SECURITY AUDIT REPORT

for

Levana

Prepared By: Xiaomi Huang

PeckShield

May 29, 2023

Document Properties

Client	Levana
Title	Security Audit Report
Target	Levana
Version	1.0
Author	Daisy Cao
Auditors	Daisy Cao, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	May 29, 2023	Daisy Cao	Final Release
1.0-rc	May 23, 2023	Daisy Cao	Release Candidate #1

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Xiaomi Huang
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the Levana protocol, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About Levana

Levana is a leveraged spot-price well-funded collateral-settled perpetual swaps protocol. It addresses the fundamental problems by introducing a novel way to delineate risks between market participants and offering an incentive structure with risk premium to the liquidity providers who take on the spot market illiquidity risk. In addition, the absence of native price discovery in spot-price settled perpetuals is addressed by introducing a capped artificial slippage mechanism. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Levana

Item	Description
Name	Levana
Website	https://levana.finance
Type	CosmWasm
Language	Rust
Audit Method	Whitebox
Latest Audit Report	May 29, 2023

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit.

- https://github.com/Levana-Protocol/levana-perps (a08a140a)

And here is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/Levana-Protocol/levana-perps (09c6ce9)

1.2 About PeckShield

PeckShield Inc. [9] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

Impact \ **Likelihood**	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [8]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact, and can be accordingly classified into four categories, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [7], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logics	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the `Levana` implementation. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	1	■
Medium	2	■ ■
Low	2	■ ■
Informational	1	■
Total	6	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 high-severity vulnerability, 2 medium-severity vulnerabilities, 2 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	High	Potential Limit Order Mis-Triggering From Lagging Cranks	Business Logic	Resolved
PVE-002	Medium	Revised Handling of SendFrom Message in liquidity_token	Business Logic	Resolved
PVE-003	Low	Disallowed Limit Order Placement at Stale State	Business Logic	Resolved
PVE-004	Medium	Trust Issue of Admin Keys	Security Features	Mitigated
PVE-005	Low	Redundant State/Code Removal	Business Logic	Resolved
PVE-006	Informational	Revisited Crank Fee Collection in Limit Order	Business Logic	Resolved

Beside the identified issues, we emphasize that for any user-facing applications and services, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 Detailed Results

3.1 Potential Limit Order Mis-Triggering From Lagging Cranks

- ID: PVE-001
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: order.rs
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

In Levana protocol, cranking works by iterating through all price updates and performing operations for each price update point. One specific operation involves the checking of limit orders with a limit price that is reached by the cranking price. If a limit order is triggered, a new position will be opened at the current price. However, since the position is opened at the current price, there is a possibility of opening a position at an undesirable or worse price than the limit price.

To elaborate, we show below the related code snippet of the limit_order_triggered_order() function. The limit order is triggered if the cranking price is lower than the limit price (line 6). However, the postion is opened with the current price. For example, in the case of a long position, if the limit price is lower than the current price, the trader will get a position at a worse price. This situation may lead to opening a position at an improper price.

```
1     pub(crate) fn limit_order_triggered_order(
2         &self,
3         storage: &dyn Storage,
4         price: Price,
5     ) -> Result<Option<OrderId>> {
6         let order = LIMIT_ORDERS_BY_PRICE_LONG
7             .prefix_range(
8                 storage,
9                 Some(PrefixBound::inclusive(PriceKey::from(price))),
10                None,
11                Order::Ascending,
12            )
```

```
                .next();

        let order = match order {
            Some(_) => order,
            None => LIMIT_ORDERS_BY_PRICE_SHORT
                .prefix_range(
                    storage,
                    None,
                    Some(PrefixBound::inclusive(PriceKey::from(price))),
                    Order::Descending,
                )
                .next(),
        };

        match order {
            None => Ok(None),
            Some(res) => {
                let ((_, order_id), ()) = res?;
                Ok(Some(order_id))
            }
        }
    }
```

Listing 3.1: `market/src/state/order::limit_order_triggered_order()`

Recommendation Revise the above function to make use of the appropriate price to trigger the limit orders.

