Gravity monorepo Security Review

Auditors

0xRajeev, Lead Security Researcher

R0bert, Lead Security Researcher

Anurag Jain, Security Researcher

Windhustler, Security Researcher

Report prepared by: Lucas Goiriz

December 14, 2024

Contents

1 About Spearbit

Spearbit is a decentralized network of expert security engineers offering reviews and other security related services to Web3 projects with the goal of creating a stronger ecosystem. Our network has experience on every part of the blockchain technology stack, including but not limited to protocol design, smart contracts and the Solidity compiler. Spearbit brings in untapped security talent by enabling expert freelance auditors seeking flexibility to work on interesting projects together.

Learn more about us at spearbit.com

2 Introduction

Gravity is an innovative hybrid infrastructure aiming to rebuild global finance on blockchain -- compliant, user-friendly, and scalable, blending the best of CeFi and DeFi.

Disclaimer: This security review does not guarantee against a hack. It is a snapshot in time of Gravity monorepo according to the specific commit. Any modifications to the code will require a new security review.

3 Risk classification

Severity level	**Impact: High**	**Impact: Medium**	**Impact: Low**
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

3.1 Impact

- High - leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
- Medium - global losses <10% or losses to only a subset of users, but still unacceptable.
- Low - losses will be annoying but bearable--applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.

3.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized
- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align, or little-to-no incentive

3.3 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

4 Executive Summary

Over the course of 28 days in total, Gravity engaged with Spearbit to review the gravity-monorepo protocol. In this period of time a total of **73** issues were found.

Summary

Project Name	Gravity
Repositories	proxy-bridging-contracts, zksync-withdrawal-finalizer & exchange-contract
Commits	c7052d93, 27bada48 & 4867f7bb
Type of Project	DeFi, AMM
Audit Timeline	Aug 16th to Sep 13th
Fix period	Nov 18 - Dec 2

Issues Found

Severity	**Count**	**Fixed**	**Acknowledged**
Critical Risk	1	1	0
High Risk	7	7	0
Medium Risk	19	13	6
Low Risk	30	20	10
Gas Optimizations	1	1	0
Informational	15	7	8
Total	**73**	**49**	**24**

5 Findings

5.1 Critical Risk

5.1.1 `_settleOptionsOrFutures()` will revert leading to blocked subAccount funds if there is more than one position opened

Severity: Critical Risk

Context: FundingAndSettlement.sol#L57, FundingAndSettlement.sol#L68

Description: The `FundingAndSettlement` contract implements the function `_settleOptionsOrFutures()`:

```
function _settleOptionsOrFutures(SubAccount storage sub, PositionsMap storage positions) internal {
  uint64 qdec = _getBalanceDecimal(sub.quoteCurrency);
  BI memory newSubBalance = BI(int64(sub.spotBalances[sub.quoteCurrency]), qdec);
  bytes32[] storage posKeys = positions.keys;
  mapping(bytes32 => Position) storage posValues = positions.values;
  uint posLen = posKeys.length;
  for (uint i; i < posLen; ++i) {
    bytes32 assetID = posKeys[i];
    (uint64 settlePrice, bool found) = _getAssetSettlementPrice(assetID);
    if (!found) {
      continue;
    }

    int64 positionBalance = posValues[assetID].balance;
    remove(positions, assetID);

    if (settlePrice == 0) {
      continue;
    }

    BI memory posBalance = BI(positionBalance, _getBalanceDecimal(assetGetUnderlying(assetID)));
    newSubBalance = newSubBalance.add(posBalance.mul(BI(int256(uint256(settlePrice)), PRICE_DECIMALS)));
  }
  sub.spotBalances[sub.quoteCurrency] = newSubBalance.toInt64(qdec);
}
```

This function updates the subAccount balance by settling the positions in futures or options (F&Os). It calculates the settlement value for each position based on the settlement price, removes the settled position and updates the subaccount's spot balance accordingly.

On the other hand, the `remove()` function deletes a specified position from the PositionsMap and maintains the integrity of the map's indices by swapping the last element with the one being removed and then popping the last element from the keys array:

```
function remove(PositionsMap storage map, bytes32 assetID) {
  if (map.values[assetID].id == 0) return;

  delete map.values[assetID];

  uint index = map.index[assetID];
  bytes32 lastKey = map.keys[map.keys.length - 1];

  map.index[lastKey] = index;
  delete map.index[assetID];

  map.keys[index] = lastKey;
  map.keys.pop();
}
```

However, the `_settleOptionsOrFutures()` function uses a storage pointer to the `position.keys` array: `bytes32[] storage posKeys = positions.keys`. When you use a storage reference like this one, the `posKeys` variable is directly referencing the `positions.keys` array in storage. This means that if `positions.keys` is updated (e.g., elements are removed) during the loop, the `posKeys` array will reflect these changes immediately. And this is exactly what happens in the `remove()` call.

Therefore, if there are multiple positions/keys, once one position is processed and then deleted (because the settlement price for that `assetId` was set), the last iteration of the for loop will try to access a position of the `position.keys` array that does no longer exists, reverting with an array out of bounds error. This is critical in the sense that the `_fundAndSettle()` function is called for any operation like transfers, liquidations and trades. Getting to this state would cause the funds to be permanently stuck in the subaccount.

Impact: High, because funds would be permanently stuck in the subaccount.

Likelihood: High, because it only requires 2 F&O positions opened concurrently in the same subaccount for this issue to be triggered when F&O's are enabled in future.

Recommendation: Consider using a `memory` pointer instead of a storage pointer, i.e.: `bytes32[] memory posKeys = positions.keys;`. Alternatively, consider moving the removal logic outside the loop.

Gravity: Fixed in PR 156.

Cantina Managed: Reviewed that PR 156 fixes this issue as recommended by moving the removal logic outside the loop.

5.2 High Risk

5.2.1 Lack of margin requirements enforced for options and futures will lead to bad debt for protocol

Severity: High Risk

Context: TradeContract.sol#L138-L140

Description: The current GRVT exchange implementation allows users to sell call options without requiring sufficient collateral or margin. Therefore, the seller may not have enough funds to cover the potential payout if the option is exercised. If the underlying asset's price rises significantly, the seller might not be have enough `spotBalance` to fulfill his obligation to deliver the asset at the strike price.

For example, lets suppose a user writes a call option for BTC with a strike price of 67000 USD. If BTC's price jumps to 80000 USD, the seller would have to face a huge liability and would not have enough collateral or liquidity to cover the difference. This shortfall would result a negative spot balance for the seller and bad debt for the protocol.

The same scenario described can occur with put options.

- Example:
    1. User1 buys a BTC/USDT call option. Strike price 65000, size 3.
    2. User2 sells a BTC/USDT call option. Strike price 65000, size 3.
    3. BTC prices raises to 70000 USD. USDT remains at 1 USD.
    4. Both positions are settled.
- Before settlement:

```
SUBACCOUNTSPOT BALANCES:
SubAccount ID -> 1
Currency.UNSPECIFIED -> 0
Currency.USD -> 0
Currency.USDC -> 0
Currency.USDT -> 6400000000
Currency.ETH -> 0
Currency.BTC -> 0

SUBACCOUNTSPOT BALANCES:
SubAccount ID -> 2
Currency.UNSPECIFIED -> 0
Currency.USD -> 0
Currency.USDC -> 0
Currency.USDT -> 13600000000
Currency.ETH -> 0
Currency.BTC -> 0

OPTIONS DATA
_subAccountId -> 1
_key -> 0
pos.id -> 0x00000000000000000000000000003b1dfde910000000068c6171400000030503
pos.balance -> 3000000000
pos.lastAppliedFundingIndex -> 0

OPTIONS DATA
_subAccountId -> 2
_key -> 0
pos.id -> 0x00000000000000000000000000003b1dfde910000000068c6171400000030503
pos.balance -> -3000000000
pos.lastAppliedFundingIndex -> 0
```

After settlement:

```
SUBACCOUNTSPOT BALANCES:
SubAccount ID -> 1
Currency.UNSPECIFIED -> 0
Currency.USD -> 0
Currency.USDC -> 0
Currency.USDT -> 21400000000
Currency.ETH -> 0
Currency.BTC -> 0

SUBACCOUNTSPOT BALANCES:
SubAccount ID -> 2
Currency.UNSPECIFIED -> 0
Currency.USD -> 0
Currency.USDC -> 0
Currency.USDT -> -1400000000 // <---------
Currency.ETH -> 0
Currency.BTC -> 0
```

Notice the negative subaccount balance(bad debt) in the seller's subaccount.

The same issue affects futures positions. Liquidations are a critical aspect of managing leveraged positions in futures trading. They protect the solvency of the trading platform by ensuring that losses do not exceed the trader's collateral in order to prevent bad debt.

Impact: High, because the protocol could incur significant bad debt.

Likelihood: Medium, because this scenario may easily be misused when option/futures support is enabled.

Recommendation: Consider enforcing a minimal margin when opening option and future positions. On the other hand, liquidations should be also supported for options and futures and not just for perpetual positions.

Gravity: Fixed in PR 166.

Cantina Managed: Reviewed that PR 166 fixes this issue by blocking non-perpetual trades.

Fix review update: The codebase now only supports perpetuals and has removed most of the logic for other instruments planned in future updates. So we can consider this issue as being mitigated.

5.2.2 Missing liquidation checks may cause unexpected behavior

Severity: High Risk

Context: LiquidationContract.sol, DataStructure.sol#L424-L442

Description: Gravity Exchange plans to support three types of liquidation based on the counterparty of the liquidation trade being their Insurance Fund, Backstop Liquidity Provider or another SubAccount that is above maintenance margin.

However, the liquidation logic is missing the following checks for different liquidation types:

1. There is no check for the specific liquidation type because the type is not captured in the liquidation order.
2. There is no check to ensure that `initiator == INSURANCE_FUND_SUB_ACCOUNT_ID` for "Full Liquidation" (in the project's terminology).
3. There is no check to ensure that `initiator != INSURANCE_FUND_SUB_ACCOUNT_ID` for "Partial Liquidation" (in the project's terminology). The `order.liquidationFees > 0` check is used as a proxy for partial liquidation but that is used only to collect liquidation fee.
4. There is no check to ensure `isAboveMaintenanceMargin(initiator)` for partial liquidation, which is a necessary requirement.
5. Liquidation to Backstop Liquidity Provider counterparty appears to be unsupported.

Impact: High, because the missing checks may cause unexpected/unsupported liquidation behavior deviating from what is specified.

Likelihood: Medium, because the different liquidation types are documented and expected to be supported at launch.

Recommendation: Consider adding the relevant missing checks to ensure implementation is in sync with documentation and specification, and onchain checks mirror all offchain (UI/Backend) ones.

Gravity: Fixed in PR 162.

Cantina Managed: Reviewed that PR 162 and other PRs significantly refactor liquidation logic to remove `LiquidationContract.sol` and integrate liquidation flow into the trade logic. These include:

1. Signature from insurance fund for all liquidation orders regardless of liquidation type.
2. Insurance Fund is backed by protocol to a certain extent and introduces socialized loss on withdrawals (SLOW).
3. Initial margin requirement is enforced on trades in the BackEnd/Risk Engine by replacing `_requireNonNegativeUsdValue()` with `isAboveMaintenanceMargin()`.
4. Partial liquidation may technically be supported by backend but will not be used in practice. ADL will not be supported. Liquidation to Backstop Liquidity Provider will not be supported.

5.2.3 Incorrect maintenance margin tier selection may lead to incorrect margin requirements and potential liquidations

Severity: High Risk

Context: LiquidationContract.sol#L198-L202

Description: The maintenance margin tier is incorrectly selected. Contract will always select the last margin tier even when user is eligible for lower margin tier. This is happening because the loop for selecting tier starts from 1 instead of 0, so that first tier can never be selected. Also `break` is missing in case a match is found and thus last tier is always selected.

Impact: High, because the ratio for the last margin tier will be high which means users will need to maintain more margin than required and not doing so will incorrectly increase the risk of liquidation.

Likelihood: Medium, because if users have typical position sizes in lower tiers then, due to this issue, incorrect margin will be required.

Proof of Concept:

1. Lets say, Tier 1 was for trade size up to `5000` USD and Tier 2 was for trade size up to `10000` USD.
2. Now if user trade size was `4000` USD then ideally Tier 1 should be selected but because loop starts from `i=1` `if (size.cmp(configs[i].size) < 0) {` which is `4000 < 10000` and so is true, idx becomes Tier 2 even though correct tier was Tier 1.

Recommendation: Revise the contract like below:

1. `break` introduced in the loop cycle.
2. `i` starts from 0.
3. In case of no match it means user trade is above last tier and so last tier should get selected. For this `idx` should be initialized with `configs.length-1`:

```
function _getMaintenanceMarginRatio(
    BI memory size,
    MaintenanceMarginConfig[MAX_M_MARGIN_TIERS] memory configs
  ) internal pure returns (BI memory) {
    uint idx = configs.length-1;

    for (uint i = 0; i < configs.length; i++) {
      if (size.cmp(configs[i].size) < 0) {
        idx = i;
        break;
      }
    }
    return configs[idx].ratio;
  }
```

Gravity: Fixed in PR 177.

Cantina Managed: The maintenance margin logic has been significantly refactored to address this issue. Reviewed that PR 177 with the refactored maintenance margin logic resolves this issue.

New logic now looks like:

```
0+ size, 1%
10+ size, 1.5%
100+ size, 2%
User A has size 14
Effective Margin Rate = (1% * 10 + 1.5% * 4) / 14 = 1.143%
User B has size 125
Effective Margin Rate = (1% * 10 + 1.5% * 90 + 2% * 25) / 125 = 1.56%
```

5.2.4 Hardcoded currency assumptions may lead to missed liquidations and bad debt for protocol

Severity: High Risk

Context: LiquidationContract.sol#L124-L140, LiquidationContract.sol#L215-L235, DataStructure.sol#L38-L45

Description: Gravity Exchange plans to support trading of spots, futures & options and perpetuals across different cryptocurrencies. The current launch aims to only support perpetuals trading of ETH and BTC. However, there are hardcoded currency assumptions about the initial launch target in the `_getAllMaintenanceMarginConfig()` logic:

1. `uint numCurrencies = uint(Currency.BTC) - uint(Currency.ETH) + 1;` assumes that any new currency will be added between BTC and ETH in `enum Currency`.
2. `uint numCurrencies = uint(Currency.BTC) - uint(Currency.ETH) + 1;` also assumes that derivatives will not be supported for USDC and USDT, which are placed before ETH in the enum.

These assumptions will lead to incorrect/non-existent maintenance margin configs from `_getAllMaintenanceMarginConfig()`, which will return a potentially lower maintenance margin for a subaccount from `_getSubMaintenanceMargin()`.

There are also hardcoded decimal usages which assume a value of 9 for ETH/BTC underlyings.

Impact: High, because using a potentially lower maintenance margin for a subaccount may lead to missed liquidations and protocol bad debt.

Likelihood: Medium, because Gravity Exchange anticipates supporting trading of spots, futures & options and perpetuals across different cryptocurrencies beyond the initial ETH and BTC. Adding those to the `enum Currency` or supporting USDC/USDT derivatives will require updating the above hardcoded logic.

Recommendation: Consider removing any hardcoded logic which assumes only ETH and BTC to appropriately make this future-proof for extending configuration/trading support to other currencies:

1. Evaluate iterating through all currencies listed in the enum and skip the ones with no Maintenance Margin config in `_getAllMaintenanceMarginConfig()`.
2. Evaluate the use of hardcoded decimals which assume a value of 9 for ETH/BTC underlyings but do not hold for USDC/USDT (6 decimals) or other future currencies with different precision expectations.

Gravity: Fixed in PR 177.

Cantina Managed: The maintenance margin logic has been significantly refactored to address the issues raised. Given that these changes far exceed typical fixes, this entire logic (including the fixes) will therefore be re-reviewed.

Fix review update: the refactored maintenance margin logic fixes the issue as recommended.

5.2.5 Wrong assumption that `_requireMarkPriceBI` function returns price in USD may lead to missed liquidations and bad debt creation

Severity: High Risk

Context: RiskCheck.sol#L38, LiquidationContract.sol#L231

Description: `_getPositionsUsdValue` internal function is used to compute the USD value of PERPS/FUTURES/OPTIONS positions the subAccount holds. The name would indicate that it returns the USD value of the positions, but it actually returns the value in the quote currency of the position. The key line is the `markPrice` retrieval for the `assetID` via `_requireMarkPriceBI` function:

```
function _requireMarkPriceBI(bytes32 assetID) internal view returns (BI memory) {
  (uint64 markPrice, bool found) = _getMarkPrice9Decimals(assetID);
  require(found, "mark price not found");
  return BI(int256(uint256(markPrice)), PRICE_DECIMALS);
}

function _getMarkPrice9Decimals(bytes32 assetID) internal view returns (uint64, bool) {
  Kind kind = assetGetKind(assetID);

  // If spot, process separately
  if (kind == Kind.SPOT) {
    return _getQuoteMarkPrice9Decimals(assetGetUnderlying(assetID));
  }

  Currency quote = assetGetQuote(assetID);
  // Only derivatives remaining
  (uint64 underlyingPrice, bool found) = _getUnderlyingMarkPrice9Decimals(assetID);
  if (!found) {
    return (0, false);
  }

  // If getting price in USD, we can simply scale and return
  if (quote == Currency.USD) {
    return (underlyingPrice, true);
  }

  // Otherwise, we have to convert to USDT/USDC price
  (uint64 quotePrice, bool quoteFound) = _getQuoteMarkPrice9Decimals(quote);
  if (!quoteFound) {
    return (0, false);
  }

  return (uint64((uint(underlyingPrice) * (10 ** PRICE_DECIMALS)) / uint(quotePrice)), true);
}
```

It can be seen that for non-SPOT assets that are not quoted in USD, the `markPrice` is expressed in the quote currency of the asset. The Gravity team has confirmed that this will usually be USDT.

