Frank castle

Bruv audit

January 2025

Bruv Security Audit Report

Content

About Frank Castle 🦀

Frank Castle is a profissional 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 ,Pump.fun, LayerZero, Synthetix , Hydration ,DUB Social and several multi-million protocols.

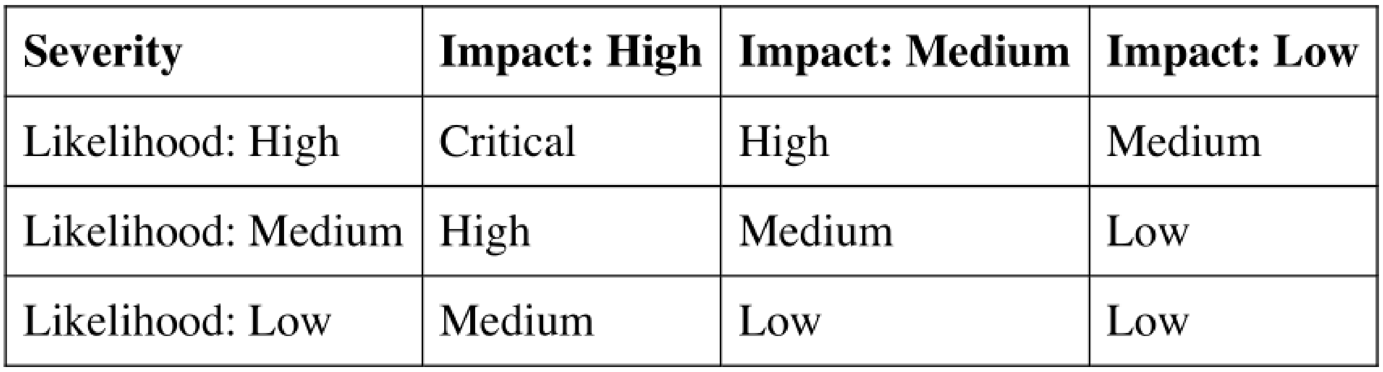
with more than ~25 Rust Audit , ~15 Solana Audits , and +100 criticals/highs found , All the reports can be found here

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

Disclaimer

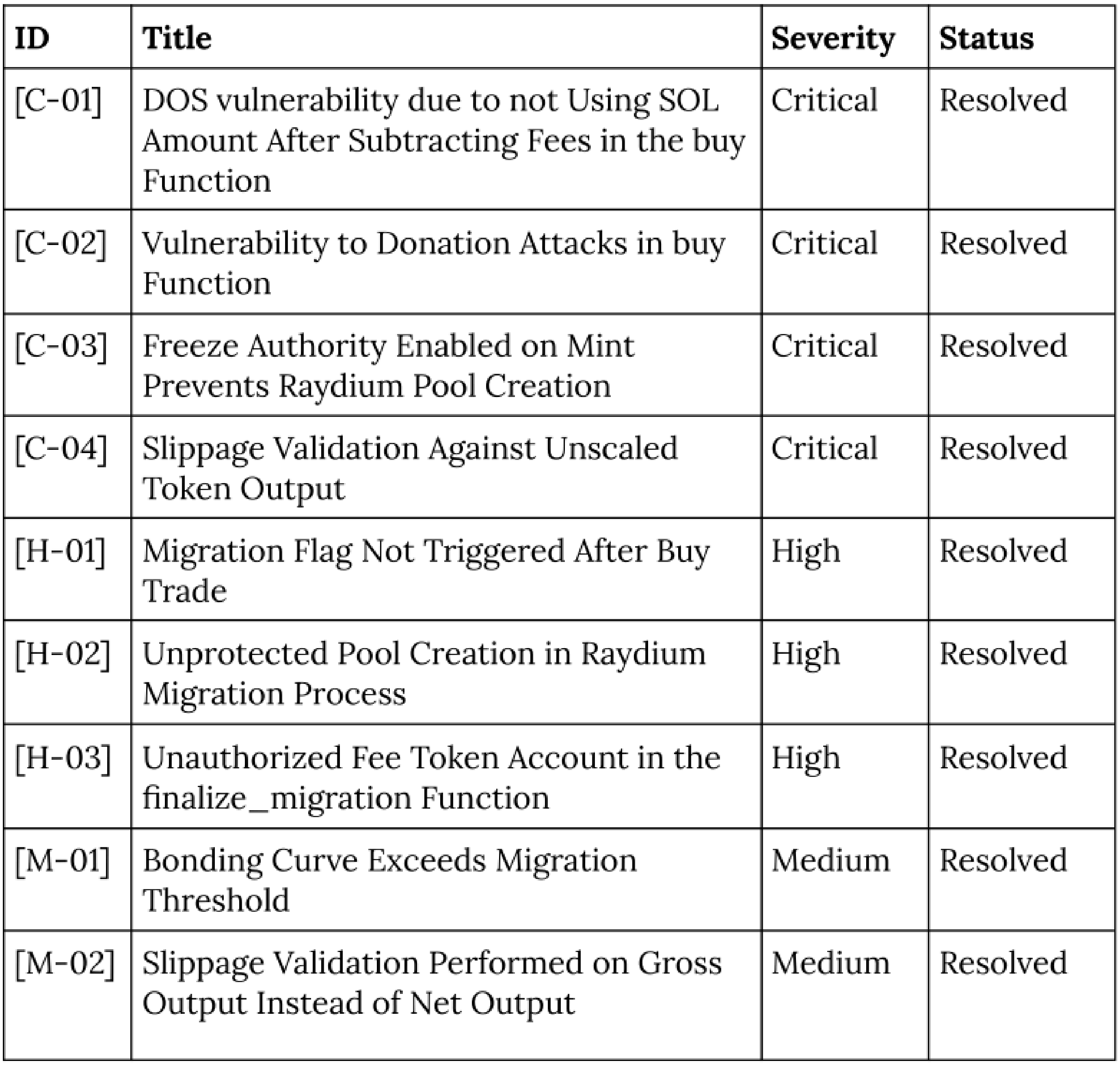
A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with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

