PUBLIC

Code Assessment

of the V3 Vaults 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 V3 Vaults according to Scope to support you in forming an opinion on their security risks.

Yearn implements VaultsV3, an unopinionated ERC-4626 compliant system designed to distribute depositor funds into various strategies and manage accounting robustly. Depositors receive ERC-20 compliant shares that can be redeemed at any time.

The most critical subjects covered in our audit are security, functional correctness and the proper accounting of the assets and shares.

During the review, no critical or highly severe issues were uncovered. Two medium severity correctness issues have been found which have been resolved after the intermediate report.

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.

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	2
• Code Corrected	2
Low-Severity Findings	10
• Code Corrected	7
• Specification Changed	1
• Risk Accepted	1
• Acknowledged	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 V3 Vaults repository based on the documentation files.

The scope consists of the two vyper smart contracts:

1. ./contracts/VaultFactory.vy
2. ./contracts/VaultV3.vy

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

V	Date	Commit Hash	Note
1	11 April 2023	05fbd377a7778c660034f17c11b32e3767ff9166	Initial Version
2	2 May 2023	a53ed5878bbdd8d305e6a6bc3cbea05e8acd569a	After Intermediate Report
3	3 May 2023	953f8b663ed2658c9cc937d380e3b6beefdecd18	Updated Decimal Check

For the vyper smart contracts, the compiler version `0.3.7` 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.

Yearn implements an ERC-4626 compliant VaultV3 that functions as a secure and efficient debt allocator for an underlying ERC-20 compliant token. A factory contract is utilized to enable permissionless deployment of a vault. Users can mint shares of a vault by depositing underlying tokens, with the expectation of receiving passive yield yet taking on the risk of potential loss. The vault's share is simply an ERC-20 token which cannot be transferred to the vault itself or zero address. Shares can be redeemed later to withdraw the underlying tokens. ERC-4626 compliant strategies can be added to the vault to generate yield on the underlying tokens. The vault 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 vault's debt and idle. (2) A profit locking mechanism designed by V3 Vaults locks profits or accountant's refunds by issuing new shares to the vault itself that are slowly burnt over the an unlock period. (3) In the event of losses or fees, the vault 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, or a loss occurs upon force revoking a strategy. It will increase when the profit is slowly unlocked as time goes by.

2.2.1 Vault Factory

The `VaultFactory` contract implements a factory where all vaults are deployed in a permissionless manner. Any address can call `deploy_new_vault` to create a vault from blueprint by the `CREATE2` opcode. Several vaults can exist for one underlying token. There is a `governance` role that sets the protocol fee and the fee recipient. The transfer of `governance` role involves two phases: first, the `governance` assigns the `pending_governance`, and later, the `pending_governance` takes the initiative to accept the nomination.

2.2.2 VaultV3

The `VaultV3` contract implements all the logic for funds allocation, profits and losses reporting, fees assessment, and vault maintenance. Users can `deposit` funds to `mint` shares and `redeem` their shares to `withdraw` funds in a permissionless manner. Roles-specific functions are defined to manage the vault, rebalance the strategies and report the profits, loss and fees.

Users Permissionless Entry Points

The `deposit` and `mint` functions would transfer funds to the vault, increase the `total_idle`, and issue new shares to the user when the vault is not shutdown and `deposit_limit` is not reached.

Upon `withdraw` and `redeem`, the vault burns the shares and transfers the funds to the user if `total_idle` is sufficient. Otherwise, following steps will be taken to increase `total_idle`:

- The vault iterates through an array of `strategies` specified either by the user or the `queue_manager` to withdraw funds.
- In case the strategy has any unrealized loss since the last report, the user would bear part of it as well as the potential token transfer loss during withdraw from the strategy.

A redemption may not be successful if there are not enough funds available to be provided.

Roles

`role_manager` is the administrator set in constructor that controls all roles. The transfer of `role_manager` follows the same two-phase transition as `VaultFactory`. Different roles are assigned to separate the critical functionalities of vault management. Roles can be filled by EOA, smart contract like a multisig or a governance module that relays calls.

- `ADD_STRATEGY_MANAGER` can add strategies to the vault.
- `REVOKE_STRATEGY_MANAGER` can remove strategies from the vault.
- `FORCE_REVOKE_MANAGER` can force remove a strategy causing a loss.
- `ACCOUNTANT_MANAGER` can set the `accountant` module address that assesses fees and potentially refunds to the vault in case of a loss.
- `QUEUE_MANAGER` can set the `queue_manager` module address that can provide and override the withdraw queue.
- `REPORTING_MANAGER` calls report for strategies.
- `DEBT_MANAGER` adds and removes debt from strategies.
- `MAX_DEBT_MANAGER` can set the max debt for a strategy.
- `DEPOSIT_LIMIT_MANAGER` sets deposit limit for the vault.
- `MINIMUM_IDLE_MANAGER` sets the `minimum_total_idle` the vault should keep.
- `PROFIT_UNLOCK_MANAGER` sets the `profit_max_unlock_time`.
- `SWEEPER` can sweep tokens from the vault.

- EMERGENCY_MANAGER can shutdown vault in an emergency.