Following this fact the:

- `_getPositionsUsdValue` function doesn't return the USD value of positions.
- The rest of the logic inside the `_getSubAccountUsdValue` returns the USD value of the SPOT balances.

Mixing of USD and USDT values can lead to incorrect reporting of the SubAccount's USD value if USDT de-pegs from USD.

In other instances, if the quote currency is not a stablecoin, the value of the SubAccount will be inflated or deflated depending on the price of the quote currency. With the current version of the codebase, only PERPS trading is implemented and there is no support for quoting PERPS in BTC/ETH but if this is added in the future the positions should be valued in USD and other appropriate logic modified accordingly.

- Other occurrences: `_getSubMaintenanceMargin` internal function should return the maintenance margin for the given SubAccount based on the margin configuration and perpetual positions.

  This value is compared with the SubAccount's USD value to determine if the SubAccount is liquidatable. The issue is that `_getSubMaintenanceMargin` returns the margin value in the quote asset, not in USD. While the SubAccount's balance is expected to be in USD. So again, it's comparing values in different units. This is due to using the `_requireMarkPriceBI` function used to compute the margin for each position:

```
BI memory sizeBI = BI(int256(size), _getBalanceDecimal(underlyingCurrency));
uint mmConfigIdx = uint(underlyingCurrency) - uint(Currency.ETH);
BI memory ratio = _getMaintenanceMarginRatio(sizeBI, mmConfigs[mmConfigIdx]);
// The charge for a perpetual positions = (Size) * (Mark Price) * (Maintenance Ratio)
BI memory charge = ratio.mul(_requireMarkPriceBI(id)).mul(sizeBI); // <<<
totalCharge = totalCharge.add(charge);
```

Impact: High, because it leads to incorrect reporting of the SubAccount's USD value which is vital for liquidations, trading, and risk management. The historical data is laid out in the likelihood section but considering a price of USDT equal to 0.9 USD would allow trading on subAccounts that might not have a positive USD balance and wouldn't trigger liquidations on subAccounts when they should occur causing bad debt in the system.

Likelihood: Medium, because significant depeg of stablecoins has happened before. By observing the historical price data for USDC and USDT we can see that there were two major depeg moments. In March 2023, the USDC price fell to 0.97 USD, while in April 2017, the USDT price was 0.91 USD. These are average prices while on some exchanges they were trading much lower.

Recommendation: Consider returning the USD value of the positions in the `_getPositionsUsdValue` function. This is already available by the `_getUnderlyingMarkPrice9Decimals` function. Similarly `_getSubMaintenance-Margin` function should always return the margin value in USD.

Gravity: Fixed in PR 191.

Cantina Managed: Reviewed that PR 191 fixes the issue by using `markPrice` in USD for `subAccount` value and Maintenance Margin calculation.

5.2.6 Addition of new currency will cause users to lose their spot balances

Severity: High Risk

Context: LiquidationContract.sol

Description: Once new currencies are added in future, BTC spot balances will become zero. This is explained in the proof of concept below.

Impact: High, because this would cause users to lose their BTC balance in Gravity exchange.

Likelihood: Medium, because Gravity Exchange anticipates supporting other cryptocurrencies beyond the initial ETH and BTC in the near future.

Proof of Concept:

1. Current currency enum is:

```
enum Currency {
  UNSPECIFIED, // 0
  USD, // 1
  USDC, // 2
  USDT, // 3
  ETH, // 4
  BTC // 5
}
```

2. This means the spot balance of BTC is tracked as `account.spotBalances[5]`.
3. `_getAllMaintenanceMarginConfig()` logic implies that new currencies will get added in-between ETH & BTC which means that some other currency may hold currency index 5 in future upon its addition.

```
function _getAllMaintenanceMarginConfig()
  private
  view
  returns (MaintenanceMarginConfig[MAX_M_MARGIN_TIERS][] memory)
{
  uint numCurrencies = uint(Currency.BTC) - uint(Currency.ETH) + 1;

  MaintenanceMarginConfig[MAX_M_MARGIN_TIERS][] memory configs = new
↪  MaintenanceMarginConfig[MAX_M_MARGIN_TIERS][](
    numCurrencies
  );

  // Add the maintenance margin config for each currency
  for (uint i = 0; i < numCurrencies; i++) {
    configs[i] = _getMaintenanceMarginConfigByCurrency(Currency(i + uint(Currency.ETH)));
  }
  return configs;
}
```

4. This will cause the new currency balance to become equal to BTC balance and BTC balance will become 0.

Recommendation: Consider placing any new currency added below BTC so that spot balances do not change. `_getAllMaintenanceMarginConfig()` should be changed accordingly to use currency enum length for navigating the loop.

Gravity: Fixed in PR 179.

Cantina Managed: Reviewed that PR 179 fixes this by hardcoding support for only BTC/ETH at launch. Future currencies/changes will be considered as part of upgrades.

5.2.7 Users may be unable to withdraw funds depending on the currency type

Severity: High Risk

Context: TransferContract.sol, TradeContract.sol

Description: Attacker can create a situation where protocol will not have funds in required currency to repay users. This will cause Victims to have funds but with no way of withdrawal. This is possible because current code cannot handle multiple quote currencies. We can open position using one quote but maintenance margin amount from another quote currency.

Impact: High, because Victim will not be able to withdraw their funds.

Likelihood: Medium, because USDC deposits are not currently allowed. Only USDT deposits are allowed.

Proof of Concept:

1. Lets say User A and User B has below balances (bridged from l1 and transferred to subAccount). Lets say subAccount quote currency is USDT:

```
User A
spot[USDC] = 70k
spot[USDT] = 0
```

```
User B
spot[USDC] = 70k
spot[USDT] = 0
```

2. User A opens long position of 1 BTC/USDT with limit price 65K and margin required is 65k. Since we have margin of 70k, this position is accepted with User B who opened same as short position:

```
User A
perp[BTC/USDT] = 1
spot[USDT] = -65k
spot[USDC] = 70k
```

```
User B
perp[BTC/USDT] = -1
spot[USDT] = 65k
spot[USDC] = 70k
```

3. subAccount (User A) transfers 5k USDT to main account M1, this is success since overall subAccount is still above/equal margin:

```
User A
perp[BTC/USDT] = 1
spot[USDT] = -70k
spot[USDC] = 70k
M1.spot[USDT] = 5k
```

4. M1 transfers this 5k to another account M2

```
M1.spot[USDT] = 0
M2.spot[USDT] = 5k
```

5. Attacker now simply closes the BTC/USDT position which makes balances:

```
User A
perp[BTC/USDT] = 0
spot[USDT] = -70k + 65k = -5k
spot[USDC]=70k
M1.spot[USDT] = 0
M2.spot[USDT] = 5k
```

```
User B
perp[BTC/USDT] = 0
spot[USDT] = 65k - 65k =0
spot[USDC] = 70k
```

6. Attacker transfers the 65k USDC to main account and withdraws it.
7. Now if M2 tries to withdraw it will fail since protocol does not have 5k USDT (protocol only has 70k USDC).
8. If the protocol had funds then also it would become an issue for the one withdrawing last for whom protocol wont have required funds.
9. The victim will incur loss of 5k.

- Case 2.

    1. Lets say User A has the below balances (bridged from L1 and transferred to subAccount). Let's say `subAccount` quote currency is USDT.

    ```
    User A
    spot[USDC] = 65k
    spot[USDT] = 5k
    ```

    2. User B has the below balance (bridged from L1 and transferred to `subAccount`):

    ```
    User B
    spot[USDC] = 65k
    spot[USDT] = 0
    ```

3. So the exchange has 130k USDC and 5k USDT.
4. User A opens a long position of 1 BTC/USDT with limit price 65K and margin required is 65k. Since we have margin of 70k (combined USDC and USDT), this position is accepted:

```
User A
perp[BTC/USDT] = 1
spot[USDC] = 65k
spot[USDT] = -65k + 5k = -60k
```

5. User B opens a short position (which was matched with User A in Step 4) of 1 BTC/USDT with limit price 65K and margin required is 65k. Since we have margin of 65k, this position is accepted:

```
User B
perp[BTC/USDT] = -1
spot[USDT] = 65k
spot[USDC] = 65k
```

6. User A now simply closes the long BTC/USDT position in profit of 5k, so sell price is 70k. This means User B incurs a loss of 5k:

```
User A
perp[BTC/USDT] = 0
spot[USDT] = -60k + 70k = 10k
spot[USDC] = 65k
```

```
User B
perp[BTC/USDT] = 0
spot[USDT] = 65k - 70k = -5k
spot[USDC] = 65k
```

7. Attacker transfers the 10k USDT to main account and tries withdrawing it.
8. It will fail since the protocol does not have 10k USDT (protocol only has 5k USDT balance) .

Recommendation: Until the design is compatible with multiple currencies, `subAccount` should only be allowed spot balance in their own quote currency. So if `subAccount` has quote currency of USDT then it should only be allowed to have USDT spot balance, which also means it can only open positions with USDT as quote.

Gravity: Fixed in PR 166.

Cantina Managed: Reviewed that PR 166 fixes this as recommended.

5.3 Medium Risk

5.3.1 Incorrect `order.liquidationFees` for multi-legged orders will cause users to pay more fees than intended

Severity: Medium Risk

Context: LiquidationContract.sol#L116

Description: The `LiquidationContract` contract implements the function `liquidateOneOrder()`:

```
function liquidateOneOrder(
  LiquidationOrder calldata order,
  uint64 passiveSubAccountID,
  LiquidationType liquidationType
) private {
  SubAccount storage initiator = _requireSubAccount(order.subAccountID);
  SubAccount storage passive = _requireSubAccount(passiveSubAccountID);

  _fundAndSettle(initiator);
```

```
    _fundAndSettle(passive);

    // Validate the liquidation order based on its type
    if (liquidationType == LiquidationType.LIQUIDATE) {
      require(!isAboveMaintenanceMargin(passive), "Margin is above maintenance level");
    } else if (liquidationType == LiquidationType.AUTO_DELEVERAGE) {
      require(isAboveMaintenanceMargin(passive), "Margin is below maintenance level");
      require(order.liquidationFees == 0, "ADL takes no fee");
    }

    // Validate: The initiator must have the permission to trade
    _requireSubAccountPermission(initiator, order.signature.signer, SubAccountPermTrade);
    // Validate the initiator's signature must be valid
    _preventReplay(hashLiquidationOrder(order), order.signature);

    // Update positions and spot balances based on whether the initiator is buying or selling the asset
    for (uint i; i < order.legs.length; i++) {
      OrderLeg calldata leg = order.legs[i];
      Position storage initiatorPos = _getOrCreatePosition(initiator, leg.assetID);
      Position storage passivePos = _getOrCreatePosition(passive, leg.assetID);

      int64 sizeDelta = leg.isBuyingAsset ? -int64(leg.size) : int64(leg.size);
      _requireReduceOnly(passivePos.balance, sizeDelta);

      Currency quoteCurrency = assetGetQuote(leg.assetID);
      BI memory size = BI(int256(uint256(leg.size)), _getBalanceDecimal(assetGetUnderlying(leg.assetID)));
      BI memory limitPrice = BI(int256(uint256(leg.limitPrice)), PRICE_DECIMALS);
      int64 notionalValue = size.mul(limitPrice).toInt64(_getBalanceDecimal(quoteCurrency));
      if (leg.isBuyingAsset) {
        initiatorPos.balance += int64(leg.size);
        passivePos.balance -= int64(leg.size);
        initiator.spotBalances[quoteCurrency] -= notionalValue;
        passive.spotBalances[quoteCurrency] += notionalValue - int64(order.liquidationFees);
      } else {
        initiatorPos.balance -= int64(leg.size);
        passivePos.balance += int64(leg.size);
        initiator.spotBalances[quoteCurrency] += notionalValue;
        passive.spotBalances[quoteCurrency] -= notionalValue + int64(order.liquidationFees);
      }
    }

    // Post trade validation, all parties should have equity >= 0
    _requireNonNegativeUsdValue(initiator);
    _requireNonNegativeUsdValue(passive);

    // Pay fees to insurance fund, in case of partial liquidation
    if (order.liquidationFees > 0) {
      (uint64 feeSubID, bool feeSubFound) = _getUintConfig(ConfigID.ADMIN_LIQUIDATION_SUB_ACCOUNT_ID);
      require(feeSubFound, "fee account not found");
      _requireSubAccount(feeSubID).spotBalances[order.feeCurrency] += int64(order.liquidationFees);
    }
}
```

The `liquidateOneOrder()` function facilitates the liquidation process of a single order, either by liquidating a position due to insufficient maintenance margin or through an automatic deleveraging mechanism. It ensures that the subaccounts involved (initiator and passive) are properly funded and settled before liquidation, validates the conditions for liquidation and updates the positions and spot balances accordingly. The function also handles the transfer of any liquidation fees to the designated fee sub-account.

However, the implementation subtracts the `order.liquidationFees` from the passive subaccount's balance within a loop that iterates over all legs of the order. This means the fees are being deducted multiple times—once per

leg—while the addition to the fee subaccount occurs only once.

This is incorrect because the `order.liquidationFees` represent the total fee for the entire order, not for each individual leg. The fees should only be deducted once from the passive account, regardless of the number of legs in the order.

Notice how the fee substraction from the passive subaccount is performed within a for loop, once per leg while the addition of the `order.liquidationFees` is done just once to the `feeSubID` subaccount:

```
for (uint i; i < order.legs.length; i++) {
  // ...
  if (leg.isBuyingAsset) {
    // ...
    passive.spotBalances[quoteCurrency] += notionalValue - int64(order.liquidationFees); // <----------
  } else {
    // ...
    passive.spotBalances[quoteCurrency] -= notionalValue + int64(order.liquidationFees); // <----------
  }
  // ...
}
// ...
// Pay fees to insurance fund, in case of partial liquidation
if (order.liquidationFees > 0) {
  (uint64 feeSubID, bool feeSubFound) = _getUintConfig(ConfigID.ADMIN_LIQUIDATION_SUB_ACCOUNT_ID);
  require(feeSubFound, "fee account not found");
  _requireSubAccount(feeSubID).spotBalances[order.feeCurrency] += int64(order.liquidationFees); //
↪   <----------
}
```

Impact: Medium, because users would be paying higher `liquidationFees` with every liquidation that has multiple legs. If a liquidation had 10 legs, the total `liquidationFees` paid by the user would be multiplied by 10.

Likelihood: Medium, because this scenario should occur only with multi-legged orders.

Recommendation: Consider updating the `liquidateOneOrder()` function to only subtract once the `order.liquidationFees` from the `passive` subaccount balance.

Gravity: Fixed in PR 162.

Cantina Managed: Reviewed that PR 162 fixes this issue as recommended.

Fix review update: The codebase in scope for fix-review has completely refactored this liquidation logic. In the new code, this function is not implemented and therefore we can consider this issue solved.

5.3.2 `tradeDeriv()` will always revert because of incorrect whole/partial check when multiple order legs operate on same asset

Severity: Medium Risk

Context: TradeContract.sol#L181-L187

Description: `tradeDeriv()` will revert when multiple order legs operate on the same asset. This is because `total` (see below snippet) checks the cumulative total of matched size of the order for assets. `total` will always be greater than `leg.size` for second leg onwards if an order has multiple legs operating on the same asset.

```
bool isWholeOrder = order.timeInForce == TimeInForce.ALL_OR_NONE || order.timeInForce ==
↪   TimeInForce.FILL_OR_KILL;
for (uint i; i < legsLen; ++i) {
    OrderLeg calldata leg = legs[i];
    uint64 total = executedSize[leg.assetID] + tradeSizes[i];
    require(isWholeOrder ? total == leg.size : total <= leg.size, ERR_INVALID_MATCHED_SIZE);
    executedSize[leg.assetID] = total;
}
```

For example, a partial order with two legs, e.g. with `tradeSizes` = [500, 200] and legs of sizes [500, 500] with same asset ID, the `require` will revert for `total <= leg.size` in the second leg because `total == 700` and `leg.size == 500`.

For another example, a whole order with two legs, e.g. with `tradeSizes` = [500, 500] and legs of sizes [500, 500] with same asset ID, the `require` will revert for `total == leg.size` in the second leg because `total == 1000` and `leg.size == 500`.

Impact: Medium, because `tradeDeriv()` will revert which will prevent that order and related future orders from executing. Such orders with multiple legs operating on same asset would have to be prevented from being created or rejected from execution at the Gravity Exchange UI or Backend, and also in onchain contracts for mirroring offchain-onchain checks and state. If these have to be allowed then the Exchange contract will have to be upgraded and its state would need to sync-up with that of Gravity Exchange afterwards.

Likelihood: Medium, because this issue affects orders with multiple legs that operate on same asset, which is not prevented or checked in the onchain contracts. Also, it is unclear if this is prevented/checked in Gravity Exchange UI or Backend because that is out of scope for this review.

An example of why such an order is reasonable to expect is illustrated below:

Leg-1: The taker places an order-leg to buy a 3 BTC/USDT call option at a limit price of 1200. Leg-2: The trader places another order-leg to sell a 3 BTC/USDT call option (i.e. same asset as Leg 1) at a limit price of 1500.

This order with two legs can be used to capture profits from market volatility. The taker is setting up an order to capture this volatility where they will buy the asset at a lower price in Leg-1 and sell it at a higher price in Leg-2, locking in a profit if both legs are executed. This could be used in a range-bound market where the trader expects the price of BTC/USDT to oscillate between the two limit prices, allowing them to buy low and sell high automatically within the same order.

Recommendation: Consider:

1. Changing simplifying the check as illustrated below:

```
bool isWholeOrder = order.timeInForce == TimeInForce.ALL_OR_NONE || order.timeInForce ==
↪  TimeInForce.FILL_OR_KILL;
for (uint i; i < legsLen; ++i) {
    OrderLeg calldata leg = legs[i];
    require(isWholeOrder ? tradeSizes[i] == leg.size : tradeSizes[i] <= leg.size,
↪  ERR_INVALID_MATCHED_SIZE);
}
```

2. Checking to prevent such orders both in Gravity Exchange UI+Backend and in the onchain contracts for mirroring offchain-onchain checks and state.

