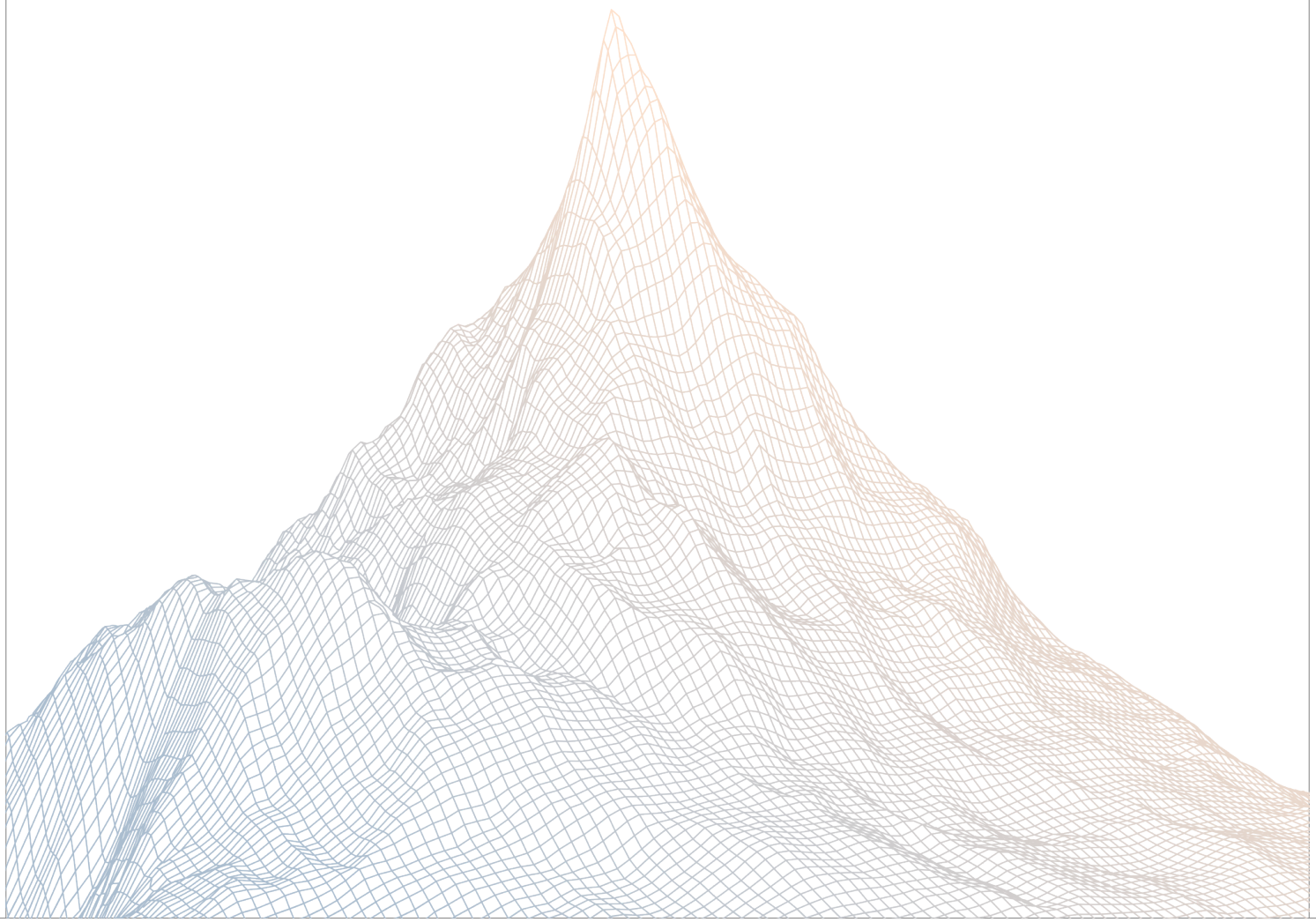
Zenith

Dripster

Smart Contract Security Assessment

VERSION 1.1



AUDIT DATES: March 12th to March 20th, 2025

AUDITED BY: J4X
Peakbolt

Contents

1 Introduction

1.1 About Zenith

Zenith is an offering by Code4rena that provides consultative audits from the very best security researchers in the space. We focus on crafting a tailored security team specifically for the needs of your codebase.

Learn more about us at https://code4rena.com/zenith.

1.2 Disclaimer

This report reflects an analysis conducted within a defined scope and time frame, based on provided materials and documentation. It does not encompass all possible vulnerabilities and should not be considered exhaustive.

