Frank Castle Audits

Wick Protocol Audit-v2

April 2026

wick_final_report

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) — V2 Yield Donation Feature

Commit: 636941f968c62653c713e043b4768b7d5da3f6a3

Date: May 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 and designate a target token; an off-chain cranker periodically harvests accrued staking yield, converts it via AMM swap, and deposits the proceeds into the user's target vault. The V2 yield donation feature introduces a `yield_recipient` field on `UserPosition`, allowing users to redirect their harvested yield to a designated third-party address.

Executive Summary

Severity	Count	Status
Critical	0	-
High	0	-
Medium	5	Open
Low	1	Open
Informational	4	Open

Scope

- **YieldConverter (V2 yield-donation feature):** `programs/yield-swap`

Commit: `636941f968c62653c713e043b4768b7d5da3f6a3`

Findings

Medium Severity

[M-01] Frozen / Non-Existent Recipient ATA Permanently Locks User Principal

Severity: Medium

Description

In withdraw.rs:85-99, claim.rs:58-70, and set_yield_recipient.rs:41-48, once `yield_recipient` is set it is immutable — any subsequent call to `set_yield_recipient` reverts with `YieldRecipientAlreadySet`. Both `claim` and `withdraw` require `user_target_account.owner == yield_recipient` whenever a recipient is configured. Critically, `withdraw` transfers `claimable_amount` to the recipient ATA atomically with the LST principal return inside a single handler, meaning any failure in the claimable transfer reverts the entire transaction — including the return of the user's LST principal.

Any of the following conditions permanently traps user funds:

1. The recipient's ATA is frozen (Token-2022 freeze extension, or the recipient is the freeze authority on a vanity mint).
2. The recipient's ATA does not exist and the recipient refuses to create it.
3. The recipient has lost their keys or the address is compromised.
4. The target mint has a transfer hook that reverts for the recipient.

Because there is no `revoke_yield_recipient` instruction, `partial_withdraw` allows the user to drain down to `MIN_DEPOSIT` but not below, so at minimum `MIN_DEPOSIT` LST plus the full `claimable_amount` are permanently irrecoverable.

Exploit scenario:

1. User sets `yield_recipient` to a friend's address.
2. Friend's ATA for the target token is subsequently frozen by the mint authority.
3. User calls `withdraw` — the `claimable_amount` transfer to the frozen ATA fails.
4. The entire transaction reverts; the user cannot retrieve their LST principal.
5. No on-chain path exists to revoke the recipient or decouple the withdrawal.

Impact

A user who sets a `yield_recipient` whose ATA becomes frozen, non-existent, or transfer-hook-gated permanently loses access to their LST principal. The coupling of principal return and claimable transfer inside `withdraw` means a single token transfer failure causes total position lockup with no recovery path.

Recommendation

Apply three independent fixes:

(a) Add an owner-only `revoke_yield_recipient` instruction that clears the field, or allow overwriting the recipient in `set_yield_recipient`.

(b) Decouple the LST principal return from the claimable transfer inside `withdraw` so the principal is always returned to the user regardless of whether the claimable transfer succeeds. Leave `claimable_amount` in place if the transfer fails; let the user claim separately.

(c) On `withdraw`, route any `claimable_amount` to the owner's ATA unconditionally — a full exit should not be treated as a donation.

```
// In withdraw.rs — decouple claimable transfer
let principal_transfer_ok = token::transfer(/* LST principal to owner */);

if position.claimable_amount > 0 {
    let claim_result = token::transfer(/* claimable to yield_recipient or
owner */);
    if claim_result.is_err() {
        // Leave claimable_amount in position for a separate claim
instruction
        msg!("Claimable transfer failed; principal returned, yield left for
claim.");
    }
}
```

[M-02] Retroactive Redirection of Pre-Existing Claimable Amount

Severity: Medium

Description

In set_yield_recipient.rs:31-55, the `set_yield_recipient` handler places no requirement on `position.claimable_amount`. A user (or a malicious phishing transaction) that calls this instruction immediately redirects all previously earned yield — including yield that accrued months before the donation decision — to the newly set recipient.

Because `yield_recipient` is immutable after being set and there is no revocation mechanism, a social-engineering attack can redirect an arbitrarily large historical `claimable_amount` in a single signed transaction. Combined with the recipient's ability to race-call `claim` before the user notices, this enables full theft of the user's historical claimable balance. The emitted `YieldRecipientSet` event does not snapshot `claimable_amount`, so front-ends cannot surface a warning to the user about the size of the impending redirection.

