Veridise. Auditing Report

Hardening Blockchain Security with Formal Methods

FOR

Tonic-Perps

Veridise Inc.

February 13, 2023

► **Prepared For:**

Tonic Foundation
tonic.foundation

► **Prepared By:**

Benjamin Mariano
Benjamin Sepanski
Andreea Buţerchi

► **Contact Us:**

contact@veridise.com

► **Version History:**

Feb 13, 2023 V1

Contents

Contents iii

1 Executive Summary 1

2 Project Dashboard 3

3 Audit Goals and Scope 5
- 3.1 Audit Goals . . . 5
- 3.2 Audit Methodology & Scope . . . 5
- 3.3 Classification of Vulnerabilities . . . 6

4 Vulnerability Report 7
- 4.1 Detailed Description of Bugs . . . 8
  - 4.1.1 V-TNC-VUL-001: Potential unlimited LP Token minting/burning . . . 8
  - 4.1.2 V-TNC-VUL-002: Reserved amounts not preserved when burning LP Tokens . . . 10
  - 4.1.3 V-TNC-VUL-003: Limit orders are merged even with different collateral types . . . 11
  - 4.1.4 V-TNC-VUL-004: No callback given for call to ft_transfer . . . 14
  - 4.1.5 V-TNC-VUL-005: Potential DOS on asset withdrawal . . . 15
  - 4.1.6 V-TNC-VUL-006: Limit on max asset price change could be abused by short takers . . . 16
  - 4.1.7 V-TNC-VUL-007: Public initialization function . . . 19
  - 4.1.8 V-TNC-VUL-008: Storage of unchecked String . . . 20
  - 4.1.9 V-TNC-VUL-009: Assumed collateral and underlying assets are the same in value calculation . . . 21
  - 4.1.10 V-TNC-VUL-010: Lost funds on cancelled limit orders . . . 22
  - 4.1.11 V-TNC-VUL-011: Users liquidated when below minimum leverage . . . 23
  - 4.1.12 V-TNC-VUL-012: set_user_referral_code uses predecessor instead of signer 24
  - 4.1.13 V-TNC-VUL-013: Stale state before external call . . . 25
  - 4.1.14 V-TNC-VUL-014: Short position checks total stable available liquidity instead of collateral available liquidity . . . 27
  - 4.1.15 V-TNC-VUL-015: Withdrawals from NEAR asset never check storage requirements . . . 31
  - 4.1.16 V-TNC-VUL-016: Storage Taking attack: transferring LP Tokens to bogus accounts . . . 32
  - 4.1.17 V-TNC-VUL-017: Malicious set user referral code induces large storage cost 33
  - 4.1.18 V-TNC-VUL-018: ft_on_transfer uses signer account rather than sender . 34
  - 4.1.19 V-TNC-VUL-019: Possible limit order ID collisions . . . 35
  - 4.1.20 V-TNC-VUL-020: Pool can lose money on liquidation . . . 36
  - 4.1.21 V-TNC-VUL-021: Confusing function usage . . . 37
  - 4.1.22 V-TNC-VUL-022: Multiple functions made payable unnecessarily . . . 38
  - 4.1.23 V-TNC-VUL-023: is_liquidator uses unmodified field Contract::liquidators 39
  - 4.1.24 V-TNC-VUL-24: Out-of-date class documentation for LimitOrderID . . . 40

4.1.25 V-TNC-VUL-025: Out-of-date function documentation for get_lp_redemption_amount . . . 41
4.1.26 V-TNC-VUL-026: Add limit order does not check if NEAR sent on decrease 42
4.1.27 V-TNC-VUL-027: Referral code creation may charge more than CREATE_REFERRER_FEE . . . 43
4.1.28 V-TNC-VUL-028: Minimum amount out in swap is checked before fees . 44
4.1.29 V-TNC-VUL-029: Redundant function call . . . 45
4.1.30 V-TNC-VUL-030: Incorrect type annotation . . . 46
4.1.31 V-TNC-VUL-031: Code structure suggestion: use Rust Tuple Structs to track currency unit types . . . 47
4.1.32 V-TNC-VUL-032: Code structure suggestion: check contract invariants first 48
4.1.33 V-TNC-VUL-033: Code structure suggestion: split logic for short and long positions . . . 49
4.1.34 V-TNC-VUL-034: Add in logging for internal transfer failure . . . 50
4.1.35 V-TNC-VUL-035: Use #[must_use] for any functions which return balances that must be sent to a user . . . 51
4.1.36 V-TNC-VUL-036: Replace complicated limit order merge logic . . . 52

1 Executive Summary

From Jan. 9 to Feb. 13, Tonic engaged Veridise to review the security of their Tonic-Perps project. The review covered the on-chain contracts that implement the protocol logic. Veridise conducted the assessment over 12 person-weeks, with 3 engineers reviewing code over 4 weeks on commit `1f30b00`. The auditing strategy involved an analysis of the source code performed by Veridise engineers involving extensive manual auditing.

Summary of issues detected. The audit uncovered 36 issues, 6 of which are assessed to be of high or critical severity by the Veridise auditors. Potential consequences of these issues include unlimited minting and burning of the LP token (V-TNC-VUL-001), reserve funds being depleted (V-TNC-VUL-002), and funds depleted by allowing invalid limit order merging (V-TNC-VUL-003). In addition to these critical/high severity bugs, auditors also found many moderate severity issues. These include three different storage attacks that could be used to drain some contract funds (V-TNC-VUL-008, V-TNC-VUL-016, V-TNC-VUL-017), poor program logic leading to lost user funds (V-TNC-VUL-010), and multiple unnecessarily payable functions that could cause users to lose funds (V-TNC-VUL-022).

Code assessment. Tonic-Perps is an orderbook-based exchange built on the NEAR blockchain. The protocol enables users to exchange tokens and take out both short and long positions on assets in the pool. Unlike AMM liquidaty pools which use a constant product rule to price pairs of assets, Tonic-Perps has a single pool containing all assets and determines prices via oracles. Like most exchanges, Tonic-Perps incentivizes liquidity providers by rewarding them with fees assessed by the protocol.

Tonic provided the source code for the Tonic-Perps contracts for review. The contract contained a test suite which achieved 82.14% line coverage (see Table 1.1). The code has some documentation that was shared with auditors, and the code contains some moderate commenting.

Suggestions. After auditing the protocol, auditors had a number of suggestions that we believe should be taken by Tonic-Perps developers before releasing the protocol. First, we suggest that developers more thoroughly test the code (currently test coverage is only 82.14%). We believe more thorough testing could have identified some of the major issues discovered in this audit, such as reserved amounts not being preserved (V-TNC-VUL-002). Further, several files have low or zero coverage, such as `referrals.rs` or `lp_token/storage.rs`. Increased testing based on the test coverage results might catch issues similar to V-TNC-VUL-014.

Additionally, we found that the treatment of different currency units in the code is potentially error-prone and difficult to maintain. Different currency units are (mostly) denoted only with naming convention. We suggest developers differentiate these units using Rust's Tuple Structs so that issues can be detected automatically by Rust's compiler*. Third, we suggest developers change the coding style to do input validation and invariant checking at the beginning of

* See also the New Type Idiom

Table 1.1: Test Coverage.

Filename	Lines	Lines Missed	Line Coverage
actions.rs	6	0	100.00%
admin.rs	336	68	79.76%
fees/mod.rs	160	6	96.25%
lib.rs	93	13	86.02%
lp_token/ft.rs	119	96	19.33%
lp_token/mint.rs	194	4	97.94%
lp_token/mod.rs	65	6	90.77%
lp_token/storage.rs	21	8	61.90%
oracle.rs	109	58	46.79%
perps/limit_order.rs	518	41	92.08%
perps/limit_order_id.rs	86	16	81.40%
perps/mod.rs	1139	61	94.64%
perps/position_id.rs	56	18	67.86%
referrals.rs	94	94	0.00%
switchboard.rs	8	8	0.00%
token_receiver.rs	86	6	93.02%
trading.rs	154	3	98.05%
upgrade.rs	25	25	0.00%
util.rs	61	20	67.21%
vault/asset.rs	578	70	87.89%
vault/mod.rs	42	0	100.00%
views.rs	372	157	57.80%
withdrawal_history.rs	85	9	89.41%
TOTAL	4407	787	82.14%

functions. Checks are peppered throughout the code, which makes reasoning about the code difficult. Finally, we suggest the logic for long and short positions be separated. The logic for these two very different actions is mixed together in the same functions. We believe this increases the chance of unsafe behavior, especially in future iterations of the codebase.

