _orderType == ITradingStorage.PendingOrderType.SL_CLOSE
                ? tradeInfo.slLastUpdatedBlock
                : _orderType == ITradingStorage.PendingOrderType.TP_CLOSE
                    ? tradeInfo.tpLastUpdatedBlock
    @>              : tradeInfo.createdBlock
        );
    }
```

When the oracle returns the prices, it will return prices with 3 values:

- _a.open: current price.
- _a.high: higest price from `tradeInfo.createdBlock` to the present.
- _a.low: lowest price from `tradeInfo.createdBlock` to the present.

If the liquidation price falls within the `_a.low` and `_a.high` range, it means `exactExecution` and the `executionPrice` is equal to `liqPrice`.

```
function executeTriggerCloseOrderCallback(
        ITradingCallbacks.AggregatorAnswer memory _a
    ) internal onlyTradingActivated(true) {
        ...
            if (o.orderType == ITradingStorage.PendingOrderType.LIQ_CLOSE) {
                v.liqPrice = _getMultiCollatDiamond().getTradeLiquidationPrice(
                    IBorrowingFees.LiqPriceInput(
                        t.collateralIndex,
                        t.user,
                        t.pairIndex,
                        t.index,
                        t.openPrice,
                        t.long,
                        uint256(t.collateralAmount),
                        t.leverage
                    )
                );
            }


                         v.executionPrice = o.orderType == ITradingStorage.PendingOrderType
                ? t.tp
                :
                  (o.orderType == ITradingStorage.PendingOrderType.SL_CLOSE ? t.sl : v.liqP


                         v.exactExecution = v.executionPrice > 0 && _a.low <= v.executionPr

            if (v.exactExecution) {

                                 v.reward1 = o.orderType == ITradingStorage.PendingOrderTyp
                    ? (uint256(t.collateralAmount) * 5) / 100
                    : (v.positionSizeCollateral * _getMultiCollatDiamond
                      ().pairTriggerOrderFeeP(t.pairIndex)) /
                        100 /
                        PRECISION;
            } else {
                v.executionPrice = _a.open;
        ...
    }
```

The issue arises as the liquidation price incrementally elevates over time due to the accrual of borrowing fees, reducing the `liqPriceDistance` and thereby increasing the `liqPrice`.

```
function _getTradeLiquidationPrice(
        uint256 _openPrice,
        bool _long,
        uint256 _collateral,
        uint256 _leverage,
        uint256 _borrowingFeeCollateral,
        uint128 _collateralPrecisionDelta
    ) internal pure returns (uint256) {
        ...
        int256 borrowingFeeInt = int256
        //(_borrowingFeeCollateral * precisionDeltaUint * 1e3); // 1e18 * 1e3

        // PRECISION
        int256 liqPriceDistance = (openPriceInt *
        //(collateralLiqNegativePnlInt - borrowingFeeInt)) / // 1e10 * 1e18 * 1e3
            int256(_collateral) /
            int256(_leverage) /
            int256(precisionDeltaUint); // 1e10

        int256 liqPrice = _long ? openPriceInt - liqPriceDistance : openPriceInt
        // + liqPriceDistance; // 1e10

        return liqPrice > 0 ? uint256(liqPrice) : 0; // 1e10
    }
```

This mechanism does not sufficiently consider the profitability of a position. For example, a trader's highly profitable position can be subject to liquidation if the borrowing fees adjust the liquidation price to within the low and high price range, despite favorable market trends. If a trade position is opened long enough, the liquidation price can increase and be equal to the `_a.low`. At that moment the position is liquidable regardless of its current profit level.

Let's consider a scenario when the price of the pair increases and the trade has no stop limit.

- A trader enters a long position at $1505 with 100x leverage. The liquidation price is calculated at $1494
- The market appreciates and the trader keeps the position open.
- After 3500 blocks (~ 7000 seconds ~ 1.95 hours), the price of the pairs increases to `$1550` and the position is very profitable. However, due to the borrowing fee, the liquidation price is increased to `$1506`.
- So anyone can trigger a liquidation for this position because we have `_a.low ($1505) < liqPrice ($1506) < _a.high ($1550)`. And the execution price in liquidation is $1506.
- Because the execution price in liquidation (`$1506`) is much lower than the current price of the pairs (`$1550`), instead of having 3x profitability, the trader loses all of the collateral amount due to borrowing fee and receives no profit.

Recommendations

Adjust the liquidation price calculation to incorporate the current profitability of a position alongside the open price and borrowing fees.

[M-07] Liquidation failure for traders on USDC blacklist

Severity

Impact: High

Likelihood: Low

Description

During the process of liquidating an account, the associated trade is unregistered, and any remaining collateral is returned to the trader. In case liquidation happens, the `tradeValueCollateral` is 0.

