Code Assessment
of the ytranche Smart Contracts

PUBLIC

Date July 15, 2026

Produced for yearn

By CHAINSECURITY

Contents

1 Executive Summary

Dear Yearn Team,

Thank you for trusting us to help Yearn with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of ytranche according to **Scope** to support you in forming an opinion on their security risks.

Yearn implements ytranche, a tranche layer over a shared Yearn V3 vault. A central controller pools deposits from several ERC-4626 tranche strategies into one main vault and allocates target yield, excess profit, and losses across the tranches through a configurable priority waterfall.

The most critical subjects covered in our audit are the correctness of the profit and loss waterfall, rounding, and access control. Security regarding all the aforementioned subjects is satisfactory. The issues we uncovered are limited to low-severity edge cases. These include rounding effects whose impact is bounded by the trust assumption that vault share prices cannot be manipulated, see **Withdraw Rounding Leaks Surplus Vault Assets to the Withdrawing Tranche** and **Reserve Deposits Can Freeze Settlement**. The reserve-deposit issue was corrected, while the withdraw-rounding risk was accepted. We also identified low-severity timing and lifecycle issues around loss realization, fresh-tranche donations, and heavily impaired tranches, see **Losses Can Be Front-Run Before Tranche Settlement**, **A Donation Enables a Bounded First-Depositor Attack on a Fresh Tranche**, and **Tranche Losses Can Drive Share Price to Dust, Bricking or Inflating the Tranche**. The timing and donation risks were accepted. The impaired-tranche issue was partially corrected, with the remaining risk being accepted.

The general subjects covered are documentation, events, and gas efficiency. Security regarding all the aforementioned subjects is high.

In summary, we find that the codebase provides a satisfactory level of security.

It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered, and how they have been addressed. We are happy to receive questions and feedback to improve our service.

Sincerely yours,
ChainSecurity

2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

2.1 Scope

The assessment was performed on the source code files inside the ytranche repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

Version	Date	Commit Hash	Note
1	28 June 2026	ee49dd8acc22348c779e99cd8b4f967d531937e4	Initial review
2	10 July 2026	b80d27053674f518f6eaa34ae7954b904e8afea6	Fixes
3	15 July 2026	58182cf41f38aa86365942a23af4876fbd76fca2	Final version

For the Solidity smart contracts, the compiler version `0.8.23` was chosen.

```
src/
├── TrancheStrategy.sol
├── LockedTrancheStrategy.sol
├── TrancheController.sol
├── Hook.sol
└── periphery/
    └── Authorizer.sol
```

2.1.1 Excluded from scope

Any files not listed above were excluded from the scope of this assessment. This includes, but is not limited to, the following files:

- `lib/tokenized-strategy/src/BaseStrategy.sol`
- `lib/tokenized-strategy/src/TokenizedStrategy.sol`
- `lib/tokenized-strategy-periphery/src/Bases/Hooks/Hooks.sol`
- `lib/tokenized-strategy-periphery/src/Bases/Hooks/BaseHooks.sol`
- `lib/tokenized-strategy-periphery/src/Bases/HealthCheck/BaseHealthCheck.sol`
- `lib/yearn-vaults-v3/contracts/VaultV3.sol`

The system requires manual intervention by trusted parties in certain situations. This review excludes the assessment of the procedures and processes for monitoring and managing it.

3 System Overview

This system overview describes the most recent version (**Version 3**) of the contracts as defined in the Assessment Overview.

Furthermore, in the findings section, we have added a version icon to each of the findings to increase the readability of the report.

Yearn implements ytranche, a tranche layer for a shared Yearn V3 vault. The system lets several user-facing ERC-4626 strategies accept the same underlying asset while a central controller allocates principal, target yield, excess yield, and losses across those strategies according to a configurable priority order. Users enter and exit through tranche strategy contracts; the deposited assets are then pooled into one Yearn V3 vault.

The contracts are designed as a generic tranche framework rather than as a fixed senior/junior/equity product. A tranche is represented by a registered strategy address and by the economic parameters assigned to it: its priority in the waterfall, its target rate, and its share of excess profit.

In normal operation, a user deposits into a tranche strategy. The strategy applies its deposit checks and forwards the assets to the `TrancheController`, which credits the depositing tranche and deposits the underlying assets into the main Yearn V3 vault. Periodically, a keeper settles the system. Settlement compares the value held through the main vault against the tranches' accrued claims, allocates profits or losses through the waterfall, and optionally draws on a reserve vault before losses reach tranches. Withdrawals follow the opposite route: a tranche asks the controller to redeem from the main vault, and the assets received by the controller are returned to the tranche strategy.

Several important parts of that flow are inherited from, or delegated to, components outside the assessment scope. The tranche strategies use Yearn's tokenized strategy and periphery hook/health-check contracts for ERC-4626 behavior, reports, management handoffs, shutdown behavior, and per-tranche deposit gating. The main vault is an external Yearn V3 vault, and any reserve is an external same-asset ERC-4626 vault. The system may also be operated together with out-of-scope keeper and emergency helper contracts.

3.1 TrancheController

`TrancheController` is the accounting and settlement layer of the system. It is bound at deployment to one asset and one Yearn V3 main vault. It is also the holder of the main-vault shares backing tranche claims.

Governance registers tranche strategy addresses through `registerTranche()` or `registerTrancheAt()`. Registration gives a tranche its place in the priority order, its annualized target rate, and its share of post-target excess profits. The controller tracks each tranche's accrued baseline assets, any unsettled excess profit, the last accrual timestamp, and whether target accrual is paused. There is no removal path; a tranche that should be wound down can have its target and excess share set to zero while existing holders exit.

Only registered tranches can call the controller's asset-routing functions. On deposit, the controller accrues the calling tranche, increases its baseline by the deposited amount, pulls the underlying asset from the tranche strategy, and deposits it into the main vault. On withdrawal, it accrues and debits the calling tranche, redeems from the main vault, and returns the assets actually received. This means a shortfall realized during redemption is passed back through the tranche withdrawal path.

The controller's `settle()` function is keeper-restricted and is the point where the waterfall is applied. Settlement first accrues all registered tranches and compares the main-vault assets against total claims:

- If the system is profitable, the controller records each tranche's configured excess share as `pendingExcess`. A strictly profitable settlement also resumes target accrual for every tranche that a prior loss had paused. That amount does not immediately enter the strategy's live NAV and is instead realized during the tranche's next report.

- If the system is loss-making, the controller draws from the optional reserve vault first and then allocates any remaining loss from the most junior tranche upward. Losses consume pending excess before baseline assets and pause target accrual for affected tranches until management or a profitable settlement resumes it.

The controller may use an optional same-asset ERC-4626 reserve vault as a first-loss buffer. Anyone can fund the reserve, while governance can configure the reserve vault, withdraw reserve shares, and sweep stray ERC-20 tokens other than protected main-vault and reserve-vault shares.

3.2 TrancheStrategy

`TrancheStrategy` is the base user-facing vault for a tranche with atomic deposits and withdrawals. It inherits Yearn's tokenized-strategy machinery and uses the controller as its source of truth for total assets. As a result, the strategy itself does not implement the waterfall and does not independently price its claim; it asks the controller for `liveAssets(address(this))`.

During deposits, the inherited Yearn flow calls `_deployFunds()`, and the strategy forwards the assets to `CONTROLLER.depositFromTranche()`. During withdrawals, `_freeFunds()` calls `CONTROLLER.withdrawFromTranche()`. The strategy also realizes controller-assigned excess during `report()` by calling `CONTROLLER.realizeExcess()`, so lump-sum settlement profits pass through Yearn's normal reporting and profit-unlocking mechanism rather than appearing instantly in strategy NAV.