Disclaimer. We hope that this report is informative but provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the system is secure in all dimensions. In no event shall Veridise or any of its employees be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the results reported here.

2 Project Dashboard

Table 2.1: Application Summary.

Name	Version	Type	Platform
Tonic-Perps	`1f30b00`	Rust	NEAR

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
Jan. 9 - Feb. 13, 2022	Manual	3	12 person-weeks

Table 2.3: Vulnerability Summary.

Name	Number	Resolved
Critical-Severity Issues	3	3
High-Severity Issues	3	3
Medium-Severity Issues	5	5
Low-Severity Issues	9	9
Warning-Severity Issues	6	6
Informational-Severity Issues	10	9
TOTAL	36	35

Table 2.4: Category Breakdown.

Name	Number
Logic Error	19
Validation	1
Denial of Service	3
Access Control	1
Missing/Incorrect Events	1
Gas Optimization	1
Maintainability	8
Usability	2

3 Audit Goals and Scope

3.1 Audit Goals

The engagement was scoped to provide a security assessment of the on-chain portion of the Tonic-Perps defined below. In our audit, we sought to answer the following questions:

- Can a malicious user manipulate the balance of assets held in the vault?
- Can a malicious user steal another user's collateral?
- Can a user's earnings or collateral be lost?
- Does the protocol maintain the appropriate reserve amounts?
- Are failed external calls appropriately handled via callback functions?
- Is the protocol vulnerable to storage cost attacks?
- Can a user illegally withdraw their collateral for an open position?

3.2 Audit Methodology & Scope

Audit Methodology. To address the questions above, our audit involved an extensive manual audit. This included auditors reading documentation, reviewing code, writing new test cases, and performing test coverage analysis, among other tasks.

Scope. This audit reviewed the on-chain behaviors contained in the following files of the Tonic-Perps.

- `tonic-perps/crates/tonic-perps/src/actions.rs`
- `tonic-perps/crates/tonic-perps/src/admin.rs`
- `tonic-perps/crates/tonic-perps/src/constants.rs`
- `tonic-perps/crates/tonic-perps/src/events.rs`
- `tonic-perps/crates/tonic-perps/src/lib.rs`
- `tonic-perps/crates/tonic-perps/src/oracle.rs`
- `tonic-perps/crates/tonic-perps/src/referrals.rs`
- `tonic-perps/crates/tonic-perps/src/switchboard.rs`
- `tonic-perps/crates/tonic-perps/src/token_receiver.rs`
- `tonic-perps/crates/tonic-perps/src/trading.rs`
- `tonic-perps/crates/tonic-perps/src/upgrade.rs`
- `tonic-perps/crates/tonic-perps/src/util.rs`
- `tonic-perps/crates/tonic-perps/src/views.rs`
- `tonic-perps/crates/tonic-perps/src/withdrawal_history.rs`
- `tonic-perps/crates/tonic-perps/src/fees/mod.rs`
- `tonic-perps/crates/tonic-perps/src/lp_token/ft.rs`
- `tonic-perps/crates/tonic-perps/src/lp_token/mint.rs`
- `tonic-perps/crates/tonic-perps/src/lp_token/mod.rs`
- `tonic-perps/crates/tonic-perps/src/lp_token/storage.rs`

- `tonic-perps/crates/tonic-perps/src/perps/limit_order.rs`
- `tonic-perps/crates/tonic-perps/src/perps/limit_order_id.rs`
- `tonic-perps/crates/tonic-perps/src/perps/mod.rs`
- `tonic-perps/crates/tonic-perps/src/perps/position_id.rs`
- `tonic-perps/crates/tonic-perps/src/vault/asset.rs`
- `tonic-perps/crates/tonic-perps/src/vault/mod.rs`

3.3 Classification of Vulnerabilities

When Veridise auditors discover a possible security vulnerability, they must estimate its severity by weighing its potential impact against the likelihood that a problem will arise. Table 3.1 shows how our auditors weigh this information to estimate the severity of a given issue.

Table 3.1: Severity Breakdown.

	Somewhat Bad	Bad	Very Bad	Protocol Breaking
Not Likely	Info	Warning	Low	Medium
Likely	Warning	Low	Medium	High
Very Likely	Low	Medium	High	Critical

In this case, we judge the likelihood of a vulnerability as follows:

Not Likely	A small set of users must make a specific mistake
Likely	Requires a complex series of steps by almost any user(s) - OR - Requires a small set of users to perform an action
Very Likely	Can be easily performed by almost anyone

In addition, we judge the impact of a vulnerability as follows:

Somewhat Bad	Inconveniences a small number of users and can be fixed by the user
Bad	Affects a large number of people and can be fixed by the user - OR - Affects a very small number of people and requires aid to fix
Very Bad	Affects a large number of people and requires aid to fix - OR - Disrupts the intended behavior of the protocol for a small group of users through no fault of their own
Protocol Breaking	Disrupts the intended behavior of the protocol for a large group of users through no fault of their own

4 Vulnerability Report

In this section, we describe the vulnerabilities found during our audit. For each issue found, we log the type of the issue, its severity, location in the code base, and its current status (i.e., acknowledged, fixed, etc.). Table 4.1 summarizes the issues discovered:

Table 4.1: Summary of Discovered Vulnerabilities.

ID	Description	Severity	Status
V-TNC-VUL-001	Potential unlimited LP token minting/burning	Critical	Fixed
V-TNC-VUL-002	Reserved amounts not preserved when burning	Critical	Fixed
V-TNC-VUL-003	Unchecked limit order merging	Critical	Fixed
V-TNC-VUL-004	No callback given for call to ft_transfer	High	Won't Fix
V-TNC-VUL-005	Potential DOS on asset withdrawal	High	Fixed
V-TNC-VUL-006	Abuse of limited asset price change using shorts	High	Acknowledged
V-TNC-VUL-007	Public initialization function	Medium	Fixed
V-TNC-VUL-008	Storage of unchecked string	Medium	Fixed
V-TNC-VUL-009	Assumed same collateral and underlying type	Medium	Fixed
V-TNC-VUL-010	Lost funds on cancelled limit orders	Medium	Fixed
V-TNC-VUL-011	Users liquidated when below minimum leverage	Medium	Acknowledged
V-TNC-VUL-012	Using predecessor instead of signer	Low	Fixed
V-TNC-VUL-013	Stale state before external call	Low	Acknowledged
V-TNC-VUL-014	Short checks wrong liquidity amount	Low	Fixed
V-TNC-VUL-015	Withdrawals don't check storage requirements	Low	Won't Fix
V-TNC-VUL-016	Storage attack via bogus transfers	Low	Fixed
V-TNC-VUL-017	Storage attack via user referral code	Low	Fixed
V-TNC-VUL-018	`ft_on_transfer` uses signer account	Low	Fixed
V-TNC-VUL-019	Possible limit order ID collisions	Low	Acknowledged
V-TNC-VUL-020	Pool can lose money on liquidation	Low	Fixed
V-TNC-VUL-021	Confusing function usage	Warning	Acknowledged
V-TNC-VUL-022	Function unnecessarily payable	Warning	Fixed
V-TNC-VUL-023	Unused variable	Warning	Fixed
V-TNC-VUL-024	Out-of-date class documentation	Warning	Fixed
V-TNC-VUL-025	Out-of-date function documentation	Warning	Fixed
V-TNC-VUL-026	Potential NEAR loss on add decrease limit order	Warning	Invalid
V-TNC-VUL-027	Referral code creation too expensive	Info	Fixed
V-TNC-VUL-028	Minimum amount checked before fees	Info	Intended Behavior
V-TNC-VUL-029	Redundant function call	Info	Fixed
V-TNC-VUL-030	Incorrect type annotation	Info	Fixed
V-TNC-VUL-031	Use Rust Tuple Structs to track currency units	Info	Open
V-TNC-VUL-032	Check contract invariants first	Info	Fixed
V-TNC-VUL-033	Split logic for short and long positions	Info	Won't Fix
V-TNC-VUL-034	Add logging for internal transfer failure	Info	Fixed
V-TNC-VUL-035	Use `#[must_use]` for funcs returning balance	Info	Fixed
V-TNC-VUL-036	Replace complicated limit order merge logic	Info	Won't Fix

4.1 Detailed Description of Bugs

4.1.1 V-TNC-VUL-001: Potential unlimited LP Token minting/burning

Severity	Critical	**Commit**	1f30b00
Type	Logic Error	**Status**	Fixed
Files	`lp_token/mint.rs`, `vault/asset.rs`		
Functions	`Contract::mint_lp_token`		

By default, the `withdrawal_limit_bps` for an asset is set to `100%`, meaning that a user can withdraw an arbitrary amount of the asset by burning LP Tokens.

```
pub fn new(
        asset_id: AssetId,
        decimals: u8,
        stable: bool,
        token_weight: u32,
        base_funding_rate: u64,
    ) -> Self {
        Self {
            asset_id,
                        ..
            withdrawal_limit_bps: 10000,
        }
    }
```

Impact

- A user can manipulate the total supply of LP Tokens without limit.
- A user can manipulate the composition of assets in the protocol (e.g., completely replace all USDT with NEAR).

