Frank castle

Pump-fun AMM audit

August 2024

about FrankCastle

I am Frank Castle, a smart contract security researcher who specializes in auditing Rust-based projects on different ecosystems such as Polkadot (substrate), Cosmos (Cosmwasm), and Solana (anchor).

My work focuses on fortifying the security of smart contracts through meticulous audits, with a passion for enhancing the blockchain's integrity. Beyond identifying risks, I contribute to developing innovative solutions aimed at preempting security flaws.

For private audit or consulting requests please reach out to me via Telegram @castle_chain

Disclaimer

Security review cannot guarantee 100% the safeness of the protocol. In the Auditing process, we try to get all possible issues, and we can not be sure if we missed something or not.

I am not responsible for any misbehavior, bugs, or exploits affecting the audited code or any part of the deployment phase.

Any change to the code after the mitigation process puts the protocol at risk and should be audited again.

Findings summary

ID	Title	Severity
[H-01]	DoS Vulnerability in Swapping Functions After Pool Creation	High
[M-01]	Decimal Misalignment in Liquidity Pool Creation Leading to Incorrect `lp_mint` Calculation	Medium
[M-02]	Loss of value impact due to Collecting Fees Exclusively in `Quote_Token`	Medium
[M-03]	Liquidity Provider Manipulation Leading to Price Volatility and Trade Disruption Due to Fee-Free Liquidity Provision and Removal	Medium
[M-04]	DoS Vulnerability Due to Global Disable Flags Across All Pools	Medium
[L-01]	Price Manipulation Due to the `Disable` Instruction	LOW

Findings

High severity findings

[H-01] DoS Vulnerability in Swapping Functions After Pool Creation

Severity

Impact: Medium

Likelihood: High

Description

The `sell` and `buy` instructions are vulnerable to a Denial of Service (DoS) until the `protocol_fee_recipient`'s associated token account (ATA) is created and initialized by the `fee_receiver` for the specific `quote_mint` of the liquidity pool.

The `create_pool` function does not deduct fees from the pool creator. Consequently, if a pool is created without the `protocol_fee_recipient`'s token account being set up to receive fees, the swapping functionality will be disabled until the token account (ATA) is created. Since pool creation is permissionless and should activate without requiring any permissions or whitelisting, it is expected that swapping within the pool will be functional immediately after the pool is created.

As illustrated in the code snippet below:

```
pub struct CreatePool<'info> {
    #[account(
        init,
        payer = creator,
        space = 8 + PoolAccount::INIT_SPACE,
        seeds = [
            POOL_SEED.as_bytes(),
            &index.to_le_bytes(),
            creator.key().as_ref(),
            base_mint.key().as_ref(),
            quote_mint.key().as_ref(),
        ],
        bump
    )]
    pub pool_account: Box<Account<'info, PoolAccount>>,
    pub amm_config: Box<Account<'info, AmmConfig>>,

    #[account(mut)]
    pub creator: Signer<'info>,

    #[account(
        mint::token_program = base_token_program,
```

```
    )]
    pub base_mint: Box<InterfaceAccount<'info, Mint>>,

    #[account(
        mint::token_program = quote_token_program,
    )]
    pub quote_mint: Box<InterfaceAccount<'info, Mint>>,

    #[account(
        init,
        payer = creator,
        seeds = [
            POOL_LP_MINT_SEED.as_bytes(),
            pool_account.key().as_ref(),
        ],
        bump,
        mint::decimals = 9,
        mint::authority = pool_account,
        mint::token_program = token_2022_program,
    )]
    pub lp_mint: Box<InterfaceAccount<'info, Mint>>,

    #[account(mut)]
    pub user_base_token_account: Box<InterfaceAccount<'info, TokenAccount>>,

    #[account(mut)]
    pub user_quote_token_account: Box<InterfaceAccount<'info,
TokenAccount>>,

    #[account(
        init,
        payer = creator,
        associated_token::mint = lp_mint,
        associated_token::authority = creator,
        associated_token::token_program = token_2022_program,
    )]
    pub user_pool_token_account: Box<InterfaceAccount<'info, TokenAccount>>,

    #[account(
        mut,
        associated_token::mint = base_mint,
        associated_token::authority = pool_account,
        associated_token::token_program = base_token_program,
```

```
    )]
    pub pool_base_token_account: Box<InterfaceAccount<'info, TokenAccount>>,

    #[account(
        mut,
        associated_token::mint = quote_mint,
        associated_token::authority = pool_account,
        associated_token::token_program = quote_token_program,
    )]
    pub pool_quote_token_account: Box<InterfaceAccount<'info,
TokenAccount>>,

    pub system_program: Program<'info, System>,
    pub token_2022_program: Program<'info, Token2022>,
    pub base_token_program: Interface<'info, TokenInterface>,
    pub quote_token_program: Interface<'info, TokenInterface>,
    pub associated_token_program: Program<'info, AssociatedToken>,
}
```

