PUBLIC

# Code Assessment

## of the Tokenized Strategy Smart Contracts

May 4, 2023

Produced for

by

# Contents

# 1 Executive Summary

Dear all,

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

Tokenized Strategy offers a framework for developers to easily create ERC-4626 compliant tokenized strategies by implementing only the strategy-specific logic, as it provides the core accounting functionality.

The most critical subjects covered in our audit are security and functional correctness.

During the review, no critical or high severity issues were uncovered. The report highlights a medium and a few low severity issues, one of which highlights a significant inaccuracy in the documentation. After the intermediate report, all issues have been addressed.

The general subjects covered are adherence to the implemented standards, code complexity and gas efficiency.

In summary, we find that the codebase provides a good level of security. We have to emphasize that the project reviewed is a template only, not an actual implementation of a strategy.

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

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

Sincerely yours,

ChainSecurity

## 1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

| | |
|---|---|
| Critical-Severity Findings | 0 |
| High-Severity Findings | 0 |
| Medium-Severity Findings | 1 |
| • Specification Changed | 1 |
| Low-Severity Findings | 5 |
| • Code Corrected | 4 |
| • Specification Changed | 1 |

# 2 Assessment Overview

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

## 2.1 Scope

The assessment was performed on the source code files inside the Tokenized Strategy repository based on the documentation files.

The scope consists of the two solidity smart contracts:

1. ./src/BaseTokenizedStrategy.sol
2. ./src/TokenizedStrategy.sol

**The contracts reviewed serve as building blocks only, they are not a working strategy.** They facilitate building an actual strategy: The abstract contract of BaseTokenizedStrategy is to be inherited, the code of TokenizedStrategy handling the core accounting logic is to be executed as delegatecall. Custom code can break core functionality, any strategy built on top should be audited separately.

The table below indicates the code versions relevant to this report and when they were received.

| V | Date | Commit Hash | Note |
|---|---|---|---|
| 1 | 21 April 2023 | 105c2f9340c05d661e07e36ff3acc9c75c99c0eb | Initial Version |
| 2 | 2 May 2023 | 71b8184d8bfcaeada74052900c58edfc0329e443 | After Intermediate Report |
| 3 | 3 May 2023 | c0da749dbe2ac0794c8086228a85c214db4045be | Updated Recipient Check |

For the solidity smart contracts, the compiler version `0.8.18` was chosen.

### *2.1.1 Excluded from scope*

Any other file not explicitly mentioned in the scope section. In particular tests, scripts, external dependencies, and configuration files are not part of the audit scope.

## 2.2 System Overview

This system overview describes the initially received version (Version 1) of the contracts as defined in the Assessment Overview.

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

The project reviewed provides a template that enables developers to easily create an ERC-4626 compliant tokenized strategy by only implementing the strategy-specific logic, as the core accounting functionality is already supplied.

In such a strategy users can mint shares of a strategy by depositing underlying tokens, with the expectation of receiving yield yet taking on the risk of potential loss. The strategy's share is an ERC-20

token non-transferrable to the strategy itself or zero address. Shares can be redeemed later to withdraw the underlying tokens. The strategy utilizes several mechanisms to mitigate price per share (pps) fluctuations and manipulation: (1) Internal accounting is used instead of `balanceOf()` to keep track of the strategy's debt and idle. (2) A profit locking mechanism locks profits by issuing new shares to the strategy itself that are slowly burnt over an unlock period. (3) In the event of losses or fees, the strategy will always try to offset them by burning locked shares it owns. The price per share is expected to decrease only when excess losses or fees occur upon processing a report. It will only increase when the profit is slowly unlocked as time goes by.

An immutable proxy pattern is utilized to abstract the important and risky functionalities away from the customized strategy. All customized strategies should inherit `BaseTokenizedStrategy` and override several simple functions that interact with the underlying yield source. It is expected that all the standard-compliant function calls will enter the fallback to delegatecall the `TokenizedStrategy` implementation for essential internal accounting. In case a strategy-specific operation is required during the execution, the implementation will callback to the customized strategy.

