verichains

SECURITY AUDIT OF

DRADEX SMART CONTRACT

DRADEX

Public Report

Aug 09, 2022

Verichains Lab

info@verichains.io

https://www.verichains.io

Driving Technology > Forward

ABBREVIATIONS

Name	Description
Solana	A decentralized blockchain built to enable scalable, user-friendly apps for the world.
SOL	A cryptocurrency whose blockchain is generated by the Solana platform.
Lamport	A fractional native token with the value of 0.000000001 sol.
Program	An app interacts with a Solana cluster by sending it transactions with one or more instructions. The Solana runtime passes those instructions to program.
Instruction	The smallest contiguous unit of execution logic in a program.
Cross-program invocation (CPI)	A call from one smart contract program to another
Anchor	A framework for Solana's Sealevel runtime providing several convenient developer tools for writing smart contracts.
DAO	A digital Decentralized Autonomous Organization and a form of investor-directed venture capital fund.

EXECUTIVE SUMMARY

This Security Audit Report prepared by Verichains Lab on Aug 09, 2022. We would like to thank the Dradex for trusting Verichains Lab in auditing smart contracts. Delivering high-quality audits is always our top priority.

This audit focused on identifying security flaws in code and the design of the Dradex Smart Contract. The scope of the audit is limited to the source code files provided to Verichains. Verichains Lab completed the assessment using manual, static, and dynamic analysis techniques.

During the audit process, the audit team had identified some vulnerable issues in the smart contracts code.

Dradex team fixed the code, according to Verichains's private report.

TABLE OF CONTENTS

1. MANAGEMENT SUMMARY

1.1. About Dradex Smart Contract

Dradex introduces unified Order Book & AMM model, fully integrated to facilitate trades with minimum slippage. Their algorithm automatically executes orders at the best price from a seamless combination of either the Order Book or the AMM Liquidity Pool.

1.2. Audit scope

This audit focused on identifying security flaws in code and the design of the Dradex Smart Contract. It was conducted on the source code provided by the Dradex team.

The following files were made available in the course of the review:

SHA256 Sum	File
2d4d0cebb379592acfe9a7f622ae690abe22284389c3ccc20426118e0cf88d0a	dexpow.zip
83440f4d0e84a5ce49fc016e0ea140a049e0cf8b75281a054fd4d00264baa00c	Dradex_Whitepaper_audit.pdf

1.3. Audit methodology

Our security audit process for Solana smart contract includes two steps:

- Smart contract codes are scanned/tested for commonly known and more specific vulnerabilities using our in-house smart contract security analysis tool.
- Manual audit of the codes for security issues. The contracts are manually analyzed to look for any potential problems.

Following is the list of commonly known vulnerabilities that was considered during the audit of the Solana smart contract:

- Arithmetic Overflow and Underflow
- Signer checks
- Ownership checks
- Rent exemption checks
- Account confusions
- Bump seed canonicalization
- Closing account
- Signed invocation of unverified programs
- Numerical precision errors
- Logic Flaws

For vulnerabilities, we categorize the findings into categories as listed in table below, depending on their severity level:

SEVERITY LEVEL	DESCRIPTION
CRITICAL	A vulnerability that can disrupt the contract functioning; creates a critical risk to the contract; required to be fixed immediately.
HIGH	A vulnerability that could affect the desired outcome of executing the contract with high impact; needs to be fixed with high priority.
MEDIUM	A vulnerability that could affect the desired outcome of executing the contract with medium impact in a specific scenario; needs to be fixed.
LOW	An issue that does not have a significant impact, can be considered as less important.

Table 1. Severity levels

1.4. Disclaimer

Please note that security auditing cannot uncover all existing vulnerabilities, and even an audit in which no vulnerabilities are found is not a guarantee for a 100% secure smart contract. However, auditing allows discovering vulnerabilities that were unobserved, overlooked during development and areas where additional security measures are necessary.