Exploit scenario:

1. User has accumulated 500 USDC in `claimable_amount` over six months.
2. Attacker crafts a phishing transaction that calls `set_yield_recipient` with the attacker's address.
3. User signs, believing they are only redirecting future yield.
4. Attacker immediately calls `claim` through the recipient — all 500 USDC is transferred to the attacker's ATA.
5. User has no recourse: the recipient is locked in and `claimable_amount` is now zero.

Impact

Any `claimable_amount` accumulated prior to setting a recipient is immediately and irrevocably redirected to that recipient upon the next `claim` call. A phishing attack or user misunderstanding can drain the entire historical yield balance in a single transaction, with no on-chain warning, no revocation, and no recovery path.

Recommendation

Enforce one of the following mitigations in `set_yield_recipient`:

(a) Require `claimable_amount == 0` before setting a recipient, forcing the user to claim their existing yield to their own wallet first:

```
require!(
    position.claimable_amount == 0,
    YieldSwapError::ClaimBeforeSettingRecipient
);
```

(b) Auto-claim the current `claimable_amount` to the owner's ATA inside the handler before flipping `yield_recipient`.

(c) Include `claimable_at_set: u64` in the `YieldRecipientSet` event so wallets can surface the donation amount to the user before they sign.

[M-03] toggle_pause Can Leave L1=false While L2=true

Severity: Medium

Description

In toggle_pause.rs:21-24, the `toggle_pause` handler flips `paused` unconditionally without checking `emergency_paused`. The `emergency_pause` instruction sets both `paused = true` and `emergency_paused = true`; `lift_emergency_pause` clears only `emergency_paused` (L2). An admin who calls `emergency_pause` followed by `toggle_pause` reaches the contradictory state `paused = false, emergency_paused = true`:

- `deposit` and `crank_harvest` pass their checks (they check only L1 / `paused`).
- `claim`, `withdraw`, and `partial_withdraw` revert (they check L2 / `emergency_paused`).

This directly contradicts the documented invariant in `DEPLOYMENT.md` that "L2 blocks everything." Users can deposit new funds and the cranker can harvest, but no one can withdraw or claim — the protocol silently accepts inflows while locking all outflows.

Exploit scenario:

1. Admin calls `emergency_pause` in response to a suspected exploit — both flags are true.
2. Admin investigates and concludes the threat was a false alarm.
3. Admin calls `toggle_pause` to resume operations — `paused` becomes false.
4. `emergency_paused` remains true; the protocol accepts new deposits but blocks all withdrawals.
5. Users are silently trapped without any on-chain indication that L2 is still active.

Impact

A simple admin sequencing error produces a state where deposits are accepted but withdrawals are permanently blocked until `lift_emergency_pause` is called, which may not be obvious to the admin. Any new funds deposited in this window are immediately locked.

Recommendation

Add a guard in `toggle_pause` that prevents it from executing while `emergency_paused` is active:

```
// toggle_pause.rs
require!(
    !state.emergency_paused,
    YieldSwapError::CannotToggleWhileEmergencyPaused
);
state.paused = !state.paused;
```

[M-04] Token-2022 TransferFee Mints Cause Silent Vault Insolvency

Severity: Medium

Description

In init_target_vault.rs, the handler accepts any `InterfaceAccount<Mint>` without calling `Mint::get_extension_types()` to check for the `TransferFee` extension. In deposit_yield.rs:102-123, the protocol credits `position.claimable_amount += amount` using the nominal transfer amount rather than the actual delta received by the vault.

For fee-bearing Token-2022 mints, each deposit into the shared `target_vault` transfers less than `amount` (the remainder is withheld as a fee by the mint), but `claimable_amount` is incremented by the

full nominal figure. Over time, the vault's real balance falls behind the sum of all `claimable_amount` values recorded across positions. Late claimers encounter `InsufficientFunds` because the vault cannot cover the nominal amounts it has promised.

The practical risk is gated behind admin curation of target mints; no user can introduce a fee-bearing mint without admin action. However, misconfiguration at the admin level is an operational risk with no on-chain protection.

Impact

If a `TransferFee`-extension mint is accepted as a target vault, the shared vault becomes progressively insolvent as yield is deposited. Early claimers are paid in full; later claimers receive nothing. The discrepancy compounds with every `deposit_yield` call and is undetectable without off-chain monitoring of the vault delta.