```
function _unregisterTrade(
        ITradingStorage.Trade memory _trade,
        bool _marketOrder,
        int256 _percentProfit,
        uint256 _closingFeeCollateral,
        uint256 _triggerFeeCollateral
    ) internal returns (uint256 tradeValueCollateral) {
        ...
            if (tradeValueCollateral > collateralLeftInStorage) {
                vault.sendAssets
                  (tradeValueCollateral - collateralLeftInStorage, _trade.user);
                _transferCollateralToAddress(
                  _trade.collateralIndex,
                  _trade.user,
                  collateralLeftInStorage
                );
            } else {
                _sendToVault(
                  _trade.collateralIndex,
                  collateralLeftInStorage-tradeValueCollateral,
                  _trade.user
                );
                _transferCollateralToAddress
                  (_trade.collateralIndex, _trade.user, tradeValueCollateral);
            }

            // 4.2 If collateral in vault, just send collateral to trader from
            // vault
        } else {
            vault.sendAssets(tradeValueCollateral, _trade.user);
        }
    }
```

However, this process is failed if the trader has been blacklisted by the USDC contract. Specifically, the liquidation attempt fails when trying to transfer a `tradeValueCollateral` of 0, due to a revert in the `_transferCollateralToAddress` function. This can lead to financial losses for the vault, as positions may continue to depreciate without the possibility of liquidation.

Recommendations

Instead of pushing the collateral amount to traders, let them claim it (Pull over push pattern).

[M-08] Trader can evade loss when Stop order is not fulfilled in time

Severity

Impact: High

Likelihood: Low

Description

For Stop orders, inexact executions (`exactExecution == false`) could occur in certain scenarios, which will fulfill the orders based on market price instead of open/stop-loss price.

One of these scenarios is a price gap during market re-opening, such that the market open price has gone far over the open/stop-loss price while `_a.low == 0 && _a.high == 0`. In this scenario, the Stop orders will be fulfilled based on the market price instead of open/stop-loss price as there have been no prices in the past 1 hour due to market closure. Note that oracles only return price answers up to 1 hour for order fulfillment.

However, if the Stop orders are not fulfilled within 1 hour of market re-opening due to sequencer downtime, it could actually incorrectly fulfill orders at the open/stop-loss price (`exactExecution == true`).

Suppose the scenario for Stop-Loss,

1. Trader A opens trade at `1000` with Stop-Loss at `900`.
2. Market is closed shortly after.
3. Now sequencer is down, preventing trigger bots from triggering orders.
4. Market re-opens and the price gapped down to `500`.
5. However, as the sequencer is down, the inexact execution to fulfill Stop-Loss at a market price of `500` will fail.
6. After more than 1 hour, the sequencer recovers.
7. The price has now increased to `900`.

8. The order is now triggered again, and because there are prices in the past 1 hour, it will fulfill the Stop-Loss with exact execution, and close it at the price of `900`.
9. Now Trader A has managed to evade the loss as the trade is closed at `900` instead of `500`, causing vault users to incur a loss of profit.

For Stop-Open orders, the issue could also cause incorrect exact execution to occur as well, such that it will open the Order at an open price during market re-opening, instead of market price. This means that the trader can potentially open at a better price and avoid the price gap, to evade loss.

Recommendations

When the sequencer is down within 1 hour of market re-opening, the oracles should still use the price answers at the point of market re-opening so that it will correctly fulfill the Stop orders based on market price.

This mitigation applies for Stop-Loss and Stop-Open orders that were set before the sequencer was last offline.

8.4. Low Findings

[L-01] Unnecessary `_transferCollateralToAddress()` used

Within `TradingCallBacksUtils`, the functions `_distributeStakingReward()`, `_sendToVault()` and `_unregisterTrade()` will call `_transferCollateralToAddress()` to transfer the collateral to `address(this)`. That is unnecessary as the collaterals are already within the contract itself.

This can be resolved by removing the `_transferCollateralToAddress()` calls.