The strategy's direct configuration is intentionally small. Governance can rotate the `Hook` with `setHook()`. Deposits are bounded by `availableDepositLimit()`, which combines the inherited Yearn health-check/per-tranche gating with the hook's deposit cap. Withdrawals are bounded by `availableWithdrawLimit()`, which returns the hook's withdrawal cap. After either flow, the strategy calls the hook to consume the corresponding rolling rate-limit capacity.

3.3 LockedTrancheStrategy

`LockedTrancheStrategy` is the cooldown-gated version of `TrancheStrategy`. Deposits are unchanged, but withdrawals require the user to prepare shares with `startCooldown(shares)` while the cooldown is enabled.

Each user has one cooldown bucket. Starting a cooldown records the number of shares, the time at which the cooldown ends, and the time at which the withdrawal window expires; starting a new cooldown replaces the previous bucket. Once the cooldown has matured and the withdrawal window is still open, the user's withdrawal limit is the lower of the hook cap and the assets represented by the cooled shares. The user can also clear the pending bucket with `cancelCooldown()`.

Shares in cooldown remain economically exposed to the tranche. The contract therefore blocks transfers of shares that are still covered by an active cooldown, while allowing transfers outside the active cooldown window and during shutdown mode. Management can adjust the cooldown duration and withdrawal window subject to the contract's maximum cooldown and minimum window bounds.

3.4 Hook

`Hook` is the system's operational policy layer. It is intended to be installed as the Yearn V3 main vault's deposit and withdrawal hook, and it is also called by tranche strategies after their own deposits and withdrawals. The hook does not perform tranche accounting; instead, it meters flow and controls direct access to the main vault.

For tranche strategies, the hook exposes `depositCap()` and `withdrawCap()`:

- `depositCap(tranche)` is capped by both the tranche's own deposit headroom and the main vault's actual deposit headroom, because tranche deposits are immediately routed into the main vault through the controller. This includes aggregate and rolling limits as well as the main vault's paused or shutdown state.
- `withdrawCap(tranche)` is capped by the tranche's rolling withdrawal limit and by `CONTROLLER.vaultMaxWithdraw()`, so redemptions cannot exceed the currently withdrawable main-vault liquidity.

For direct main-vault interactions, the hook uses the Yearn vault hook interface:

- `available_deposit_limit()` enforces the `open` flag and the `allowed` mapping. If the main vault is closed and the receiver is not allow-listed, direct deposits are blocked. Deposits whose receiver is the controller are exempt from this gate (intended for controller-routed tranche deposits), and reserve deposits made by the controller during a reserve draw bypass the rate limit as well.
- `available_withdraw_limit()` combines the main vault's rolling withdrawal limit with `VaultV3WithdrawLimit.maxWithdraw()` for the provided Yearn withdrawal queue and maximum loss parameter.

After a deposit or withdrawal completes, `post_deposit()` and `post_withdraw()` consume the relevant rolling bucket. The bucket is keyed by the caller, so tranche strategy calls consume that tranche's bucket, while main-vault hook calls consume the main vault bucket. Management can configure aggregate deposit limits, rolling deposit and withdrawal limits, the shared rate-limit window, the main-vault open flag, and allow-list entries. The hook deliberately does not block tranche withdrawals based on tranche coverage; solvency and claim coverage are tracked by the controller.

3.5 Authorizer, Authorized, and Roles

`Authorizer` is the shared access-control authority. Contracts inherit `Authorized` to delegate permission checks to it, while `Roles` defines the common role identifiers used throughout the system:

- Governance acts as the runtime superuser for checks made through `isAuthorized()`.
- Management handles operational configuration.
- Keepers perform settlement.
- Emergency actors are intended for halt-related actions.

The contract separates governance from OpenZeppelin's `DEFAULT_ADMIN_ROLE`. They are initialized to the same address but can diverge through independent two-step handoffs. Both core roles are single-holder roles, cannot be renounced, and cannot be directly revoked while live. Holders can renounce non-core roles, and the default admin can update their role-admin relationships. By default, management administers keeper and emergency membership.

The distinction between `isAuthorized()` and `hasRole()` is relevant for integrating contracts. `isAuthorized(role)` accepts either the role holder or governance, while `hasRole(role)` requires the role itself without the governance override.

3.6 Changes in Version 2

- Tranche deposit limits now account for the main vault's actual deposit capacity, including its paused and shutdown states.
- Holders can renounce non-core `Authorizer` roles, while governance and default-admin renunciation remains disabled.

- Reserve-vault shares can no longer be transferred through the generic token sweep and must be withdrawn through the dedicated reserve function.
- The factory and deployment script now configure a performance-fee recipient and explicitly set newly deployed tranches' performance fees to zero. Tranches deployed directly retain the inherited 10% default fee until reconfigured.

3.7 Changes in Version 3

No important functional changes were made in version 3, only bug fixes were applied.

4 Trust Model

This section describes the trust assumptions for the reviewed ytranche contracts. The contracts are not upgradeable, but privileged roles and configured operator addresses can materially change system behavior.

4.1 Roles

The system combines four permission surfaces:

- `Authorizer` roles: `GOVERNANCE_ROLE`, `PENDING_GOVERNANCE_ROLE`, `DEFAULT_ADMIN_ROLE`, `PENDING_DEFAULT_ADMIN_ROLE`, `MANAGEMENT_ROLE`, `KEEPER_ROLE`, and `EMERGENCY_ROLE`.
- Tranche strategy operator addresses inherited from Yearn Tokenized Strategy: `management`, `pendingManagement`, `keeper`, `emergencyAdmin`, and `performanceFeeRecipient`.
- Yearn V3 main vault permissions: bitmask roles from Yearn's `Roles` library, plus the `role_manager` and `future_role_manager` handoff addresses.
- Deployment helper state: `TrancheFactory.management`, `TrancheFactory.keeper`, `TrancheFactory.emergencyAdmin`, `TrancheFactory.performanceFeeRecipient`, and the deployed `EmergencyAdmin.sol` helper.

Based on the `Deploy.s.sol` script, we define the roles below as a combination of `Authorizer` roles, tranche strategy operator addresses, Yearn V3 vault roles, and deployment helper state. Note that roles might be reassigned, and the `Deploy.s.sol` script is not part of the audited codebase. There is no mechanism to enforce that the roles remain unified across the different permission surfaces.

- `Governance`:
  - `Authorizer.GOVERNANCE_ROLE`
  - Yearn V3 vault `role_manager` and `Roles.ALL` holder
- `Default Admin`: `Authorizer.DEFAULT_ADMIN_ROLE`.
- `Pending Governance` / `Pending Default Admin`: the corresponding pending `Authorizer` handoff roles.
- `Management`:
  - `Authorizer.MANAGEMENT_ROLE`
  - `TrancheFactory.management`
  - tranche strategy `management`
- `Pending Management`: tranche strategy `pendingManagement`.
- `Keeper`:
  - `Authorizer.KEEPER_ROLE`
  - tranche strategy `keeper`
  - Yearn V3 vault `REPORTING_MANAGER` and `DEBT_MANAGER`
- `Emergency Actor`:
  - `Authorizer.EMERGENCY_ROLE`
- `EmergencyAdmin Helper` (the deployed `EmergencyAdmin.sol` address):
  - tranche strategy's `emergencyAdmin`
  - Yearn V3 vault `EMERGENCY_MANAGER` and `MAX_DEBT_MANAGER`
- `Strategy Performance Fee Recipient`: the tranche strategy `performanceFeeRecipient`.

- `Vault V3 Future Role Manager`: the Yearn V3 vault `future_role_manager`, which can accept the vault `role_manager` role.

4.1.1 Trust Assumptions

Governance

- Trust level: Fully trusted.
- Permissions: Governance satisfies all `Authorizer.isAuthorized()` checks. It can register and order tranches, set target rates and excess-profit shares, configure and withdraw from the reserve vault, sweep stray ERC-20 tokens except protected main-vault shares, rotate tranche hooks, call keeper-gated yTranche functions, and command `EmergencyAdmin` helper functions. In the intended deployment, it can also administer the Yearn V3 main vault through `role_manager` and `Roles.ALL`.
- Worst-case impact: Malicious or compromised governance can cause loss of funds or denial of service through adverse configuration.

