Pump Security Review

Pashov Audit Group

Conducted by: ubermensch, FrankCastle, Koolex, carrotsmuggler, defsec, dirk_y

October 11th - October 26th

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of multiple teams of some of the best smart contract security researchers in the space. Having a combined reported security vulnerabilities count of over 1000, the group strives to create the absolute very best audit journey possible - although 100% security can never be guaranteed, we do guarantee the best efforts of our experienced researchers for your blockchain protocol. Check our previous work here or reach out on Twitter @pashovkrum.

2. 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.

3. Introduction

A time-boxed security review of the **pump-fun/pump-contracts-solana** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About Pump

Pump on Solana is a platform for launching SPL coins that can be traded on a bonding curve without needing to provide initial liquidity. Once the coin reaches a particular market cap, liquidity is deposited from the bonding curve to Raydium, and the received LP tokens are burnt.

5. Risk Classification

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

5.1. Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- Low - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

5.2. Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

5.3. Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

6. Security Assessment Summary

review commit hash - 72cb997ca547dabcb58116f16a4f888337e551e5

fixes review commit hash - bd794c918e133c3c125c0ef43068b7b2cc190c02

Scope

The following smart contracts were in scope of the audit:

- `lib`
- `migrate`
- `realloc_bonding_curve`
- `realloc_global`
- `update_global_authority`

7. Executive Summary

Over the course of the security review, ubermensch, FrankCastle, Koolex, carrotsmuggler, defsec, dirk_y engaged with Pump to review Pump. In this period of time a total of **13** issues were uncovered.

Protocol Summary

Protocol Name	Pump
Repository	https://github.com/pump-fun/pump-contracts-solana
Date	October 11th - October 26th
Protocol Type	Bonding Curve tokensale

Findings Count

Severity	**Amount**
High	1
Medium	1
Low	11
Total Findings	**13**

Summary of Findings

ID	Title	Severity	Status
[H-01]	Front-running on the migration process	High	Resolved
[M-01]	Insufficient Token Handling in buy()	Medium	Resolved
[L-01]	Virtual Reserves must exceed or match Real Reserves	Low	Resolved
[L-02]	Rounding down of fees in get_fee()	Low	Resolved
[L-03]	State inconsistency due to Solana rollback	Low	Acknowledged
[L-04]	Default withdraw authority will cause loss of funds during the migration	Low	Acknowledged
[L-05]	Pool migration fee is under constrained	Low	Resolved
[L-06]	Missing production program ID Configuration	Low	Acknowledged
[L-07]	Add validation check for duplicate authority	Low	Acknowledged
[L-08]	Tokens can be donated before migration to reduce the listing price after migration	Low	Acknowledged
[L-09]	Missing checked arithmetic operations on the lib	Low	Resolved
[L-10]	Bonding can be delayed indefinitely	Low	Acknowledged
[L-11]	Extend account can be called to unnecessarily waste native tokens	Low	Resolved

8. Findings

8.1. High Findings

[H-01] Front-running on the migration process

Severity

Impact: High

Likelihood: Medium

Description

An attacker can front-run the migration transaction by sending a small amount of token to the `pool_authority_mint_account`, which can cause the account closure to fail and disrupt the entire migration process.

Relevant code:

```
fund_pool_authority_mint_account_from_associated_bonding_curve(&ctx)?;


        let sol_amount = ctx.accounts.bonding_curve.real_sol_reserves - pool_migratio
    fund_pool_authority_wsol_account_from_bonding_curve(&ctx, sol_amount)?;

    create_pool(&ctx, sol_amount)?;

    close_token_account(
        &ctx,
        ctx.accounts.pool_authority_mint_account.to_account_info(),
    )?;
```

The attack scenario is as follows:

1. An attacker submits their own transaction to send a dust amount to a pre-created `pool_authority_mint_account`.
2. If the attacker's transaction is processed before the migration transaction, `pool_authority_mint_account` will have a non-zero balance.

