yAudit

May 2026

Prepared for
Yearn stYFI

Audited by
Panda
Watermelon

stYFI Teams PR

Smart Contract Security Assessment

Contents

1 Review Summary **2**
- 1.1 Protocol Overview . . . 2
- 1.2 Audit Scope . . . 2
- 1.3 Risk Assessment Framework . . . 2
  - 1.3.1 Severity Classification . . . 3
- 1.4 Key Findings . . . 3
- 1.5 Overall Assessment . . . 4

2 Audit Overview **4**
- 2.1 Project Information . . . 4
- 2.2 Audit Team . . . 4
- 2.3 Audit Timeline . . . 4
- 2.4 Audit Resources . . . 4
- 2.5 Critical Findings . . . 6
- 2.6 High Findings . . . 6
- 2.7 Medium Findings . . . 6
  - 2.7.1 `BonusDistributor.finalize_period` can permanently brick all calls to `claim` for the related period . . . 6
- 2.8 Low Findings . . . 7
  - 2.8.1 Registry migration can strand team funding state on the old distributor . . . 7
  - 2.8.2 `RevenueRecipient.sweep` of the main token breaks per-category budget invariant . . . 8
- 2.9 Gas Savings Findings . . . 10
  - 2.9.1 `RevenueOracle.convert` re-approves the vault on every call . . . 10
  - 2.9.2 `BonusPriceOracle.price()` recomputes current period . . . 11
- 2.10 Informational Findings . . . 11
  - 2.10.1 Incorrect access control comment in `BonusDistributor.finalize_period` . . . 11

1 Review Summary

1.1 Protocol Overview

Yearn's Staked YFI is a liquid governance and revenue sharing vault wrapping the YFI token. The reviewed PR implements team-based funding accounting and distribution on-chain.

1.2 Audit Scope

This audit covers 8 smart contracts totaling approximately 1144 lines of code across 3 days of review.

```
contracts
├── BonusDistributor.vy
├── BonusPriceOracle.vy
├── FundingDistributor.vy
├── RevenueOracle.vy
├── RevenueRecipient.vy
├── Team.vy
├── TeamAccountant.vy
└── TeamRegistry.vy
```

1.3 Risk Assessment Framework

1.3.1 Severity Classification

Severity	Description	Potential Impact
Critical	Immediate threat to user funds or protocol integrity	Direct loss of funds, protocol compromise
High	Significant security risk requiring urgent attention	Potential fund loss, major functionality disruption
Medium	Important issue that should be addressed	Limited fund risk, functionality concerns
Low	Minor issue with minimal impact	Best practice violations, minor inefficiencies
Undetermined	Findings whose impact could not be fully assessed within the time constraints of the engagement. These issues may range from low to critical severity, and although their exact consequences remain uncertain, they present a sufficient potential risk to warrant attention and remediation.	Varies based on actual severity
Gas	Findings that can improve the gas efficiency of the contracts.	Increased transaction costs
Informational	Code quality and best practice recommendations	Reduced maintainability and readability

Table 1: severity classification

1.4 Key Findings

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	0
Medium	1
Low	2
Informational	1

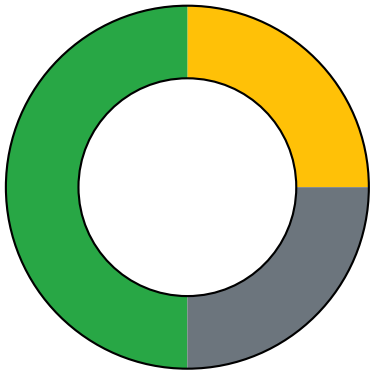

Figure 1: Distribution of security findings by impact level

1.5 Overall Assessment

PR#3 is correctly implemented with minor severity issues identified, triggerable either in low-probability scenarioes or by a trusted party's actions. While its core functionality has been found to function correctly and as expected, an incorrect value initialization in BonusDistributor's constructor was found to potentially lead to a bonus claim DoS, triggered by an unexpected division by zero. Furthermore, RevenueRecipient's sweep method presents an incorrectly handled edge case by which, if the contract's main token is swept, its per-category budget accounting system could break temporarily. While the highlighted issues require minor changes to be addressed, they remain tied to management-triggered actions and do not impact the system's main execution flows.