Gravity: Fixed in PR 161.

Cantina Managed: Reviewed that PR 161 fixes this issue by checking that orders containing multiple legs have unique `assetID`.

5.3.3 `_findLegIndex()` could allow draining of protocol by creating orders containing multiple legs with the same `assetID`

Severity: Medium Risk

Context: TradeContract.sol#L67

Description: The `_findLegIndex()` function implemented in the `TradeContract` is designed to locate the index of a leg within an array of `OrderLeg` objects based on the `assetID`. However, the current implementation becomes problematic when an order contains multiple legs with the same `assetID`.

```
function _findLegIndex(OrderLeg[] calldata legs, bytes32 assetID) private pure returns (uint) {
  uint len = legs.length;
  for (uint i; i < len; ++i) if (legs[i].assetID == assetID) return i;
  revert(ERR_NOT_FOUND);
}
```

For instance, consider the following scenario where a taker submits an order with two legs that have the same `assetID`:

```
// Leg 1: Buy the asset
legs[0].assetID = encodeAssetID(asset);
legs[0].size = uint64(3_000000000);
legs[0].limitPrice = uint64(1200_000000000);
legs[0].isBuyingAsset = true;

// Leg 2: Sell the asset at a higher price
legs[1].assetID = encodeAssetID(asset);
legs[1].size = uint64(3_000000000);
legs[1].limitPrice = uint64(1500_000000000);
legs[1].isBuyingAsset = false;
```

In this scenario, the taker aims to buy and then sell the same asset (BTC/USDT call options) within the same order. This could be an effective strategy to capture profits from market volatility. However, the `_findLegIndex()` function will only find and process the first leg with the matching `assetID`. The second leg will be ignored. These inconsistencies caused in the `takerMatchedSizes` array can result in an incorrect order execution and also in the manipulation of the trading system. Specifically, a malicious user could abuse this to, for example, purchase CALL options for free.

Impact: High, because it could lead to loss of funds for the protocol.

Likelihood: Low, because it requires orders containing multiple legs with the same `assetID` and the sum of the order `matchedSizes` for the legs with the same `assetID` to be at least less than the half of the `legs.size`.

Recommendation: Consider updating the `_findLegIndex()` function to handle scenarios where multiple legs with the same `assetID` exist within an order. Another possible solution is to enforce at smart contract level that no legs has the same `assetID` within an order.

Gravity: Fixed in PR 161.

Cantina Managed: Reviewed that PR 161 fixes this issue by checking that orders containing multiple legs have unique `assetID`.

5.3.4 Incomplete support for futures and options will lead to incorrect behavior

Severity: Medium Risk

Context: Global scope

Description: Gravity Exchange plans to support trading of spots, futures and options (F&O) and perpetuals. While there is considerable logic and validation to support F&O, it is incomplete in areas of liquidation and margin coverage among potentially many other related aspects.

Impact: High, because orders trading F&O will not execute as expected leading to incorrect accounting and state updates.

Likelihood: Low, because Gravity Exchange plans to launch only with support for perpetuals and so it is expected that user orders for F&O will be prevented in their UI or Backend logic.

Recommendation: Consider:

1. Preventing F&O order creation in UI or Backend logic.

2. Preventing F&O order execution even in `tradeDeriv()` by explicitly checking for them to mirror onchain and offchain checks for transparency.
3. Documenting and highlighting to users about this critical missing feature.

Gravity: Fixed in PR 166.

Cantina Managed: Reviewed that PR 166 adds `revert("not supported")` in `settlementPriceTick()` (and few other relevant functions), which should effectively prevent any accidentally placed but unsupported trades related to futures and options. However, there is still logic present in the codebase for F&O's which will need to be reviewed carefully in the context of any future updates made to enable their trading.

Fixm review update: Support for F&O's has been mostly removed in the recent refactors.

5.3.5 Missing check for perpetual assets may cause unexpected behavior

Severity: Medium Risk

Context: TradeContract.sol#L16-L56, LiquidationContract.sol#L83-L87

Description: Gravity Exchange plans to support trading on spots, futures and options (F&O) and perpetuals but is anticipating to launch only with support for perpetuals. As such, this check to ensure that trades are being executed and liquidated *only* for perpetual assets is critical but is missing across contract logic.

Impact: High, because trades and liquidations on any asset kind other than perpetuals will result in unexpected behavior due to missing or incorrect/untested logic.

Likelihood: Low, because Gravity Exchange anticipates to allow only perpetual asset trading on launch and therefore plans to enforce relevant checks on their Backend/UI.

Recommendation: Consider enforcing `assetGetKind(leg.assetID) == Kind.PERPS` check across all contracts where assets are being received for execution or liquidation. Ensuring mirroring of onchain-offchain checks is critical for transparency, which is one of the project's stated motivation for their hybrid architecture.

Gravity: Fixed in PR 166.

Cantina Managed: Reviewed that PR 166 adds `revert("not supported")` in `settlementPriceTick()` (and few other relevant functions), which should effectively prevent any accidentally placed but unsupported trades related to futures and options.

Fix review update: Support for F&O's has been mostly removed in the recent refactors.

5.3.6 `limitPrice` not being checked against matching orders allows them to be executed at a higher price than intended

Severity: Medium Risk

Context: TradeContract.sol#L58

Description: Each order has one or more legs, and each leg has a `limitPrice` and size. The notional value that will be deducted from the subAccount's spot balance is calculated as `limitPrice * size` for all the legs. In the current implementation of the `tradeDeriv()` function, only the `limitPrice` of the maker orders is taken into account.

Take the following example:

- The maker order is selling 3 BTC PERP for 60k USDT each.
- The taker order is buying 3 BTC PERP for 50k USDT each.
- This gets matched and the taker's subAccount will be deducted 180k USDT (`3 * 60k`) instead of 150k USDT (`3 * 50k`).

Impact: High, because it ignores the `limitPrice` of the matching order, which allows orders to be executed at a higher price than intended.

Likelihood: Low, because the backend is expected to validate the order for `limitPrice` before sending it to the smart contract.

Recommendation: Consider adding checks for enforcing that the `limitPrice` of the maker and taker orders are the same before executing the trade.

Gravity: Fixed in PR 196.

Cantina Managed: Reviewed that PR 196 fixes the issue by ensuring that taker is not buying for a higher price or selling for a lower price than what is specified in the `limitPrice`.

5.3.7 Delayed funding settlement can skip liquidation leading to bad debt for protocol

Severity: Medium Risk

Context: FundingAndSettlement.sol

Description: The funding settlement on Perp positions is only done when trader performs some operation on the exchange. This means exchange will not know if User account can be liquidated after fund settlement if User is not performing any operation.

Impact: High, because Exchange will miss calling liquidation on accounts which will give rise to bad debt for protocol.

Likelihood: Low-Medium, because the backend liquidation engine was not keeping track of User balance after every fund settlement.

Note: Since this requires an external factor (Liquidation engine) that is out of scope for this review (does not impact contract), severity is downgraded to Medium.

Proof of Concept:

1. Trader A has an active perp position.
2. Post 8hrs new funding rate comes which would have cause liquidation to Trader A position.
3. Since Trader A has not performed any operation on his sub Account, thus new funding rate does not apply.
4. Since the backend liquidation engine only checks the current balance of Trader A, it will seem healthy even though it is not actually.

Recommendation: Backend liquidation engine should keep track of User's position with current funding rate so that it can call liquidation at right time.

Gravity: This is fixed in our Backend, there's no follow up in the contract because it's not responsible for detecting liquidatable trading account in real time. Our contract only receive liquidation transaction from Backend and carry out the instruction.

Cantina Managed: Acknowledged. Fix has been made on Backend which is not in scope of this review.

5.3.8 User can bypass margin requirements by splitting positions across accounts

Severity: Medium Risk

Context: LiquidationContract.sol

Description: A trader can split his perp position among several sub accounts so that he will fall under margin tier with lower charge.

Impact: Medium, because Exchange will hold lower collateral than required. This causes the platform to have a lower overall margin buffer, increasing the risk of default, especially during volatile market conditions.

Likelihood: Medium, because this will require user to plan his positions within multiple sub account so that overall margin requirement becomes lesser. If planned well by Exchange, this will be difficult to execute.

Proof of Concept:

1. Lets say platform requires `5%` for up to `100` USD and above that `10%` maintenance margin.
2. If user uses `200` USD from a single subAccount, then he will need to maintain maintenance margin of `20` USD.
3. However if he splits it between 2 subAccount with `100` USD each then maintenance margin will be `5 + 5 = 10`.
4. This causes the platform to have a lower overall margin buffer, increasing the risk of default, especially during volatile market conditions.

Recommendation: 1. Exchange can choose maintenance tier from account's overall position, if account type is User Account. 2. Exchange can make tiers carefully so that splitting wont be much beneficial.

Gravity: Will not fix. Reason: too costly to do this on chain.

1. We acknowledge that there's no check in the smart contract to prevent this abuse.
2. However, we decided not to add this check to the contract, and instead add more controls in our BE: limits to the number of subaccounts + requiring more initial margin if you have more subaccounts.

Cantina Managed: Acknowledged. Team will be adding more checks on Backend and not on Contracts due to above stated reasons. Because Backend is not part of scope for this audit, this is marked as Acknowledged.

5.3.9 Margin checks missing in multiple flows may lead to bad debt for protocol

Severity: Medium Risk

Context: TransferContract.sol, TradeContract.sol

Description: While transferring from subAccount, there is no check to see that subAccount is above Maintenance Margin post transfer. This can allow subAccount to create position with 0 collateral/margin which on loss will create bad debt. This bad debt will then need to be covered by insurance causing loss to protocol. This also happens while trading where you can open a perp position without having any margin.

Impact: High, because this will cause bad debt which will need to covered by insurance causing loss to protocol.

Likelihood: Low, because Project mentioned that Backend (BE) / Risk Engine (RE) is performing this check and this check is only missing in contracts.

Proof of Concept: 1. Say, User A has margin requirement of `100` USD and position opened is `1000` USD. 2. Instantly, User can transfer this `100$` to another subAccount. This should be allowed as User net position after transfer is `0` (perp position has not moved yet as price is same). 3. With any negative movement in perp position, liquidation will be called causing bad debt to protocol as collateral is not present.

- Another case:
    1. User has 0 balance.
    2. Users signs a taker order such that:

```
a. Leg 1 buys BTC at X
b. Leg 2 sells BTC at X+20
```

    3. If this order is matched and executed, User will have an open position even though he does not maintain required margin.

Recommendation: Consider replacing all instances of `_requireNonNegativeUsdValue` with `isAboveMaintenanceMargin` in transfer and trading flows.

Gravity: Our decision is to keep the checks as is and not add initial margin check. This is because

1. In our risk engine, all `_requireNonNegativeUsdValue` checks are initial margin (IM) check (margin required to open a position, always higher than maintenance margin(MM)):

- That is, once a sub account falls below initial margin, even if it has non-zero equity and is above maintenance margin, we don't allow transfers out of it.

- Initial margin checks are not implemented on chain because it is expensive to(option IM calculation requires Black–Scholes model).

2. It is reasonable and in the platform's interest to block transfers that causes subaccount to fall below IM.
3. Our contract is focused on protecting our users' interest. The platform can protect its own interest with the risk engine.

Cantina Managed: Acknowledged. Product team mentioned that they have placed initial margin checks in the Backend at all places where `_requireNonNegativeUsdValue` is used. Because Backend is not in scope for this audit, this is marked Acknowledged.

5.3.10 `GRVTExchange` not processing `IMMEDIATE_OR_CANCEL` orders correctly may lead to unexpected behavior

Severity: Medium Risk

Context: TradeContract.sol#L67

Description: The `IMMEDIATE_OR_CANCEL` (IOC) order type is designed to execute immediately, filling as much of the order as possible. Any portion of the order that cannot be instantly filled should be canceled. However, in the current implementation of the `GRVTExchange`, this behavior is not fully enforced.

For instance, if Alice submits a taker buy order for 10000 USDT, and it is matched with a sell order for only 5000 USDT, the remaining 5000 USDT of Alice's order should be automatically canceled. However, under the current implementation, this remaining portion of the order can still be matched with another maker order through a subsequent `tradeDeriv()` call, which is not in line with the expected behavior of IOC orders.

Impact: Medium, because users may not get the expected behavior from the protocol.

Likelihood: Medium, because partial fills will be common, especially in volatile markets.

Recommendation: Consider updating the `GRVTExchange.tradeDeriv()` function to ensure that any unfilled portions of an IOC order are automatically canceled after the first execution. This can be done by implementing a check within the function to cancel any remaining order size that was not matched during the initial call. For example:

```
function tradeDeriv(int64 timestamp, uint64 txID, Trade calldata trade) external {
  // ...

  // After matching the order, cancel any remaining size for IOC orders
  if (takerOrder.timeInForce == TimeInForce.IMMEDIATE_OR_CANCEL && remainingSize > 0) {
    // Logic to cancel the remaining unfilled portion of the order
    cancelOrder(remainingSize);
  }

  // ...
}
```

Gravity: Fixed in PR 197.

Cantina Managed: Reviewed that PR 197 fixes this issue by preventing execution of IOC orders that have a non-zero `executedSize`.

5.3.11 TradeContract not checking that `asset.expiration` is greater than `state.timestamp` allows draining the exchange

Severity: Medium Risk

Context: TradeContract.sol#L16

Description: The `TradeContract` implements the function `tradeDeriv()`. This function allows trading with perpetuals, options and futures. However, the function does not enforce that the `asset.expiration` that is part of the `assetId` is actually higher than the current `state.timestamp`. Therefore, a malicious user could open a position with an `assetId` that was already settled and simply close the position right away ensuring a profit.

Impact: High, because it would allow draining the exchange.

Likelihood: Low, because this check is expected to be already implemented in the Risk Engine/backend.

Recommendation: Consider adding a require check in the `tradeDeriv()` function that enforces that the `asset.expiration` is always higher than `state.timestamp + MINIMUM_PERIOD` where `MINIMUM_PERIOD` is, for e.g, 1 day.

Gravity: Fixed in PR 180.

Cantina Managed: Reviewed that PR 180 fixes this issue by enforcing a `require(expiry > timestamp, "asset expired");` check for futures and options.

5.3.12 `isAboveMaintenanceMargin()` not checking `MarginType` of subaccount may lead to unexpected liquidations

Severity: Medium Risk

Context: LiquidationContract.sol#L179-L184

Description: The `LiquidationContract` implements the function `isAboveMaintenanceMargin()`:

```
/**
 * @dev Check the current subaccount margin level. If the subaccount is below the maintenance margin,
 * it is liquidatable.
 * @param subAccount The subaccount to check.
 * @return True if the subaccount is below the maintenance margin, false otherwise.
 */
function isAboveMaintenanceMargin(SubAccount storage subAccount) private returns (bool) {
  uint usdDecimals = _getBalanceDecimal(Currency.USD);
  int64 subAccountValue = _getSubAccountUsdValue(subAccount).toInt64(usdDecimals);
  uint64 maintenanceMargin = _getSubMaintenanceMargin(subAccount);
  return subAccountValue >= 0 && uint64(subAccountValue) >= maintenanceMargin;
}
```

This function checks the current subaccount margin level. If the subaccount is below the maintenance margin, it is liquidatable.

On the other hand, when a subaccount is created the user can choose different margin types. The expected behaviours of the different margin types are:

- `ISOLATED`: The margin assigned to a position is limited to a specific amount that is isolated from the rest of the account's balance. If the position incurs a loss that exceeds the isolated margin, the position will be liquidated, but the loss will not affect other positions or balances in the account.
- `SIMPLE_CROSS_MARGIN`: All the margin available in the account is pooled together to cover losses on any position. This means that the entire balance of the account can be used to prevent liquidation of any individual position. Losses on one position can be offset by gains on another.
- `PORTFOLIO_CROSS_MARGIN`: The margin requirements are calculated based on the overall risk of the entire portfolio of positions. This type of margin considers correlations between different positions and potentially reduces the margin requirement if the portfolio as a whole is deemed to be less risky.

However, based on the current `isAboveMaintenanceMargin()` implementation, the contract always assumes that the `MarginType` is `SIMPLE_CROSS_MARGIN` as the `_getSubAccountUsdValue()` function iterates over all the different positions opened to calculate the total USD value of the account.

Impact: High, because this may lead to unexpected liquidations in future when the different margin types are supported. If the users are correctly informed beforehand, then this could be an expected behavior. They would simply have to create and operate on different subaccounts if they want an `ISOLATED` `MarginType`.

Likelihood: Low, because although this affects every subaccount created, the project plans to launch only with `SIMPLE_CROSS_MARGIN` support for now. The initial GRVT Smart Contracts Launch Parameters can be found in this google document.

Recommendation: Consider:

1. Updating the `isAboveMaintenanceMargin()` and `_getSubAccountUsdValue()` functions to take into account the `MarginType` of the subaccount in future.
2. Replacing `require(marginType != MarginType.UNSPECIFIED, "invalid margin");` with `require(marginType == MarginType.SIMPLE_CROSS_MARGIN, "invalid margin");` in `setSubAccountMarginType()` for now.
3. Allowing only `marginType == MarginType.SIMPLE_CROSS_MARGIN` in `createSubAccount()` for now.

Gravity: Fixed in PR 166 and PR 176.

Cantina Managed: Reviewed that PR 166 and PR 176 fix this issue as recommended.

5.3.13 Incorrect choice and time of risk check may allow margin bypass or deny valid order execution

Severity: Medium Risk

Context: TradeContract.sol#L138-L140, TradeContract.sol#L232, RiskCheck.sol#L14-L52, LiquidationContract.sol#L179-L184

Description: `tradeDeriv()._verifyAndExecuteOrder()` performs `_requireNonNegativeUsdValue(sub)` before and after `_executeOrder()` to ensure "sufficient balance pre and post trade." However, `_executeOrder()` performs `_fundAndSettle(sub)` before executing the order to fund any perpetual positions and settle any future/option positions.