```
function _distributeStakingReward(
      uint8_collateralIndex,
      address_trader,
      uint256_amountCollateral
    ) internal {
        //@audit redundant transfer as it is transferring to itself
        _transferCollateralToAddress(_collateralIndex, address
          (this), _amountCollateral);
        IGNSStaking(AddressStoreUtils.getAddresses
          ().gnsStaking).distributeReward(
            _getMultiCollatDiamond().getCollateral(_collateralIndex).collateral,
            _amountCollateral
        );
        emit ITradingCallbacksUtils.GnsStakingFeeCharged
          (_trader, _collateralIndex, _amountCollateral);
    }

    function _sendToVault(
      uint8_collateralIndex,
      uint256_amountCollateral,
      address_trader
    ) internal {
        //@audit redundant transfer as it is transferring to itself
        _transferCollateralToAddress(_collateralIndex, address
          (this), _amountCollateral);
        _getGToken(_collateralIndex).receiveAssets(_amountCollateral, _trader);
    }


    function _unregisterTrade(
        ITradingStorage.Trade memory _trade,
        bool _marketOrder,
        int256 _percentProfit,
        uint256 _closingFeeCollateral,
        uint256 _triggerFeeCollateral
    ) internal returns (uint256 tradeValueCollateral) {
             ....
            // 5. gToken vault reward
            uint256 vaultClosingFeeP = uint256(_getStorage().vaultClosingFeeP);
            v.reward2 = (_closingFeeCollateral * vaultClosingFeeP) / 100;

            //@audit redundant transfer as it is transferring to itself
            _transferCollateralToAddress(_trade.collateralIndex, address
              (this), v.reward2);
            vault.distributeReward(v.reward2);
            ...
    }
```

[L-02] Chainlink feeds could have 18 decimals

Multiple places in the code are built with the assumption that all Chainlink USD price feeds have 8 decimals, for example:

```
File: PriceAggregatorUtils.sol
    function getLinkFee(
        uint8 _collateralIndex,
        uint16 _pairIndex,
        uint256 _positionSizeCollateral // collateral precision
    ) internal view returns (uint256) {
        (, int256 linkPriceUsd, , , ) = _getStorage().linkUsdPriceFeed.latestRoundData();

