Perena Prime

Security Review

Cantina Managed review by:

FrankCastle, Security Researcher

0xHuy0512, Associate Security Researcher
Jdiggidy, Associate Security Researcher

April 25, 2025

Contents

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

Cantina Managed provides a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While Cantina Managed endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that the Cantina Managed security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	Description
Critical	*Must* fix as soon as possible (if already deployed).
High	Leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
Medium	Global losses <10% or losses to only a subset of users, but still unacceptable.
Low	Losses will be annoying but bearable. Applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

Perena is building the foundational infrastructure for the future of digital money, solving critical challenges in the $170B+ stablecoin market.

From Mar 15th to Mar 22nd the Cantina team conducted a review of perena-prime on commit hash 43b08c2c. The team identified a total of **20** issues:

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	4	4	0
High Risk	3	3	0
Medium Risk	4	3	1
Low Risk	3	3	0
Gas Optimizations	0	0	0
Informational	6	6	0
Total	**20**	**19**	**1**

The Cantina Managed team reviewed Perena's `perena-prime` holistically on commit hash 662622ba and concluded that all the issues were addressed and no new vulnerabilities were identified.

3 Findings

3.1 Critical Risk

3.1.1 Incorrect Decimal conversion Leads to Significant Fund Loss

Severity: Critical Risk

Context: *(No context files were provided by the reviewer)*

Description: In the `usd_star_mint` and `usd_star_burn` functions, the decimal scaling logic is incorrect, leading to a significant loss of funds for both users and the protocol. The issue arises in the following snippet:

```
if asset_info.decimals > usd_star_decimal {
    amount = amount
        .checked_mul(
            10u128
                .checked_pow((asset_info.decimals - usd_star_decimal) as u32)
                .ok_or(ProgramError::ArithmeticOverflow)?,
        )
        .ok_or(ProgramError::ArithmeticOverflow)?;
} else if asset_info.decimals < usd_star_decimal {
    amount = amount
        .checked_div(
            10u128
                .checked_pow((usd_star_decimal - asset_info.decimals) as u32)
                .ok_or(ProgramError::ArithmeticOverflow)?,
        )
        .ok_or(ProgramError::ArithmeticOverflow)?;
}
```

Here, the `amount` value is initially scaled according to the asset's decimals. However, the conversion logic mistakenly applies the scaling operation in the wrong direction. This results in incorrect token amounts being minted, causing substantial discrepancies in value.

Impact:

- Users may receive significantly fewer tokens than expected.
- The protocol may suffer a substantial loss of funds due to incorrect scaling.
- This could lead to arbitrage exploits and financial imbalances in the system.

Recommendation: To ensure correct decimal scaling, swap the multiplication and division operations in the conditional statements:

```
  if asset_info.decimals > usd_star_decimal {
      amount = amount
-         .checked_mul(
+         .checked_div(
              10u128
                  .checked_pow((asset_info.decimals - usd_star_decimal) as u32)
                  .ok_or(ProgramError::ArithmeticOverflow)?,
          )
          .ok_or(ProgramError::ArithmeticOverflow)?;
  } else if asset_info.decimals < usd_star_decimal {
      amount = amount
-         .checked_div(
+         .checked_mul(
            10u128
                  .checked_pow((usd_star_decimal - asset_info.decimals) as u32)
                  .ok_or(ProgramError::ArithmeticOverflow)?,
          )
          .ok_or(ProgramError::ArithmeticOverflow)?;
  }
```

By making this correction, the amount will be properly scaled to match the `usd_star_decimal`.

Perena: Fixed in PR 38.

Cantina Managed: Verified.

