# Frank castle

**DUB audit**

**Septemper 2024**

# DUB_Audit

## About Frank Castle

**Frank Castle** is a seasoned smart contract security researcher with a focused expertise in auditing **Rust-based** contracts and decentralized infrastructure across leading blockchain ecosystems, including **Solana** , **Polkadot** , and **Cosmos** (CosmWasm). Frank's experience includes work with industry-renowned audit firms such as Pashov Audit Group and Shieldify Audit Company, in addition to competitive auditing platforms like **Code4rena** and **Cantina** .

### Contanct Info

Telegram : castle_chain
X : 0xcastle_chain
Gmail : castlechain99@gmail.com

## Scope

- programs/vesting
- programs/Lock
- programs/ bonding-curve

## Finding summary

| Issue ID | Title |
|---|---|
| C-01 | Sending PC tokens directly to `pool_pc_token` account leads to DoS |
| M-01 | Lack of trading pause after migration |
| M-02 | Pool balance manipulation before migration |
| M-03 | Migration to Raydium fails for pools with tokens having freeze authority enabled |
| M-04 | Freeze authority on base mint in `deploy_bonding_mint` |
| M-05 | Premature token releases in Lock program |
| M-06 | State inconsistency due to Solana rollback |
| M-07 | DoS for legitimate AMM creators is possible |

## [C-01] Sending PC tokens directly to `pool_pc_token` account leads to DOS

### Severity

**Impact:** High

**Likelihood:** High

### Description

Consider a scenario where the pool has successfully sold most of its PC tokens and accumulated a significant amount of WSOL. At this point, the reserve ratio stands at `30,000,000 : 100,000,000,000`. A malicious user who initially purchased `30,000,000` PC tokens at a low price during the early stages of the bonding curve can exploit this situation.

The first `30_000_000` will cost `1.387` Sol , which is a very low cost for the malicious user.

This malicious user can send those tokens directly to the `pool_pc_account` without initiating a swap. As a result, the following check in the migration function will always fail:

```
let pool_pc_balance = ctx.accounts.pool_token_pc.amount;

const MAX_POOL_PC_BALANCE: u64 = 30_000_000 * (10u64.pow(6));
assert!(pool_pc_balance < MAX_POOL_PC_BALANCE, "Migration can only happen
when only 30mm tokens are left in the pool.");
```

Because the pool uses internal accounting for swaps, this donation of tokens does not affect the price or the reserve ratio. The reserve ratio remains unchanged, as shown in the swap logic:

```
let a_reserves = ctx.accounts.pool.reserves_a;
let b_reserves = ctx.accounts.pool.reserves_b;

let output: u64 = if swap_a {
    ((input as u128) * (b_reserves as u128))
        .checked_div((a_reserves as u128) + (input as u128))
        .ok_or(BondingCurveError::Overflow)? as u64
} else {
    ((input as u128) * (a_reserves as u128))
        .checked_div((b_reserves as u128) + (input as u128))
        .ok_or(BondingCurveError::Overflow)? as u64
};
```

Since the constant product market maker (CPMM) mechanism makes it impossible to swap out the entire token reserve, it becomes impossible to resume the migration. The migration function expects a transfer of `30,000,000` PC tokens, which is now blocked due to the inflated token balance. This results in a permanent and unrecoverable denial of service (DoS) to the migration process.

This issue can occur with various reserve ratios and only requires a donation of PC tokens, making swaps of the pc_amount that will make the pool_pc_balance lower than the `MAX_POOL_PC_BALANCE` prohibitively expensive or even impossible.

### Recommendation

To prevent this manipulation, the check for the maximum pool PC balance should compare against `pool.reserves_b` instead of the actual balance of the pool. This change will ensure that token donations do not interfere with the migration process and prevent a denial of service.

```
let pool_pc_balance = pool.reserves_a;

const MAX_POOL_PC_BALANCE: u64 = 30_000_000 * (10u64.pow(6));
assert!(pool_pc_balance < MAX_POOL_PC_BALANCE, "Migration can only happen
when only 30mm tokens are left in the pool.");
```

## [M-01] Lack of trading pause after migration

### Severity

**Impact:** Medium

**Likelihood:** Medium

### Description

In the `migrate_cpmm` function, after a successful migration, trading is not paused. This creates a vulnerability where any assets left in the pool can be permanently locked. Since the pool is not designed to allow token withdrawal after migration and only allows swapping, any assets that will be traded in the pool will be inaccessible to users, leading to a permanent lock of funds for those attempting to swap after the migration.

