Security Assessment Report

Huma Vault

January 04, 2026

Summary

The Sec3 team was engaged to conduct a thorough security analysis of the Huma Vault program.

The artifact of the audit was the source code of the following programs, excluding tests, in a private repository.

The initial audit focused on the following versions and revealed 16 issues or questions.

Task	Type	Commit
Huma Vault	Solana	c084dd7 (Dec 02, 2025)

The post-audit review was concluded on commit 769001b (Jan 04, 2026).

This report provides a detailed description of the findings and their respective resolutions.

Table of Contents

Result Overview

Issue	Impact	Status
HUMA VAULT		
[C-01] Missing account validation in `PSTStrategy`	Critical	Resolved
[M-01] Targeted Denial of Service via malicious NFT transfer and forced merge	Medium	Resolved
[L-01] Huma fees continue to accrue during Pre-Closure status	Low	Resolved
[L-02] Missing `deposit_mint` check in `submit_redemption_request`	Low	Resolved
[L-03] `max_deposit_amount` configuration is not enforced in deposits	Low	Resolved
[L-04] `huma_fee` is not checked in `LockupState::ready_to_close`	Low	Resolved
[L-05] `min_deposit_amount` check bypassed in `migrate`	Low	Resolved
[I-01] Floating-point math in asset projection	Info	Resolved
[I-02] `enable_vault` allows invalid state transitions	Info	Acknowledged
[I-03] Missing validation for Token 2022 extensions in `create_vault`	Info	Resolved
[I-04] Potential arithmetic underflow panic in `process_redemption_request`	Info	Resolved
[I-05] Inconsistent access control documentation for `withdraw_after_vault_closure`	Info	Resolved
[I-06] `strategy_tokens_requested` should be capped at `lender_mode_token.amount`	Info	Acknowledged
[I-07] Arbitrage risk in `recover_loss`	Info	Acknowledged
HUMA VAULT- ISSUES ALSO IDENTIFIED BY THE HUMA TEAM		
[01] Denial of Service via merging of escrowed Deposit Receipts		Resolved
[02] Logic flaws in `submit_redemption_request`		Resolved

Findings in Detail

HUMA VAULT

[C-01] Missing account validation in `PSTStrategy`

The `PSTStrategy` module implements the `VaultStrategyInterface` to manage interactions between the Vault and the underlying Permissionless Pool. Several functions, including `migrate`, `setup_for_deployment`, `deploy`, and `submit_redemption_request`, rely on accounts passed via the `accounts` slice (remaining accounts) to execute logic.

However, the current implementation fails to perform essential validation on these accounts. This oversight leads to critical vulnerabilities involving CPI (Cross-Program Invocation) hijacking and fund theft.

1. Fund Theft via Unchecked Program in `deploy` In the `deploy` function, the `permissionless_program_info` account is used to perform a CPI to the underlying pool. Crucially, this CPI is signed by the `vault_authority`.

```
/* programs/huma-vault/src/strategies/pst_strategy.rs */
191 | let signer_seeds = &[&vault_authority_seeds[..]];
192 | let cpi_ctx = CpiContext::new(
193 |     permissionless_program_info.to_account_info(),
...
210 | .with_signer(signer_seeds);
211 | // Always deposit without commitment.
212 | let num_strategy_tokens =
213 |     cpi::deposit(cpi_ctx, amount, String::from(NO_COMMITMENT), false)?.get();
```

Because the code does not check if `permissionless_program_info` is the genuine Permissionless Pool program, an attacker can supply a malicious program ID. Since the `vault_authority` signs the instruction sent to this malicious program, the malicious program can utilize the signer privilege to CPI into the SPL Token Program and transfer all assets held by the `vault_authority` to an attacker-controlled account.