Note that the actual customized strategies implementations which will be built using this framework are unknown. They must adhere to the design rules, notably they must not interfere with the accounting and not overwrite the special storage of TokenStrategy. Otherwise functionality might be compromised.

### 2.2.1 BaseTokenizedStrategy

The `BaseTokenizedStrategy` is an abstract contract. Its state variables will be initialized by the `initialize` function, which is also executed after cloning a new instance of the strategy. Several strategy-specific functions are left to be overridden by the strategist:

- `invest()`, `_invest()` deposit funds into the yield source.
- `freeFunds()`, `_freeFunds()` withdraw funds from the yield source.
- `totalInvested`, `_totalInvested()` harvest all rewards, reinvests and returns a trusted and accurate amount of funds currently held by the Strategy.
- `tendThis()`, `_tend()` can be used by the management or the keeper to harvest and compound rewards, deposit idle funds, perform needed position maintenance or anything else that doesn't need a full report for. Note this call can be sandwiched if a swap is involved.
- `tendTrigger()` returns whether tend should be called by the management and the keeper or not.
- `availableDepositLimit()` returns the deposit limit for a user.
- `availableWithdrawLimit()` returns the withdraw limit for a user.

All the state-modifiable functions above are protected by a `onlySelf` modifier to avoid a malicious user bypassing the internal accounting. Namely none of them should be called directly on the `BaseTokenizedStrategy`, and they should only be invoked by a callback from the `TokenizedStrategy`. The customized strategy should follow this design to function in a secure and efficient way as expected. A strategist may add customized storage variables on demand as well. Due to the `StrategyData` residing at storage slots with high addresses, unintentional storage collision with variables of the custom strategy implementation are unlikely to happen.

### 2.2.2 TokenizedStrategy

The `TokenizedStrategy` contract implements the core logic including the ERC-4626 functionality which is common for all custom strategies. It is deployed only once and it's code is executed by the individual strategies using delegatecall. To prevent unintended storage conflicts with variables of the custom strategy, it stores all it's variable in a struct residing at a high storage address. This way, it prevents conflicts with variables of the custom implementation with high probability.

**Users Permissionless Entry Points**

The `deposit` and `mint` functions would transfer funds to the strategy and issue new shares to the user when the strategy is not shutdown and user's `maxDeposit` is not reached.

Upon `withdraw` and `redeem`, the strategy burns the shares within `maxRedeem` and transfers the funds to the user if `idle` is sufficient. Otherwise, a callback to `BaseTokenizedStrategy` is invoked to free funds from the yield source. Any discrepancy between the *idle* and the required assets is regarded as a loss taken by the withdrawer. A disproportionate share of unrealized loss may exist depending on the design of `_freeFunds()`. In addition, there is no slippage protection for `withdraw` and `redeem`.

Note that `deposit()` and `mint()` to a dead strategy does not revert (i.e. `shares=!0` but `assets==0`). In this case, the depositer's shares are diluted by the shares left immediately and lose part of the assets. And this contract can receive Ether without further logic to deal with it.

**Strategy Operations and Management**

The following are the most important permissioned entry points to operate the strategy, which can only be called by the keeper or the management.

`tend()` is called by the keeper or the management to `tend` the strategy if a custom `tendTrigger()` is implemented.

*report()*` is called by the keeper or the management to record all profits and losses since last report and charge fees accordingly:

- Protocol fee will be calculated based on the old total assets.
- An accurate account for all funds including those potentially airdropped is retrieved by a callback to `totalInvested()` on `BaseTokenizedStrategy`.
- The performance fee will be charged based on the profit. Newly locked shares will be issued to self if there is a net profit, otherwise, previously locked shares will be burnt to cover the loss and fees.
- The new profit unlocking schedule will be computed as a weighted average of the previously locked shares and newly locked shares.