3.1.2 Token Decimals Are Not Handled in `usd_prime_burn` and `usd_prime_mint`, Leading to Huge Loss of Funds for Both Protocol and Users

Severity: Critical Risk

Context: *(No context files were provided by the reviewer)*

Description: In the functions `usd_prime_burn` and `usd_prime_mint`, the decimals of the asset and `prime_usd` are not properly handled, which can result in a huge loss of funds for both the protocol and users. Example: Issue in `usd_prime_burn`: The `amount` is in `usd_prime` decimals, and it gets divided by the price (which is a scalar). This results in an amount in `prime_decimal` instead of the correct asset decimals. Problematic Code:

```
let mut amount = instruction_data
    .prime_amount
    .checked_mul(FEE_RATE_SCALE - bank.prime_fee_rate as u64)
    .ok_or(ProgramError::ArithmeticOverflow)?
    .checked_div(FEE_RATE_SCALE)
    .ok_or(ProgramError::ArithmeticOverflow)?;

amount = amount
    .checked_div(asset_price.price as u64)
    .ok_or(ProgramError::ArithmeticOverflow)?;

amount = amount
    .checked_mul(
        10u64
            .checked_pow(asset_price.exponent.unsigned_abs())
            .ok_or(ProgramError::ArithmeticOverflow)?,
    )
    .ok_or(ProgramError::ArithmeticOverflow)?;
```

Example Scenario:

- USD Prime Decimals = 6.
- Asset Decimals = 9.
- Fees = 0 (for simplicity).
- Asset Price = `5 * 10^5`.
- Exponent = -5.
- USD Prime Amount = `5 * 10^6`.

Calculation:

```
amount = (5 * 10^6) / 5 = 10^6
```

The asset amount to be transferred is 10^6 instead of 10^9. Since the asset price is 5 USD and we are burning 5 units of USD Prime, the user should receive 10^9, but instead, they receive 10^6, causing a massive loss of funds for users. Another Scenario Where the Protocol Loses Funds: If `usd_prime` decimals are greater than the asset decimals, the protocol will suffer from this vulnerability and lose funds. To correct this, the amount should be multiplied by `10^(asset_decimals - usd_prime_decimals)` to get the correct value.

Recommendation: Convert the decimals of the amount in the functions `usd_prime_burn` and `usd_prime_mint` as shown here:

```
// usd_prime_burn
if asset_info.decimals > usd_prime_decimal {
    amount = amount
        .checked_mul(
            10u128
                .checked_pow((asset_info.decimals - usd_prime_decimal) as u32)
                .ok_or(ProgramError::ArithmeticOverflow)?,
        )
        .ok_or(ProgramError::ArithmeticOverflow)?;
} else if asset_info.decimals < usd_prime_decimal {
    amount = amount
        .checked_div(
            10u128
                .checked_pow((usd_prime_decimal - asset_info.decimals) as u32)
                .ok_or(ProgramError::ArithmeticOverflow)?,
        )
        .ok_or(ProgramError::ArithmeticOverflow)?;
}
```

Applying this fix ensures precision is maintained and prevents fund losses for both users and the protocol.

Perena: Fixed in PR 38.

Cantina Managed: Verified.

3.1.3 Swap Function Vulnerable to Self-Transfer Attack, Leading to Pool Drainage

Severity: Critical Risk

Context: *(No context files were provided by the reviewer)*

Description: In the `swap_out()` function, there is a critical vulnerability in the transfer operation for the input asset. The function does not properly validate the input accounts, which allows an attacker to manipulate the transaction to perform a self-transfer while still receiving tokens from the reserve.

The root cause is the `swap_out()` function accepts user-provided accounts without validating that:

1. The `user_info` account is actually the user's wallet and not the `bank_info` account.
2. The `input_user_ata` account is actually the user's associated token account and not the bank's reserve account (`input_reserve_info`).

This allows an attacker to construct a transaction where they provide the `bank_info` as the `user_info` and the reserve account as the `input_user_ata`. When this happens, the first transfer effectively moves tokens from the reserve back to itself (a no-op) instead of moving tokens from the attacker to the reserve as intended, while the second transfer still sends the output tokens to the attacker.

Since no actual input tokens are transferred from the attacker, they can execute this attack repeatedly to drain all tokens from the output reserve without providing any input tokens in return.