3. When the migration transaction executes, the operation will fail because the account has a non-zero balance.

Recommendations

Implement Balance Check and Sweep: At the beginning of the migration process, check the balance of `pool_authority_mint_account` and sweep any unexpected funds to a designated address. For example:

```
let unexpected_balance = ctx.accounts.pool_authority_mint_account.amount;
if unexpected_balance > 0 {
    // Transfer unexpected balance to a designated address
    sweep_token(ctx, unexpected_balance, designated_address)?;
}
// Proceed with migration process
```

8.2. Medium Findings

[M-01] Insufficient Token Handling in `buy()`

Severity

Impact: Medium

Likelihood: Medium

Description

In the `buy` function, the user specifies an `amount` of tokens to buy from the bonding curve by paying SOL. However, if the requested `amount` exceeds the available `real_token_reserves`, the function reverts with an underflow error. This issue occurs when the function attempts to subtract an `amount` greater than the available tokens from the `real_token_reserves`. As a result, valid swaps that could partially fulfill the user's request are prevented, and the bonding curve is not marked as completed.

For example, in the `buy` function:

```
pub fn buy(ctx: Context<Buy>, amount: u64, max_sol_cost: u64) -> Result<()> {
    // calculate the sol cost and fee
    let sol_cost = ctx.accounts.bonding_curve.buy_quote(amount as u128);
    let fee = ctx.accounts.global.get_fee(sol_cost);

    // check that the sol cost is within the slippage tolerance
    require!(
        sol_cost + fee <= max_sol_cost,
        PumpError::TooMuchSolRequired
    );
    require_keys_eq!(
        ctx.accounts.associated_bonding_curve.mint,
        ctx.accounts.mint.key(),
        PumpError::MintDoesNotMatchBondingCurve
    );
    require!(
        !ctx.accounts.bonding_curve.complete,
        PumpError::BondingCurveComplete
    );

    // update the bonding curve parameters
    ctx.accounts.bonding_curve.virtual_token_reserves -= amount;
    --> ctx.accounts.bonding_curve.real_token_reserves -= amount;  // Reverts if
    // `amount` is greater than reserves
    ctx.accounts.bonding_curve.virtual_sol_reserves += sol_cost;
```

If the `amount` exceeds the available tokens in `real_token_reserves`, the function will fail, halting the execution of the swap instead of selling the remaining tokens to the user and marking the bonding curve as complete.

Impact

This issue prevents partial purchases when the requested token `amount` exceeds the remaining tokens in the `real_token_reserves`. Users may be unable to buy tokens even though a portion of their request could be fulfilled. Additionally, the bonding curve will remain incomplete, causing potential disruption to the functionality of the protocol and frustrating user experience.

Recommendation

Modify the `buy` function to handle the case where the requested `amount` exceeds the available tokens in the `real_token_reserves`. If the requested `amount` is greater, the function should sell the remaining tokens in the reserve to the user, update the bonding curve, and mark it as complete. This will prevent the underflow error and allow valid transactions to be processed.

Suggested Fix:

```
pub fn buy(ctx: Context<Buy>, amount: u64, max_sol_cost: u64) -> Result<()> {

+
+     // Check if the requested amount exceeds the remaining tokens in the real reserv
+    if amount > ctx.accounts.bonding_curve.real_token_reserves {
+
+         amount = ctx.accounts.bonding_curve.real_token_reserves;  // Adjust to sell
+
+         ctx.accounts.bonding_curve.complete = true;  // Mark the bonding curve as co
+    }

    // Proceed with the regular buy logic
    let sol_cost = ctx.accounts.bonding_curve.buy_quote(amount as u128);
    let fee = ctx.accounts.global.get_fee(sol_cost);

    // Check slippage tolerance and other conditions...
}
```

8.3. Low Findings

[L-01] Virtual Reserves must exceed or match Real Reserves

Description

The bonding curve's sell logic relies on virtual reserves for calculations. However, if the real token reserves fall below the virtual token reserves, underflow can occur, preventing the complete migration of the pool. Currently, the logic does not enforce the condition that the `virtual_token_reserves` must be greater than or equal to the `real_token_reserves`. This can lead to a DoS (Denial of Service) issue where the migration of the bonding curve is blocked indefinitely, as the `real_token_reserves` may never reach zero.