Risk Classification



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

Summary of Findings



ID	Title	Severity	Status
[C-01]	DOS vulnerability due to not Using SOL Amount After Subtracting Fees in the buy Function	Critical	Resolved
[C-02]	Vulnerability to Donation Attacks in buy Function	Critical	Resolved
[C-03]	Freeze Authority Enabled on Mint Prevents Raydium Pool Creation	Critical	Resolved
[C-04]	Slippage Validation Against Unscaled Token Output	Critical	Resolved
[H-01]	Migration Flag Not Triggered After Buy Trade	High	Resolved
[H-02]	Unprotected Pool Creation in Raydium Migration Process	High	Resolved
[H-03]	Unauthorized Fee Token Account in the finalize_migration Function	High	Resolved
[M-01]	Bonding Curve Exceeds Migration Threshold	Medium	Resolved
[M-02]	Slippage Validation Performed on Gross Output Instead of Net Output	Medium	Resolved

Findings

1. Critical Findings

1.1. [C-01] DOS vulnerability because of not Using SOL Amount After Subtracting Fees in the buy Function

Impact: High

Likelihood: High

Description

In the `buy` function, the SOL amount provided by the user represents the total amount they want to pay, including fees. To ensure accurate token distribution, the fees must be subtracted from the SOL amount before calculating the tokens to be received by the user.

Failure to do so results in incorrect token calculations. Specifically, the tokens sent to the user will exceed the correct amount, causing the bonding curve to assume a higher SOL reserve than it actually has. This discrepancy creates a situation where small token sales push the bonding curve into a state where the new SOL reserve is incorrectly calculated as greater than the current reserve. This can lead to a revert due to a math error or an overflow when processing small sales.

Impact

This issue results in transaction failures when selling small amounts of tokens and disrupts the proper functioning of the bonding curve, affecting user experience and the overall stability of the system.

Recommendation

In the `calculate_tokens_out` function, use the `sol_amount` after subtracting the fees to accurately reflect the actual SOL being added to the bonding curve.