Recommendation: Implement proper account validation before performing transfers. This should include:

1. Verify that `user_info` is the signer of the transaction.
2. Verify `input_user_ata` and `output_user_ata` are valid ATAs owned by `user_info` (This should apply to all mint, burn, and swap functions).
3. Add checks that input and output accounts are distinct to prevent self-transfers (This should apply to all mint, burn, and swap functions).

Perena: Fixed in PR 43.

Cantina Managed: Verified.

3.1.4 Missing Validation on USD-Star Mint Address Allows Unlimited USD-Prime Minting

Severity: Critical Risk

Context: *(No context files were provided by the reviewer)*

Description: An exploitable lack of validation exists in the `unstake` processor function in `program/src/processors/unstake.rs`. This vulnerability allows attackers to bypass intended controls and mint unlimited USD-Prime tokens.

The core issue is that when processing an unstake request, the code fails to validate that the `usd_star_mint_info` provided by the user is the legitimate USD-Star token mint. This oversight creates a critical vulnerability. The attack path is straightforward:

- An attacker creates a worthless "trash" token they fully control.
- They pass this token mint as `usd_star_mint_info` when calling the `unstake()` function.
- The function proceeds to burn their worthless tokens and mint legitimate USD-Prime tokens in return.
- The amount of USD-Prime minted is based on calculations involving the supposed USD-Star price.
- Attacker uses the newly minted USD_Prime to swap to other tokens, effectively draining all the vaults.

This vulnerability effectively creates an unlimited minting vulnerability for USD-Prime tokens, allowing attackers to drain the protocol by exchanging worthless tokens for valuable ones.

Recommendation: Add proper validation for the `usd_star_mint_info` parameter similar to how `usd_prime_mint_info` is validated in both `stake()` and `unstake()` functions.

Perena: Fixed in PR 48.

Cantina Managed: Verified.

3.2 High Risk

3.2.1 Integer Overflow in `usd_prime_mint` Calculation will lead to making it impossible to mint USD tokens under normal conditions.

Severity: High Risk

Context: usd_prime_mint.rs#L69

Description: The `usd_prime_mint` function multiplies `asset_amount` by `asset_price.price` and then divides by `10.pow(asset_price.exponent)`. However, the multiplication is performed in a `u64` variable, which is too small to hold large intermediate results before division, leading to an overflow error in normal minting scenarios.

In the function `usd_prime_mint`, the calculation of the USD amount follows this formula:.

$$\text{amount} = \frac{\text{asset\_amount} \times \text{price}}{10^{\text{price.exponent}}}$$

To minimize precision loss, multiplication is performed before division. However, since `u64` can only store values up to 1.8×10^{19}, large token amounts and prices can cause an overflow before division occurs.

Proof of Concept: The vulnerable code snippet:

```
let mut amount: u64 = instruction_data
    .asset_amount
    .checked_mul(asset_price.price as u64)
    .ok_or(ProgramError::ArithmeticOverflow)?;

amount = amount
    .checked_div(
        10u64
            .checked_pow(asset_price.exponent.unsigned_abs())
            .ok_or(ProgramError::ArithmeticOverflow)?,
    )
    .ok_or(ProgramError::ArithmeticOverflow)?;
```

Vulnerability Scenario:

- `u64` can hold up to 1.8×10^{19}.
- A normal token supply might be `10^9`, with 9 decimal places.

- Token amounts would therefore have up to 18 decimal places.
- Price feeds often have 6-8 decimal places (`~700 feeds use 5 decimals, ~600 feeds use 8 decimals`).
- The multiplication step results in 10^{26}, far exceeding `u64`'s limit, causing an overflow.

Example Overflow Case:

- `amount` $= 10_000 \times 10^9$.
- `price`$= 100 \times 10^8$.
- Multiplication result: 10^{23}, which exceeds `u64`'s capacity.

Impact: This issue will cause the function to revert due to an arithmetic overflow, making it impossible to mint USD tokens under normal conditions.