Default Admin

- Trust level: Fully trusted.
- Permissions: Grants and revokes non-core `Authorizer` roles, updates role-admin relationships for non-core roles, and starts the default-admin handoff by assigning `PENDING_DEFAULT_ADMIN_ROLE`.
- Worst-case impact: A malicious or compromised default admin can grant `MANAGEMENT_ROLE` and reassign role admins, ultimately allowing any non-core role, including keeper and emergency, to be granted to malicious accounts.

Pending Governance and Pending Default Admin

- Trust level: Fully trusted during handoff.
- Permissions: Claim the corresponding live governance or default-admin role.
- Worst-case impact: A malicious pending holder can complete the handoff and obtain the live role. An unavailable pending holder can delay an intended transfer until the current holder changes the pending assignment.

Management

- Trust level: Fully trusted.
- Permissions: Management can resume tranche accrual after losses; configure hook limits, rate-limit windows, the main-vault open flag, and allow-list; administer keeper and emergency `Authorizer` roles; and command `EmergencyAdmin.shutdownVault()` / `shutdownStrategy()`. In the strategies, it can also configure cooldowns, strategy operators, deposit gates, health-check ratios, performance-fee settings, profit-unlock timing, and strategy names. In the `TrancheFactory`, it can update future deployment defaults.
- Worst-case impact: Malicious or compromised management can block deposits or withdrawals, grant operational roles to malicious accounts, deploy future tranches with malicious defaults, change fee or profit-locking parameters, trigger irreversible shutdowns through the helper, or otherwise disrupt protocol operation.

Pending Management

- Trust level: Fully trusted during handoff.
- Permissions: Can accept tranche strategy management through the inherited Tokenized Strategy handoff.
- Worst-case impact: A malicious pending management address can become strategy management for that tranche.

Keeper

- Trust level: Fully trusted.
- Permissions: It can call `TrancheController.settle()` when granted `Authorizer.KEEPER_ROLE`, call inherited strategy `report()` and `tend()`, and call Yearn V3 reporting or debt-management functions.
- Worst-case impact: A malicious or unavailable keeper can delay settlement, strategy reporting, pending-excess realization, loss waterfall updates, and accrual resumption after profitable settlements. The keeper cannot choose settlement results directly but can influence fairness between tranches by settling at strategic times, as described in A Recovery After a Significant Loss Flows to Junior Tranches, Not to the Affected Senior Tranches.

Emergency Actor

- Trust level: Partially trusted.
- Permissions: An emergency actor can command `EmergencyAdmin.pauseVault()`, `EmergencyAdmin.emergencyWithdraw()`, and `EmergencyAdmin.zeroMaxDebtForStrategy()`. It is not assumed to directly hold the target strategy `emergencyAdmin` address or Yearn V3 emergency roles; in the intended deployment, those target-level calls are made by the `EmergencyAdmin` helper.
- Worst-case impact: A malicious or compromised emergency actor can cause denial of service by pausing targets or zeroing vault strategy max debt through the helper.

EmergencyAdmin Helper

- Trust level: Partially trusted helper.
- Permissions: When configured as in `Deploy.s.sol`, the helper is the tranche strategy `emergencyAdmin` and holds the Yearn V3 vault `EMERGENCY_MANAGER` and `MAX_DEBT_MANAGER` roles. It executes target-level pause, emergency-withdraw, and max-debt-zero calls when commanded by emergency actors, and shutdown calls when commanded by management or governance.
- Worst-case impact: Misconfiguring the helper for unintended targets lets authorized emergency or management actors affect those targets through the helper.

Strategy Performance Fee Recipient

- Trust level: Untrusted as an operator; trusted only as the intended fee recipient.
- Permissions: Receives strategy performance-fee shares through inherited Tokenized Strategy accounting.
- Worst-case impact: An incorrect recipient can receive fees intended for another party, but this address does not by itself grant administrative control.

Vault V3 Future Role Manager

- Trust level: Fully trusted during handoff.
- Permissions: Can accept the Yearn V3 vault `role_manager` role.
- Worst-case impact: A malicious future role manager can accept control and then assign vault roles.

End Users

- Trust level: Untrusted.
- Permissions: Users can deposit, mint, withdraw, redeem, start or cancel cooldowns, transfer shares subject to cooldown restrictions, fund the reserve, call permissionless views, and deploy tranche strategies through the factory. Factory-deployed tranches still require governance registration before entering the waterfall.
- Worst-case impact: Users can consume rate-limit capacity or attempt to deposit or withdraw around settlement, but should not be able to bypass configured hooks, cooldowns, or role checks.

4.2 External Dependencies

Yearn V3 Main Vault

- Trust level: Fully trusted external dependency.
- Assumptions: The main vault correctly implements its ERC-4626 and Yearn V3 interfaces, including accounting, share conversion, withdrawal queues, max-redeem behavior, and configured hook calls. Its price per share is accurate and cannot be manipulated or inflated by external actors. The vault is assumed to be seeded before integration, and its total assets must remain nonzero for share conversion in tranche stategies.

Yearn Tokenized Strategy and Periphery Contracts

- Trust level: Fully trusted external dependency.
- Assumptions: The inherited tokenized-strategy, hook, health-check, reporting, profit-locking, management, emergency-admin, and shutdown logic behaves as documented and is configured consistently with the yTranche roles.

Yearn Strategies Used by the Main Vault

- Trust level: Trusted within the Yearn vault's own trust model.
- Assumptions: Strategies report assets correctly, respect withdrawal limits, and expose losses or illiquidity through the main vault's accounting and withdrawal queue.

Optional Reserve Vault

- Trust level: Trusted external dependency when configured.
- Assumptions: The reserve vault is a same-asset ERC-4626 vault and remains liquid enough to serve as a first-loss buffer when needed. Its price per share is accurate and cannot be manipulated or inflated by external actors. Anyone may fund the reserve; reserve funders act as altruistic first-loss providers and require no trust, but they in turn trust governance not to withdraw the buffer.

Underlying ERC-20 Asset

- Trust level: Trusted external dependency.
- Assumptions: The asset follows expected ERC-20 behavior and does not introduce unsupported fee-on-transfer, rebasing, blacklist, pause, or callback behavior. The total supply of the asset is assumed to always be smaller than `2**128`.

4.3 General Assumptions

- The system is deployed on Ethereum Mainnet.
- The number of tranches remains low enough for the system's O(n) settlement and loss-absorption gas costs.
- The protocol seeds tranches as the first depositor before external users and resulting shares cannot be ever redeemed.
- Governance and management carefully configure deposit limits, rate-limit windows, tranche `excessShareBps` and `targetRatePerSecondWad`, and profit and loss limit ratios, as these parameters materially affect system behavior.
- Tranche health checks admit expected controller losses/profits. If a loss or profit exceeds the health-check limits, redeemers may exit and depositors may deposit at a better price than remaining holders, see Large Tranche PnL Can Misprice Entry and Exit and Block Reports

- A keeper performs reporting and settlement atomically. No user flow is expected while `pendingExcess` or a reported loss has been recognized by one accounting layer but not the next. Otherwise, users could benefit from inflated share conversions or leave losses to later redeemers.
- Trusted parties monitor the system and trigger settlement at relevant times so that accounting reflects the main vault's current state. See A Recovery After a Significant Loss Flows to Junior Tranches, Not to the Affected Senior Tranches.
- A large loss realized by the underlying strategies may require governance or management intervention, such as pausing or reconfiguring parameters. This review does not define the intervention thresholds or operational steps; trusted parties are assumed to handle both correctly and promptly off-chain.

