Frank Castle Audits

Wick Protocol Audit

Mar 2026

Frank Castle Audits

Table of Contents

About Frank Castle Audits

Frank Castle is a professional smart contract security researcher specializing in **Rust-based smart contracts and decentralized infrastructure**, with a strong focus on the **Solana ecosystem**.

He has conducted security reviews for a wide range of high-profile and multi-million-dollar protocols, including
Lido, GMX, Pump.fun, LayerZero, Synthetix, Hydration, Perena, ORO, DUB Social, and others.

Frank Castle has successfully completed **over 50 Solana security audits**, building a proven track record of rigor, depth, and reliability in smart contract security. He is trusted by leading audit firms and security organizations, including **Pashov Audit Group, Spearbit, Cantina, Shieldify, Zenith**, and **BailSec**.

His comprehensive experience and hands-on expertise in Rust-based ecosystems reflect a strong commitment to advancing blockchain security and promoting industry best practices.

Portfolio: https://github.com/Frankcastleauditor/public-audits

X (Twitter): https://x.com/0xcastle_chain

Telegram: https://t.me/castle_chain

Security Review

Project: YieldConverter (yield-swap)

Commit: 55aa4a49d50ffc165b89e0a015d6d2708003072f

Date: March 2026

Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

This audit report is prepared by **Frank Castle Audits** and is provided on an "as is" basis. The findings and recommendations contained in this report are based on the reviewer's analysis of the codebase at the time of review.

Frank Castle Audits shall not be held liable for any losses or damages arising from the use of this report or reliance on its findings.

Risk Classification

Findings are classified according to their **impact** and **likelihood**, following an industry-standard severity model.

Severity	Description
Critical	Issues that can lead to direct loss of funds, permanent protocol failure, or complete compromise of core functionality.
High	Issues that significantly affect protocol security or availability and may lead to loss of funds or major disruption under certain conditions.
Medium	Issues that can cause incorrect behavior, state corruption, or DoS scenarios but require specific conditions or privileges.
Low	Minor issues that do not pose an immediate security risk but may affect correctness, efficiency, or best practices.
Informational	Observations and recommendations that improve code quality, maintainability, or design clarity.

About YieldConverter

YieldConverter is a yield conversion protocol on **Solana**. Users deposit Liquid Staking Tokens (LST) into per-user position vaults. An off-chain cranker operator periodically harvests accrued staking yield from each position, executes an AMM swap to convert yield into the user's target token, and deposits the proceeds into the target vault. The protocol supports configurable fee tiers, a per-call harvest rate cap, and a 24-hour timelock on critical admin operations such as stake pool updates.

Executive Summary

Severity	Count	Status
Critical	0	-
High	0	-
Medium	4	Fixed
Low	2	Fixed
Informational	8	Fixed

All findings have been fixed by the YieldConverter development team and reviewed by Frank Castle.

Scope

- **YieldConverter:** `programs/yield-swap`

Commit: `55aa4a49d50ffc165b89e0a015d6d2708003072f`

Findings

Medium Severity

[M-01] Inflated Exchange Rate Generates Protocol Fees from User Principal

Severity: Medium

Fixed

Description

In crank_harvest.rs:95-199 and utils.rs:50-77, the cranker can supply an exchange rate up to 50 bps above the on-chain rate (enforced by `validate_exchange_rate`). This inflated rate increases the calculated `yield_lst`, which in turn increases the protocol fee since the fee is taken as a percentage of yield. The additional yield and fee do not come from real staking rewards but from the user's deposited principal.

If the cranker repeats this pattern each epoch, the protocol continuously collects fees on inflated yield. The per-call harvest cap limits the amount per call but does not prevent this pattern since the inflated rate still falls within the tolerance band.

This creates a misaligned incentive: the protocol benefits from the cranker inflating rates, as both parties extract value from user principal. If the admin and cranker are the same entity or are colluding, they are incentivized to consistently inflate within the tolerance band.

Additionally, the inflated rate advances `last_harvested_rate` beyond the real on-chain rate, which front-loads yield from future epochs and reduces the user's future harvestable yield. Users who deposit with an inflated rate also set a higher checkpoint, reducing their own future yield.