Recommendation: To prevent overflow, the multiplication should be performed using `u128`, which can store values up to 3.4×10^{38} before downcasting to `u64`. Here is the corrected code:

```
let amount: u128 = (instruction_data.asset_amount as u128)
    .checked_mul(asset_price.price as u128)
    .ok_or(ProgramError::ArithmeticOverflow)?;

let divisor: u128 = 10u128
    .checked_pow(asset_price.exponent.unsigned_abs() as u32)
    .ok_or(ProgramError::ArithmeticOverflow)?;

let amount = amount
    .checked_div(divisor)
    .ok_or(ProgramError::ArithmeticOverflow)?;

// Safely downcast to u64 if it fits, otherwise return an overflow error
let amount: u64 = amount
    .try_into()
    .map_err(|_| ProgramError::ArithmeticOverflow)?;
```

Fix Summary:

- Use `u128` for multiplication to prevent intermediate overflows.
- Perform division using `u128` before converting the result to `u64`.
- Safely downcast to `u64` only if the final value fits within `u64`'s range.

This ensures that large token amounts and high-precision price feeds do not cause arithmetic overflows.

Perena: Fixed in commit 9baf3dff.

Cantina Managed: Verified.

3.2.2 Division Before Multiplication Issue Will Lead to Precision Loss and Rounding Down to Zero, Causing Loss of Funds to Users

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Description: In the function `usd_prime_burn`, we get the amount of asset to be transferred to the user. This amount is calculated by the following code:

```
amount = amount
    .checked_div(asset_price.price as u64)
    .ok_or(ProgramError::ArithmeticOverflow)?;

amount = amount
    .checked_mul(
        10u64
            .checked_pow(asset_price.exponent.unsigned_abs())
            .ok_or(ProgramError::ArithmeticOverflow)?,
    )
    .ok_or(ProgramError::ArithmeticOverflow)?;
```

The amount is calculated by dividing the amount by the price first and then multiplying by `10.pow(exponent)`. Since the majority of prices have 6 or 8 decimals, and the USD value has 6 decimals (such as USDC), this will lead to rounding down to zero, which can cause a huge loss of funds for users.

Example Scenario:

- Price = `10.pow(8)`.
- Prime amount = `99 * 10.pow(6)`.

This calculation results in:

$$\text{amount} = (99 \times 10^6)/(10^8) = 0$$

This will round down to zero, causing the `prime_amount` to be burned without transferring the asset to the user:

```
transfer(
    token_program_info,
    token_2022_program_info,
    asset_mint_info,
    bank_info,
    bank_asset_info,
    user_asset_info,
    amount,
    asset_info.decimals,
    bank_bump,
    program_id,
)?;

// burn
burn(
    signer_info,
    usd_prime_mint_info,
    user_prime_info,
    bank_bump,
    instruction_data.prime_amount,
    usd_prime_decimal,
)?;
```

Recommendation: Perform multiplication before division to prevent precision loss:

```
amount = amount
    .checked_mul(
        10u64
            .checked_pow(asset_price.exponent.unsigned_abs())
            .ok_or(ProgramError::ArithmeticOverflow)?,
    )
    .ok_or(ProgramError::ArithmeticOverflow)?;

amount = amount
    .checked_div(asset_price.price as u64)
    .ok_or(ProgramError::ArithmeticOverflow)?;
```

This ensures that the calculation maintains precision and prevents rounding errors that could lead to fund loss.

Perena: Fixed in PR 40.

Cantina Managed: Verified.

