Frank castle

Boop Audit

March 2025

Boop Final Report

Content

About Frank Castle 🦀

Frank Castle is a professional 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 Castle has audited Lido, GMX ,Pump.fun, LayerZero, Synthetix , Hydration, something.cool ,DUB Social and several multi-million protocols.

with more than ~35 Rust Audit , ~25 Solana Audits , and +200 criticals/highs found , All the reports can be found here

For private audit or consulting requests please reach out to me via Telegram @castle_chain , **Twitter** (@0xfrank_auditor) or **Discord** (castle_chain).

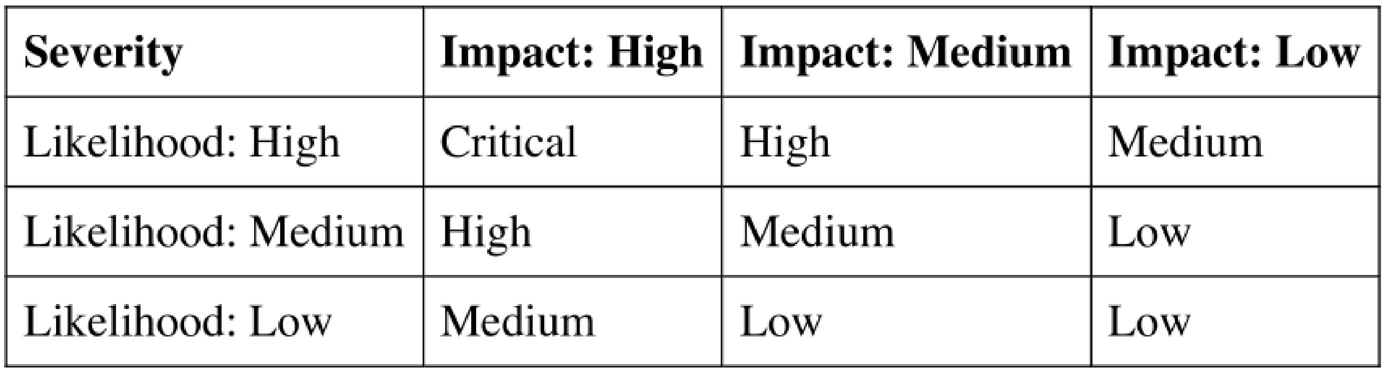
Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

Summary of Findings

ID	Title	Severity	Status
[H-01]	Incorrect Comparison in `buy_token` Leading to Graduation Threshold Violation	High	Resolved
[M-01]	Slippage Protection Bypass in `buy_token`	Medium	Resolved
[L-01]	Incorrect Macro Usage for Pubkey Comparison	Low	Resolved
[L-02]	Insecure Authority Transfer Leading to Potential DoS	Low	Resolved
[L-03]	Setting the destination address to a PDA of a Token Account Closure Leading to SOL Lock	Low	Resolved
[L-04]	Missing Validation for Raydium Program Address	Low	Resolved
[L-05]	Missing Validation for `amount > 0` in `wrap` Function	Low	Resolved
[L-06]	Missing Validation for `protocol_fee_recipient` in `update_config`	Low	Resolved
[L-07]	Permanent Freezing of `create_raydium_pool` Due to Seed Constraint	Low	Resolved

Risk Classification



Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

Audit Scope

All the programs in `boop-solana/programs` :

- boop
- fee_spiltter
- reward_pool
- sboop

Commit Hash : https://github.com/Boop-fun/boop-solana/commit/b7f7533ad783c1a489e4977e013f18ece4bca277

Findings

1. High Findings

1.1 Incorrect Comparison in `buy_token` Leading to Graduation Threshold Violation

Description

In the `buy_token` function, the comparison between `net_buy_amount` and `max_buy_amount_with_fees` is incorrect. The issue arises because `max_buy_amount_with_fees` represents the remaining tokens in the bonding curve after adding fees, while `net_buy_amount` is the user's token amount after subtracting fees.