The review and accompanying report are presented on an "as-is" and "as-available" basis, without any express or implied warranties.

Furthermore, this report neither endorses any specific project or team nor assures the complete security of the project.

1.3 Risk Classification

SEVERITY LEVEL	IMPACT: HIGH	IMPACT: MEDIUM	IMPACT: LOW
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

2

Executive Summary

2.1 About Dripster

Dripster.fun is a platform that enables users to design, launch, and trade "dripcoins," bringing top product concepts to life.

2.2 Scope

The engagement involved a review of the following targets:

Target	dripster-protocol
Repository	https://github.com/bloom-art/dripster-protocol/
Commit Hash	`b764613536cd55d242d9d94fb1c6dfcd26cd2607`
Files	`programs/dripster/src/* (excl. test & mock files)`

2.3 Audit Timeline

March 12, 2025	Audit start
March 20, 2025	Audit end
March 18, 2025	Report published

2.4 Issues Found

SEVERITY	COUNT
Critical Risk	0
High Risk	2
Medium Risk	6
Low Risk	11
Informational	4
Total Issues	**23**

3

Findings Summary

ID	Description	Status
H-1	buy() fails to adjust lamports_upper_bound when purchased token amount is reduced	Resolved
H-2	Missing Access Control on Pool Creation	Resolved
M-1	Users rent could be used on buy/sell	Resolved
M-2	withdraw() leaves hardcoded rent in bonding_curve	Resolved
M-3	Rent for bonding curve isn't refunded to creator	Resolved
M-4	Missing master_kill_switch check for certain instructions	Resolved
M-5	clean_up() can be DoS by transfering dust token to bonding_curve_vault	Resolved
M-6	create_pool() will fail when bonding curve token mint key is greater than native mint key	Resolved
L-1	Fees are rounded down	Resolved
L-2	Varying CU for buy/sell due to search for canonical bump	Resolved
L-3	Too low base_graduation_fee_in_lamports could lead to locked pools	Resolved
L-4	compute_buy_price_in_lamports() could divide by 0	Resolved
L-5	cleanup() will fail if signer token account has already been closed	Resolved
L-6	Sell will revert although user has enough to cover fee	Resolved
L-7	set_config() should validate base_on_hand_token_decimal_reserve	Resolved
L-8	Wrong graduation reward would lead to unintended pool initalization	Resolved
L-9	Missing check for price overflow in set_config	Resolved
L-10	Graduation will be potentially blocked if graduation fee is 0	Resolved

ID	Description	Status
L-11	Graduation will be potentially blocked if graduation reward is 0	Resolved
I-1	Missing usage of under/over-flow checks in codebase	Resolved
I-2	graduate() will fail for bonding curve that were created using token2022 program	Resolved
I-3	Incorrect error msg for bonding_curve.graduated constraint	Resolved
I-4	Unreliable underflow check	Resolved

4

Findings

4.1 High Risk

A total of 2 high risk findings were identified.

[H-1] `buy()` fails to adjust `lamports_upper_bound` when purchased token amount is reduced

SEVERITY: High	IMPACT: High
STATUS: Resolved	LIKELIHOOD: Medium

Target

- buy.rs

Description:

The instruction `buy()` allows the user to provide a `lamports_upper_bound`, which is the user's max total price (x) to pay for `y` tokens. so max token price that user is willing to pay is `x/y`.

Also, when there is insufficient tokens to fulfil the user's requested token amount, `buy()` will adjust the purchase token amount by the leftover supply in the token reserve.

Despite the adjustment to the purchased token amount, `lamports_upper_bound` remains the same.

When supply is left with `z` such that `z < y`, `x` should be adjusted down based on what the user will receive (z), and not `y`. so lamports_upper_bound (max total price) should be `x/y * z`.

That is because the token price of the remaining `z` tokens will be more expensive than token price of the requested `y`, due to the bonding curve .

This will cause users will pay more per token, even though the total price is lower.

```
pub fn buy(
    ctx: Context<BuyInfo>,
    token_decimal_amount: u64,
    lamports_upper_bound: u64,
) → Result<()> {
    require!(
```

```
        token_decimal_amount > 0,
        CustomError::InvalidTokenDecimalAmount
    );

    let maximum_token_decimal_amount =
        compute_maximum_token_decimal_amount(&ctx.accounts.bonding_curve,
    token_decimal_amount)?;

    let price_in_lamports =
        compute_buy_price_in_lamports(&ctx.accounts.bonding_curve,
    maximum_token_decimal_amount)?;

    require!(
        price_in_lamports <= lamports_upper_bound,
        CustomError::PriceAboveLamportUpperBound
    );
```

Recommendations:

Adjust the `lamports_upper_bound` when purchased token amount is capped by the remaining token supply in the reserve.