3.2.3 Arithmetic Overflow Risk in Token Supply Calculation

Severity: High Risk

Context: helpers/mod.rs#L49-L57

Description: In the `get_mint_supply()` function of `src/helpers/mod.rs`, there is a significant risk of arithmetic overflow when calculating token supply with high values. The function attempts to normalize token supply to a standardized decimal representation (using `DENOMINATOR = 1_000_000`). The issue occurs in this calculation:

- helpers/mod.rs#L49-L57:

```
supply = supply
    .checked_mul(DENOMINATOR)
    .ok_or(ProgramError::ArithmeticOverflow)?
    .checked_div(
        10u64
            .checked_pow(decimals as u32)
            .ok_or(ProgramError::ArithmeticOverflow)?,
    )
    .ok_or(ProgramError::ArithmeticOverflow)?;
```

For tokens with 6 decimals (which appears to be the case for both USD Star and USD Prime), this calculation becomes problematic when the supply reaches approximately 19 million tokens. At this point:

- The raw supply would be $19\text{M} \times 10^6 = 19 \times 10^{12}$.
- When multiplied by `DENOMINATOR` (`1_000_000`), this exceeds the maximum value for `u64`.

This is particularly concerning because:

1. The current supply of USD Star is reported to be around 14 million (14×10^{12} raw value).
2. Growing to 19 million is reasonably likely in the near future.

The impact of this overflow would be severe - it would cause a denial of service in critical token operations:

- `usd_prime_mint()` and `usd_prime_burn()` would fail when USD Prime reaches the 19 Million supply.
- `usd_star_mint()` and `usd_star_burn()` would fail when USD Star reaches the 19 Million supply.

Recommendation: Since the function's primary purpose appears to be providing token decimals rather than normalized supply calculations, the simplest solution is to remove the supply normalization logic. If normalized supply is actually needed by callers, consider using a larger integer type (e.g., `u128`) for the calculation to avoid overflow.

Perena: `DENOMINATOR` is removed in commit e4ee6af9.

Cantina Managed: Verified.

3.3 Medium Risk

3.3.1 Unfiltered Support for Token-2022 Extensions such as fees-on-transfer Break Protocol Logic which leads to loss of funds

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Summary: Allowing arbitrary Token-2022 extensions can disrupt protocol logic, particularly in minting and burning functions. Some extensions, such as fees-on-transfer, deduct a fee when transferring tokens, leading to discrepancies between expected and actual received amounts, which can cause fund loss.

Description: The protocol assumes that the amount specified by the user in minting and burning operations matches the actual amount received. However, Token-2022 extensions can introduce unexpected behavior. Specifically, fees-on-transfer extensions deduct a portion of the transferred amount as a fee, meaning the protocol will receive less than intended.

This discrepancy leads to:

- Loss of protocol funds since the system operates on incorrect balance assumptions.
- Broken protocol logic where minting and burning functions no longer behave as expected.

Proof of Concept: Scenario demonstrating fund loss:

1. A user initiates a minting operation with `100` tokens.
2. The protocol assumes it will receive `100` tokens.
3. If the token has a fees-on-transfer extension (e.g., 2% fee), only `98` tokens reach the protocol.
4. The discrepancy causes incorrect accounting, leading to fund loss or unexpected failures.

Impact:

- Loss of protocol funds due to inaccurate balance calculations.
- Minting/burning logic failure if assumptions about received amounts are incorrect.

Recommendation: Restrict the supported Token-2022 extensions to only those compatible with the protocol. Implement a filter function to validate mints and ensure only intended extensions are allowed:

```
pub fn is_supported_mint(mint_account: &InterfaceAccount<Mint>) -> bool {
    let mint_info = mint_account.to_account_info();
    let mint_data = mint_info.data.borrow();

    let mint = StateWithExtensions::<spl_token_2022::state::Mint>::unpack(&mint_data)
        .map_err(|_| return false)
        .unwrap();

    let extensions = mint.get_extension_types().unwrap();

    for e in extensions {
        // Only allow extensions explicitly compatible with the protocol
        if e != ExtensionType::MetaDataPointer {
            return false;
        }
    }

    true
}
```

Fix Summary:

- Validate Token-2022 mints before processing to prevent unexpected behaviors.
- Explicitly allow only safe extensions that do not interfere with protocol logic.
- Reject tokens with fees-on-transfer or other unsupported behaviors to avoid fund loss.

This ensures the protocol functions correctly without unexpected side effects from arbitrary Token-2022 extensions.