In the `buy_quote` function, the virtual reserves are used in calculations:

```
pub fn buy_quote(&self, amount: u128) -> u64 {
    let virtual_sol_reserves = self.virtual_sol_reserves as u128;
    let virtual_token_reserves = self.virtual_token_reserves as u128;
    let sol_cost: u64 =
        ((amount * virtual_sol_reserves) /
          (virtual_token_reserves - amount)) as u64;

    sol_cost + 1 // always round up
}
```

When tokens are sold, both real and virtual reserves are updated:

```
ctx.accounts.bonding_curve.virtual_token_reserves -= amount;
ctx.accounts.bonding_curve.real_token_reserves -= amount;
ctx.accounts.bonding_curve.virtual_sol_reserves += sol_cost;
ctx.accounts.bonding_curve.real_sol_reserves += sol_cost;
```

If the `real_token_reserves` do not reach zero, the bonding curve cannot complete migration, which can result in the following issue:

```
if ctx.accounts.bonding_curve.real_token_reserves == 0 {
    ctx.accounts.bonding_curve.complete = true;
}
```

This issue can lead to a perpetual DoS where the migration of the bonding curve is blocked because the real reserves never reach zero, due to an underflow or misalignment between the real and virtual reserves. As a result, liquidity remains locked, preventing users from executing valid swaps and fully migrating the pool to another AMM.

Recommendation

Ensure that virtual reserves are always greater than or equal to real reserves during the setup of the bonding curve, and update the logic for marking the bonding curve as complete during migration. Specifically:

1. **Set a Condition for Migration**: Use a calculated threshold (difference between virtual and real reserves) to mark the bonding curve as complete:

```
let migration_limit = virtual - real;

if ctx.accounts.bonding_curve.virtual_token_reserves <= migration_limit {
    ctx.accounts.bonding_curve.complete = true;
}
```

2. **Check During Parameter Setting**: Enforce that virtual reserves exceed real reserves during initialization and parameter updates, to prevent underflow:

```
pub fn set_params(
    ctx: Context<SetParams>,
    fee_recipient: Pubkey,
    initial_virtual_token_reserves: u64,
    initial_virtual_sol_reserves: u64,
    initial_real_token_reserves: u64,
    token_total_supply: u64,
    fee_basis_points: u64,
    withdraw_authority: Pubkey,
    disable_withdraw: bool,
    enable_migrate: bool,
    pool_migration_fee: u64,
    creator_fee: u64,
) -> Result<()> {
    require!(ctx.accounts.global.initialized, PumpError::NotInitialized);
    require!(
        initial_virtual_token_reserves >= initial_real_token_reserves,
        PumpError::VirtualLessThanRealReserves
    );
}
```

By enforcing these changes, the bonding curve will be able to gracefully handle pool migration and prevent underflow errors, ensuring liquidity can be fully withdrawn and swapped.

[L-02] Rounding down of fees in `get_fee()`

Description

The `get_fee` function is responsible for calculating the protocol fee based on a percentage of the transaction amount. It currently rounds down the fee, which can result in scenarios where the fee is reduced to zero, especially for small transactions. This can cause the protocol to lose potential revenue from fees.

The rounding down issue is evident in the `get_fee` function, as shown below:

```
pub fn get_fee(&self, amount: u64) -> u64 {
    let fee = (amount as u128 * self.fee_basis_points as u128) / 10_000;
    fee as u64
}
```

This function is used in both the `buy` and `sell` functions to calculate the fees deducted from the SOL amount, but the current implementation can lead to a loss of fees, as it rounds down instead of rounding up.

Rounding the fee down, especially for small amounts, can result in a significant loss of fees for the protocol over time. For smaller transactions, the fee could effectively become zero, which reduces the protocol's income. Over many transactions, this issue accumulates and impacts the protocol's long-term sustainability.

Recommendation

To ensure that the protocol receives the correct fee, the `get_fee` function should round the fee up instead of down. This guarantees that even in cases where the fee calculation results in a fractional value, it is rounded up to the nearest integer to prevent fee loss.

Suggested Fix:

```
pub fn get_fee(&self, amount: u64) -> u64 {
    let fee =
    //(amount as u128 * self.fee_basis_points as u128 + 9_999) / 10_000;  // Round up
    fee as u64
}
```

By adding `9_999` before division, the fee calculation will always round up, ensuring that the protocol does not lose revenue due to small or fractional fee

amounts.

[L-03] State inconsistency due to Solana rollback

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

- **Withdraw Pause Mechanism:** If the protocol uses a pause mechanism to prevent withdrawals due to a detected security threat, a Solana rollback could revert the paused state back to active. This would expose the protocol to the very security threat it was attempting to mitigate.
- **Setting Config Parameters:** In the event of a Solana rollback, global configuration parameters could become outdated. This would pose a significant risk to the system, as the protocol could operate with old, invalid settings, potentially leading to malfunction or vulnerabilities.