Example

- Assume the pool has `50` NEAR and `50` USDT, the price of NEAR to USDT is 1:1, and there are `100` LP Tokens.
- Alice takes out a loan for `50` NEAR.
- Alice mints `50` LP Tokens for `50` NEAR, meaning the pool now has `100` NEAR, `50` USDT, and there are now `150` LP Tokens.
- Alice burns `50` LP Tokens for `50` USDT, meaning the pool now has `100` NEAR, `0` USDT, and there are `100` LP Tokens.
- Alice pays back her loan using `50` USDT (plus some minor fees that are a cost she must incur for the attack).

Recommendation Set the default withdrawal limit to be less than 100% (preferably much less to avoid this).

The Tonic team has established a withdrawal limit of 50%, necessitating the frequent oversight of team members to avoid depleting the pool below this threshold. A reduction of the withdrawal

limit would facilitate less frequent monitoring. For instance, a limit of 6.25% would require oversight only every 8 hours to prevent the pool from falling below the 50% mark. The team will employ Grafana to monitor balances to ensure they are monitoring the pool at least once every hour.

4.1.2 V-TNC-VUL-002: Reserved amounts not preserved when burning LP Tokens

Severity	Critical	**Commit**	1f30b00
Type	Logic Error	**Status**	Fixed
Files	`lp_token/mint.rs`		
Functions	`Contract::burn_lp_token`		

`burn_lp_token` does not check that the reserved portion of an asset is maintained.

Impact The protocol may fail to pay out profits because too much liquidity has been removed.

Recommendation Add a check that the reserved amount remains during any withdrawal from burning LP tokens.

4.1.3 V-TNC-VUL-003: Limit orders are merged even with different collateral types

Severity	Critical	**Commit**	1f30b00
Type	Logic Error	**Status**	Fixed
Files	`perps/limit_order.rs`		
Functions	`Contract::add_limit_order`		

When multiple limit orders of the same type are made at the same price, they are merged together. However, no check ensures their underlying collateral types are the same. Instead, the collateral assets are assumed to be the same and the amounts are simply added.

```
let id = if let Some((existing_id, existing_order)) = limit_orders
            .get_range(
                limit_order.price,
                limit_order.price,
                limit_order.is_long,
                limit_order.threshold,
            )
            .find(|(_, lo)| lo.owner == params.owner && lo.order_type == params.
    order_type)
        {
            limit_order.collateral_delta += existing_order.collateral_delta;
            limit_order.attached_collateral += existing_order.attached_collateral;
            limit_order.size_delta += existing_order.size_delta;

            self.check_limit_order(&limit_order);

            *existing_id
        }
        ...
```

Impact A user can exploit this to drain funds by making multiple limit orders at the same price with different collateral types and then immediately removing the order and claiming the returns in the more valuable collateral type. This can be devastating if asset values and/or denominations are very different (as is the case with NEAR and USDC).

Recommendation Add a check that the collateral types are the same or convert to the correct collateral type.

```
#[test]
fn test_bad_merge() {
    let (mut context, mut vcontract) = setup();
    set_predecessor(&mut context, Admin);

    // add liquidity to NEAR
    vcontract
        .contract_mut()
        .add_liquidity(&AssetId::NEAR, near(100));

```



```
    // add liquidity to USDC
    vcontract
        .contract_mut()
        .add_liquidity(&AssetId::Ft(usdc_id().parse().unwrap()), dollars(1000));

    update_near_price(&mut vcontract, dollars(5));

    // Add a limit order for a long of NEAR using 5 NEAR collateral (1x leverage)
    set_predecessor(&mut context, Alice);
    set_deposit(&mut context, near(5));
    let params = LimitOrderParameters {
        price: dollars(5).into(),
        size_delta: dollars(25).into(),
        underlying_id: near_id(),
        collateral_id: None,
        is_long: true,
        expiry: None,
        order_type: OrderType::Increase,
        collateral_delta: None,
    };
    let limit_order_id =vcontract.add_limit_order(params.clone());

    // Add a limit order for additional increase at same price w/ $5 USDC collateral
    (1x leverage)
    set_signer(&mut context, Alice);
    set_predecessor(&mut context, Alice);
    set_predecessor_token(&mut context, usdc_id());
    vcontract.ft_on_transfer(
        get_account(Alice),
        dollars(5).into(),
        serde_json::to_string(&Action::PlaceLimitOrder(LimitOrderParameters {
            price: dollars(5).into(),
            size_delta: dollars(5).into(),
            underlying_id: near_id(),
            collateral_id: None,
            is_long: true,
            expiry: None,
            order_type: OrderType::Increase,
            collateral_delta: None,
        }))
        .unwrap(),
    );

    set_predecessor(&mut context, Alice);
    vcontract.remove_limit_order(limit_order_id);

}
```

When run, this prints out:

```
{"type":"OracleUpdate","data":{"asset_id":"usdc","price":"1000000","spread_bps":0,"
    source":"tonic"}}
{"type":"EditPoolBalance","data":{"amount_native":10000000000000000000000000,"
    new_pool_balance_native":10000000000000000000000000,"increase":true,"account_id"
```

```
    :"charlie","asset_id":"near"}}
{"type":"EditPoolBalance","data":{"amount_native":1000000000,"new_pool_balance_native
    ":1000000000,"increase":true,"account_id":"charlie","asset_id":"usdc"}}
{"type":"OracleUpdate","data":{"asset_id":"near","price":"5000000","spread_bps":0,"
    source":"tonic"}}
{"type":"PlaceLimitOrder","data":{"account_id":"alice","limit_order_id":"8
    uNfjwMUkeiy1yPjtPTu4t","collateral_token":"near","underlying_token":"near","
    order_type":"increase","threshold_type":"above","collateral_delta_usd":"0","
    attached_collateral_native":"500000000000000000000000","size_delta_usd":"
    25000000","price_usd":"5000000","expiry":"2678400000","is_long":true}}
{"type":"TokenDepositWithdraw","data":{"amount_native":"500000000000000000000000","
    deposit":true,"method":"add_limit_order","receiver_id":"alice","account_id":"
    alice","asset_id":"near"}}
{"type":"PlaceLimitOrder","data":{"account_id":"alice","limit_order_id":"8
    uNfjwMUkeiy1yPjtPTu4t","collateral_token":"usdc","underlying_token":"near","
    order_type":"increase","threshold_type":"above","collateral_delta_usd":"0","
    attached_collateral_native":"500000000000000000005000000","size_delta_usd":"
    30000000","price_usd":"5000000","expiry":"2678400000","is_long":true}}
{"type":"TokenDepositWithdraw","data":{"amount_native":"5000000","deposit":true,"
    method":"ft_on_transfer","receiver_id":"alice","account_id":"alice","asset_id":"
    usdc"}}
{"type":"RemoveLimitOrder","data":{"account_id":"alice","underlying_token":"near","
    limit_order_id":"8uNfjwMUkeiy1yPjtPTu4t","reason":"removed","liquidator_id":null
    }}
{"type":"TokenDepositWithdraw","data":{"amount_native":"500000000000000005000000","
    deposit":false,"method":"remove_limit_order","receiver_id":"alice","account_id":"
    alice","asset_id":"usdc"}}
test test_bad_merge ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished
    in 0.00s
```

In this test case, Alice deposits 5 NEAR (worth $25) in the first limit order and $5 in the second limit order. Per the last line about "TokenDepositWithdraw", she gets back $5000000000000000005 (USDC decimals is 6).