2. Share Theft via Self-Transfer in `migrate` In the `migrate` function, the `vault_pst_ata_info` account (intended to be the Vault's ATA) is retrieved from the `accounts` slice but is never validated.

```
/* programs/huma-vault/src/strategies/pst_strategy.rs */
088 | // Transfer strategy tokens from the depositor to the vault.
089 | token_utils::deposit_tokens(
090 |     num_strategy_tokens,
091 |     &InterfaceAccount::<'info, Mint>::try_from(mode_mint_info.as_ref())?,
092 |     depositor_info.to_account_info(),
093 |     vault_pst_ata_info.to_account_info(),
094 |     depositor_pst_ata_info.to_account_info(),
```

```
095 |     pst_token_program_info.to_account_info(),
096 | )?;
```

The function calls `deposit_tokens`, which executes a `transfer_checked` instruction. While `token_interface::transfer_checked` ensures that the provided accounts are valid Token Accounts matching the mint, it does not verify that `vault_pst_ata_info` is actually owned by the Vault.

An attacker can invoke `migrate` and provide their own Token Account as both the source (`depositor_pst_ata_info`) and the destination (`vault_pst_ata_info`). The transfer will succeed (Self-transfer), effectively moving no funds to the Vault. However, the `migrate` function assumes the Vault received the tokens and proceeds to mint new Vault shares to the attacker. This allows the attacker to steal Vault shares without depositing any assets.

3. Accounting Manipulation in `migrate` The `migrate` function deserializes `mode_mint_info` to read the token supply for exchange rate calculations without verifying that the mint belongs to the expected pool or mode configuration. An attacker can provide a fake mint with a manipulated supply to distort the exchange rate calculation, leading to incorrect accounting.

Resolution

This issue has been fixed by `bec7191` and `efb33a1`.

HUMA VAULT

[M-01] Targeted Denial of Service via malicious NFT transfer and forced merge

The `merge` instruction allows combining multiple `DepositRecord` accounts into a single one. This instruction is permissionless regarding the `depositor` account, which is passed as an `UncheckedAccount` and not required to be a signer. Any user can invoke this instruction to merge deposit receipts belonging to another user.

During the merge process, the new `DepositRecord` inherits the `deposited_at` timestamp from the most recent input record (`last_deposited_at.max(deposit_record.deposited_at)`).

```
/* programs/huma-vault/src/instructions/depositor/merge.rs */
112 | // Accumulate values. Use the last deposit timestamp for the merged position.
113 | total_shares = total_shares + deposit_record.shares;
114 | last_deposited_at = last_deposited_at.max(deposit_record.deposited_at);
```

This design enables a targeted Denial of Service (DoS) attack where a malicious actor can forcibly lock a victim's liquid assets.

Attack Scenario:

1. **Preparation:** The attacker makes a small deposit into the vault. This generates a new Deposit Receipt (NFT) with a fresh `deposited_at` timestamp, which is subject to the `redemption_cooldown_period_secs`.
2. **Poisoning:** The attacker transfers this "locked" NFT to the victim's wallet. (Since deposit receipts are standard MPL Core assets, they are transferable unless specific hooks prevent it).
3. **Execution:** The attacker calls the `merge` instruction, passing the victim's public key as the `depositor`. Since `depositor` is not required to sign, the transaction proceeds.
4. **Lock-in:** The protocol verifies that the victim owns both the original (liquid) NFT and the transferred (locked) NFT. It then merges them.
5. **Impact:** The new merged record inherits the `deposited_at` timestamp from the attacker's fresh deposit. Consequently, the victim's previously liquid funds are now locked for the full duration of the redemption cooldown period.

An attacker can repeat this process indefinitely, permanently preventing the victim from redeeming their funds.

Resolution

This issue has been fixed by `96e428f`.

HUMA VAULT

[L-01] Huma fees continue to accrue during Pre-Closure status

According to the protocol documentation, Huma fees should be refreshed one final time when a vault enters the `PreClosure` status, after which no further fees should accrue. This behavior is correctly implemented in `enter_pre_closure` and `refresh_assets_for_all_lockups`, which skip fee accrual if the vault is in pre-closure.