2. AUDIT RESULT

2.1. Overview

The Dradex Smart Contract was written in `Rust` programming language and `Anchor` framework.

It provides a decentralized exchange where users can create markets, liquidity pools and farms for their pools, make limit/market orders with unified liquidity pool and order book. Each order will be charged a percentage amount of fees, the majority of fees are deposited into the Liquidity Pool to reward Liquidity Providers, the remaining are sent to the DAO fund which can be collected by DAO fund manager (set by the program owner).

2.2. Findings

During the audit process, the audit team found some vulnerabilities in the given version of Dradex Smart Contract.

Dradex team fixed the code, according to Verichains's private report.

2.2.1. Liquidity can be added below `MINIMUM_LIQUIDITY` MEDIUM

`add_liquidity` function requires the deposited liquidity must be greater than the `MINIMUM_LIQUIDITY` but anyone can bypass this requirement by `add_liquidity` enough liquidity then using `remove_liquidity` to reduce the liquidity to below the `MINIMUM_LIQUIDITY`.

```
pub fn add_liquidity(
    ctx: Context<LPOperation>,
    t0_amount: u64,
    t1_amount: u64,
) -> ProgramResult {
    ...
    if is_new_pool {
        liquidity = u64::try_from(
            sqrt((t0_amount as u128).checked_mul(t1_amount as u128).unwrap()).unwrap(),
        )
            .unwrap();
        require!(liquidity > MINIMUM_LIQUIDITY, InsufficientLiquidityAdded);
        liquidity = liquidity.checked_sub(MINIMUM_LIQUIDITY).unwrap()
    } else {
        t1_amount = (t0_amount as u128).fmul64(U64Frac(pool[1], pool[0]));
        liquidity = u64::try_from(
            (t0_amount as u128)
                .checked_mul(lp_token_total)
                .unwrap()
                .checked_div(pool[0] as u128)
                .unwrap(),
        )
```

```
            .unwrap();
    }
    ...
}

pub fn remove_liquidity(ctx: Context<LPOperation>, amount: u64) -> ProgramResult {
    let mut logger = new_logger!(ctx);
    let lp_token_total = ctx.accounts.lp_token_mint.supply;
    let pool = ctx.accounts.market.pool;
    let pool_share = U64Frac(amount, lp_token_total);

    // burn the LP tokens
    token::burn(ctx.accounts.into_lp_token_burn_context(), amount)?;

    // withdraw tokens from the pool
    let t0_amount = (pool[0] as u128).fmul64(pool_share);
    let t1_amount = (pool[1] as u128).fmul64(pool_share);
    require!(t0_amount > 0 && t1_amount > 0, InsufficientLiquidityRemoved);
    ctx.accounts.market.pool = [
        pool[0].checked_sub(t0_amount).unwrap(),
        pool[1].checked_sub(t1_amount).unwrap(),
    ];
    token::transfer(
        ctx.accounts
            .into_t0_withdraw_context()
            .with_signer(&[&MASTER_SEEDS[..]]),
        t0_amount,
    )?;
    token::transfer(
        ctx.accounts
            .into_t1_withdraw_context()
            .with_signer(&[&MASTER_SEEDS[..]]),
        t1_amount,
    )?;
    log!(
            logger,
            "pool {:?} {:?}",
            ctx.accounts.market.pool[0],
            ctx.accounts.market.pool[1]
        );
    log!(logger, "lp_supply {:?}", lp_token_total - amount);
    logger.flush()
}
```

RECOMMENDATION

Permanently lock the first `MINIMUM_LIQUIDITY` like other DEXs (uniswap, pancakeswap, ...) to prevent liquidity to be drained to below `MINIMUM_LIQUIDITY`.

UPDATES

- *July 01, 2022*: This issue has been acknowledged and fixed by the Dradex team.