4.1.4 V-TNC-VUL-004: No callback given for call to ft_transfer

Severity	High	**Commit**	1f30b00
Type	Logic Error	**Status**	Won't Fix
Files	`Vault/mod.rs`		
Functions	`Contract::internal_send_ft`		

The following call to `ft_transfer`* does not specify a callback function.

```
pub fn internal_send_ft(
        &self,
        receiver_id: &AccountId,
        token_id: &AccountId,
        amount: Balance,
    ) -> Promise {

        ext_ft_core::ext(token_id.clone())
            .with_attached_deposit(1)
            .with_static_gas(Gas::ONE_TERA * TGAS_FOR_FT_TRANSFER)
            .with_unused_gas_weight(0)
            .ft_transfer(receiver_id.clone(), amount.into(), None)
    }
}
```

Impact If the call fails, a user could end up losing funds. For example, consider the following example:

- Alice has some fungible token `ftA` and wants to convert the token to `ftB` and send to Bob.
- Alice calls `ftA::ft_transfer_call` with `receiver_id=<VContract ID>` and `msg=Action::Swap`.
- `ftA` invokes `VContract::ft_on_transfer`, which in turn invokes a `swap_and_send`.
- The `swap_and_send` correctly handles the internal account for `ftA`/`ftB` assets, then uses `internal_send` to send `amount_out` many `ftB` to Bob.
- Now suppose Bob isn't registered with `ftB`. Then, `amount_out ftB` tokens were just destroyed, and Alice lost her money.

Recommendation Add a callback in the event that `ft_transfer` fails.

* NEP-141: Fungible Token standard

4.1.5 V-TNC-VUL-005: Potential DOS on asset withdrawal

Severity	High	**Commit**	1f30b00
Type	Denial of Service	**Status**	Fixed
Files	`lp_token/mint.rs`, `vault/asset.rs`		
Functions	`Contract::mint_lp_token`		

If the `withdrawal_limit_bps` is small enough and an attacker has sufficient funds in the protocol, they can use `burn_lp_token` to withdraw the maximum amount for the current time window. They can do this every time the time window is up to block others who want to withdraw. They can do this at a relatively small loss by minting LP token after each burn, only losing the amount of fees on each burn/mint.

Impact An attacker can prevent an owner of an LP token from being able to withdraw their funds from a particular asset (or from any asset if this attack is performed on all assets in the protocol).

Recommendation Track net withdrawals (i.e., withdrawals - deposits).

4.1.6 V-TNC-VUL-006: Limit on max asset price change could be abused by short takers

Severity	High	Commit	1f30b00
Type	Incorrect Events	Status	Acknowledged
Files	`oracle.rs`		
Functions	NA		

`Contract::update_index_price` limits price changes to a maximum of `max_price_change_bps` per second.

Impact If an adversary realizes that an asset price is dropping faster than `max_price_change_bps` per second, they can take out a short on the asset. Assuming the price continues to drop (or at least the price does not recover quickly), the adversary will be able to pull as much money out of the contract as they are able to leverage, multiplied by the difference in "actual price change bps" and `max_price_change_bps`.

An example attack is shown below, which extracts $300 dollars from the liquidity pool by noticing that, although the NEAR price has reduced from $10 to $5, the pool has only dropped the price to $8 because of `max_price_change_bps`. The adversary then takes out a $100 short before the prices converge.

```
#[test]
fn test_oracle_max_change_short_drain() {
    let (mut context, mut vcontract) = setup();
    set_predecessor(&mut context, Admin);

    // Limit price changes at 20%/second
    vcontract.set_max_asset_price_change(near_id(), Some(U128(2000)));

    vcontract
        .contract_mut()
        .add_liquidity(&AssetId::NEAR, near(100));
    vcontract
        .contract_mut()
        .add_liquidity(&AssetId::Ft(usdc_id().parse().unwrap()), dollars(1000));
    update_near_price(&mut vcontract, dollars(10));

    let assets = vcontract.get_assets();
    let near = assets.iter().find(|asset| asset.id == "near");
    assert_eq!(dollars(10), near.unwrap().average_price.0);

    // After some time, the near price decreases by 50%!
    let new_time = near_sdk::env::block_timestamp() + std::time::Duration::from_secs
    (1).as_nanos() as u64;
    context.block_timestamp(new_time);
    testing_env!(context.build());
    update_near_price(&mut vcontract, dollars(5));

    // Recorded price should only decrease by 20%
    let assets = vcontract.get_assets();
```

```
    let near = assets.iter().find(|asset| asset.id == "near");
    assert_eq!(dollars(8), near.unwrap().average_price.0);

    // Now an adversary notices the gap, short near as large as we can!
    // Note that because the "near" price is still too high at $8, this helps us
    leverage
    // more value
    set_predecessor(&mut context, Alice);
    set_deposit(&mut context, common::near(10));
    let position_size = dollars(800);
    let position_id = vcontract.increase_position(common::IncreasePositionRequest {
        underlying_id: "near".to_string(),
        size_delta: U128(position_size),
        is_long: false,
        referrer_id: None,
    });

    // Now over time the prices stabilize to the $5
    set_predecessor(&mut context, Admin);
    let new_time = near_sdk::env::block_timestamp() + std::time::Duration::from_secs
    (5).as_nanos() as u64;
    context.block_timestamp(new_time);
    testing_env!(context.build());
    update_near_price(&mut vcontract, dollars(5));

    // Recorded price should now be true price of $5
    let assets = vcontract.get_assets();
    let near = assets.iter().find(|asset| asset.id == "near");
    assert_eq!(dollars(5), near.unwrap().average_price.0);

    let dollars_before = vcontract.get_assets()
        .into_iter()
        .find(|asset| asset.id == usdc_id())
        .unwrap()
        .pool_amount.0;

    // Now Alice pulls out the short
    set_predecessor(&mut context, Alice);
    vcontract.decrease_position(common::DecreasePositionRequest{
        size_delta: U128(position_size),
        position_id,
        referrer_id: None,
        collateral_delta: U128(40),
        output_token_id: None,
    });

    let dollars_after = vcontract.get_assets()
        .into_iter()
        .find(|asset| asset.id == usdc_id())
        .unwrap()
        .pool_amount.0;
    // Alice just profited $300 off the liquidity pool!
    assert_eq!(dollars_after + dollars(300), dollars_before);
```



```
80 }
```

Recommendation

- `max_price_change_bps` should be chosen judiciously. This value should also take into account how often prices are updated.
- When price changes are limited by `max_price_change_bps`, some action should be taken. For full security, we recommend temporarily pausing perps and limit orders until the price stabilizes. Further discussion is warranted on the action to take.

4.1.7 V-TNC-VUL-007: Public initialization function

Severity	Medium	**Commit**	1f30b00
Type	Access Control	**Status**	Fixed
Files	`lib.rs`		
Functions	`VContract::new()`		

The `init` method is not declared private.

```
#[allow(clippy::new_without_default)]
#[init]
pub fn new() -> Self {
    let owner_id = env::predecessor_account_id();
    let mut admins = UnorderedMap::new(StoragePrefix::Admins);
    admins.insert(&owner_id, &AdminRole::FullAdmin);
    ...
```

In the NEAR documentation (https://docs.near.org/develop/contracts/anatomy), they suggest either declaring it `#[private]` or using batch initialization. Neither appears to be done.

Impact Arbitrary users can call this function, setting important variables like the owner id.

Recommendation Add the `#[private]` annotation.

4.1.8 V-TNC-VUL-008: Storage of unchecked String

Severity	Medium	**Commit**	1f30b00
Type	Data Validation	**Status**	Fixed
Files	`referrals.rs`		
Functions	`Contract::create_referral_code`		

The public function `VContract::create_referral_code` stores user-supplied argument `referral_code : String` (via invocation of `Contract::create_referral_code`) without checking the length of `referral_code` (e.g. as in `Contract::set_user_referral_code`).

```
#[near_bindgen]
impl VContract {

  #[payable]
    pub fn create_referral_code(&mut self, referral_code: String) {
        // ....
        contract.create_referral_code(env::predecessor_account_id(), referral_code);
                // .....
    }
}

impl Contract {
    pub fn create_referral_code(&mut self, account_id: AccountId, referral_code:
    String) {
        if referral_code.is_empty() {
            env::panic_str("Referral code length can not be 0");
        }
        if self
            .referral_code_owners
            .insert(&referral_code, &(account_id.clone(), ReferrerTier::Tier1))
            .is_some()
        {
            env::panic_str("Referral code already exists");
        }
                // ......
    }
}
```

Impact

- Malicious users can pay to force the contract to store arbitrarily long (up to transaction size limits) referral codes.
- Well-intentioned users may accidentally create a referral code which is unusable by `set_user_referral_code`.

Recommendation Add a `check_referral_code` function which enforces valid referral code lengths, and call it at the beginning of `Contract::set_user_referral_code` and `Contract::create_referral_code`.