The `create_pool` instruction does not automatically create the ATA for the `protocol_fee_recipient` based on the `quote_mint`. As a result, the swapping functions, which require the `protocol_fee_recipient_token_account`, will fail:

The swapping functions such as `Buy` and `Sell` require a `protocol_fee_recipient_token_account` account to send the fees to it; the account should be created and Initialized in order to this instruction call to pass.

```
#[account(
    mut,
    associated_token::mint = quote_mint,
    associated_token::authority = amm_config.protocol_fee_recipient,
    associated_token::token_program = quote_token_program,
)]
pub protocol_fee_recipient_token_account: Box<InterfaceAccount<'info,
TokenAccount>>;
```

Recommendations

To allow swapping on the pool immediately after its creation, include the creation and initialization of the `protocol_fee_recipient_token_account` in the `create_pool` instruction. Utilize the `associated_token_program` to create the ATA for the `protocol_fee_recipient`:

```
/// Program to create an ATA for receiving protocol fees
pub associated_token_program: Program<'info, AssociatedToken>,
```

Add this account to the instruction `create_pool` to make sure that the ATA of the `protocol_fee_recipient` is initialized .

```
    #[account(
        init,
        payer = creator,
        associated_token::mint = quote_mint,
        associated_token::authority = protocol_fee_recipient,
        associated_token::token_program = token_2022_program,
    )]
  pub protocol_fee_recipient_token_account:
```

Another solution:
`init_if_needed` can be used to initialize the `protocol_fee_recipient_token_account` if it needs to be , which can also mitigate this vulnerability .

This ensures that the swapping functionality is available right after pool creation, preventing any potential DoS scenarios.

Medium severity findings

[M-01] Decimal Misalignment in Liquidity Pool Creation Leading to Incorrect `lp_mint` Calculation

Description

The `base_mint` and `quote_mint` tokens can have different decimal values, but the `lp_mint` token must have `9 decimals`, as shown in the following code:

```
#[account(
    init,
    payer = creator,
    seeds = [
        POOL_LP_MINT_SEED.as_bytes(),
        pool_account.key().as_ref(),
    ],
    bump,
    mint::decimals = 9,
    mint::authority = pool_account,
    mint::token_program = token_2022_program,
)]
pub lp_mint: Box<InterfaceAccount<'info, Mint>>,
```

In the `create_pool` function, the `initial_liquidity` is calculated by taking the square root of the product of the two token amounts. However, this calculation does not correctly account for the differing decimals of each token, nor does it scale the amounts to match the `lp_mint` decimals.

```
//@audit findings: if the two tokens have different decimals, this will
result in an incorrect amount of `lp_mint` tokens.
let initial_liquidity = U128::from(base_amount_in)
    .checked_mul(U128::from(quote_amount_in))
    .unwrap()
    .integer_sqrt()
    .try_into()
    .unwrap();
```

For example, if the `quote_mint` has 16 decimals and the `base_mint` has 8 decimals, and `500 units` of each token are initially provided, the correct amount of `lp_mint` tokens minted to the user should be `500 units` with 9 decimals. However, the actual calculation will be:

```
initial_liquidity = sqrt(500*(10^16) * 500*(10^8)) = 500*(10^14)
```

The calculated value is `10^3` times greater than the correct value.

