# Yearn v2 Vaults

Security Assessment

**April 30, 2021**

Prepared For:
Daniel Lehnberg | *Yearn Finance*
daniel.lehnberg@protonmail.com

Prepared By:
Gustavo Grieco | *Trail of Bits*
gustavo.grieco@trailofbits.com

Mike Martel | *Trail of Bits*
mike.martel@trailofbits.com

## Executive Summary

From April 12 to April 30, 2021, Yearn Finance engaged Trail of Bits to review the security of the Yearn v2 Vaults. Trail of Bits conducted this assessment over six person-weeks, with two engineers working from commit f8a5f1d4 on the `v1.0.0-audit.0` branch of the `yearn-vaults` repository, as well as from PR 273, at the request of the Yearn Finance team.

During the first week, we focused on gaining an understanding of the codebase and started to review the vault contract for the most common Vyper and Solidity flaws. During the second week, we primarily assessed the interactions between the vault and strategies, looking for flaws that would allow an attacker to bypass access controls or disrupt or abuse the contracts in unexpected ways. We dedicated the final week to a review of the system properties using our tools, in addition to an audit of the `BaseWrapper`, `yToken`, `AffiliateToken`, and `Registry` contracts.

Our review resulted in 19 findings ranging from high to informational severity, primarily involving missing or incorrect input validation. The most severe issues include the ability of a strategy owner to manipulate a debt ratio using a flash loan (TOB-YEARN-007), the risk of unexpected withdrawal blockages (TOB-YEARN-010), inconsistencies in debt calculations (TOB-YEARN-011), and a lack of consistency checks on upgrades to the `Registry` in the `yToken` contract (TOB-YEARN-012).

The interactions with ERC20 tokens are another area of concern; in particular, the value of `decimals` from ERC20 tokens should be carefully checked in Vyper (TOB-YEARN-005). Additionally, the use of deflationary or inflationary tokens can have unintended effects on the internal bookkeeping of the vault (TOB-YEARN-017). Finally, the `name`, `symbol`, and `decimals` values of the `yToken` contract can change during its lifetime, disrupting third-party interactions with it (TOB-YEARN-013).

Other issues stem from privileged users' and governance's ability to perform the privileged operations of changing important parameters of the vault and upgrading the strategies. Because of a lack of validation, these changes can lead to errors (TOB-YEARN-002, TOB-YEARN-004, TOB-YEARN-008, TOB-YEARN-009).

In addition to the security findings, we identified code quality issues not related to any particular vulnerability, which are discussed in Appendix C. Appendix D contains recommendations on interacting with arbitrary ERC20 tokens, which we would suggest reviewing before deploying any new vault. Finally, Appendix E details the expected security and correctness properties that we assumed to be true during this audit.

Although the code was battle-tested after a period of use by numerous actors, we still found high-severity issues that could be exploited by privileged users or result from a mistake in a transaction produced by a governance action. The use of Vyper is another area of concern, since its compiler is still in beta; as such, Yearn Finance should carefully review the security advisories, open issues, and current language limitations. To reduce the possibility of incorrect upgrades and the use of invalid parameters in the contracts, we recommend introducing additional checks and developing a detailed specification on how to perform these critical actions. Finally, the strategy contracts and their integrations with the vault might lead to vulnerabilities; however, these contracts were out of the scope of this review.

Trail of Bits recommends addressing the findings identified in this report and subsequently performing another security review that includes the actual strategies in use to ensure that there are no further issues.

*After the audit, Trail of Bits reviewed fixes for the issues identified in this report. The details of this review are provided in [Appendix F](#).*

# Project Dashboard

### Application Summary

| | |
|---|---|
| Name | Yearn v2 Vaults |
| Versions | f8a5f1d4, PR 273 |
| Types | Vyper, Solidity |
| Platform | Ethereum |

### Engagement Summary

| | |
|---|---|
| Dates | April 12–April 30, 2021 |
| Method | Full knowledge |
| Consultants Engaged | 2 |
| Level of Effort | 6 person-weeks |

### Vulnerability Summary

| | | |
|---|---|---|
| Total High-Severity Issues | 4 | ■ ■ ■ ■ |
| Total Medium-Severity Issues | 4 | ■ ■ ■ ■ |
| Total Low-Severity Issues | 7 | ■ ■ ■ ■ ■ ■ ■ |
| Total Informational-Severity Issues | 4 | ■ ■ ■ ■ |
| Total | 19 | |

### Category Breakdown

| | | |
|---|---|---|
| Data Validation | 11 | ■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ■ |
| Access Controls | 2 | ■ ■ |
| Undefined Behavior | 2 | ■ ■ |
| Timing | 2 | ■ ■ |
| Arithmetic | 1 | ■ |
| Error Reporting | 1 | ■ |
| Total | 19 | |

## Code Maturity Evaluation

In the table below, we review the maturity of the codebase and the likelihood of future issues. We rate the maturity of each area of control from strong to weak, or missing, and briefly explain our reasoning.

| Category Name | Description |
|---|---|
| Access Controls | **Satisfactory.** There were appropriate access controls on privileged operations performed by certain users and contracts. However, it was still possible for unauthorized users to successfully call a function that should be callable only by the strategies or governance (TOB-YEARN-004). |
| Arithmetic | **Moderate.** The contracts used safe arithmetic. No overflows were possible in areas where these functions were not used. However, certain arithmetic operations in corner cases could produce unexpected results (TOB-YEARN-003, TOB-YEARN-015). |
| Assembly Use | **Satisfactory.** The contracts used assembly only to implement EIP-1167 and to obtain the chain ID value. Note that in Solidity versions greater than 0.8, `block.chainid` should be used instead (Appendix C). |
| Centralization | **Satisfactory.** While the protocol relied on an owner to correctly deploy the initial contracts, privileged operations were handled either by special users (management) or through governance. |
| Contract Upgradeability | **Weak.** Each contract's upgradeability mechanisms allowed governance to change important parameters of the contract; however, updates to these values can be error-prone due to a lack of validation (TOB-YEARN-002, TOB-YEARN-004, TOB-YEARN-008). |
| Function Composition | **Satisfactory.** While most of the functions and contracts were organized and scoped appropriately, some of the functions were too long. Additionally, the interdependency of the vault and strategies during reporting and withdrawal operations is unclear. |
| Front-Running | **Moderate.** We identified several opportunities for users or strategy owners to take advantage of unconfirmed transactions through front-running (TOB-YEARN-019). |
| Monitoring | **Satisfactory.** The events produced by the smart contract code were capable of monitoring on-chain activity. |
| Specification | **Satisfactory.** Yearn Finance provided some documentation describing the functionality of the protocol and listing system |

| | |
|---|---|
| | invariants. However, there was no formal specification available to review. |
| Testing & Verification | **Satisfactory.** The contracts included a sufficient number of unit tests and a small number of more advanced testing techniques such as fuzzing. However, there is room for improvement in the test coverage, as described in Appendix C. Specifically, additional integration tests would provide full coverage of complex contract interactions. |

## Engagement Goals

We sought to answer the following questions:

- Do the system components have appropriate access controls?
- Is it possible to manipulate the system by front-running transactions?
- Is it possible for participants to steal or lose tokens or shares?
- Are there any circumstances under which arithmetic errors can affect the system?
- Can participants perform denial-of-service or phishing attacks against any of the system components?
- Can flash loans negatively affect the system's operation?
- Does the arithmetic for liquidations and shutdown bookkeeping hold?

Issues in the deployed strategies were outside the scope of this assessment.

## Coverage

The engagement focused on the following components:

- **Vault contract:** This contract holds an underlying token and allows users to interact with the Yearn Finance ecosystem through strategies connected to the vault. Vaults can have an unlimited number of strategies. Through a manual and automated review, we assessed the soundness of the arithmetic, the accuracy of the bookkeeping operations, and the access controls that ensure that the functionality can be invoked only by the appropriate core contracts. Additionally, we focused on the vault's interactions with other components of the ecosystem, as the vault's behavior largely depends on properties beyond its control.

- **BaseStrategy abstract contract:** This contract implements all of the functionalities required in interactions with the vault contract. This contract should be inherited, and the abstract methods should be implemented such that the strategy performs as expected. We conducted a manual and automated review of operations that could result in unexpected behavior during interactions with the vault contract. We also looked for instances in which privileged operations could be problematic. Finally, we focused on understanding how the behavior of a strategy could impact system integrity.

- **yToken, AffiliateToken, and BaseWrapper contracts:** These contracts define an ERC20-standard token used to deposit, hold, and withdraw user shares from a vault. Using a manual and automated review, we assessed the conformance of the tokens with ERC20 standards, ensuring that the tokens would not allow for undefined or unexpected behavior; we also reviewed the contracts' interactions with the vault API.

- **Registry:** This contract stores pointers to all current and former versions of the deployed vaults. Using primarily manual techniques, we looked for instances in which privileged operations were not appropriately protected and checked the correctness of vault releases.

- **Access controls:** Many parts of the system exposed privileged functionalities that should be invoked only by other system contracts. We reviewed these functionalities to ensure that they could be triggered only by the intended components and did not contain unnecessary privileges that could be abused.

- **Arithmetic:** We reviewed the calculations for logical inconsistencies, rounding issues, and scenarios in which reverts caused by overflows could disrupt the use of the protocol.

Deployed strategies and off-chain code components were outside the scope of this assessment.

## Automated Testing and Verification

Trail of Bits used automated testing techniques—specifically Echidna, a smart contract fuzzer—to enhance coverage of certain areas of the vault contracts. Echidna can rapidly test security properties via malicious, coverage-guided test case generation. We used Echidna to test the expected system properties, simulating interactions between a user, a strategy, and the vault.