4.1.9 V-TNC-VUL-009: Assumed collateral and underlying assets are the same in value calculation

Severity	Medium	Commit	1f30b00
Type	Logic Error	Status	Fixed
Files	perps/limit_order.rs		
Functions	Contract::get_collateral_in_usd		

In `get_collateral_in_usd`, the value of the collateral in USD (at the execution time) is determined by multiplying the collateral asset amount and the underlying price. This will only work as intended if the two are the same asset type.

```
fn get_collateral_in_usd(&self, limit_order: &LimitOrder) -> DollarBalance {
        if matches!(limit_order.order_type, OrderType::Decrease) {
            return limit_order.collateral_delta;
        }

        if limit_order.is_long {
            ratio(
                limit_order.attached_collateral,
                limit_order.price,
                self.assets
                    .unwrap(&limit_order.collateral_id)
                    .denomination(),
            )
        } else {
            limit_order.attached_collateral
        }
    }
```

Impact When issuing an order for a long position where the collateral and underlying assets are different, this check may rule out valid limit orders.

Recommendation Do appropriate conversions to ensure that the check computes the value in USD of the collateral at the execution time.

4.1.10 V-TNC-VUL-010: Lost funds on cancelled limit orders

Severity	Medium	Commit	1f30b00
Type	Logic Error	Status	Fixed
Files	perps/limit_order.rs		
Functions	Contract::update_limit_orders		

In `update_limit_orders`, limit orders that are not valid are removed from the set of limit orders. In the case that a limit order is for increasing a position, the user attaches collateral to the limit order. However, when such a limit order is removed in this function, the attached collateral is never returned to the user.

```
pub fn update_limit_orders(&mut self, account_id: &AccountId, position: &Position) {
        if let Some(user_orders) = self.limit_order_ids_map.get(account_id) {
            let underlying_id = position.underlying_id.clone().into();
            let underlying = self.assets.unwrap(&underlying_id);
            let limit_orders = if let Some(limit_orders) = self.limit_orders.get(&
    underlying_id) {
                limit_orders
            } else {
                return;
            };

            for (limit_order_id, asset_id) in user_orders {
                if asset_id != underlying_id {
                    continue;
                }

                let limit_order = limit_orders.get_by_id(&limit_order_id).unwrap();

                if limit_order.collateral_id == position.collateral_id.clone().into()
                    && limit_order.is_long == position.is_long
                {
                                        ...
                    // Remove limit orders that are not valid.
                    self.remove_limit_order(
                        &position.account_id,
                        &limit_order_id,
                        RemoveOrderReason::Invalid,
                    );
                }
            }
        }
    }
```

Impact Users can lose their funds if certain limit orders are removed.

Recommendation Check the output of `remove_limit_order` which returns a users collateral when necessary.

4.1.11 V-TNC-VUL-011: Users liquidated when below minimum leverage

Severity	Medium	**Commit**	1f30b00
Type	Logic Error	**Status**	Acknowledged
Files	`perps/mod.rs`		
Functions	`Contract::liquidate_position`		

In liquidate position, a user's position can be liquidated and their funds lost if they are deemed to have below the minimum allowed leverage. Being below this threshold means they have too much collateral backing their position — this does not seem like a reason to liquidate someone's account.

Impact Users who have provided plenty of collateral for their positions could be liquidated.

Recommendation Only liquidate for accounts that are over-leveraged.

4.1.12 V-TNC-VUL-012: set_user_referral_code uses predecessor instead of signer

Severity	Low	**Commit**	1f30b00
Type	Logic Error	**Status**	Fixed
Files	`referrals.rs`		
Functions	`VContract::set_user_referral_code`		

`VContract::set_user_referral_code` assigns the referral code to the predecessor, rather than the signer.

```
impl VContract {
    #[payable]
    pub fn set_user_referral_code(&mut self, referral_code: String) {
        let contract = self.contract_mut();
        contract.assert_running();
        contract.set_user_referral_code(env::predecessor_account_id(), referral_code)
    ;

        // ....
    }
}
```

In the case of actions initiated using attached FT tokens (e.g. a `swap_and_send` initiated via `ft_on_transfer`), the FT token contract will receive the referral code instead of the user who initiated the sequence with an `ft_transfer_call`[†].

Impact A user invoking an `Action` with an attached referral code via `someFT::ft_transfer_call` will not cause the contract to set the referral code for the user. Instead, the contract will set the referral code for `someFT`.

Recommendation Use `env::signer_account_id()` in place of `env::predecessor_account_id()`.

[†] NEP-141: Fungible Token standard

4.1.13 V-TNC-VUL-013: Stale state before external call

Severity	Low	Commit	1f30b00
Type	Logic Error	Status	Acknowledged
Files	lp_token/ft.rs		
Functions	Contract::ft_transfer_call		

The call to `self.internal_transfer(&sender_id, &receiver_id, amount, memo)` is processed before it is known that the call will be successful.

```
fn ft_transfer_call(
        &mut self,
        receiver_id: AccountId,
        amount: U128,
        memo: Option<String>,
        msg: String,
    ) -> PromiseOrValue<U128> {
                ...
        self.internal_transfer(&sender_id, &receiver_id, amount, memo);
        // Initiating receiver's call and the callback
        ext_ft_receiver::ext(receiver_id.clone())
            .with_static_gas(env::prepaid_gas() - GAS_FOR_FT_TRANSFER_CALL)
            .ft_on_transfer(sender_id.clone(), amount.into(), msg)
            .then(
                ext_ft_resolver::ext(env::current_account_id())
                    .with_static_gas(GAS_FOR_RESOLVE_TRANSFER)
                    .ft_resolve_transfer(sender_id, receiver_id, amount.into()),
            )
            .into()
    }
```

The callback function will only return funds to the sender if the receiver has enough funds to cover the rebate.

```
pub fn internal_ft_resolve_transfer(
        &mut self,
        sender_id: &AccountId,
        receiver_id: AccountId,
        amount: U128,
    ) -> u128 {
        let amount: Balance = amount.into();

        // Get the unused amount from the 'ft_on_transfer' call result.
        let unused_amount = match env::promise_result(0) {
            PromiseResult::NotReady => env::abort(),
            PromiseResult::Successful(value) => {
                if let Ok(unused_amount) = near_sdk::serde_json::from_slice::<U128>(&
    value) {
                    std::cmp::min(amount, unused_amount.0)
                } else {
                    amount
                }
            }
```



```
            PromiseResult::Failed => amount,
        };

        if unused_amount > 0 {
            let receiver_balance = self.accounts.get(&receiver_id).unwrap_or(0);
            if receiver_balance > 0 {
                let refund_amount = std::cmp::min(receiver_balance, unused_amount);
                self.internal_save_balance(&receiver_id, receiver_balance -
    refund_amount);

                let sender_balance = self.internal_unwrap_balance_of(sender_id);
                self.internal_save_balance(sender_id, sender_balance + refund_amount)
    ;

                FtTransfer {
                    old_owner_id: &receiver_id,
                    new_owner_id: sender_id,
                    amount: &U128(refund_amount),
                    memo: Some("refund"),
                }
                .emit();
                return amount - refund_amount;
            }
        }
        amount
    }
```

Impact This could be vulnerable to a front-running attack. In particular, if the receiver of the transaction is malicious and knows the transaction will fail (perhaps by monitoring transactions on chain), they could front-run a transaction that drains their funds (e.g., a swap or withdraw) to avoid having to pay back the sender of the original transfer. Because the sender is only refunded if the receiver has sufficient funds, their funds have effectively been stolen by the receiver.

Example Suppose Alice wants to send some LP token to a contract C which excepts bids for an Auction. However, Alice doesn't realize when she sends her LP token bid that the Auction is now closed. Alice's bid will eventually revert, but when she calls `ft_on_transfer`, the contract C will be granted the LP token. Because the auction is now closed, the auction owner withdraws their LP token, leaving their balance at 0. When Alice's bid fails, in the callback, the auction contract no longer has the balance to pay Alice back, so she is out her investment.

Recommendation In this case, waiting to update internal state until after the callback could also be unsafe, as it would allow the sender to perform a similar attack. This design pattern is borrowed from the example fungible token contracts released by NEAR; we are still in discussions with NEAR developers about the best way to resolve this issue. We encourage the developers to follow-up with NEAR dev team to best resolve this issue.

4.1.14 V-TNC-VUL-014: Short position checks total stable available liquidity instead of collateral available liquidity

Severity	Low	Commit	1f30b00
Type	Logic Error	Status	Fixed
Files	perps/mod.rs		
Functions	Contract::increase_position		

When increasing a position, some amount of collateral must be reserved (`reserve_delta`). Before recording `reserve_delta` as reserved inside the collateral asset, a check is performed to ensure the collateral has enough liquidity, copied below:

```
if is_long {
  if reserve_delta > collateral.available_liquidity() {
      env::panic_str("Not enough reserve to allow the long position");
  }
} else {
  let total_available_liquidity = self.total_stable_available_liquidity();
  if size_delta > total_available_liquidity {
      env::panic_str("Not enough reserve to allow the short position");
  }
}
```

In the short case, we check that there is enough liquidity **across all stable assets**, rather than just the collateral. This could lead to a state in which the reserve amount of collateral is larger than the balance.

