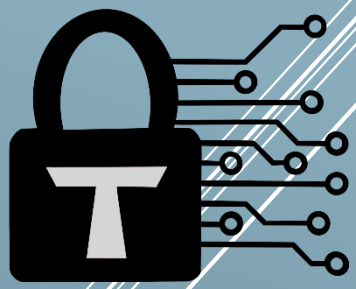
Trust Security



Smart Contract Audit

DTF On Solana

18/05/25

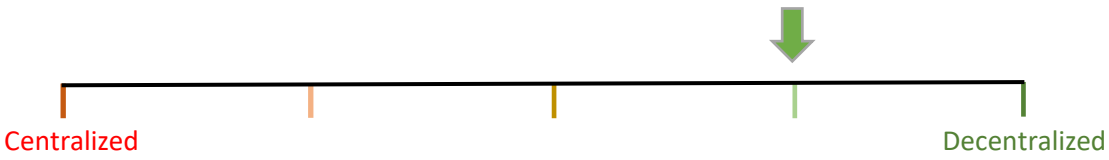
Executive summary

Category	Portfolio Management
Audited file count	100
Lines of Code	7441
Auditor	Carrotsmuggler Frank Castle
Time period	21/04/25-18/05/25

Findings

Severity	Total	Fixed	Acknowledged
High	3	-	-
Medium	3	-	-
Low	5	-	-

Centralization score



Signature

Document properties

Versioning

Version	Date	Description
0.1	18/05/25	Client report
0.2		Mitigation Review

Contact

Trust

trust@trust-security.xyz

Introduction

Trust Security has conducted an audit at the customer's request. The audit is focused on uncovering security issues and additional bugs contained in the code defined in scope. Some additional recommendations have also been given when appropriate.

Scope

- programs/folio/*
- programs/folio-admin/*
- programs/rewards/*

Repository details

- **Repository URL:** https://github.com/reserve-protocol/reserve-index-dtfs-solana
- **Commit hash:** 5352288a38a28085a63be91ebd23493297ceba4b
- **Mitigation review hash:** 1f9133c4c947bdc84f41cbda9a58c60a56ab10e5

About Trust Security

Trust Security has been established by top-end blockchain security researcher Trust, in order to provide high quality auditing services. Since its inception it has safeguarded over 30 clients through private services and over 30 additional projects through bug bounty submissions.

About the Auditors

Carrotsmuggler competes in public audit contests on various platforms with multiple Top 3 finishes. He has experience reviewing contracts on diverse EVM and non-EVM platforms.

Frank Castle is a professional smart contract security researcher specializing in auditing Rust-based contracts and decentralized infrastructure. He has conducted over 35 Solana security audits for some of the largest protocols in the ecosystem.

Disclaimer

Smart contracts are an experimental technology with many known and unknown risks. Trust Security assumes no responsibility for any misbehavior, bugs or exploits affecting the audited code or any part of the deployment phase.

Furthermore, it is known to all parties that changes to the audited code, including fixes of issues highlighted in this report, may introduce new issues and require further auditing.

Methodology

In general, the primary methodology used is manual auditing. The entire in-scope code has been deeply looked at and considered from different adversarial perspectives. Any additional dependencies on external code have also been reviewed.

Qualitative analysis

Metric	Rating	Comments
Code complexity	**Good**	Project kept code complexity low, reducing attack risks
Documentation	**Good**	Project is mostly very well documented.
Best practices	**Good**	Project consistently adheres to industry standards.
Centralization risks	**Good**	Project does not introduce significant unnecessary centralization risks.

Findings

High severity findings

TRST-H-1 Incorrect token extension check

- **Category:** Input validation
- **Source:** token_util.rs
- **Status:** Open

Description

The *Folio* program currently defines a set of forbidden SPL Token-2022 extensions for both mint and token accounts. However, it mistakenly includes *TransferFeeConfig* in the list of forbidden token extensions, even though it is a **mint-only** extension according to the official Token-2022 documentation and implementation. As a result, the program does not properly restrict mints that use the *TransferFeeConfig* extension, potentially allowing tokens with this configuration to be used within the protocol. It could lead to unexpected behavior, security failures, or loss of funds if such tokens are accepted and processed by the protocol.

```
/// The forbidden mint extension types.
pub const FORBIDDEN_MINT_EXTENSION_TYPES: [ExtensionType; 3] = [
    ExtensionType::TransferHook,
    ExtensionType::ConfidentialTransferMint,
    ExtensionType::PermanentDelegate,
];

/// The forbidden token extension types.
pub const FORBIDDEN_TOKEN_EXTENSION_TYPES: [ExtensionType; 2] = [
    ExtensionType::TransferFeeConfig,
    ExtensionType::MemoTransfer,
];
```