We developed and implemented nine Echidna properties related to the vault contract:

| Property | Result |
|---|---|
| If the preconditions are met, a vault deposit operation should never revert. | PASSED |
| If the preconditions are met, a withdrawal from a vault should never revert. | PASSED |
| `totalDebt` is less than or equal to `totalAssets()`. | PASSED |
| The `debtRatio` of each strategy is less than or equal to the `debtRatio` of the vault. | PASSED |
| The `debtRatio` of the vault is less than or equal to `MAX_BPS`. | PASSED |
| For all strategies, if the vault is initialized, a call to `creditAvailable` will never revert. | PASSED |
| For all strategies, if the vault is initialized, a call to `expectedReturn` will never revert. | PASSED |
| If the vault is initialized, a call to `pricePerShare` will never revert. | PASSED |
| If the vault is initialized, a call to `maxAvailableShares` will never revert. | PASSED |

## Recommendations Summary

This section aggregates all the recommendations made during the engagement. Short-term recommendations address the immediate causes of issues. Long-term recommendations pertain to the development process and long-term design goals.

### Short Term

❑ **Disallow the use of `0x0` as a recipient of `deposit` functions.** This will prevent users from burning shares without decreasing the total supply. TOB-YEARN-001

❑ **Disallow the use of `0x0` and the contract address as `rewards` addresses and properly document these corner cases.** Disallowing the use of these addresses will prevent unexpected reverts. TOB-YEARN-002

❑ **Document the vault initialization process to ensure that the deployer's first deposit is large enough to prevent corner cases in which users do not receive shares after making deposits.** TOB-YEARN-003

❑ **Add an `assert` after the `decimals` call to validate that the value returned by `decimals` is less than 256.** This check will ensure that ERC20 tokens return correct data types. TOB-YEARN-005

❑ **Clearly document the vaults' intended security properties and the potential pitfalls in their access control mechanisms.** This will facilitate testing and verification of the security properties. TOB-YEARN-006

❑ **Revise the penalty computation to avoid using `_totalAssets()`, which depends on the balance of the vault.** This will prevent attackers from manipulating token balances. TOB-YEARN-007

❑ **Clearly document the fact that `setwithdrawalQueue` does not check for duplicated strategies so that management and governance will be aware of this issue before performing reordering operations.** This will ensure that the `withdrawalQueue` behaves as expected when used by other vault functions. TOB-YEARN-008

❑ **Consider replacing the strategy migration process with a simpler approach in which old strategies are retired and newer ones are added.** This safer, more predictable alternative would prevent the many potential pitfalls of the existing migration system. TOB-YEARN-009

❑ **Clearly document withdrawal-related properties to make users of the vaults aware of them.** This will ensure that users have confidence in the vaults. TOB-YEARN-010

❑ **Modify the behavior of `debtOutstanding` to account for the total vault debt.** That way, the debt of a given strategy will be correct regardless of whether it is calculated by `debtOutstanding` or `creditAvailable`. TOB-YEARN-011

❑ **Clearly document the expected process for updating a vault registry and consider adding more consistency checks or forcing a cache rebuild.** Taking these steps will increase users' confidence that the registry will operate as anticipated after an update. TOB-YEARN-012

❑ **Develop clear documentation for users and third-party contracts on the behavior of the `name`, `symbol`, and `decimals` functions.** This will ensure that other contracts and off-chain systems behave as expected in the event of an ERC20 property change. TOB-YEARN-013

❑ **Document the expected behavior of strategies when reporting losses to vaults, including the timelines for repaying excess debt.** This will assure users that the strategies are acting within expected parameters. TOB-YEARN-014

❑ **Ensure that when the performance fee or strategy queue is changed, the contract checks that neither the vault `performanceFee` nor any strategy-specific `performanceFee` exceeds the limit of 100% of the total fee amount.** Otherwise, strategies may be unable to report back to the vault, which could impact user confidence. TOB-YEARN-015

❑ **Either add a minimum requirement for management fees or clearly document the corner case in which these fees can be eliminated to make sure that it is not inadvertently triggered.** This will ensure that appropriate fees are paid. TOB-YEARN-016

❑ **Clearly document how vaults work to inform users that they are not designed to use certain ERC20 tokens.** That way, in the event that a vault is created with an inappropriate token, users will be able to make informed investment decisions. TOB-YEARN-017

❑ **Do not merge PR 273 without making further changes and performing another review.** These steps will ensure that the new code does not introduce any security or correctness issues. TOB-YEARN-018

❑ **Review the incentives created by pricing and reporting mechanisms as well as the governance control structures to ensure that opportunities for front-running are**

**understood and documented.** This review can inform future efforts to develop new features and can help mitigate potential pitfalls. TOB-YEARN-0019

## Long Term

❑ **Review the list of global invariants to make sure there is no way to break them.** This will also ensure that the contracts' behavior is known to developers and integrators and that they work as expected. TOB-YEARN-001, TOB-YEARN-002, TOB-YEARN-003

❑ **Review the access controls of every function and make sure that the documentation is correct and complete.** That way, developers will be able to detect cases in which necessary access controls are not enforced. TOB-YEARN-004

❑ **Ensure that all external data used in a Vyper contract is validated so that type mismatches will be prevented or appropriately handled.** This will prevent unexpected issues, especially when interoperating with Solidity-based contracts. TOB-YEARN-005

❑ **Ensure that the mechanisms for accessing contract functionalities are sound, complete, and compliant with the defined security properties.** That way, the properties will be testable and verifiable. TOB-YEARN-006

❑ **Review the specification on incentives and penalties to make sure that they do not depend on variables that can be manipulated by users.** This will help identify potential issues in the code. TOB-YEARN-007

❑ **Review the security and correctness properties and use Echidna or Manticore to test them.** TOB-YEARN-008

❑ **Review the current vault functionality to identify high-risk operations and simplify its use as much as possible.** TOB-YEARN-009

❑ **Review how the system invariants affect the usability of the contract and properly document corner cases so that users will know what to expect if a contract interaction is blocked.** This will increase user confidence in the system. TOB-YEARN-010

❑ **Clarify the expected properties of a vault's debt-related behavior; for example, consider requiring that the total debt of a vault remain below the amount warranted by its `debtRatio`.** Additionally, document these properties to inform users of the potential variance in vault risk management parameters. TOB-YEARN-011

❑ **Review the proper way for governance to update critical values, and document related corner cases to make sure that users will know what to expect when they**

**manage their contracts.** This will prevent governance from making costly or difficult-to-repair mistakes. TOB-YEARN-012

❑ **Consider introducing additional logic into the vault contract to enable clawbacks of excess debt at the time that losses are reported.** This will provide a mechanism for enforcing vault debt and risk properties and could inspire user confidence in the system. TOB-YEARN-014

❑ **Ensure that all contract invariants are documented and enforced in the code where possible to mitigate any issues.** This will simplify the testing and verification of properties. TOB-YEARN-015

❑ **Ensure that assets flow from users to strategies and privileged users as expected so that fees will be paid when necessary.** This will prevent actors in the system from depriving other actors of the fees owed to them. TOB-YEARN-0016

❑ **Review the Token Integration Checklist and implement its recommendations to make sure that ERC20 tokens used by the vault behave as expected.** TOB-YEARN-013, TOB-YEARN-017

❑ **Document assumptions and invariants that can influence vault behavior and thoroughly test mechanisms that may have unintended effects, such as locking profits.** This will increase confidence in the mechanisms and help prevent the introduction of new vulnerabilities. TOB-YEARN-0018

❑ **Carefully monitor the blockchain to detect front-running attempts and to identify the most problematic front-running scenarios**. TOB-YEARN-0019

## Findings Summary

| # | Title | Type | Severity |
|---|---|---|---|
| 1 | Shares are indirectly transferable to 0x0 | Data Validation | Low |
| 2 | Use of zero or contract address as rewards address can block fee computations | Data Validation | Low |
| 3 | Division rounding may affect issuance of shares | Arithmetic | Medium |
| 4 | revokeStrategy function can be error-prone | Undefined Behavior | Low |
| 5 | Vault initialize function does not validate ERC20 decimals | Data Validation | Informational |
| 6 | Vault deposits can bypass guest list deposit limits | Access Controls | Informational |
| 7 | Strategy owner can reduce or bypass loss penalty | Data Validation | High |
| 8 | setWithdrawalQueue allows for duplicated strategies | Data Validation | Low |
| 9 | Strategy migrations can be problematic and should be avoided | Access Controls | High |
| 10 | Large withdrawals can block other users from making withdrawals | Error Reporting | Medium |
| 11 | Current debt calculations can differ depending on context | Data Validation | Medium |
| 12 | Registry cache is not verified when registry address is updated | Data Validation | High |
| 13 | name, symbol, and decimals functions can change during the lifetime of yToken | Undefined Behavior | Medium |

|  |  |  |  |
|---|---|---|---|
| 14 | A strategy declaring a loss can keep excess debt | Data Validation | Informational |
| 15 | Performance fees can exceed 100% | Data Validation | Informational |
| 16 | Management fees can be avoided | Timing | Low |
| 17 | Vault should not use inflationary or deflationary ERC20 tokens | Data Validation | High |
| 18 | PR 273 introduces multiple issues | Data Validation | Low |
| 19 | Front-running opportunities | Timing | Low |

## 1. Shares are indirectly transferable to 0x0

Severity: Low
Type: Data Validation
Target: `Vault.vy`

Difficulty: Low
Finding ID: TOB-YEARN-001

**Description**

Users can bypass the transfer protections to send shares to the `0x0` address, burning them.