This state is be exhibited in the following test case:

```
#[test]
fn test_open_short_position_multiple_stables() {
    let (mut context, mut vcontract) = setup();

    // Add new stable coin: USDT
    set_predecessor(&mut context, Admin);
    vcontract.add_asset("usdt".to_string(), 6, true, 50);

    // add $1000 liquidity split evenly between both stable coins USDC
    let total_liquid_usd = dollars(1000);
    let asset_id_names = ["usdt".to_string(), usdc_id()];
    let asset_ids: Vec<_> = asset_id_names.iter()
        .map(|name| AssetId::Ft(name.parse().unwrap()))
        .collect();
    for asset_id in &asset_ids {
        vcontract
            .contract_mut()
            .add_liquidity(asset_id, total_liquid_usd / asset_ids.len() as u128);
    }

    // add liquidity to NEAR
    vcontract
        .contract_mut()
        .add_liquidity(&AssetId::NEAR, near(100));
```

```
25
26      // Set stable coins to $1 and near to $5
27      update_near_price(&mut vcontract, dollars(5));
28      let update_requests = asset_id_names.iter()
29          .cloned()
30          .map(|asset_id| UpdateIndexPriceRequest{
31              asset_id,
32              price: U128::from(dollars(1)),
33              spread: None,
34          })
35          .collect();
36      vcontract.update_index_price(update_requests);
37
38      // Open a 4x leveraged position short NEAR - stable USDT
39      // Collateral = $200, size = $800
40      //
41      // This should cause an error! There is not enough usdt to cover the short
42      set_predecessor_token(&mut context, "usdt".to_string());
43      vcontract.ft_on_transfer(
44          get_account(Alice),
45          dollars(200).into(),
46          serde_json::to_string(&Action::IncreasePosition(IncreasePositionRequest {
47              underlying_id: near_id(),
48              size_delta: dollars(800).into(),
49              is_long: false,
50              referrer_id: None,
51          }))
52          .unwrap(),
53      );
54
55      // We can observe the error by comparing reserved amount to poool balance
56      let assets = vcontract.contract().get_assets();
57      let usdt = assets.get(&AssetId::Ft("usdt".parse().unwrap()))
58          .expect("Expected usdt");
59      assert!(usdt.reserved_amount <= usdt.pool_balance,
60          "Reserved amount is greater than pool balance ({} > {})",
61          usdt.reserved_amount, usdt.pool_balance
62      );
63  }
```

which errors with the following output:

```
1  running 1 test
2  test test_open_short_position_multiple_stables ... FAILED
3
4  failures:
5
6  ---- test_open_short_position_multiple_stables stdout ----
7  {"type":"OracleUpdate","data":{"asset_id":"usdc","price":"1000000","spread_bps":0,"
       source":"tonic"}}
8  {"type":"EditPoolBalance","data":{"amount_native":500000000,"new_pool_balance_native"
       :500000000,"increase":true,"account_id":"charlie","asset_id":"usdt"}}
9  {"type":"EditPoolBalance","data":{"amount_native":500000000,"new_pool_balance_native"
       :500000000,"increase":true,"account_id":"charlie","asset_id":"usdc"}}
```

```
{"type":"EditPoolBalance","data":{"amount_native":100000000000000000000000000,"
    new_pool_balance_native":100000000000000000000000000,"increase":true,"account_id"
    :"charlie","asset_id":"near"}}
{"type":"OracleUpdate","data":{"asset_id":"near","price":"5000000","spread_bps":0,"
    source":"tonic"}}
{"type":"OracleUpdate","data":{"asset_id":"usdt","price":"1000000","spread_bps":0,"
    source":"tonic"}}
{"type":"OracleUpdate","data":{"asset_id":"usdc","price":"1000000","spread_bps":0,"
    source":"tonic"}}
Position Size: 0, cum Fun rate: 480, entry rate: 0
{"type":"EditFees","data":{"fee_native":0,"fee_usd":0,"fee_type":"funding","
    new_accumulated_fees_native":0,"new_accumulated_fees_usd":0,"increase":true,"
    account_id":"charlie","asset_id":"usdt"}}
{"type":"EditFees","data":{"fee_native":0,"fee_usd":0,"fee_type":"position","
    new_accumulated_fees_native":0,"new_accumulated_fees_usd":0,"increase":true,"
    account_id":"charlie","asset_id":"usdt"}}
Position Size: 800000000, cum Fun rate: 480, entry rate: 480
{"type":"EditReservedAmount","data":{"amount_native":800000000,"
    new_reserved_amount_native":800000000,"increase":true,"account_id":"charlie","
    asset_id":"usdt"}}
{"type":"EditPosition","data":{"direction":"increase","account_id":"charlie","
    position_id":"DKpTpMoqf5pL8dTyCoNWgEvwxYgWYVsGExZCSkMMmaHz","collateral_token":"
    usdt","underlying_token":"near","collateral_delta_native":"200000000","
    collateral_delta_usd":"200000000","size_delta_usd":"800000000","new_size_usd":"
    800000000","is_long":false,"price_usd":"5000000","usd_out":"0","total_fee_usd":"0
    ","margin_fee_usd":"0","position_fee_usd":"0","total_fee_native":"0","
    margin_fee_native":"0","position_fee_native":"0","referral_code":null,"
    realized_pnl_to_date_usd":"0","adjusted_delta_usd":"0","state":"created","
    limit_order_id":null,"liquidator_id":null}}
{"type":"TokenDepositWithdraw","data":{"amount_native":"200000000","deposit":true,"
    method":"ft_on_transfer","receiver_id":"alice","account_id":"alice","asset_id":"
    usdt"}}
thread 'test_open_short_position_multiple_stables' panicked at 'Reserved amount is
    greater than pool balance (800000000 > 500000000)', crates/tonic-perps/tests/
    test_increase_position.rs:194:5
note: run with 'RUST_BACKTRACE=1' environment variable to display a backtrace

failures:
    test_open_short_position_multiple_stables

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured; 22 filtered out;
    finished in 0.00s
```

Impact Users taking out large shorts may be able to increase the reserve amount past the pool balance. In this case, the contract may not have enough funds to pay out the short. For instance, if all the USDC liquid is spent, and the short pays out maximally, the contract will be insolvent.

Note that, once this state is reached, it can be resolved by swapping from one stable coin into the "over-reserved" coin until the reserved amount is below the pool balance.

Recommendation Replace the check against `total_stable_liquidity` with a check against `collateral.available_liquidity()`.

4.1.15 V-TNC-VUL-015: Withdrawals from NEAR asset never check storage requirements

Severity	Low	**Commit**	1f30b00
Type	Denial of Service	**Status**	Won't Fix
Files	`perps/mod.rs`, `trading.rs`, `mint.rs`		
Functions	NA		

The liquidity pool relies on the pool balance of NEAR to cover storage costs. By repeatedly burning LP Tokens for NEAR, exchanging NEAR for some other asset on an external exchange, and then minting LP Tokens with the new asset, an attacker could greatly reduce the pool's NEAR balance before launching some sort of storage-taking denial of service attack. See, for example, V-TNC-VUL-014 and V-TNC-VUL-015.

Impact If attackers drain the NEAR pool balance, the entire pool becomes more vulnerable to storage-taking attacks.

Recommendation Check storage usage before allowing a withdrawal from NEAR. If the pool comes close to its storage limit, take action to either boost the NEAR supply or reduce storage.

4.1.16 V-TNC-VUL-016: Storage Taking attack: transferring LP Tokens to bogus accounts

Severity	Low	**Commit**	1f30b00
Type	Denial of Service	**Status**	Fixed
Files	`lp_token/ft.rs`		
Functions	`Contract::ft_transfer`		

Based on our profiling, performing an `ft_transfer` to a bogus account costs more to the contract (via increased storage) than it does to the attacker.

```
Attack: LP Token FT Transfer Attack
           Gas:    0.6294453907627 TGas
   Atk Deposit: 0.000000000000000000001 mNear
      Atk Cost:   0.06294453907627 mNear
       Storage:                  75 bytes
  Storage Cost:               0.075 mNear
    Attack Eff: 119.15251283216544 %
```

Impact A dedicated enough adversary may perform a denial of service attack by minting LP tokens, then distributing out the token into its 10^18 parts amongst 10^18 bogus accounts (or however many bogus accounts are required to shut down the contract).