2 Audit Overview

2.1 Project Information

Protocol Name: Yearn stYFI
Repository: https://github.com/yearn/stYFI
Commit Hash: a921a4f4c58e49d1c1400e135fd76344ddceb531
Commit URL:
https://github.com/yearn/stYFI/tree/a921a4f4c58e49d1c1400e135fd76344ddceb531

2.2 Audit Team

Panda, Watermelon

2.3 Audit Timeline

The audit was conducted from May 4 to 6, 2026.

2.4 Audit Resources

- Code repositories and documentation

Category	Mark	Description
Access Control	Good	Adequate access control is implemented within the reviewed contracts, with a minor inconsistency identified between a method's documentation and its implementation.
Mathematics	Good	The mathemathical relations employed are simple and concise, although subject to issues under certain administrative actions.
Complexity	Good	The implemented system maintains a modular architecture with several components tightly aggregated.
Libraries	Good	No use of external libraries is made within the reviewed contracts. Some common functionality, like period calcuations and two-step ownership transfers could be implemented in standalone modules and imported in every contract in order to reduce code repetition.
Decentralization	Low	Management and operator actors are granted large power over the reviewed contracts' states.
Documentation	Average	The code is documented via contract-leven and method-level natspec documentation.
Testing and verification	Good	Adequate unit tests and fork tests have been implemented to ensure the contracts work as expected.

Table 2: Code Evaluation Matrix

2.5 Critical Findings

None.

2.6 High Findings

None.

2.7 Medium Findings

2.7.1 `BonusDistributor.finalize_period` can permanently brick all calls to `claim` for the related period

Technical Details

BonusDistributor's `constructor` initializes `growth_rate_cap` to `BPS_PRECISION`. However, `BonusDistributor.set_growth_rate_cap` enforces an upper bound of `BPS_PRECISION * 4 // 5` on any future update.
Within `BonusDistributor.finalize_period`, the cap is applied to the `growth_factor` and the bonus token price is adjusted accordingly. With the default `growth_rate_cap` set to `BPS_PRECISION`, the local `growth_rate_cap` evaluates to `PRECISION`, allowing `growth_factor` to be capped at `2 * PRECISION`. When this upper cap is hit, `adjusted_price` evaluates to `0`, and is then stored in `self.parameters[period]` as the period's `bonus_price`.
This condition is reached whenever the ratio between the current `ema_revenue` and the previous one is greater than or equal to `2`, which can occur in two scenarios:

1. A period reports zero revenue. Subsequent periods will set `prev_ema_revenue` to `1`, causing `growth_factor` to be saturated at the cap as soon as any non-trivial revenue is reported.
2. A revenue period whose smoothed revenue is at least double that of the previous one is encountered.

Once `adjusted_price` is stored as `0`, every subsequent call to `BonusDistributor.claim` for that period reverts due to division by zero. Because `claim` iterates sequentially through periods starting from `self.pending_claims[_team]` and updates the cursor only at the end of the loop, all teams whose claim cursor has not yet advanced past the affected period are unable to claim any bonus, including for any subsequent period.

Impact

Medium. All bonus claims for affected teams are bricked until management intervenes.

Recommendation

Initialize `growth_rate_cap` in the constructor to the same maximum value enforced by `set_growth_rate_cap` (80%):

```
@@ -102,7 +102,7 @@
     self.management = msg.sender
     self.operator = msg.sender
     self.smoothing_factor = PRECISION
-    self.growth_rate_cap = BPS_PRECISION
+    self.growth_rate_cap = BPS_PRECISION * 4 // 5
     self.bonus_factor = BPS_PRECISION
```

This ensures that `2 * PRECISION - growth_factor` cannot evaluate to `0` and that `adjusted_price` remains strictly positive for any oracle price greater than zero.

Developer Response

Fixed in 022243c.