5 System Considerations

We leverage this section to highlight findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require a modification inside the project. Instead, they should raise awareness in order to improve the overall understanding for users and developers.

SC1 A Recovery After a Significant Loss Flows to Junior Tranches, Not to the Affected Senior Tranches

System Consideration | Version 1

In TrancheController, `settle()` distributes profit only as `pendingExcess`, weighted by each tranche's `excessShareBps`. Losses are absorbed in reverse priority, so a loss large enough to empty the junior tranches cuts a senior tranche's `baselineAssets` and pauses its accrual.

If the vault later recovers, that recovery is registered as profit and split by `excessShareBps`. Senior tranches are configured with little or no excess share, so the recovery flows to the junior tranches that hold the excess shares. The senior's cut `baselineAssets` is not restored, and its target accrual resumes on the reduced baseline, so the senior's principal loss is permanent while the junior tranches that first absorbed the loss recover part or all of their loss.

A loss deep enough to reach a senior tranche is expected to breach the strategy health-check bounds and require management intervention. Measures should then be taken to ensure a fair recovery.

SC2 Frequent Accrual Makes a Tranche Perform Slightly Better

System Consideration | Version 1

In TrancheController, `_accrue()` adds a tranche's interval accrual to its `baselineAssets` and resets `lastAccrual`. Within an interval the target accrues linearly as `baselineAssets * targetRatePerSecondWad * dt`, but because each `_accrue()` re-bases `baselineAssets`, accrual compounds across checkpoints.

Any state-changing tranche call invokes `_accrue()`, so a holder of a tranche with a positive target rate can raise its realized yield by triggering accrual often. The gain over the configured simple rate is bounded by continuous compounding (`e^r - 1 - r`, about 13 bps per year at a 5% target compared to no compounding) and costs gas per trigger, so it is economically marginal.

SC3 One Withdrawal Path Can Consume Shared Exit Headroom

System Consideration | Version 1

In Hook, `available_withdraw_limit()` and `post_withdraw()` use `withdrawRateLimit[address(VAULT)]` for every withdrawal executed by the main vault. This bucket is shared by direct main-vault withdrawals and by controller-routed tranche exits, because `TrancheController.withdrawFromTranche()` ultimately calls `VAULT.redeem()`.

Once any main-vault withdrawal exhausts the shared `withdrawRateLimit[address(VAULT)]`, `vaultMaxWithdraw()` fails for the controller and every tranche. A direct main-vault holder can therefore consume the shared bucket and block tranche exits until the window resets if direct main-vault deposits are enabled. Even without direct main-vault users, one tranche can consume the main-vault bucket and reduce exit capacity for the other tranches.

SC4 Registered Tranches Cannot Be Removed

System Consideration | Version 1

In TrancheController, `registerTranche()` and `registerTrancheAt()` add a Tranche to `tranchesByPriority`, but no function removes entries from that array. Once registered, a Tranche remains in the waterfall.

The available retirement path is only a soft disable: governance can set both target yield and excess-share bps to `0`, and set the deposit cap to `0`. This stops new yield allocation to the Tranche, but it still keeps its place in the profit and loss waterfalls so existing holders can wind down.

SC5 Strategy Emergency Withdrawals Pull No Funds

System Consideration | Version 1

In TrancheStrategy, `emergencyWithdraw()` inherits Yearn's emergency path, but the strategy never overrides `_emergencyWithdraw()`. `TokenizedStrategy.emergencyWithdraw()` requires the strategy to be paused or shut down, then calls `shutdownWithdraw()`, which dispatches to BaseStrategy's empty default hook:

```
function _emergencyWithdraw(uint256 _amount) internal virtual {}
```

TrancheStrategy only overrides `_deployFunds()`, `_freeFunds()`, and `_harvestAndReport()`. LockedTrancheStrategy inherits the same behavior. As a result, EmergencyAdmin's documented `emergencyWithdraw()` lever can succeed after passing authorization and pause-or-shutdown checks while pulling no assets back from the controller or main vault.

SC6 Excess-Share Changes Redistribute Accrued Profit

System Consideration | Version 1

In TrancheController, `settle()` splits the pending surplus using each Tranche's current `excessShareBps`. The surplus was earned since the previous settlement, but the split does not account for the configuration that was active while it accrued.

Governance can change the shares mid-window through `setTrancheExcessShareBps()`, `registerTranche()`, or `registerTrancheAt()`. If the change lands before `settle()`, the new configuration applies to all already-earned surplus: newly added or increased Tranches receive part of the past profit, while reduced Tranches forfeit part of the profit earned under their old share.

If the change lands after `settle()`, the previous configuration receives the round instead. The excess split for a settlement window therefore depends on transaction ordering between the configuration update and `settle()`.

SC7 Strategy Has Two Access-Control Registries

System Consideration | Version 1

LockedTrancheStrategy extends TrancheStrategy, so one deployed strategy exposes admin setters gated by two unrelated authority sources. In TrancheStrategy, `setHook()` uses `isAuthorized(GOVERNANCE_ROLE)`, which resolves through the shared Authorizer. In LockedTrancheStrategy, `setCooldownDuration()` and

`setWithdrawalWindow()` use the TokenizedStrategy-provided `onlyManagement` modifier, which checks the `management` address held in TokenizedStrategy storage.

The Authorizer role set and the TokenizedStrategy `management` address are configured independently, and nothing keeps them in sync. Rotating governance in the Authorizer does not change who holds `management`, and vice versa. The two admin surfaces on the same contract can therefore be controlled by different parties.

SC8 Large Tranche PnL Can Misprice Entry and Exit and Block Reports

System Consideration | Version 2

`BaseHealthCheck.strategyTotalAssets()` clamps `CONTROLLER.liveAssets()` to the configured profit and loss limits around `lastTotalAssets`. TokenizedStrategy then uses this clamped value to price deposits and withdrawals, even when the controller reports a different tranche value.

After a large loss, the lower clamp overstates the tranche's value. Early redeemers can withdraw at the stale price and leave more of the loss to remaining holders.

Large profits have the inverse effect. Settlement profit remains outside live NAV as `pendingExcess` until it is reported, while target-rate accrual above `profitLimitRatio` is exposed only up to the upper clamp. Depositors can enter at the understated price and later share in the unrecognized profit, whereas redeemers leave without their full share. Excess target-rate accrual is recognized gradually as the clamp advances across later blocks.

In either direction, `report()` can revert if the raw value returned by `harvestAndReport()` remains outside the health-check range after its initial clamped accrual. For settlement profit, the revert also rolls back `realizeExcess()` and leaves the profit pending until management intervenes.

As per the Trust Model, `BaseHealthCheck` is expected to be configured to admit expected controller losses and profits, reporting and settlement are expected to be performed atomically, and governance and management are expected to intervene if a loss or profit exceeds the health-check limits.

6 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- **Likelihood** represents the likelihood of a finding to be triggered or exploited in practice
- **Impact** specifies the technical and business-related consequences of a finding
- **Severity** is derived based on the likelihood and the impact

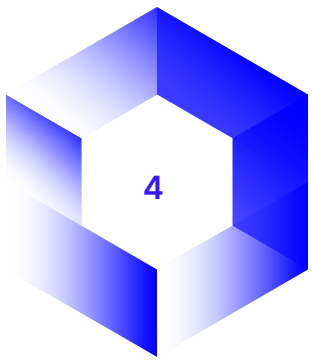
We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

Likelihood	**Impact**		
	High	Medium	Low
High	**Critical**	**High**	**Medium**
Medium	**High**	**Medium**	**Low**
Low	**Medium**	**Low**	**Low**

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

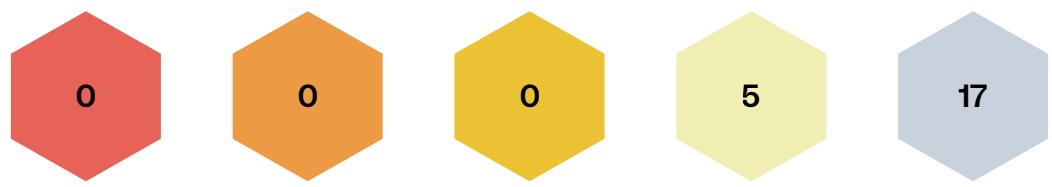
7 Findings

In this section, we describe the findings identified during the course of the engagement. The findings are categorized by severity as explained in the Terminology section. Below we provide an overview of the total number of findings per severity, as well as the number of open issues.