Current migration function:

```
pub fn migrate_cpmm(
    ctx: Context<MigrateCPMM>,
) -> Result<()> {
    assert_eq!(ctx.accounts.pool.complete, true);
    assert_eq!(ctx.accounts.payer.key(), ctx.accounts.amm.migrator);
    ...
    let cpi_accounts = Burn {
        mint: ctx.accounts.amm_lp_mint.to_account_info(),
        from: ctx.accounts.user_token_lp.to_account_info(),
        authority: ctx.accounts.payer.to_account_info(),
    };
    let cpi_ctx: CpiContext<Burn> =
CpiContext::new(ctx.accounts.token_program.to_account_info(), cpi_accounts);
    token::burn(cpi_ctx, amount)?;
    Ok(())
}
```

If trading is not paused after migration, funds that remain in the pool post-migration will become permanently locked. This issue can lead to significant financial losses for users who try to swap after migration, as there will be no mechanism to withdraw funds from the pool.

## Recommendations

1. **Pause trading after migration:** Add a flag to the pool structure, such as `pool.trade_paused`, and set it to `true` after the migration process is complete. This flag should be checked on every trade attempt to prevent further swaps after migration.
2. **Allow withdrawal after migration:** Implement a mechanism to allow users to withdraw their funds from the pool after the migration, ensuring that no assets are permanently locked.

Example modification:

```
pub fn migrate_cpmm(
    ctx: Context<MigrateCPMM>,
) -> Result<()> {
    assert_eq!(ctx.accounts.pool.complete, true);
    assert_eq!(ctx.accounts.payer.key(), ctx.accounts.amm.migrator);

    // Pause trading after migration
    ctx.accounts.pool.trade_paused = true;

    let cpi_accounts = Burn {
        mint: ctx.accounts.amm_lp_mint.to_account_info(),
        from: ctx.accounts.user_token_lp.to_account_info(),
        authority: ctx.accounts.payer.to_account_info(),
    };
    let cpi_ctx: CpiContext<Burn> =
CpiContext::new(ctx.accounts.token_program.to_account_info(), cpi_accounts);
    token::burn(cpi_ctx, amount)?;
    Ok(())
}
```

This ensures that the pool is no longer usable for trading after migration and prevents funds from becoming inaccessible.

## [M-02] Pool balance manipulation before migration

### Severity

**Impact:** Medium

**Likelihood:** Medium

#### Description

In the `migrate_cpmm` function, the pool migration is restricted by a maximum pool balance limit of 30 million `pc` tokens, as shown in the following snippet:

```
let pool_pc_balance = ctx.accounts.pool_token_pc.amount;
assert!(pool_pc_balance < MAX_POOL_PC_BALANCE, "Migration can only happen
when only 30mm tokens are left in the pool.");
```

A malicious user can exploit this by observing the upcoming migration transaction and swapping `pc` tokens for `coin` to artificially increase the balance of `pool_token_pc`. By doing so, they can cause the `pool_pc_balance` to exceed the 30 million token limit, resulting in the migration transaction failing. This can disrupt the migration process and delay or prevent the protocol from completing the migration.

Successful exploitation of this vulnerability can block the migration of the bonding curve to the Raydium AMM. This could lead to prolonged downtime, affecting liquidity and user trust. If such attacks are performed repeatedly, it could also create a denial-of-service (DoS) scenario for the migration process.

#### Recommendations

Pause trading When the migration goal is reached:

- When the migration process is triggered and the balance is about to reach the target limit (30 million `pc` tokens or less), all trading on the pool should be paused immediately. This will prevent further swaps from artificially inflating the pool balance during the migration process.