However, the `recover_loss` and `update_huma_fee_bps` (part of `manage_vault_config`) instructions bypass this logic. Both instructions explicitly call `vault_state.accrue_huma_fee()`, which does not check the vault's status.

```
/* programs/huma-vault/src/instructions/admin/manage_vault_config.rs */
240 | if old_huma_fee_bps != new_huma_fee_bps {
241 |     let vault_state = ctx.accounts.vault_state.as_mut();
242 |     vault_state.accrue_huma_fee(old_huma_fee_bps)?;
243 |     vault_config.huma_fee_bps = new_huma_fee_bps;

/* programs/huma-vault/src/instructions/admin/recover_loss.rs */
014 | huma_config.require_protocol_on()?;
016 | vault_state.require_vault_not_closed()?;
020 | let loss_recovered = vault_state.recover_loss(amount, vault_config.huma_fee_bps)?;

/* programs/huma-vault/src/account_types/vault/state.rs */
485 | pub fn recover_loss(
486 |     &mut self,
487 |     total_loss_recovery: u64,
488 |     huma_fee_bps: u16,
489 | ) -> Result<Vec<u64>> {
497 |     self.accrue_huma_fee(huma_fee_bps)?;
```

Since `recover_loss` is permitted (and often necessary) during the `PreClosure` phase (as it only requires the vault to be "not closed"), calling it will inadvertently trigger additional fee accrual. This contradicts the system's design and documentation, potentially leading to accounting discrepancies or complications in finalizing vault closure.

Resolution

This issue has been fixed by `bb27bf8`.

HUMA VAULT

[L-02] Missing `deposit_mint` check in `submit_redemption_request`

The `submit_redemption_request` instruction accepts a `deposit_mint` account to validate the `vault_deposit_token` account (via the `associated_token` constraint). However, it fails to verify that this `deposit_mint` matches the actual deposit mint configured in the `VaultConfig`.

```
/* programs/huma-vault/src/instructions/vault_funds/submit_redemption_request.rs */
101 | pub deposit_mint: InterfaceAccount<'info, Mint>,
102 |
103 | #[account(
104 |     associated_token::mint = deposit_mint,
105 |     associated_token::authority = vault_authority,
106 |     associated_token::token_program = deposit_token_program
107 | )]
108 | pub vault_deposit_token: Box<InterfaceAccount<'info, TokenAccount>>,
```

The instruction uses `ctx.accounts.vault_deposit_token.amount` to determine the quantity of assets available for redemption. If a caller supplies an arbitrary mint and a corresponding token account (owned by the vault authority), the instruction will calculate the redemption request based on the balance of that incorrect token account.

This can lead to two scenarios:

1. **Under-redemption:** If the incorrect token account has a lower balance than the actual deposit token account, the vault will request fewer shares than necessary to cover withdrawals. This can be remediated by a subsequent correct call to the instruction.
2. **Over-redemption:** If the incorrect token account has a higher balance (e.g., if the vault holds other assets), the vault might request more shares than needed. This results in inefficient capital utilization, as excess assets would sit idle in the vault instead of earning yield in the strategy. This can be fixed by redeploying the excess assets via the `deploy` instruction.

While the impact is limited to operational inefficiency and does not directly lead to fund loss, the lack of validation deviates from the intended logic.

Resolution

This issue has been fixed by `8771740`.

HUMA VAULT

[L-03] `max_deposit_amount` configuration is not enforced in deposits

The `DepositConfig` struct defines a `max_deposit_amount` field, which is intended to limit the maximum amount of assets that can be deposited in a single transaction. This configuration is validated during updates to ensure it is not less than `min_deposit_amount`.

However, the `deposit` instruction fails to enforce this limit. While the `min_deposit_amount` is explicitly passed to the `depositor_utils::deposit` function (ensuring the minimum threshold is met), there is no corresponding check for `max_deposit_amount` against the provided `assets` amount.

```
/* programs/huma-vault/src/instructions/depositor/deposit.rs */
033 | let shares = depositor_utils::deposit(
034 |     assets,
035 |     vault_config.deposit_config.min_deposit_amount,
...
```

This omission allows users to deposit amounts exceeding the configured maximum, bypassing the vault's intended risk management or concentration control policies.

Resolution

This issue has been fixed by `ae685ce`.