Open Issues

Total number of findings



Critical High Medium Low Informational

7.1 Overview

The following table lists all identified findings along with their severity and current status. A finding is considered resolved once it has been fully corrected or the specification has been changed accordingly. Non-informational findings with any other status, including partial fixes, acknowledgements, and accepted risks, remain open.

Low	#001	A Donation Enables a Bounded First-Depositor Attack on a Fresh Tranche	Risk Accepted
Low	#002	Losses Can Be Front-Run Before Tranche Settlement	Risk Accepted
Low	#003	Reserve Deposits Can Freeze Settlement	Code Corrected
Low	#004	Tranche Losses Can Drive Share Price to Dust, Bricking or Inflating the Tranche	Code Part. Corrected / Risk Accepted
Low	#005	Withdraw Rounding Leaks Surplus Vault Assets to the Withdrawing Tranche	Risk Accepted
Info	#006	`fundReserve()` Lacks a Guard for an Unset Reserve Vault	Code Corrected
Info	#007	`sweep()` Can Withdraw Reserve Shares	Code Corrected
Info	#008	A Rolling Cooldown Keeps a Holder Exit-Ready at No Cost	Acknowledged
Info	#009	Any Caller Can Deposit into the Gated Main Vault	Acknowledged
Info	#010	Deposit Limits Stay Positive While the Main Vault Is Paused	Code Corrected
Info	#011	Deposit Rounding Can Leave Tranche Claims Underbacked	Acknowledged
Info	#012	Dust Shares Can Block Reserve-Vault Replacement	Acknowledged
Info	#014	Gas Optimizations	Code Corrected
Info	#015	Performance Fee Override Emits No Event	Code Corrected
Info	#016	Performance Fees Can Accrue to the Factory	Code Corrected
Info	#017	Reserve Rounding Can Subsidize Tranches	Acknowledged
Info	#018	Roles Cannot Be Renounced	Code Corrected
Info	#019	Small Tranches Can Miss Target Accrual	Specification Changed
Info	#021	Unbounded Withdrawal Windows Can Freeze Cooldowns	Acknowledged
Info	#022	Unregistered Tranche Views Return Defaults	Code Corrected

Info	#023	Zero Pending Governance Stalls Handoffs	Acknowledged	⊖
Info	#024	Donated Idle Assets Can Block Tranche Deposits	Acknowledged	⊖

7.2 Descriptions

#001 A Donation Enables a Bounded First-Depositor Attack on a Fresh Tranche

Security	Low	Version 1	Risk Accepted

TrancheStrategy prices its shares off the controller's live value, and a quirk in how deposits are booked lets that value be inflated with a plain token transfer.

When someone deposits, the strategy sweeps its whole loose balance into the controller through `depositFromTranche()`, but it credits only the depositor's own `assets` to `lastTotalAssets`. So tokens sent to the strategy beforehand get pushed into the controller baseline without being recorded as anyone's principal, and on the next accrual they surface as profit. Because the tranche accrues live rather than only at `report()`, that profit immediately lifts the share price for whoever already holds shares, with no profit-lock and no revert. The one limiter is the health-check clamp, which caps how far the recorded value can rise in a single accrual, roughly a doubling at the default profit limit. Because accrual advances at most once per block, the price cannot be inflated in one transaction. It has to be doubled block by block, so climbing from a price of 1 to the target takes as many blocks as doublings are needed, about 66 in the run below.

Yearn's usual defense, the dead-share mint, does not close this off. It redirects profit to `DEAD_ADDRESS` only while the share supply is strictly below `MINIMUM_SUPPLY` (1000), so seeding exactly 1000 shares costs a few wei and steps past it.

What `MINIMUM_SUPPLY` does do is make the required capital for the attack larger. The most a first-depositor attack can take from one victim is their rounding loss, at most a single share's worth of assets. But pushing the share price high enough to cause that loss means locking an amount equal to the supply times the price, and the supply has to be at least 1000 to clear the dead-share floor. So the attacker always locks on the order of a thousand times what they extract, and a bigger victim does not help, since attacking it needs a proportionally higher price and proportionally more locked capital.

A concrete flow, on an empty tranche with an 18-decimal asset:

1. Seed 1000 shares so the supply clears the dead-share floor. The price starts at 1.
2. For about 66 blocks (~13 minutes), repeatedly donate the current baseline and make a dust deposit. Each donation is swept in and recognized on the next accrual, roughly doubling the price per block.
3. Once the price sits just above half of a target deposit (around `50.25e18` for a `100e18` deposit), that deposit mints a single share worth about `50e18` and loses the rest, roughly `49.75e18` or just under 50%.
4. The attacker redeems their ~1000 shares and recovers the locked capital plus almost all of the victim's loss.

So netting about 50 tokens meant locking roughly 50,250, keeping it there through the whole visible 13-minute ramp, and counting on no one else depositing in the meantime, since any deposit during the ramp shares in the donation. The attack also needs the tranche to be empty and the victim to deposit without checking how many shares they will receive. One smaller side effect: a donation large enough to exceed the profit limit in a single step makes `report()` revert until management disables the health check.

Note that TokenizedStrategy is not in the audit scope. This finding covers how TrancheStrategy's deposit routing interacts with it.

Risk Accepted

Yearn accepted the risk and noted:

> No economic motive. Would also expect live deployments to have both tighter health-check bounds on profit limit as well as being seeded by the protocol as first depositor before any external users.

#002 Losses Can Be Front-Run Before Tranche Settlement

Security	Low	Version 1	Risk Accepted

In TrancheController, tranche NAV is based on `baselineAssets` through `_liveAssetsView()` and is independent of `vaultAssets()` until `settle()` rewrites tranche accounting. After the main vault processes a strategy loss, `vaultAssets()` drops immediately, but tranche share prices still use stale baselines until the keeper calls `settle()`.

The reporting stack must run bottom-up:

1. Underlying strategies call `report()`.
2. The main vault calls `process_report()` to fold each strategy's gain or loss into vault accounting.
3. TrancheController calls `settle()` to compare `vaultAssets()` against tranche claims.
4. Each tranche strategy calls `report()` to realize `pendingExcess`.

If `process_report()` has already realized a loss but `settle()` has not yet run, the loss is visible on-chain while tranche shares remain priced at their pre-loss baseline.

During that window, a holder can withdraw or redeem before the keeper settlement transaction. `withdrawFromTranche()` accrues and debits the requested `_amount` from the tranche baseline, then redeems enough main-vault shares to receive `_amount` when liquidity is available. The exiting holder therefore leaves at stale tranche value, and the pending loss remains to be absorbed by the remaining holders when `settle()` later cuts baselines junior-first.

If the loss is still unrealized at the main-vault strategy level, the vault withdraw path accounts for the unrealized loss and the exiting holder bears it. However, the exiting holder may still get a better price than the remaining holders if:

- the vault has idle assets that can satisfy the withdrawal;
- the vault has a lossless strategy that can withdraw assets while other strategies have unrealized losses.

This is loss-side only. Main-vault profits are locked by `process_report()`, then held as `pendingExcess` by `settle()`, then realized into tranche strategy accounting during tranche `report()`. They do not create an immediate tranche price-per-share jump for a just-in-time depositor.