Recommendations

To mitigate this issue, each token amount should be scaled to the correct number of decimals (i.e., 9 decimals) before performing the liquidity calculation. This can be achieved as follows:

```
scaled_amount = amount * (10^(9 - mint_decimals));
```

By scaling the token amounts correctly, the calculation will yield the proper number of `lp_mint` tokens, preventing any discrepancies caused by differing token decimals.

[M-02] Loss of value impact due to Collecting Fees Exclusively in `Quote_Token`

Severity

Impact: Medium

Likelihood: Medium

Description

In the swapping functions (`Buy` and `Sell`), the fees are currently implemented in such a way that they are only collected from one type of token, the `quote_token`. This approach directs the `protocol_fees` to the `protocol_fee_recipient` and retains the `lp_fees` within the liquidity pool.

Consequently, each swap results in an increase in the `quote_token` reserve, which in turn negatively affects the price of the `quote_token`, leading to a loss of value for liquidity providers.

In the `Buy` Function:

```
let lp_fee_basis_points = ctx.accounts.amm_config.lp_fee_basis_points;
let lp_fee = fee(quote_amount_in, lp_fee_basis_points);

let quote_amount_in_with_lp_fee = quote_amount_in + lp_fee;

let protocol_fee_basis_points =
ctx.accounts.amm_config.protocol_fee_basis_points;
let protocol_fee = fee(quote_amount_in, protocol_fee_basis_points);
```

All fees are deducted from the `quote_token` and kept in the pool for liquidity providers.

In the `Sell` Function:

```
let lp_fee_basis_points = ctx.accounts.amm_config.lp_fee_basis_points;
let lp_fee = fee(quote_amount_out, lp_fee_basis_points);

let protocol_fee_basis_points =
ctx.accounts.amm_config.protocol_fee_basis_points;
let protocol_fee = fee(quote_amount_out, protocol_fee_basis_points);
```

If the price of the `quote_token` drops, the value of the accumulated fees will also decline. Over time, the value of these fees decreases as the `quote_reserves` inflate.

Recommendations

To mitigate this issue, fees should be taken from the token that is leaving the pool, which would provide liquidity providers with more valuable fees and prevent the depreciation of the `quote_token`.

For example:

- In the `Buy` function, fees should be taken from the `base_token`.
- In the `Sell` function, fees can continue to be taken from the `quote_token`.

  This approach will ensure that fees do not negatively impact the price of the `quote_token`, thereby protecting the value held by liquidity providers.

[M-03] Liquidity Provider Manipulation Leading to Price Volatility and Trade Disruption Due to Fee-Free Liquidity Provision and Removal

Severity

Impact: Low

Likelihood: High

Description

This vulnerability allows a liquidity provider to manipulate price exposure during swaps, causing unexpected volatility with minimal cost ,because no fees are applied on liquidity provision and removal. Even with slippage protection in place, it can disrupt swaps or block specific trades, such as arbitrage opportunities, from being executed , to be taken by the liquidity provider .

Proof of Concept (PoC)

Consider a liquidity pool containing `token_1` and `token_2` with the following reserves:

```
token_1_res = 500
token_2_res = 500
lp_supply = 100
```

If a liquidity provider (LP) wants to prevent a trade from occurring, they can simply withdraw a portion of their liquidity before the trade, leaving only a small amount in the pool. This will cause the trade to fail due to insufficient liquidity.

For example, the liquidity provider withdraws `50 units of lp_tokens`:

```
token1_res = 250
token_2_res = 250
lp_supply = 50
```

If a user attempted to swap `100 units of token_1` before the LP removed their liquidity, they would expect to receive `83.333 units` of `token_2`. Thus, they might set the `min_amount_out` to `83 units`:

```
token_2_amount_out = (100 * 500) / (500 + 100) = 83.333 units
```

However, after the liquidity removal, the amount the user would actually receive is `71.428 units`:

```
token_2_amount_out = (100 * 250) / (100 + 250) = 71.428 units
```

The user expected to receive `83.33 units` of `token_2`, but due to the liquidity removal, the calculated amount is only `71.428 units`. This discrepancy can cause the transaction to fail if the user has set a slippage tolerance, or, if not, it may result in a loss of funds. The LP could take advantage of the arbitrage opportunity, or the user could miss out on it entirely.