Status This issue has been fixed in the following commit: `9521aca`

3.2 Revised Handling of SendFrom Message in liquidity_token

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `market.rs`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The `Levana` protocol has a core `liquidity` contract which enforces the logic to ensure the execution must come from the `liquidity_tokens proxy` contract. With that, the function `market_execute_liquidity_token()` makes a special handling of the `LiquidityTokenExecuteMsg::Send` message. The goal here is the messages to the destination contract come from the proxy rather than the market. While reviewing their logic, we notice that the current handling may be expanded to cover another `LiquidityTokenExecuteMsg::SendFrom` message.

In the following, we show the related `market_execute_liquidity_token()` function that handles the `LiquidityTokenExecuteMsg::Send` message. However, it comes to our attention that the similar handling should be applied to another message `LiquidityTokenExecuteMsg::SendFrom`. Otherwise, the `SendFrom` message may be sent from the market contract, not the proxy.

```
    pub(crate) fn market_execute_liquidity_token(
        &self,
        ctx: &mut StateContext,
        sender: Addr,
        msg: LiquidityTokenExecuteMsg,
    ) -> Result<()> {
        let (msg, send) = match msg {
            // Send needs special handling to ensure the messages to the destination contract come from the proxy, not the market.
            LiquidityTokenExecuteMsg::Send {
                contract,
                amount,
                msg,
            } => {
                let send = ReceiverExecuteMsg::Receive(Cw20ReceiveMsg {
                    sender: sender.clone().into(),
                    amount,
                    msg,
                });
                let msg = LiquidityTokenExecuteMsg::Transfer {
                    recipient: contract.clone(),
                    amount,
                };
                (msg, Some((contract, send)))
            }
            msg => (msg, None),
        };
        ctx.response.add_execute_submessage_oneshot(
            self.market_addr(ctx.storage)?,
            &MarketExecuteMsg::LiquidityTokenProxy {
                sender: sender.into(),
                kind: get_kind(ctx.storage)?,
                msg,
            },
        )?;

        // We need to sequence this submessage after the previous one to ensure the
        // receiving contract has the expected balance.
        if let Some((contract, send)) = send {
            ctx.response
                .add_execute_submessage_oneshot(contract, &send)?;
        }

        Ok(())
    }
```

Listing 3.2: `liquidity_token/src/state/market::market_execute_liquidity_token()`

Recommendation Revise the above routine to similarly handle the `LiquidityTokenExecuteMsg::SendFrom` message.

Status This issue has been fixed in the following commit: `6196b42`

3.3 Disallowed Limit Order Placement at Stale State

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `order.rs`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The `Levana` protocol introduces the concept of staleness. If a price update has been delayed for too long, or any position has reached the point where liquidation margin cannot be guaranteed, the protocol becomes stale. A stale protocol should not allow positions to be opened, updated, or closed. To exit a stale period, the protocol will require new price updates and/or cranking tasks performed.