Exploit scenario:

1. User deposits 10,000 LST at rate `1.000000000`
2. On-chain rate remains at `1.000000000` (no epoch boundary)
3. Cranker calls `crank_harvest` with rate `1.004999999` (+49 bps)
   - `rate_diff = 0.004999999`
   - `yield_lst = 10,000 * 0.004999999 / 1.004999999 ≈ 49.75 LST`
   - `fee_lst = 49.75 * 500 / 10,000 ≈ 2.49 LST`

- cranker receives `47.26 LST`
- `deposited = 9,950.25 LST`

4. User withdraws immediately — user receives `9,950.25 LST` and target token worth of `49.75 LST`; protocol collected `2.49 LST` in fees on phantom yield

If the user stays deposited, the on-chain rate may eventually catch up to the inflated `last_harvested_rate`, at which point no additional loss occurs. The impact is realized when the user withdraws before the rate catches up.

Impact

The cranker can systematically inflate yield calculations within the tolerance band, causing the protocol to collect fees from user principal rather than real staking rewards. Over time, users' deposited LST is silently eroded via fee extraction on phantom yield, with the worst case realized at withdrawal.

Recommendation

Use the on-chain rate directly for yield calculations instead of the caller-provided rate. The on-chain rate is already read via `read_onchain_rate` and is available in both `crank_harvest` and `deposit`. This eliminates the tolerance window entirely and removes the incentive for rate inflation.

[M-02] Admin Can Swap Stake Pool to Corrupt Yield Accounting Across All Positions

Severity: Medium

Fixed

Description

In update_stake_pool.rs:42-68 and initialize.rs:45-69, the admin can use `update_stake_pool` to switch the protocol's stake pool reference to an unrelated pool. Each SPL Stake Pool has a unique pool mint, so a different pool produces exchange rates with no relationship to the protocol's LST or existing positions' `last_harvested_rate` checkpoints.

If the new pool's rate is higher than the current checkpoints, phantom yield appears across all positions. The cranker can then harvest this phantom yield from every position, extracting LST from user vaults. The protocol also collects fees on this phantom yield. If the new pool's rate is lower, all harvests stop because the rate decrease check in `crank_harvest` fails.

Neither `update_stake_pool` nor `initialize` validate that the stake pool's `pool_mint` matches `protocol_state.lst_mint`. The 24h timelock on `update_stake_pool` gives users a window to exit, but most users are unlikely to monitor stake pool change proposals.

Exploit scenario:

1. Multiple users have deposited a total of 100,000 LST; `last_harvested_rate` ≈ `1.05`
2. Admin proposes a new stake pool whose rate is `2.00` (unrelated pool)
3. 24h timelock expires; admin accepts
4. Cranker harvests each position:
   - `rate_diff = 2.00 - 1.05 = 0.95`
   - `yield_lst per position ≈ deposited * 0.95 / 2.00 ≈ 47.5%` of principal
   - protocol collects fees on all harvested yield
5. Users' vaults are drained of nearly half their principal

Impact

A malicious or compromised admin can drain nearly all deposited principal across every user position in a single epoch by pointing the protocol at a stake pool with a higher exchange rate. This is a critical trust assumption: the entire protocol's accounting depends on the stake pool reference remaining correct.

Recommendation

Remove `update_stake_pool`, `accept_stake_pool`, and `cancel_propose_stake_pool` entirely. Each SPL Stake Pool has a unique pool mint, so there is no valid stake pool to switch to that would preserve correct yield accounting. The stake pool is set once at `initialize` and should be immutable. Additionally, validate that `lst_mint` matches the stake pool's `pool_mint` at initialization:

```
let stake_pool = StakePool::try_from_slice(&ctx.accounts.stake_pool.try_borrow_dat
a()?)?;
require!(
    ctx.accounts.lst_mint.key() == stake_pool.pool_mint,
    YieldSwapError::MintMismatch
);
```

[M-03] Position Vault Can Be Griefed to Prevent Closing

Severity: Medium

Fixed

Category: Denial of Service

Location: close_position_vault.rs:37, withdraw.rs:116-130

Description

`close_position_vault` calls the token program's `close_account`, which requires the vault balance to be zero. The `withdraw` instruction transfers `user_position.lst_deposited` from the vault, not the vault's actual token balance. An attacker can send 1 LST lamport directly to any user's position vault after withdrawal but before the user calls `close_position_vault`. Since the vault now has a non-zero balance, `close_account` fails and the user cannot close the vault or recover rent (~0.002 SOL).

The cost to the attacker is 1 LST lamport per griefed position. The impact per user is the loss of rent (~0.002 SOL), which is minor but unrecoverable. The attack can be repeated indefinitely.