This mismatch can cause the function to allow purchases beyond the graduation threshold, leading to users paying more for fewer tokens at an inflated price.

Impact

This issue Leads to Graduation Threshold Violation which is core variant in the system , and results in users buying more tokens at a much higher price than expected due to exceeding the bonding curve's graduation threshold. Specifically, the bonding curve's SOL reserves (`sol_reserves`) can exceed the intended graduation reserve, leading to unfair pricing and potential loss of funds for the user.

Scenario:

- **Amount in:** 55 SOL
- **Fee:** 10% (0.1)
- **Max amount without fees:** 45 SOL
- **Max amount with fees:** 50 SOL
- **Net buy amount:** 49.5 SOL

Current Condition:

```
if net_buy_amount <= max_buy_amount_with_fees {
    actual_buy_amount = net_buy_amount; // amount after taking fees
    actual_swap_fee = swap_fee;
} else {
    actual_buy_amount = max_buy_amount_without_fees; // amount without
taking fees
    actual_swap_fee = max_buy_amount_with_fees
        .checked_sub(max_buy_amount_without_fees)
        .unwrap();
};
```

Actual Values in This Scenario:

- **actual_buy_amount:** 49.5 SOL
- **actual_swap_fee:** 5.5 SOL

Since `actual_buy_amount` is incorrectly calculated since it is greater than `Max amount without fees` which is the actual amount of sol need in the curve , the bonding curve's reserves are updated incorrectly:

```
ctx.accounts.bonding_curve.sol_reserves = ctx
    .accounts
    .bonding_curve
    .sol_reserves
    .checked_add(actual_buy_amount)
    .unwrap();
```

This results in the bonding curve surpassing the graduation threshold, causing users to receive fewer tokens than expected while paying an inflated price.

Recommendation

The `net_buy_amount` should be compared with `max_buy_amount_without_fees`, and the equal sign should be removed from the condition:

```diff
- if net_buy_amount <= max_buy_amount_with_fees {
+ if net_buy_amount < max_buy_amount_without_fees {
    actual_buy_amount = net_buy_amount; // amount after taking fees
    actual_swap_fee = swap_fee;
} else {
    actual_buy_amount = max_buy_amount_without_fees; // amount without
taking fees
    actual_swap_fee = max_buy_amount_with_fees
        .checked_sub(max_buy_amount_without_fees)
        .unwrap();

    //@audit-issue The curve should be marked as graduated here.
};
```

This correction ensures that the bonding curve does not exceed the graduation threshold, preventing unfair pricing and loss of funds.

2. Medium Findings

2.1 Slippage Protection Bypass in `buy_token`

Description

In the `buy_token` and `sell_token` functions, users can specify a slippage parameter to protect their trades from executing at unfavorable prices. This is enforced by checking the trade's resulting amount against the user's slippage constraints.

However, in the `buy_token` function, the input amounts are adjusted based on the remaining amount before reaching the graduation threshold. This adjustment overrides the user's originally intended trade amount, making the `amount_out_min` parameter ineffective.

Issue in Code:

```rust
actual_buy_amount = max_buy_amount_without_fees; // amount without taking
fees
actual_swap_fee = max_buy_amount_with_fees
    .checked_sub(max_buy_amount_without_fees)
    .unwrap();
```

Since `actual_buy_amount` is set to `max_buy_amount_without_fees`, the final trade execution does not respect the user's original slippage constraints, as `amount_out_min` was set based on the user's initial `amount_in`.

Impact

- The trade can execute at a price worse than what the user intended.
- Users are exposed to price slippage beyond their specified tolerance.
- This results in a **loss of funds**, as trades may complete at unpreferred or unfair prices.

Recommendation

To maintain slippage protection, a **slippage factor** should be calculated based on the original input amounts. This factor should then be applied to the adjusted amounts to ensure the final trade respects the user's slippage constraints.