Users can deposit funds into the Yearn vaults to obtain shares. The shares are ERC20 compatible and can be transferred freely, except to the address of the contract or the `0x0` address:

```
@internal
def _transfer(sender: address, receiver: address, amount: uint256):
    # See note on `transfer()`.

    # Protect people from accidentally sending their shares to bad places
    assert not (receiver in [self, ZERO_ADDRESS])
    self.balanceOf[sender] -= amount
    self.balanceOf[receiver] += amount
    log Transfer(sender, receiver, amount)


@external
def transfer(receiver: address, amount: uint256) -> bool:
    self._transfer(msg.sender, receiver, amount)
    return True
```

*Figure 1.1: A transfer implementation without comments*

However, shares can be transferred to the `0x0` address during the initial deposit if the recipient is specified as `0x0`:

```
 @internal
 def _issueSharesForAmount(to: address, amount: uint256) -> uint256:
     # Issues `amount` Vault shares to `to`.
     # Shares must be issued prior to taking on new collateral, or
     # calculation will be wrong. This means that only *trusted* tokens
     # (with no capability for exploitative behavior) can be used.
     shares: uint256 = 0
     # HACK: Saves 2 SLOADs (~4000 gas)
     totalSupply: uint256 = self.totalSupply
     if totalSupply > 0:
         # Mint amount of shares based on what the Vault is managing overall
         # NOTE: if sqrt(token.totalSupply()) > 1e39, this could potentially revert
         precisionFactor: uint256 = self.precisionFactor
         shares = precisionFactor * amount * totalSupply / self._totalAssets() /
precisionFactor
     else:
         # No existing shares, so mint 1:1
         shares = amount

     # Mint new shares
     self.totalSupply = totalSupply + shares
     self.balanceOf[to] += shares
     log Transfer(ZERO_ADDRESS, to, shares)

     return shares

@external
@nonreentrant("withdraw")
def deposit(_amount: uint256 = MAX_UINT256, recipient: address = msg.sender) -> uint256:
    …

    # Issue new shares (needs to be done before taking deposit to be accurate)
    # Shares are issued to recipient (may be different from msg.sender)
    # See @dev note, above.
    shares: uint256 = self._issueSharesForAmount(recipient, amount)

    # Tokens are transferred from msg.sender (may be different from _recipient)
    self.erc20_safe_transferFrom(self.token.address, msg.sender, self, amount)

    return shares  # Just in case someone wants them
```

*Figure 1.2: Part of the `deposit` implementation without comments*

This corner case allows a user to emit a `Transfer` event that looks like a deposit (because the sender of the event is `0x0`) but also looks like a burn (because the destination is `0x0`).

**Exploit Scenario**
Eve deposits funds into a vault using `0x0` as the recipient, effectively burning shares. However, the total supply is not updated, and events that look like a burn are emitted, which could cause unexpected behavior in off-chain components.

**Recommendations**
Short term, disallow the use of `0x0` as a recipient of `deposit` functions.

Long term, review the list of global invariants to make sure there is no way to break them.

## 2. Use of zero or contract address as rewards address can block fee computations

Severity: Low
Difficulty: Low
Type: Data Validation
Finding ID: TOB-YEARN-002
Target: `Vault.vy`

**Description**

If governance tries to disable the payment of rewards by setting the recipient's address to `0x0` or the address of the contract, it can block the fee computation.

Users can deposit funds into the Yearn vaults to obtain shares. The shares are ERC20 compatible and can be transferred freely, except to the address of the contract or the `0x0` address:

```
@internal
def _transfer(sender: address, receiver: address, amount: uint256):
    # See note on `transfer()`.

    # Protect people from accidentally sending their shares to bad places
    assert not (receiver in [self, ZERO_ADDRESS])
    self.balanceOf[sender] -= amount
    self.balanceOf[receiver] += amount
    log Transfer(sender, receiver, amount)


@external
def transfer(receiver: address, amount: uint256) -> bool:
    self._transfer(msg.sender, receiver, amount)
    return True
```

*Figure 2.1: A transfer implementation without comments*

Certain important operations in the vault, such as the computation of fees, can trigger a transfer of shares:

```
 @internal
def _assessFees(strategy: address, gain: uint256) -> uint256:
    # Issue new shares to cover fees
    # NOTE: In effect, this reduces overall share price by the combined fee
    # NOTE: may throw if Vault.totalAssets() > 1e64, or not called for more than a year
    ...
        # NOTE: Governance earns any dust leftover from flooring math above
        if self.balanceOf[self] > 0:
            self._transfer(self, self.rewards, self.balanceOf[self])
    return total_fee
```

*Figure 2.2: Part of the `assessFees` implementation*

Governance can set the `self.rewards` address using `setRewards`:

```
@external
def setRewards(rewards: address):
    """
    @notice
        Changes the rewards address. Any distributed rewards
        will cease flowing to the old address and begin flowing
        to this address once the change is in effect.

        This will not change any Strategy reports in progress, only
        new reports made after this change goes into effect.

        This may only be called by governance.
    @param rewards The address to use for collecting rewards.
    """
    assert msg.sender == self.governance
    self.rewards = rewards
    log UpdateRewards(rewards)
```

*Figure 2.3: Part of the `deposit` implementation without comments*

However, governance can set the reward address to any address, including `0x0` or the address of the contract.

**Exploit Scenario**
Governance decides to disable the payment of rewards and sets the reward address to either the `0x0` address or that of the contract. As a result, when the contract tries to transfer rewards to the `rewards` address, the fee computation will revert, preventing the vault from working properly.

**Recommendations**
Short term, disallow the use of `0x0` and the contract address as `rewards` addresses and properly document these corner cases.

Long term, review the list of global invariants to make sure there is no way to break them.

## 3. Division rounding may affect issuance of shares

Severity: Medium
Difficulty: High
Type: Arithmetic
Finding ID: TOB-YEARN-003
Target: `Vault.vy`

**Description**
Users might not receive shares in exchange for their deposits if the total asset amount is manipulated by another user through a large "donation."

The `_issueSharesForAmount` function calculates the amount of shares to be issued:

```
@internal
def _issueSharesForAmount(to: address, amount: uint256) -> uint256:
    # Issues `amount` Vault shares to `to`.
    # Shares must be issued prior to taking on new collateral, or
    # calculation will be wrong. This means that only *trusted* tokens
    # (with no capability for exploitative behavior) can be used.
    shares: uint256 = 0
    # HACK: Saves 2 SLOADs (~4000 gas)
    totalSupply: uint256 = self.totalSupply
    if totalSupply > 0:
        # Mint amount of shares based on what the Vault is managing overall
        # NOTE: if sqrt(token.totalSupply()) > 1e39, this could potentially revert
        precisionFactor: uint256 = self.precisionFactor
        shares = precisionFactor * amount * totalSupply / self._totalAssets() /
precisionFactor
    else:
        # No existing shares, so mint 1:1
        shares = amount

    # Mint new shares
    self.totalSupply = totalSupply + shares
    self.balanceOf[to] += shares
    log Transfer(ZERO_ADDRESS, to, shares)

    return shares
```
*Figure 3.1: A transfer implementation without comments*

Essentially, the number of shares that a user will receive depends on the total supply, the total amount of assets, and the amount of deposited assets. The total asset amount is computed by the following code:

```
@view
@internal
def _totalAssets() -> uint256:
    # See note on `totalAssets()`.
    return self.token.balanceOf(self) + self.totalDebt
```
*Figure 3.2: A `revokeStrategy` implementation without comments*

However, if a user makes a large “donation” to the vault, increasing the total supply, other users may not receive shares after making deposits.

**Exploit Scenario**

1. Alice deploys a Yearn vault. She makes a minimal initial deposit to obtain one unit of shares, so the `totalSupply` is one.
2. Eve wants to block Alice's vault, so she transfers a large number of tokens to the vault address, increasing the `totalAssets` value without increasing the `totalSupply`.
3. A user calls `deposit` in order to obtain shares. Because his deposit is smaller than Eve’s, he does not receive any shares, even though the transaction succeeds. As a result, the vault has more `totalAssets`, but the same number of shares (one).

If more users deposit small amounts, they will not receive any shares in return. The problem will get worse until a user with enough money makes a large deposit.

**Recommendations**

Short term, document the vault initialization process to ensure that the deployer’s first deposit is large enough to prevent corner cases in which users do not receive shares after making deposits.

Long term, review the list of global invariants to make sure there is no way to break them.

## 4. revokeStrategy function can be error-prone

Severity: Low
Difficulty: Low
Type: Undefined Behavior
Finding ID: TOB-YEARN-004
Target: `Vault.vy`

**Description**
The function to revoke strategies may cause errors because of issues in its implementation and documentation.

Yearn vaults require strategies to generate yields. A strategy can be added or removed and, in certain cases, revoked:

```
@external
def revokeStrategy(strategy: address = msg.sender):
    assert msg.sender in [strategy, self.governance, self.guardian]
    self._revokeStrategy(strategy)

@internal
def _revokeStrategy(strategy: address):
    self.debtRatio -= self.strategies[strategy].debtRatio
    self.strategies[strategy].debtRatio = 0
    log StrategyRevoked(strategy)
```

*Figure 4.1: A `revokeStrategy` implementation without comments*

The `revokeStrategy` implementation has three issues:

- Any address, even that of a non-strategy sender, can successfully call it. While this call should not cause a state change, a `StrategyRevoke` event with an unexpected value will be emitted.
- It does not remove the revoked strategy from the withdrawal queue. While this is intended behavior, it should be clearly explained in the documentation so that callers will not forget to remove strategies if necessary.
- The documentation incorrectly states that a strategy "will only revoke itself during emergency shutdown." In actuality, a strategy can do that only in "emergency exit mode."

**Exploit Scenario**
Eve repeatedly calls `revokeStrategy` with different addresses to generate confusing events. Users notice the events and, in a panic, sell their shares, thinking that all the vault's strategies were suddenly revoked.

**Recommendations**
Short Term
- Prevent senders that are not strategies from calling `revokeStrategy`.
- Clearly define when a revoked strategy should be removed from the withdrawal queue.
- Correct the part of the documentation that details when a strategy can revoke itself.