The management role is the admin of the strategy and has privilege to call the following setters:

- `setManagement()` instantly transfers the management role to another address.
- `setKeeper()` sets the keeper role to another address.
- `setPerformanceFee()` sets the performance fee to be charged on a reported gains.
- `setPerformanceFeeRecipient()` sets a new address to receive performance fees.
- `setProfitMaxUnlockTime()` sets the time for profits to be unlocked over.
- `shutdownStrategy()` turns the strategy into shutdown mode irreversibly, where no further deposit is allowed.

A clone function is provided to create a new clone of the calling strategy as a minimal proxy and initialize it with the parameters passed. Note only the original strategy should be cloned as all the clones are purely a minimal proxy without the strategy-specific runtime code.

### 2.2.3 Roles and Trust Model

The framework reviewed is provided to build custom strategies. The strategist is trusted to respect the template design rules and to implement the required functionality correctly. Among others, the custom strategy implementation must not modify the variables in `StrategyData` nor interfere with the accounting in other ways, otherwise the internal accounting in `TokenizedStrategy` can be compromised.

The underlying token should be ERC-20 compliant without weird behaviors such as double entry points, rebase mechanism, transfer fees, irrational return values, high decimals, etc.

We assume the `management` and the `keeper` of the strategy are fully trusted to behave honestly and correctly at all times, and never act maliciously or against the interest of the system users.

# 3 Limitations and use of report

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

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

# 4 Terminology

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

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

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

| Likelihood | Impact | | |
|---|---|---|---|
| | High | Medium | Low |
| High | Critical | High | Medium |
| Medium | High | Medium | Low |
| Low | Medium | Low | Low |

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

# 5 Findings

In this section, we describe any open findings. Findings that have been resolved have been moved to the Resolved Findings section. The findings are split into these different categories:

- Design: Architectural shortcomings and design inefficiencies
- Correctness: Mismatches between specification and implementation

Below we provide a numerical overview of the identified findings, split up by their severity.

| | |
|---|---|
| Critical-Severity Findings | 0 |
| High-Severity Findings | 0 |
| Medium-Severity Findings | 0 |
| Low-Severity Findings | 0 |

# 6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Findings section.

Below we provide a numerical overview of the identified findings, split up by their severity.

| | |
|---|---|
| Critical-Severity Findings | 0 |
| High-Severity Findings | 0 |
| Medium-Severity Findings | 1 |

- Rogue Strategy Can Override Storage Specification Changed

| | |
|---|---|
| Low-Severity Findings | 5 |

- Initializing TokenizedStrategy Code Corrected
- Non ERC-4626 Compliant Functions Code Corrected
- Payable Fallback Functions Code Corrected
- Problematic Self-Minting When Fee Recipient Is the Contract Itself Code Corrected Risk Accepted
- Staticcall Specification Changed

## 6.1 Rogue Strategy Can Override Storage

Correctness Medium Version 1 Specification Changed

*CS-YTS-004*

Core accounting logic is done by the code of TokenizedStrategy which is executed as Delegatecall inside the context of the strategy.

The documentation states the following:

```
In order to limit the strategists need to think about their storage variables
all TokenizedStrategy specific variables are held within and controlled by the
TokenizedStrategy. A BaseStrategyData struct is help at a custom storage location
that is high enough that no normal implementation should be worried about hitting.

This means all high risk storage updates will always be handled by the
TokenizedStrategy, can not be overriden by a rogue or reckless strategist
and will be entirely standardized across every strategy deployed, no matter
the chain or specific implementation.
```

A rogue or reckless strategist can overwrite any storage slot, including those at the address `keccak256("yearn.base.strategy.storage") - 1` and subsequent addresses.

While a genuine strategy wouldn't do this and the concept to separate the storage ensures that with high probability a specific implementation is unlikely do to so by accident, a rogue or reckless strategies can do so intentionally.

---

**Specification changed:**

Yearn acknowledged this risk and corrected the specification.

## 6.2 Initializing TokenizedStrategy

Design | Low | Version 1 | Code Corrected

*CS-YTS-008*

The deployed instance of TokenizedStrategy is only intended to be used via Delegatecall by the custom strategies.