Dripster: Resolved with[PR-6]https://github.com/zenith-security/2025-03-dripster/issues/6)

Zenith: Verified. Resolved with new `adjust_upper_bound_based_on_remaining_token_reserve()`.

[H-2] Missing Access Control on Pool Creation

SEVERITY: High	IMPACT: High
STATUS: Resolved	LIKELIHOOD: High

Target

- create_pool.rs#L171

Description:

The `create_pool` instruction creates a Raydium pool for the newly launched token. Currently, this IX can be called by anyone.

```
/// Address paying to create the pool. Can be anyone
#[account(mut)]
pub creator: Signer<'info>,
```

Due to this being open to any caller, someone else could create the pool before the graduation authority and claim the LP tokens.

Recommendations:

We recommend restricting access to the `create_pool` instruction to the `graduation_authority`.

Dripster: Resolved with PR-8

Zenith: Verified.

4.2 Medium Risk

A total of 6 medium risk findings were identified.

[M-1] Users rent could be used on buy/sell

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: High

Target

- sell_helpers.rs#L33
- buy_helpers.rs#L54

Description:

On buying/selling the contract checks, if the caller has enough lamports to fund the tx.

```
pub fn ensure_buyer_has_enough_lamports(
    price_in_lamports: u64,
    transfer_fee_in_lamports: u64,
    signer: &Signer,
) -> Result<()> {
    let total_transaction_amount_in_lamports = price_in_lamports +
    transfer_fee_in_lamports;

    require!(
        signer.lamports() ≥ total_transaction_amount_in_lamports,
        CustomError::NotEnoughLamportsInBuyerAccount
    );

    Ok(())
}
```

```
pub fn ensure_seller_has_enough_lamports(
    transfer_fee_in_lamports: u64,
    signer: &Signer,
) -> Result<()> {
```

```
    require!(
        signer.lamports() ⩾ transfer_fee_in_lamports,
        CustomError::NotEnoughLamportsInSellerAccount
    );

    Ok(())
}
```

However, the functionality does not account for possible needed rent. As a result, it will also take the user's rent money, which could lead to the buyer's account getting garbage collected.

Recommendations:

We recommend checking the caller's required rent balance to ensure it can not fall below the rent exempt minimum through a call.

Dripster: Resolved with PR-9

Zenith: Verified

[M-2] `withdraw()` leaves hardcoded rent in `bonding_curve`

SEVERITY: Medium	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: High

Target

- withdraw_helpers.rs#L41

Description:

In the `withdraw` instruction most of the lamports from the bonding curve are withdrawn once trading has finished.

```
pub fn send_all_lamports_to_withdraw_authority<'info>(
    bonding_curve: &mut Account<'info, BondingCurve>,
    graduation_authority: &AccountInfo<'info>,
) → Result<()> {
    let lamport_amount = bonding_curve.get_lamports() - 10000000; // need to
    keep account opened for the rest of the graduation
    bonding_curve

    .transfer_sol_from_bonding_curve(&graduation_authority.to_account_info(),
    lamport_amount)?;

    Ok(())
}
```

Due to the code, a hardcoded value of `10000000` lamports is left in the account to cover rent. However, this is not the actual balance required for rent exception of the bonding curve. As a result, this amount might be more or less, and the account would carry more balance than needed or be garbage collected.

Additionally, this could lead to a DOS on low-volume pools where the total lamports received are less than 10000000. In that case, the calculation would underflow, resulting in a revert due to the account not having enough funds.

Recommendations:

We recommend directly calculating the needed lamports for the rent exception in code. This way, no additional funds will be left in the account, and it can't be accidentally garbage collected if its size increases.

Dripster: Resolved with PR-17

Zenith: Verified

[M-3] Rent for bonding curve isn't refunded to creator

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: High

Target

- clean_up.rs#L70

Description:

Any user can create a bonding curve using the `create` instruction. When creating the account, the user must cover the rent for the bonding curve.

```
#[account(
    init,
    space = ANCHOR_DISCRIMINATOR_SIZE + BondingCurve::INIT_SPACE,
    payer = signer,
    seeds = [
        BondingCurve::SEED_PREFIX.as_bytes(),
        token_mint.key().as_ref()
    ],
    bump
)]
pub bonding_curve: Box<Account<'info, BondingCurve>>,
```