Long term, review the access controls of every function and make sure that the documentation is correct and complete.

## 5. Vault initialize function does not validate ERC20 decimals

Severity: Informational
Difficulty: High
Type: Data Validation
Finding ID: TOB-YEARN-005
Target: `Vault.vy`

**Description**
The function to initialize a vault uses the `decimals` function from the ERC20 interface.
A vault needs to be initialized with a proper precision factor, which depends directly on the number of decimals a token reports.

```
self.decimals = DetailedERC20(token).decimals()
if self.decimals < 18:
  self.precisionFactor = 10 ** (18 - self.decimals)
else:
  self.precisionFactor = 1
```

*Figure 5.1: The part of the initialize function that sets the precision factor based on `decimals`*

However, the `decimals` value returns a `uint8`. Because this is not a data type native to Vyper, the value will be stored as `uint256`.

**Exploit Scenario**
Eve creates an ERC20 token that returns a `decimals` value greater than 255. Functions that depend on this value and external integrations that expect the value to be within the range of `uint8` may exhibit unexpected or undefined behavior.

**Recommendations**
Short term, add an `assert` after the `decimals` call to validate that the value returned by `decimals` is less than 256.

Long term, ensure that all external data used in a Vyper contract is validated so that type mismatches will be prevented or appropriately handled.

**References**
- Vyperlang/vyper advisory, VVE-2020-0001: Interfaces returning integer types less than 256 bits can be manipulated if uint256 is used

## 6. Vault deposits can bypass guest list deposit limits

Severity: Informational
Difficulty: Low
Type: Access Controls
Finding ID: TOB-YEARN-006
Target: `Vault.vy`

**Description**
A user can easily bypass the deposit limits defined by the guest list by making multiple calls.

When a user of a vault contract calls the deposit function, a check ensures that the user is authorized to make a deposit of the given amount.

```
# Ensure deposit is permitted by guest list
if self.guestList.address != ZERO_ADDRESS:
    assert self.guestList.authorized(msg.sender, amount)
```

*Figure 6.1: The part of the vault deposit function that uses `guestList.authorized`*

However, the user's `GuestList` function is only a view function and will not necessarily update deposit limits between the user's successive calls to `deposit`.

```
interface GuestList:
    def authorized(guest: address, amount: uint256) -> bool: view
```

*Figure 6.2: The `GuestList` interface definition*

This may allow a user to make successive deposits at or below the deposit limit, a practice that may contravene the intended security properties.

**Exploit Scenario**
Eve wants to deposit 100 tokens into a vault, but the vault's `GuestList` has set a deposit limit of 50 tokens on her address. By calling the deposit function twice with 50 tokens each time, Eve is able to deposit 100 tokens.

**Recommendations**
Short term, clearly document the vaults' intended security properties and the potential pitfalls in their access control mechanisms.

Long term, ensure that the mechanisms for accessing contract functionalities are sound, complete, and compliant with the defined security properties.

## 7. Strategy owner can reduce or bypass loss penalty

Severity: High
Difficulty: Medium
Type: Data Validation
Finding ID: TOB-YEARN-007
Target: `Vault.vy`

**Description**

The vaults penalize strategies that report losses, but a strategy owner can reduce or bypass a penalty using a flash loan.

The strategy owner should call `report` on his or her strategies, which will then call the report function in the vault specifying gains and losses:

```
@external
def report(gain: uint256, loss: uint256, _debtPayment: uint256) -> uint256:
    ...
    # Only approved strategies can call this function
    assert self.strategies[msg.sender].activation > 0
    # No lying about total available to withdraw!
    assert self.token.balanceOf(msg.sender) >= gain + _debtPayment

    # We have a loss to report, do it before the rest of the calculations
    if loss > 0:
        self._reportLoss(msg.sender, loss)
    ...
```

*Figure 7.1: The header of the `report` function in the vault*

If a strategy reports losses, its debt ratio should be reduced. The following formula calculates the amount of that reduction:

```
 @internal
 def _reportLoss(strategy: address, loss: uint256):
     # Loss can only be up the amount of debt issued to strategy
     totalDebt: uint256 = self.strategies[strategy].totalDebt
     assert totalDebt >= loss
     self.strategies[strategy].totalLoss += loss
     self.strategies[strategy].totalDebt = totalDebt - loss
     self.totalDebt -= loss

     # Also, make sure we reduce our trust with the strategy by the same amount
     debtRatio: uint256 = self.strategies[strategy].debtRatio
     precisionFactor: uint256 = self.precisionFactor
     ratio_change: uint256 = min(precisionFactor * loss * MAX_BPS / self._totalAssets() /
precisionFactor, debtRatio)
     self.strategies[strategy].debtRatio -= ratio_change
     self.debtRatio -= ratio_change
```

*Figure 7.2: The `_reportLoss` function*

The computation of the debt ratio change depends on the total asset amount, which is calculated using the balance of the vault. However, a strategy owner can temporarily increase this value by making a series of calls to do the following in a single transaction:

1. Take out a flash loan
2. Make a deposit
3. Call `report`
4. Make a withdrawal
5. Repay the flash loan

Since the value of `_totalAssets()` will increase (without exceeding the deposit limit), the `ratio_change` amount can be reduced, even to zero, allowing the user to avoid the debt change.

**Exploit Scenario**
Alice is a user interested in investing in a vault. Eve owns a strategy in that vault. An audit of Eve's strategy finds that it works as expected, but it is very risky. Eve calls `report` to disclose her losses but uses a flash loan to exploit the `reportLoss` function, avoiding the debt penalty.

Alice checks the debt ratio of Eve's strategy and finds it satisfactory. She decides to buy shares, but because Eve manipulated the debt ratio, she underestimates the risk.

**Recommendations**
Short term, revise the penalty computation to avoid using `_totalAssets()`, which depends on the balance of the vault.

Long term, review the specification on incentives and penalties to make sure that they do not depend on variables that can be manipulated by users. Use Echidna or Manticore to test them.

## 8. setWithdrawalQueue allows for duplicated strategies

Severity: Low
Difficulty: High
Type: Data Validation
Finding ID: TOB-YEARN-008
Target: `Vault.vy`

**Description**
The function to set the order of the withdrawal queue does not check for duplicated strategies, which can force a vault into an invalid state.

Once strategies have been added to a vault, they will automatically be included in the withdrawal queue.

```
 @external
def addStrategy(
    strategy: address,
    debtRatio: uint256,
    minDebtPerHarvest: uint256,
    maxDebtPerHarvest: uint256,
    performanceFee: uint256,
):
    ...
    log StrategyAdded(strategy, debtRatio, minDebtPerHarvest, maxDebtPerHarvest,
performanceFee)

    # Update Vault parameters
    self.debtRatio += debtRatio

    # Add strategy to the end of the withdrawal queue
    self.withdrawalQueue[MAXIMUM_STRATEGIES - 1] = strategy
    self._organizeWithdrawalQueue()
```

*Figure 8.1: Part of the `addStrategy` function*

The vault's management or governance can reorder a queue using the `setWithdrawalQueue` function:

```
 @external
 def setWithdrawalQueue(queue: address[MAXIMUM_STRATEGIES]):
     assert msg.sender in [self.management, self.governance]
     # HACK: Temporary until Vyper adds support for Dynamic arrays
     for i in range(MAXIMUM_STRATEGIES):
         if queue[i] == ZERO_ADDRESS and self.withdrawalQueue[i] == ZERO_ADDRESS:
             break
         assert self.strategies[queue[i]].activation > 0
         self.withdrawalQueue[i] = queue[i]
     log UpdateWithdrawalQueue(queue)
```

*Figure 8.2: The `setWithdrawalQueue` function*

However, this function does not check whether the elements in the queue are duplicated, which would break an important invariant of the vault.

**Exploit Scenario**
Alice is the manager of a vault. During a call to `setWithdrawalQueue`, she includes the same strategy twice by mistake. As a result, users trying to withdraw their shares could experience unexpected reverts, since the duplicated strategy will cause the strategy’s debt to decrease by more than the expected amount.

**Recommendations**
Short term, clearly document the fact that `setWithdrawalQueue` does not check for duplicated strategies so that management and governance will be aware of this issue before performing reordering operations.

Long term, review the security and correctness properties and use Echidna or Manticore to test them.

## 9. Strategy migrations can be problematic and should be avoided

Severity: High
Difficulty: High
Type: Access Controls
Finding ID: TOB-YEARN-009
Target: `Vault.vy`

**Description**
If they are not performed manually, strategy migrations can have serious effects on the security and correctness of the vaults. It is best to disallow them and to instead force strategies to be re-added.

Governance can trigger a strategy migration using the following function:

```
@external
def migrateStrategy(oldVersion: address, newVersion: address):
    """
    @notice
        Migrates a Strategy, including all assets from `oldVersion` to
        `newVersion`.

        This may only be called by governance.
    @dev
        Strategy must successfully migrate all capital and positions to new
        Strategy, or else this will upset the balance of the Vault.

        The new Strategy should be "empty" e.g. have no prior commitments to
        this Vault, otherwise it could have issues.
    @param oldVersion The existing Strategy to migrate from.
    @param newVersion The new Strategy to migrate to.
    """
    assert msg.sender == self.governance
    assert newVersion != ZERO_ADDRESS
    assert self.strategies[oldVersion].activation > 0
    assert self.strategies[newVersion].activation == 0

    strategy: StrategyParams = self.strategies[oldVersion]

    ...

    Strategy(oldVersion).migrate(newVersion)
    log StrategyMigrated(oldVersion, newVersion)

    for idx in range(MAXIMUM_STRATEGIES):
        if self.withdrawalQueue[idx] == oldVersion:
            self.withdrawalQueue[idx] = newVersion
            return  # Don't need to reorder anything because we swapped
```

*Figure 9.1: Part of the `migrateStrategy` function in a vault*

As the documentation states, governance should carefully perform the migration process to make sure that all assets are migrated to a new address and that the address has never interacted with the vault. If either of these conditions is not met, the vault can enter an invalid state.