        // NOTE: all [token / USD] feeds are 8 decimals
        return
            (getUsdNormalizedValue(
                _collateralIndex,
                _getMultiCollatDiamond().pairOracleFeeP(_pairIndex) * _positionSizeCollateral
            ) * 1e8) /
            uint256(linkPriceUsd) /
            PRECISION /
            100;
    }
```

However, there are USD price feeds that have 18 decimals, for example, PEPE/USD on Arbitrum: Arbiscan link

While this token is not expected as a protocol collateral at this moment, some future tokens with 18 decimal feeds could be, and integration of such tokens would require a protocol upgrade.

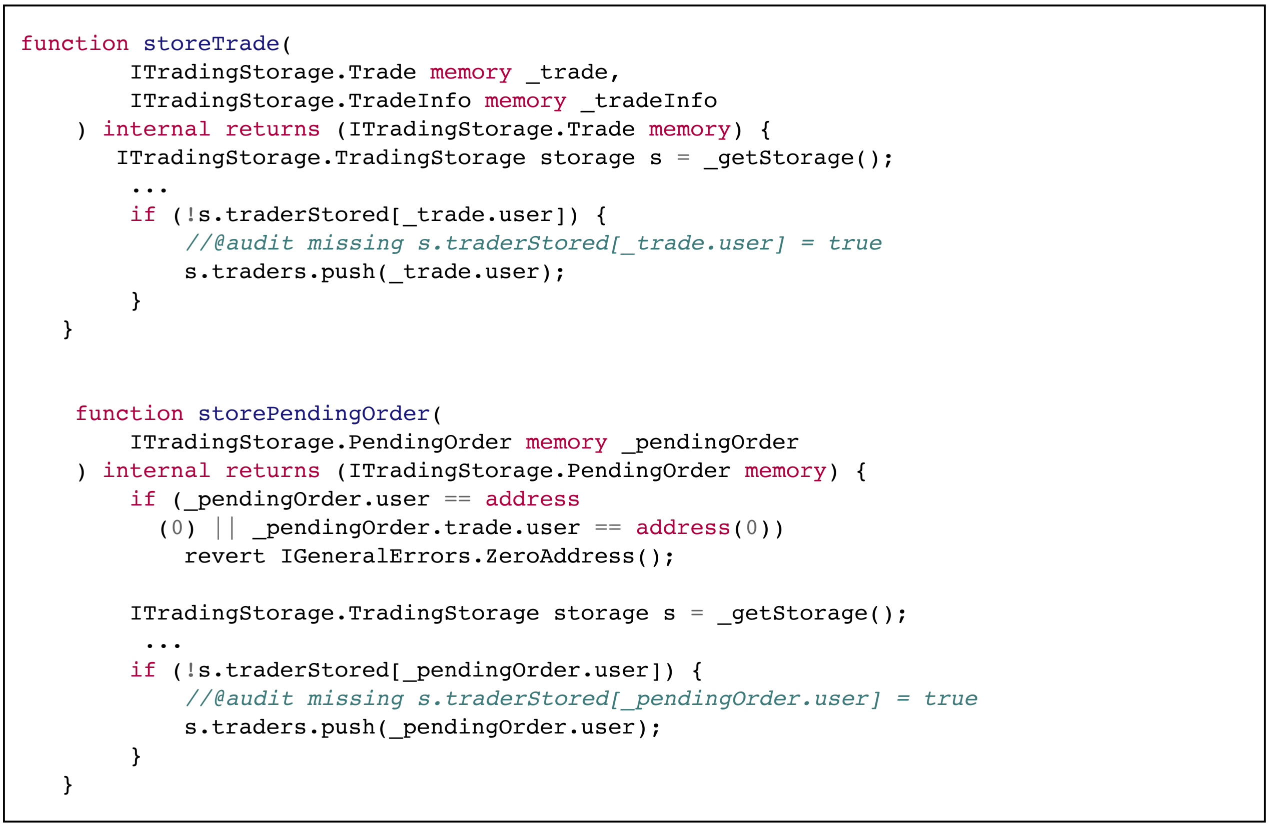
Consider saving price feed decimals number per each collateral during its adding and use this value in price calculations.

[L-03] `s.traders` will contain duplicates and affect `getTraders()`

Both `storeTrade()` and `storePendingOrder()` will populate the `s.traders[]` array to keep track of the list of traders that have created a trade or pending order. It checks for existing traders using the `s.traderStored[]` mapping to prevent adding duplicate addresses.

However, it fails to set `s.traderStored[_trade.user] = true` after pushing the trader address into the `s.traders[]` array. This will cause `s.traders[]` to increase on every trade/order and contain duplicate trader addresses. It will lead to an incorrect result for `getTraders()`.

To fix this, add `s.traderStored[_trade.user] = true` to `storeTrade()` and `storePendingOrder()`.