Code Fix

```
+ let sol_amount_sub_fees = sol_amount.checked_sub(fees).unwrap();
        let tokens_out = calculate_tokens_out(
-             sol_amount,
+             sol_amount_sub_fees,
        ctx.accounts.bonding_curve.virtual_sol_reserves
        )?;
```

1.2. [C-02] Vulnerability to Donation Attacks in `buy` Function

Severity

Impact: High

Likelihood: High

Description

The `buy` function checks the migration threshold using the account's `current_balance`, which includes all SOL in the program account. This approach exposes the protocol to donation attacks, where an attacker can donate SOL to artificially inflate the balance and prematurely trigger the migration without completely sell the tokens and distribute them to the users . Once triggered, this prevents token trading and potentially renders the protocol unusable.

Impact

This vulnerability allows an attacker to manipulate the migration process by triggering it without fully selling the available tokens. As a result, the token distribution remains incomplete, potentially leading to user dissatisfaction and loss of trust in the protocol. Additionally, since trading halts once migration is triggered, the protocol becomes non-operational, directly impacting its utility and revenue generation. In extreme cases, this could cause a complete shutdown of the protocol's functionality, severely affecting its users and stakeholders.

Recommendation

To mitigate donation attacks, the `buy` function should validate the increase in `virtual_sol_reserves` independently of the account's `current_balance`. Use `virtual_sol_reserves` to track the actual SOL reserves and update them only with legitimate trades. Replace the migration threshold check logic as follows:

```
        let new_balance = current_balance.checked_add(sol_after_fee)
            .ok_or(ErrorCode::MathError)?;
        //@audit critical donation attacks , we should validate the increase
in the virtual reserves
        if ctx.accounts.bonding_curve.check_migration_threshold(new_balance)
{

ctx.accounts.bonding_curve.update_migration_status(MigrationStatus::Triggere
d)?;
        }
```

we also need to change the function `check_migration_threshold` to check that the `virtual_sol_reserve` is greater than or equal `115 * 1_000_000_000` , this number represent the initial 30 sol in addition to the 85 sol collected from the bonding curve , appling this fix will prevent donation attacks since sending sol to the curve will not affect the virtual reserves.

Fix

```
fn buy() {
    if
ctx.accounts.bonding_curve.check_migration_threshold(ctx.accounts.bonding_cu
rve.virtual_sol) {

ctx.accounts.bonding_curve.update_migration_status(MigrationStatus::Triggere
d)?;
    }


+   // Use virtual_sol_reserves to validate migration threshold
+   let updated_virtual_reserves =
ctx.accounts.bonding_curve.virtual_sol_reserves
+       .checked_add(sol_after_fee)
+       .ok_or(ErrorCode::Overflow)?;
+   ctx.accounts.bonding_curve.virtual_sol_reserves =
updated_virtual_reserves;
}
    pub fn check_migration_threshold(&self ,  balance: u64) -> bool {
-              balance >= MIGRATION_THRESHOLD
+       self.virtual_sol_reserves >= (115 * 1_000_000_000) // 30 initial SOL
+ 85 SOL from bonding curve
    }
```

1.3. [C-03] Freeze Authority Enabled on Mint Prevents Raydium Pool Creation

Severity

Impact: High
Likelihood: High

Description

In the `create` function, the mint account is initialized with its freeze authority set to the mint authority:

```
pub struct Create<'info> {
    #[account(
        init,
        payer = payer,
        mint::decimals = 6,
        mint::authority = mint_authority,
        mint::freeze_authority = mint_authority
    )]
    pub mint: Account<'info, Mint>,
}
```

According to the Raydium documentation SPL tokens must have the freeze authority feature **disabled** for a liquidity pool to be created. Enabling freeze authority prevents migration to Raydium and the creation of liquidity pools pairing these tokens.

This issue will have a **critical impact** on the protocol, as it will permanently block migration to Raydium and render the protocol unusable. Additionally, all funds collected for this purpose will remain locked, leading to a complete failure in achieving the protocol's objectives.

Recommendations

Disable the freeze authority when initializing the mint account.