## [M-03] Migration to Raydium fails for pools with tokens having freeze authority enabled

### Severity

**Impact:** Medium

**Likelihood:** Medium

### Description

The Dub allows for the creation of pools with tokens that may have freeze authority enabled. However, the migration process to Raydium requires that all tokens in the pool have their freeze authority disabled. This mismatch creates a critical issue where migration attempts for pools containing tokens with enabled freeze authority will fail.

```
raydium_cp_swap::cpi::initialize(
        cpi_ctx,
        token_0_amount,
        token_1_amount,
        0,
    )?;
    }
```

bonding-curve/src/instructions/migrate_cpmm.rs:L145

As a result, migrators attempting to migrate their liquidity from pools containing tokens with enabled freeze authority will experience failed transactions which leads to user funds becoming stuck in Dub pool.

The issue affects all pools created in Dub protocol that contain tokens with enabled freeze authority, potentially impacting a significant portion of pools and users.

References:

> For SPL tokens, pool creation requires freeze authority disabled

https://docs.raydium.io/raydium/pool-creation/creating-a-constant-product-pool

### Recommendations

Add a check to ensure the tokens doesn't have an active freeze authority. If you still want to support a few tokens that have active freeze authority, consider migrating these pools to an alternative DEX.

## [M-04] Freeze authority on base mint in `deploy_bonding_mint`

#### Severity

**Impact:** Medium

**Likelihood:** Medium

#### Description

In the `deploy_bonding_mint` function, the `base_mint` of the pool is set, and the mint is initialized. However, there is no check to verify whether the `freeze_authority` of the `base_mint` token is set. If the `freeze_authority` is assigned to a public key, the account holding this authority can freeze critical token accounts, such as the `fee_vault`, which is used during swaps to collect fees. This would result in a permanent denial of service (DOS) on the swapping functionality for all AMMs relying on this bonding curve.

Here is the relevant portion of the `deploy_bonding_mint` function:

```
#[account(
    init,
    payer = payer,
    seeds = [
        amm.key().as_ref(),
        mint_bump.key().as_ref()
    ],
    bump,
    mint::decimals = 6,
    mint::authority = amm_authority
)]
pub mint: Box<Account<'info, Mint>>,
// #[account]
/// CHECK:
pub mint_base: AccountInfo<'info>,
```

During the swap process, a fee is calculated and sent to the `fee_vault`. If the `fee_vault` is frozen due to an unchecked `freeze_authority`, it would prevent swaps, as shown below in the `swap_exact_tokens_for_tokens` function:

```
let tax = input_pretax * amm.fee as u64 / 10000;
token::transfer(
    CpiContext::new(
        ctx.accounts.token_program.to_account_info(),
        Transfer {
            from: ctx.accounts.trader_account_a.to_account_info(),
            to: ctx.accounts.fee_vault_token_account.to_account_info(),
            authority: ctx.accounts.trader.to_account_info(),
        },
    ),
    tax,
)?;
```

This freeze could render the entire AMM system unusable, as no trades could be processed due to the inability to transfer fees.

If the `freeze_authority` of the `base_mint` is exploited or utilized, it could cause a permanent DOS on the swapping functionality, making the bonding curve and its AMM useless. Without safeguards, any malicious or compromised actor with the `freeze_authority` could disrupt the AMM operations.

Also migrators will not be able to migrate their liquidity from pools due to the migration goal did not get reached , which leads to user funds becoming stuck in Dub pool.

### Recommendation

1. **Check for `freeze_authority`:** Ensure that the `base_mint` does not have a `freeze_authority` set, or that it is handled carefully. An example of the check could be:

   ```
   if mint_account.freeze_authority.is_some() {
       return Err(Error::MintHasFreezeAuthority);
   }
   ```

2. **Display Warnings for Tokens with `freeze_authority`:** If supporting tokens with a `freeze_authority` is necessary, display a warning to users in the UI, informing them of the risks associated with trading or interacting with these tokens.
3. **Implement Allowlist for Trusted Tokens:** For regulated stablecoins like USDC, which have a `freeze_authority` for security reasons, implement an allowlist for trusted tokens while applying strict checks on other tokens. This ensures the protocol can support widely-used tokens while minimizing risk.

By applying these changes, the protocol will mitigate the risk of an unexpected DOS due to the freezing of critical token accounts.