While reviewing their logic, we notice that the limit order placement should be disallowed when the protocol enters the stale state. If the protocol is in a stale state, the guarantee of being well-funded may not hold. Therefore, from a security standpoint, the staleness check should be performed during the limit order placement as well.

```
    pub(crate) fn limit_order_set_order(
        &self,
        ctx: &mut StateContext,
        owner: Addr,
        trigger_price: PriceBaseInQuote,
        collateral: NonZero<Collateral>,
        leverage: LeverageToBase,
        direction: DirectionToNotional,
        max_gains: MaxGainsInQuote,
        stop_loss_override: Option<PriceBaseInQuote>,
        take_profit_override: Option<PriceBaseInQuote>,
    ) -> Result<()> {
        let last_order_id = LAST_ORDER_ID
            .may_load(ctx.storage)?
            .unwrap_or_else(|| OrderId::new(0));
        let order_id = OrderId::new(last_order_id.u64() + 1);
        LAST_ORDER_ID.save(ctx.storage, &order_id)?;

        let order_fee = Collateral::try_from_number(
            collateral
                .into_number()
```

```
                .checked_mul(self.config.limit_order_fee.into_number())?,
        )?;
        let collateral = collateral.checked_sub(order_fee)?;
        let price = self.spot_price(ctx.storage, None)?;
        self.collect_limit_order_fee(ctx, order_id, order_fee, price)?;

        let crank_fee_usd = self.config.crank_fee_charged;
        let crank_fee = price.usd_to_collateral(crank_fee_usd);
        self.collect_crank_fee(ctx, TradeId::LimitOrder(order_id), crank_fee,
            crank_fee_usd)?;
        let collateral = collateral
            .checked_sub(crank_fee)
            .context("Insufficient funds to cover fees, failed on crank fee")?;

        let order = LimitOrder {
            order_id,
            owner: owner.clone(),
            trigger_price,
            collateral,
            leverage,
            direction,
            max_gains,
            stop_loss_override,
            take_profit_override,
        };

        self.limit_order_validate(ctx.storage, &order)?;

        LIMIT_ORDERS.save(ctx.storage, order_id, &order)?;

        let market_type = self.market_type(ctx.storage)?;
        match direction {
            DirectionToNotional::Long => LIMIT_ORDERS_BY_PRICE_LONG.save(
                ctx.storage,
                (trigger_price.into_price_key(market_type), order_id),
                &(),
            )?,
            DirectionToNotional::Short => LIMIT_ORDERS_BY_PRICE_SHORT.save(
                ctx.storage,
                (trigger_price.into_price_key(market_type), order_id),
                &(),
            )?,
        }

        LIMIT_ORDERS_BY_ADDR.save(ctx.storage, (&owner, order_id), &())?;

        let direction_to_base = direction.into_base(market_type);
        ctx.response.add_event(PlaceLimitOrderEvent {
            market_type,
            collateral: order.collateral,
            collateral_usd: price.collateral_to_usd_non_zero(collateral),
            leverage: order.leverage.into_signed(direction_to_base),
```

```
            direction: direction_to_base,
            max_gains,
            stop_loss_override,
            order_id,
            owner,
            trigger_price,
            take_profit_override,
        });

        Ok(())
    }
```

Listing 3.3: `market/src/state/order::limit_order_set_order()`

Recommendation Add same adjustment for `LiquidityTokenExecuteMsg::SendFrom` in the above mentioned function.

Status This issue has been fixed in the following commit: `06ea87`

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `Multiple contracts`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

In the `Levana` protocol, there is a privileged account, i.e., `owner`. This account plays a critical role in regulating the protocol-wide operations (e.g., configure parameters, assign other roles, as well as collect `DAO` fee). Our analysis shows that this privileged account needs to be scrutinized.

In the following, we use the market contract as an example and show the representative functions potentially affected by the privileges of the `owner` account.

```
    pub fn execute(deps: DepsMut, env: Env, info: MessageInfo, msg: ExecuteMsg) ->
        Result<Response> {
        if msg.requires_owner() && info.sender != get_owner(deps.storage)? {
            perp_bail!(
                ErrorId::Auth,
                ErrorDomain::Default,
                "{} is not the auth contract owner",
                info.sender
            )
        }