Impact

Any attacker can permanently prevent any user from closing their position vault and recovering rent at negligible cost. This is a persistent denial-of-service where users' rent lamports remain locked in position vaults after a full withdrawal.

Recommendation

In `close_position_vault`, check the vault balance and transfer any remaining tokens to the user before calling `close_account`:

```
let remaining_balance = ctx.accounts.position_vault.amount;
if remaining_balance > 0 {
    token::transfer(
        CpiContext::new_with_signer(/* ... */),
        remaining_balance,
    )?;
}
close_account(/* ... */)?;
```

Additionally, consider updating `withdraw` to transfer `position_vault.amount` instead of `user_position.lst_deposited` to drain the vault completely and eliminate the griefing window

at source.

[M-04] `MIN_DEPOSIT` Too Low Causes Fee Rounding to Zero and Makes Crank Operations Unprofitable

Severity: Medium

Fixed

Description

Two related invariants break at the current `MIN_DEPOSIT = 10,000` lamports: the fee calculation silently rounds to zero for small positions, and the protocol loses money on every crank cycle for those same positions.

Fee Rounding to Zero

The fee calculation in `crank_harvest` uses integer division that truncates toward zero:

```
let fee_lst = u64::try_from(
    (yield_lst as u128)
        .checked_mul(fee_bps)
        .ok_or(YieldSwapError::MathOverflow)?
        .checked_div(10_000)
        .ok_or(YieldSwapError::MathOverflow)?,
)
.map_err(|_| error!(YieldSwapError::MathOverflow))?;
```

`fee_lst = yield_lst * fee_bps / 10,000`

For `fee_lst` to round to 0: `yield_lst * fee_bps < 10,000`. At the maximum harvest cap (`MAX_HARVEST_RATE_BPS = 25`), substituting `yield_lst = lst_deposited * 25 / 10,000` gives the zero-fee threshold per fee tier:

`fee_bps`	Fee %	Zero-Fee Threshold	Affected Range (with `MIN_DEPOSIT = 10,000`)
100	1%	< 40,000 lamports	10,000 – 39,999 pay zero fee
200	2%	< 20,000 lamports	10,000 – 19,999 pay zero fee
300	3%	< 13,333 lamports	10,000 – 13,332 pay zero fee
≥ 400	≥ 4%	< 10,000 lamports	No valid positions affected

At `fee_bps < 400`, all positions between `MIN_DEPOSIT` and the zero-fee threshold receive full yield while paying zero protocol fee.

Crank Operations Are Unprofitable

Even above the rounding threshold, `MIN_DEPOSIT` is too low for the protocol to recover its crank gas costs. The per-position overhead consists of two individually-required instructions — `crank_harvest` (~10,000 lamports) and `deposit_yield` (~5,000 lamports) — totaling ~15,000 lamports per position. The AMM swap is batched across positions and excluded from this calculation.

Applying a 50% safety factor on effective yield and solving for break-even (`fee ≥ 15,000 lamports`):

```
lst_deposited >= 120,000,000,000 / fee_bps
```

`fee_bps`	Fee %	Min Deposit for Break-Even
100	1%	~1.2 SOL
200	2%	~0.6 SOL
500	5%	~0.24 SOL
1000	10%	~0.12 SOL

The current `MIN_DEPOSIT = 10,000` lamports is **~120,000x below** the break-even threshold at 1% fee. A concrete example at `fee_bps = 100`, `deposit = 1 SOL`:

- Per-position gas spent: **15,000 lamports**
- Effective yield at 50% harvest cap: `1,000,000,000 * 0.00125 = 1,250,000` lamports
- Fee collected: `1,250,000 * 100 / 10,000 = 12,500` lamports
- **Net: −2,500 lamports per harvest cycle**

At `MIN_DEPOSIT = 10,000` lamports specifically, the rounding issue compounds this: effective yield is 12 lamports, fee rounds to zero, and the cranker spends 15,000 lamports to earn the protocol nothing.

Impact

Positions below the zero-fee threshold receive full yield with no protocol fee collected — the fee invariant is silently violated. Crank operations are net-negative for the protocol across a wide range of realistic deposit sizes and fee tiers. Both effects scale linearly with the number of small positions; at 10,000 such positions the protocol bleeds ~0.15 SOL per harvest cycle in

unrecoverable gas while collecting zero fees. No funds are directly at risk, but sustained operation under these conditions erodes protocol solvency.