Recommendation Require account minimums close to the cost of this attack, or reduce the number of decimals in LP Token.

4.1.17 V-TNC-VUL-017: Malicious set user referral code induces large storage cost

Severity	Low	Commit	1f30b00
Type	Denial of Service	Status	Fixed
Files	`referrals.rs`		
Functions	`VContract::set_user_referral_code`		

Based on our profiling, performing a `set_user_referral_code` costs more to the contract (via increased storage) than it does to the attacker.

```
Attack: Set User Referral Code Length: 32
 Sybil Acc Gas:        0.180657225 TGas // Gas to make new acct
           Gas:    0.6265628230021 TGas
   Atk Deposit:                  0 mNear
      Atk Cost:   0.08072200480021 mNear
       Storage:                264 bytes
  Storage Cost:              0.264 mNear
    Attack Eff: 327.04836884739166 %
```

Impact Any attacker may create a new account, then set a referral code for that user to incur a large storage cost to the client. This attacker needs only 1/3 the pool's NEAR balance in order to be successful.

Recommendation Check that user referral tokens are valid and that they are only assigned to a limited number of users at any given time. The cost to create a referral code is then amortized across the calls to set the user referral code.

4.1.18 V-TNC-VUL-018: ft_on_transfer uses signer account rather than sender

Severity	Low	**Commit**	1f30b00
Type	Logic Error	**Status**	Fixed
Files	`token_receiver.rs`		
Functions	`Contract::ft_on_transfer`		

The `Swap` and `MintLp` actions use parameter `sender_id` as the relevant action initiator. However, `IncreasePosition` and `PlaceLimitOrder` use the signer account ID.

Based on the description of `ft_resolve_transfer` in the NEP-141 standard, the `sender_id` should be treated as the initiator of the action.

Impact A long sequence of cross-contract calls ending in a call to `ft_transfer_call` on perps could lead to the incorrect account being assigned ownership of a limit order or position.

Recommendation Use the `sender_id` in place of `env::signer_account_id()`.

4.1.19 V-TNC-VUL-019: Possible limit order ID collisions

Severity	Low	**Commit**	1f30b00
Type	Logic Error	**Status**	Acknowledged
Files	perps/limit_order_id.rs		
Functions	new		

When creating an ID for a new limit order, it is assumed the price (which is set by the user) is 64 bits. However, the `limit_order.price` field is a `u128`, meaning it can contain `128` bits. Thus, there two limit orders with different prices could have the same ID.

```
const SEQUENCE_MASK: u128 = (1u128 << 62) - 1;

// 64 bits starting at the second bit
const PRICE_MASK: u128 = ((1u128 << 126) - 1) - ((1u128 << 62) - 1);

impl LimitOrderId {
    pub fn new(limit_order: &LimitOrder, seq_number: u64) -> LimitOrderId {
        let first_bit = if limit_order.is_long { 0 } else { 1u128 << 127 };
        let second_bit = if matches!(limit_order.threshold, ThresholdType::Below) {
            0
        } else {
            1u128 << 126
        };

        let seq = SEQUENCE_MASK & (seq_number as u128);

        LimitOrderId(first_bit | second_bit | ((limit_order.price << 62) & PRICE_MASK
    ) | seq)
    }

        ...
}
```

Impact In the code, it is assumed every limit order for a given user and underlying asset type has a unique ID. Thus, in the event of a collision, a limit order could be dropped and attached collateral lost.

Recommendation Change the type of `limit_order.price` to `u64`.

4.1.20 V-TNC-VUL-020: Pool can lose money on liquidation

Severity	Low	**Commit**	1f30b00
Type	Logic Error	**Status**	Fixed
Files	`perps/mod.rs`		
Functions	`Contract::liquidate_position`		

The liquidation reward is a flat fee of 25USD. See, e.g. its usage in `liquidate_position`, and its definition in `VContract::new`.

Since the contract must pay out the fee to the liquidator, the contract will lose money when the liquidating collateral is less than the reward.

Impact An adversarial liquidator could consistently take money from the pool by opening a small short position which can be liquidated before its losses exceed 25USD (or whatever the reward is). Then, the adversary can hedge their short position by liquidating as soon as the position is insolvent, gaining a profit from the pool.

Recommendation Make the liquidation fee a percentage of the liquidated collateral.

4.1.21 V-TNC-VUL-021: Confusing function usage

Severity	Warning	**Commit**	1f30b00
Type	Maintainability	**Status**	Acknowledged
Files	`trading.rs`		
Functions	`Contract::swap`		

In `swap`, a call is made to the function `convert_assets` as follows:

```
let amount_out = {
            convert_assets(
                amount_in,
                asset_in.min_price(),
                asset_out.denomination(),
                asset_out.max_price(),
                asset_in.denomination(),
            )
        };
```

The implementation of this function is the following:

```
pub fn convert_assets(
    amount_in: Balance,
    num_1: u128,
    num_2: u128,
    denom_1: u128,
    denom_2: u128,
) -> u128 {
    let num = BN!(num_1).mul(num_2).as_u128();
    let denom = BN!(denom_1).mul(denom_2).as_u128();
    ratio(amount_in, num, denom)
}
```

The 3rd and 4th arguments are referred to as `denom_1` and `denom_2` which we believe is meant to refer to the fact they are used as "denominators". However, because the function is often passed "denominations" (such as the call-site form `swap` pictured above), we suspect future developer s may confuse the argument ordering leading to errors.

Impact If developers confuse these arguments in the future, it could lead to drastically incorrect calculations which could have wide-ranging consequences.

Recommendation Rename arguments and improve documentation for `convert_assets` to clarify what the function does and the intended arguments.

4.1.22 V-TNC-VUL-022: Multiple functions made payable unnecessarily

Severity	Warning	**Commit**	1f30b00
Type	Logic Error	**Status**	Fixed
Files	`perps/mod.rs`, `referrals.rs`		
Functions	multiple		

The functions `decrease_position`, `liquidate_position`, `remove_limit_order`, `execute_limit_order`, and `remove_outdated_limit_order` in `perps/mod.rs` are marked as `#[payable]` but it is not clear why / if it is necessary. Similarly, `set_user_referral_code` in `referrals.rs` is `#[payable]` but does not need to be.

Impact Users can unnecessarily send NEAR to these functions and lose their money with no benefit.

Recommendation Remove the `#[payable]` annotations.

4.1.23 V-TNC-VUL-023: is_liquidator uses unmodified field Contract::liquidators

Severity	Warning	**Commit**	1f30b00
Type	Logic Error	**Status**	Fixed
Files	`views.rs`, `lib.rs`		
Functions	`VContract::is_liquidator`		

The field `Contract::liquidators` is never modified. The only time it is used is in `VContract::is_liquidator`, which will always return false.

Another list of liquidators is maintained in the `Contract::admins` field, which is used elsewhere in the codebase.

Impact The view function `VContract::is_liquidator` may produce incorrect results.

Recommendation Remove the field `Contract::liquidators` and alter all uses of `liquidators` to check the `Contract::admins` field using the appropriate functions (e.g. `Contract::check_admin_role`).

The `Contract::is_admin` function might also want to check that the role is `AdminRole::FullAdmin` in order to be consistent with `Contract::assert_admin`.

4.1.24 V-TNC-VUL-24: Out-of-date class documentation for LimitOrderID

Severity	Warning	**Commit**	1f30b00
Type	Maintainability	**Status**	Fixed
Files	`perps/limit_order_id.rs`		
Functions	`struct LimitOrderId`		

The documentation of `LimitOrderID` does not match the implementation. The documentation describes

- 1 bit for long or short.
- 64 bits for price.
- 63 bits for sequence number.

However, the implemented representation uses

- 1 bit for long or short.
- **1 bit for threshold type.**
- 64 bits for price.
- **62** bits for sequence number.

Impact Future developers may be confused about the implemented layout, and misuse bit patterns.

Recommendation Update the documentation to reflect the changes made to `LimitOrderId`.

4.1.25 V-TNC-VUL-025: Out-of-date function documentation for get_lp_redemption_amount

Severity	Warning	**Commit**	1f30b00
Type	Maintainability	**Status**	Fixed
Files	`lp_token/mint.rs`		
Functions	`Contract::get_lp_redemption_amount`		

The documentation states that `get_lp_redemption_amount`

```
1 /// Returns [Err] when pool liquidity is insufficient to honor the redemption.
```

However, this is not the case. The function will only error in cases of over/underflow in the multiplied arguments.

Impact Future library developers may rely on the method throwing an error when the liquidity is insufficient to honor the redemption.

Recommendation Remove this line from the documentation.