The practical impact is limited. Senior tranche holders have no cooldown but are loss-last, so the issue matters there only for losses large enough to reach the senior layer. Junior and equity tranche holders are gated by LockedTrancheStrategy cooldowns, which are not a hard barrier given A Rolling Cooldown Keeps a Holder Exit-Ready at No Cost.

> **Risk Accepted**
>
> Yearn accepted the risk and noted that it should be mitigated by keeping the cooldown period longer than the settlement window.

#003 Reserve Deposits Can Freeze Settlement

Correctness	Low	Version 2	Code Corrected

In TrancheController's loss path, `_drawReserveToMain()` redeems reserve-vault shares and deposits any received assets into the main vault. A Yearn V3 vault rejects deposits that mint zero shares, but the helper only

checks that `received > 0`. A reserve draw worth less than one main-vault share therefore makes `VAULT.deposit()` revert and blocks `settle()`.

This is not limited to tiny losses. Since the reserve draw is `min(loss, reserveAssets())`, a large loss combined with a small reserve can produce the same sub-one-share deposit. Settlement then remains blocked until the reserve is funded enough to mint a share or governance withdraws it.

Settlement also remains blocked for any loss that draws from the reserve after the main vault is shut down, because shutdown permanently disables deposits. Governance must withdraw the entire reserve so the draw is skipped, defeating its purpose as a first-loss buffer during wind-down.

Code Corrected

`_drawReserveToMain()` now checks the results of `previewRedeem()` on the reserve and `previewDeposit()` on the main vault. If the previewed redemption returns 0 assets, the entire draw is skipped. Otherwise, the shares are redeemed from the reserve. If depositing the received assets would mint 0 shares, the main-vault deposit is skipped. A sub-one-share reserve draw can therefore no longer freeze `settle()`. Additionally, the function returns 0 if the main vault is shut down.

`_drawReserveToMain()` can leave underlying assets stranded in the controller. If `redeem()` returns a non-zero amount of assets but `deposit()` would mint 0 shares, the controller will retain those assets until governance sweeps them. The stranded amount is bounded by the value of one main-vault share per occurrence.

#004 Tranche Losses Can Drive Share Price to Dust, Bricking or Inflating the Tranche

Design	Low	Version 1	Code Partially Corrected / Risk Accepted

In TrancheController, `settle()` absorbs losses by reducing a tranche's `baselineAssets`. TrancheStrategy uses `CONTROLLER.liveAssets(address(this))` as its asset source, so once the loss is recorded in the inherited TokenizedStrategy accounting, the tranche strategy can have nonzero share supply with zero or near-zero `lastTotalAssets`.

- If `lastTotalAssets` reaches zero while supply remains nonzero, new deposits cannot mint shares: `convertToShares` returns zero because `totalAssets` is zero, so `deposit` / `mint` revert. There is no management setter that raises `baselineAssets` directly: `resumeAccrual()` only unpauses, and accrual on a zero baseline still yields zero. The tranche therefore cannot accept new deposits until a strictly profitable `settle()` records `pendingExcess` for it (only possible if its `excessShareBps > 0`) and a subsequent `report()` realizes that excess into the baseline via `realizeExcess()`. That report only succeeds if management first sets `doHealthCheck` to false, since with `lastTotalAssets = 0` the health-check band is `[0, 0]` and `_executeHealthCheck` reverts on any `_newTotalAssets > 0`.
- If `lastTotalAssets` is only dust, deposits mint a very large number of shares because the denominator is tiny. This can repeat for a tranche that absorbs first losses. If a tranche has supply `S`, loss leaves assets `d`, and the next refill deposits `D`, TokenizedStrategy mints about `D * S / d` shares, so the new supply is about `S * (1 + D / d)`.

Step	Supply	Note
Start	`1e18`	Baseline case
After cycle 1	`1e39`	Above `2**128`, so LockedTrancheStrategy cooldown storage can truncate
After cycle 2	`1e60`	Still fits in `uint256`

Step	Supply	Note
Cycle 3 refill	`1e60`	Deposit reverts: it would mint `1e81` shares, above `2**256`

The table uses `D = 1e22` and `d = 10`, so each successful refill multiplies supply by about `1e21`. More generally, deposits stop once `S * (D / d) ** n` exceeds `type(uint256).max`.

The revert occurs in the deposit share calculation, not in the multiplication itself. OpenZeppelin `Math.mulDiv()` uses a 512-bit intermediate for `assets * supply`, then reverts with `"Math: mulDiv overflow"` when the final quotient cannot fit in `uint256`. Exit reads use `shares * assets / supply`, with the inflated supply in the denominator, so existing holders can still redeem while new deposits revert.

LockedTrancheStrategy additionally stores cooldown amounts as `uint128` and casts with `uint128(_shares)`. A single dust-refill cycle can already push balances above `2**128`, so `startCooldown()` can silently record a truncated cooldown amount and require exits in `<= 2**128`-share chunks.

Code Partially Corrected

A check for `_shares < type(uint128).max` was added in `LockedTrancheStrategy.startCooldown` to prevent silent truncation of cooldown amounts. After the vault's total assets went to a dust amount, users should not deposit if the price per share is too low, as they could need a large, potentially unrealistic, number of cooldown cycles to fully exit their position that could be much larger than the `uint128` limit.

Risk Accepted

Yearn accepted the remaining risks of this issue and answered:

> If total assets = 0, deposits reverting is correct behavior, and should only allow recovery if a settle has funds for it. Dust accepted, low likelihood and no economic risk.

#005 Withdraw Rounding Leaks Surplus Vault Assets to the Withdrawing Tranche

Correctness	Low	Version 1	Risk Accepted

In TrancheController, `withdrawFromTranche()` debits `_amount` from `baselineAssets` before redeeming `VAULT.previewWithdraw(_amount)` shares from the main vault. `previewWithdraw()` rounds shares up, so redeeming those shares can return more assets than the debited amount.

For example, if one vault share is worth 3 assets and the tranche withdraws 2 assets, `previewWithdraw(2)` returns 1 share. Redeeming 1 share returns 3 assets, while only 2 assets were removed from `baselineAssets`. The tokenized strategy transfers only the requested withdrawal amount to the user, leaving the extra asset as idle balance in the strategy. That idle balance can later satisfy another withdrawal or be redeposited and credited to the tranche's baseline in the controller.

This can be exploited by repeatedly withdrawing small amounts from a tranche, inflating that tranche's `baselineAssets` by up to one share's worth of assets each time.

The surplus benefits the withdrawing tranche and is therefore a loss socialized junior-first to all tranches in the controller. This remains low severity because the trust model assumes the Vault V3 share price is neither manipulable nor inflatable, so one wei of vault shares should be worth very little in assets.

Risk Accepted

Yearn accepted the risk.

#006 `fundReserve()` Lacks a Guard for an Unset Reserve Vault

Informational | **Version 1** | **Code Corrected**

In TrancheController, `reserveVault` is optional: `setReserveVault()` accepts `address(0)` to clear it, and `reserveAssets()` returns `0` when no reserve vault is set. `fundReserve()` has no equivalent guard and calls `forceApprove()` and `deposit()` directly on `reserveVault`.

While the call to `deposit` will fail and revert the call, a clearer `"reserve not set"` error would be more helpful than the low-level failure from calling `address(0)`.

> **Code Corrected**
>
> `fundReserve()` now includes a `require` statement that reverts with a clear error when no reserve vault is configured.

#007 `sweep()` Can Withdraw Reserve Shares

Informational | **Version 1** | **Code Corrected**

In TrancheController, both `withdrawReserve()` and `sweep()` are gated by `GOVERNANCE_ROLE` and only transfer ERC20 tokens to a receiver. `withdrawReserve()` transfers `reserveVault` shares, while `sweep()` transfers any `_token`.

Because `sweep()` blocks only `VAULT`, governance can pass `address(reserveVault)` and move the same shares through `sweep()`. This makes `withdrawReserve()` redundant and leaves the protection model inconsistent: reserve shares are system backing, yet they are not treated as a protected token.