## [M-05] Premature token releases in Lock program

### Severity

**Impact:** Medium

**Likelihood:** Medium

### Description

The `handle_release` function in `release.rs` lacks proper time-based checks and validation of unlock criteria before setting `claim_approved` to `true`. This could allow premature token releases, potentially enabling attacks using flash loans to unlock and dump tokens before their intended vesting date.

```
// Check if the unlock criteria have been met
    match lockbox.unlock_criteria {
        UnlockCriteria::None => {
```

```
            // No specific criteria, always allow release
        },
        UnlockCriteria::BondingPrice { .. } => {
            // let current_time = Clock::get()?.unix_timestamp;
            // require!(current_time >= unlock_time,
ErrorCode::UnlockConditionsNotMet);
            // TODO
        },
        UnlockCriteria::RaydiumPrice { .. } => {
            // Price-based criteria should be checked elsewhere, possibly
off-chain
            // Here we assume it's already verified if this criteria is set
        },
    }

    // Set claim_approved to true
    lockbox.claim_approved = true;
```

lock/src/instructions/release.rs:L42

### Recommendations

Update `handle_release` to include Time-based checks ensuring current timestamp ≥ scheduled unlock time. Or Implement a proper validation of specified unlock criteria (e.g., bonding price thresholds).

## [M-06] State inconsistency due to Solana rollback

### Severity

**Impact:** Medium

**Likelihood:** Medium

### Description

Two critical functions in the protocol are vulnerable to state inconsistencies in the event of a Solana rollback:

- Merkle Root Updates: The `handle_update_root` function in `merkle_distributor/update_root.rs` updates the Merkle root, max claim amounts, and resets claim counters. A rollback after this update could create a mismatch between off-chain data and on-chain state, potentially allowing unintended claims, exceeding claim limits, or enabling double-claiming.

- AMM Pause Mechanism: the `update_pause` function in `bonding_curve/update.rs` controls the AMM's ability to halt trading in emergency situations. However, in the event of a Solana rollback, this critical security feature could be compromised. If the AMM was paused due to a detected security threat, a rollback could revert it to an active state, potentially exposing the protocol to the very threat it was trying to mitigate.

## Recommendations

Utilize the `LastRestartSlot` sysvar to detect outdated configuration states. If the config is outdated, the protocol should automatically pause until an action is taken by the admin.

Example as an inspiration:

- Add a `last_updated_slot` field to the AMM state struct:

```
pub struct AmmState {
    // ... other fields ...
    pub last_updated_slot: u64,
    pub trading_paused: bool,
    // ... other fields ...
}
```

- Add a function to check if the config is outdated:

```
fn is_config_outdated(amm_state: &AmmState) -> Result<bool> {
    let last_restart_slot = LastRestartSlot::get()?;
    Ok(amm_state.last_updated_slot <=
last_restart_slot.last_restart_slot)
}
```

- Modify the `swap_exact_tokens_for_tokens` and other critical functions to check for outdated config

```
pub fn swap_exact_tokens_for_tokens(ctx:
Context<SwapExactTokensForTokens>, ...) -> Result<()> {
    let amm = &ctx.accounts.amm;

    if is_config_outdated(amm)? || amm.trading_paused {
        return err!(BondingCurveError::TradingPaused);
    }

    // ... rest of the function
}
```

- Update the `last_updated_slot` in the `update_pause` function

```
pub fn update_pause(ctx: Context<UpdatePause>, new_pause: bool) ->
Result<()> {
    require!(ctx.accounts.admin.key() == ctx.accounts.amm.admin,
BondingCurveError::Unauthorized);
    let amm = &mut ctx.accounts.amm;
    amm.trading_paused = new_pause;
    amm.last_updated_slot = Clock::get()?.slot;
    Ok(())
}
```

## [M-07] DoS for legitimate AMM creators is possible

### Severity

**Impact:** Medium

**Likelihood:** Medium

### Description

`create_amm` function in bonding curve is vulnerable to front-running attacks.

A scenario of how this could occur:

- A legitimate user submits a transaction to create an AMM with a specific ID.
- The attacker quickly submits their own transaction to create an AMM with the same ID, but with higher gas fees.
- The attacker's transaction is processed first, creating the AMM with the attacker's parameters.
- The legitimate user's transaction fails due to the AMM ID already being in use.