In the `clean_up` instruction, the account will be closed again.

```
pub fn close_bonding_curve_vault<'info>(
    bonding_curve_vault: &mut InterfaceAccount<'info, TokenAccount>,
    signer: &Signer<'info>,
    token_mint: &InterfaceAccount<'info, Mint>,
    token_program: &AccountInfo<'info>,
    bonding_curve_bump: u8,
) -> Result<()> {
    anchor_spl::token_interface::close_account(CpiContext::new_with_signer(
        token_program.to_account_info(),
        anchor_spl::token_interface::CloseAccount {
            account: bonding_curve_vault.to_account_info(),
            destination: signer.to_account_info(),
```

```
            authority: bonding_curve_vault.to_account_info(),
        },
        &[&[
            BondingCurve::TOKEN_ACCOUNT_SEED_PREFIX.as_bytes(),
            token_mint.key().as_ref(),
            &[bonding_curve_bump],
        ]],
    ))?;

    Ok(())
}
```

However, this will refund the rent to the `graduation_authority`, not the actual creator, resulting in a hidden fee for the user.

Recommendations:

We recommend saving the creator of the bonding curve in the account and refunding it using the `clean_up` instruction.

Dripster: Resolved with PR-18

Zenith: Verified

[M-4] Missing `master_kill_switch` check for certain instructions

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Medium

Target

- withdraw.rs
- cleanup.rs

Description:

The `master_kill_switch` can be set to true to enable a protocol-wide pause on critical operations.

However, it is not present in `create`, `withdraw`, `clean_up` instructions. This allows these instructions to be performed even when `master_kill_switch == true`.

Recommendations:

Consider adding the `!dripster_config.master_kill_switch` constraints for `withdraw` and `clean_up` instructions.

Dripster: Resolved with PR-19

Zenith: Resolved by adding the kill switch constraints for withdraw and cleanup.

[M-5] `clean_up()` can be DoS by transfering dust token to `bonding_curve_vault`

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Medium

Target

- clean_up.rs

Description:

`clean_up()` can only be execute when `bonding_curve_vault.amount == 0`.

However, anyone can stop `clean_up()` from executing by sending dust token to the `bonding_curve_vault`.

This issue will prevent collection of the rent for the accounts.

```
pub fn clean_up(ctx: Context<CleanUpInfo>) -> Result<()> {
    require!(
        ctx.accounts.bonding_curve_vault.amount == 0,
        CustomError::TokenAccountNotEmpty
    );

    close_bonding_curve_vault(
        &mut ctx.accounts.bonding_curve_vault,
        &ctx.accounts.signer,
        &ctx.accounts.token_mint,
        &ctx.accounts.token_program,
        ctx.bumps.bonding_curve_vault,
    )?;
```

Recommendations:

This can be resolved by transferring out any remaining token in `bonding_curve_vault` to another token account at the beginning of `clean_up()`.

Dripster: Resolved with PR-29

Zenith: Resolved by removing the `require!()` check.

[M-6] `create_pool()` will fail when bonding curve token mint key is greater than native mint key

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Medium

Target

- create_pool.rs

Description:

In `create_pool()` there is a constraint `token_0_mint.key() < token_1_mint.key()`, which assumes that the native mint id `So11111111111111111111111111111111111111112` is always token_0 and the `bonding_curve` token will be token_1.

However, that is not always true as its possible that the `bonding_curve` token can be token_0, and have a mint key that is smaller than the native mint key.

For example, the JUL-SOL pool on raydium has JUL as token_0 and WSOL as token_1.

Recommendations:

There are two possible solutions.

1. In `create()` check that the token mint generated is smaller than native mint id, but that will possibly cause certain `create()` to fail.
2. In `create_pool()`, set token_0 and token_1 accordingly to respect the constraint `token_0_mint.key() < token_1_mint.key()`.

Dripster: Resolved with PR-28

Zenith: Resolved by handling the correct order of token mints in `create_pool()`.

4.3 Low Risk

A total of 11 low risk findings were identified.

[L-1] Fees are rounded down

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: High

Target

- graduate_helpers.rs#L11

Description:

The `compute_transfer_fee` function is used to calculate the fees.

```
pub fn compute_transfer_fee(
    dripster_config: &Account<DripsterConfiguration>,
    total_transaction_amount_in_lamports: u64,
) -> Result<u64> {
    let transfer_fee_in_lamports = total_transaction_amount_in_lamports
        * dripster_config.transfer_fees_in_hundredths_of_percent
        / 10_000;

    let minimum_lamport_fee = 1_000_000;
    Ok(transfer_fee_in_lamports.max(minimum_lamport_fee))
}
```

Due to the division, the fees are rounded down, allowing a user to pay less than intended.