As a result, some extensions, such as *TransferFeeConfig*, can only be initialized with mint accounts, not token accounts.

Recommended mitigation

Move *ExtensionType::TransferFeeConfig* from the *FORBIDDEN_TOKEN_EXTENSION_TYPES* array to the *FORBIDDEN_MINT_EXTENSION_TYPES* array. This will ensure that mint extensions are validated properly and prevent support for incompatible or dangerous configurations.

Team response

Team changed checks to an allowed list of extensions.

Mitigation Review

The fix looks good.

TRST-H-2 Multiple simultaneous auctions can be started for the same token pair

- **Category:** Input validation
- **Source:** open_auction.rs
- **Status:** Open

Description

The current program implementation allows multiple simultaneous auctions to be opened for the same token pair. The *auction_ends* account creation uses *init_if_needed*, and then uses the *process_init_if_needed* function call to initialize it if necessary. However, this function blindly overwrites the account, assuming it is fresh.

```
    pub fn process_init_if_needed(
        &mut self,
        bump: u8,
        sell_token: Pubkey,
        buy_token: Pubkey,
        rebalance_nonce: u64,
    ) -> Result<()> {
        let (token_mint_1, token_mint_2) = AuctionEnds::keys_pair_in_order(sell_token,
buy_token);


        self.bump = bump;
        self.rebalance_nonce = rebalance_nonce;
        self.token_mint_1 = token_mint_1;
        self.token_mint_2 = token_mint_2;
        self.end_time = 0;


        Ok(())
    }
```

There is no validation to check if an auction for the same pair is already in progress, allowing multiple overlapping auctions for the same pair to be created and initialized. There is a validation present in the internal *open_auction* call which checks the *end_time*, but by then this value will have already been overwritten.

```
        {
            check_condition!(
                current_time > auction_ends.end_time + auction_buffer,
                AuctionCollision
            );
        }
```

Opening simultaneous auctions can lead to multiple problems. Reopening auctions resets the price to a higher value, delaying auction closure and griefing bidders. Each re-open also resets the *end_time*, breaking expected timing constraints. Thus, rebalances can be indefinitely delayed, which is an essential part of the protocol.

Recommended mitigation

Modify the *process_init_if_needed* function to check if auctions are ongoing. If so, new auctions should not be started.

Team response

Added a check to skip assignment if the bump and tokens match the current auction.

Mitigation Review

Fixed by blocking simultaneous auctions.

TRST-H-3 Missing basket state update during migration

- **Category:** Incorrect accounting
- **Source:** migrate_folio_tokens.rs
- **Status:** Open

Description

Folios can be migrated. In this case, a new folio account is created and the token balances are migrated from the old folio to the new one. This is handled by the *migrate_folio_tokens* instruction.

The issue is that after transferring the tokens to the new folio, the program correctly updates the state of the folio basket of the old folio, but does not do so for the new one.

```
// Remove the token from the old folio basket
old_folio_basket.remove_token_mint_from_basket(token_mint.key())?;
@audit missing similar update for the new folio basket
```

The program tracks its token balances in its *basket.token_amounts* account. This is updated during mints, burns and bids. This is necessary to track the folio minted assets, and to ignore the folio pending assets. The raw balance of the basket account will include both these balances, but *basket.token_amounts* only tracks the minted balances, which is the amount actually needed for calculating mints and burns.

However, since the new folio basket's *token_amounts* account is not updated, these migrated tokens won't be seen as minted assets in the new folio. Thus, mints and burns in the new folio will give incorrect results. The *folio_basket.rs* file already implements the *add_tokens_to_basket* function which is designed to be used for this update and needs to be called for the new folio.