This could lead to:

- DoS for legitimate AMM creators
- Creation of malicious AMMs that mimic legitimate ones

The current code doesn't have any mechanism to prevent this type of attack:

```
#[derive(Accounts)]
#[instruction(id: Pubkey, fee: u16)]
pub struct CreateAmm<'info> {
    #[account(
        init,
        payer = payer,
        space = Amm::LEN,
        seeds = [id.as_ref()],
        bump,
        constraint = fee < 30000 @ BondingCurveError::InvalidFee,
    )]
    pub amm: Box<Account<'info, Amm>>,
    // ... other fields
}
```

bonding-curve/src/instructions/create_amm.rs:L42

The AMM is created using only the provided `id`.

## Recommendations

To mitigate this, I recommend implementing a sequential numbering. This can be achieved by:

- Adding a new account to store the latest AMM sequence number.
- Modifying the `CreateAmm` struct and `create_amm` function to use this sequence number.

Here's a proposed implementation:

```
#[derive(Accounts)]
#[instruction(id: Pubkey, fee: u16)]
pub struct CreateAmm<'info> {
    #[account(
        init,
        payer = payer,
        space = Amm::LEN,
        seeds = [id.as_ref(),
amm_sequence.current_sequence.to_le_bytes().as_ref()],
        bump,
        constraint = fee < 30000 @ BondingCurveError::InvalidFee,
    )]
    pub amm: Box<Account<'info, Amm>>,

    #[account(mut)]
    pub amm_sequence: Account<'info, AmmSequence>,

    // ... other fields
}

#[account]
pub struct AmmSequence {
    pub current_sequence: u64,
}
```

```
pub fn create_amm(ctx: Context<CreateAmm>, /* ... other params */) ->
Result<()> {
    // ... existing code

    // Increment and use the sequence number
    ctx.accounts.amm_sequence.current_sequence += 1;
    ctx.accounts.amm.sequence = ctx.accounts.amm_sequence.current_sequence;

    // ... rest of the function
}
```

Please make sure that the `AmmSequence` account can only be modified by the program itself.

## [M-08] Fee rounding enables zero-fee swaps

### Severity

**Impact:** Medium

**Likelihood:** Medium

#### Description

In the current swap fee calculation, fees are rounded down, which can result in zero fees being applied when swapping small amounts of low decimal tokens. This is particularly problematic for tokens with a low decimal precision (e.g., 3 decimals). For example, if the fee is set at 1% (or 100 basis points) and the amount to swap is `99`, no fee is applied, effectively allowing a user to swap tokens for free. In the case of low decimal tokens, a user could repeatedly swap small amounts without incurring any fees.

As shown in the following snippet, the fee is calculated by dividing `input_pretax * amm.fee` by `10000`:

```
let tax = input_pretax * amm.fee as u64 / 10000;
```

The issue arises because the division rounds down, which results in `0` for small input amounts.

This test case demonstrates how the current fee calculation can round down to zero:

```
#[test]
fn test_fee_rounding_down(){
    let fee = 100; // 1.00% fee
    let input_pretax = 99;
    let tax = input_pretax * fee as u64 / 10000;

    let input_amount_post_fee = input_pretax - tax;
    println!("fee amount: {:?}", tax);
    println!("input amount post fee: {:?}", input_amount_post_fee);

    // logs:
    // fee amount : 0
    // input amount post fee 99
}
```

In this example, the fee amount is calculated as `0`, meaning the swap proceeds without any fee. This issue could be exploited by swapping `99` units of a low decimal token repeatedly to avoid paying fees.

This issue allows users to swap certain token amounts without incurring fees, particularly for low decimal tokens. This can be exploited, leading to potential revenue loss for the protocol and undermining the fee structure of the bonding curve.

### Recommendation

To prevent the fee from rounding down to zero, use a ceiling division (`ceil_div`) instead of a standard division (`/`), ensuring that the fee is always rounded up
or this can be used :

```
let tax = input_pretax * amm.fee as u64
    .checked_add(9999) // Add this to round up when dividing
    .ok_or(BondingCurveError::Overflow)?
    / 10000;
```

This change will ensure that even for small amounts of tokens, a non-zero fee will be taken, preventing users from bypassing the fee system.