Recommendations:

We recommend rounding up on fee calculation so as not to allow for a bypass.

Dripster: Resolved with PR-10

Zenith: Verified

[L-2] Varying CU for buy/sell due to search for canonical bump

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- create.rs
- buy.rs
- sell.rs

Description:

The buy/sell instructions uses `bump` instead of a stored bump in the account constraints for `bonding_curve` and `bonding_curve_vault`. This causes it to search for the canonical bump on each transaction.

If the bumps of the `bonding_curve` and `bonding_curve_vault` are 'far away', it could require a high amount of CU to find the canonical bump as described here.

That could cause the tx to exceed the max CU limit set as buy/sell likely will use more CU than the `create` instruction. When that happens, it will require re-executing the tx with a higher CU limit.

```
#[account(
    mut,
    constraint = bonding_curve.funded @ CustomError::NotFunded,
    constraint = !bonding_curve.end_reached @ CustomError::EndReached,
    constraint = !bonding_curve.drained @ CustomError::AlreadyDrained,
    constraint = !bonding_curve.graduated @ CustomError::AlreadyGraduated,
    constraint = bonding_curve.virtual_token_decimal_reserve
    > bonding_curve.virtual_token_decimal_reserve_stop_point @
    CustomError::EndReachedBasedOnReserve,
    constraint = bonding_curve.on_hand_token_decimal_reserve > 0 @
    CustomError::NotEnoughTokenInVault,
    seeds = [
        BondingCurve::SEED_PREFIX.as_bytes(),
        token_mint.key().as_ref()
    ],
    bump
)]
```

```
pub bonding_curve: Box<Account<'info, BondingCurve>>,

#[account(
    mut,
    token::mint = token_mint,
    token::authority = bonding_curve_vault,
    token::token_program = token_program,
    seeds = [
        BondingCurve::TOKEN_ACCOUNT_SEED_PREFIX.as_bytes(),
        token_mint.key().as_ref()
    ],
    bump
)]
pub bonding_curve_vault: Box<InterfaceAccount<'info, TokenAccount>>,
```

Recommendations:

In create instruction, store the `ctx.bumps.bonding_curve` in `bonding_curve.bump` and load it for buy/sell instructions.

Dripster: Resolved with PR-11

Zenith: Resolved by storing bumps for bonding curve and its vault, and then loading the bump for instructions that require the bonding curve and vault accounts.

[L-3] Too low `base_graduation_fee_in_lamports` could lead to locked pools

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- graduate_helpers.rs#L40

Description:

On graduation of a bonding curve the following calculation takes place:

```
transfer_graduation_fee_to_fee_recipient(
    &ctx.accounts.signer,
    &ctx.accounts.fee_recipient,
    ctx.accounts.bonding_curve.graduation_fee_in_lamports - 200_000_000, // Keeping 0.2 SOL for Raydium fees
    &ctx.accounts.system_program,
)?;
```

When a bonding curve gets funded, the `graduation_fee_in_lamports` gets set to `dripster_config.base_graduation_fee_in_lamports`.

```
self.graduation_fee_in_lamports = dripster_config.base_graduation_fee_in_lamports;
```

However, the `dripster_config.base_graduation_fee_in_lamports` is only checked to be bigger than 1000 in `set_config()`

```
require!(
    base_graduation_fee_in_lamports > 1000_u64,
    CustomError::InvalidGraduationFee
);
```

So if the only constraint is that `graduation_fee_in_lamports > 1000` but the calculation at graduation calculates `graduation_fee_in_lamports - 200_000_000`, which does not have an underflow check. As a result, the calculation will underflow, and the contract will try to

transfer an immensely huge amount of lamports, which the account won't have, and revert as a result, completely blocking graduation.

Recommendations:

We recommend checking that `base_graduation_fee_in_lamports > 200_000_000`

```
require!(
    base_graduation_fee_in_lamports > 200000000_u64,
    CustomError::InvalidGraduationFee
);
```

Dripster: Resolved with PR-12

Zenith: Verified

[L-4] `compute_buy_price_in_lamports()` could divide by 0

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- buy_helpers.rs#L27

Description:

The `compute_buy_price_in_lamports()` function calculates the price on the bonding curve.

```
let numerator =
    (bonding_curve.virtual_lamport_reserve as u128) *
    (maximum_token_decimal_amount as u128);
let denominator = (bonding_curve.virtual_token_decimal_reserve as u128)
    - (maximum_token_decimal_amount as u128);

let price_in_lamports_u128 = if numerator % denominator == 0 {
    numerator / denominator
} else {
    numerator / denominator + 1
};
```

The denominator here is `virtual_token_decimal_reserve - maximum_token_decimal_amount`. However the check done before in `compute_maximum_token_decimal_amount()` allows for `maximum_token_decimal_amount == virtual_token_decimal_reserve`.

```
require!(
    maximum_token_decimal_amount ⩽
    bonding_curve.virtual_token_decimal_reserve,
    CustomError::NotEnoughTokenInVault
);
```

As a result, the denominator could become 0, leading to a division by zero and a revert.