```

```
        let (state, mut ctx) = StateContext::new(deps, env)?;

        match msg {
            ExecuteMsg::AddMarket {
                new_market:
                    NewMarketParams {
                        market_id,
                        token,
                        config,
                        price_admin,
                        initial_borrow_fee_rate,
                    },
            } => {
                if get_market_addr(ctx.storage, &market_id).is_ok() {
                    return Err(anyhow!("market already exists for {market_id}"));
                }
                let migration_admin: Addr = get_admin_migration(ctx.storage)?;

                reply_set_instantiate_market(
                    ctx.storage,
                    InstantiateMarket {
                        market_id: market_id.clone(),
                        migration_admin: migration_admin.clone(),
                        price_admin: price_admin.validate(state.api)?,
                    },
                )?;

                let label_suffix = get_label_suffix(ctx.storage)?;

                ctx.response.add_instantiate_submessage(
                    ReplyId::InstantiateMarket,
                    &migration_admin,
                    get_market_code_id(ctx.storage)?,
                    format!("Levana Perps Market - {market_id}{label_suffix}"),
                    &msg::contracts::market::entry::InstantiateMsg {
                        factory: state.env.contract.address.into(),
                        config,
                        market_id,
                        token,
                        initial_borrow_fee_rate,
                    },
                )?;
            }

            ExecuteMsg::SetMarketCodeId { code_id } => {
                set_market_code_id(ctx.storage, code_id.parse()?)?;
            }
            ExecuteMsg::SetPositionTokenCodeId { code_id } => {
                set_position_token_code_id(ctx.storage, code_id.parse()?)?;
            }
            ExecuteMsg::SetLiquidityTokenCodeId { code_id } => {
                set_liquidity_token_code_id(ctx.storage, code_id.parse()?)?;
```

```
            }

            ExecuteMsg::SetOwner { owner } => {
                set_owner(ctx.storage, &owner.validate(state.api)?)?;
            }

            ExecuteMsg::SetDao { dao } => {
                set_dao(ctx.storage, &dao.validate(state.api)?)?;
            }

            ExecuteMsg::SetKillSwitch { kill_switch } => {
                set_kill_switch(ctx.storage, &kill_switch.validate(state.api)?)?;
            }

            ExecuteMsg::SetWindDown { wind_down } => {
                set_wind_down(ctx.storage, &wind_down.validate(state.api)?)?;
            }

            ExecuteMsg::SetMarketPriceAdmin {
                market_addr,
                admin_addr,
            } => {
                set_admin_market_price(
                    ctx.storage,
                    &market_addr.validate(state.api)?,
                    &admin_addr.validate(state.api)?,
                )?;
            }

            ExecuteMsg::TransferAllDaoFees {} => {
                let addrs = MARKET_ADDRS
                    .range(ctx.storage, None, None, cosmwasm_std::Order::Ascending)
                    .map(|res| res.map(|(_, addr)| addr).map_err(|err| err.into()))
                    .collect::<Result<Vec<Addr>>>()?;

                for addr in addrs {
                    ctx.response
                        .add_execute_submessage_oneshot(addr, &MarketExecuteMsg::
                            TransferDaoFees {})?;
                }
            }
            ExecuteMsg::Shutdown {
                markets,
                impacts,
                effect,
            } => shutdown(&mut ctx, &info, markets, impacts, effect)?,
        }

        Ok(ctx.response.into_response())
    }
```

Listing 3.4: `market/src/contract::execute()`

We understand the need of the privileged functions for proper operations, but at the same time the extra power to the `owner` may also be a counter-party risk to the `Levana` users. Therefore, we list this concern as an issue here from the audit perspective and highly recommend making these privileges explicit or raising necessary awareness among protocol users.

Recommendation Make the list of extra privileges granted to `Levana` explicit to `Levana` users.

Status This issue has been confirmed and mitigated with a multisig account with the planned `DAO`-like governance in the future.

3.5 Redundant State/Code Removal

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `sanity.rs`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1041 [1]

Description

In the `Levana` implementation, the `liquidation_prices()` function performs a sanity check on the limit price. The limit price-related information can only exist in either `LIQUIDATION_PRICES_PENDING` or `PRICE_TRIGGER_DESC/ASC`, but not in both. While examining its logic, we observe the inclusion of certain unused code or the presence of unnecessary redundancies that can be safely removed.

To elaborate, we show below the related code snippet of the `sanity.rc` file. The function `ensure_missing()` is used to ensure that `PRICE_TRIGGER_DESC` and `PRICE_TRIGGER_ASC` do not contain position-related information. However, it performs redundant checks, which can be safely removed).

```
fn liquidation_prices(store: &dyn Storage, _env: &Env) -> Result<()> {
    for res in OPEN_POSITIONS.range(store, None, None, cosmwasm_std::Order::
        Ascending) {
        let (posid, pos) = res?;

        let pending_time = LIQUIDATION_PRICES_PENDING_REVERSE.may_load(store, posid)
            ?;
        let pending = match pending_time {
            Some(pending_time) => {
                Some(LIQUIDATION_PRICES_PENDING.load(store, (pending_time, posid))?)
            }
            None => None,
        };

        match pending {
            ...
```

```
            Some(_) => {
                ensure_missing(store, PRICE_TRIGGER_DESC, posid)?;
                ensure_missing(store, PRICE_TRIGGER_ASC, posid)?;
                ensure_missing(store, PRICE_TRIGGER_ASC, posid)?;
                ensure_missing(store, PRICE_TRIGGER_DESC, posid)?;
            }
        }
    }

    Ok(())
}
```

Listing 3.5: `market/src/state/sanity::liquidation_prices()`

Moreover, the protocol emits the `DeltaNeutralityRatioEvent` event when the `delta neutrality ratio` is updated. However, we notice the same event is emitted twice in both `update_position_size()` and `update_position_leverage()` functions from the `market/src/state/position/update` contract.