However the functions of the contract are also directly callable. For example the first caller of `TokenizedStrategy.init()` can initialize the contract. While this doesn't break it's intended use as base for the delegatecalls, it's not desirable.

It's worth noting that after initialization, deposits will still fail due to the callback to the `invest()` function.

Other functions may execute successfully, such as approvals, role assignments, and parameter updates.

---

**Code corrected:**

A constructor has been added to `TokenizedStrategy` which initializes the implementation with `_strategyStorage().asset=address(1)`. As a result, further direct calls to `initialize()` will revert.

## 6.3 Non ERC-4626 Compliant Functions

Correctness | Low | Version 1 | Code Corrected

*CS-YTS-007*

`maxMint` may revert due to an overflow in a calculation, however according to the specification this function must not revert. This may happen in an edge case the `availableDepositLimit` returns a large number and `pps<1`, `convertToShares` may overflow.

```
function maxMint(address _owner) public view returns (uint256 _maxMint) {
    _maxMint = IBaseTokenizedStrategy(address(this)).availableDepositLimit(
        _owner
    );
    if (_maxMint != type(uint256).max) {
        _maxMint = convertToShares(_maxMint);
    }
}
```

In case the strategy is in shutdown mode, no further deposit can be made. However, `maxDeposit()` may not return 0 when the strategy is shutdown.

The ERC-4626 specification however requires the function to return 0 in this case:

```
... if deposits are entirely disabled (even temporarily) it MUST return 0.
```

More informational, the ERC-4626 specification is loosely defined in these corner cases for these functions. Nevertheless we want to highlight the potentially unexpected amounts returned:

`previewRedeem()`: In case `totalAssets` is zero, the conversion is done at a 1:1 ratio. At this point either no shares exist (I) or the value of the existing shares has been dilluted to 0 (II). For (I) the returned value of 0 is appropriate. For (II) `previewRedeem()` does not revert while `redeem()` reverts; the specification reads:

```
MAY revert due to other conditions that would also cause redeem to revert.
```

`previewWithdraw()` returns the amount in a 1:1 exchange rate when `assets==0` but `shares!=0`. Again for non-zero values the amount returned may be misleading.

Strictly speaking the value returned is not breaking the specification but might be unexpected by the caller. The caller should be aware of this and any external system should exercise caution when integrating with these functions.

---

**Code corrected:**

A comment has been added to `availableDepositLimit` to alert the strategist of the potential overflow of `maxMint()` if the deposit limit is too large. In addition, `maxDeposit()` and `maxRedeem()` have been updated to return 0 when the strategy is shutdown. `previewWithdraw()` and `convertToShares()` have been adjusted to return 0 instead of `pps=1` in case `assets==0` but `supply>0`. Yearn also acknowledged the potential misleading non-zero return value of `previewRedeem()` if all shares are diluted to 0. Strategists and external systems are expected to be aware of these behaviors.

## 6.4 Payable Fallback Functions

Design | Low | Version 1 | Code Corrected

*CS-YTS-006*

The fallback function of `` `BaseTokenizedStrategy` `` is marked as `payable`. However, the code of the delegatecalled TokenizedStrategy contract doesn't feature any functionality able to receive Ether. Any such call with a non zero `msg.value` will revert.

Furthermore, there is a `receive()` function:

```
/**
 * We are forced to have a receive function do to
 * implementing a fallback function.
 *
 * NOTE: ETH should not be sent to the strategy unless
 * designed for within the Strategy. There is no defualt
 * way to remove eth incorrectly sent to a strategy.
 */
receive() external payable {}
```

There is no requirement to implement a receive function when incorporating a fallback function. In the absence of a receive function, plain Ether transfers would be handled by the fallback function, which then delegatecalls into the TokenizedStrategy. However, this would cause the call to revert since the contract does not support Ether reception. By including a receive function, the strategy can be enabled to accept Ether. As the comment states, Ether shouldn't be sent to the strategy unless the strategy is design for it.