2.2.2. Crank rewards stuck in pair when `settle_funds` MEDIUM

When `settle_funds`, users process the event queue and remove their orders from the queue but the crank reward is not distributed like in `consume_events` so the SOL charged when kicking orders will be stuck in the pair.

```
pub fn settle_funds(ctx: Context<MarketOperation>) -> ProgramResult {
   ...
    // apply events in the queue
    let mut event_queue = ctx.accounts.event_queue.load_mut()?;
    let mut queue_updated = false;
    let mut processed_list = Vec::<String>::with_capacity(event_queue.len());
    for item in event_queue
        .items()
        .iter()
        .filter(|item| item.owner == owner)
    {
        if item.is_cancel() {
            market_user.order_cancelled(item.side(), item.price().get(), item.quantity);
        }
        queue_updated = true;
        processed_list.push(item.key.to_string());
    }
    if queue_updated {
        event_queue.retain(|item| item.owner != owner);
        log!(logger, "rm {}", processed_list.join(" "));
    }
    ...
}

pub fn consume_events(ctx: Context<ConsumeEvents>, input: ConsumeEventsInput) ->
ProgramResult {
    let mut logger = new_logger!(ctx);
    let mut event_queue = ctx.accounts.event_queue.load_mut()?;
    let limit = input.limit.unwrap_or(u32::MAX);
    let mut count = 0u32;
    let mut removed_ids = HashSet::<u128>::new();
    for item in event_queue.items() {
        if count >= limit {
            break;
        }
        let result = ctx
            .remaining_accounts
            .binary_search_by_key(&item.owner, |account| account.key().to_aligned_bytes());
        if let Ok(index) = result {
            let info = &ctx.remaining_accounts[index];
```

```
            let mut user = Account::<'_, MarketUser>::try_from(info).unwrap();
            if item.is_cancel() {
                user.order_cancelled(item.side(), item.price().get(), item.quantity);
            }
            let mut buf = info.try_borrow_mut_data().unwrap();
            let mut cursor = std::io::Cursor::new(buf.as_mut());
            user.try_serialize(&mut cursor)?;
            count += 1;
            removed_ids.insert(item.key);
            log!(
                    logger,
                    "market_user {:?} {:?} {:?} {:?} {:?}",
                    info.key,
                    user.t0_pending,
                    user.t1_pending,
                    user.t0_unlocked,
                    user.t1_unlocked
                );
        }
    }

    if count > 0 {
        log!(
                logger,
                "rm {}",
                removed_ids
                    .iter()
                    .map(|key| key.to_string())
                    .collect::<Vec<String>>()
                    .join(" ")
            );
        // save queue
        event_queue.retain(|item| !removed_ids.contains(&item.key));

        // claim crank rewards
        let vault_account = ctx.accounts.pair.to_account_info();
        let mut vault_lamports = vault_account.lamports.as_ref().borrow_mut();
        let vault_min = ctx.accounts.rent.minimum_balance(vault_account.data_len());
        if vault_lamports.clone() > vault_min {
            let reward_max = vault_lamports.clone() - vault_min;
            let reward_amount = (ctx.accounts.master.crank_penalty as u64)
                .checked_mul(count as u64)
                .unwrap()
                .min(reward_max);
            if reward_amount > 0 {
                let mut signer_lamports =
ctx.accounts.signer.lamports.as_ref().borrow_mut();
                **signer_lamports =
signer_lamports.clone().checked_add(reward_amount).unwrap();
                **vault_lamports = vault_lamports.clone() - reward_amount;
                log!(logger, "crank_reward {:?}", reward_amount);
```

```
            }
        }
    }
    logger.flush()
}
```

RECOMMENDATION

Sending crank reward to user when `settle_funds`.

UPDATES

- *Aug 09, 2022*: This issue has been acknowledged and fixed by the Dradex team.

2.2.3. Users can kick order in worst group with better price than their price MEDIUM

When `create_order`, if the order book is full, an order in worst group may be removed from the order book to make room for new order if the price of the new order is better.