There are two issues with this:

1. The choice of `_requireNonNegativeUsdValue(sub)` instead of `isAboveMaintenanceMargin()` allows order execution from sub accounts that do not meet the margin requirement and have not been liquidated yet for some reason. This breaks a critical invariant of the protocol.
2. Performing the risk check before `_fundAndSettle(sub)` causes stale position values to be used. If the risk check fails for scenarios where calling `_fundAndSettle(sub)` would in fact have allowed it to pass, that causes a DoS for valid order executions.

Impact: High, because

1. It breaks critical protocol invariant of margin maintenance for order execution which leads to bad debt, however such orders may be susceptible to liquidation thereafter and...
2. Valid order executions may be prevented by delayed application of `_fundAndSettle(sub)`.

Likelihood: Low, because this requires specific conditions to be met on subaccount's positions, funding and settlement and Gravity Exchange Backend is expected to have relevant checks.

Recommendation: Consider replacing `_requireNonNegativeUsdValue(sub)` with `isAboveMaintenanceMargin()` and calling `_fundAndSettle(sub)` before applying the risk check.

Gravity: Fixed in PR 157 and PR 194.

Cantina Managed: Reviewed that PR 157 and PR 194 fix the issues as recommended.

5.3.14 Critical exchange parameters are missing configuration timelock rules

Severity: Medium Risk

Context: ConfigContract.sol#L403-L662, ConfigContract.sol#L35-L43

Description: Exchange configuration parameters are declared in `ConfigContract.sol` and allowed to be changed by `CONFIG_ADDRESS` by first scheduling a change behind a timelock rule via `scheduleConfig()` and then executing that change after timelock expiration via `setConfig()`. Every config change is expected to be timelocked per comment: "*Every config change is timelocked*".

However, timelock rules are currently configured only for `ConfigID.SM_FUTURES_INITIAL_MARGIN` (which is not used in the initial launch supporting only perpetuals) and is missing for critical exchange parameters including `ADMIN_RECOVERY_ADDRESS`, `ORACLE_ADDRESS`, `CONFIG_ADDRESS`, `MARKET_DATA_ADDRESS`, `ERC20_ADDRESSES`, `L2_SHARED_BRIDGE_ADDRESS`, `BRIDGING_PARTNER_ADDRESSES` and all the others. This allows all of them to be changed to different values immediately without any time delay because `_getLockDuration()` returns 0 when `rules.length == 0`.

Also, there is no way to add rules to existing parameters and this has to be initialized in the default configuration during deployment.

Impact: High, because this breaks the requirement of every config change being timelocked, which could have serious implications if/when the values are accidentally changed to incorrect values or maliciously updated to attacker-controlled/-favoring values.

Likelihood: Low, because this requires accidental changes or compromised roles. While only `CONFIG_ADDRESS` is authorized to call `scheduleConfig()` and `setConfig()`, documentation indicates that this will be an EOA controlled by Gravity team.

Recommendation: 1. Consider adding appropriate timelock rules to all exchange config parameters. 2. Consider using an appropriately chosen/configured multisig for `CONFIG_ADDRESS`.

Gravity: Fixed in PR 168 and PR 186.

Cantina Managed: Reviewed that PR 168 and PR 186 fix the issues as recommended.

5.3.15 Admin will not be able to withdraw fees completely leading to stuck funds

Severity: Medium Risk

Context: TransferContract.sol

Description: Admin will never be able to withdraw the collected withdraw fees fully. This happens because while withdrawing, admin is again required to give withdrawal fees. This means that a small amount of fees always remain stuck with no way of recovery.

Impact: Low, because the minimum withdrawal fees per currency will remain stuck with no way of recovery.

Likelihood: High, because withdrawal always takes fees even when Admin is withdrawing.

Proof of Concept:

1. `feeSubAccId` collects `5 USDT`.
2. `feeSubAccId` subAccount transfer the fees to Admin account.
3. Admin tries withdrawing the `5 USDT`.
4. Protocol takes withdrawal fees of say `1 USDT` and adds this `1 USDT` in `feeSubAccId` subAccount even though withdrawer is Admin.
5. Admin gets `4 USDT` and `1 USDT` is stuck with `feeSubAccId`.
6. Admin cannot withdraw the `1 USDT` as again he has to give `1 USDT` as fees making resulting withdrawal as `0`. Similar situation for any other currency.

Recommendation: Consider not taking withdrawal fees if withdrawer is Admin account.

Gravity: After internal alignment we decided not to fix this issue, as GRVT is the only party affected by it, and as the amount involved is small, we would like to keep the logic simple here.

Cantina Managed: Acknowledged as accepted risk.

5.3.16 Not tracking/enforcing staleness of mark and funding prices may lead to unexpected behavior due to incorrect valuations

Severity: Medium Risk

Context: OracleContract.sol#L25-L52, OracleContract.sol#L97-L139

Description: `markPriceTick()` is expected to be called by the `ORACLE_ADDRESS` at regular time intervals to update the mark prices of different assets. `_verifyPriceUpdateSig()` has a `sig.expiration >= timestamp - MAX_PRICE_TICK_SIG_EXPIRY && sig.expiration <= timestamp` check to ensure that Oracle's `sig.expiration` is within the bounds of `timestamp` of the price tick (`MAX_PRICE_TICK_SIG_EXPIRY` is currently configured as one minute). While this ensures that the Oracle's signature/timestamp is not more than one minute older than the price tick's `timestamp`, there is no association and tracking of this `timestamp` with the mark price to enforce a maximum staleness threshold against the next price tick update.

`fundingPriceTick()` has a similar issue where the funding price for perpetuals is updated by the `MARKET_DATA_ADDRESS`. `_verifyFundingTickSig` not only has a `sig.expiration >= timestamp - MAX_PRICE_TICK_SIG_EXPIRY && sig.expiration <= timestamp` check to ensure that Market Data's `sig.expiration` is within the bounds of `timestamp` of the price tick (`MAX_PRICE_TICK_SIG_EXPIRY` is currently configured as one minute) but also has a `sig.expiration >= state.prices.fundingTime + ONE_MINUTE_NANOS` check to prevent funding ticks from coming in at quick succession. While this ensures that the Market Data's signature/timestamp is not more than one minute older than the funding price's `timestamp` and it is at least one minute after the previous funding time, there is no association and tracking of this `timestamp` with the funding price to enforce a maximum staleness threshold against the next funding price update.

Impact: High, because stale mark and funding prices may lead to unexpected behavior due to incorrect valuations leading to liquidation issues, arbitrage opportunities and ineffective risk management, among other things. Currently, the last updated mark and funding prices are used without any staleness checks.

Likelihood: Low, because it is assumed that the Gravity Backend has a robust mechanism to interface with multiple/decentralized Oracle and Market Data sources so that mark and funding prices are always updated at the expected cadence.

Recommendation: Consider tracking the `timestamp` with mark and funding prices to allow enforcing a maximum threshold when those prices are used for trades.

Gravity: Acknowledged. We have decided to implement this check only in our Backend.

Cantina Managed: Acknowledged.

5.3.17 Privileged roles and actions across various contracts lead to centralization risks for users

Severity: Medium Risk

Context: Global scope (*in particular all exchange contracts, bridging contracts, backend, risk Engine...*)

Description: The Gravity Exchange contracts are deployed on a private ZK chain and their functions are expected to be triggered appropriately by the trusted Gravity UI/Backend after any required authorizations and risk tests are successful. The centralized points of failure include:

1. Backend which has exclusive access to the Exchange contracts deployed on the private ZK chain.
2. Risk Engine which implements critical checks some of which are not currently mirrored in the contracts .
3. Contract functions that do no have access controls enforced by signature verification or other means.
4. The use of `timestamp` in `_setSequence()` which can be manipulated.

5. Privileged roles of Config, Oracle, Market Data and Recovery addresses.
6. Blockscholes as the Oracle for mark, settlement and interest data.
7. Upgradeable contracts.
8. `takerFeePercentageCap`/`makerFeePercentageCap`/`feeCharged` (`feePerLegs`) is simply chosen by the RE/backend and signed by the `INSURANCE_FUND_SUB_ACCOUNT_ID`. The whole order could be taken as a fee by the insurance fund.

The `GRVTBridgeProxy` is going to be deployed on Ethereum L1 allowing the users to bridge/deposit tokens to the private ZK chain. The contract includes a few centralized points of failure:

1. Contract function to set new `BridgeHub` inside the `GRVTBridgeProxy` which would disable claiming failed deposits.
2. `merkleProof` for claiming failed deposits through `GRVTBridgeProxy` can only be produced by the Gravity team.
3. The `deposit` of certain tokens can be disabled by withholding the deposit approval signature or removing the token from the `allowedTokens` mapping.
4. Anyone getting a hold of `baseToken` (`GRVTBaseToken`) is able to call `BridgeHub.requestL2TransactionDirect` function that allows to send a transaction from L1 that interacts with any contract deployed on L2. This impact is critical as it allows calling functions on the exchange contract, making it possible to deposit tokens, perform malicious trades, etc...

Impact: High, because exploiting any of these centralized points of failure by trusted insiders/components can cause significant harm to users and protocol.

Likelihood: Low, assuming the centralized points of failure are adequately mitigated using security measures that are outside the scope of this evaluation.

Recommendation: Consider:

1. Ensuring that access to the private ZK chain node APIs is secured appropriately.
2. Documenting all the privileged roles and actions for protocol user awareness.
3. Enforcing role-based access control where different privileged roles control different protocol aspects. and are backed by different keys to follow separation-of-privileges security design principle.
4. Enforcing reasonable thresholds and checks wherever possible.
5. Putting privileged actions affecting critical protocol semantics behind timelocks so that users can decide to exit/engage.
6. Following the strictest opsec guidelines for privileged keys e.g. use of reasonable multisig and hardware wallets.

Gravity: Acknowledged.

Cantina Managed: Acknowledged.

5.3.18 Users can acquire `baseToken` of private ZK chain through refunds allowing them to maliciously call exchange contract functions

Severity: Medium Risk

Context: GRVTBridgeProxy.sol#L225

Description: `GRVTBridgeProxy.deposit` function is used to deposit/bridge tokens from L1 (Ethereum) to Gravity's private ZK Chain. The zkSync architecture allows the gas token on the private zk chain to be represented by an ERC20 defined by the Gravity team on L1. This token is referred to as `baseToken` in the `GRVTBridgeProxy` contract. Exchange contracts were designed to be only callable by Gravity's backend infrastructure and the way of authorizing this is by disallowing users to get hold of this `baseToken`, which serves as the gas token on the private L2.

By observing the deposit function, the refund recipient address is controlled by the user because `_l1Sender` is the address that is providing the `l1Token` to bridge, e.g. USDT.

```
txHash = bridgeHub.requestL2TransactionTwoBridges(
  L2TransactionRequestTwoBridgesOuter({
    chainId: chainID,
    mintValue: baseCost,
    l2Value: 0,
    l2GasLimit: L2_GAS_LIMIT_DEPOSIT,
    l2GasPerPubdataByteLimit: REQUIRED_L2_GAS_PRICE_PER_PUBDATA,
    refundRecipient: address(_l1Sender), // <<<
    secondBridgeAddress: sharedBridge,
    secondBridgeValue: 0,
    secondBridgeCalldata: abi.encode(_l1Token, _amount, _l2Receiver)
  })
);
```

The issue is that in case the L1 -> L2 call is unsuccessful:

- `l1Token` can be claimed back by calling `claimFailedDeposit()`.
- `baseToken` is refunded to the `l1Sender` on L2.

Even if the transaction is successful, there could be `baseToken` refunds due to overestimating the transaction fees. For users to call the exchange contracts on L2 directly:

- They need to acquire gas tokens for L2.
- They need to have access to internal RPC only available to the Gravity team.

If the internal L2 RPC somehow gets compromised, an attacker can perform several malicious actions like deposit tokens, perform erroneous trades, etc...

The core of this issue is the user acquiring gas tokens for the private L2 chain, but it's important to highlight the impact of someone acquiring `GRVTBaseToken` on L1. Owning `GRVT` tokens would allow calling exchange contract functions on L2 through L1 by invoking `requestL2TransactionDirect` on the `BridgeHub` contract. This is a problem because `requestL2TransactionDirect` function allows to send a transaction from L1 that interacts with any contract deployed on L2.

Impact: High, because having direct access to the exchange contracts on L2 makes it possible to deposit/mint unlimited tokens, perform malicious trades, etc...

Likelihood: Low, because the L2 endpoint must be compromised to exploit the exchange contract. Only having gas tokens for the L2 chain is not sufficient.

Recommendation: Consider changing the `refundRecipient` to an address controlled by the Gravity team on L2.

Gravity: Fixed in PR 11.

Cantina Managed: Reviewed that PR 11 fixes this as recommended by hardcoding the `refundRecipient` to the owner of the `GRVTBridgeProxy` contract.

5.3.19 Insurance fund not subject to any margin or total USD value restriction may lead to bad debt for protocol

Severity: Medium Risk

Context: PR 162

Description: The PR 162 implemented a new liquidation logic in the `tradeDeriv()` function. All the liquidations can now only be initiated by the `INSURANCE_FUND_SUB_ACCOUNT_ID` where this subaccount is never subject to any margin or spot balance restriction after its trades/liquidations:

```
function _requireValidMargin(SubAccount storage sub, bool isLiquidation, bool beforeTrade) internal
↪  view {
  (uint64 liquidationSubID, bool liquidationSubConfigured) =
↪  _getUintConfig(ConfigID.INSURANCE_FUND_SUB_ACCOUNT_ID);

  // Insurance Fund can Trade when under MM, and in Negative Equity
  if (liquidationSubConfigured && sub.id == liquidationSubID) {
    return;
  }

  if (isLiquidation && beforeTrade) {
    require(!isAboveMaintenanceMargin(sub), "subaccount liquidated is above maintenance margin");
  } else {
    _requireNonNegativeUsdValue(sub);
  }
}
```

Basically the `INSURANCE_FUND_SUB_ACCOUNT_ID` subaccount can trade:

- When under Maintenance Margin.
- With a negative total USD balance.

The implementation of an insurance fund subaccount that does not have any margin or total USD value requirements and can continue liquidating accounts regardless of its own balance has multiple impacts:

- Unlimited debt risk: The insurance fund subaccount can accumulate infinite debt, as it is allowed to liquidate positions without requiring a positive balance.
- Market instability and reduced trust in the project: The ability to liquidate positions without collateral backing may detriment the market confidence, reducing participation and increasing volatility.

While liquidations that are executed on time typically should improve the margin of the insurance fund, especially due to the liquidation fee, during periods of extreme price volatility or when liquidations are delayed, the insurance fund may incur in losses instead. In these scenarios, the fund could be forced to cover more losses than anticipated, depleting its reserves or even driving it into a negative balance.

Impact: High, because if the insurance fund is not operated with enough funds bad debt will be created.

Likelihood: Low, because bad debt will only be created if:

- Liquidations are not executed on time.
- Extreme price fluctuations.
- The insurance fund does not have enough liquidity.

Recommendation: Implement a mechanism that ensures that the insurance fund is always healthy with enough margin and liquidity to cover for any potential liquidation needed. A margin below the maintenance margin on this subaccount should only be allowed temporary and should always be corrected as soon as possible.

Gravity: Fixed in PR 216.

Cantina Managed: Reviewed that PR 216 fixes this issue with the introduction of a Socialized Loss on withdrawals.

5.4 Low Risk

5.4.1 `LiquidationContract` not removing zero balance positions may lead to stuck funds on reaching block gas limit

Severity: Low Risk

Context: LiquidationContract.sol#L95-L105

Description: `LiquidationContract` implements the `liquidateOneOrder()` function. This function facilitates the liquidation process of a subaccount by validating conditions based on the type of liquidation (e.g., margin maintenance or auto-deleveraging). It adjusts the positions and balances between the initiator and the passive subaccount accordingly, ensures all trades are within permissible limits and handles the transfer of liquidation fees.

However, once a perpetual position balance is set to 0 it does not call the `remove()` function to remove it from the `sub.perps.keys` array. Therefore, perpetual positions are permanent and the `sub.perps.keys` array is expected to grow indefinitely. As the `_fundPerp()` function iterates through all the perpetual positions, the gas costs will keep increasing and, with it, the possibility of reaching block gas limit DoS'ing any interaction with the protocol and therefore blocking all the funds in the subaccount.

Impact: High, because funds could get stuck in the subaccount if block gas limit is reached when calling the `_fundPerp()` function.

Likelihood: Very Low, because this requires many null-balance perpetual positions left opened and the block gas limit for Gravity private L2 is really high at 2^{50}.

Recommendation: Consider updating the `liquidateOneOrder()` function to call the `remove()` function when the position balance is zero.

Gravity: Fixed in PR 162.

Cantina Managed: Verified that PR 162 fixes this issue. Trades and liquidations are now always performed within the `tradeDeriv()` function that always removes the positions if their balance is zero.

5.4.2 Potential DoS for reaching block gas limit due to excessive number of positions opened

Severity: Low Risk

Context: FundingAndSettlement.sol#L32-L34, FundingAndSettlement.sol#L57-L60

Description: The `FundingAndSettlement` contract implements two functions, `_fundPerp()` and `_settleOptionsOrFutures()`, which both iterate over all positions opened within a subaccount. These functions are designed to periodically update the balances of accounts based on funding rates and settlement prices:

```
function _fundPerp(SubAccount storage sub) internal {
  // ...
  uint len = keys.length;
  for (uint i; i < len; ++i) {
    bytes32 assetID = keys[i];
    // ...
  }
  // ...
}

function _settleOptionsOrFutures(SubAccount storage sub, PositionsMap storage positions) internal {
  // ...
  uint posLen = posKeys.length;
  for (uint i; i < posLen; ++i) {
    bytes32 assetID = posKeys[i];
    // ...
  }
  // ...
}
```

As the number of positions grows, the amount of computation required for these iterations increases proportionally.