Also note that governance can manually trigger a migration on a strategy itself instead of using a function in the vault:

```
     function migrate(address _newStrategy) external {
        require(msg.sender == address(vault) || msg.sender == governance());
        require(BaseStrategy(_newStrategy).vault() == vault);
        prepareMigration(_newStrategy);
        SafeERC20.safeTransfer(want, _newStrategy, want.balanceOf(address(this)));
    }
```

*Figure 9.2: The `migrate` function in the base strategy*

However, it is unclear why governance is able to call `migrate` outside of the vault context, as updates to vault values may not occur as intended, resulting in an incoherent vault state.

**Exploit Scenario**
A strategy is migrated to a newer version. However, because some of the invariants are not checked, the vault enters an invalid state. As a result, all the strategies have to be removed, and the vault should be redeployed.

**Recommendations**
Short term, consider replacing the strategy migration process with a simpler approach in which old strategies are retired and newer ones are added.

Long term, review the current vault functionality to identify high-risk operations and simplify its use as much as possible.

## 10. Large withdrawals can block other users from making withdrawals

Severity: Medium
Difficulty: Low
Type: Error Reporting
Finding ID: TOB-YEARN-010
Target: `Vault.vy`

**Description**
When a user calls the `withdraw` function, the amount withdrawn is determined using the total number of free tokens in the vault and the amount of `totalDebt` of each strategy in the `withdrawalQueue`.

```
        for strategy in self.withdrawalQueue:
            if strategy == ZERO_ADDRESS:
                break

            vault_balance: uint256 = self.token.balanceOf(self)
            if value <= vault_balance:
                break

            amountNeeded: uint256 = value - vault_balance

            amountNeeded = min(amountNeeded, self.strategies[strategy].totalDebt)
            if amountNeeded == 0:
                continue

            loss: uint256 = Strategy(strategy).withdraw(amountNeeded)
            withdrawn: uint256 = self.token.balanceOf(self) - vault_balance
```

*Figure 10.1: Part of the vault's withdrawal function detailing the maximum withdrawal amount for each strategy*

Since the vault can contain more assets than a user can withdraw, a large withdrawal can block other vault users from making subsequent withdrawals.

**Exploit Scenario**
Eve makes a large withdrawal from a vault, draining the available tokens from the vault and the strategies in the `withdrawalQueue`. Other users of the vault panic because they cannot retrieve funds.

**Recommendations**
Short term, clearly document withdrawal-related properties to make users of the vaults aware of them.

Long term, review how the system invariants affect the usability of the contract and properly document corner cases so that users will know what to expect if a contract interaction is blocked.

## 11. Current debt calculations can differ depending on context

Severity: Medium
Difficulty: High
Type: Data Validation
Finding ID: TOB-YEARN-011
Target: `Vault.vy`

**Description**
The vault extends credit to a strategy based on both the `debtRatio` of the strategy and the number of available excess tokens (as determined by the vault's `debtRatio`).

```
    # Start with debt limit left for the Strategy
    available: uint256 = strategy_debtLimit - strategy_totalDebt

    # Adjust by the global debt limit left
    available = min(available, vault_debtLimit - vault_totalDebt)

    # Can only borrow up to what the contract has in reserve
    # NOTE: Running near 100% is discouraged
    available = min(available, self.token.balanceOf(self))
```

*Figure 11.1: The part of the vault's `_creditAvailable` function that calculates the loan amount for a strategy*

A strategy can use the `debtOutstanding` function to calculate the number of excess tokens it has in order to pay down excess debt. However, the function does not take into account the current state of the vault. Instead, it uses only the strategy-specific `debtRatio`.

```
    strategy_totalDebt: uint256 = self.strategies[strategy].totalDebt

    if self.emergencyShutdown:
        return strategy_totalDebt
    elif strategy_totalDebt <= strategy_debtLimit:
        return 0
    else:
        return strategy_totalDebt - strategy_debtLimit
```

*Figure 11.2: The part of the vault's `_debtOutstanding` function that calculates the amount of debt a strategy can repay if its debt has exceeded its `debtLimit`*

The two calculations return different answers on how much credit a strategy should have. As a result, a strategy may be able to maintain an excessive amount of debt relative to its `debtRatio`, and the vault may extend more debt to the strategy than warranted by the total `debtRatio`.

**Exploit Scenario**
Eve observes that a vault has extended more debt to a strategy than stated as a result of the strategy's underestimating its debt payments. Eve is able to avoid incurring a loss on a withdrawal, as she withdraws funds from an overextended strategy rather than encountering a strategy with a pending loss later in the `withdrawalQueue`. Her withdrawal is fulfilled by the strategy with excess debt but no current losses.

**Recommendations**
Short term, modify the behavior of `debtOutstanding` to account for the total vault debt. That way, the debt of a given strategy will be correct regardless of whether it is calculated by `debtOutstanding` or `creditAvailable`.

Long term, clarify the expected properties of a vault's debt-related behavior; for example, consider requiring that the total debt of a vault remain below the amount warranted by its `debtRatio`. Additionally, document these properties to inform users of the potential variance in vault risk management parameters.

## 12. Registry cache is not verified when registry address is updated

Severity: High
Difficulty: High
Type: Data Validation
Finding ID: TOB-YEARN-012
Target: `BaseWrapper.sol`

**Description**
Updates to the registry address made by a wrapper contract will not be properly applied to the old registry cache. This can cause the contract to produce outdated results.

Wrapper contracts contain a registry pointer that determines the latest vault version. This version information is essential to computations of values such as total assets and the execution of operations such as deposits and withdrawals. To reduce the computational costs of operations that loop over the list of vaults, the wrapper provides a vault cache:

```
    function allVaults() public virtual view returns (VaultAPI[] memory) {
        uint256 cache_length = _cachedVaults.length;
        uint256 num_vaults = registry.numVaults(address(token));

        // Use cached
        if (cache_length == num_vaults) {
            return _cachedVaults;
        }

        VaultAPI[] memory vaults = new VaultAPI[](num_vaults);

        for (uint256 vault_id = 0; vault_id < cache_length; vault_id++) {
            vaults[vault_id] = _cachedVaults[vault_id];
        }

        for (uint256 vault_id = cache_length; vault_id < num_vaults; vault_id++) {
            vaults[vault_id] = VaultAPI(registry.vaults(address(token), vault_id));
        }

        return vaults;
    }

    function _updateVaultCache(VaultAPI[] memory vaults) internal {
        // NOTE: even though `registry` is update-able by Yearn, the intended behavior
        //       is that any future upgrades to the registry will replay the version
        //       history so that this cached value does not get out of date.
        if (vaults.length > _cachedVaults.length) {
            _cachedVaults = vaults;
        }
    }
```

*Figure 12.1: The `allVaults` and `_updateVaultCache` functions*

This cache is used when the vaults are accessed to prevent iterations over that list. The registry pointer can also be updated by the wrapper governance:

```
function setRegistry(address _registry) external {
    require(msg.sender == registry.governance());
    // In case you want to override the registry instead of re-deploying
    registry = RegistryAPI(_registry);
    // Make sure there's no change in governance
    // NOTE: Also avoid bricking the wrapper from setting a bad registry
    require(msg.sender == registry.governance());
}
```

*Figure 12.2: The `setRegistry` function*

However, when the registry is updated, there is no check that the updated registry is consistent with the previous one; nor is the current cache recreated.

**Exploit Scenario**
The governance of the wrapper contract updates the registry address to one that is not consistent with the current cache. This causes unexpected results when users ask for the total amount of assets or call the deposit or withdrawal functions.

**Recommendations**
Short term, clearly document the expected process for updating a vault registry and consider adding more consistency checks or forcing a cache rebuild.

Long term, review the proper way for governance to update critical values, and document related corner cases to make sure that users will know what to expect when they manage their contracts.

## 13. name, symbol, and decimals functions can change during the lifetime of yToken

Severity: Medium
Difficulty: Medium
Type: Undefined Behavior
Finding ID: TOB-YEARN-013
Target: `yToken.sol`

**Description**
The `yToken` contract allows users to update the values used by ERC20 functions (such as `name`, `symbol`, and `decimals`), which can produce unexpected behavior in Yearn vaults and third-party contracts.

The `yToken` contract adheres to the ERC20 standard when defining the `name`, `symbol`, and `decimals` functions:

```
function name() external view returns (string memory) {
    VaultAPI _bestVault = bestVault();
    return _bestVault.name();
}

function symbol() external view returns (string memory) {
    VaultAPI _bestVault = bestVault();
    return _bestVault.symbol();
}

function decimals() external view returns (uint256) {
    VaultAPI _bestVault = bestVault();
    return _bestVault.decimals();
}
```

*Figure 13.1: The implementation of the `name`, `symbol`, and `decimals` functions in the `BaseWrapper` contract*

It is very unlikely that users will expect these values to change over time, as that possibility is not mentioned in the ERC20 standard. If governance updates these values during the lifetime of the `yToken` contract, or if they are changed because of registry updates, it can lead to unexpected results.

For instance, when a vault is initialized with an ERC20 token, the `name`, `symbol`, and `decimals` values are fetched and saved in vault state variables. After that, they are not expected to change:

```
@external
def initialize(
    token: address,
    governance: address,
    rewards: address,
    nameOverride: String[64],
    symbolOverride: String[32],
    guardian: address = msg.sender,
):
    ...
    assert self.activation == 0  # dev: no devops199
    self.token = ERC20(token)
    if nameOverride == "":
        self.name = concat(DetailedERC20(token).symbol(), " yVault")
    else:
        self.name = nameOverride
    if symbolOverride == "":
        self.symbol = concat("yv", DetailedERC20(token).symbol())
    else:
        self.symbol = symbolOverride
    self.decimals = DetailedERC20(token).decimals()
    if self.decimals < 18:
      self.precisionFactor = 10 ** (18 - self.decimals)
    else:
      self.precisionFactor = 1
```

*Figure 13.2: Part of the vault initialization function*

If the `decimals` value changes, the vault should force a computation of the precision factor. Otherwise, the results of token deposits and withdrawals may be invalid.