Recommended mitigation

Call *add_tokens_to_basket* on the new folio basket to update the recorded minted token amounts.

Team response

Added functions through which the new folio can update its state from the old folio.

Mitigation Review

Fixed as long as future versions implement the function to update their state right after migration.

Medium severity findings

TRST-M-1 *claim_rewards* doesn't claim all rewards

- **Category:** Logical flaws
- **Source:** claim_rewards.rs
- **Status:** Open

Description

In the *claim_rewards* handler, only a few of the remaining accounts are sent forward to the *accrue_rewards* call. *REMAINING_ACCOUNTS_UPPER_INDEX_FOR_ACCRUE_REWARDS* is defined as 4, so only the first 4 accounts are sent forward.

```
    RewardsProgramInternal::accrue_rewards(
        &ctx.accounts.system_program,
        &ctx.accounts.token_program,
        &ctx.accounts.realm,
        &ctx.accounts.governance_token_mint,
        &ctx.accounts.governance_staked_token_account,
        &ctx.accounts.user,
        &ctx.accounts.caller_governance_token_account,
        &ctx.accounts.reward_tokens,
        // Only the first 4 accounts are needed for the accrue rewards instruction, the
last one is for claim only
        &ctx.remaining_accounts[0..REMAINING_ACCOUNTS_UPPER_INDEX_FOR_ACCRUE_REWARDS],
        true,
    )?;
```

The issue is that the *accrue_rewards* function expects to be sent in all the accounts, and process them 4 at a time. Each block of 4 accounts represents the accounts necessary to process a single reward token mint.

```
        let reward_token = next_account(
            &mut remaining_accounts_iter,
            false,
            false,
            &token_program_id,
        )?;
        let reward_info = next_account(
            &mut remaining_accounts_iter,
            false,
            true,
            &RewardsProgram::id(),
        )?;
        // Token rewards' token account
        let token_rewards_token_account = next_account(
            &mut remaining_accounts_iter,
            false,
            token_reward_token_account_is_mutable,
            &token_program_id,
        )?;
        let caller_reward_info = next_account(
            &mut remaining_accounts_iter,
            false,
            true,
            &RewardsProgram::id(),
        )?;
```

Thus, since only the first 4 accounts are sent to it, it only calculates the rewards for the first reward token. The other rewards are not calculated. However, when it is time to distribute the rewards, the *claim_rewards* handler tries to distribute the rewards for all the token mints, even though they haven't been calculated yet. Thus, this function correctly dispenses rewards only for the first reward token mint sent in the remaining accounts array.

Recommended mitigation

Call *accrue_rewards* with all the reward token mints so that all the rewards are correctly updated before distributing tokens.

Team response

accrue_rewards is now called for each reward token mint to update all the rewards.

Mitigation Review

Fixed following recommendation.

TRST-M-2 Deferred prices are not handled in permissioned auction opens, leading to DoS

- **Category:** Denial of Service
- **Source:** auction.rs
- **Status:** Open

Description

The *open_auction* function correctly prevents permissionless auction opens when deferred prices are involved. However, it fails to perform similar validation when permissioned actors call the function. This oversight leads to unvalidated usage of potentially zero-valued prices in calculations, which can result in a division-by-zero error.

```
        if is_permissionless {
            // Only open auctions that have not timed out (ttl check) and are available
to be opened permissionlessly.
            check_condition!(
                current_time >= rebalance.restricted_until
                    && current_time <= rebalance.available_until,
                AuctionCannotBeOpenedPermissionlesslyYet
            );
            // If any price is non-zero, all are non-zero.
            check_condition!(
                buy_details.prices.low != 0,
                AuctionCannotBeOpenedPermissionlesslyWithDeferredPrice
            );
        }
```

If any deferred price is left at zero, the function later performs division on it when calculating the start and end prices, leading to division by zero errors.

```
    let old_start_price = Decimal::from_scaled(sell_details.prices.high)
            .mul(&Decimal::ONE_E18)?
```

```
            .div(&Decimal::from_scaled(buy_details.prices.low))?
            .to_scaled(Rounding::Ceiling)?;


        let old_end_price = Decimal::from_scaled(sell_details.prices.low)
            .mul(&Decimal::ONE_E18)?
            .div(&Decimal::from_scaled(buy_details.prices.high))?
            .to_scaled(Rounding::Ceiling)?;
```