The current implementation only check that the price of the new order is better than the first one in worst group, so users can kick out any orders in the worst group by sending only the owner of targeted order in `wo_users` (`remaining_accounts`). The targeted order will be kicked out even though the price of that order is better than the new order.

```
pub fn new_order<'info, 'a, 'b, 'c>(
    &'_ mut self,
    params: OrderParams,
    logger: &mut Logger,
    market_user: &'_ mut Box<Account<'info, MarketUser>>,
    wo_users: &'_ [AccountInfo<'c>],
) -> Result<OrderResult, ProgramError> {
    ...
    if ob.is_full() {
        // if order book is full, we have to kick out another order
        let mut removed_id: u128;
        let mut wo_user: Option<&AccountInfo<'c>> = None;
        let mut wo_owner = market_user.key();
        {
            // any of the order in the worst group can be kicked
            let removables = ob
                .iter()
                .rev()
                .take(NUM_REMOVABLE_ORDERS)
                .collect::<Vec<&Order>>();
            let wo = removables[0];
            removed_id = wo.order_id();
            require!(
                        side.is_better_or_bigger(
                            new_order.price().get(),
```

```
                                wo.price().get(),
                                new_order.quantity(),
                                wo.quantity()
                            ),
                            OrderbookFull
                        );

            // select a bottom order to kick
            msg!("orders full! booting out one of the worst orders...");
            for removable in removables {
                wo_owner = cast(removable.owner());
                msg!("wo_owner {:?}", wo_owner);
                wo_user = wo_users.iter().find(|u| u.key() == wo_owner);
                if wo_user.is_some() {
                    removed_id = removable.order_id();
                    break;
                }
            }
        }
        ...
    }
    ...
}
```

RECOMMENDATION

Checking if the price of the new order is better than order in `removables` before kicking.

UPDATES

- *Aug 09, 2022*: This issue has been acknowledged and fixed by the Dradex team.

2.2.4. Duplicated code INFORMATIVE

`find_insert_index` and `find_item_index` function can share the logic code to avoid duplication.

```
fn find_insert_index(&self, k: u128, descending: bool) -> usize {
    let ObPartsMut { header, items } = self;
    let mut lo: i64 = 0;
    let mut hi: i64 = header.len as i64 - 1;
    while lo <= hi {
        let mid = (hi + lo) / 2;
        let current = &items[mid as usize];
        let id = current.order_id();
        if id == k {
            return mid as usize;
        }
        if (id > k) != descending {
            hi = mid - 1
```

```
        } else {
            lo = mid + 1
        }
    }
    (hi + 1) as usize
}

fn find_item_index(&self, k: u128, descending: bool) -> Option<usize> {
    let ObPartsMut { header, items } = self;
    let mut lo: i64 = 0;
    let mut hi: i64 = header.len as i64 - 1;
    while lo <= hi {
        let mid = (hi + lo) / 2;
        let current = &items[mid as usize];
        let id = current.order_id();
        if id == k {
            return Some(mid as usize);
        }
        if (id > k) != descending {
            hi = mid - 1
        } else {
            lo = mid + 1
        }
    }
    None
}
```

UPDATES

- *Aug 09, 2022*: This issue has been acknowledged and fixed by the Dradex team.