```
function storeTrade(
        ITradingStorage.Trade memory _trade,
        ITradingStorage.TradeInfo memory _tradeInfo
    ) internal returns (ITradingStorage.Trade memory) {
       ITradingStorage.TradingStorage storage s = _getStorage();
        ...
        if (!s.traderStored[_trade.user]) {
            //@audit missing s.traderStored[_trade.user] = true
            s.traders.push(_trade.user);
        }
   }


    function storePendingOrder(
        ITradingStorage.PendingOrder memory _pendingOrder
    ) internal returns (ITradingStorage.PendingOrder memory) {
        if (_pendingOrder.user == address
          (0) || _pendingOrder.trade.user == address(0))
            revert IGeneralErrors.ZeroAddress();

        ITradingStorage.TradingStorage storage s = _getStorage();
         ...
        if (!s.traderStored[_pendingOrder.user]) {
            //@audit missing s.traderStored[_pendingOrder.user] = true
            s.traders.push(_pendingOrder.user);
        }
   }
```

[L-04] `_correctTp` and `_correctSl` should be called when `updateTrade` is called

When a trade is created and stored using storeTrade, `_correctTp` and `_correctSl` are called to adjust TP and SL. However, if traders update the trade's TP and SL by calling `updateOpenOrder` and eventually trigger `updateTrade`, the TP and SL are not adjusted using `_correctTp` and `_correctSl`. Consider also performing TP and SL correction when `updateTrade` is triggered.

[L-05] Extra dust amount of GNS minted

The `TriggerRewardsUtils#distributeTriggerReward` function counts an equal amount of rewards per oracle at L46:

```
File: TriggerRewardsUtils.sol
42:     function distributeTriggerReward(uint256 _rewardGns) internal {
43:         ITriggerRewards.TriggerRewardsStorage storage s = _getStorage();
44:
45:         address[] memory oracles = _getMultiCollatDiamond().getOracles();
46:         uint256 rewardPerOracleGns = _rewardGns / oracles.length;
47:
48:         for (uint256 i; i < oracles.length; ++i) {
49:             s.pendingRewardsGns[oracles[i]] += rewardPerOracleGns;
50:         }
51:
52:         IERC20(AddressStoreUtils.getAddresses().gns).mint(address
  (this), _rewardGns);
53:
54:         emit ITriggerRewardsUtils.TriggerRewarded
  (rewardPerOracleGns, oracles.length);
55:     }
```

However, due to rounding down the sum of saved rewards `rewardPerOracleGns * oracles.length` often would be less than the actual minted amount at L52, meaning that dust extra amounts of GNS tokens would be minted.

Consider calling the `mint` function with the correct sum of rewards - `rewardPerOracleGns * oracles.length`.

[L-06] `getAllTrades` and `getAllTradeInfos` should exit the loop earlier

`getAllTrades` and `getAllTradeInfos` will loop over all traders and `Trade`/`TradeInfo` to get the array of data, based on the provided `_offset` and `_limit`. However, the loop will continue even when `currentTradeIndex` reaches `_limit`.

```
function getAllTrades(
      uint256_offset,
      uint256_limit
    ) internal view returns (ITradingStorage.Trade[] memory
        // Fetch all traders with open trades
        //(no pagination, return size is not an issue here)
        address[] memory traders = getTraders(0, 0);

        uint256 currentTradeIndex; // current global trade index
        uint256 currentArrayIndex; // current index in returned trades array

        ITradingStorage.Trade[] memory trades = new ITradingStorage.Trade[]
          (_limit - _offset + 1);

        // Fetch all trades for each trader
        for (uint256 i; i < traders.length; ++i) {
            ITradingStorage.Trade[] memory traderTrades = getTrades(traders[i]);

            // Add trader trades to final trades array only if within _offset
            // and _limit
            for (uint256 j; j < traderTrades.length; ++j) {

                if
                  (currentTradeIndex >= _offset && currentTradeIndex <= _limit) {
                    trades[currentArrayIndex++] = traderTrades[j];
                }
                // @audit - will continue even when _limit is reached
                currentTradeIndex++;
            }
        }

        return trades;
    }
```

Consider breaking the loop when the `currentTradeIndex` reaches `_limit`.