The objective of the deferred price is that if the rebalance manager choses to defer prices, the auction opener can supply their own prices in the config, essentially giving a different role the ability to set prices. However, this functionality is broken here due to the division by zero error. When prices are deferred, permissionless auction starts aren't allowed and permissioned auction starts aren't possible due to the issue highlighted above. Therefore, this deferred rebalance configuration cannot be used by anyone and leads to DoS until the rebalance configuration is changed by the *RebalanceManager* role.

Recommended mitigation

For permissioned callers, the start and end prices don't need to be calculated since they are set from the config. Bypass this behavior to prevent the division by zero errors for deferred prices.

Team response

In case of deferred prices, start and end prices are not calculated.

Mitigation Review

Fixed by avoiding division by zero calculation.

TRST-M-3 Possible re-entrancy attack during bids

- **Category:** Reentrancy attacks
- **Source:** bid.rs
- **Status:** Open

Description

The *bid.rs* handler function handles user bids during rebalance operations. In this function, the user has the option to also specify a callback cpi call. When a callback is specified, the program records the current token balance, does a cpi call to the specified account and then checks its balance again, expecting it to have increased by the required amount.

```
        let raw_folio_buy_balance_before = ctx.accounts.folio_buy_token_account.amount;


        cpi_call(ctx.remaining_accounts, callback_data)?;


        // Validate we received the proper funds
        ctx.accounts.folio_buy_token_account.reload()?;


        check_condition!(
```

```
            ctx.accounts
                .folio_buy_token_account
                .amount
                .checked_sub(raw_folio_buy_balance_before)
                .ok_or(ErrorCode::MathOverflow)?
                >= raw_bought_amount,
            InsufficientBid
        );
```

The issue here is in the callback, the user can also increase the program's balance by doing a normal deposit through calling *add_to_pending_basket*, satisfying the bid balance increment check without actually paying for the bid.

The program tries to combat this by adding a check in the *cpi_call* function, making sure that the folio program itself cannot be the callback address.

```
        check_condition!(
            callback_program.key() != FOLIO_PROGRAM_ID,
            InvalidCallbackProgram
        );
```

However, this can be bypassed by using another program in between, where the user calls a middleman program which calls the folio program doing the deposit.

In the current state, this reentrancy is prevented due to the cpi call depth limit. Solana only allows a max of 4 nested cpi calls and the middleman program attack vector would result in a depth of 5. However, the current *cpi_call* check is still not secure enough to completely eliminate this attack vector and thus should be patched.

Recommended mitigation

In the *cpi_call* function, make sure none of the account addresses passed in via the *remaining_accounts* variable correspond to the folio program address.

Team response

Added a check to make sure the folio account is not present in the remaining accounts for callback.

Mitigation Review

Fixed by making sure folio account cannot be accessed during callback.

Low severity findings

TRST-L-1 Reward tokens once removed cannot be re-added

- **Category:** Missing functionality
- **Source:** remove_reward_token.rs
- **Status:** Open

Description

When a reward token is removed, the *is_disallowed* value in its info account is set to false. All other calculations check this value, and if false, skip calculations, basically bypassing the system for disallowed tokens.

```
        // Set to null in reward token list
        self.reward_tokens[reward_token_position.unwrap()] = Pubkey::default();


        reward_info.is_disallowed = true;
```

The issue is that this token can never be added in again as a reward token. So, the only way to add it back in is by re-deploying the whole program or upgrading it, since there is no functionality to change this flag back to true.

Recommended mitigation

Consider allowing the token to be added back in again via the *add_reward_token* function after clearing its info state.

Team response

Acknowledged.

Mitigation Review

Acknowledged.