HUMA VAULT

[L-04] `huma_fee` is not checked in `LockupState::ready_to_close`

The `LockupState::ready_to_close` method is used to determine if a lockup option is eligible for removal from the vault. Currently, it only verifies that `assets`, `losses`, and `supply` are zero.

```
/* programs/huma-vault/src/account_types/vault/lockup.rs */
313 | pub fn ready_to_close(&self) -> bool {
314 |       self.assets == 0 && self.losses == 0 && self.supply == 0
315 | }
```

However, the `LockupState` also tracks protocol fees in the `huma_fee` field (of type `Fee`). This struct contains `accrued`, `pending_redemption`, and `pending_withdrawal` balances.

If a `LockupState` is removed via `remove_lockup_option` while it still holds non-zero values in `huma_fee`, the record of these fees is destroyed, effectively causing a loss of claimable fees.

Resolution

This issue has been fixed by `9a5b98f`.

HUMA VAULT

[L-05] `min_deposit_amount` check bypassed in `migrate`

The `VaultState::migrate` function handles the migration of assets from a strategy into the vault. This process converts strategy tokens into underlying assets (`assets_migrated`) and adds them to a lockup option, effectively acting as a deposit.

However, unlike the standard `deposit` instruction, the `migrate` logic fails to verify that the `assets_migrated` amount meets the `min_deposit_amount` requirement configured in the `VaultConfig`.

This omission allows users to migrate dust amounts into the vault, bypassing the minimum deposit threshold intended to prevent spam or inefficient small positions.

Resolution

This issue has been fixed by `15a329b`.

HUMA VAULT

[I-01] Floating-point math in asset projection

The `LockupConfig` implementation relies on `f64` floating-point arithmetic to calculate the periodic APY and projected asset growth over time, specifically utilizing the `powf` and `powi` functions.

```
/* programs/huma-vault/src/account_types/vault/lockup.rs */
179 | fn calculate_periodic_apy(target_apy_bps: u16) -> f64 {
180 |     if target_apy_bps == 0 {
181 |         return 0.0;
182 |     }
183 |
184 |     let hundred_percent_bps = HUNDRED_PERCENT_BPS as f64;
185 |     let seconds_in_a_year = SECONDS_IN_A_YEAR as f64;
186 |     let periodic_rate_decimal =
187 |         (1.0 + target_apy_bps as f64 / hundred_percent_bps).powf(1.0 / seconds_in_a_year) - 1.0;
188 |     periodic_rate_decimal * hundred_percent_bps * seconds_in_a_year
189 | }

/* programs/huma-vault/src/account_types/vault/lockup.rs */
352 | fn calculate_projected_assets(&self, seconds_from_now: u64, yield_bps: f64) -> u64 {
353 |     let current_assets = self.assets;
354 |     if seconds_from_now == 0 || yield_bps == 0f64 {
355 |         return current_assets;
356 |     }
357 |
358 |     let yield_rate_per_second = yield_bps / (SECONDS_IN_A_YEAR * HUNDRED_PERCENT_BPS) as f64;
359 |     // Compute the updated assets by compounding once per second.
360 |     (current_assets as f64 * (1_f64 + yield_rate_per_second).powi(seconds_from_now as i32))
361 |         as u64
362 | }
```

This approach presents two main issues. First, floating-point arithmetic can be computationally expensive in the Solana SVM, particularly transcendental functions like `powf`. Heavy usage of these instructions consumes significant Compute Units (CU), potentially limiting the composability of the instruction or causing it to exceed transaction limits. Second, floating-point operations lack the strict rounding guarantees preferred for financial smart contracts, introducing potential non-determinism or precision errors compared to fixed-point arithmetic.

It is recommended to replace the current discrete compounding logic with a continuous interest rate formula ($A = P \cdot e^{rt}$). This can be implemented efficiently using a Taylor series expansion to approximate the exponential function. This approach reduces CU consumption and ensures more predictable behavior.

Resolution

This issue has been fixed by `ae685ce`. The development team decided to maintain the use of floating-point arithmetic while removing the periodic APY calculation. The `powf` function is now utilized directly within `calculate_projected_assets` after verifying that the CU consumption remains within acceptable limits.