3. Low Findings

3.1 Incorrect Macro Usage for Pubkey Comparison

Description

The `require_neq!` macro is used to ensure two non-Pubkey values are not equal. However, in the given code snippet, it is incorrectly used to compare two public keys:

```
require_neq!(creator, Pubkey::default(), ErrorCode::CreatorIsNotProvided);
```

According to the documentation, `require_neq!` should not be used for Pubkey comparisons. Instead, `require_keys_neq!` is the appropriate macro for comparing two public keys.

```
/// Ensures two NON-PUBKEY values are not equal.
///
/// Use [require_keys_neq](crate::prelude::require_keys_neq)
/// to compare two pubkeys.
///
/// Can be used with or without a custom error code.
///
/// # Example
/// ```rust,ignore
/// pub fn set_data(ctx: Context<SetData>, data: u64) -> Result<()> {
///     require_neq!(ctx.accounts.data.data, 0);
///     ctx.accounts.data.data = data;
///     Ok(());
/// }
/// ```
```

Recommendation

Replace `require_neq!` with `require_keys_neq!` when comparing public keys to ensure proper validation:

```
require_keys_neq!(creator, Pubkey::default(),
ErrorCode::CreatorIsNotProvided);
```

3.2 Insecure Authority Transfer Leading to Potential DoS

Description

In the `transfer_authority` function, if the authority is transferred to an invalid or inaccessible address, it could result in a complete denial of service (DoS) for protocol configurations and core functionalities. Since there is no verification mechanism ensuring that the `new_authority` is a valid and active entity, the protocol may become permanently locked if an incorrect address is set.

Recommendation

To prevent this issue, it is recommended to:

1. **Require the new authority to be a signer** for the transaction, ensuring that they acknowledge and accept the role.
2. **Implement a two-step authority transfer process** by introducing a `claim_authority` function. This approach ensures that the new authority explicitly claims their role before the transfer is finalized.

Suggested Fix: Require the New Authority as a Signer

Modify the `TransferAuthority` struct to enforce that the `new_authority` signs the transaction:

```
#[derive(Accounts)]
pub struct TransferAuthority<'info> {
    pub authority: Signer<'info>, // Current authority

    #[account(mut, has_one = authority @ ErrorCode::Unauthorized)]
    pub config: Account<'info, Config>, // The config account whose
authority is being transferred

    pub new_authority: Signer<'info>, // Ensure the new authority is a valid
signer
    pub system_program: Program<'info, System>,
}
```

This ensures that the `new_authority` is actively participating in the transfer, preventing accidental or malicious misconfiguration.

3.3 Setting the destination address to a PDA of a Token Account Closure Leading to SOL Lock

Description

Closing a token account transfers its SOL balance to the specified destination account. However, the destination account must be able to move SOL. Setting the destination to a PDA can result in locked SOL since a PDA cannot directly move SOL.

In the following function call, the destination is set to a PDA:

```
close_vault(
    &ctx.accounts.token_program,
    &ctx.accounts.bonding_curve_vault.to_account_info(),
    &ctx.accounts.vault_authority.to_account_info(),
@>     &ctx.accounts.recipient_token_account.to_account_info(),
    vault_authority_signer,
)?;
```

This setup causes SOL to be locked until `recipient_token_account` is closed.

Recommendation

Set the destination address to the `recipient` directly instead of a PDA. This ensures that the recipient can receive and move the SOL immediately after closure.

3.4 Missing Validation for Raydium Program Address

Description

In the `swap_tokens_for_sol_on_raydium` function, the Raydium program address (`cp_swap_program`) is not explicitly validated.

The current implementation:

```
/// input_token_mint and output_token_mint have the same token program
pub token_program: Interface<'info, TokenInterface>,

pub cp_swap_program: Program<'info, RaydiumCpmm>,
```

Recommendation

To enforce security and ensure the correct Raydium program is used, add explicit validation by specifying the expected address:

```
#[account(address = raydium_cpmm_cpi::ID)]
pub cp_swap_program: Program<'info, RaydiumCpmm>,
```

3.5 Missing Validation for `amount > 0` in `wrap` Function

Description

The `wrap` function is expected to validate that the `amount` parameter is greater than zero before proceeding. However, no such validation is currently implemented. This contradicts the function's documentation, which states that an error should be thrown if `amount` is zero.