2.8 Low Findings

2.8.1 Registry migration can strand team funding state on the old distributor

Technical Details

A Team contract resolves its funding distributor dynamically at call time by querying `self.registry.funding_distributor()`. When a registry is deprecated and a successor registry is configured, any user can permissionlessly call `TeamRegistry.migrate_team()` to update the team's registry pointer. The `Team.migrate()` function replaces `self.registry` with the successor registry address without preserving any reference to the original registry. After migration, the team's `claim_funding()` and `return_funding()` methods query the successor registry's distributor. However, funding approvals and cost accounting state remain stored in the original FundingDistributor. Each approval is indexed locally within its distributor (`approvals[_idx]`), and refund accounting depends on per-team, per-period cost records (`costs[team][period][token]`) that exist only in the original distributor.
If the successor registry points to a different FundingDistributor instance or reuses indices for unrelated approvals, the team loses its only callable path to interact with the old distributor. Direct calls to the old FundingDistributor are blocked because `FundingDistributor.claim()` and `FundingDistributor.refund()` require `msg.sender` to be the approval's team contract. The migration flow in `TeamRegistry.migrate_team()` verifies only that a successor exists and that `team_retirements[_team] > current_period + 1`; it performs no check for outstanding approvals or refundable balances on the old distributor.
The issue surfaces when management deprecates the old registry and configures the successor to use a different funding distributor while teams still have unclaimed approvals or refundable balances on the old distributor.

Impact

Low. Teams that are migrated with outstanding funding state on the old distributor lose access to unclaimed approved funding and the ability to refund previously claimed amounts. Because the Team contract's `claim_funding()` and `return_funding()` methods now resolve to the successor's distributor, any unclaimed approvals or refundable balances recorded in the original distributor become unreachable through the canonical team interface. This can result in lost funds for teams and stranded approval state in the original distributor.

Recommendation

During registry migration, management should keep the successor registry pointing to the same FundingDistributor until all teams have claimed or expired their old approvals and no longer need to refund balances. To enforce this operationally, consider adding a check in `migrate_team()` that blocks migration when the successor's `funding_distributor()` differs from the current registry's distributor:

```
@external
def migrate_team(_team: address):
    successor: address = self.successor
    assert successor != empty(address)
    assert self.team_retirements[_team] > self._period() + 1

    # Block migration if successor uses a different funding distributor
    old_distributor: address = staticcall self.funding_distributor()
    new_distributor: address = staticcall IRegistry(successor).funding_distributor()
    assert old_distributor == new_distributor, "Successor must use same distributor"

    extcall ITeam(_team).migrate(successor)
    log MigrateTeam(team=_team)
```

Alternatively, extend the Team contract to preserve a reference to the original distributor for legacy claims and refunds, allowing the team to resolve the correct distributor based on approval origin.

Developer Response

Ackwnoledged. Management should only initiate migrating to a new registry if there is very good reason to do so, and in that case it should take into account any existing unused approvals. But we give as much flexibility to management as possible, therefore we do not want to encode a requirement like that on-chain. Note that the same issue would occur by simply setting a new funding distributor on the existing registry, it would block teams from getting approved funding on the old distributor.

2.8.2 `RevenueRecipient.sweep` of the main token breaks per-category budget invariant

Technical Details

`RevenueRecipient.sweep` lets management withdraw any token held by the contract. When `_token == token.address`, it reduces `sum_balance` and `last_balance` by the swept

amount but does not adjust the `used` array, which tracks how much of each split has already been disbursed.
Per-category budgets in `_use_balance` are computed as `sum_balance * token_split[_i] // BPS_PRECISION`. Sweeping the main token therefore shrinks every budget proportionally while `used[_i]` stays put, breaking the invariant that the per-category available budget never exceeds `last_balance`. Two consequences follow:

1. Any category for which `used[_i]` exceeds its post-sweep budget is permanently bricked. `_use_balance` asserts `used <= budget` and reverts on any nonzero `_amount` for that category until enough revenue accrues to lift `sum_balance` back up.
2. The remaining drawable category can pass the `used <= budget` assert with an `_amount` greater than `last_balance`, then revert with an underflow on `self.last_balance -= _amount`.