Recommendation

Reject `TransferFee`-extension mints at `init_target_vault` initialization time:

```
use spl_token_2022::extension::{BaseStateWithExtensions,
StateWithExtensions};
use spl_token_2022::extension::transfer_fee::TransferFeeConfig;

let mint_data = ctx.accounts.target_mint.to_account_info().data.borrow();
let mint_state = StateWithExtensions::
<spl_token_2022::state::Mint>::unpack(&mint_data)?;
require!(
    mint_state.get_extension::<TransferFeeConfig>().is_err(),
    YieldSwapError::TransferFeeMintNotAllowed
);
```

Alternatively, switch to balance-delta accounting in `deposit_yield` — record the vault balance before and after the CPI transfer and credit only the actual delta.

[M-05] Hard Rate-Diff Cap Causes Permanent Harvest Blockage After Brief Cranker Downtime

Severity: Medium

Description

In `crank_harvest`, V2 replaced the V1 partial-harvest mechanism with a hard `require!(rate_diff <= max_rate_diff, YieldSwapError::HarvestCapExceeded)`. Because the on-chain stake pool exchange rate never decreases, any gap that causes `rate_diff` to exceed `MAX_RATE_INCREASE_BPS` (10 bps) grows monotonically and can never self-heal — the harvest permanently reverts for the affected position.

The cap is extremely tight relative to normal yield. At 7% APY with ~146 epochs per year, per-epoch rate growth is approximately 3.8 bps. The 10 bps cap allows only ~2.5 normal epochs of accumulation (approximately 5–6 days) before triggering a permanent block. Common operational events — a cranker wallet running out of SOL, server downtime, a 24-hour admin timelock on cranker rotation — can all produce gaps of this length.

Proof of concept:

```
Assumptions:
  - Stake pool APY: 7%
  - last_harvested_rate: 1_089_000_000
  - MAX_RATE_INCREASE_BPS: 10 (0.1%)
  - Daily rate growth at 7% APY: ~208,849 lamports/day

Failure scenario — cranker misses ~5 days:
  - max_rate_diff = 1_089_000_000 × 10 / 10_000 = 1_089_000 lamports
  - Day 5: rate_diff ≈ 1,044,245 → passes (under cap)
  - Day 6: rate_diff ≈ 1,253,094 → REVERTS (exceeds cap by 15%)
  - Day 6+: rate only grows. Position can never be harvested again.
  - All future yield is permanently lost for the affected position.
```

Impact

Any cranker downtime exceeding approximately 5–6 days at typical APY permanently bricks harvest for affected positions. All accrued yield from that point forward is irrecoverable. Because the on-chain rate is monotonically increasing, there is no way to recover without a program upgrade or admin intervention.

Recommendation

Restore V1's partial-advance logic: when `rate_diff` exceeds the cap, clamp to `max_rate_diff` and advance `last_harvested_rate` by the clamped amount rather than reverting. The remainder is harvestable in subsequent calls.

```
let (effective_rate_diff, _capped) = if rate_diff > max_rate_diff {
    (max_rate_diff, true)
} else {
    (rate_diff, false)
};

let yield_lst = checked_mul_div(
    position.lst_deposited,
    effective_rate_diff,
    current_exchange_rate,
)?;
```

```
// Advance checkpoint by effective_rate_diff, not current_exchange_rate
position.last_harvested_rate = position.last_harvested_rate
    .checked_add(effective_rate_diff)
    .ok_or(YieldSwapError::MathOverflow)?;
```

Low Severity

[L-01] Timelocked Proposals Never Expire

Severity: Low

Description

All five `accept_*` handlers — `accept_fee`, `accept_admin`, `accept_cranker`, `accept_stake_pool`, and `accept_harvest_limit` — check only `elapsed >= TIMELOCK_SECONDS` with no corresponding upper-bound check. A timelocked proposal therefore remains actionable indefinitely after the timelock window passes.

This means a stale proposal from months ago can be accepted at any future point in time, potentially in an environment where the original context (market conditions, team composition, protocol state) has changed significantly. There is no mechanism for the proposer or a governance body to cancel the proposal if circumstances change, and no on-chain indication of how old a pending proposal is.

Impact

Stale timelocked proposals can be silently accepted long after their intended execution window. A compromised admin key obtained months after a proposal was created could accept arbitrary parameter changes against a protocol that has evolved past the original intent of the proposal.