4.1.26 V-TNC-VUL-026: Add limit order does not check if NEAR sent on decrease

Severity	Warning	**Commit**	1f30b00
Type	Logic Error	**Status**	Invalid
Files	`perps/mod.rs`		
Functions	`Contract::add_limit_order`		

The function `add_limit_order` is required to be `#[payable]` for adding Increase orders. However, on Decrease orders, any attached NEAR will just be lost.

Impact Users can unnecessarily send NEAR to this functions and lose their money with no benefit.

Recommendation Add an assertion that no NEAR are attached when Decrease orders are sent.

4.1.27 V-TNC-VUL-027: Referral code creation may charge more than CREATE_REFERRER_FEE

Severity	Info	Commit	1f30b00
Type	Usability Issue	Status	Fixed
Files	`referrals.rs`		
Functions	`VContract::create_referral_code`		

`VContract::create_referral_code` may charge the entire attached amount rather than the `CREATE_REFERRER_FEE`.

Impact If this is the intended behavior, it is not an issue.

Otherwise, users who send much more than the required 0.05 NEAR may be upset if the excess amount is not refunded.

Recommendation If the attached deposit is much larger than the `CREATE_REFERRER_FEE`, return excess attached NEAR to the predecessor.

4.1.28 V-TNC-VUL-028: Minimum amount out in swap is checked before fees

Severity	Info	Commit	1f30b00
Type	Usability Issue	Status	Intended Behavior
Files	trading.rs		
Functions	Contract::swap		

`Contract::swap` checks that `amount_out` is at least `min_amount_out` before fees have been deducted.

```
impl Contract {
    fn swap(..., min_amount_out: Option<Balance>, ...) -> Balance {
        // ...
        if let Some(min_amount_out) = min_amount_out {
        assert!(amount_out >= min_amount_out, "Exceeded slippage tolerance");
    }
        let (after_fee_amount, fees, swap_fee_bps) = // ...
        // ....

        return after_fee_amount;
    }
}
```

Impact If this is not the intended interpretation of `min_amount_out`, then users may receive less from a swap than they desired.

If this is the intended interpretation, then there is not an issue.

Recommendation If `min_amount_out` is intended to refer to the final after fee amount, check that `after_fee_amount >= min_amount_out` in place of checking `amount_out >= min_amount_out`.

4.1.29 V-TNC-VUL-029: Redundant function call

Severity	Info	**Commit**	1f30b00
Type	Gas Optimization	**Status**	Fixed
Files	`lp_token/mod.rs`		
Functions	`FungibleTokenFreeStorage::internal_withdraw`		

The call to `self.accounts.insert(account_id, &new_balance)` in `internal_withdraw` is redundant as this is performed in `self.internal_save_balance(account_id, new_balance)`.

```
pub fn internal_save_balance(&mut self, account_id: &AccountId, balance: Balance) {
        if balance > 0 {
            self.accounts.insert(account_id, &balance);
        } else {
            self.accounts.remove(account_id);
        }
    }
...
pub fn internal_withdraw(&mut self, account_id: &AccountId, amount: Balance) {
        let balance = self.internal_unwrap_balance_of(account_id);
        if let Some(new_balance) = balance.checked_sub(amount) {
            self.accounts.insert(account_id, &new_balance);
            self.internal_save_balance(account_id, new_balance);
            self.total_supply = self
                .total_supply
                .checked_sub(amount)
                .unwrap_or_else(|| env::panic_str("Total supply overflow"));
        } else {
            env::panic_str("The account doesn't have enough balance");
        }
    }
```

Impact This extra call increases gas costs for users unnecessarily.

Recommendation Remove the redundant call.

4.1.30 V-TNC-VUL-030: Incorrect type annotation

Severity	Info	**Commit**	src/lp_token/mint.rs
Type	Maintainability	**Status**	Fixed
Files	`lp_token/mint.rs`		
Functions	`get_lp_mint_amount`		

The `after_fee_argument` passed to `get_lp_mint_amount` is listed as type `DollarBalance`, but actual parameters and usages are of type `Balance` (both `Balance` and `DollarBalance` are aliased to type `U128`).

An example usage can be found here in `mint_lp_token`.

Impact Maintainers and new developers may be slightly confused about the usage of this function.

Recommendation Change `after_fee_amount` from type `DollarBalance` to type `Balance`.

Refactoring to use Tuple Structs without Named Fields could prevent similar errors at the cost of some additional code. See V-TNC-VUL-027.

4.1.31 V-TNC-VUL-031: Code structure suggestion: use Rust Tuple Structs to track currency unit types

Severity	Info	**Commit**	1f30b00
Type	Maintainability	**Status**	Open
Files		NA	
Functions		NA	

While auditing the code, our auditors found that a common source of error and confusion in the code has to do with the interchanging of various currency types (i.e., NEAR vs. USDC vs. underlying asset). Right now, the developers distinguish these values by using some common naming schemes, such as using "_native" to indicate NEAR. We suggest that the developers instead use Rust's Tuple Structs‡ to differentiate these types, which would enable Rust to automatically rule out certain mistakes. We believe this will make the current code more robust and will significantly reduce the risk of introducing errors in future generations of the code.

‡ See also the New Type Idiom

4.1.32 V-TNC-VUL-032: Code structure suggestion: check contract invariants first

Severity	Info	**Commit**	1f30b00
Type	Maintainability	**Status**	Fixed
Files		NA	
Functions		NA	

While auditing the code, our auditors found that tracking the safety of certain portions of the code was made significantly more challenging by virtue of the fact that parameter validation and contract invariant checking was often performed throughout the computation, as opposed to only at the beginning. As an example, validation for adding multiple limit orders of the same type for a current position occurs at the end of the `add_limit_order` function, while most of the checking occurs before this in `check_limit_order`.[§]

[§] See also: the Checks-Effects-Interactions Pattern from Solidity

4.1.33 V-TNC-VUL-033: Code structure suggestion: split logic for short and long positions

Severity	Info	**Commit**	1f30b00
Type	Maintainability	**Status**	Won't Fix
Files		NA	
Functions		NA	

While auditing the code, our auditors found that understanding the logic for long and short positions was made significantly more challenging by the fact that a single function handles increasing/decreasing both longs and shorts. To make this more understandable now, and to avoid issues in the future when updating the code, we suggest splitting this logic into separate functions for longs and shorts, abstracting common logic into a separate function.

4.1.34 V-TNC-VUL-034: Add in logging for internal transfer failure

Severity	Info	**Commit**	1f30b00
Type	Logic Error	**Status**	Fixed
Files	`lp_token/mod.rs`		
Functions	NA		

In the event that an `internal_transfer` fails, there is currently no callback used to track failures. As stated in V-TNC-VUL-004, we suggest the users provide callback logic here to avoid potential risks involved with failure of this call. However, developer's have suggested such failures will be avoided with frontend screening. If this is the case, we at least suggest developers add a callback to log failed transfers such that failures could be rectified manually by administrators as warranted.

4.1.35 V-TNC-VUL-035: Use #[must_use] for any functions which return balances that must be sent to a user

Severity	Info	**Commit**	1f30b00
Type	Maintainability	**Status**	Fixed
Files		NA	
Functions		NA	

Some functions in the protocol return a balance that is expected to be sent to a user. We found one bug (V-TNC-VUL-010) which happened because such a return value was ignored. Rust allows the use of `#[must_use]`, which can statically determine if such a bug occurs. Thus, we suggest any function which has a return value indicating a balance that should be sent (or really any function whose return value should be used) should be annotated with `#[must_use]`.

4.1.36 V-TNC-VUL-036: Replace complicated limit order merge logic

Severity	Info	**Commit**	1f30b00
Type	Maintainability	**Status**	Won't Fix
Files	`perps/limit_order.rs`		
Functions	`Contract::add_limit_order`		

When multiple limit orders of the same type with the same underlying/collateral asset types are submitted, they are merged to save storage. When existing limit orders are fetched, they are fetched in multiple stages. First, the limit orders for the current underlying asset are fetched. Then, all limit orders from on that underlying asset with a matching price, threshold type, and long vs. short setting are fetched from a B-tree which is presumably used for efficient lookups. Finally, these are filtered by owner and order type to find the appropriate limit order to merge (if there is any). We believe this process is somewhat confusing and error prone (we found one critical limit order merging issue in our audit). While this approach could offer some modest efficiency improvements, we believe that it could lead to bugs in the future.

Suggestion To reduce the likelihood of bugs being introduced in the future, we suggest that all fields relevant to merging be included in the computation of limit order ids, and ids be used as the mechanism for detecting limit order which should be merged.