Recommended Implementation

```diff
pub struct Create<'info> {
    #[account(
        init,
        payer = payer,
        mint::decimals = 6,
        mint::authority = mint_authority,
-         mint::freeze_authority = mint_authority
    )]
    pub mint: Account<'info, Mint>,
}
```

1.4. [C-04] Slippage Validation Against Unscaled Token Output

Severity

Impact: High
Likelihood: High

Description

The `buy` function contains a critical flaw in the slippage validation logic. The slippage parameter (`min_token_amount`) is intended to protect users by ensuring the trade results in a minimum acceptable amount of tokens. However, this parameter is validated against the unscaled `tokens_out` value, which does not account for the token's decimal scaling (6 decimals in this case). Since the slippage parameter represents the token amount with 6 decimals, comparing it against the unscaled value renders the validation logic invalid.

This issue allows trades to proceed with prices outside the acceptable slippage range, leading to a potential loss of funds for users.

Affected Code

```
pub fn buy(ctx: Context<Buy>, sol_amount: u64, min_token_amount: u64) ->
Result<()> {
    let tokens_out = calculate_tokens_out(
        sol_amount,
        ctx.accounts.bonding_curve.virtual_sol_reserves
    )?;
    msg!("Calculated tokens_out: {}", tokens_out);

    // Check min tokens
    require!(tokens_out >= min_token_amount, ErrorCode::ExcessiveSlippage);

    let scaled_virtual_tokens = tokens_out.checked_mul(1_000_000)
        .ok_or_else(|| {
            msg!("Error scaling tokens_out");
            ErrorCode::Overflow
        })?;
    msg!("Scaled virtual tokens: {}", scaled_virtual_tokens);
}
```

Impact

- Invalid slippage validation allows users to execute trades at unintended prices.
- This can result in significant financial losses for users, undermining the protocol's reliability and security.

Recommendation

To ensure proper slippage validation, compare the `min_token_amount` parameter against the scaled token output (`scaled_virtual_tokens`) rather than the unscaled value. Update the validation logic as follows:

```
require!(scaled_virtual_tokens >= min_token_amount,
ErrorCode::ExcessiveSlippage);
```

2. High Findings

2.1. [H-01] Migration Flag Not Triggered After Buy Trade

Severity

Impact: High

Likelihood: Medium

Description

In the current implementation of the `buy` function, the migration threshold is checked before the bonding curve is updated. This leads to a situation where a buy trade can result in a token amount that exceeds the migration limit, but the migration flag will not be set. Consequently, the trading will continue even though the migration threshold has been surpassed.

For instance, if the current state of the curve results in 84 SOL in funds, and the trade pushes this to 86 SOL, the migration should be triggered. However, since the flag is set before the trade, the migration will not be initiated, which could leave the system in an inconsistent state.

The current implementation does not account for the updated reserves post-trade, which would impact whether the migration threshold has been exceeded.

Recommendations

To fix this issue, the migration threshold check should occur **after** the buy transaction has been completed, ensuring that the migration flag is set based on the updated reserves. This way, if the migration threshold is met after the trade, the flag will be correctly triggered, stopping further trading as expected.

Here’s the updated implementation:

```
pub fn buy(ctx: Context<Buy>, sol_amount: u64, min_token_amount: u64) ->
Result<()> {
    msg!("Starting buy operation with {} sol", sol_amount);
    msg!("Current virtual_sol_reserves: {}",
ctx.accounts.bonding_curve.virtual_sol_reserves);
    msg!("Current virtual_token_reserves: {}",
ctx.accounts.bonding_curve.virtual_token_reserves);

    // Check flag to ensure program is not paused
    require!(!ctx.accounts.bonding_curve.is_paused(),
ErrorCode::ProgramPaused);


ctx.accounts.bonding_curve.log_migration_state(&ctx.accounts.bonding_curve.t
o_account_info());

    // Make sure the migration is not in progress
    require!(
        ctx.accounts.bonding_curve.can_trade(),
        ErrorCode::MigrationInProgress
    );

    // Perform the buy operation and calculate tokens out
    let tokens_out = calculate_tokens_out(
        sol_amount,
        ctx.accounts.bonding_curve.virtual_sol_reserves
    )?;

    msg!("Calculated tokens_out: {}", tokens_out);

    // Update reserves based on the trade


    // Check migration threshold AFTER the trade
    if
ctx.accounts.bonding_curve.check_migration_threshold(ctx.accounts.bonding_cu
rve.virtual_sol_reserves) {

ctx.accounts.bonding_curve.update_migration_status(MigrationStatus::Triggere
d)?;
    }

    // Ensure the user gets the expected amount of tokens (minimum check)
    require!(tokens_out >= min_token_amount,
ErrorCode::InsufficientOutputAmount);

    Ok(())
}
```

Key Changes:

- The migration threshold check is moved **after** the buy operation to ensure that the updated state of the reserves is considered when evaluating whether migration should be triggered.
- Reserves are updated after the buy operation, ensuring that the new balance reflects the trade's impact before triggering the migration flag.