**Exploit Scenario**
Alice deploys a `yToken` contract and deposits funds into it. Her token is used to initialize another Yearn vault. When the value of `decimals` in her `yToken` contract is updated, the value is not updated in the corresponding vault, leading to unexpected results.

**Recommendations**
Short term, develop clear documentation for users and third-party contracts on this unexpected property.

Long term, review the Token Integration Checklist and implement its recommendations to make sure that ERC20 tokens used by the vault behave as expected.

## 14. A strategy declaring a loss can keep excess debt

Severity: Informational
Difficulty: High
Type: Data Validation
Finding ID: TOB-YEARN-014
Target: `Vault.vy`

**Description**
Strategies are extended credit by the vault and can declare a gain or loss when reporting results. If a strategy declares a gain, the strategy must have a token balance of at least the amount of the gain, as the funds will be redistributed by the vault. However, if it incurs a loss and its credit is clawed back, the vault contract will not require it to repay its debt.

```
    assert self.token.balanceOf(msg.sender) >= gain + _debtPayment

    # We have a loss to report, do it before the rest of the calculations
    if loss > 0:
        self._reportLoss(msg.sender, loss)
```

*Figure 14.1: Part of the vault's reporting function, including initial conditions for token balances and subsequent loss-reporting logic*

As a result, a strategy may have access to more debt than the adjustment in `_reportLoss` entitles it to, and it may be able to avoid repayment until a subsequent reporting period. There is no logic in the vault contract itself to force a clawing back of the debt.

**Exploit Scenario**
Eve observes that a strategy has declared a loss and has more debt from the vault than it should. If this strategy appears earlier in the `withdrawalQueue`, Eve will be able to withdraw more funds from the strategy than she otherwise could.

**Recommendations**
Short term, document the expected behavior of strategies when reporting losses to vaults, including the timelines for repaying excess debt.

Long term, consider introducing additional logic into the vault contract to enable clawbacks of excess debt at the time that losses are reported.

## 15. Performance fees can exceed 100%

Severity: Informational
Difficulty: High
Type: Data Validation
Finding ID: TOB-YEARN-015
Target: `Vault.vy`

**Description**
When a vault is created, a vault-wide performance fee is set; a performance fee is also set for each strategy when it is added to the vault. The vault contract assumes that the sum of the vault performance fee and the performance fee of a given strategy will be at most 100%.

```
    if total_fee > gain:
        total_fee = gain
        # if total performance fee is greater than 100% then this will cause an underflow
        management_fee = gain - performance_fee - strategist_fee
```

*Figure 15.1: The part of the vault's `_assessFees` function that depends on a maximum performance fee*

The maximum performance fee is correctly validated in the `addStrategy` function.

```
assert performanceFee <= MAX_BPS - self.performanceFee
```

*Figure 15.2: The part of the vault's `addStrategy` function that validates the fee limit*

However, if governance sets the `performanceFee` of the vault after the vault has been initialized and a strategy has been added, it will be able to set the performance fee above the 100% limit. As a result, functions like `_assessFees` will revert when a strategy calls `report`.

```
    assert fee <= MAX_BPS
    self.performanceFee = fee
```

*Figure 15.3: Part of the vault's `setPerformanceFee` function*

**Exploit Scenario**
Eve is able to modify the `performanceFee` of a vault and sets it such that a strategy has a fee greater than 100%. Until the fee is corrected, all `report` calls made by that strategy will revert, disrupting the vault behavior.

**Recommendations**
Short term, ensure that when the performance fee or strategy queue is changed, the contract checks that neither the vault `performanceFee` nor any strategy-specific `performanceFee` exceeds the limit of 100%.

Long term, ensure that all contract invariants are documented and enforced in the code where possible to mitigate these types of issues.

## 16. Management fees can be avoided

Severity: Low
Difficulty: High
Type: Timing
Finding ID: TOB-YEARN-0016
Target: `Vault.vy`

**Description**
If a strategy owner can take out credit, use it to turn a profit, and then repay the debt in the same block, the management fee will be eliminated.

The management fee directly depends on two factors: the amount of total debt and the time between the calls to the `report` function (which affects the `lastReport` variable).

```
management_fee: uint256 = (
    precisionFactor *
    (
        (self.strategies[strategy].totalDebt - Strategy(strategy).delegatedAssets())
        * (block.timestamp - self.strategies[strategy].lastReport)
        * self.managementFee
    )
    / MAX_BPS
    / SECS_PER_YEAR
    / precisionFactor
)
```

*Figure 16.1: The computation of the management fee in the `assessFees` function*

However, if a strategy can take out credit, generate gains, and repay its debt in a single transaction, the management fee for those calls will be eliminated.

**Exploit Scenario**
Alice, the owner of a strategy, generates a profit by taking out credit and repaying it to the vault in a single transaction. As a result, the management fee is set to zero.

**Recommendations**
Short term, either add a minimum requirement for management fees or clearly document this corner case to make sure that it is not inadvertently triggered.

Long term, ensure that assets flow from users to strategies and privileged users as expected so that fees will be paid when necessary.

## 17. Vaults should not use inflationary or deflationary ERC20 tokens

Severity: High
Difficulty: High
Type: Data Validation
Finding ID: TOB-YEARN-017
Target: `Vault.vy`

**Description**
If a vault uses an ERC20 token that is inflationary, is deflationary, or has any kind of dynamic supply or balance, the internal bookkeeping of the vault may be severely affected.

The vault contract was not designed for tokens that can suddenly change the value returned by `balanceOf` or can transfer an unexpected number of tokens. When a user makes a deposit, the balance of the contract and the number of the tokens deposited are used in several operations:

```
@external
@nonreentrant("withdraw")
def deposit(_amount: uint256 = MAX_UINT256, recipient: address = msg.sender) -> uint256:
    ...
    # If _amount not specified, transfer the full token balance,
    # up to deposit limit
    if amount == MAX_UINT256:
        amount = min(
            self.depositLimit - self._totalAssets(),
            self.token.balanceOf(msg.sender),
        )
    else:
        # Ensure deposit limit is respected
        assert self._totalAssets() + amount <= self.depositLimit
    ...

    # Issue new shares (needs to be done before taking deposit to be accurate)
    # Shares are issued to recipient (may be different from msg.sender)
    # See @dev note, above.
    shares: uint256 = self._issueSharesForAmount(recipient, amount)

    # Tokens are transferred from msg.sender (may be different from _recipient)
    self.erc20_safe_transferFrom(self.token.address, msg.sender, self, amount)

    return shares  # Just in case someone wants them
```

*Figure 17.1: Part of the vault's deposit function*

The vault will check that a deposit does not exceed the deposit limit. The deposit function also assumes that the same set of tokens is transferred and received, which is not always the case when inflationary tokens are used.

The following widely used ERC20 tokens can cause issues:

- The Tether token can charge a fee for each transfer, meaning that users will receive fewer tokens than expected. Currently, this fee is set to zero, but if it is increased, Tether will become a deflationary token.

- Ampleforth executes daily rebases on contracts or increases the total supply (and therefore the balance of each user) so that its expected price will be pegged to USD 1.

**Exploit Scenario**
Alice deploys a vault and initializes it to use a token that has a dynamic supply. Users perform deposits and withdrawals. At some point, there is an error in the bookkeeping, so users do not receive the expected number of tokens for their shares.

**Recommendations**
Short term, clearly document this behavior to inform users that vaults are not designed to use the abovementioned types of ERC20 tokens.

Long term, review the Token Integration Checklist and implement its recommendations to make sure that ERC20 tokens used by the vault behave as expected.

## 18. PR 273 introduces multiple issues

Severity: Low
Difficulty: High
Type: Data Validation
Finding ID: TOB-YEARN-0018
Target: `Vault.vy`

**Description**
The Yearn Finance team asked Trail of Bits to review changes introduced in pull request 273. We uncovered several issues in our review.

The first issue involves the locked profit calculation, which occurred after the `lastReport` timestamp was updated to the current `block.timestamp`. As a result, calls to the new `_calculateLockedProfit` function always returned the total `lockedProfit` amount without any degradation over time. This issue was resolved in a subsequent commit.

The second issue also relates to locked profits; if they are not fully dispensed before a call to the `report` function, locked profits can be carried forward for significantly longer periods than anticipated.

Finally, the `withdraw` function can trigger an issue if a strategy reports gains and, during the locked profit degradation period, reports a loss of a similar magnitude. In such cases, a withdrawal will trigger the loss-reporting process, which will modify the value of `totalAssets`. However, `_shareValue` is also called from `withdraw`; if a strategy reports a significant loss, the existing locked profits may cause an underflow in the `freeFunds` calculation, in which case any withdrawal executed by a user will revert.

```
freeFunds: uint256 = self._totalAssets() - self._calculateLockedProfit()
```

*Figure 18.1: The computation of `freeFunds` in `_shareValue`, which is called from `withdraw`*

**Exploit Scenario**
Eve notices changes in how the `lockedProfit` amount is carried forward and realizes that because of the underflow in `freeFunds`, strategies that have incurred significant losses will be unable to report them. Eve withdraws her shares before the losses of these strategies can be realized, negatively affecting other shareholders.

**Recommendations**
Short term, do not merge PR 273 without making further changes and performing another review.

Long term, document assumptions and invariants that can influence vault behavior and thoroughly test mechanisms that may have unintended effects, such as locking profits. This will increase confidence in the mechanisms and help prevent the introduction of new vulnerabilities.