2.2.5. Unnecessary `i64` for `start_at` and `end_at` in `Farm` struct INFORMATIVE

Function `create_farm` requires `start_at >= now` and `now` (`ctx.accounts.clock.slot`) is `u64` so `start_at` and `end_at` can not be negative. It is unnecessary to use `i64` for `Farm` struct.

```
pub struct Farm {
    pub authority: Pubkey,
    pub market: Pubkey,
    pub token: Pubkey,
    pub amount: u64,
    pub start_at: i64,
    pub end_at: i64,
    pub duration: u64,
    pub amount_staked: u64,
    pub lp_token: Pubkey,
}

pub fn create_farm(ctx: Context<CreateFarm>, input: CreateFarmInput) -> ProgramResult {
    let mut logger = new_logger!(ctx);
```

```
    require!(input.amount > 0, InvalidAmount);
    require!(input.duration > 0, InvalidFarmDuration);
    let now = ctx.accounts.clock.slot as i64;
    let start_at = if input.start_at == 0 {
        now
    } else {
        input.start_at as i64
    };
    require!(start_at >= now, InvalidFarmStartTime);
    let end_at = start_at.checked_add(input.duration as i64).unwrap();
    let farm = &mut ctx.accounts.farm;
    farm.market = ctx.accounts.market.key();
    farm.token = ctx.accounts.token_mint.key();
    farm.amount = input.amount;
    farm.duration = input.duration;
    farm.end_at = end_at;
    farm.start_at = start_at;
    farm.authority = ctx.accounts.signer.key();
    farm.lp_token = ctx.accounts.lp_token_mint.key();
    log!(logger, "start_at {:?}", farm.start_at);

    // deposit tokens to emit
    token::transfer(ctx.accounts.into_token_deposit_context(), input.amount)?;
    logger.flush()
}
```

RECOMMENDATION

Using `u64` for `start_at` and `end_at` to avoid confusing and casts.

UPDATES

- *July 01, 2022*: This issue has been acknowledged and fixed by the Dradex team.

2.2.6. Require `amount > 0 || harvest` to avoid unnecessary running INFORMATIVE

`farm_withdraw` function should require `amount > 0 || harvest` to avoid unnecessary running.

```
pub fn farm_withdraw(ctx: Context<FarmOperation>, amount: u64, harvest: bool) ->
ProgramResult {
    let mut logger = new_logger!(ctx);
    let now = ctx.accounts.clock.slot as i64;
    log!(logger, "at {:?}", now);
    let unclaimed_reward: u64;
    {
        let user = &ctx.accounts.farm_user;
        require!(amount <= user.amount_staked, InvalidAmount);
        let new_reward = ctx.accounts.farm.get_farm_reward(now, user);
        unclaimed_reward = user.unclaimed_reward.checked_add(new_reward).unwrap();
    }
```

```
    // withdraw lp tokens
    if amount > 0 {
        token::transfer(
            ctx.accounts
                .into_lp_token_withdraw_context()
                .with_signer(&[&MASTER_SEEDS[..]]),
            amount,
        )?;
        let user = &mut ctx.accounts.farm_user;
        user.unclaimed_reward = unclaimed_reward;
        user.amount_staked = user.amount_staked.checked_sub(amount).unwrap();
        user.staked_at = now;
        log!(logger, "unclaimed_reward {:?}", user.unclaimed_reward);
        log!(logger, "user_amount_staked {:?}", user.amount_staked);
        let farm = &mut ctx.accounts.farm;
        farm.amount_staked = farm.amount_staked.checked_sub(amount).unwrap();
        log!(logger, "farm_amount_staked {:?}", farm.amount_staked);
    }

    // harvest rewards
    if harvest {
        log!(logger, "harvest {:?}", unclaimed_reward);
        token::transfer(
            ctx.accounts
                .into_token_withdraw_context()
                .with_signer(&[&MASTER_SEEDS[..]]),
            unclaimed_reward,
        )?;
        let user = &mut ctx.accounts.farm_user;
        user.unclaimed_reward = 0;
        user.staked_at = now;
    }
    logger.flush()
}
```

UPDATES

- *July 01, 2022*: This issue has been acknowledged and fixed by the Dradex team.

3. VERSION HISTORY

Version	Date	Status/Change	Created by
1.0	*June 22, 2022*	Private Report	Verichains Lab
1.1	*July 01, 2022*	Public Report	Verichains Lab
1.2	*Aug 09, 2022*	Public Report	Verichains Lab

Table 2. Report versions history