For more information please refer to the Solidity documentation: https://docs.soliditylang.org/en/v0.8.18/contracts.html#receive-ether-function

---

**Code corrected:**

The payable modifier and the receive function have been removed to avoid unintentional Ether reception.

## 6.5 Problematic Self-Minting When Fee Recipient Is the Contract Itself

Correctness | Low | Version 1 | Code Corrected | Risk Accepted

*CS-YTS-005*

Transferring shares of the strategy to itself is prevented since it can interfere with the locked shares mechanism which guards against abrupt price per share increases. Unlike `_transfer()`, `_mint()` does not feature this restriction since it is intended to mint shares for this contract as part of the profit locking mechanism. An explicit check must be done in the function calling `_mint()`. While this is done in `_deposit()`, such a check isn't done on the fee recipients.

When the `performanceFeeRecipient` is set as the strategy itself, it becomes possible to mint additional shares to the strategy, which are not intended to be locked shares.

Once enough time passes and the `fullProfitUnlockDate` is reached, `_unlockedShares()` will treat the entire balance of this contract, including these additional shares, as unlocked shares.

```
if (_fullProfitUnlockDate > block.timestamp) {
    unchecked {
        unlockedShares =
            (S.profitUnlockingRate * (block.timestamp - S.lastReport)) /
            MAX_BPS_EXTENDED;
    }
} else if (_fullProfitUnlockDate != 0) {
    // All shares have been unlocked.
    unlockedShares = S.balances[address(this)];
}
```

Due to the presence of extra shares, there may be a sudden increase when querying the unlockedShares just before and right after the `fullProfitUnlockDate`.

This effect may have an impact whenever `_totalSupply()` is called and may influence the the price per share.

Additionally `process_report()` is affected. In case the `fullProfitUnlockDate` has already been reached these shares would simply get burned in `_burnUnlockedShares()`. Otherwise these shares will be considered as part of the `previouslyLockedShares` and are locked in the new locking period. Note as this is an increase of the previouslyLockedShares it will impact the calculation of and reduce the `newProfitLockingPeriod`:

```
// new_profit_locking_period is a weighted average between the remaining
    // time of the previously locked shares and the PROFIT_MAX_UNLOCK_TIME
    uint256 newProfitLockingPeriod = (previouslyLockedShares *
        remainingTime +
        sharesToLock *
        _profitMaxUnlockTime) / totalLockedShares;
```

The issue description focuses on the `performanceFeeRecipient` as fee recipient, in theory the same situation could arise if the `protocolFeesRecipient` is set as the strategy contract.

---

**Code corrected:**

An extra check has been added in `setPerformanceFeeRecipient()` as well as in `init()` which prevents setting the fee recipient to `address(this)`.

**Risk accepted:**

The `protocolFeeRecipient` is set once for all strategies by Yearn Governance and should not be an issue.

## 6.6 Staticcall

**Correctness** **Low** **Version 1** **Specification Changed**

*CS-YTS-009*

```
 * Using address(this) will mean any calls using this variable will lead
 * to a static call to itself. Which will hit the fallback function and
 * delegateCall that to the actual TokenizedStrategy.

ITokenizedStrategy internal TokenizedStrategy;
```

The comment says that using `address(this)` will result in a static call to itself, but the term "static call" might be misleading. In Ethereum, a "static call" typically refers to a STATICCALL, which is a read-only call that cannot modify the contract state. However, in this case, the comment seems to be referring to the fact that the call will simply be to the contract itself. Such calls can lead to state changes.

---

**Specification changed:**

Yearn has rephrased the comment to avoid misunderstandings. A legitimate strategist should not use this variable for state-changing calls.

# 7 Informational

We utilize this section to point out informational findings that are less severe than issues. These informational issues allow us to point out more theoretical findings. Their explanation hopefully improves the overall understanding of the project's security. Furthermore, we point out findings which are unrelated to security.