Recommendations:

We recommend adapting the check to only allow for `maximum_token_decimal_amount < virtual_token_decimal_reserve`

Dripster: Resolved with PR-14

Zenith: Verified

[L-5] `cleanup()` will fail if signer token account has already been closed

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- cleanup.rs

Description:

`cleanup()` will close the bonding curve's PDAs (`bonding_curve_vault`, `bonding_curve`) and ATA (`signer_token_account`).

However, it is possible for the `signer_token_account` ATA to be closed before `cleanup()` if the `graduation_authority` performed the close directly using the Associated Token Program.

That will cause `cleanup()` to fail as it will then encounter an error when closing the `signer_token_account` ATA in `close_signer_token_account_if_empty()`.

```
pub fn close_signer_token_account_if_empty<'info>(
    signer_token_account: &mut InterfaceAccount<'info, TokenAccount>,
    signer: &Signer<'info>,
    token_program: &AccountInfo<'info>,
) -> Result<()> {
    if signer_token_account.amount == 0 {
        anchor_spl::token_interface::close_account(CpiContext::new(
            token_program.to_account_info(),
            anchor_spl::token_interface::CloseAccount {
                account: signer_token_account.to_account_info(),
                destination: signer.to_account_info(),
                authority: signer.to_account_info(),
            },
        ))?;
    }
    Ok(())
}
```

Recommendations:

Consider catching the error when `close_account` fails in `close_signer_token_account_if_empty()`.

Dripster: Resolved with PR-15

Zenith: Resolved by catching and logging the error of `close_signer_token_account_if_empty()` so that cleanup will proceeds as usual.

[L-6] Sell will revert although user has enough to cover fee

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- sell.rs#L111

Description:

In the `sell()` function, users can sell their tokens to regain lamports. On this sale, a fee will be charged. The current process is:

1. Calculate the total lamports user will get for sale
2. Calculate the fee
3. Check if the user has enough lamports to cover the fee (percentage of his sale)
4. Charge the user the fee
5. Swap tokens and send lamports to user

As this will check the user's balance being >= than the fee before he has received the lamports for his sale, it might revert while a user could cover the fee.

Recommendations:

We recommend checking if `transfer_fee_in_lamports ⩽ user.lamports + price_in_lamports` and transferring the fee after the user has received the rewards from his sale.

Dripster: Resolved with PR-30

Zenith: Verified

[L-7] `set_config()` should validate `base_on_hand_token_decimal_reserve`

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Medium

Target

- set_config.rs#L38-L69

Description:

`set_config()` allows changes to the base configuration for creation of new bonding curve.

However, it does not validate `base_on_hand_token_decimal_reserve` against `base_virtual_token_decimal_reserve` and `base_virtual_token_decimal_reserve_stop_point`.

If `base_on_hand_token_decimal_reserve < base_virtual_token_decimal_reserve - base_virtual_token_decimal_reserve_stop_point`, it will prevent the bonding curve from completing and graduating as the tradable amount of tokens is less than what will be minted to the bonding curve.

And `base_on_hand_token_decimal_reserve` should also consider the amount of tokens left after completion, to be graduated to raydium.

```
pub fn set_config(
    ctx: Context<SetConfigInfo>,
    base_graduation_fee_in_lamports: u64,
    base_on_hand_token_decimal_reserve: u64,
    base_token_decimal_graduation_reward: u64,
    base_virtual_lamport_reserve: u64,
    base_virtual_token_decimal_reserve: u64,
    base_virtual_token_decimal_reserve_stop_point: u64,
    creation_fee_in_lamports: u64,
    master_kill_switch: bool,
    transfer_fees_in_hundredths_of_percent: u64,
) -> Result<()> {
    require!(
        transfer_fees_in_hundredths_of_percent <= 10000_u64,
        CustomError::InvalidFee
    );
```

```
    require!(
        base_virtual_lamport_reserve > 1000_u64,
        CustomError::InvalidBaseVirtualLamportReserve
    );
    require!(
        base_graduation_fee_in_lamports > 1000_u64,
        CustomError::InvalidGraduationFee
    );
    require!(
        base_virtual_token_decimal_reserve >
base_virtual_token_decimal_reserve_stop_point,
        CustomError::InvalidBaseVirtualTokenDecimalReserveStopPoint
    );
    require!(
        base_token_decimal_graduation_reward ≤
base_virtual_token_decimal_reserve_stop_point,
        CustomError::InvalidBaseTokenDecimalGraduationReward
    );
```