Perena Addressed in PR 37. Other than `TransferFeeAmount`, what other extensions would cause loss of funds?

Cantina Managed: Verified.

3.3.2 Missing Confidence Validation in Pyth Oracle Price

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: The trading functions that use the Pyth oracle price do not validate confidence values (`conf` and `ema_conf`). They only use `get_price_no_older_than` to check staleness but do not ensure that the confidence percentage remains within an acceptable threshold. This omission can lead to:

- Using prices during high uncertainty periods, which may not reflect accurate market conditions.
- Missing market anomaly signals, potentially overlooking extreme price deviations.
- Potential incorrect liquidations during low market confidence, leading to unfair liquidations or improper risk calculations.

As per Pyth's best practices, confidence intervals should be considered when evaluating price validity.

Recommendation: Introduce a configurable `max_confidence_pct` parameter in the price account and validate the confidence percentage before using the price. Updated `buy` Function with Confidence Validation:

```
pub fn get_price(pyth_info: &AccountInfo, asset: &AssetInfo) -> Result<Price, ProgramError> {
// Fetch price and confidence from Pyth oracle
let pyth_price_result =
            price_update.get_price_no_older_than(&Clock::get()?, maximum_age, &feed_id);
let (pyth_price, conf) = match pyth_price_result {
            Ok(price) => (price.price, price.conf),
            Err(e) => Err(e)?,
        };

// Validate confidence percentage
let confidence_pct = (conf as u128)
            .checked_mul(100)?
            .checked_div(pyth_price.abs() as u128)?;
require!(
            confidence_pct <= max_confidence_pct as u128,
CustomError::PriceConfidenceTooHigh
        );


Ok(())
}
```

Perena: Fixed in PR 36.

Cantina Managed: Fix verified.

3.3.3 Token account creation got bypassed through zero lamport check

Severity: Medium Risk

Context: helpers/mod.rs#L137, helpers/mod.rs#L168

Description: In the `mint_to()` function in `helpers/mod.rs`, there is a vulnerability in how the code verifies whether an Associated Token Account (ATA) exists before attempting to create one. The current implementation checks whether the ATA has any lamports:

```
if ata_info.lamports() == 0 {
    // create ATA if it doesn't exist
}
```

This check is problematic because it only verifies if the account has SOL (lamports), not whether it's actually initialized as a valid token account. A malicious user could exploit this by sending a minimal amount of lamports to an uninitialized ATA address, thereby bypassing the account creation logic.

When this happens, the subsequent call to `mint_to_checked()` would fail because the ATA hasn't been properly initialized as a token account, despite having non-zero lamports. This creates a Denial of Service (DoS) condition where users cannot receive minted tokens.

Recommendation: Instead of checking the lamport balance of the ATA, we can call to `create_associated_token_account_idempotent()` without condition, because the CPI will handle the case when the account already exists.

Perena:

- We added this condition to save CU.
- A malicious user could send a minimal amount of lamports, then fail at next CPI `mint_to_checked`.

Cantina Managed: Verified.

3.3.4 Incorrect Fee Rate Used in Unstake Function Leads to Financial Loss

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: In the `stake.rs` processor, there is an issue where the code uses the wrong fee rate variable when calculating the token conversion amount. Specifically, in the `handle()` function when a user stakes their USD Prime tokens to receive USD Star tokens, the calculation uses `bank.star_fee_rate` instead of

the correct `bank.prime_fee_rate`. The problematic calculation appears when determining how many USD Star tokens to mint after burning USD Prime tokens:

```
amount = instruction_data.star_amount * (FEE_RATE_SCALE - bank.star_fee_rate) / FEE_RATE_SCALE
```

Since the operation is converting from USD Prime to USD Star, the calculation should be using `bank.prime_fee_rate`. The calculation is intended to apply the appropriate fee for the burned token (USD Prime) but incorrectly applies the fee rate for the minted token (USD Star). This issue has direct financial implications. If the `prime_fee_rate` is higher than the `star_fee_rate`, users will receive more USD Prime tokens than they should, resulting in an economic loss for the protocol. Conversely, if the `prime_fee_rate` is lower, users will receive fewer tokens than they should, resulting in financial losses for the users.