> **Code Corrected**
>
> `sweep()` now treats the `reserveVault` shares as a protected token alongside `VAULT` shares.

#008 A Rolling Cooldown Keeps a Holder Exit-Ready at No Cost

Informational | **Version 1** | **Acknowledged**

In LockedTrancheStrategy, redemptions are gated by a cooldown: a holder calls `startCooldown()`, waits `cooldownDuration`, then redeems within `withdrawalWindow`. The gate is meant to impose a notice period before exit.

Root cause: cooled shares remain economically active and keep accruing, so maintaining a standing cooldown costs nothing (the only forfeited ability is transferring those shares, which a redeeming holder does not need). `startCooldown()` sets `cooldownEnd = block.timestamp + cooldownDuration` and overwrites the prior record, so a holder arms once, becomes redeemable for `withdrawalWindow`, re-arms after it lapses, and repeats.

This keeps the holder exit-ready for `withdrawalWindow` out of every `cooldownDuration + withdrawalWindow`, about 42% of the time at a 7-day cooldown and 5-day window. The cooldown therefore enforces only a probabilistic, duty-cycle-limited delay rather than a reliable notice period, so a holder can stay positioned to redeem on short notice at no cost.

Acknowledged

Yearn acknowledged the issue and noted:

> Will adjust cooldown and window if seen happening in live deployments to discourage.

#009 Any Caller Can Deposit into the Gated Main Vault

Informational	Version 1	Acknowledged

In Hook, `available_deposit_limit()` is meant to keep the main vault closed to direct depositors: when `open` is false and the receiver is not in `allowed`, it returns zero. The comment states:

The controller itself (Tranche-routed deposits) is always permitted

But the exemption for the Controller keys on `_receiver`, not on the caller. The main vault calls this hook during `VAULT.deposit(assets, receiver)`, so any address can pass `receiver = CONTROLLER` and skip the gate, depositing into the main vault while it is closed.

The minted shares are credited to the Controller, so the depositor cannot recover the assets, which instead surface as profit at the next `settle()`.

Acknowledged

Yearn acknowledged the finding.

#010 Deposit Limits Stay Positive While the Main Vault Is Paused

Informational	Version 1	Code Corrected

In Hook, `depositCap()` bounds tranche deposits by `_vaultDepositLimit()` for both the tranche and the main vault. `_vaultDepositLimit()` only checks `depositRateLimit`, `depositLimits`, and `totalAssets()`, so it can return a positive main-vault deposit cap while the main vault is paused.

The withdraw path is pause-aware because `withdrawCap()` is bounded by `TrancheController.vaultMaxWithdraw()`, which reads `VAULT.maxRedeem()`, returning zero when the vault is paused. The deposit path has no equivalent pause check. Therefore TrancheStrategy `availableDepositLimit()` can show a positive limit even though `TrancheController.depositFromTranche()` must later revert in `VAULT.deposit()`, because Yearn's `_max_deposit()` returns zero when paused.

Code Corrected

`depositCap()` now bounds the main-vault deposit cap using the main vault's `maxDeposit()`, which applies the vault's pause check (re-entering the hook after it).

#011 Deposit Rounding Can Leave Tranche Claims Underbacked

Informational	Version 1	Acknowledged

In TrancheController, `depositFromTranche()` credits the tranche's `baselineAssets` by the full `_amount`, then deposits `_amount` into the main vault. Yearn V3 `deposit()` mints shares with `previewDeposit()`

semantics, which round down, so the controller can receive slightly less vault-share backing than the claim it just recorded.

`vaultAssets()` measures only the assets attributable to the controller's shares. If any main-vault supply is held outside the controller, the unminted fractional share value is shared with that supply instead of remaining fully attributable to the controller. As a result, total tranche claims can increase by `_amount` while controller-owned main-vault backing increases by slightly less. The same rounding applies when `_drawReserveToMain()` deposits `received` into the main vault.

The deficit is bounded by one share worth of assets, and it can later appear as negative PnL in `settle()` and be absorbed through the waterfall. The impact is informational because the trust model assumes the Vault V3 share price is not manipulable, so one share wei should be worth very little in assets.

Acknowledged

Yearn acknowledged the issue.

#012 Dust Shares Can Block Reserve-Vault Replacement

Informational | **Version 1** | **Acknowledged**

In TrancheController, `setReserveVault()` refuses to replace the current reserve vault unless the controller holds none of its shares:

```
if (address(reserveVault) != address(0)) {
    require(reserveVault.balanceOf(address(this)) == 0, "reserve not empty");
}
```

Reserve-vault shares are ordinary ERC4626/ERC20 tokens. Anyone can mint them directly or through the permissionless `fundReserve()`, then transfer dust to the controller. A single 1-wei share transfer makes the guard revert.

Governance can sweep with `withdrawReserve()`, but unless the sweep and `setReserveVault()` are batched atomically, an attacker can front-run each replacement attempt with fresh dust. Cost is 1 wei of shares plus gas per attempt; impact is griefing DoS on reserve-vault replacement and clearing.

Acknowledged

Yearn acknowledged the finding and noted:

> No incentive. Governance can do both atomically.

#014 Gas Optimizations

Informational | **Version 1** | **Code Corrected**

The following are some gas optimizations that can be made to the codebase:

1. In `setTrancheTargetBps`, the newly written `tranche.targetRatePerSecondWad` is re-read from storage for the event, which is avoidable.
2. In `fundReserve`, `reserveVault` is read twice from storage, the second read is avoidable.
3. In `reserveAssets`, `reserveVault` is read three times from storage, the second and third reads are avoidable.

4. In `_drawReserveToMain`, `reserveVault` is read three times from storage, the second and third reads are avoidable.
5. In `settle`, `tranchesByPriority` is indexed from storage element-by-element in both the accrual loop and the profit/loss loop, instead of being cached to a memory array once (as `totalClaims` and `trancheCoverage` already do). The array is walked twice per call, so the second walk pays an avoidable storage read per tranche.
6. In `settle`, after `_accrue(tranche)` writes `tranche.baselineAssets`, that field is immediately re-read from storage to accumulate `totalClaim`; `_accrue` could return the updated baseline instead.
7. `_accrue` calls `_liveAssetsView(Tranche memory)`, so passing the storage struct to a memory parameter copies the entire struct into memory, SLOAD-ing all seven fields (packed in 5 storage slots) when `_liveAssetsView` only needs to read four slots.

Code Corrected

All optimizations have been implemented except for item 6.

#015 Performance Fee Override Emits No Event

Informational	Version 1	Code Corrected

In TrancheStrategy, the constructor overrides the inherited performance fee with a direct storage write:

```
TokenizedStrategy.strategyStorage().performanceFee = 0;
```

The setter cannot be used at that point. `setPerformanceFee()` first calls `_accrue()`, which reads `controller.liveAssets(strategy)` for a Tranche that has not yet been registered with the controller and reverts. The direct write is therefore the intended workaround.

The side effect is that construction emits no `UpdatePerformanceFee` event for the override. `TokenizedStrategy.initialize()` sets `performanceFee = 1_000` (10%) as the documented default, so off-chain consumers that seed state from the default and then follow `UpdatePerformanceFee` events can report a 10% fee while the on-chain value is actually 0. The discrepancy persists until management later calls `setPerformanceFee()` after registration, which emits the event and re-syncs those consumers.

Code Corrected

The factory now sets the performance fee to zero using the setter. This is possible because `TrancheController.liveAssets()`, called by the setter, is now non-reverting for an unregistered tranche.