To mitigate the risks of Solana rollbacks, implement the following strategies:

1. **Detect Outdated Configurations:** Utilize the `LastRestartSlot` sysvar to check whether the global configuration has been affected by a rollback. If the configuration is outdated, the protocol should automatically pause to prevent further actions until an admin intervenes.
2. **Add `last_updated_slot` Field:** Include a `last_updated_slot` field in the bonding curve state to track when the configuration was last updated.
3. **Outdated Configuration Check Function:** Implement a function to verify if the global configuration is outdated. If it is, take necessary measures to prevent further operations until the configuration is updated.

Here is an example function to check for the outdated global state:

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

[L-04] Default withdraw authority will cause loss of funds during the migration

The global `withdraw_authority` in the bonding curve is responsible for managing liquidity during the migration process, including receiving migration fees and withdrawing liquidity. This authority can either be a valid public key or the default public key (`Pubkey::default()`).

In the `withdraw` function, if the `withdraw_authority` is set to the default public key, the code verifies the user's public key against a constant key (`config_feature::withdraw_authority::ID`):

```
if ctx.accounts.global.withdraw_authority == Pubkey::default() {
    require_keys_eq!(
        config_feature::withdraw_authority::ID,
        ctx.accounts.user.key(),
        PumpError::NotAuthorized
    );
} else {
    require_keys_eq!(
        ctx.accounts.user.key(),
        ctx.accounts.global.withdraw_authority,
        PumpError::NotAuthorized
    );
}
```

However, in the `migrate` function, this case is not handled. The migration process is permissionless, and the `withdraw_authority` is used directly without validation. If the `withdraw_authority` is the default public key, it will be used to receive the migration fees and pool authority's lamports during the migration:

```
transfer_from_pool_authority(
    &ctx,
    ctx.accounts.withdraw_authority.to_account_info(),
    ctx.accounts.pool_authority.lamports(),
)?;
```

Importantly, setting the `withdraw_authority` to the default key is not an administrative mistake. This scenario is already handled in the `withdraw` function, so it must also be handled similarly in the `migrate` function to prevent misuse.

If the global `withdraw_authority` is set to the default public key, it will lead to a loss of migration fees and the rent from the pool authority's accounts, as the

default public key would receive these funds.

Implement the same validation logic for the `withdraw_authority` in the `migrate` function as is done in the `withdraw` function. Ensure that the `withdraw_authority` is correctly set before the migration proceeds.

Add the following code to handle cases where the `withdraw_authority` is set to the default public key:

```
if ctx.accounts.global.withdraw_authority
        == Pubkey::default() {
    withdraw_authority = config_feature::withdraw_authority::ID;
}
```

This ensures that the correct authority is used during the migration process and prevents potential loss of funds.

[L-05] Pool migration fee is under constrained

In the in-scope diff to `lib.rs` the following constraint was added to `pool_migration_fee`:

```
require_gt!(
 pool_migration_fee,
 creator_fee,
 PumpError::CreatorFeeShouldBeLessThanPoolMigrationFee
);
```

This is important during migration as the lamports transferred to the pool authority account cover the creator fee that is sent to the creator. However, technically this is under-constrained since `pool_migration_fee - creator_fee` has to cover the rent exemption lamports to open 7 different accounts:

- `pool_base_token_account`
- `pool_quote_token_account`
- `pool_authority_mint_account`
- `pool_authority_mint_account`
- `pool_account`
- `lp_mint`
- `user_pool_token_account`

It might be worth making that constraint a bit stricter by adding a buffer (rather than calculating the exact lamports required for each account type).

[L-06] Missing production program ID Configuration

The code contains an unimplemented TODO for the pump_amm program ID configuration:

```
declare_program!(pump_amm);
```

This missing configuration presents several risks:

- Potential deployment failures
- Incorrect program invocations

Consider adding proper program ID configuration.