TRST-L-2 The *sell_tokens* amount should be rounded down

- **Category:** Rounding error
- **Source:** auction.rs
- **Status:** Open

Description

During auctions, the amount of tokens to be sold is calculated based on an admin provided fraction and the current amount of share tokens in circulation. This is done in the auction.rs file.

```
            let sell_tokens = scaled_folio_token_total_supply
                .mul(&Decimal::from_scaled(auction_spot_sell_limit))?
                .div(&Decimal::ONE_E18)?
                .to_scaled(Rounding::Ceiling)?;
```

The issue is that this calculation rounds up the result. Thus, for high values of *auction_spot_sell_limit*, this rounding up can lead to *sell_tokens* being higher than the available amount, making the transaction revert.

Recommended mitigation

For the tokens to be sold, round the calculation down.

Team response

Fixed by rounding down the *sell_tokens* amount.

Mitigation Review

Fixed following recommendation.

TRST-L-3 Rebalance nonce doesn't reflect number of rebalances

- **Category:** Incorrect accounting
- **Source:** rebalance.rs
- **Status:** Open

Description

The nonce field in the rebalance account is intended to track unique rebalances, but in the current implementation, it behaves more like a counter of how many times *start_rebalance* was called, even for the same rebalance initialization. This is because the nonce is incremented in the *start_rebalance* function call, regardless of whether the nonce was updated already, or this rebalance will actually take place or not.

```
        self.nonce = self
            .nonce
            .checked_add(1)
            .ok_or(error!(ErrorCode::MathOverflow))?;
```

Rebalances are only confirmed and put into place when its *all_rebalance_details_added* flag is set to 1. Until then, the rebalance is still in its editable phase and not finalized. So, the nonce does not count the number of finalized rebalances, and instead counts the number of times *start_rebalance* has been called.

Recommended mitigation

If the nonce is expected to reflect the number of finalized rebalances, consider moving the rebalance nonce update to a point after the rebalance state has been frozen.

Team response

Acknowledged.

Mitigation Review

Acknowledged.

TRST-L-4 Unaccrued rewards are lost when *reward_token* is removed

- **Category:** Missing functionality
- **Source:** remove_reward_token.rs
- **Status:** Open

Description

When reward tokens are removed by calling the *remove_reward_token* instruction, the info account's *its_disallowed* flag is changed to true.

```
        self.reward_tokens[reward_token_position.unwrap()] = Pubkey::default();


        reward_info.is_disallowed = true;
```

This means that on future *accrue_rewards* calls, this token's reward calculation will be skipped due to the check shown below.

```
        check_condition!(!self.is_disallowed, DisallowedRewardToken);
```

Because of this check, users who triggered the *accrue_rewards* instruction prior to the token's disablement will have their accrued rewards recorded and can still claim them. However, users who didn't call this instruction before the token was disallowed will permanently lose access to their pending rewards since the accrual calculation will be skipped for disallowed tokens.

Recommended mitigation

If unaccrued rewards are expected to be available, call *accrue_rewards* in the *remove_reward_token* function before removing the token. Furthermore, in the *accrue_rewards* function, calculate the rewards for the disallowed tokens as well based on the stored *scaled_delta_index* values.

Team response

Fixed by allowing reward updates even when token disabled. Global reward updates are skipped.

Mitigation Review

Fixed by skipping global reward updates but allowing user reward updates once tokens are disabled.

TRST-L-5 *burn_folio_token* missing slippage control

- **Category:** Slippage
- **Source:** burn_folio_token.rs
- **Status:** Open

Description

The *burn_folio_token* instruction burns a specified amount of folio tokens and pays out the constituent tokens of the folio according to their composition. The issue is that the instruction does not take any expected amounts as parameters, so the tokens received can be different from what the user expects.

It is common practice to add a *minimum_shares_received* parameter for mint operations and a *minimum_tokens_received* parameter for burn operations when interacting with a vault. Otherwise, the user can receive an unexpected number of tokens if the composition of the vault changes while they are making the call.

While the program here implements a *min_raw_shares* during mint instructions, such a slippage parameter is missing during the burn instruction. Thus, if the composition of the vault changes due to a rebalance auction result while a user is trying to burn their share tokens, they can get an unexpected mix of the constituent tokens.

Recommended mitigation

Consider adding a *minimum_expected_amounts* array parameter which will contain the expected mints and their amounts from the burn operation.

Team response

Implemented an optional user argument to specify slippage for a select number of tokens.

Mitigation Review

Fixed by allowing users to specify a minimum output number of tokens.