HUMA VAULT

[I-02] `enable_vault` allows invalid state transitions

The `VaultState` struct uses the `VaultStatus` enum to manage the vault's lifecycle, defining four distinct states: `Off`, `On`, `PreClosure`, and `Closed`. The `enable_vault` instruction is designed to transition the vault to the `On` status, effectively activating it.

However, the current validation logic only checks that the vault is *not* currently `On`:

```
/* programs/huma-vault/src/account_types/vault/state.rs */
106 | if !self.is_vault_on() {
107 |     self.status = VaultStatus::On;
108 |     return Ok(true);
109 | }
```

This verification is insufficient because it permits invalid state transitions. Specifically, a vault that has entered the `PreClosure` phase (preparing for shutdown) or has reached the `Closed` state (permanently inactive) can be improperly reverted to `On` by the admin.

This violates the intended lifecycle state machine of the protocol. A `Closed` vault represents a terminal state where no further deposits or normal operations should occur. Allowing a closed vault to be re-enabled could lead to inconsistent system states and unexpected behaviors, such as users depositing into a vault that was supposed to be defunct.

To ensure the integrity of the vault's lifecycle, the `enable_vault` instruction should explicitly require the vault to be in the `Off` state before activation.

Resolution

The team clarified that this behavior is intentional, serving as an administrative mechanism to re-enable vaults in case of accidental transitions.

HUMA VAULT

[I-03] Missing validation for Token 2022 extensions in `create_vault`

The `create_vault` instruction allows initializing a vault with a `deposit_mint` that uses the Token 2022 standard (via `InterfaceAccount`). However, the implementation does not validate which extensions are enabled on this mint.

Token 2022 introduces extensions such as Transfer Fees. If the deposit mint has transfer fees enabled, the vault's internal accounting will drift from the actual token balance, as the vault typically assumes the full amount of a transfer is received (or calculates fees based on its own logic, not the token's). This leads to accounting errors.

To ensure protocol stability and correct accounting, it's recommended to explicitly restrict which extensions are supported.

Resolution

This issue has been fixed by `34ef16f`.

HUMA VAULT

[I-04] Potential arithmetic underflow panic in `process_redemption_request`

In `VaultState::process_redemption_request`, the total `redeemed_fee` is calculated by aggregating fees from all lockup states. Inside the iterator, `complete_redemption` verifies if the vault has enough assets to cover the fee for *that specific* lockup state.

However, the validation logic passes the initial `available_assets` to every call within the iterator. It does not account for the cumulative depletion of assets as fees are redeemed.

```
/* programs/huma-vault/src/account_types/vault/state.rs */
417 | let mut available_assets = self.get_available_assets(deposit_token_balance);
418 | let redeemed_fee = self
419 |       .lockup_states
420 |       .iter_mut()
421 |       .map(|s| s.huma_fee.complete_redemption(available_assets)) // Validation logic flaw
422 |       .sum::<Result<u64>>()?;
423 | available_assets -= redeemed_fee;
```

If the sum of `redeemed_fee` across multiple lockups exceeds `available_assets` (even if each individual fee is smaller than the total), the subtraction on line 423 will fail.

It's recommended to use `checked_sub` to safely handle the asset deduction and return a proper error if funds are insufficient.

Resolution

This issue has been fixed by `6db2692`.

HUMA VAULT

[I-05] Inconsistent access control documentation for `withdraw_after_vault_closure`

The documentation for the `withdraw_after_vault_closure` instruction explicitly states that "Only depositors can call this instruction." However, the implementation defines the `depositor` account as an `UncheckedAccount` without a `Signer` check.

```
/* programs/huma-vault/src/lib.rs */
577 | /// # Access Control
578 | /// Only depositors can call this instruction.
579 | pub fn withdraw_after_vault_closure(ctx: Context<WithdrawAfterVaultClosure>) -> Result<()> {

/* programs/huma-vault/src/instructions/depositor/withdraw_after_vault_closure.rs */
082 | pub depositor: UncheckedAccount<'info>,
```

This implementation allows any third party to call the instruction on behalf of a depositor. While the instruction ensures funds are returned to the correct associated token account, it permits unauthorized actors to close the depositor's position and burn their deposit receipt NFT. This contradicts the documentation and the implied permission model.