In a scenario where a user has opened an excessively large number of positions, the gas required to execute these functions could become substantial. This could result in transactions reaching the block gas limit, thereby causing them to fail. If the number of positions is large enough, it could potentially cause a Denial of Service (DoS) by preventing the settlement or funding of positions entirely, as the contract would not be able to complete the operation within the gas limit.

Moreover, do also note, that in the `RiskCheck` contract, the `_getSubAccountUsdValue()` function, used by the `_requireNonNegativeUsdValue()` function, also iterates over all the different positions opened.

Impact: High, because it could lead to a Denial of Service, preventing the settlement and funding processes which at the same time would prevent any operation such as trading, liquidations, transfers etc. leaving the funds permanently stuck in the subaccount.

Likelihood: Very Low, because it requires a very large amount of positions opened to reach the block gas limit, which is set very high for Gravity L2.

Recommendation: Consider implementing a mechanism that limits the maximum number of positions a single user can open.

Gravity: This issue will not be fixed for 2 reasons:

1. Number of positions opened per subaccount is capped by the number of assets.
2. The contract will be run in a private L2, where we have the control over the block gas limit.

Cantina Managed: Acknowledged.

5.4.3 Orders may be incorrectly invalidated for fee cap checks even when fee charging is not configured

Severity: Low Risk

Context: TradeContract.sol#L197-L216

Description: `tradeDeriv()` checks whether the exchange configured order fee is within the fee cap signed by the user. However, this check is performed irrespective of where the exchange is also configured to charge fees.

Impact: Medium, because `tradeDeriv()` may revert due to potential invalidation of an order based on fee that may be charged (and exceed the user-signed cap) but when the system is configured to not charge fees.

Likelihood: Low, because this requires the exchange configurations of:

1. Fee to exceed the user-specified cap and...
2. `(uint64 feeSubID, bool isFeeCharged) = _getUintConfig(ConfigID.ADMIN_FEE_SUB_ACCOUNT_ID);)` where `isFeeCharged` is `false`.

Recommendation: Consider predicating this fee-check logic with a check on `isFeeCharged`.

Gravity: An order is considered invalid if fee in the order(not signed by user) is above fee cap. This doesn't depend on whether the fee is charged. The backend implements the same check and hence, acknowledged.

Cantina Managed: Acknowledged.

5.4.4 Attacker can squat a Victim's Gravity Account ID forcing them to register with a different address

Severity: Low Risk

Context: AccountContract.sol

Description: Attacker can choose an arbitrary user's Account ID by signing message with `accountID` that is the Victim's address. This will make Attacker the Signer Admin of that account.

Impact: Low, because Attacker can prevent Victim from getting their L1 address as account ID on Gravity L2.

Note: The project clarified that even if Attacker takes Victim's Account ID, the resulting account will get mapped to Attacker and will in no way impact Victim (ensured via UI & Backend). When Victim registers on the platform, they will be asked to create a new account.

Likelihood: Low, because Attacker cannot exploit Victim in any way besides squatting their potential L2 account ID.

Proof of Concept:

1. Attacker signs Account creation message with `accountID` as Victim address.
2. An account gets created using Victim address with Attacker as Admin signer for this account:

```
Account storage acc = state.accounts[accountID];
// ...
acc.signers[sig.signer] = AccountPermAdmin;
```

Recommendation: Consider implementing the below check in `createAccount` function:

```
require(sig.signer==accountID, "Invalid accountID");
```

Gravity: Fixed in PR 156.

Cantina Managed: Reviewed that PR 156 fixes this as recommended.

5.4.5 Attacker can DoS Victim's trades by overwriting their session key

Severity: Low Risk

Context: SubAccountContract.sol

Description: Attacker can overwrite user's `sessionKey` by simply calling `addSessionKey` function with Victim's `sessionKey`.

Impact: High, because all Victim trades will fail as Account will lack required trade permission. This become risky as Victim may want to close an underwater position but will be unable to do so.

Likelihood: Very Low, because Attacker will need to guess the `sessionKey` for a Victim using some other side-channel.

Proof of Concept:

1. SubAccount A (having necessary trade permission) added session key S1.
2. SubAccount B calls `addSessionKey` function with session key S1 and keyExpiry as 1. This will overwrite SubAccount A session key.
3. All trades for SubAccount A by session key S1 will fail since session key is now expired:

```
address subAccountSigner = order.signature.signer;
    Session storage session = state.sessions[subAccountSigner];
    if (session.expiry != 0) {
      require(session.expiry >= timestamp, ERR_SESSION_EXPIRED);
      subAccountSigner = session.subAccountSigner;
    }
    _requireSubAccountPermission(sub, subAccountSigner, SubAccountPermTrade);
```

Another way of doing this is by overwriting `sessionKey` with a signer who has no trade permission.

Recommendation: Instead of using a global variable for `state.sessions[sessionKey]` , consider making session key per subAccount so that one subAccount will not be able to impact other subAccount. For example: `sub.sessions[sessionKey]`.

Gravity: Fixed in PR 156 and PR 185.

Cantina Managed: Reviewed that PR 156 and PR 185 fixes this by the following changes:

1. Session keys can be set only once.
2. Session key check relaxation, i.e. trades are allowed if any of these is `true`:
    - Order's signer is in the session key map, and session hasn't expired, and the sessionKey's signer has trade permission.
    - Order's signer has trade permission.

5.4.6 Missing checks for maker order may lead to unexpected behavior

Severity: Low Risk

Context: TradeContract.sol#L47, DataStructure.sol#L330-L366

Description: When a taker order is matched with one or many maker orders, it is expected to have checks to ensure that those orders are indeed satisfying different properties specifically applicable to a maker order. However, maker orders are missing the below property checks:

1. `makerOrder.subAccountID` should not be the same as `trade.takerOrder.subAccountID` to prevent same-/cross-account trades.
2. `timeInForce` should not be `IMMEDIATE_OR_CANCEL` or `FILL_OR_KILL` because that is applicable only to taker order.
3. `isMarket` should be false because that is applicable only to taker order.
4. `_requireSubAccount(takerOrder.subAccountID).quoteCurrency` should be equal to `_requireSubAccount(makerOrder.subAccountID).quoteCurrency` since fee is assuming that both quote currency are same:

```
_requireSubAccount(feeSubID).spotBalances[quoteCurrency] += totalMakersFee + takerFee;
```

Impact: Medium, because incorrectly constructed maker orders may result in unexpected behavior.

Likelihood: Low, because Gravity Exchange anticipates enforcing these relevant checks on their Backend/UI.

Recommendation: Consider enforcing above listed missing checks. Ensuring mirroring of onchain-offchain checks is critical for transparency, which is one of project's state motivation for their hybrid architecture.

Gravity: Fixed in PR 173.

Cantina Managed: Reviewed that PR 173 fixes the issue as recommended.

5.4.7 Mismatched lengths of `matchedSize` and `legs` arrays will cause a revert

Severity: Low Risk

Context: TradeContract.sol#L50

Description: The length of the `MakerTradeMatch.matchedSize` array should match the length of the `MakerTradeMatch.makerOrder.legs` array. Otherwise, the `tradeDeriv` function will revert with an index-out-of-bounds error.

```
uint64[] calldata matchSizes = makerMatch.matchedSize; // <<<
Order calldata makerOrder = makerMatch.makerOrder;
uint makerLegsLen = makerOrder.legs.length;
for (uint legIdx; legIdx < makerLegsLen; ++legIdx) {
  uint64 size = matchSizes[legIdx]; // <<< Reverts if matchSizes.length < makerOrder.legs.length
  if (size == 0) {
    continue;
  }
```

Impact: Low, as it causes a revert with an index-out-of-bounds error and temporary DoS.

Likelihood: Low, as the length of the `matchedSize` array is checked by the backend.

Recommendation: Consider adding a check in the smart contract to ensure that the length of the `matchedSize` array matches the length of the `legs` array.

Gravity: Fixed in PR 174.

Cantina Managed: Reviewed that PR 174 fixes this as recommended.

5.4.8 Missing `postOnly` and `reduceOnly` checks may lead to unexpected behavior

Severity: Low Risk

Context: DataStructure.sol#L352-L363

Description: When a taker order is matched with one or many maker orders, it is expected to have checks to ensure that those orders are indeed satisfying different properties specifically applicable to taker/maker orders. However, orders are missing the below two property checks (unimplemented as commented):

1. `postOnly` should be True only for maker orders.
2. `reduceOnly` if True for an order must reduce the position size, or be cancelled.

Impact: Medium, because incorrectly constructed orders with these options may result in unexpected behavior.

Likelihood: Low, because Gravity Exchange anticipates enforcing these relevant checks if/when implemented on their Backend/UI.

Recommendation: Consider enforcing above listed missing checks. Ensuring mirroring of onchain-offchain checks is critical for transparency, which is one of project's state motivation for their hybrid architecture.

Gravity: Fixed in PR 178.

Cantina Managed: Reviewed that PR 178 fixes the issue as recommended.

5.4.9 Missing `IsOCO` and `ocoLimitPrice` checks may lead to unexpected behavior

Severity: Low Risk

Context: DataStructure.sol#L376-L380, TradeSig.sol#L43-L52

Description: An OCO (One-Cancels-the-Other) limit order is a type of conditional order that combines two limit orders, where if one order executes, the other is automatically canceled. This typically consists of two limit orders: one above and one below the current market price and is used to limit potential losses while also setting a target for profits. Gravity Exchange contracts have fields and comments indicating future full-support for OCO orders but the trade signature logic currently implements partial support for it.

However, OCO orders are missing the below two property checks (unimplemented as commented):

1. Missing `IsOCO` field to help determine if `ocoLimitPrice` should be considered or not.
2. Missing sanity check to ensure `ocoLimitPrice == 0` for now.

Impact: Medium, because incorrectly constructed orders with this option may result in unexpected behavior when full support is added.

Likelihood: Low, because Gravity Exchange anticipates enforcing these relevant checks if/when implemented on their Backend/UI.

Recommendation: Consider implementing above listed missing field and check. Ensuring mirroring of onchain-offchain checks is critical for transparency, which is one of project's state motivation for their hybrid architecture.

Gravity: Fixed in PR 188.

Cantina Managed: Reviewed that PR 188 fixes the issue by removing all considerations/comments of `ocoLimitPrice`.

5.4.10 No upper bound on signers allowed per account may cause DoS

Severity: Low Risk

Context: AccountContract.sol

Description: If a user adds thousands of signers and makes the Multisig threshold also equal to thousand then any further operation could cause DoS due to out of gas error.

Impact: Low, because it could cause potential self DoS on user account while looping through thousands of signers who signed for the user account operation.

Likelihood: Low, because it is not expected that users will normally add too many signers.

Proof of Concept:

1. User A adds thousands of signers for their account and sets the same number in Multisig threshold.
2. Any admin operation on account will require to check all provided signatures which might cause out of gas issue.

Recommendation: Consider adding an upper bound on number of signers allowed per account. The same restriction could also be placed on subAccounts.

Gravity: Will not fix. We already have this limit in our Backend, we prefer not to place this limit in the contract since we want to have the flexibility to configure this limit per account basis.

Cantina Managed: Acknowledged.

5.4.11 Missing support for `GOOD_TILL_TIME` orders may lead to unexpected behavior

Severity: Low Risk

Context: DataStructure.sol#L21

Description: Gravity Exchange plans to support `GOOD_TILL_TIME` (GTT) orders, which are orders that remain active in the market until a specified time unless filled or cancelled before then.

While the expiration time of an order signature may currently be used as a proxy for specifying the GTT time, there appears to be no explicit support for specifying and checking GTT time for the different orders/legs in the contracts. This may lead to unexpected behavior for such orders if accepted.

Impact: Medium, because incorrectly constructed orders with this option may result in unexpected behavior when full support is added.

Likelihood: Low, because Gravity Exchange anticipates enforcing relevant checks if/when implemented on their Backend/UI.

Recommendation: Consider implementing explicit support for `GOOD_TILL_TIME` orders and checks to prevent such orders until then. Ensuring mirroring of onchain-offchain checks is critical for transparency, which is the project's state motivation for their hybrid architecture.

Gravity: For GTT order, the expiry of that order is set in the `Order.Signature.expiration` field. In `tradeDeriv` function we reject any order whose signature expires, therefore guaranteeing that we execute the order per user's instruction.

Cantina Managed: Acknowledged that current implementation is per Gravity's specification.

5.4.12 Duplicate deposit requests may fail

Severity: Low Risk

Context: GRVTBridgeProxy.sol

Description: If user makes duplicate deposit signature request from the backend such that `(_l1Sender, _l2Receiver, _l1Token, _amount, _deadline)` is the same then logic will fail for the second request because `usedDepositHashes[msgHash]` will already be set to true.

```
bytes32 msgHash = keccak256(
  abi.encodePacked(
    abi.encodePacked("\x19\x01", DOMAIN_SEPARATOR),
    keccak256(abi.encode(DEPOSIT_APPROVAL_TYPEHASH, _l1Sender, _l2Receiver, _l1Token, _amount,
↪   _deadline))
  )
);

// TODO: we can consider using nouce here, this imposes a limit of 1 tx per second, but requires
// the signer service to read the nonce from the contract
require(!usedDepositHashes[msgHash], "grvtBP: deposit approval already used");
```

Impact: Low, because user will need to take reapproval for the deposit amount from backend.

Likelihood: Very Low, because it requires 2 deposit approvals with same deadline and amount.

Recommendation: Consider:

1. Using nonces which will make each request different.
2. Assigning different deadline for each deposit approval request in backend.

Gravity: We chose to use second precision timestamp over nonce to save gas. In practice, this will not affect user experience as it is almost impossible to deposit twice in the same second from Frontend.

Cantina Managed: Acknowledged as Accepted Risk.

5.4.13 Not removing recovery address while removing signer may be unexpected

Severity: Low Risk

Context: AccountContract.sol, WalletRecoveryContract.sol

Description: While removing the signer, the associated recovery address for the signer is not removed.

Impact: Medium, because signer would have trusted the previously added recovery address and so chances of deceit from that recovery address is minimal.

Likelihood: Low, because if signer is re-added then previously added recovery address will still hold good for that signer even when signer did not expect that.

Proof of Concept:

1. While removing a signer, there is no call in `removeAccountSigner()` to remove the associated recovery address of account.
2. This means that even after removing the signer, the associated recovery address of signer holds good and could behave maliciously if/when this signer is re-added back to account.

Recommendation: Consider removing the recovery address of signer if/when signer is removed.

Gravity: Will not fix. After convening with the team, we have deemed that by fixing this it will break a common UX flow below, especially when you're an institution and have multiple sub accounts and signers

With the fix

1. Sub acc A $\Rightarrow$ trader a has permission X.
2. Trader a is removed from sun acc a.
3. Head of trading adds trader to sub acc B.
4. Trader a loses all recovery wallets when this happens if we go with this new flow.
5. As you pointed out that the likelihood is Low since signer would have trusted the past added recovery address so chances of deceit from past added recovery address is very Low. Also when you are removed as a signer, you are stripped off from all permissions, so even if a malicious actor can obtain the recovery wallet, he wouldn't gain any access to the account.

Cantina Managed: Acknowledged as intended functionality.

5.4.14 Precision loss while calculating Withdrawal fees may cause platform to lose dust amount

Severity: Low Risk

Context: TransferContract.sol

Description: Platform will lose dust amount on Withdrawal fees due to precision loss while calculating fees.

Impact: Low, because loss to protocol will be dust amounts.

Likelihood: Medium, because withdrawals in future will include BTC and ETH (which have 9 decimal places) besides the current USDT.

Proof of Concept:

1. Withdrawal fees is calculated as:

```
uint64 tokenDec = _getBalanceDecimal(currency);
withdrawalFeeCharged = _getWithdrawalFee().div(spotMarkPrice).toInt64(tokenDec);
```

2. `_getWithdrawalFee().div(spotMarkPrice)` has 6 decimal places, so we might lose precision here if the withdrawn token decimal is more than 6.

Recommendation: Consider scaling `_getWithdrawalFee()` to 9 decimal places to avoid this issue.

Gravity: This is a broader class of issue -- division currently uses the decimal places of the BI a, which might be too small. We decided to fix by making div return the higher decimal place of the 2 inputs. See PR 193.

Cantina Managed: Reviewed that PR 193 fixes this issue.

5.4.15 `MaintenanceMarginConfig` may consider unset tiers leading to incorrect tier being chosen

Severity: Low Risk

Context: LiquidationContract.sol

Description: `MaintenanceMarginConfig` length will be `12` for a currency even when fewer configs exist. So the rest of the unset configs will have both size and ratio set as 0, which can lead to incorrect tier selection in extreme cases.

Impact: Low, because incorrect tier could get selected if tier config is set incorrectly.

Likelihood: Low, because tiers will be set by protocol in such a way that user trade will always be matched with a tier.

Proof of Concept:

1. Lets say `tier[5]` was the last tier set with config size say 100k USD:

```
MaintenanceMarginConfig[MAX_M_MARGIN_TIERS] memory configs;
for (uint i = lo; i <= hi; i++) {
    (bytes32 mmBytes32, bool found) = _getByte32Config2D(ConfigID(i), currencyConfig);
    if (!found) {
      break;
    }
```

2. Then remaining `tier[6] - tier[12]` will have 0 value since loop will break post tier 5.
3. Now if user trade was above 100k USD then it will check tier 6-12 (which were not even set) with no match and select default tier 1 instead of selecting tier 5.

Recommendation: Consider limiting `MaintenanceMarginConfig` array size to the number of configs actually found for that currency.

Gravity: Fixed in PR 177.

Cantina Managed: The maintenance margin logic has been significantly refactored to address these issues. Reviewed that PR 177 with the refactored maintenance margin logic resolves this issue.