#016 Performance Fees Can Accrue to the Factory

Informational	Version 1	Code Corrected

Tranches are deployed by `TrancheFactory`, which is out of scope of this review. During construction, in `BaseStrategy`, the initial `management`, `keeper`, and `performanceFeeRecipient` are all set to `msg.sender`, the factory. `TrancheFactory._configureStrategy()` then overrides `keeper`, `emergencyAdmin`, and pending management via `setPendingManagement()`, but never updates `performanceFeeRecipient`. It therefore remains set to the factory.

This is dormant while fees are disabled. `TrancheStrategy`'s constructor sets `performanceFee = 0`, so `_chargeFees()` mints nothing. Once the configured management accepts the role and calls

`setPerformanceFee()` with a non-zero value, however, `_chargeFees()` mints the strategy's fee shares to `S.performanceFeeRecipient`.

If management does not first call `setPerformanceFeeRecipient()`, those shares accrue to `TrancheFactory`. The factory has no function to transfer, redeem, or sweep strategy shares, so the fee shares are effectively burned.

Code Corrected

The factory now sets the performance fee recipient to a management-configured address during strategy deployment. This is possible because `TrancheController.liveAssets()`, called by the setter, is now non-reverting for an unregistered tranche.

#017 Reserve Rounding Can Subsidize Tranches

Informational	Version 1	Acknowledged

In TrancheController, `settle()` draws from the reserve through `_drawReserveToMain()` to cover negative PnL. `_drawReserveToMain()` redeems `reserveVault.previewWithdraw(_amount)` shares, and `previewWithdraw()` rounds shares up, so `received` can be greater than `_amount`.

The helper deposits the full `received` amount into the main vault. `settle()` then reduces `loss` by `_min(loss, drawn)`, so any surplus over the loss remains in the main vault and can surface as socialized profit in a later settlement.

The impact is informational because the trust model assumes the reserve vault share price is not manipulable, so one share wei should be worth very little in assets.

Acknowledged

Yearn acknowledged this finding.

#018 Roles Cannot Be Renounced

Informational	Version 1	Code Corrected

In Authorizer, `renounceRole()` always reverts with `"renounce disabled"`. This is broader than the core-role protection used in `revokeRole()`, which blocks only `GOVERNANCE_ROLE` and `DEFAULT_ADMIN_ROLE` so they can move only through the two-step claim path.

As a result, holders of `MANAGEMENT_ROLE`, `KEEPER_ROLE`, `EMERGENCY_ROLE`, or any custom role cannot voluntarily remove themselves. Even a compromised or unwanted non-core key must wait for its admin to call `revokeRole()`.

Code Corrected

The function `renounceRole()` was updated to only block the core roles, allowing non-core role holders to renounce their own roles. The function's documentation was subsequently updated to reflect the new behavior.

#019 Small Tranches Can Miss Target Accrual

Informational | **Version 1** | **Specification Changed**

In TrancheController, `_liveAssetsView()` computes target accrual as `baselineAssets * targetRatePerSecondWad * dt / WAD`. The division floors each accrual interval, so frequent `_accrue()` calls can round a small tranche's interest to zero.

At a 5% target rate, `_bpsToPerSecondWad()` returns `1,584,436,925`. With 12-second blocks, a checkpoint earns non-zero interest only when `baselineAssets` is at least `52,594,921` base units. For an 18-decimal asset this is dust, but for a 6-decimal asset, for example, it is about 52 tokens. A smaller tranche accrued every block earns no target interest until its balance or the elapsed time is large enough.

Specification Changed

The following comment was added to `setTrancheTargetBps()` to clarify that small tranches may not accrue interest:

```
/// @dev Accrual floors `baseline * rate * dt / WAD`, so an interval below
///      `ceil(WAD / (rate * dt))` base units rounds to zero (remainder is not
///      carried). This could effect accruals of assets checkpointed near every block
///      e.g. 1.5% on 8-decimal cbBTC floors at ~1.75 units per 12s checkpoint.
```

#021 Unbounded Withdrawal Windows Can Freeze Cooldowns

Informational | **Version 1** | **Acknowledged**

In LockedTrancheStrategy, `setWithdrawalWindow()` enforces only a minimum withdrawal window:

```
require(_newWithdrawalWindow >= MIN_WITHDRAWAL_WINDOW, ...);
```

There is no upper bound. `startCooldown()` computes `windowEnd = cooldownEnd + withdrawalWindow` in `uint256`, but stores the result as `uint64`:

```
uint256 windowEnd = cooldownEnd + withdrawalWindow;
cooldowns[msg.sender] =
    UserCooldown({cooldownEnd: uint64(cooldownEnd), windowEnd: uint64(windowEnd), shares:
        uint128(_shares)});
```

If management sets `withdrawalWindow` large enough that `block.timestamp + cooldownDuration + withdrawalWindow` exceeds `type(uint64).max`, the `uint64(windowEnd)` cast silently truncates. The stored `windowEnd`, after the modular reduction, can represent a timestamp in the past. Each later `startCooldown()` records the same stale window, and `availableWithdrawLimit()` returns 0 for the affected user because `block.timestamp > cooldown.windowEnd`. Cooldown redemptions remain unavailable until management resets the window to a sane value.

Acknowledged

Yearn acknowledged this finding.

#022 Unregistered Tranche Views Return Defaults

Informational	Version 1	Code Corrected

In TrancheController, the view functions `pendingExcess()` and `isAccrualPaused()` read straight from the `tranches` mapping without checking the `registered` flag. Other tranche-keyed entry points guard with `require(tranche.registered, "!tranche")`.

For an address that was never registered, the mapping yields a zero-initialized struct, so `pendingExcess()` returns `0` and `isAccrualPaused()` returns `false`. A caller cannot distinguish an unregistered tranche from a registered one that genuinely has zero pending excess or unpaused accrual.

> **Code Corrected**
>
> The two functions were not updated, instead `liveAssets` and `trancheCoverage` were made non-reverting on unregistered tranches to make the behavior of view functions consistent on unregistered tranches. The `isTranche` function should instead be used to check if a tranche is registered before calling these view functions.

#023 Zero Pending Governance Stalls Handoffs

Informational	Version 1	Acknowledged

In Authorizer, governance and default-admin handoffs are two-step. The current holder proposes a pending holder through `grantRole()`, and the pending holder later claims the live role. For pending roles, `grantRole()` only checks that `getRoleMemberCount(role) == 0` and does not reject the zero address. As a result, `grantRole(PENDING_GOVERNANCE_ROLE, address(0))` and `grantRole(PENDING_DEFAULT_ADMIN_ROLE, address(0))` both succeed.

Once the pending slot holds `address(0)`, the handoff cannot complete. The zero address can never call `grantRole()` to claim the live role, and any later `grantRole(PENDING_*, otherAddress)` reverts on the `"pending role set"` guard because the slot is already occupied.

This does not permanently brick governance. `revokeRole()` blocks only the live core roles (`GOVERNANCE_ROLE` and `DEFAULT_ADMIN_ROLE`), so `revokeRole(PENDING_*, address(0))` can be used to clear the pending slot and propose a new holder.

> **Acknowledged**
>
> Yearn acknowledged this finding.

#024 Donated Idle Assets Can Block Tranche Deposits

Informational	Version 2	Acknowledged

In TrancheStrategy, `availableDepositLimit()` bounds only the requested deposit by the shared main-vault headroom, while the inherited deposit flow calls `deployFunds()` with the strategy's entire loose balance. `_deployFunds()` routes that balance through TrancheController's `depositFromTranche()` into the main vault, so pre-existing idle assets make the forwarded amount exceed the same limit that the requested deposit passed.

Anyone can create idle balance by transferring assets directly to the tranche, or, alternatively, idle balance can accumulate as per Withdraw Rounding Leaks Surplus Vault Assets to the Withdrawing Tranche. This can have the following effects:

1. A deposit at the advertised `maxDeposit()` can revert.
2. Idle balance at least equal to the remaining main-vault headroom blocks every positive deposit until capacity increases.

> **Acknowledged**
>
> Yearn acknowledged the finding.

8 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.