Resolution

This issue has been fixed by `11882d9`.

HUMA VAULT

[I-06] `strategy_tokens_requested` should be capped at `lender_mode_token.amount`

In the `PSTStrategy::submit_redemption_request` function, the amount of strategy tokens to request for redemption (`strategy_tokens_requested`) is calculated based on the requested vault shares, current exchange rates, and an estimated safety buffer (`redemption_buffer_bps`).

This calculated amount is then passed directly to the `add_redemption_request_v2` CPI. However, if the calculated amount (plus buffer) exceeds the actual balance of strategy tokens held by the vault (in the `lender_mode_token` account), the underlying Permissionless Pool program will revert the transaction with an `InsufficientSharesForRequest` error.

```
/* programs/permissionless/src/mode/instructions/add_redemption_request_v2.rs */
024 | let total_shares = ctx.accounts.lender_mode_token.amount as u128;
025 | require!(total_shares >= shares, Error::InsufficientSharesForRequest);
```

It's recommended to retrieve the balance of the `lender_mode_token` account and cap `strategy_tokens_requested` to this balance before making the CPI call.

Resolution

The team acknowledged the issue but decided against implementing the cap due to the added complexity of recalculating the maximum submittable vault shares. Since the underlying permissionless program already enforces the balance check and reverts with a clear error, the current behavior is considered correct and safe. This improvement is left as a future optimization.

HUMA VAULT

[I-07] Arbitrage risk in `recover_loss`

The `recover_loss` mechanism allows the protocol to inject funds to cover previously declared losses, effectively increasing the share price of the affected lockup option. However, this creates a potential arbitrage opportunity where users can deposit funds after a loss is declared (at a depressed share price) and profit immediately upon recovery (diluting the recovery for existing users).

Resolution

The team clarified that the behavior is intended. While depositors post-loss may profit from recovery, they also assume greater risk as recovery is not guaranteed. In a loss scenario, withdrawals are more likely than deposits, making new deposits inherently riskier.

Issues Also Identified by the Huma Team

HUMA VAULT

[01] Denial of Service via merging of escrowed Deposit Receipts

This issue was also identified by the Huma team during the audit.

The `request_redemption` instruction correctly transfers the Deposit Receipt NFT to the `vault_authority` (escrow) when a full redemption is requested. This is intended to ensure the NFT is available to be burned during the subsequent `process_redemption_request`.

```
/* programs/huma-vault/src/instructions/depositor/request_redemption.rs */
041 | if all_shares_redeemed {
042 |     // If all shares are being redeemed, then transfer the deposit receipt NFT to the vault authority so that
043 |     // we can burn it later.
044 |     depositor_utils::transfer_deposit_receipt(
045 |         &collection.to_account_info(),
046 |         &deposit_receipt.to_account_info(),
047 |         payer.as_ref(),
048 |         depositor.as_ref(),
049 |         ctx.accounts.vault_authority.as_ref(),
050 |         ctx.accounts.mpl_core_program.as_ref(),
051 |     )?;
052 | }
```

However, the `merge` instruction is permissionless regarding the `depositor` account. This allows any user to call `merge` passing the `vault_authority` PDA as the `depositor`.

Because the `vault_authority` now owns the escrowed NFT, the `merge` instruction validates ownership successfully. Crucially, `merge` invokes `burn_deposit_receipt` to burn the NFT.

This effectively allows an attacker to burn the escrowed NFT and merge it into a new NFT (also owned by the vault authority). When the administrator later attempts to execute `process_redemption_request`, the instruction tries to burn the *original* NFT (referenced by `deposit_receipt_to_burn` in the request). Since that NFT has already been burned by the malicious merge, the transaction fails.

This results in a Denial of Service for the redemption queue.

It's recommended to add a check in the `merge` instruction to explicitly forbid the `depositor` account from being the `vault_authority`.

Resolution

This issue has been fixed by `dc11021`.

HUMA VAULT

[02] Logic flaws in `submit_redemption_request`

This issue was also identified by the Huma team during the audit.