This attack is feasible for any liquidity provider and can effectively prevent any trade from being executed.

Recommendations

Implement fees on the `deposit` and `withdraw` instructions to mitigate this vulnerability and discourage manipulative liquidity provision and removal.

[M-04] DoS Vulnerability Due to Global Disable Flags Across All Pools

Severity

Impact: Low

Likelihood: High

Description

The current implementation uses a global `amm_config` for all liquidity pools, which includes settings for fees, the fee recipient, and a disable flag that controls which functionalities are disabled:

```
pub enum DisableFlag {
    CreatePool = 0,
    Deposit = 1,
    Withdraw = 2,
    Buy = 3,
    Sell = 4,
}
```

This global configuration means that all functionalities—such as creating a pool, depositing, withdrawing, buying, and selling—can be disabled for all pools simultaneously. If one pool is compromised and the disable flag is set, it will disable all the functions for every pool created by the program, leading to a Denial of Service (DoS) attack on all active pools.

Disabling all liquidity pools simultaneously is not a practical feature and could cause significant disruption to all unaffected liquidity pools.

The issue stems from having only one global `amm_config` for all pools, as illustrated below:

```
pub struct CreateConfig<'info> {
    #[account(
        mut,
        address = crate::admin::id() @ PumpAmmError::InvalidAdmin,
    )]
```

```
    pub admin: Signer<'info>,

    #[account(
        init,
        seeds = [
            AMM_CONFIG_SEED.as_bytes()
        ],
        bump,
        payer = admin,
        space = 8 + AmmConfig::INIT_SPACE,
    )]
    pub amm_config: Account<'info, AmmConfig>,

    pub system_program: Program<'info, System>,
}
```

The `amm_config` is global for all liquidity pools because the seed used to generate it is a constant, leading to a single configuration for all pools.

Recommendations

To mitigate this issue, consider one of the following approaches:

1. **Create an Individual `amm_config` for Each Pool:**
   Generate a separate `amm_config` for each pool during its creation. This will allow each pool to have its own configuration and disable flags, ensuring that disabling one pool does not affect others.
2. **Add a Mapping for Each Pool's Configuration in the Global `amm_config`:**
   Modify the global `amm_config` to include a map that stores individual configurations for each pool. This would separate the disable flags and other settings for each liquidity pool, preventing a single compromised pool from impacting the entire system.

Low finding

[L-01] Price Manipulation Due to the `Disable` Instruction

The `disable` instruction is intended to deactivate specific functionalities of liquidity pools, such as deposits, withdrawals, or trading operations. However, it also allows for the selective disabling of `buy` or `sell` operations individually. This selective blocking can affect asset prices by allowing trades to occur in only one direction while blocking the other, leading to potential price distortions.

This is an unintended behavior. The disable feature is meant to be used in emergency situations and should not influence asset prices. However, the current implementation could open the door to price manipulation across liquidity pools.

Code Example:

```
pub enum DisableFlag {
    CreatePool = 0,
    Deposit = 1,
    Withdraw = 2,
    Buy = 3,
    Sell = 4,
}
```

As shown in the code, `Buy` and `Sell` operations can be disabled separately, which could impact asset prices across all liquidity pools.

Recommendations

Instead of disabling `buy` and `sell` operations separately, implement a single `DisableFlag` for swapping. This flag would disable both buying and selling simultaneously, preventing unintended price manipulation and ensuring that the disable feature functions correctly during emergency situations.

```
pub enum DisableFlag {
    CreatePool = 0,
    Deposit = 1,
    Withdraw = 2,
    Swap = 3 ,
}
```