Current implementation:

```
pub fn wrap(ctx: Context<Wrap>, amount: u64) -> Result<()> {
    let transfer_accounts = TransferChecked {
        from: ctx.accounts.depositor_boop_token_account.to_account_info(),
        mint: ctx.accounts.boop.to_account_info(),
        to: ctx.accounts.boop_vault.to_account_info(),
        authority: ctx.accounts.depositor.to_account_info(),
    };

    let cpi_ctx = CpiContext::new(
        ctx.accounts.token_program.to_account_info(),
        transfer_accounts,
    );

    transfer_checked(cpi_ctx, amount, ctx.accounts.boop.decimals)?;
```

Without this check, the function may proceed with an invalid zero-amount transfer.

Recommendation

Add a validation check to ensure that `amount` is greater than zero:

```
require!(amount > 0, ErrorCode::ZeroAmount);
```

This ensures that the function behaves as documented and prevents unintended zero-value transfers.

3.6 Missing Validation for `protocol_fee_recipient` in `update_config`

Description

In the `update_config` function, the `protocol_fee_recipient` address is updated without validation. If the recipient address is set to the default `Pubkey::default()`, it can lead to a denial-of-service (DoS) issue, preventing core protocol functionalities from operating correctly.

Current implementation:

```
ctx.accounts.config.protocol_fee_recipient = new_protocol_fee_recipient;
```

This allows an invalid recipient address to be set, which may cause transactions requiring fee distribution to fail.
Also the same issue exist in the sboop program in the function `update_config`

Recommendation

Add a validation check to ensure that `new_protocol_fee_recipient` is not set to the default public key:

```
require!(
    new_protocol_fee_recipient != Pubkey::default(),
    ErrorCode::InvalidProtocolFeeRecipient
);
```

3.7 Permanent Freezing of `create_raydium_pool` Due to Seed Constraint

Description

The `create_raydium_pool` function is vulnerable to permanent freezing because the `pool_state` account is derived from a fixed seed. If the same `pool_state` is created on Raydium before this migration, the initialization will fail with the error `poolAlreadyCreated`, making the function permanently unusable.

The `pool_state` account is generated using the following seed constraint:

```
seeds = [
    POOL_SEED.as_bytes(),
    amm_config.key().as_ref(),
    token_0_mint.key().as_ref(),
    token_1_mint.key().as_ref(),
],
```

This constraint enforces a deterministic address for the pool, which causes conflicts if the pool has already been created on Raydium. This issue was prevalent in bonding curves, prompting the Raydium team to remove this constraint and allow any random account to serve as `pool_state`, provided it signs the transaction.

Updated Raydium Implementation

The Raydium team resolved this issue by modifying the constraint, as shown below:

```
/// CHECK: Initialize an account to store the pool state, init by contract
/// PDA account:
/// seeds = [
///     POOL_SEED.as_bytes(),
///     amm_config.key().as_ref(),
///     token_0_mint.key().as_ref(),
///     token_1_mint.key().as_ref(),
/// ],
///
/// Or random account: must be signed by CLI
#[account(mut)]
pub pool_state: UncheckedAccount<'info>,
```

Impact

- The `create_raydium_pool` function will be permanently frozen if the same pool has already been initialized on Raydium.

Recommendation

Remove the seed constraint to allow successful pool initialization. Instead, store the pool address off-chain or on-chain as needed.

Updated Implementation

```
/// CHECK: Initialize an account to store the pool state, init by cp-swap
#[account(mut)]
pub pool_state: UncheckedAccount<'info>,
```

This approach ensures that the pool creation is not blocked by pre-existing deployments and aligns with Raydium’s updated methodology.