Recommendation

Set `MIN_DEPOSIT` to a value that simultaneously satisfies both constraints: it must be high enough that all valid positions always produce a non-zero fee at the lowest expected `fee_bps`, and high enough that the protocol breaks even on crank gas costs.

Low Severity

[L-01] No Validation That `lst_mint` Matches Stake Pool's `pool_mint` in Initialize

Severity: Low

Fixed

Location: initialize.rs:45-69

Description

The `initialize` instruction accepts `lst_mint` and `stake_pool` as separate accounts but does not verify that `lst_mint` matches the stake pool's `pool_mint` field. If the admin passes a stake pool that corresponds to a different LST, all subsequent yield calculations would use an exchange rate from an unrelated pool.

This is not currently exploitable as `initialize` is a one-time admin-only call, but the validation would prevent misconfiguration at deployment time.

Impact

An incorrect stake pool configuration at initialization produces exchange rates with no relationship to the deposited LST, silently corrupting all yield calculations from the first epoch onward. This is a deployment-time footgun with no on-chain recovery path.

Recommendation

Validate that `lst_mint` matches the stake pool's `pool_mint` during initialization. Using the SDK:

```
let stake_pool = StakePool::try_from_slice(&ctx.accounts.stake_pool.try_borrow_dat
a()?)?;
require!(
    ctx.accounts.lst_mint.key() == stake_pool.pool_mint,
    YieldSwapError::MintMismatch
);
```

The same check should be applied in `update_stake_pool` to ensure the new stake pool still corresponds to the protocol's `lst_mint`.

[L-02] No Mechanism to Deprecate or Disable Target Mints Once Enabled

Severity: Low

Fixed

Location: deposit.rs:71-76

Description

Once a target vault PDA is created, it permanently enables deposits for that mint. The `deposit` instruction's only gate is whether a `target_vault` PDA exists:

```
#[account(
    seeds = [TARGET_VAULT_SEED, target_mint.key().as_ref()],
    bump,
    constraint = target_vault.mint == target_mint.key() @ YieldSwapError::NoTarget
Vault,
)]
pub target_vault: Box<InterfaceAccount<'info, TokenAccount>>,
```

There is no `active` or `deprecated` status checked anywhere. The `ProtocolState` struct has no field tracking which target mints are active. The only protocol-wide control is the `paused` flag, which blocks all deposits across all target mints.

This means:
- Admin **cannot** selectively stop new deposits for a specific target mint
- Admin **cannot** close a target vault PDA
- If a token is sanctioned, delisted, or becomes problematic, the admin's only option is to pause the entire protocol

Impact

No granular access control exists over target mints. The admin cannot selectively disable deposits for specific target mints without pausing the entire protocol. If a jurisdiction requires the protocol to stop facilitating conversion to a specific token, there is no on-chain mechanism to enforce this. The cranker must manually skip deprecated mints off-chain, and frontends must maintain their own blacklists with no on-chain source of truth.

Recommendation

Add a `TargetMintConfig` account per target mint to track active status:

```
#[account]
#[derive(InitSpace)]
pub struct TargetMintConfig {
    pub mint: Pubkey,
    pub active: bool,
    pub bump: u8,
}
```

Derive it as a PDA seeded by the target mint key. Initialize it in `init_target_vault`. Add a `deprecate_target_mint` admin instruction that sets `active = false`. Check active status in `deposit`:

```
#[account(
    seeds = [b"target_config", target_mint.key().as_ref()],
    bump = target_mint_config.bump,
    constraint = target_mint_config.active @ YieldSwapError::TargetMintDeprecated,
)]
pub target_mint_config: Account<'info, TargetMintConfig>,
```

This gives the admin granular control to disable new deposits per mint while existing positions can still be harvested, claimed, and withdrawn normally.

Informational

[I-01] Missing Type Discriminator Check on Stake Pool Account

Severity: Informational

Fixed

Location: utils.rs:8-43, initialize.rs:49-57

Description

`read_onchain_rate` validates that the stake pool account is owned by the SPL Stake Pool program but does not verify the account's type discriminator. The SPL Stake Pool program owns multiple account types (`StakePool`, `ValidatorList`, etc.). Any of these would pass the owner check.

The function reads raw bytes at hardcoded offsets (258, 266) and interprets them as `total_lamports` and `pool_token_supply`. If a non-`StakePool` account is passed, these offsets contain unrelated data, producing an arbitrary exchange rate.