## 19. Front-running opportunities

Severity: Low
Difficulty: High
Type: Timing
Finding ID: TOB-YEARN-0019
Target: `Vault.vy`

**Description**
Several interactions between the vault and strategies create opportunities for transaction front-running.

Consider the following scenario: A vault user withdraws shares before some strategy reports a loss. The user may be penalized if that strategy appears earlier in the `withdrawalQueue`. However, the user could avoid this loss by withdrawing no more than the number of free tokens in the vault. This would allow the user to extract a higher value per share than warranted.

In another scenario, before a strategy calls `report`, a strategist or vault user can inject tokens into the vault. Then when `report` is called, the strategy will be extended credit and will receive a share of the newly injected tokens. If the vault has enough available tokens to compensate for the tokens injected into the strategy during the reporting period, a user will be able to inject additional tokens into it, rebalancing the vault at no cost by withdrawing the initial deposit.

Finally, if a strategy is found to be exploitable or is otherwise susceptible to manipulation during withdrawals, a user will be able to withdraw shares from the vault ahead of governance calls to remove the strategy from the `withdrawalQueue`.

**Exploit Scenario**
Eve is able to front-run transactions with vault contracts. She times her withdrawals such that they occur ahead of loss-reporting operations, maximizing her share value and evading the costs of losses.

**Recommendations**
Short term, review the incentives created by pricing and reporting mechanisms as well as the governance control structures to ensure that opportunities for front-running are understood and documented.

Long term, carefully monitor the blockchain to detect front-running attempts.

## A. Vulnerability Classifications

| **Vulnerability Classes** | |
|---|---|
| **Class** | **Description** |
| Access Controls | Related to authorization of users and assessment of rights |
| Auditing and Logging | Related to auditing of actions or logging of problems |
| Authentication | Related to the identification of users |
| Configuration | Related to security configurations of servers, devices, or software |
| Cryptography | Related to protecting the privacy or integrity of data |
| Data Exposure | Related to unintended exposure of sensitive information |
| Data Validation | Related to improper reliance on the structure or values of data |
| Denial of Service | Related to causing a system failure |
| Error Reporting | Related to the reporting of error conditions in a secure fashion |
| Patching | Related to keeping software up to date |
| Session Management | Related to the identification of authenticated users |
| Testing | Related to test methodology or test coverage |
| Timing | Related to race conditions, locking, or the order of operations |
| Undefined Behavior | Related to undefined behavior triggered by the program |

| **Severity Categories** | |
|---|---|
| **Severity** | **Description** |
| Informational | The issue does not pose an immediate risk but is relevant to security best practices or Defense in Depth. |
| Undetermined | The extent of the risk was not determined during this engagement. |
| Low | The risk is relatively small or is not a risk the customer has indicated is important. |

| | |
|---|---|
| Medium | Individual users' information is at risk; exploitation could pose reputational, legal, or moderate financial risks to the client. |
| High | The issue could affect numerous users and have serious reputational, legal, or financial implications for the client. |

| **Difficulty Levels** | |
|---|---|
| **Difficulty** | **Description** |
| Undetermined | The difficulty of exploitation was not determined during this engagement. |
| Low | The flaw is commonly exploited; public tools for its exploitation exist or can be scripted. |
| Medium | An attacker must write an exploit or will need in-depth knowledge of a complex system. |
| High | An attacker must have privileged insider access to the system, may need to know extremely complex technical details, or must discover other weaknesses to exploit this issue. |

## B. Code Maturity Classifications

| Code Maturity Classes | |
|---|---|
| **Category Name** | **Description** |
| Access Controls | Related to the authentication and authorization of components |
| Arithmetic | Related to the proper use of mathematical operations and semantics |
| Assembly Use | Related to the use of inline assembly |
| Centralization | Related to the existence of a single point of failure |
| Upgradeability | Related to contract upgradeability |
| Function Composition | Related to separation of the logic into functions with clear purposes |
| Front-Running | Related to resilience against front-running |
| Key Management | Related to the existence of proper procedures for key generation, distribution, and access |
| Monitoring | Related to use of events and monitoring procedures |
| Specification | Related to the expected codebase documentation |
| Testing & Verification | Related to the use of testing techniques (unit tests, fuzzing, symbolic execution, etc.) |

| Rating Criteria | |
|---|---|
| **Rating** | **Description** |
| Strong | The component was reviewed, and no concerns were found. |
| Satisfactory | The component had only minor issues. |
| Moderate | The component had some issues. |
| Weak | The component led to multiple issues; more issues might be present. |
| Missing | The component was missing. |

| | |
|---|---|
| Not Applicable | The component is not applicable. |
| Not Considered | The component was not reviewed. |
| Further Investigation Required | The component requires further investigation. |

# C. Code Quality Recommendations

The following recommendations are not associated with specific vulnerabilities. However, they enhance code readability and may prevent the introduction of vulnerabilities in the future.

**Vault**

- **Correct the misleading comment at the end of the `withdraw` function**. This comment implies that there is a withdrawal fee, which is incorrect. This will make the code easier to understand, maintain, and review.

**BaseStrategy**

- **Consider using `external` instead of `public` for functions**. Multiple functions (`isActive`, `tendTrigger`, `harvestTrigger`, and `allVaults`) could be declared `external` instead of `public` to help users save gas.

**AffiliateToken**

- **Consider adding a specific event to the `acceptAffiliate` function**. This function does not emit an event when the `affiliate` is updated, which would make the contract easier to monitor.
- **Consider requiring a non-zero address in `setAffiliate`.** It is expected that an affiliate could always have a non-zero address, so verifying that it is not set to zero can minimize errors in practice.
- **`_getChainId` should use `block.chainid` instead of assembly in Solidity 0.8 and above.** This will minimize the use of inline assembly, which can be error-prone.

**yToken**

- **Multiple functions (`transfer`, `approve`, `transferFrom`, `increaseAllowance`, `decreaseAllowance`, and `permitAll`) should be declared `external`.** That way, each function will consume a smaller amount of gas and will behave in a functionally identical manner.
- **The `transferFrom` function should assert that the appropriate allowance is available prior to calling `_transfer` and should subsequently call `_withdraw`.** This will result in gas savings in the event that an appropriate allowance does not already exist.

**General**

- **Consider using a consistent pragma statement in Solidity code.** This will enable users to understand the active versions of Solidity used in the project.
- **Consider expanding functional test coverage to include more edge cases, especially cases that test the invariants highlighted in Appendix E.** This will

further isolate edge cases, which will increase users’ confidence in the functionality of each component of the vault system.

- **Consider implementing additional multi-step integration tests that involve all of the relevant vault contract types.** This will help detect issues stemming from complex chained behavior that is difficult to isolate.

# D. Token Integration Checklist

The following checklist provides recommendations for interactions with arbitrary tokens. Every unchecked item should be justified, and its associated risks, understood. An up-to-date version of the checklist can be found in crytic/building-secure-contracts.

For convenience, all Slither utilities can be run directly on a token address, such as the following:

```
slither-check-erc 0xdac17f958d2ee523a2206206994597c13d831ec7 TetherToken
```

To follow this checklist, use the below output from Slither for the token:

```
- slither-check-erc [target] [contractName] [optional: --erc ERC_NUMBER]
- slither [target] --print human-summary
- slither [target] --print contract-summary
- slither-prop . --contract ContractName # requires configuration, and use of Echidna and
Manticore
```

## General Security Considerations

- ❏ **The contract has a security review.** Avoid interacting with contracts that lack a security review. Check the length of the assessment (i.e., the level of effort), the reputation of the security firm, and the number and severity of the findings.
- ❏ **You have contacted the developers.** You may need to alert their team to an incident. Look for appropriate contacts on blockchain-security-contacts.
- ❏ **They have a security mailing list for critical announcements.** Their team should advise users (like you!) when critical issues are found or when upgrades occur.

## ERC Conformity

Slither includes a utility, `slither-check-erc`, that reviews the conformance of a token to many related ERC standards. Use slither-check-erc to review the following:

- ❏ **`Transfer` and `transferFrom` return a boolean.** Several tokens do not return a boolean on these functions. As a result, their calls in the contract might fail.
- ❏ **The `name`, `decimals`, and `symbol` functions are present if used.** These functions are optional in the ERC20 standard and may not be present. If they are present, make sure that they cannot change over the lifetime of a token.
- ❏ **`Decimals` returns a `uint8`.** Several tokens incorrectly return a `uint256`. In such cases, ensure that the value returned is below 255.

- ❏ **The token mitigates the known ERC20 race condition.** The ERC20 standard has a known ERC20 race condition that must be mitigated to prevent attackers from stealing tokens.
- ❏ **The token is not an ERC777 token and has no external function call in `transfer` or `transferFrom`.** External calls in the transfer functions can lead to reentrancies.

Slither includes a utility, `slither-prop`, that generates unit tests and security properties that can discover many common ERC flaws. Use slither-prop to review the following:

- ❏ **The contract passes all unit tests and security properties from `slither-prop`.** Run the generated unit tests and then check the properties with Echidna and Manticore.

Finally, there are certain characteristics that are difficult to identify automatically. Conduct a manual review of the following conditions:

- ❏ **`Transfer` and `transferFrom` should not take a fee.** Deflationary tokens can lead to unexpected behavior.
- ❏ **Potential interest earned from the token is taken into account.** Some tokens distribute interest to token holders. This interest may be trapped in the contract if not taken into account.