5.4.16 Possible type casting overflow in `BIMath.toInt64()` function

Severity: Low Risk

Context: BIMath.sol#L79

Description: The `BIMath` library implements the function `toInt64()`:

```
function toInt64(BI memory a, uint decimals) internal pure returns (int64) {
  int256 c;
  if (a.dec == decimals) {
    c = a.val;
  } else if (a.dec > decimals) {
    c = a.val / int256(10) ** (a.dec - decimals);
  } else {
    c = a.val * int256(10) ** (decimals - a.dec);
  }
  return int64(uint64(uint(c)));
}
```

This function converts a `BIMath` struct into an `int64` number with the given decimals which are passed as parameter. To achieve that, a final type casting to `int64` is performed: `int64(uint64(uint(c)))`. However, if `c` is higher than `type(int64).max` a silent type casting overflow will occur. As the `BIMath.toInt64()` function is used to calculate the margins, funding payments, settlements, withdrawal fees... an overflow here, even if unlikely, would be very harmful for the protocol.

Impact: High, because an overflow in this function could affect the integrity of the whole exchange: Liquidations, funding payments, settlements, withdrawal fees etc...

Likelihood: Very low, because the protocol would have to be operating with very high values, > `type(int64).max`.

Recommendation: Consider adding a require check in the `BIMath.toInt64()` function that enforces that `c` is higher than `type(int64).max`. Moreover, a similar check should also be added to the `BIMath.toUint64()` function as this function also performs an unsafe type casting: `return uint64(uint256(c));`.

Gravity: Fixed in PR 192.

Cantina Managed: Reviewed that PR 192 fixes this issue by using OpenZeppelin `SafeCast`.

5.4.17 Configuration parameters of types `BYTE32` and `BYTE322D` can never be configured

Severity: Low Risk

Context: ConfigContract.sol#L264-L309, ConfigContract.sol#L211, ConfigContract.sol#L248, ConfigContract.sol#L649-L653

Description: Exchange configuration parameters are declared in `ConfigContract.sol` and allowed to be changed by `CONFIG_ADDRESS` by first scheduling a change behind a timelock rule via `scheduleConfig()` and then executing that change after timelock expiration via `setConfig()`. `_getLockDuration()` gets the timelock duration provided the configuration parameter has hardcoded timelock rules. Every config change is expected to be timelocked (comment: "Every config change is timelocked" indicates that may be the case) although this is currently missing (addressed by the issue "Critical exchange parameters are missing configuration timelock rules").

`_getLockDuration()` obtains the lock duration for a config "key" depending on its type and the rules declared. However, there are missing checks for `ConfigType.BYTE32` and `ConfigType.BYTE322D`.

Impact: Low, because this will cause `_getLockDuration()` to revert and therefore prevent `scheduleConfig()` and `setConfig()` calls for configuration parameters of types `ConfigType.BYTE32` and `ConfigType.BYTE322D`, but they are expected to be set to reasonable values by default. For example, `MAINTENANCE_MARGIN_TIER_01-MAINTENANCE_MARGIN_TIER_12`, which are of type `ConfigType.BYTE322D` will revert while scheduling/setting their configurations. Maintenance margin tiers can never be updated from their default values.

Likelihood: Medium, because while this will always happen for configuration parameters of types `ConfigType.BYTE32` and `ConfigType.BYTE322D`, it is not clear if/when they will need to be changed.

Recommendation: Consider adding support for configuration parameters of types `ConfigType.BYTE32` and `ConfigType.BYTE322D` in `_getLockDuration()`.

Gravity: Fixed in PR 189.

Cantina Managed: Reviewed that PR 189 fixes this by removing types `ConfigType.BYTE32` and `ConfigType.BYTE322D`.

5.4.18 Default exchange addresses are hardcoded for development environment

Severity: Low Risk

Context: ConfigContract.sol#L671-L680

Description: The default addresses of Config, Oracle, Market Address and Recovery used in exchange configuration are currently hardcoded for development environment. This has to be changed to the production environment addresses.

Impact: High, because the developer environment addresses will very likely not be the correct ones for production and may not have the same security protections for their keys. This will cause Oracle and Market Address functionalities of `markPriceTick()` and `fundingPriceTick()` to not work as expected.

Likelihood: Very Low, because these default addresses have been acknowledged as currently being used for development and are expected to be reset to the production ones before deployment either during initialization or otherwise.

Recommendation: Change these addresses to the production version before deployment.

Gravity: We have a setup procedure where during the whole Layer 2 setup, we will replace the addresses with our production values with better security guarantee You rightly pointed out that the hardcoded addresses are used for development purpose.

Update: we have removed all hardcoded addresses and default config by introducing an `initializeConfig` function in PR 199.

Cantina Managed: Reviewed that PR 199 fixes the issue.

5.4.19 Accumulated bad debt will not allow all users to withdraw from the exchange

Severity: Low Risk

Context: TransferContract.sol#L112, L2SharedBridge.sol#L134

Description: Bad debt is an inherent risk in any perpetual decentralized exchange, primarily because it is impossible to fully shield against extreme price fluctuations. No matter how robust the mechanisms for margin calls, liquidations, or collateral management are, sudden and significant market movements can lead to situations where traders' positions fall below the required maintenance margin faster than they can be liquidated. In such scenarios, the losses incurred by the trader exceed the collateral posted, resulting in bad debt that the protocol must cover. Therefore, while risk management strategies can mitigate the likelihood and impact of bad debt, they cannot eliminate it entirely, especially during periods of high volatility or rapid price changes.

Taking this into consideration, we can assume that the protocol bad debt will grow over time. On the other hand, every time a user deposits into the `GRVTExchange` `L2StandardERC20 tokens` are sent to the `GRVTExchange` and when these users withdraw they are burnt in the call to `IL2StandardToken(_l2Token).bridgeBurn(msg.sender, _amount);` where `msg.sender` will always be the `GRVTExchange` address.

This bad debt can allow some users to "withdraw" more value than they should have been able to. In this case, the `GRVTExchange` ends up covering the shortfall, which means the platform's overall reserves are diminished. This reduction in reserves can impact the ability of other users to withdraw their funds, as the `GRVTExchange` may no longer have `L2StandardERC20` tokens to cover all withdrawals, or which is the same, not enough `L2StandardERC20` tokens to burn, especially if the bad debt is significant and goes unaddressed.

Impact: Medium, because a user or a small number of users would not be able to withdraw their full spot balances from the `GRVTExchange`.

Likelihood: Very low, because bad debt should be significant for this to occur. Also to cause any impact to any user, most of the `GRVTExchange` users would have to withdraw simultaneously.

Recommendation: Ensure that the protocol's fee account (`feeAccId`) is used as a safety buffer to cover any bad debt. This account, which accumulates fees from trades, can act as a reserve to mitigate losses from extreme market fluctuations or insolvencies. On the other hand, consider establishing an insurance fund funded by a portion

of trading fees to cover any shortfall resulting from liquidations or defaults. This fund should act as a secondary reserve to protect the protocol and its users against unexpected losses.

The bad debt accumulated by the `GRVTExchange` should be covered as a whole by the insurance fund and the protocol's fee account. This way, bad debt will never impact the users.

Gravity: We are aware of and acknowledge this issue, but the Socialized Loss mechanism will only be introduced in the next round.

Fix review update: Fixed in PR 216.

Cantina Managed: Reviewed that PR 216 introduces the Socialized Loss on Withdrawal (SLOW) mechanism, which addresses this issue as recommended.

5.4.20 Exchange withdrawals will always fail if `L1SharedBridge` contract is paused

Severity: Low Risk

Context: L1SharedBridge.sol#L548

Description: The `GRVTExchange.withdraw()` function allows the withdrawal of spot balances from a user's sub-account. This function implements the following flow:

1. `GRVTExchange.withdraw()`.
2. `l2SharedBridge.withdraw(recipient, getCurrencyERC20Address(currency), erc20WithdrawalAmount);`.
3. `IL2StandardToken(_l2Token).bridgeBurn(msg.sender, _amount);` where `msg.sender` will be always the `GRVTExchange` address.
4. The `WithdrawalFinalizer` contract is subscribed to the `BridgeBurn` event, so once a `BridgeBurn` event is emitted `finalizeWithdrawals()` is called.
5. `L1SharedBridge.finalizeWithdrawal()` is called which internally calls the `_finalizeWithdrawal()` internal function.
6. `_finalizeWithdrawal()` uses the `whenNotPaused` modifier.

```
function _finalizeWithdrawal(
    uint256 _chainId,
    uint256 _l2BatchNumber,
    uint256 _l2MessageIndex,
    uint16 _l2TxNumberInBatch,
    bytes calldata _message,
    bytes32[] calldata _merkleProof
) internal nonReentrant whenNotPaused returns (address l1Receiver, address l1Token, uint256 amount) {
    require(!isWithdrawalFinalized[_chainId][_l2BatchNumber][_l2MessageIndex], "Withdrawal is already
↪   finalized");
    isWithdrawalFinalized[_chainId][_l2BatchNumber][_l2MessageIndex] = true;

    // Handling special case for withdrawal from zkSync Era initiated before Shared Bridge.
    if (_isEraLegacyEthWithdrawal(_chainId, _l2BatchNumber)) {
        // Checks that the withdrawal wasn't finalized already.
        bool alreadyFinalized = IGetters(ERA_DIAMOND_PROXY).isEthWithdrawalFinalized(
            _l2BatchNumber,
            _l2MessageIndex
        );
        require(!alreadyFinalized, "Withdrawal is already finalized 2");
    }

    MessageParams memory messageParams = MessageParams({
        l2BatchNumber: _l2BatchNumber,
        l2MessageIndex: _l2MessageIndex,
        l2TxNumberInBatch: _l2TxNumberInBatch
    });
```

```
    (l1Receiver, l1Token, amount) = _checkWithdrawal(_chainId, messageParams, _message, _merkleProof);

    if (!hyperbridgingEnabled[_chainId]) {
        // Check that the chain has sufficient balance
        require(chainBalance[_chainId][l1Token] >= amount, "ShB not enough funds 2"); // not enough
↪   funds
        chainBalance[_chainId][l1Token] -= amount;
    }

    if (l1Token == ETH_TOKEN_ADDRESS) {
        bool callSuccess;
        // Low-level assembly call, to avoid any memory copying (save gas)
        assembly {
            callSuccess := call(gas(), l1Receiver, amount, 0, 0, 0, 0)
        }
        require(callSuccess, "ShB: withdraw failed");
    } else {
        // Withdraw funds
        IERC20(l1Token).safeTransfer(l1Receiver, amount);
    }
    emit WithdrawalFinalizedSharedBridge(_chainId, l1Receiver, l1Token, amount);
}
```

Given this function has the `whenNotPaused` modifier, if `L1SharedBridge` contract was paused, any withdrawal could be initiated by the backend in the L2 but then revert in the L1, when calling the `finalizeWithdrawal()` function.

Impact: High, because withdrawals could be potentially lost.

Likelihood: Very low, because this scenario would require:

1. The backend initiating withdrawals while the `L1SharedBridge` contract is paused.
2. The backend not checking the result of the `L1SharedBridge.finalizeWithdrawal()` calls.

Recommendation: Consider implementing a check in the backend system to verify that the `L1SharedBridge` is not paused before initiating any withdrawal process. On the other hand, ensure that the backend monitors the status of the `L1SharedBridge.finalizeWithdrawal()` calls. In case of failure, some mechanism should be implemented to retry the withdrawal in the future.

Gravity: Acknowledged. We will add alerts when this happens in `withdrawal-finaliser`. We will added such monitoring/alerts, and will work with ZkSync so that we can be notified in advance before this happen.

Cantina Managed: Acknowledged.

5.4.21 Allowing subAccounts to be created with USD as `quoteCurrency` will make them non-operational

Severity: Low Risk

Context: SubAccountContract.sol#L49

Description: While creating a subAccount user can specify its quote currency. If he specifies USD as the quote currency, considering that the current implementation only supports trading perpetuals, his account will be useless.

Consider the following:

1. PERPS can't be quote in USD.
2. `markPrice` for PERPS will be in the quote currency different from USD.
3. delta is also denominated in the quote currency different from USD.
4. fundingPayment then also must be denominated in the quote currency different from USD.
5. `spotBalances[quoteCurrency]` are deducted/added in the quote currency different from USD.

6. This leads to the fact that the `quoteCurrency` of subAccount can't be in USD if the person wants to hold PERPS.

Impact: Low, because the user can't do anything with his subAccount but there is no security impact.

Likelihood: Low, because the Gravity team intends to support only subAccounts quoted in USDT.

Recommendation: Consider disallowing USD as a quote currency for subAccounts and instead enforcing it to be USDT.

Gravity: Fixed in PR 166.

Cantina Managed: Reviewed that PR 166 fixes this as recommended.

5.4.22 `_getBalanceDecimal()` hardcoding decimals based on `enum Currency` may lead to unexpected precision

Severity: Low Risk

Context: BaseContract.sol#L153-L163, DataStructure.sol#L38-L45

Description: `_getBalanceDecimal()` determines the decimal precision for different currencies. However, this is based on the current `enum Currency` declaration where positions 1-3 (USD, USDC, USDT) are determined to have 6 decimals and all currencies beyond them (ETH, BTC for now) are determined to have 9 decimals.

Impact: Low, because this may lead to unexpected precision of 9 decimals for new currencies if they are added to the end of `enum Currency` during an upgrade.

Likelihood: Low, because the upgrade may also appropriately modify `_getBalanceDecimal()` to keep it consistent with the expected decimal precision for the newly added currencies.

Recommendation: Consider specifying decimals for each individual currency independent of their position in the `enum Currency` so that `_getBalanceDecimal()` does not hardcode positional assumptions and will necessarily require a modification upon upgrades adding new currencies.

Gravity: Fixed in PR 181.

Cantina Managed: Reviewed that PR 181 fixes the issue as recommended.

5.4.23 Short positions bear additional risk due to USDC/USDT depeging from USD

Severity: Low Risk

Context: OracleContract.sol#L135

Description: Perpetuals are meant to be quoted in USDC/USDT so there is always the exchange rate risk associated with holding such positions. The risk primarily comes from the fact that USDC/USDT can depeg from USD. Observing the historical data for `USDC` and `USDT` we can see that the price fluctuates around 1 USD.

It also can be observed that there is a much higher likelihood of the price breaking below 1 USD than above 1 USD. There were a few major depeg events in the past, primarily due to concerns around backing reserves.

Although the risk is relatively low, depeging below 1 USD can negatively affect short positions.

Take the example:

- A user opens a short BTC PERPS position for -3 BTC.
- The price of BTC on the open market is $50,000.
- His account is healthy holding -3 BTC PERPS with 160k USDT SPOT position.
- There is a depeg event and the USDT price gets to $0.9.
- His account is subject to liquidation as his spot USDT balance is worth 160k USDT * 0.9 = 144k USD while the negative balance of 3 BTC is still worth 150k USD.

Impact: Medium, because depegging of USDC/USDT can have negative implications for short positions which bear this additional risk on top of the usual market risk.

Likelihood: Low/medium, because as shown by the historical data, major depeg events can happen.

Recommendation: Consider documenting the risk associated with USDC/USDT depeging and make sure users are aware of it.

Gravity: Whether greater funding payment is to help or make things worse for short position holders depends on the sign of the funding rate. To add on, USD-pegged stablecoins can depeg in 2 ways. Although they mostly depeg to a lower value than USD, we've seen prices a few percentage points higher than 1 USD a few times between 2017 and 2019. In case of a upward depegging, though unlikely, the risk you mentioned is reversed.

Based on our discussion, I think this is an inherent risk in all stablecoin-quoted derivative contracts. The best we can do here is to document this risk as you recommended.

Cantina Managed: Acknowledged.

5.4.24 `fundingPriceTick()` may revert instead of enforcing the expected clamping requirement for Funding Rate

Severity: Low Risk

Context: OracleContract.sol#L117-L128

Description: When `fundingPriceTick()` updates the funding prices for perpetuals, the new funding prices are expected to be within the configured rates as determined by `ConfigID.FUNDING_RATE_HIGH` and `ConfigID.FUNDING_RATE_LOW`. The expected clamping requirement is documented as "Funding Rate = Max(Min(Funding Rate, 5%), -5%)".

However, while the implemented check `newFunding >= fundingLow && newFunding <= fundingHigh` ensures the new price to be within the configured rates by reverting, it does not enforce the formula of Max(Min(Funding Rate, 5%), -5%) to clamp when exceeded. For example, if the new price is 6%, the current implementation will revert (exceeds the high threshold), while the clamping requirement would have bound this to 5% with Max(Min(6%, 5%), -5%).

Impact: Low, because funding price updates can be set to be within bounds to prevent such reverts.

Likelihood: Low, because Market Data is assumed to keep funding prices within bounds.

Recommendation: Consider clamping instead of reverting if that is the expected behavior. If not, fix the documentation.

Gravity: We will update the comment, the expected behavior is to reject funding tick that doesn't confirm pass this validation. Fixed in PR 172.

Cantina Managed: Acknowledged. Reviewed that PR 172 updated comment.

5.4.25 Exchange allowing positions of any small size may cause overhead and errors

Severity: Low Risk

Context: TradeContract.sol#L16

Description: The current GRVT exchange implementation allows users to open positions of any size. Allowing positions of any size, including extremely small ones like a BTC/USDT position with a size worth 0.0000001$, presents several concerns for the GRVT exchange.

Firstly, such small positions are often not economically viable for traders due to fees that exceed the potential profits, making them impractical and contributing to increased operational overhead for the exchange/backend.

Secondly, these small positions could cause precision errors during calculations, especially when handling fees, funding rates, or settlement calculations, resulting in unfair user balances or rounding errors.

Impact: Low, because the value of these positions is minimal.

Likelihood: Medium, because the costs for a malicious user opening many of these small positions is zero. It just requires the user to sign multiple trades.