Recommendation

Add a `MAX_PROPOSAL_LIFETIME` constant and enforce an upper bound across all `accept_*` handlers:

```
const MAX_PROPOSAL_LIFETIME: i64 = 7 * 24 * 60 * 60; // 7 days

require!(
    elapsed >= TIMELOCK_SECONDS,
    YieldSwapError::TimelockNotElapsed
);
require!(
    elapsed <= MAX_PROPOSAL_LIFETIME,
    YieldSwapError::ProposalExpired
);
```

Informational

[I-01] Legacy Positions Brick After Upgrade Due to Missing Realloc Migration

Severity: Informational

Description

The V2 upgrade adds a `yield_recipient: Pubkey` field (+32 bytes) to `UserPosition` and switches to `#[derive(InitSpace)]` for space calculation. A `migrate_position.rs` instruction exists in the codebase but is scoped to target mint changes, not account-size migration. No permissionless `realloc_position` instruction is included in this diff.

All existing `UserPosition` accounts allocated under V1 are 32 bytes short of the new `INIT_SPACE`. Any instruction that writes to the new `yield_recipient` field on a legacy account will cause a memory out-of-bounds write, which the Solana runtime will reject. Legacy positions are effectively bricked for any V2-specific instruction until realloc is performed.

Recommendation

Ship a permissionless `realloc_position` instruction using Anchor's realloc constraint:

```
#[account(
    mut,
    realloc = 8 + UserPosition::INIT_SPACE,
    realloc::zero = true,
    realloc::payer = payer,
)]
pub position: Account<'info, UserPosition>,
```

This zero-inits the new `yield_recipient` field (defaulting to the zero pubkey, which means "no recipient set") and must be run against all legacy positions before enabling V2 instructions in production.

[I-02] Removed EmergencyClearPending Leaves No User Self-Rescue Path

Severity: Informational

Description

V1 provided an `EmergencyClearPending` instruction that allowed users to force-clear a stuck `pending_harvest_lst` after a 7-day timeout if the cranker disappeared. V2 removed this instruction without providing a replacement.

If the cranker vanishes mid-harvest cycle (between `crank_harvest` and `deposit_yield`), the user's position is stuck in a pending state. The user can call `withdraw` to retrieve their LST principal, but the position account cannot be closed because `pending_harvest_lst > 0` prevents clean account

closure. Rent is locked and any pending yield is irrecoverable until either the admin rotates the cranker (24-hour timelock) and the new cranker completes the deposit, or a program upgrade adds a recovery path.

Recommendation

Restore a permissionless self-rescue path. After a configurable timeout (e.g., `EMERGENCY_CLEAR_TIMEOUT = 7 days`), allow the position owner to clear `pending_harvest_lst` and `pending_harvest_timestamp` and close the position:

```
require!(
    clock.unix_timestamp - position.pending_harvest_timestamp >=
EMERGENCY_CLEAR_TIMEOUT,
    YieldSwapError::EmergencyTimeoutNotElapsed
);
position.pending_harvest_lst = 0;
position.pending_harvest_timestamp = 0;
```

[I-03] Stale Comment in deposit_yield References Removed Emergency Feature

Severity: Informational

Description

The comment at deposit_yield.rs:95 still references "Emergency clear (7 days)" — a feature that was removed in V2 alongside `EmergencyClearPending`. The referenced functionality no longer exists in the codebase. Stale comments of this kind mislead auditors and developers into believing a safety mechanism is present when it is not.

Recommendation

Remove or update the stale comment to reflect V2 reality:

```
// Remove the following comment:
// Emergency clear (7 days)

// Replace with accurate description of current behavior, e.g.:
// Clears pending harvest state after deposit_yield completes.
```

[I-04] pending_harvest_timestamp Stored But Never Used for Documented Self-Rescue Flow

Severity: Informational

Description

`UserPosition` stores a `pending_harvest_timestamp: i64` field that is set in `crank_harvest` and cleared in `deposit_yield`. The field is described in state comments and referenced in `DEPLOYMENT.md` as the basis for a 7-day user self-rescue mechanism. However, no instruction in the V2 codebase reads this timestamp for any purpose — the self-rescue flow it was designed to enable was removed along with `EmergencyClearPending` (see I-02).

The field occupies 8 bytes of every `UserPosition` account and is written on every harvest cycle, but its value is never consumed. This creates a discrepancy between documented behavior and on-chain implementation that could mislead users, integrators, and future auditors.

Recommendation

Either implement the documented 7-day emergency-clear flow using `pending_harvest_timestamp` (see recommendation in I-02), or remove the field entirely and update all associated comments and documentation to reflect its absence. Leaving an unused field in a security-critical struct with documentation claiming it enables a recovery mechanism is a maintenance and trust hazard.

Contact

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