Recommendations:

Ensure that `base_on_hand_token_decimal_reserve` is much greater than `base_virtual_token_decimal_reserve - base_virtual_token_decimal_reserve_stop_point`, so that there is sufficient token left for graduation.

Dripster: Fixed in PR-23

Zenith: Verified

[L-8] Wrong graduation reward would lead to unintended pool initalization

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- set_config.rs#L38

Description:

After a bondign curve has reached the `base_virtual_token_decimal_reserve_stop_point` the `base_token_decimal_graduation_reward` is sent to the creator and the rest of the tokens are used for the initial liquidity of the pool. When these parameters get changed the current constraint is the following:

```
require!(
    base_token_decimal_graduation_reward ⩽
    base_virtual_token_decimal_reserve_stop_point,
    CustomError::InvalidBaseTokenDecimalGraduationReward
);
```

This allows for `base_token_decimal_graduation_reward` = `base_virtual_token_decimal_reserve_stop_point`. If this is the case, no tokens would be left for the liquidity pool, which would only be initialized with WSOL.

Recommendations:

We recommend checking that `graduation_reward < stop_point` in the `set_config` so that at least one token is put into the pool.

```
require!(
    base_token_decimal_graduation_reward <
    base_virtual_token_decimal_reserve_stop_point,
    CustomError::InvalidBaseTokenDecimalGraduationReward
);
```

Dripster: Resolved with PR-23

Zenith: Verified

[L-9] Missing check for price overflow in `set_config`

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Medium

Target

- set_config.rs#L38

Description:

The `compute_buy_price_in_lamports()` function is used to calculate the price that a user needs to pay for his tokens.

```
pub fn compute_buy_price_in_lamports(
  bonding_curve: &Account<BondingCurve>,
  maximum_token_decimal_amount: u64,
) -> Result<u64> {
  let numerator =
      (bonding_curve.virtual_lamport_reserve as u128) *
    (maximum_token_decimal_amount as u128);
  let denominator = (bonding_curve.virtual_token_decimal_reserve as u128)
      - (maximum_token_decimal_amount as u128);

  let price_in_lamports_u128 = if numerator % denominator == 0 {
      numerator / denominator
  } else {
      numerator / denominator + 1
  };

  require!(
      price_in_lamports_u128 ≤ u64::MAX as u128,
      CustomError::PriceExceedsU64Max
  );

  let price_in_lamports = price_in_lamports_u128 as u64;

  require!(price_in_lamports > 0, CustomError::InvalidLamportPrice);

  Ok(price_in_lamports)
}
```

The function includes a check that the price does not go above `u64::MAX`. However when setting the new configuration parameters in `set_config` it is never checked that the price can not reach that max, which would result in the bonding curve never being finalized.

Recommendations:

We recommend checking for `(virtual_lamport_reserve * (virtual_token_decimal_reserve - virtual_token_decimal_reserve_stop_point)) / (virtual_token_decimal_reserve_stop_point) > u64::MAX` in `set_config()`

Dripster: Resolved with PR-24

Zenith: Verified

[L-10] Graduation will be potentially blocked if graduation fee is 0

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- graduate_helpers.rs#L23

Description:

When the curve is graduated, a potential graduation fee will be paid to the fee recipient. To ensure that the `graduation_authority` has enough funds to cover the fee, the `validate_lamports_requirements()` function is called.

```
pub fn validate_lamports_requirements(
    bonding_curve: &BondingCurve,
    lamports_balance: u64,
) -> Result<()> {
    require!(lamports_balance > 0, CustomError::NoLamportsInBondingCurve);
    require!(
        lamports_balance >= bonding_curve.graduation_fee_in_lamports,
        CustomError::NotEnoughLamportsInBondingCurve
    );

    Ok(())
}
```

In some cases, the `graduation_fee_in_lamports` could be set to zero. However, the check before still requires that the graduation authority has lamports in its balance, so the call will not be possible. This will result in the graduation reverting without a reason.