The same issue exists in `initialize`, which validates the stake pool's owner and data length but not its account type before storing it in protocol state.

This is not currently exploitable as the stake pool account is set exclusively by the admin via `initialize` and `update_stake_pool` (which has a 24h timelock).

Recommendation

Add a type check using the `account_type` field at byte offset 0 to reject non-`StakePool` accounts. Consider using the `spl-stake-pool` SDK for easier deserialization and to avoid reliance on hardcoded offsets:

```
# Cargo.toml
spl-stake-pool = { version = "4", features = ["no-entrypoint"] }
```

```
use spl_stake_pool::state::{StakePool, AccountType};

let stake_pool = StakePool::try_from_slice(&stake_pool_info.try_borrow_data()?)?;
require!(
    stake_pool.account_type == AccountType::StakePool,
    YieldSwapError::InvalidStakePool
);
let rate = (stake_pool.total_lamports as u128)
    .checked_mul(RATE_PRECISION as u128)
    .ok_or(YieldSwapError::MathOverflow)?
    .checked_div(stake_pool.pool_token_supply as u128)
    .ok_or(YieldSwapError::MathOverflow)?;
```

[I-02] Store Raw Rate Components Instead of Precomputed Exchange Rate

Severity: Informational

Fixed

Location: utils.rs, state.rs:48 (`last_harvested_rate`)

Description

The protocol reads `total_lamports` and `pool_token_supply` from the stake pool account, then immediately collapses them into a single exchange rate using `RATE_PRECISION` (1e9). This introduces rounding error at the point of storage, which compounds when two imprecise rates are subtracted during yield calculation in `crank_harvest`.

Recommendation

Store both `total_lamports` and `pool_token_supply` in `UserPosition` instead of a precomputed `last_harvested_rate`. Defer the division to the point of use, dividing only once at the end of the yield formula:

```
yield = deposited * (curr_lamports * last_supply - last_lamports * curr_supply)
        / (curr_lamports * last_supply)
```

This preserves full precision, eliminates the `RATE_PRECISION` constant, and follows the same pattern used by the SPL Stake Pool program itself, which stores the raw components and computes rates on the fly.

[I-03] Harvest Window Cap Limits Protocol Scalability

Severity: Informational

Fixed

Location: constants.rs:72

Description

`MAX_HARVEST_PER_WINDOW_CAP` is hardcoded at 50 LST. As total deposited LST grows, the cranker cannot clear each epoch's yield within the available harvest windows and falls permanently behind. The admin can configure `max_harvest_per_window` up to this ceiling via `update_harvest_limit`, but the ceiling itself requires a program redeployment to change.

From `crank_harvest`, yield per epoch across all positions:

```
yield_lst = total_deposited * rate_diff / current_rate
```

Since `rate_diff / current_rate ≈ per_epoch_rate_increase ≈ 0.00048`:

```
yield_lst ≈ total_deposited * 0.00048
```

Total Deposited	Epoch Yield	Windows Needed	Time to Clear
100K LST	~48 LST	1	2 hours
500K LST	~240 LST	5	10 hours
1M LST	~480 LST	10	20 hours
2.5M LST	~1,200 LST	24	48 hours
5M LST	~2,400 LST	48	96 hours

At roughly 3M LST deposited, the cranker can no longer clear yield within one epoch (~60 hours) and falls permanently behind.

Recommendation

Estimate the maximum lSOL the protocol is expected to handle and set `MAX_HARVEST_PER_WINDOW_CAP` based on the resulting per-epoch yield at that scale. The cap

should allow the cranker to clear a full epoch's yield within the epoch duration.

[I-04] Per-User Harvest Cap Is on Yield Amount Rather Than Rate Increase

Severity: Informational

Fixed

Location: crank_harvest.rs:147-160

Description

The per-user harvest cap limits `yield_lst` to `MAX_HARVEST_RATE_BPS` (25 bps = 0.25%) of `lst_deposited` per call. Because the cap is applied to an absolute LST amount derived from the deposit size, its effectiveness varies with position size. A position with 10 LST and one with 10,000 LST are both capped at 0.25%, but the rate advancement allowed is identical in both cases since the yield formula `yield_lst = deposited * rate_diff / current_rate` scales linearly with deposit size.

The cap is generous relative to expected yield. At ~7% APY with ~146 epochs per year, per-epoch rate increase is approximately 4.8 bps. The 25 bps cap allows roughly 5 epochs worth of yield in a single call, meaning the cap rarely binds under normal operation and provides limited protection against a cranker inflating the exchange rate within the 50 bps tolerance.