Recommendation Consider the removal of the redundant code with a simplified, consistent implementation.

Status The issue has been fixed by this commit: `09c6ce9`

3.6 Revisited Crank Fee Collection in Limit Order

- ID: PVE-006
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `order.rs`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1041 [1]

Description

In `Levana`, each crank operation may charge the associated crank fee for the crank to run. Positions that schedule cranks for later execution must reserve sufficient collateral to pay the fee. While examining current logic, we observe that the limit order's crank fee is collected at the time when the limit order is placed. However, when the same limit order is cancelled before being triggered, the collected crank fee may not return.

To elaborate, we show below the related code snippet of the `limit_order_set_order()` function. The function is used to place a limit order with crank fee collected. A possibly improved alternative is to collect the crank fee when the limit order is triggered, not at its placement.

```
pub(crate) fn limit_order_set_order(
    &self,
    ctx: &mut StateContext,
```

```
        owner: Addr,
        trigger_price: PriceBaseInQuote,
        collateral: NonZero<Collateral>,
        leverage: LeverageToBase,
        direction: DirectionToNotional,
        max_gains: MaxGainsInQuote,
        stop_loss_override: Option<PriceBaseInQuote>,
        take_profit_override: Option<PriceBaseInQuote>,
    ) -> Result<()> {
        let last_order_id = LAST_ORDER_ID
            .may_load(ctx.storage)?
            .unwrap_or_else(|| OrderId::new(0));
        let order_id = OrderId::new(last_order_id.u64() + 1);
        LAST_ORDER_ID.save(ctx.storage, &order_id)?;

        let order_fee = Collateral::try_from_number(
            collateral
                .into_number()
                .checked_mul(self.config.limit_order_fee.into_number())?,
        )?;
        let collateral = collateral.checked_sub(order_fee)?;
        let price = self.spot_price(ctx.storage, None)?;
        self.collect_limit_order_fee(ctx, order_id, order_fee, price)?;

        let crank_fee_usd = self.config.crank_fee_charged;
        let crank_fee = price.usd_to_collateral(crank_fee_usd);
        self.collect_crank_fee(ctx, TradeId::LimitOrder(order_id), crank_fee,
            crank_fee_usd)?;
        let collateral = collateral
            .checked_sub(crank_fee)
            .context("Insufficient funds to cover fees, failed on crank fee")?;

        let order = LimitOrder {
            order_id,
            owner: owner.clone(),
            trigger_price,
            collateral,
            leverage,
            direction,
            max_gains,
            stop_loss_override,
            take_profit_override,
        };

        self.limit_order_validate(ctx.storage, &order)?;

        LIMIT_ORDERS.save(ctx.storage, order_id, &order)?;
        ...
    }
```

Listing 3.6: `market/src/state/order::limit_order_set_order()`

Recommendation Collect the crank fee when the limit order is triggered.

Status This issue has been resolved as the team confirms the purpose is to implement certain spam prevention and add a new potential failure vector.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Levana` protocol, which is a leveraged spot-price well-funded collateral-settled perpetual swaps protocol. It introduces a novel way to delineate risk between market participants. The incentive structure offers risk premium to the liquidity providers taking on the spot market illiquidity risk. The fundamental problems `Levana` addresses is the risk of illiquidity and market manipulation. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Meanwhile, we need to emphasize that smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

References

[1] MITRE. CWE-1041: Use of Redundant Code. https://cwe.mitre.org/data/definitions/1041.html.

[2] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[3] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

[4] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

[5] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

[6] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

[7] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

[8] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[9] PeckShield. PeckShield Inc. https://www.peckshield.com.