Recommendations:

We recommend removing the check for `lamports_balance > 0`.

Dripster: Resolved with PR-25

Zenith: Verified

[L-11] Graduation will be potentially blocked if graduation reward is 0

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- graduate_helpers.rs#L11

Description:

When the curve is graduated, a potential graduation rewards will be paid to the creator. To ensure that the `graduation_authority` has enough funds to cover the fee, the `validate_graduation_reward_requirements()` function is called.

```
pub fn validate_graduation_reward_requirements(
    bonding_curve: &BondingCurve,
    token_decimal_balance: u64,
) → Result<()> {
    require!(
        token_decimal_balance > 0,
        CustomError::NoTokensInBondingCurve
    );
    require!(
        token_decimal_balance ⩾
    bonding_curve.token_decimal_graduation_reward,
        CustomError::NotEnoughTokensInBondingCurve
    );

    Ok(())
}
```

In some cases, the `token_decimal_graduation_reward` could be set to zero. However, the check before still requires that the graduation authority has tokens in its balance, so the call will not be possible. This will result in the graduation reverting without a reason.

Recommendations:

We recommend removing the check for `token_decimal_balance > 0`.

Dripster: Resolved: PR-26

Zenith: Verified

4.4 Informational

A total of 4 informational findings were identified.

[I-1] Missing usage of under/over-flow checks in codebase

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- All files

Description:

Throughout the codebase, multiple additions and subtractions occur. However, no under/over-flow checks are implemented for these. This could lead to potential unintended behavior.

Recommendations:

We recommend adding under/over-flow checks by using `checked_sub` or `checked_add`

Dripster: Resolved: PR-27

Zenith: Verified

[I-2] `graduate()` will fail for bonding curve that were created using token2022 program

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- graduate.rs
- create.rs

Description:

In `create()`, it allows the user to specify either the original Token Program or the new Token2022 Program.

However, in `graduate()`, it only supports Token Program due to `signer_wsol_account` using the original Token Program, based on the native mint address `So11111111111111111111111111111111111111112`.

This will attempt to load the `signer_token_account` using the original Token Program as well, as it is using the same `token_program` as `signer_wsol_account`.

```
pub struct GraduateInfo<'info> {
      ...
    #[account(
        mut,
        associated_token::mint = wsol_mint,
        associated_token::authority = signer,
        associated_token::token_program = token_program,
    )]
    pub signer_wsol_account: Box<InterfaceAccount<'info, TokenAccount>>,

    #[account(
        mut,
        associated_token::mint = token_mint,
        associated_token::authority = signer,
        associated_token::token_program = token_program,
    )]
    pub signer_token_account: Box<InterfaceAccount<'info, TokenAccount>>,
```

```
    pub associated_token_program: Program<'info, AssociatedToken>,
    pub system_program: Program<'info, System>,
    pub token_program: Interface<'info, TokenInterface>,
}
```

Recommendations:

Consider updating `create()` to only allow creation of bonding token using the original Token Program.

Dripster: Fixed in PR-20

Zenith: Verified

[I-3] Incorrect error msg for `bonding_curve.graduated` constraint

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- create_pool.rs

Description:

In both `CreatePoolInfo` and `GraduateInfo` context, the error msg is incorrectly set for `CustomError::AlreadyGraduated`, when the constraint `bonding_curve.graduated` is not met.

Recommendations:

Fix the error message as follows,

```
#[account(
    mut,

constraint = bonding_curve.graduated @ CustomError::AlreadyGraduated,
   constraint = bonding_curve.graduated @ CustomError::NotYetGraduated,
    seeds = [
        BondingCurve::SEED_PREFIX.as_bytes(),
        token_1_mint.key().as_ref()
    ],
    bump
)]
pub bonding_curve: Box<Account<'info, BondingCurve>>,
```

Dripster: Fixed in PR-21.

Zenith: Verified. Resolved by changing to `NotGraduated` error in `create_pool.rs`.

[I-4] Unreliable underflow check

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- buy_helpers.rs#L16

Description:

In the `compute_maximum_token_decimal_amount()` calculation calculates the tokens left to buy.

```
let maximum_left_to_buy = (bonding_curve.virtual_token_decimal_reserve
    - bonding_curve.virtual_token_decimal_reserve_stop_point)
    .max(0);
```

Here, a `.max(0)` protects against underflows. However, this will never be triggered as the calculation subtracts two unsigned integers, which in case of underflow will also result in a number greater than zero.

Recommendations:

We recommend using `checked_sub()` instead.

Dripster: Resolved with PR-22

Zenith: Verified