Recommendation: Consider implementing a minimum position size threshold which balances market access with economic viability. This threshold should be set based on the smallest meaningful trade size relative to the assets' value and trading fees, ensuring that positions below this size are rejected.

Gravity: Will not fix. These limitations are already implemented at the backend level where we can effectively enforce the minimum size per account/trading account.

Cantina Managed: Acknowledged.

5.4.26 Missing call to `_disableInitializers()` in proxy implementations may cause unexpected behavior

Severity: Low Risk

Context: GRVTExchange.sol, GRVTBridgeProxy.sol, OpenZeppelin Documentation

Description: `GRVTExchange` and `GRVTBridgeProxy` are proxied implementations that use OpenZeppelin Initializable but do not have a constructor that calls `_disableInitializers()` as recommended by OpenZeppelin documentation. This leaves the implementation contract uninitialized.

Impact: High, if the uninitialized implementation contract can be taken over by an attacker to make a delegate call that self-destructs to brick the proxy contract.

Likelihood: Very Low, because there are no further delegate calls in the current implementations. However, future upgrades may make this attack surface exploitable.

Recommendation: Consider calling `_disableInitializers()` from OpenZeppelin's Initializable in all implementation contract constructors as done in `GRVTBaseToken`.

Gravity: Fixed in PR 171 and PR 13.

Cantina Managed: Reviewed that PR 171 and PR 13 fix this as recommended.

5.4.27 Single-step ownership change is risky

Severity: Low Risk

Context: GRVTBridgeProxy.sol#L24

Description: `GRVTBridgeProxy` inherits `OwnableUpgradeable` which performs a single-step ownership change in `transferOwnership()` and also supports `renounceOwnership()`. There are several critical `onlyOwner` functions supported: `approveBaseToken()`, `addAllowedToken()`, `removeAllowedToken()`, `setBridgeHub()`, `setDepositApprover()` and `mintBaseTokenL2()`, which need to be invoked only by the authorized owner. The owner is documented to be a Gravity-controlled EOA.

Impact: High, because using an incorrect address while transferring ownership or renouncing ownership will make all these functions inaccessible to the expected owner.

Likelihood: Very low, if it is assumed that the Gravity EOA owner will never make a mistake while transferring ownership.

Recommendation: Consider implementing a two-step ownership transfer process by using `Ownable2StepUpgradeable` instead of `OwnableUpgradeable`.

Gravity: Fixed in PR 12.

Cantina Managed: Reviewed that PR 12 fixes this as recommended.

5.4.28 Hardcoded value for `l2GasPerPubdataByteLimit` while bridging tokens through `GRVTBridgeProxy` may lead to reverts

Severity: Low Risk

Context: GRVTBridgeProxy.sol#L224

Description: One of the parameters passed to the `BridgeHub.requestL2TransactionTwoBridges` function is the `l2GasPerPubdataByteLimit` which is hardcoded to a constant `REQUIRED_L2_GAS_PRICE_PER_PUBDATA` equal to 800. It's used to derive the price for L2 gas in the base token to be paid for the transaction.

A comment inside `Mailbox.sol` contract indicates that this value could change and shouldn't be hardcoded.

Impact: Low, because based on changes in the Mailbox contract there could be reverts while trying to bridge tokens from L1 → L2 through the `GRVTBridgeProxy` contract.

Likelihood: Low, because the current state of the `Mailbox` contract is compatible with the hardcoded value.

Recommendation: Consider having this value configurable by the owner inside the `GRVTBridgeProxy` contract.

Gravity: Fixed in PR 17.

Cantina Managed: Reviewed that PR 17 fixes this as recommended.

5.4.29 Config settings and timelock rules are immutable

Severity: Low Risk

Context: PR 168

Description: The PR 168 implemented the following functionality:

- Removed unused configs in the contract.
- Renamed some `configIDs`.
- Replaced ConfigID `adminLiquididationSubAccountID` with `insuranceFundSubAccountID` since they are the same and only one is needed.
- Added timelocks just for margin/fee/risk configs.

Considering the `ConfigSetting` struct:

```
struct ConfigSetting {
  // the type of the config. UNSPECIFIED if this config setting is not set
  ConfigType typ;
  // the timelock rules for this config
  ConfigTimelockRule[] rules;
  // the schedules where we can change this config.
  mapping(bytes32 => ConfigSchedule) schedules;
}
```

The ConfigSettings added which are implemented in the `_setDefaultConfigSettings()` function and are immutable are:

Key	Type	Rule
`ConfigID.SIMPLE_CROSS_MAINTENANCE_MARGIN_TIER_01`	`ConfigType.BYTE322D`	N/A
`ConfigID.SIMPLE_CROSS_MAINTENANCE_MARGIN_TIER_02`	`ConfigType.BYTE322D`	N/A
`ConfigID.SIMPLE_CROSS_MAINTENANCE_MARGIN_TIER_03`	`ConfigType.BYTE322D`	N/A

Key	Type	Rule
ConfigID.SIMPLE_CROSS_FUTURES_INITIAL_MARGIN	ConfigType.CENTIBEEP2D	Rule(int64(2 * ONE_WEEK_NANOS), 0, 0)
ConfigID.ADMIN_RECOVERY_ADDRESS	ConfigType.BOOL2D	N/A
ConfigID.ORACLE_ADDRESS	ConfigType.BOOL2D	N/A
ConfigID.CONFIG_ADDRESS	ConfigType.BOOL2D	N/A
ConfigID.MARKET_DATA_ADDRESS	ConfigType.BOOL2D	N/A
ConfigID.ERC20_ADDRESSES	ConfigType.ADDRESS2D	N/A
ConfigID.L2_SHARED_BRIDGE_ADDRESS	ConfigType.ADDRESS	N/A
ConfigID.ADMIN_FEE_SUB_ACCOUNT_ID	ConfigType.UINT	N/A
ConfigID.INSURANCE_FUND_SUB_ACCOUNT_ID	ConfigType.UINT	N/A
ConfigID.FUNDING_RATE_HIGH	ConfigType.CENTIBEEP2D	Rule(int64(2 * ONE_WEEK_NANOS), 0, 0)
ConfigID.FUNDING_RATE_LOW	ConfigType.CENTIBEEP2D	Rule(int64(2 * ONE_WEEK_NANOS), 0, 0)
ConfigID.FUTURES_MAKER_FEE_MINIMUM	ConfigType.CENTIBEEP2D	Rule(int64(2 * ONE_WEEK_NANOS), 0, 0)
ConfigID.FUTURES_TAKER_FEE_MINIMUM	ConfigType.CENTIBEEP2D	Rule(int64(2 * ONE_WEEK_NANOS), 0, 0)
ConfigID.OPTIONS_MAKER_FEE_MINIMUM	ConfigType.CENTIBEEP2D	Rule(int64(2 * ONE_WEEK_NANOS), 0, 0)
ConfigID.OPTIONS_TAKER_FEE_MINIMUM	ConfigType.CENTIBEEP2D	Rule(int64(2 * ONE_WEEK_NANOS), 0, 0)
ConfigID.WITHDRAWAL_FEE	ConfigType.UINT	Rule(int64(2 * ONE_WEEK_NANOS), 0, 0)
ConfigID.BRIDGING_PARTNER_ADDRESSES	ConfigType.BOOL2D	N/A

While it makes perfect sense to keep the type immutable, applying the same restriction to the timelock settings is less justifiable. Timelocks may require flexibility for future updates or adjustments, whereas the type should remain consistent to ensure the integrity of the configuration.

On the other hand, the `ConfigTimelockRule` also implements the fields `deltaPositive` and `deltaNegative` which could prove useful in the future:

```
struct ConfigTimelockRule {
  // Number of nanoseconds the config is locked for if the rule applies
  int64 lockDuration;
  // This only applies for Int Configs.
  // It expresses the maximum delta (in the positive direction) that the config value
  // can be changed by in order for this rule to apply
  uint64 deltaPositive;
  // This only applies for Int Configs.
  // It expresses the maximum delta (in the negative direction) that the config value
  // can be changed by in order for this rule to apply
  uint64 deltaNegative;
}
```

Impact: Very Low, because the only impact is that these values can never be modified.

Likelihood: High, because these values are hardcoded in the `_setDefaultConfigSettings()` function.

Recommendation: Consider adding a function that allows updating the current timelock rules. On the other hand, consider also implementing a function that allows adding new `ConfigTimelockRules` to other `ConfigIDs`.

Gravity: It is expected and a product decision that the timelock rules are immutable.

Cantina Managed: Acknowledged.

5.4.30 Lack of access control in multiple critical `GRVTExchange` functions may be exploited

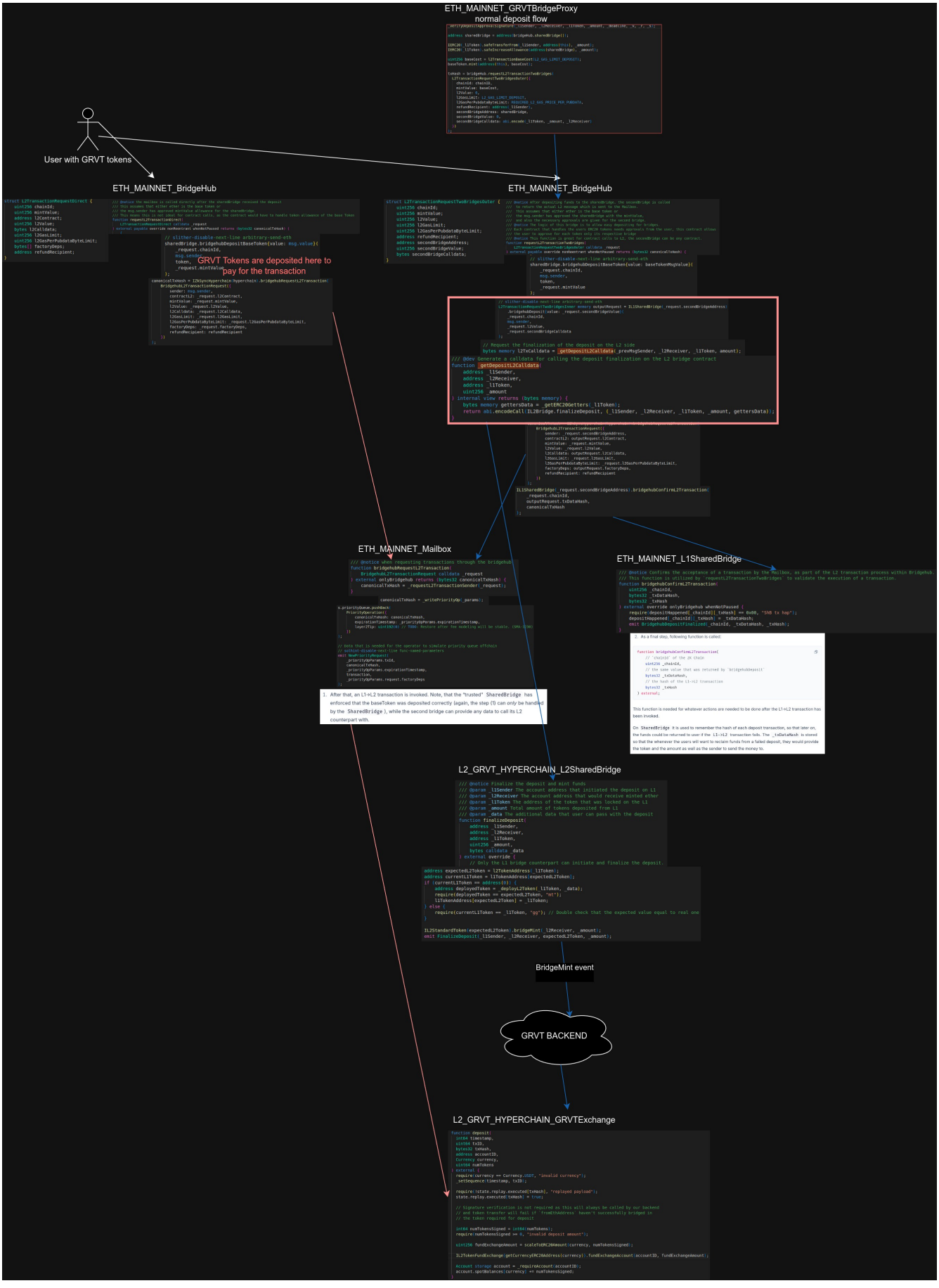
Severity: Low Risk

Context: GRVTExchange.sol

Description: The `GRVTExchange` contract implements multiple functions without any sort of access control mechanism. Some of these critical functions are:

- `deposit()`.
- `tradeDeriv()`.
- `addSessionKey()`.
- `removeSessionKey()`.

Although this contract will be deployed on the private GRVT Hyperchain, it is highly recommended to add access control to all critical functions. The reasoning for this is that even if GRVT Hyperchain is private, users that own GRVT tokens would be able to communicate directly with the GRVT Hyperchain from the Ethereum mainnet through the `BridgeHub.requestL2TransactionDirect()` function. The red arrow represents this flow:


ETH_MAINNET_GRVTBridgeProxy
normal deposit flow
User with GRVT tokens
ETH_MAINNET_BridgeHub
ETH_MAINNET_BridgeHub
GRVT Tokens are deposited here to pay for the transaction
ETH_MAINNET_Mailbox
ETH_MAINNET_L1SharedBridge
1. After that, an L1→L2 transaction is invoked. Note, that the "trusted" SharedBridge has enforced that the baseToken was deposited correctly (again, the step (3) can only be handled by the SharedBridge), while the second bridge can provide any data to call its L2 counterpart with.
L2_GRVT_HYPERCHAIN_L2SharedBridge
BridgeMint event
GRVT BACKEND
L2_GRVT_HYPERCHAIN_GRVTExchange

Impact: High, because the lack of access control would allow for trivial exploitation of the exchange.

Likelihood: Very low, because users would have to own GRVT tokens which should never be possible by design.

Recommendation: Consider implementing an additional layer of contract-level access control in all the critical functions mentioned.

Gravity: Fixed in PR 201.

Cantina Managed: Reviewed that PR 201 fixes this issue as recommended by adding access control via `CHAIN_SUBMITTER_ROLE` for all critical functions.

5.5 Gas Optimization

5.5.1 Using `abi.encodePacked` in a for loop cause huge memory expansion and increased gas costs

Severity: Gas Optimization

Context: OracleSig.sol#L14, TradeSig.sol#L53

Description: `hasOraclePrice` function encodes prices in a for loop to construct the typeHash for the signature. Incrementally encoding `pricesEncoded` with `abi.encodePacked` in a for loop causes huge memory expansion.

```
function hashOraclePrice(int64 timestamp, PriceEntry[] calldata prices) pure returns (bytes32) {
  bytes memory pricesEncoded;
  uint numValues = prices.length;
  for (uint i; i < numValues; ++i) {
    pricesEncoded = abi.encodePacked(
      pricesEncoded,
      keccak256(abi.encode(_ORACLE_VALUE_H, prices[i].assetID, prices[i].value))
    );
  }
  return keccak256(abi.encode(_ORACLE_H, keccak256(pricesEncoded), timestamp));
}
```

There is an instance of this same issue in the `hashOrderLegs` function in `TradeSig.sol`. See the proof of concept for more details.

Impact: Given a large number of prices/orderLegs, both functions can hit the block gas limit.

Likelihood: The likelihood of hitting the block gas limit is high in the case of order legs as each leg has more data and the number of legs in the trade can be high.

Proof of Concept:

```
function hashOraclePrice(int64 timestamp, PriceEntry[] memory prices) internal returns (bytes32) {
    bytes memory pricesEncoded;
    uint numValues = prices.length;
    uint256 gas = gasleft();
    for (uint i; i < numValues; ++i) {
        pricesEncoded = abi.encodePacked(
            pricesEncoded,
            keccak256(abi.encode(_ORACLE_VALUE_H, prices[i].assetID, prices[i].value))
        );
    }
    console2.log("Gas used for hashing", gas - gasleft());
    return keccak256(abi.encode(_ORACLE_H, keccak256(pricesEncoded), timestamp));
}

function hashOraclePriceAlternative(int64 timestamp, PriceEntry[] memory prices) internal returns
↪  (bytes32) {
    bytes memory pricesEncoded;
    bytes32[] memory hashedPricesElements = new bytes32[](prices.length);
    uint numValues = prices.length;
```

```
    uint256 gas = gasleft();
    for (uint i; i < numValues; ++i) {
      hashedPricesElements[i] = keccak256(abi.encode(_ORACLE_VALUE_H, prices[i].assetID,
↪   prices[i].value));
    }
    pricesEncoded = abi.encodePacked(hashedPricesElements);
    console2.log("Gas used for alternative hashing", gas - gasleft());
    return keccak256(abi.encode(_ORACLE_H, keccak256(pricesEncoded), timestamp));
}

struct PriceEntry {
    bytes32 assetID;
    int256 value;
}

function testGasConsumption() public {
    uint256 entries = 500;
    PriceEntry[] memory prices = new PriceEntry[](entries);
    for (uint i = 0; i < entries; i++) {
        prices[i] = PriceEntry(keccak256(abi.encodePacked(i)), int256(i));
    }

    bytes32 result = hashOraclePrice(100, prices);
    bytes32 result1 = hashOraclePriceAlternative(100, prices);
    assertEq(result, result1);
}
```

Running the `testGasConsumption` function compares gas costs of the original and the alternative implementation that produce the same hash. For 500 entries, the result is:

- Gas used with the current implementation: 42656822 ~ 42.6M.
- Gas used with the alternative implementation: 1556842 ~ 1.5M.

Recommendation: Use the alternative implementation laid out in the proof of concept.

Gravity: Fixed in PR 164.

Cantina Managed: Reviewed that PR 164 fixes this as recommended in the `OracleSig` and `TradeSig` contracts.