[L-07] Add validation check for duplicate authority

Currently, the `update_global_authority_impl` function allows updating the global authority to the same address as the current authority.

```rust
pub fn update_global_authority_impl
  (ctx: Context<UpdateGlobalAuthority>) -> Result<()> {
    ctx.accounts.global.authority = *ctx.accounts.new_authority.key;

    let update_global_authority_event = UpdateGlobalAuthorityEvent {
        global: ctx.accounts.global.key(),
        authority: *ctx.accounts.authority.key,
        new_authority: *ctx.accounts.new_authority.key,
    };

    emit!(update_global_authority_event);
    emit_cpi!(update_global_authority_event);

    Ok(())
}
```

Add validation to prevent updating authority to the same address, returning an appropriate error.

[L-08] Tokens can be donated before migration to reduce the listing price after migration

During migration, the entire token balance of the bonding curve is transferred to the `pool_authority` ATA.

```rust
token::transfer(
        // Other stuff
        ctx.accounts.associated_bonding_curve.amount,
    )?;
```

Then this entire amount is recorded and transferred during the pool creation process.

```rust
let mint_amount = pool_authority_mint_account(&ctx)?.amount;
create_pool(&ctx, mint_amount, sol_amount)?;
```

So if a user donates any tokens to the bonding curve or to the ATA account by pre-initializing it, they will all eventually get transferred to the pool, which will affect the listing price.

During bonding, virtual reserves start at 30 sol, 1073000 tokens, and end at 115 sol, 279913 tokens, which gives a price of 2434 token/sol.

When migrated, the new pool is created with 85 sol and 206900 tokens, with a listing price of 2434.1 token.sol.

But if 100000 tokens have been donated to the bonding curve, the pool will be created with 85 sol and 306900 tokens, with a listing price of 3610 tokens.sol.

So the listing price can be dropped without any buy or sell. The issue is that dextools, screeners, and all other apps track contract-emitted events to track buys and sells. This type of donation won't show up anywhere while affecting the price. This can be confusing to the users since it's basically a `hidden` way to dump token price.

Consider transferring only the `real_token_reserves` to the pool. The rest of the tokens can be sent to the pool_authority. This will ensure the tokens always get listed at the same price of 2434 token/sol after bonding, and not get affected by donations during the bonding process.

[L-09] Missing checked arithmetic operations on the lib

Multiple locations in the protocol lack checked arithmetic operations, potentially leading to underflows/overflows in calculations.

Impact:

- Silent underflow/overflow

  Cargo.toml enables overflow check by default. In practice, there is no overflow due to that check.

- Division by zero

Code Location : /src/lib.rs

1. **Buy Operation Arithmetic**

```
// Current implementation
ctx.accounts.bonding_curve.virtual_token_reserves -= amount;
ctx.accounts.bonding_curve.real_token_reserves -= amount;
ctx.accounts.bonding_curve.virtual_sol_reserves += sol_cost;
ctx.accounts.bonding_curve.real_sol_reserves += sol_cost;

// Required: Checked arithmetic
ctx.accounts.bonding_curve.virtual_token_reserves = ctx.accounts
    .bonding_curve.virtual_token_reserves
    .checked_sub(amount)
    .ok_or(PumpError::ArithmeticUnderflow)?;

ctx.accounts.bonding_curve.real_token_reserves = ctx.accounts
    .bonding_curve.real_token_reserves
    .checked_sub(amount)
    .ok_or(PumpError::ArithmeticUnderflow)?;

ctx.accounts.bonding_curve.virtual_sol_reserves = ctx.accounts
    .bonding_curve.virtual_sol_reserves
    .checked_add(sol_cost)
    .ok_or(PumpError::ArithmeticOverflow)?;

ctx.accounts.bonding_curve.real_sol_reserves = ctx.accounts
    .bonding_curve.real_sol_reserves
    .checked_add(sol_cost)
    .ok_or(PumpError::ArithmeticOverflow)?;
```

2. Sell Operation Arithmetic

```
// Current implementation
ctx.accounts.bonding_curve.virtual_token_reserves += amount;
ctx.accounts.bonding_curve.real_token_reserves += amount;
ctx.accounts.bonding_curve.virtual_sol_reserves -= sol_output;
ctx.accounts.bonding_curve.real_sol_reserves -= sol_output;

// Required: Checked arithmetic
ctx.accounts.bonding_curve.virtual_token_reserves = ctx.accounts
    .bonding_curve.virtual_token_reserves
    .checked_add(amount)
    .ok_or(PumpError::ArithmeticOverflow)?;
```