A cap on `rate_diff` itself (the exchange rate increase per call) would be more direct. It bounds the cranker's ability to advance the rate checkpoint regardless of position size and maps directly to the expected per-epoch yield.

Recommendation

Consider replacing the yield-amount cap with a cap on the rate increase:

```
let max_rate_diff = checked_mul_div(
    position.last_harvested_rate,
    MAX_RATE_INCREASE_BPS,
    10_000
)?;
require!(rate_diff <= max_rate_diff, YieldSwapError::HarvestCapExceeded);
```

Setting `MAX_RATE_INCREASE_BPS` to approximately 10 bps would allow a full epoch's yield (~4.8 bps) with margin, while tightly bounding the cranker's ability to inflate yield through rate tolerance.

[I-05] Unnecessary `reload()` Calls After CPI

Severity: Informational

Fixed

Description

Several `reload()` calls are performed after CPI transfers where the reloaded account is **never used again**, wasting ~2,000 CU per call:

File	Line	Account Reloaded	Used After?
`withdraw.rs`	133	`position_vault`	No — next transfer uses `target_vault`
`withdraw.rs`	154	`target_vault`	No — function ends
`claim.rs`	91	`target_vault`	No — only `user_position` accessed after

The `crank_harvest.rs:225` reload of `position_vault` **is necessary** because the vault is used in a subsequent fee transfer.

Recommendation

Remove the three unnecessary reloads to save CU:

- withdraw.rs:133 — remove `position_vault.reload()`
- withdraw.rs:154 — remove `target_vault.reload()`
- claim.rs:91 — remove `target_vault.reload()`

[I-06] Add `checked_mul_div` Utility Function

Severity: Informational

Fixed

Description

The pattern of casting two `u64` values to `u128`, multiplying, dividing, and casting back appears 7 times across utils.rs, crank_harvest.rs, and deposit.rs. This repetition increases verbosity and makes the arithmetic intent less clear at each call site.

Recommendation

Add a shared `checked_mul_div(a: u64, b: u64, divisor: u64) -> Result<u64>` utility function to reduce boilerplate:

```
/// (a * b) / divisor
pub fn checked_mul_div(a: u64, b: u64, divisor: u64) -> Result<u64> {
    u64::try_from(
        (a as u128)
            .checked_mul(b as u128)
            .ok_or(YieldSwapError::MathOverflow)?
            .checked_div(divisor as u128)
            .ok_or(YieldSwapError::MathOverflow)?,
    )
    .map_err(|_| error!(YieldSwapError::MathOverflow))
}

// Before:
let rate = u64::try_from(
    (total_lamports as u128)
        .checked_mul(RATE_PRECISION as u128)
        .ok_or(YieldSwapError::MathOverflow)?
        .checked_div(pool_token_supply as u128)
        .ok_or(YieldSwapError::MathOverflow)?,
)
.map_err(|_| error!(YieldSwapError::MathOverflow))?;

// After:
let rate = checked_mul_div(total_lamports, RATE_PRECISION, pool_token_supply)?;
```

[I-07] Consolidate `init_fee_vault` and `init_harvest_tracker` into `initialize`

Severity: Informational

Fixed

Description

`init_fee_vault` and `init_harvest_tracker` are admin-created singletons required before the protocol is operational. They are currently separate instructions, creating a multi-step deployment sequence and a window where the protocol could accept deposits before these accounts exist.

Recommendation

Merge `init_fee_vault` and `init_harvest_tracker` into `initialize`. This removes two instruction handlers and their account structs, reduces deployment steps, and eliminates any window where the protocol could be partially initialized and exploitable.

[I-08] Consider Time-Bounded Global Pause That Includes Withdrawals

Severity: Informational

Fixed

Description

Currently the `paused` flag only blocks deposits and harvests. Withdrawals, partial withdrawals, and claims are always allowed. If a bug exists in the withdrawal flow, the protocol has no way to halt it. Emergency response capability is therefore limited to deposit-side exploits only.

Recommendation

Consider a global pause that also covers withdrawals, with an automatic expiry (e.g., 24–48 hours after which withdrawals auto-resume). This would give the admin time for incident response without introducing permanent fund lockup risk. The automatic expiry prevents the pause from becoming a censorship vector while still providing a meaningful emergency window.

Contact

For questions, clarifications, or remediation review, please contact **Frank Castle Audits**.