Client-Reported issues

TRST-CL-1 Incorrect time validation in *open_auction*

- **Category:** Validation
- **Source:** auction.rs
- **Status:** Fixed

Description

The expected lifecycle of an auction is set during start_rebalance, and is defined as:

```
        check_condition!(ttl <= MAX_TTL, RebalanceTTLExceeded);
        check_condition!(
            ttl >= auction_launcher_window,
            RebalanceAuctionLauncherWindowTooLong
        );
        self.nonce = self
            .nonce
            .checked_add(1)
            .ok_or(error!(ErrorCode::MathOverflow))?;


        self.started_at = current_time;
        self.restricted_until = current_time + auction_launcher_window;
        self.available_until = current_time + ttl;
```

This creates the following timeline:

current_time --------> restricted_until --------> available_until

Auctions can be created only after *restricted_until* and before *available_until*. However, in *open_auction*, the validation logic is flawed.

```
        check_condition!(current_time <= rebalance.restricted_until, AuctionTimeout);

        check_condition!(
            current_time >= rebalance.started_at + auction_buffer
                && current_time >= rebalance.available_until
                && rebalance.rebalance_ready(),
            FolioNotRebalancing
        );
```

This expects the current time to be more than *available_until* and less than *restricted_until*. That is impossible since *available_until* is initialized to be always higher than *restricted_until*. The same logical flaw is repeated inside the *is_permissionless* clause.

```
            check_condition!(
                current_time >= rebalance.available_until
                    && current_time <= rebalance.restricted_until,
                AuctionCannotBeOpenedPermissionlesslyYet
            );
```

Implemented mitigation

Issue was fixed by fixing the checks. Now the *current_time* is expected to be between *restricted_until* and *available_until*, like designed.

TRST-CL-2 Improper *load_init* usage on existing account

- **Category:** Incorrect variable loading
- **Source:** start_rebalance.rs
- **Status:** Fixed

Description

In the *start_rebalance.rs* handler, the program initializes the rebalance account with an *init_if_needed* constraint.

```
    #[account(
        init_if_needed,
        payer = rebalance_manager,
        space = Rebalance::SIZE,
        seeds = [REBALANCE_SEEDS, folio.key().as_ref()],
        bump
    )]
    pub rebalance: AccountLoader<'info, Rebalance>,
```

This makes sure the account is initialized, and a further explicit initializer function is used.

```
    // Initialize rebalance account if needed
    Rebalance::process_init_if_needed(
        &mut ctx.accounts.rebalance,
        ctx.bumps.rebalance,
        &folio_key,
    )?;
```

However, immediately after calling *process_init_if_needed*, the code incorrectly attempts to reinitialize the same account using *load_init()*:

```
    let rebalance = &mut ctx.accounts.rebalance.load_init()?;
```

If the rebalance account already exists and was previously initialized, *load_init()* will fail with:

"The account discriminator was already set on this account."

Implemented mitigation

load_mut() is used instead to load the account.

TRST-CL-3 Incorrect status in fee-distribution

- **Category:** Validation
- **Source:** start_folio_migration.rs
- **Status:** Fixed

Description

In the *start_folio_migration* instruction, the old folio's status is set to **Migrating**, and then the *distribute_fees* function is called.

```
    {
        let old_folio = &mut ctx.accounts.old_folio.load_mut()?;
```

```
        old_folio_bump = old_folio.bump;


        ctx.accounts.validate(old_folio)?;


        // Update old folio status
        old_folio.status = FolioStatus::Migrating as u8;
    }

    // Distribute the fees
    ctx.accounts
        .distribute_fees(ctx.remaining_accounts, index_for_fee_distribution)?;
```

However, the *distribute_fees* function requires the status to be Killed or Initialized, or it skips the fee distribution.

```
            if ![FolioStatus::Killed, FolioStatus::Initialized].contains(&folio_status) {
                return Ok(());
            }
```

Since the status is changed before the fee distribution, the fee distribution step will always be skipped.

Implemented mitigation

The status update was moved to be done after the fee distribution.