Recommendation: The fee rate calculation should be corrected to use the appropriate fee rate in `stake()` function.

Perena: Using star fee rate for both is intentional

Cantina Managed: Acknowledged.

3.4 Low Risk

3.4.1 Inconsistent Error Handling Patterns

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: Throughout the codebase, there are inconsistent error handling patterns which reduces readability and maintainability. Some errors are handled using conditional statements with `return Err()` while other validations use `assert!()` macro. For example, in the `handle()` function in `asset_remove.rs`, the code uses assertions for some validations:

- `assert!(spl_token::check_id(token_program_info.key) || spl_token_2022::check_id(token_program_info.key));`.
- `assert!(reserve_info.owner == token_program_info.key);`.
- `assert!(&expected_bank_pda == bank_info.key);`.

While other validations use conditional statements with explicit error returns:

- `if !signer_info.is_signer { return Err(ProgramError::MissingRequiredSignature); }`.
- `if bank_info.owner != program_id { return Err(ProgramError::IllegalOwner); }`.
- `if bank.admin_key != *signer_info.key { return Err(ProgramError::InvalidAccountData); }`.

This inconsistency makes it harder to understand the error handling strategy of the program and could lead to confusion during code maintenance or review. It also affects error reporting since assertions will panic with less descriptive error messages compared to the explicit `ProgramError` types.

Recommendation: Standardize the error handling approach across the codebase. Given that Solana programs benefit from detailed error types, prefer using explicit error returns over assertions for all validation checks:

Perena: Fixed in PR 46.

Cantina Managed: Verified.

3.4.2 Missing ATA Initialization in Token Transfer Function

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: In the `transfer_signed()` function in `helpers/mod.rs`, there is a potential issue where the receiver's Associated Token Account (ATA) might not be initialized before attempting a token transfer. Unlike other token operation functions in the codebase (such as `mint_to()`), the `transfer_signed()` function does not check or create the receiver's ATA prior to transfer.

When transferring tokens to a user, the function assumes that the recipient's ATA already exists. However, if the recipient has never interacted with this particular token before, their ATA for that token will not exist, causing the transfer operation to fail.

The `create_associated_token_account_idempotent()` function from the SPL Associated Token Account program is already being used elsewhere in the codebase (as seen in the `mint_to()` function). This function can safely handle cases where the ATA already exists, making it ideal for this scenario.

Recommendation: Add ATA initialization support to the `transfer_signed()` function by calling `create_associated_token_account_idempotent()` before performing the token transfer, similar to how it's implemented in the `mint_to()` function.

Perena: Fixed in PR 48.

Cantina Managed: Verified.

3.4.3 Unnecessary Use of Invoke Signed in Burn Function

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: In the `burn()` function within `helpers/mod.rs`, there is an unnecessary use of `invoke_signed()` when `invoke()` would be more appropriate. The current implementation uses `invoke_signed()` to call the `burn_checked` instruction, passing a PDA (Program Derived Address) signer seed:

```
invoke_signed(
    &spl_token_2022::instruction::burn_checked(
        mint_info.owner,
        ata_info.key,
        mint_info.key,
        signer_info.key,
        &[],
        amount,
        decimals,
    )?,
    &[ata_info.clone(), mint_info.clone(), signer_info.clone()],
    &[&[BANK_SEED, &[bank_bump]]],
)?;
```

However, the CPI only needs the authority of the `signer_info` parameter rather than the PDA authority (the bank account). Since `signer_info` is presumably already a signer for the transaction, there's no need to use `invoke_signed()` which is specifically designed for when a program needs to sign on behalf of a PDA it controls. Using `invoke_signed()` unnecessarily adds complexity to the code and could lead to confusion during maintenance or future development.