PoC
Add the following test case to `tests/test_revenue_recipient.py` and execute it with `ape test tests/test_revenue_recipient.py::test_sweep_breaks_budget_invariant`:

```
def test_sweep_breaks_budget_invariant(project, deployer, alice, bob, genesis, token,
vault, recipient):
    distributor = project.RewardDistributor.deploy(genesis, vault, sender=deployer)
    recipient.set_reward_distributor(distributor, sender=deployer)
    recipient.set_operator(alice, sender=deployer)
    recipient.set_treasury(bob, sender=deployer)

    # seed the recipient with 100 UNIT of the main token (the vault)
    token.mint(deployer, 100 * UNIT, sender=deployer)
    token.approve(vault, 100 * UNIT, sender=deployer)
    vault.deposit(100 * UNIT, recipient, sender=deployer)

    # consume half of the treasury budget (cat 1)
    # splits = (8000, 1000, 1000) -> treasury budget = 10 UNIT
    recipient.to_treasury(5 * UNIT, sender=alice)
    assert recipient.sum_balance() == 100 * UNIT
    assert recipient.last_balance() == 95 * UNIT
    assert recipient.used(1) == 5 * UNIT

    # management sweeps 80 UNIT of the main token
    recipient.sweep(vault, 80 * UNIT, sender=deployer)
    assert recipient.sum_balance() == 20 * UNIT
    assert recipient.last_balance() == 15 * UNIT
    # used[] is unchanged --> invariant broken
    assert recipient.used(1) == 5 * UNIT

    # post sweep budgets: stYFI = 16 UNIT, treasury = 2 UNIT, yETH = 2 UNIT
    # treasury already used (5 UNIT) exceeds new budget (2 UNIT)
    # --> any nonzero treasury draw fails the `assert used <= budget` check
    with reverts():
        recipient.to_treasury(1, sender=alice)

    # stYFI's leftover budget (16 UNIT) exceeds last_balance (15 UNIT)
    # --> the operator passes the budget assert with _amount = 16 UNIT,
    #     then `last_balance -= 16 * UNIT` underflows on `15 - 16`
    with reverts():
        recipient.to_styfi_rewards(0, 16 * UNIT, sender=alice)
```

Impact

Low. Sweep is restricted to management, so this is a privileged foot-gun rather than an attack vector. The DoS on the over-budget categories is recoverable as additional revenue is deposited and `sum_balance` grows back above the existing `used` totals, but until that point those splits cannot disburse anything.

Recommendation

Scale the `used` array's elements down proportionally alongside `sum_balance` so that per-category budgets remain coherent with the disbursements already made.

Developer Response

The revenue token should only be sweeped out in an emergency, in which case it is okay to break the invariant as the contract wont be used afterwards. So instead we have added a kill switch in 8dee8cb, and only allow the revenue token to be sweeped if it has been activated.

2.9 Gas Savings Findings

2.9.1 `RevenueOracle.convert` re-approves the vault on every call

Technical Details

`RevenueOracle.convert` approves the vault for `_amount` on every call. The vault's `deposit` then consumes the full allowance, leaving the slot at zero. Each conversion therefore writes the allowance slot twice.
Since `RevenueOracle` is a stateless passthrough that does not hold funds across transactions, a one-time infinite approval at construction is equivalent and avoids the redundant writes.

Impact

Gas optimization.

Recommendation

```
 @deploy
 def __init__(_vault: address):
     vault = _vault
     token = staticcall IERC4626(_vault).asset()
     decimals: uint8 = staticcall IERC20Detailed(token).decimals()
     assert decimals <= 18
     scale = 10**(36 - convert(decimals, uint256))
+    assert extcall IERC20(token).approve(_vault, max_value(uint256),
default_return_value=True)

 @external
 def convert(_token: address, _amount: uint256):
```

```
12        assert _token == token
14        assert extcall IERC20(token).transferFrom(msg.sender, self, _amount,
   default_return_value=True)
15  -     extcall IERC20(token).approve(vault, _amount)
16        extcall IERC4626(vault).deposit(_amount, msg.sender)
```

Developer Response

Fixed as recommended in commit 6206111.

2.9.2 `BonusPriceOracle.price()` recomputes current period

Technical Details

`BonusPriceOracle.price()` caches the current period in current, but then calls `self._period()` again in the next line.

```
1  File: BonusPriceOracle.vy
2  81:     current: uint256 = self._period()
3  82:     assert _period < self._period()
```

Impact
Gas savings.

Recommendation

Use the cached value:

```
1  assert _period < current
```

Developer Response

Fixed in 9ac8e98.

2.10 Informational Findings

2.10.1 Incorrect access control comment in `BonusDistributor.finalize_period`

Technical Details

`BonusDistributor.finalize_period`'s docstring states:

`@dev Can only be called by the operator`

However, the actual access control check allows the call to succeed when `self.operator` is set to the zero address:

```
assert self.operator in [msg.sender, empty(address)]
```

When the operator role is disabled (i.e. set to `empty(address)` via `BonusDistributor.set_operator`), any account is able to call `finalize_period`.

Impact

Informational.

Recommendation

Either align the docstring with the implementation or restrict the access control check to a non-zero operator only.

Developer Response

Updated in 93ab181.

yAudit

stYFI Teams PR

Completed 2026-05-06