TRST-CL-4 Incorrect decimal scaling

- **Category:** Decimals
- **Source:** auction.rs
- **Status:** Fixed

Description

In the *open_auction* function, the *sell_tokens* and *buy_tokens* values are calculated in scaled 18 decimals.

```
let buy_tokens = scaled_folio_token_total_supplyAdd commentMore actions
                .mul(&Decimal::from_scaled(auction_spot_buy_limit))?
                .div(&Decimal::ONE_E18)?
                .to_scaled(Rounding::Ceiling)?;
```

These had to be calculated in the native decimals of the token instead.

Implemented mitigation

The decimals were scaled using the *to_token_amount* function to be in line with the token decimals.

Additional recommendations

TRST-R-1 Incorrect reward update in governance

In the Governance program, during deposits through the *process_deposit_governing_tokens* function, the reward debt is updated by calling *accrue_reward*. The issue is that this is called after the *mint_spl_tokens_to* call, which invokes a mint instruction, which changes the governance staked balance.

So, the code first updates the user's staked balance and **then** accrues their rewards. This leads to incorrect rewards, since the user's rewards are calculated on the freshly increased balance.

This can lead to theft of reward tokens, since the rewards are calculated on incorrect staked token amounts.

Move the reward accrual to the top, before any changes to any token amounts.

The Governance program is out of scope, hence the finding has been classified as a recommendation.

TRST-R-2 *update_fee_recipients* does not check for duplicates

The *update_fee_recipients* function is used to update the list of fee recipients for a folio. The issue is that the addition mechanism does not check for duplicates.

```
        for new_recipient in filtered_fee_recipients_to_add {
            check_condition!(add_index < MAX_FEE_RECIPIENTS, InvalidFeeRecipientCount);
            new_recipients[add_index] = new_recipient;
            add_index += 1;
```

This allows the same recipient to appear multiple times in the list, which could have been due to an admin mistake.

It is recommended to check for duplicates during recipient addition.

TRST-R-3 Missing *poke_folio* call in *add_rebalance_details*

In the *start_rebalance.rs* handler, the folio is poked to trigger its fee share calculations. While the total supply or the result of the update state isn't used anywhere in the function, it is still good to update the state before starting auction proceedings. However, while the *add_rebalance_details* instruction call also does similar things, it does not poke the folio.

It is recommended to either poke the folio in both instruction calls, or remove them from both to maintain parity between the two functions.

TRST-R-4 Conflicting comments for *MAX_REWARD_TOKENS*

In the *state.rs* file of the *rewards* program, a comment states that a max of 30 reward tokens is allowed.

```
    /// Max of 30 reward tokens.
    pub reward_tokens: [Pubkey; MAX_REWARD_TOKENS],
```

However, in the actual definition, *MAX_REWARD_TOKENS* is only 4.

```
/// MAX_REWARD_TOKENS is the maximum number of reward tokens that can be set for a folio,
4.
pub const MAX_REWARD_TOKENS: usize = 4;
```

It is recommended to correct this conflicting comment.

Centralization risks

TRST-CR-1 Folio owner has complete control

The Folio program's **owner** role can add and remove tokens to the folio as they please, with the *add_to_basket* and *remove_from_basket* functions. They can also change the folio configuration through the *update_folio* function. Thus, the Owner role has complete control over the system and is held by governance. To safeguard against private key leaks, it is recommended that the governance looks into eventually transferring the Owner to a timelocked multi-sig.

TRST-CR-2 Auction prices can be stale

The *RebalanceManager* role sets up a rebalance configuration which includes setting up price ranges for the constituent tokens. The Actual rebalance happens in a Dutch auction, where the price starts from a high value and drops over time. The issue is that these set price can be outdated. If say the price of token increases drastically, such that the initial price of the auction itself is underpriced, then the system will keep conducting unprofitable auctions, damaging the economic soundness of the system. Thus, set price ranges in the rebalance system need to be monitored and adjusted whenever needed.

Systemic risks

TRST-SR-1 Auctions dependent on network traffic

Rebalances conduct a Dutch auction, which counts on active bidders competing against each other in order to resolve auctions at the best price. The issue is that in times of high network usage or network outages, these assumptions don’t hold true. So even if bidders are unable to get their bids in, the price will continue to drop which can lead to badly priced auctions, leaking value from the protocol.