Recommendation: Replace the `invoke_signed()` call with a standard `invoke()` call since the authority being used is already a transaction signer.

Perena: Fixed in PR 48.

Cantina Managed: Verified.

3.5 Informational

3.5.1 Unnecessary Size for Assets Counter

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: In the `Bank` struct found in `src/state.rs`, the `num_assets` field is defined as a `u16` which is a 16-bit integer capable of storing values from 0 to 65,535. However, as indicated by the constant `MAX_TOKENS_PER_BANK` being set to `64`, this field will never exceed that value.

Recommendation: Replace the `u16` type with `u8` for the `num_assets` field in the Bank account since it never needs to exceed 255.

Perena: Fixed in PR 44.

Cantina Managed: Verified.

3.5.2 Unused Weight Parameter in `AssetInfo` Struct

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: In the `AssetInfo` struct in `src/state.rs`, there's a `weight` field that appears to be unused in the contract's logic. Despite being declared and stored in the struct, this field is not utilized in any meaningful way in the program's operation.

Recommendation: There are two potential courses of action:

1. If the `weight` field was intended for future functionality but is not yet implemented, it should be clearly documented as reserved for future use.
2. If the field is genuinely not needed, it should be removed from the struct to optimize storage usage and clarify the code.

Perena: Fixed in PR 44.

Cantina Managed: Verified.

3.5.3 Redundant Reserve Account Validation in Admin Swap Function

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: The `admin_swap()`, `swap_in()`, and `swap_out()` functions contain redundant account validation logic that's never executed due to earlier validation that accomplishes the same purpose. This redundancy increases code size and creates maintenance confusion without providing any security benefit.

First, the code uses `assert!` statements to verify that the reserve accounts match the expected values:

```
assert!(
    bank.assets[instruction_data.input_asset_index as usize].reserve == *input_reserve_info.key
);
assert!(
    bank.assets[instruction_data.output_asset_index as usize].reserve
        == *output_reserve_info.key
);
```

Later, the code performs nearly identical checks using an `if` statement:

```
if &input_asset.reserve != input_reserve_info.key
    || &output_asset.reserve != output_reserve_info.key
{
    return Err(ProgramError::InvalidAccountData);
}
```

This redundancy doesn't pose a security risk, but it does indicate a lack of code quality and could lead to maintenance issues if only one check is updated in the future.

Recommendation: Consider removing the redundant check after the assertions, as it will never be executed.

Perena: Fixed in PR 42.

Cantina Managed: Verified.

3.5.4 Unnecessary Type Casting on Bank Bump Variable

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: Throughout the codebase, there are unnecessary type casting operations when the `bank.bump` value is assigned to the local `bank_bump` variable:

```
let bank_bump = bank.bump as u8;
```

This cast is redundant because `bank.bump` is already of type `u8` or a compatible integer type.

Recommendation: Remove the explicit type cast since it's not necessary.

Perena: Fixed in latest `main`.

Cantina Managed: Verified.

3.5.5 Code Duplication in USD Prime Burn Processor

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: The `usd_prime_burn.rs` processor contains several instances of code duplication that reduce code quality and could lead to maintenance issues. In particular:

1. There is a duplicated assertion check for `asset_info.reserve == *bank_asset_info.key`.
2. There is a duplicated price fetching operation where `get_price(pyth_price_feed_info, &asset_info)?`.

These duplications are unnecessary and could lead to maintenance challenges.

Recommendation: Remove the redundant assertions and operations to improve code quality and maintainability.

Perena: Fixed in latest `main`.

Cantina Managed: Verified.

3.5.6 Unnecessary Account in `usd_star_mint` and `usd_star_burn` Functions

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: The `usd_prime_mint_info` account is being requested and validated in the `usd_star_mint()` and `usd_star_burn()` functions but is never used in the function's business logic. This creates unnecessary overhead in terms of transaction size and potential confusion for developers maintaining the code.

Recommendation: Remove the unnecessary account from both functions.

Perena: Fixed in PR 43.

Cantina Managed: Verified.