The `submit_redemption_request` function in `VaultState` contains logic flaws that affect the efficiency of fee collection and the accuracy of redemption requests sent to the strategy protocol.

1. Fee Redemption Blocked by Lack of User Requests The function calculates `vault_shares_submitted` (user shares needing redemption) and returns early if this value is 0.

```
/* programs/huma-vault/src/account_types/vault/state.rs */
370 | let available_assets = self.get_available_assets(deposit_token_balance);
371 | let lockup_state = &self.lockup_states[lockup_index];
372 | let shares_processable = lockup_state.convert_to_shares(available_assets);
373 | let vault_shares_submitted = self
374 |     .redemption_queue
375 |     .submit_redemption_request(shares_processable);
376 | if vault_shares_submitted == 0 {
377 |     return Ok(0);
378 | }
379 |
380 | let vault_strategy: &dyn VaultStrategyInterface = vault_strategy_type.get_strategy();
381 | let total_fee_to_redeem: u64 = self
382 |     .lockup_states
383 |     .iter_mut()
384 |     .map(|lockup_state| lockup_state.huma_fee.submit_for_redemption())
385 |     .sum();
```

This check occurs *before* protocol fees (`huma_fee`) are aggregated and submitted. Consequently, if there are no outstanding user redemption requests (or if available assets are sufficient to cover them), the protocol cannot redeem its accrued fees from the strategy. This delays fee collection until new user demands arise.

2. Underestimation of Required Liquidity

- The calculation for `shares_processable` (shares coverable by current vault assets) is simplistic, it does not account for the **redemption processing delay**. By the time assets are deployed or redeemed, the share price will likely have increased, meaning the current assets will cover fewer shares than estimated.
- The calculation for `vault_shares_submitted` does not account for shares pending processing when subtracting shares processable.

```
/* programs/huma-vault/src/account_types/vault/redemption.rs */
098 | pub fn submit_redemption_request(&mut self, shares_processable: u64) -> u64 {
099 |     let vault_shares_to_submit = self
100 |         .shares_pending_submission
101 |         .saturating_sub(shares_processable);
102 |     self.shares_pending_submission -= vault_shares_to_submit;
103 |     self.shares_pending_processing += vault_shares_to_submit;
104 |
105 |     vault_shares_to_submit
106 | }
```

Resolution

This issue has been fixed by `302cd62` and `645e178`.

Appendix: Methodology and Scope of Work

Assisted by the Sec3 Scanner developed in-house, the manual audit particularly focused on the following work items:

- Check common security issues.
- Check program logic implementation against available design specifications.
- Check poor coding practices and unsafe behavior.
- The soundness of the economics design and algorithm is out of scope of this work

DISCLAIMER

The instance report ("Report") was prepared pursuant to an agreement between Coderrect Inc. d/b/a Sec3 (the "Company") and Huma Technologies LLC (the "Client"). This Report solely includes the results of a technical assessment of a specific build and/or version of the Client's code specified in the Report ("Assessed Code") by the Company. The sole purpose of the Report is to provide the Client with the results of the technical assessment of the Assessed Code. The Report does not apply to any other version and/or build of the Assessed Code. Regardless of the contents of the Report, the Report does not (and should not be interpreted to) provide any warranty, representation or covenant that the Assessed Code: (i) is error and/or bug free, (ii) has no security vulnerabilities, and/or (iii) does not infringe any third-party rights. Moreover, the Report is not, and should not be considered, an endorsement by the Company of the Assessed Code and/or of the Client. Finally, the Report should not be considered investment advice or a recommendation to invest in the Assessed Code and/or the Client.

This Report is considered null and void if the Report (or any portion thereof) is altered in any manner.

ABOUT

The Sec3 audit team comprises a group of computer science professors, researchers, and industry veterans with extensive experience in smart contract security, program analysis, testing, and formal verification. We are also building automated security tools that incorporate static analysis, penetration testing, and formal verification.

At Sec3, we identify and eliminate security vulnerabilities through the most rigorous process and aided by the most advanced analysis tools.

For more information, check out our website and follow us on twitter.