2.2. [H-02] Unprotected Pool Creation in Raydium Migration Process

Severity

Impact: High

Likelihood: Medium

Description

The current implementation of the migration process from the existing AMM to Raydium's Pools is vulnerable to a race condition that could allow an attacker to block the migration.

The `proxy_initialize` function attempts to create a new pool in Raydium and force the `pool_state` account to be driven from this seed

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

This creates a window of opportunity for an attacker to preemptively create the same pool in Raydium, effectively blocking the official migration process.

The vulnerability's two main factors:

1. Lack of Pre-existence Check: The migration code does not verify whether a pool with the same tokens and AMM config already exists in Raydium before attempting to create one.
2. No Exclusive Creation Rights: There's no mechanism to ensure that only the official migration process can create the new pool in Raydium.

Raydium Team has addressed this issue and allowed the pool state to be an arbitrary address , but it should sign the tx by cli as per docs here

```

    /// CHECK: Initialize an account to store the pool state
    /// 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>,
```

A "Pool already created" error could occur if you try to initialize a pool that was already initialized.

https://docs.raydium.io/raydium/pool-creation-faq#pool-already-created-error

Impact

This vulnerability will lead to permanent DOS to the migration process.

Recommendations

```diff
    #[account(
        mut,
-        seeds = [
-            POOL_SEED.as_bytes(),
-            amm_config.key().as_ref(),
-            token_0_mint.key().as_ref(),
-            token_1_mint.key().as_ref(),
-        ],
-        seeds::program = cp_swap_program,
-        bump,
    )]
    pub pool_state: UncheckedAccount<'info>,
```

remove those seeds contraints to allow pool initialization at any random address.

2.3. [H-03] Unauthorized Fee Token Account in the `finalize_migration` Function

Severity

Impact: High
Likelihood: Medium

Description

In the `finalize_migration` function, the `fee_token_account` is defined as follows:

```rust
/// CHECK: Will be initialized if needed
#[account(mut)]
pub fee_token_account: AccountInfo<'info>,
```

The `fee_token_account` is not properly validated, and since this function is permissionless, it allows an attacker to set this account to a malicious address. This could potentially enable unauthorized transfers or other malicious activities involving the fee tokens.

Recommendations

To prevent such attacks, the `fee_token_account` should be validated similarly to how it is done in the `emergency_withdraw` function. Use the following pattern to ensure proper validation:

```rust
#[account(
    init_if_needed,
    payer = authority,
    associated_token::mint = mint,
    associated_token::authority = fee_account
)]
pub fee_token_account: Account<'info, TokenAccount>,
```

This approach ensures the `fee_token_account` is securely initialized and tied to the correct mint and authority, mitigating the risk of unauthorized access.

You also need to add the payer account to the list of the accounts .

3. Medium Findings

3.1. [M-01] Bonding Curve Exceeds Migration Threshold

Severity

Impact: Medium

Likelihood: Medium

Description

The `available_sol_amount` function needs to be added to ensure that the amount of SOL traded during the migration does not exceed the predefined migration threshold of 85 SOL. Without this validation, it is possible for the bonding curve to receive more SOL than intended, leading to undesired protocol behavior or potential fund mismanagement. This function calculates the remaining amount of SOL that can be traded by comparing the `migration_threshold` with the current `virtual_sol_reserve`. If the remaining amount is less than the user's intended `sol_amount`, the trade is capped at the remaining amount. The logic can be expressed as follows:

```
fn available_sol_amount(sol_amount: u64, virtual_sol_reserve: u64,
migration_threshold: u64, sol_amount_sub_fees: u64) -> u64 {
    let remaining = migration_threshold.saturating_sub(virtual_sol_reserve);
    std::cmp::min(sol_amount_sub_fees, remaining)
}
```