5.6 Informational

5.6.1 Approved deposits cannot be revoked if required

Severity: Informational

Context: GRVTBridgeProxy.sol

Description: Currently there is no way to stop a deposit that has been approved by deposit approver. Once deposit is completed, depositor can perform malicious activity on the L2 chain without restriction.

Impact: Platform will fail to prohibit a malicious user from making deposit on their platform. Also user can do trades on L2 as no restriction exist on contract level.

Likelihood: Low as platform mentioned that User account can be frozen from backend code.

Proof of Concept: 1. Deposit approver approves a transaction for a wallet which has no transaction. 2. Wallet receives stolen funds and since it has deposit approval, immediately deposits it. 2. Deposit approver realizes that transaction approved was malicious (wallet got illegal funds). 3. No way for protocol to prohibit this user trades and operations on L2 (via contract code). Although protocol mentioned that they have freezing functionality on backend that could be used to freeze such accounts.

Recommendation:

1. Add a functionality to disallow blacklisted msgHash (can add check in `_verifyDepositApprovalSignature` function of `GRVTBridgeProxy.sol`).
2. Add freezing functionality on contract side as well to improve transparency.

Gravity: From our legal team: we check proactively, not retrospectively. If the fund is tainted after it has been cleared before, we will proceed to freeze the account offchain. For this reason, we will not fix this issue we will not add freeze logic to the contract in this audit round, but are considering for the next.

Cantina Managed: Acknowledged. Product team mentioned that they have placed freezing logic on Backend and will add the same logic on contract in future. Since BE is not in scope for this audit, marking this issue as Acknowledged.

5.6.2 `LiquidationType` could be set to `UNDEFINED` to bypass all the subaccount margin checks

Severity: Informational

Context: LiquidationContract.sol#L61

Description: The `LiquidationContract` implements the function `liquidateOneOrder()` which handles the core logic for processing a liquidation order updating positions and spot balances based on the order details. This function implements the following checks:

- If `liquidationType == LiquidationType.LIQUIDATE`: `require(!isAboveMaintenanceMargin(passive))` ⇒ the subaccount margin should not be above the maintenance margin to be liquidated.
- If `liquidationType == LiquidationType.AUTO_DELEVERAGE`: `require(isAboveMaintenanceMargin(passive))` ⇒ the subaccount margin should be above the maintenance margin to be deleveraged.

However, the `LiquidationType` enum also includes the `UNDEFINED` type:

```
enum LiquidationType {
  UNDEFINED, // 0
  LIQUIDATE, // 1
  AUTO_DELEVERAGE // 2
}
```

If the liquidation is performed with an `UNDEFINED` `LiquidationType` all the margin checks would be totally skipped. Healthy subaccounts could be liquidated and unhealthy subaccounts could be deleveraged.

Impact: High, as healthy subaccounts could be liquidated and unhealthy subaccounts could be deleveraged. The protocol should guarantee that the margin requirements for liquidations and deleverages are always respected.

Likelihood: Very low as this logic is assumed to be implemented at the Risk Engine/backend and requires a malicious Risk Engine to be possible.

Recommendation: Consider removing the `UNDEFINED` type from the `LiquidationType` enum to always choose either a liquidation or a deleveraged action.

Gravity: Fixed in PR 162.

Cantina Managed: Verified that PR 162 fixes this issue as the liquidation logic was removed and implemented in the `tradeDeriv()` function where there are no `LiquidationTypes`.

5.6.3 Missing functions to remove Accounts and Subaccounts will prevent their removal

Severity: Informational

Context: AccountContract.sol#L15, SubAccountContract.sol#L19

Description: While Account and Subaccounts may be created, corresponding functions to explicitly remove them are missing. This will prevent deletion and purging of their state from the Exchange contracts, which will cause bookkeeping differences between onchain and offchain state when users want to close their Accounts/SubAccounts.

This currently only supports removing signers from Accounts and SubAccounts, where Accounts are designed to always retain one Account admin while SubAccounts may remove all their signers but will still be accessible by their Account admin(s).

Impact: Low, because it is assumed that Gravity Backend/UI will manage any deletion of Accounts/SubAccounts and prevent their use thereafter. Stale Accounts and SubAccounts in the Exchange contracts should not have any other side-effects.

Likelihood: Low, because Gravity Backend/UI is assumed to be a trusted component which should not originate/forward any trades to the contracts from/to Accounts/SubAccounts that have been closed.

Recommendation: Consider:

1. Providing explicit functions to mark Accounts and Subaccounts as removed/deleted and then purge all of their stored state.
2. Adding checks to prevent trades to deleted Accounts/SubAccounts.

Gravity: Will not fix. We don't support account deletion right now. We will consider adding this in a future version.

Cantina Managed: Acknowledged.

5.6.4 Typehash is incorrect for oracle signature

Severity: Informational

Context: OracleSig.sol#L7

Description: `PriceEntry.assetID` is of type `bytes32` while the type hash specifies it as `int256`. Moreover, typehash specifies the `Values` struct while the struct accepted as input to the `hashOraclePrice` function is `PriceEntry`. These differences can lead to generating the wrong signature off-chain.

Impact: Low, as the Gravity team will be the only one signing and sending transactions. In case of a wrong signature the transaction would revert, but it is expected this will never occur.

Likelihood: Low, as it's not expected that transactions that might revert are broadcast to the chain.

Recommendation: Consider using the `PriceEntry` struct instead of `Values` in the typehash:

```
- bytes32 constant _ORACLE_H = keccak256("Data(Values[] values,int256 timestamp)Values(int256
↪  sid,int256 v)");
- bytes32 constant _ORACLE_VALUE_H = keccak256("Values(int256 sid,int256 v)");

+ bytes32 constant _ORACLE_H = keccak256("Data(Values[] values,int256 timestamp)PriceEntry(bytes32
↪  assetID,int256 value)");
+ bytes32 constant _ORACLE_VALUE_H = keccak256("PriceEntry(bytes32 assetID,int256 value)");
```

Gravity: We decided not to fix this as the current signature format works and we need to involve our oracle partner to make this change.

Cantina Managed: Acknowledged.

5.6.5 Launching with unsupported and untested logic for anticipated features is risky

Severity: Informational

Context: Global scope (*in particular the exchange contracts*)

Description: Exchange contracts currently implement incomplete logic for supporting trading of futures, options and spots across different currencies, margin types etc..., which has resulted in many of the reported findings. The initial project launch only targets trading of perpetuals for BTC-USDT and ETH-USDT currency pairs. Future contract upgrades are expected to introduce the remaining features and configurations in stages.

It is expected that the Gravity Backend/UI will only allow launch trading parameters for now. However, it is risky to have unsupported and untested logic at launch because if such trades are accidentally allowed, they will result in unexpected behavior and state corruption.

Recommendation: Consider whether it is worth the effort/risk to refactor and remove unsupported and untested logic given the soon-approaching launch date.

Gravity: Fixed in PR 166, PR 167 and PR 168.

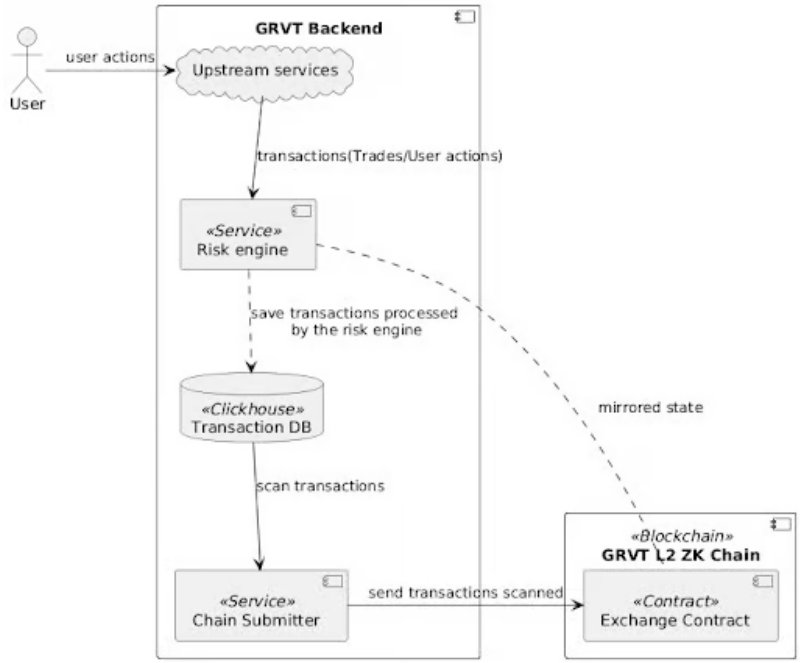
Cantina Managed: Reviewed that PR 166, PR 167 and PR 168 refactor to remove most of the unsupported logic.

5.6.6 Lack of an external security review of Backend components and Node infrastructure is risky

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: The scope of this review was limited to smart contracts. The interfacing Web application was part of an earlier review from a different security vendor. However, the Backend components (written in Go) and associated Node/hosting infrastructure as shown in the architecture below have not been part of any external security review. It was communicated that these have been internally reviewed for now.



Recommendation: While internal security reviews should definitely part of any mature SDLC, we recommend an external security review to complement that. This is especially important here given the critical role of the Backend in implementing Risk Engine checks and having exclusive access to the private ZK chain node deploying Exchange contracts.

Gravity: We will consider this internally, but there's no plan to do this before our Mainnet launch. Will seek the opinion of our CTO & CISO.

Cantina Managed: Acknowledged.

5.6.7 Configurations enabled for incompletely supported features affects readability

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: `ConfigContract.sol` defines default configurations in `_setDefaultConfigSettings()` for different aspects of the Exchange contracts. However, there are several configuration parameters enabled related to trading of futures, options and spot even though the contracts do not completely implement their supported features. These include configuration parameters such as `SM_FUTURES_MAINTENANCE_MARGIN`, `SM_OPTIONS_INITIAL_MARGIN_HIGH`, `PM_SPOT_MOVE`, `PM_INITIAL_MARGIN_FACTOR` and `FUTURE_MAKER_FEE_MINIMUM`.

This affects readability and maintainability of code. If such features are added in future then they may end up using previously set potentially stale/incorrect configuration values if those are not updated accordingly.

Recommendation: Consider removing all configurations related to unsupported features at launch and only including those affecting fully implemented and validated features. Include new configuration parameters incrementally as their features are added in future.

Gravity: Fixed in PR 168.

Cantina Managed: Reviewed that PR 168 fixes the issue as recommended.

5.6.8 Unsafe downcasting is risky

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: There are several expressions across the codebase where downcasting in performed on variables without any checks in place for their values. Downcasting in Solidity does not automatically revert on any overflow that happens during the operation.

Some examples include `int64(leg.size)`, `int64(uint64(uint256(v)))`, `int64(prices[i].value)` and `int32(uint32(uint256(c.val))`.

While we have not identified any immediate risks from overflows resulting from this because of the assumptions in-place or inferred, this is not a best-practice and is risky if the variables ever exceed the assumed thresholds.

Recommendation: Consider using a safe downcasting library such as OpenZeppelin's SafeCast, which provides wrappers over Solidity's casting operators with added overflow checks.

Gravity: Fixed in PR 192.

Cantina Managed: Reviewed that PR 192 fixes the issue as recommended by using OpenZeppelin's SafeCast.

5.6.9 Recovering from any mismatch of mirrored state may affect user-experience

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: Gravity has a hybrid exchange architecture where the Backend Risk Engine's state is mirrored by a smart contract running in a private Validium L2 ZK Chain. To guarantee consistent mirrored states in the offchain Risk Engine and the onchain Exchange contract, they should have identical logic. Any discrepancies may result in the deviation of the offchain-vs-onchain state. This is expected to be detected via their monitoring agents upon which the recovery process will need to include the following steps:

1. Identify the reason for the mismatched state.
2. Stop sending transactions from the Backend to the Exchange contract.
3. Fix the faulty logic in the Backend or Exchange contract that led to mismatched state.

4. If the Exchange contract was fixed then its implementation needs to be reviewed and upgraded.
5. Backend will forward all the pending transactions to the upgraded Exchange contract.

Note that the Backend and monitoring/recovery components were out-of-scope for this review.

This process of recovery may affect user-experience related to deposits/withdrawals of their funds.

Recommendation: Consider:

1. Documenting the mirroring/monitoring risks and its recovery process for users.
2. Validating the monitoring and recovery flows for incident response preparedness.

Gravity: Acknowledged. Our plan to deal with this state divergence is:

- Our backend is always the source of truth.
- If the contract state diverged.
- Pause the exchange (by shutting down the component the submitting new transactions on chain).
- Identify the fix on the contract, carry out an upgrade.
- Rollback the L2 chain to the last good state. Let say 1 block before this happen.
- Send the transactions from the last good state onwards.

Fix review update: PR 206 adds `AssertionContract` to assert where onchain state is as expected compared to the offchain chain.

Cantina Managed: Acknowledged. Reviewed that PR 206 helps address this issue as well.

5.6.10 The standard `__Ownable_init` library function can be used in `GRVTBridgeProxy.initialize()` instead of replicating its logic

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: `GRVTBridgeProxy` inherits from `OwnableUpgradeable` and so has the below logic in its `initialize()`:

```
require(_owner != address(0), "ShB owner 0");
_transferOwnership(_owner);
```

However, `OwnableUpgradeable` already provides a `__Ownable_init` which implements the same check-and-transfer logic:

```
function __Ownable_init(address initialOwner) internal onlyInitializing {
    __Ownable_init_unchained(initialOwner);
}

function __Ownable_init_unchained(address initialOwner) internal onlyInitializing {
    if (initialOwner == address(0)) {
        revert OwnableInvalidOwner(address(0));
    }
    _transferOwnership(initialOwner);
}
```

Recommendation: Consider using the standard `__Ownable_init` library function instead of replacing its logic.

Gravity: This function is only available in openzeppelin v5, while we are using v4.9.5. I tried upgrading to v5 but there are quite a few breaking changes. Thus, we don't think the effort is worth it, and hence will not fix this style issue.

Cantina Managed: Acknowledged. V5 may have other relevant improvements and so may be worth considering at a later time.

5.6.11 Redundant existence check for Account Identifier

Severity: Informational

Context: AccountContract.sol#L51-L52, AccountContract.sol#L86-L87

Description: `setAccountMultiSigThreshold()` and `addAccountSigner()` perform an account existence check with `acc.id != address(0)` on the return value `Account storage acc = _requireAccount(accountID)`. However, `_requireAccount(accountID)` already performs that check internally (as shown below) making this additional return-value check redundant.

```
function _requireAccount(address accID) internal view returns (Account storage) {
    Account storage acc = state.accounts[accID];
    require(acc.id != address(0), "account does not exist");
    return acc;
}
```

Recommendation: Consider removing the redundant existence check for Account Identifier.

Gravity: Fixed in PR 165.

Cantina Managed: Reviewed that PR 165 fixes this as recommended.

5.6.12 Unused code constructs indicate missing/stale functionality and affect readability

Severity: Informational

Context: OracleContract.sol#L4, LiquidationContract.sol#L244-L248, Error.sol#L10-L11, Error.sol#L14, DataStructure.sol#L13-L17

Description: There are some unused code constructs across the codebase. These either indicate missing functionality which is yet to be implemented or stale functionality which can be removed.

Recommendation: Consider implementing the missing functionality or removing these stale code constructs for now.

Gravity: Fixed in PR 167, PR 183 and PR 184.

Cantina Managed: Reviewed that PR 167, PR 183 and PR 184 fix the issue as recommended by removing the unused code.

5.6.13 USDT enabling fee-on-transfer cannot be supported

Severity: Informational

Context: GRVTBridgeProxy.sol

Description: USDT has a provision for fee-on-transfer which is currently not enabled. If enabled in future then protocol cannot use USDT for deposit. The `sharedBridge` does not support these types of tokens (see the `shared-bridge-as-standard-erc20-bridge` section in the docs).

Recommendation: If USDT enables fee-on-transfer in future then deposit functionality should be immediately stopped.

Gravity: Acknowledged.

Cantina Managed: Acknowledged.

5.6.14 The sanity check on number of tokens to deposit/withdraw/transfer can be stricter

Severity: Informational

Context: TransferContract.sol#L43, TransferContract.sol#L84, TransferContract.sol#L170

Description: There is a sanity check to ensure that `numTokensSigned >= 0` during deposit/withdraw/transfer. However, given that `numTokensSigned` should never be zero for a meaningful transaction, the check could be made stricter by changing >= to >.

Recommendation: Consider changing the require statements as shown in the example below:

```
- require(numTokensSigned >= 0, "invalid withdrawal amount");
+ require(numTokensSigned > 0, "invalid withdrawal amount");
```

Gravity: Fixed in PR 170 and PR 195.

Cantina Managed: Reviewed that PR 170 and PR 195 fixes this as recommended.

5.6.15 Use of unlicensed smart contracts

Severity: Informational

Context: Global scope

Description: All the Gravity smart contracts are currently marked as `UNLICENSED`, as indicated by the SPDX license identifier at the top of the file:

```
// SPDX-License-Identifier: UNLICENSED
```

Using an unlicensed contract could result in legal uncertainties and conflicts regarding the usage, modification and distribution rights of the code. This may deter other developers from using or contributing to the project or even lead to legal issues in the future.

Impact: Low as it is not a security concern.

Likelihood: High as most of the contracts are not using any license.

Recommendation: It is recommended to choose and apply an appropriate open-source license to the smart contracts. Some popular options for blockchain and smart contract projects include:

1. MIT License: A permissive license that allows for reuse with minimal restrictions.
2. GNU General Public License (GPL): A copyleft license that ensures derivative works are also open-source.
3. Apache License 2.0: A permissive license that provides an express grant of patent rights from contributors to users.

Gravity: Our choice to publish the contract code without open sourcing it, i.e. using an UNLICENSED license, is an intentional one. This is mostly because:

- Our contracts are very specialized to our business => open sourcing it is of little value to the community.
- Publishing the code alone is sufficient for transparency purpose and we always have an option to open source the code in the future.

Cantina Managed: Acknowledged.