### Contract Composition

- ❏ **The contract avoids unnecessary complexity.** The token should be a simple contract; a token with complex code requires a higher standard of review. Use Slither's `human-summary` printer to identify complex code.
- ❏ **The contract uses `SafeMath`.** Contracts that do not use `SafeMath` require a higher standard of review. Inspect the contract by hand for `SafeMath` usage.
- ❏ **The contract has only a few non-token-related functions.** Non-token-related functions increase the likelihood of an issue in the contract. Use Slither's contract-summary printer to broadly review the code used in the contract.
- ❏ **The token has only one address.** Tokens with multiple entry points for balance updates can break internal bookkeeping based on the address (e.g., balances[token_address][msg.sender] may not reflect the actual balance).

### Owner Privileges

- ❏ **The token is not upgradeable.** Upgradeable contracts may change their rules over time. Use Slither's `human-summary` printer to determine if the contract is upgradeable.
- ❏ **The owner has limited minting capabilities.** Malicious or compromised owners can abuse minting capabilities. Use Slither's `human-summary` printer to review minting capabilities, and consider manually reviewing the code.

- ❏ **The token is not pausable.** Malicious or compromised owners can trap contracts relying on pausable tokens. Identify pausable code by hand.
- ❏ **The owner cannot blacklist the contract.** Malicious or compromised owners can trap contracts relying on tokens with a blacklist. Identify blacklisting features by hand.
- ❏ **The team behind the token is known and can be held responsible for abuse.** Contracts with anonymous development teams or teams that reside in legal shelters require a higher standard of review.

## Token Scarcity

Reviews of token scarcity issues must be executed manually. Check for the following conditions:

- ❏ **The supply is owned by more than a few users.** If a few users own most of the tokens, they can influence operations based on the tokens' repartition.
- ❏ **The total supply is sufficient.** Tokens with a low total supply can be easily manipulated.
- ❏ **The tokens are located in more than a few exchanges.** If all the tokens are in one exchange, a compromise of the exchange could compromise the contract relying on the token.
- ❏ **Users understand the risks associated with a large amount of funds or flash loans.** Contracts relying on the token balance must account for attackers with a large amount of funds or attacks executed through flash loans.
- ❏ **The token does not allow flash minting.** Flash minting can lead to substantial swings in the balance and the total supply, which necessitate strict and comprehensive overflow checks in the operation of the token.

## E. Expected Strategy Properties

During this engagement, we reviewed the interactions of the strategies and vaults. We found that a number of system properties (listed below) must hold for all strategies to ensure the correct functioning of the vault. Since the strategy contract under review is only abstract, these properties should be verified before any additional strategy is deployed.

- When calling `Strategy.withdraw` or `Strategy.liquidatePosition`, `_liquidatedAmount + _loss <= _amountNeeded` always holds.
- When calling `Strategy.withdraw`, the sum of all the tokens received by the vault is less than or equal to `strategies[strategy].totalDebt`
- `strategies[strategy].totalDebt >= delegatedAssets()` always holds.
- Vault calls do not cause unexpected reverts.
- A strategy cannot take out a flash loan during a call to `report` or to any other aspect of the vault code.
- Multiple strategies cannot deposit tokens into the same vault. If a strategy deposits tokens into another strategy's vault, it should not have a circular dependency with any other vaults (e.g., strategy A from vault 1 deposits tokens into vault 2, and strategy B from vault 2 deposits tokens into vault 1).

Breaking any of these properties could cause the vault to exhibit unexpected behavior, with outcomes ranging from low-severity to catastrophic issues.

Additionally, the risk parameters set for both the individual strategies and the vault may not be respected until longer-term convergence occurs. This is especially evident in loss-reporting processes, as strategies may be able to hold excessive amounts of debt, or the vault may extend too much debt to a subset of active strategies. These types of events can upset the risk profile of a vault and affect fund reallocation. To bolster the integrity of the system, we recommend adding mechanisms for clawing back excess debt as well as invariants that will ensure consistency in the vaults' overall risk profile.

## F. Fix Log

After the assessment, the Yearn Finance team implemented fixes for the issues identified in this report. Trail of Bits reviewed each fix, working from two Git revisions (available here and here).

| ID | Title | Severity | Status |
|---|---|---|---|
| **01** | Shares are indirectly transferable to 0x0 | Low | Fixed |
| **02** | Use of zero or contract address as rewards address can block fee computations | Low | Fixed |
| **03** | Division rounding may affect issuance of shares | Medium | Fixed |
| **04** | revokeStrategy function can be error-prone | Low | Fixed |
| **05** | Vault initialize function does not validate ERC20 decimals | Informational | Fixed |
| **06** | Vault deposits can bypass guest list deposit limits | Informational | Fixed |
| **07** | Strategy owner can reduce or bypass loss penalty | High | Fixed |
| **08** | setWithdrawalQueue allows for duplicated strategies | Low | Fixed |
| **09** | Strategy migrations can be problematic and should be avoided | High | Fixed |
| **10** | Large withdrawals can block other users from making withdrawals | Medium | Fixed |
| **11** | Current debt calculations can differ depending on context | Medium | Fixed |
| **12** | Registry cache is not verified when registry address is updated | High | Fixed |
| **13** | name, symbol, and decimals functions can change during the lifetime of yToken | Medium | Fixed |
| **14** | A strategy declaring a loss can keep excess debt | Informational | Fixed |

|  |  |  |  |
|---|---|---|---|
| **15** | Performance fees can exceed 100% | Informational | Fixed |
| **16** | Management fees can be avoided | Low | Fixed |
| **17** | Vault should not use inflationary or deflationary ERC20 tokens | High | Fixed |
| **18** | PR 273 introduces multiple issues | Low | Not Fixed |
| **19** | Front-running opportunities | Low | Not Fixed |

For additional information on each fix, please refer to the detailed fix log on the following page.

## Detailed Fix Log

**Finding 1: Shares are indirectly transferable to 0x0**
Fixed. The codebase now checks that neither the contract address nor the 0x0 address is set as the recipient of a deposit.

**Finding 2: Use of zero or contract address as rewards address can block fee computations**
Fixed. The codebase now checks that neither the contract address nor the 0x0 address is set as the recipient of a reward payment.

**Finding 3: Division rounding may affect issuance of shares**
Fixed. The codebase now checks that the number of shares minted is greater than zero.

**Finding 4: revokeStrategy function can be error-prone**
Fixed. The codebase now checks that the revoked strategy has a `debtRatio` greater than zero.

**Finding 5: Vault initialize function does not validate ERC20 decimals**
Fixed. The codebase now checks that the ERC20 decimal value is less than 256.

**Finding 6: Vault deposits can bypass guest list deposit limits**
Fixed. The Yearn Finance team removed the guest list feature.

**Finding 7: Strategy owner can reduce or bypass loss penalty**
Fixed. The token balance is no longer used to compute the amount of a debt ratio reduction.

**Finding 8: setWithdrawalQueue allows for duplicated strategies**
Fixed. The Yearn Finance team implemented a check for duplicated strategies.

**Finding 9: Strategy migrations can be problematic and should be avoided**
Fixed. Governance no longer has the ability to migrate strategies directly. The Yearn Finance team also expanded the documentation on how to perform this delicate operation.

**Finding 10: Large withdrawals can block other users from making withdrawals**
Fixed. Yearn Finance developed additional documentation on the circumstances in which a large withdrawal can block other users from making withdrawals.

**Finding 11: Current debt calculations can differ depending on context**
Fixed. Yearn Finance addressed this issue by changing the way in which debt calculations are performed.

**Finding 12: Registry cache is not verified when registry address is updated**

Fixed. Yearn Finance developed additional documentation on how the registry should be migrated.

**Finding 13: name, symbol, and decimals functions can change during the lifetime of yToken**
Fixed. The codebase was changed to make sure the `name`, `symbol`, and `decimals` values are returned directly by the underlying ERC20 token.

**Finding 14: A strategy declaring a loss can keep excess debt**
Fixed. Yearn Finance developed additional documentation on how and when a strategy will repay its debt.

**Finding 15: Performance fees can exceed 100%**
Fixed. The Yearn Finance team limited the strategy and governance fees to less than 50% each.

**Finding 16: Management fees can be avoided**
Fixed. The codebase now checks that `_assessFees` is not called twice within the same block.

**Finding 17: Vault should not use inflationary or deflationary ERC20 tokens**
Fixed. The Yearn Finance team developed additional documentation on the use of inflationary and deflationary tokens in the vault.

**Finding 18: PR 273 introduces multiple issues**
Partially fixed. The Yearn Finance team partially fixed this issue by implementing a proper locked profit calculation. However, the fix could cause a zero-division revert, which would need to be handled appropriately.

**Finding 19: Front-running opportunities**
Not fixed. Yearn Finance stated the following:

> Broadly, it is not reasonable that Yearn can respond to any and all issues resulting from exploits of external integrations via connected strategies. The Vault and BaseStrategy integration was designed, tested and audited in order to prevent internal issues as much as possible, but there is still inherent risk for depositors in using the Vaults product.
>
> There might be some scenarios where a loss is incurred, and some lucky withdrawers notice that and take advantage of it by front running the loss and withdrawing. By placing a strategy with a potential loss at the front of the queue, the withdraw logic will fully push that loss onto the withdrawer, which should prevent this sort of front running of the loss. This feature does not block withdrawals, as this loss is opt in, and there might be scenarios where depositors might want to withdraw at some loss.
>
> The "free balance" in the Vault that is uninvested is available for withdrawing first, so there is some ability to bypass the loss by withdrawing only from the free

balance even in this situation. We do have the ability to shut down deposits into the Vault if the issue is truly severe, so there is a limit to how much withdrawers can benefit from this.

There are several things that are possible in theory as mentioned, however we have carefully considered these issues and designed passive or active mitigations for them as best as we could. To the best of our knowledge, there does not seem to be a general way to avoid these issues, so we have made these calls restricted to bonded Keep3rs (managed by our DevOps team), which use private mempools to avoid significant frontrunning potential and manipulation as much as possible. We also monitor for abuse of these mitigations and actively work on improving them over time.