This function should be invoked before calculating the `tokens_out` to ensure the bonding curve remains within the migration threshold. Below is an example of how to incorporate it:

```
let sol_amount = available_sol_amount(
    user_sol_amount,
    ctx.accounts.bonding_curve.virtual_sol_reserves,
    migration_threshold,
    sol_amount_sub_fees
);

let tokens_out = calculate_tokens_out(
    sol_amount,
    ctx.accounts.bonding_curve.virtual_sol_reserves
)?;

msg!("Calculated tokens_out: {}", tokens_out);
```

This adjustment prevents the bonding curve from exceeding the migration threshold, maintaining protocol integrity and ensuring proper fund allocation.

Recommendations

1. Integrate the `available_sol_amount` function into the migration logic to cap SOL trades at the migration threshold.

3.2. [M-02] Slippage Validation Performed on Gross Output Instead of Net Output

Severity

Impact: Medium
Likelihood: Medium

Description

The `min_sol_output` parameter is designed to protect users from excessive slippage by ensuring that the trade price is acceptable. However, in the current implementation, the slippage check is performed on the **gross output** (the amount before subtracting fees) rather than the **net output** (the amount the user will actually receive after fees).

This introduces a vulnerability, as users may receive less SOL than expected, especially when fees are significant. For example, the gross output could meet the `min_sol_output` threshold, but the actual amount received (net output) may fall below the acceptable level due to fees.

Relevant Code

```
let sol_output = calculate_sol_output(
    token_amount,
    ctx.accounts.bonding_curve.virtual_sol_reserves,
    ctx.accounts.bonding_curve.virtual_token_reserves
)?;
msg!("SOL output: {}", sol_output);

// Check min output
msg!("Check min output SOL {} >= {}", sol_output, min_sol_output);
// @audit high vulnerability: the slippage should be applied on the amount
that the user will receive (net output), not the gross output
```

```
require!(sol_output >= min_sol_output, ErrorCode::ExcessiveSlippage);

// Calculate fee
let fee = sol_output.checked_mul(FEE_NUMERATOR)
    .ok_or(ErrorCode::MathError)?
    .checked_div(FEE_DENOMINATOR)
    .ok_or_else(|| {
        msg!("Failed to calculate fee from {}", sol_output);
        ErrorCode::MathError
    })?;
msg!("Fee (1.5%): {}", fee);

// Calculate net output
let net_output = sol_output.checked_sub(fee)
    .ok_or_else(|| {
        msg!("Failed to subtract fee {} from {}", fee, sol_output);
        ErrorCode::MathError
    })?;
msg!("Net output: {}", net_output);
```

Impact

This issue can result in **loss of funds for users**, as trades may execute at unintended prices, violating their acceptable slippage settings.

Recommendations

Update the implementation to validate the `min_sol_output` parameter against the **net output** value (the amount of SOL received after fees are deducted).

Recommended Implementation

```
let sol_output = calculate_sol_output(
    token_amount,
    ctx.accounts.bonding_curve.virtual_sol_reserves,
    ctx.accounts.bonding_curve.virtual_token_reserves
)?;
msg!("SOL output: {}", sol_output);

// Calculate fee
let fee = sol_output.checked_mul(FEE_NUMERATOR)
    .ok_or(ErrorCode::MathError)?
    .checked_div(FEE_DENOMINATOR)
    .ok_or_else(|| {
        msg!("Failed to calculate fee from {}", sol_output);
        ErrorCode::MathError
    })?;
msg!("Fee (1.5%): {}", fee);

// Calculate net output
let net_output = sol_output.checked_sub(fee)
    .ok_or_else(|| {
        msg!("Failed to subtract fee {} from {}", fee, sol_output);
        ErrorCode::MathError
    })?;
msg!("Net output: {}", net_output);

// Check min output against net output
msg!("Check min output SOL {} >= {}", net_output, min_sol_output);
require!(net_output >= min_sol_output, ErrorCode::ExcessiveSlippage);
```

Key Benefits of This Fix

- Ensures users receive at least the expected amount of SOL after fees, honoring their slippage settings.