3. Fee Calculations

```
// Current implementation
(u128::from(amount) * u128::from(self.fee_basis_points) + 9_999) / 10_000

// Required: Checked arithmetic
let fee = u128::from(amount)
    .checked_mul(u128::from(self.fee_basis_points))
    .ok_or(PumpError::ArithmeticOverflow)?
    .checked_add(9_999)
    .ok_or(PumpError::ArithmeticOverflow)?
    .checked_div(10_000)
    .ok_or(PumpError::ArithmeticError)?;
```

4. Buy Quote Calculation

```
// Current implementation
let sol_cost = (amount * virtual_sol_reserves) /
  (virtual_token_reserves - amount);
sol_cost + 1

// Required: Checked arithmetic
let denominator = virtual_token_reserves
    .checked_sub(amount)
    .ok_or(PumpError::ArithmeticUnderflow)?;

require_gt!(denominator, 0, PumpError::DivisionByZero);

let sol_cost = amount
    .checked_mul(virtual_sol_reserves)
    .ok_or(PumpError::ArithmeticOverflow)?
    .checked_div(denominator)
    .ok_or(PumpError::ArithmeticError)?
    .checked_add(1)
    .ok_or(PumpError::ArithmeticOverflow)?;
```

5. **Sell Quote Calculation**

```
// Current implementation
(amount * virtual_sol_reserves) / (virtual_token_reserves + amount)

// Required: Checked arithmetic
let denominator = virtual_token_reserves
    .checked_add(amount)
    .ok_or(PumpError::ArithmeticOverflow)?;

amount
    .checked_mul(virtual_sol_reserves)
    .ok_or(PumpError::ArithmeticOverflow)?
    .checked_div(denominator)
    .ok_or(PumpError::DivisionByZero)?
```

Also, migration implementation has unsafe arithmetic on the following implementation.

```
let
   sol_amount = ctx.accounts.bonding_curve.real_sol_reserves - pool_migration_fee;
```

Code Location : migrate.rs#L149-L150


```
let
   sol_amount = ctx.accounts.bonding_curve.real_sol_reserves - pool_migration_fee;
```

Code Location : migrate.rs#L149-L150

Consider adding checked arithmetic to critical paths.

[L-10] Bonding can be delayed indefinitely

The bonding is completed when `real_token_reserves` reach 0.

```
if ctx.accounts.bonding_curve.real_token_reserves == 0 {
            // Bonding completed logic
        }
```

Users can only buy the amount of tokens they specify in the `buy` function. If they specify more than available, this function reverts.

```
require_gte!(
            ctx.accounts.bonding_curve.real_token_reserves,
            amount,
            PumpError::NotEnoughTokensToBuy
        );
```

Thus, we can have a situation where a user tries to buy up the entire remaining amount of tokens (say 1000) to complete the bonding but before they can do that, another user can sell 1 lamport of token. So now, after the user buy, there will still be 1 lamport of unbonded token remaining, preventing the bonding and migration.

So a malicious user can sell 1 lamport every block, and this will change the amount needed to finish bonding every block. Since the buyer needs to specify the this exact number to finish the bonding and it changes block to block, it can become very hard to actually bond the token even if there are sufficient interested funds in bonding it.

Thus, this technique can be used to grief a token creator.

Consider allowing users to buy up the entire remaining amount if their specified `amount` is higher than the available amount, or if they specify `amount` as 2**64-1 (max value for u64).

[L-11] Extend account can be called to unnecessarily waste native tokens

The `extend_account` function is open to be called by anyone. So any user can call this function to extend the account size up to `MAX_PERMITTED_DATA_INCREASE`, which is 10240 bytes. The required rent is paid by the account which is being extended itself.

```
let payment_required = Rent::get()?
        .minimum_balance(new_size)
        .saturating_sub(account.lamports());
```

Thus if the account being extended already has lamports, it won't charge the caller. The `bonding_curve` is allowed to be extended and it also contains the SOL tokens from the bonding process. Thus any user can increase its account size, which will taste some of the SOL in the bonding contract even if it is not necessary.

This can lead to some wastage of SOL, with users extending account sizes unnecessarily.

Consider making the `extend_account` function access controlled, so only the necessary account can be extended. Otherwise, set a restricted upper limit, instead of 10240 which is the maximum limit of Anchor. This can limit sol wastage.