## 7.1 Uncovered Loss Not Visible in Reported Event

**Informational** **Version 1**

*CS-YTS-001*

An event `Reported` will be emitted after `report()` is called. If an uncovered loss has been realized, this crucial information won't be visible in the event. In case a net loss occurs, price per share decrease (pps) instantly. Revealing this in the event may be useful.

```
event Reported(uint256 profit,uint256 loss,uint256 performanceFees,uint256 protocolFees)
```

---

Yearn states:

```
The event is meant to match the Vaults event as close as possible and only reveal
the amounts determined within the report call. It should be expected that most
reports in strategies will be done after all shares have been unlocked since the
previous reports, and therefore any loss will cause a PPS decrease. Specific
strategies can use this functionalities if desired to offset losses but is not
normal behavior, simply extra functionality. PPS is not tracked on chain.
```

## 7.2 Use ADDRESS Instead of SLOAD

**Informational** **Version 1**

*CS-YTS-002*

`BaseTokenizedStrategy.initialize()` sets the storage variable `TokenizedStrategy` to the address of the executing context:

```
// Set instance of the implementation for internal use.
TokenizedStrategy = ITokenizedStrategy(address(this));
```

To call itself, the code of the BaseTokenizedStrategy and the custom strategy implementation would use this variable which results in an SLOAD operation. Note that opcode ADDRESS (in solidity `address(this)`) would return the same address (the address of the executing account) and is significantly cheaper.

---

Yearn states:

```
The setting of the `TokenizedStrategy` variable in initialization is
meant to make it as simple as possible for a strategist to access
readable data from the StrategyData struct so having an extra SLOAD is
```

```
worth the reduced complexity of not having to understand what is being
called, just that the variable will work.
```

## 7.3 tendTrigger

**Informational** **Version 1**

*CS-YTS-003*

The description of `TokenizedStrategy.tend()` reads:

```
* @dev Both 'tendTrigger' and '_tend' will need to be overridden
* for this to be used.
```

However this is not enforced in the code, where `tendTrigger()` has no effect on `tend()`, e.g. it could return `false` and `tend()` may still execute successfully.

---

Yearn states:

```
`tendtrigger` is only to be used off chain, by a keeper bot or management to
easily determine if tend should be called, not a requirement for it to be.
Tend is able to be called at any point even if the trigger does not say it
should.
```

# 8 Notes

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

## 8.1 Withdraw With Unrealized Loss

**Note** **Version 1**

In case there is an unrealized loss, the most vigilant users will come to withdraw funds directly from the `idle` to avoid the loss. As a result, the tardy users will take the unrealized loss. Besides, tardy users may take an unrealized loss in different ways depending on the actual implementation of `_freeFunds()`.

- If custom strategy implementation simply tries to free the funds from the yield source as closer to the requested amount as possible or simply reverts due to insufficient funds, the remaining funds will be withdrawn in a FCFS way where the last users will take all of the unrealized loss and get nothing back.
- If custom strategy implementation distributes the unrealized loss according to the accounting variables in `StrategyData`, then all tardy user will share the unrealized loss proportionally.

Different strategists may take different choices, whereas the vigilant users can always drain the `idle` regardless of the unrealized loss in both cases.

---

Yearn states:

```
So for the most part those types of decisions are to be left to the strategist to
determine what to do in _freeFunds(). The majority of strategies will likely simply
withdraw the amount requested, since its 1. not applicable and 2. would require a
lot more gas and code to check the actual current state and calculate the full
unrealized loss etc. Though if a strategy expects to have this be a common case
(like with an options strategy) that specific strategist can add whatever they
wish to _freeFunds. It is recommended that _freeFunds revert if losses would be
realized by temporary situations. Such as liquidity constraints, that are not
expected to last, rather than count it as a loss.

While its possible there are unrealized losses, normal behavior is to not account
for those in between reports, but rather losses are handled withdraw by withdraw.
Though that can lead to disproportionate amounts depending on when funds are
withdrawn its much cheaper and simpler considering its a non-issue for the majority
of strategies and the ones it is can choose how to deal with it.
```