In addition, an open_role mechanism is used by the role_manager to turn a permissioned function open to public. The role_manager can set_open_role and close_open_role at its discretion.

Debt Operations

The following are the most important permissioned entry points to operate the vault, which can be called by bots or manually depending on periphery implementation:

update_debt() is called by the DEBT_MANAGER to either deposit into or withdraw from a strategy. The actual amount is bounded by the strategy's max_debt, maxDeposit and maxWithdraw and the vault's minimum_total_idle.

process_report() is called by the REPORTING_MANAGER to report profits or losses for individual strategies as well as charging fees:

- total_assets will be queried from the strategy and compared with the current_debt to compute the incoming gain and loss.
- Protocol and strategy fees are assessed and charged by minting shares to the corresponding recipients. The accountant may provide a total_refunds as newly locked shares to offset the loss.
- The vault will issue newly locked shares to itself if there is a profit. Users will bear the excess loss and fees only if they exceed the newly and previously locked shares.
- In the event of a profit, the profit releasing period will be updated by the weighted average of the remaining locked profits and the newly locked profits.

sweep() can be called by the SWEEPER to sweep the tokens from airdrop or sent by mistake. Only the tokens that are neither vault's shares nor strategies tokens can be swept.

add_strategy() can be called by the ADD_STRATEGY_MANAGER to add a new strategy with current_debt and max_debt set to 0, which forbids the allocation of funds at creation time.

revoke_strategy() can be called by the REVOKE_STRATEGY_MANAGER to revoke a strategy only when the debt of a strategy has been fully removed.

FORCE_REVOKE_MANAGER is expected to call force_revoke_strategy() on a faulty strategy only in an emergency. Because it revokes a strategy regardless of its current debt, which incurs loss to the users. If the force revoked vault is added back later, the previously lost debt will be treated as profits.

Emergency Operations

V3 Vaults designs several mechanisms to handle different emergency situations. Withdrawals and accounting are not paused or affected under any circumstances.

For the emergency of a single strategy:

- The MAX_DEBT_MANAGER can pause future allocation to the strategy by setting the strategy max_debt to 0.
- A strategy can be revoked by the REVOKE_STRATEGY_MANAGER or FORCE_REVOKE_MANAGER.

For the emergency of the vault:

- The MINIMUM_IDLE_MANAGER can set the minimum_total_idle to max(uint256), where the vault will request the debt back from strategies as well as stop new strategies from getting funds.
- The DEPOSIT_LIMIT_MANAGER can set the depositLimit to 0 which pauses future deposits.
- The EMERGENCY_MANAGER can turn the vault into shutdown mode irreversibly, where it acquires the DEBT_MANAGER role to remove debt from the strategies as soon as possible.

2.2.3 Roles and Trust Model

The `governance` of VaultFactory is fully trusted that is the `msg.sender` at deployment and can be transferred in the future. As a Vault is created in a permissionless way, we expect only the legit vault to be used by the users. Namely 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, unusually large supply, etc.

In addition, the `role_manager` and all other roles of a Vault are assumed trusted to behave honestly and correctly at all times. The strategies added to a vault are assumed to never act maliciously or against the interest of the system users.

2.2.4 Changes in Version 2

- The `QUEUE_MANAGER` can not force overwrite the user specified withdraw queue anymore, and it purely provides a default withdraw queue if the user does not specify one. Hence, users' withdrawal from preferred strategies cannot be paused by the `QUEUE_MANAGER`.
- The `max(uint256)` feature (i.e. the contract assumes the user wants to use all the balance) has been removed.
- The deposit limit is set to 0 when `shutdown_vault()` is called. In this case, the `maxDeposit()` will return 0 to comply with the standard, and the deposit limit can no longer be changed after shutdown.

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:

- Security: Related to vulnerabilities that could be exploited by malicious actors
- 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	2

5.1 Gain Exceeds Max_Debt

Correctness Low Version 1 Acknowledged

CS-YVV3-001

In contract `VaultV3`, any strategy gains upon `process_report()` will be reported by increasing the strategy's `current_debt` and the vault's `total_debt` regardless of the strategy's `max_debt` parameter. In this case, the debt of a strategy can exceed its upper bound.

Acknowledged:

Yearn states:

> `This is deemed acceptable if caused by profits. Since debt can be lowered at any time after by the DEBT_MANAGER.`

5.2 Reentrancy and `process_report()`

Security Low Version 1 Risk Accepted

CS-YVV3-002

`process_report()` can reenter functions of the Vault in the external call to `IAccountant(accountant).report()`. Note that these are trusted roles, however, if the accountant can dispatch a call from the `(FORCE_)REVOKE_STRATEGY_MANAGER` role, a strategy could be revoked during the process of reporting it, which breaks the correct execution flow.

Furthermore, similarly a strategy may enter into `proces_report()` while an update of its debt is in process (`update_debt()`). Roles are trusted to not misbehave, the smart contract implementation however does not prevent this scenario.

Risk accepted:

Yearn states:

> `Reentrancy was intentionally left off process_report() so that an accountant can reenter ‘deposit’ if need be to issue refunds. It is expected that the accountant never be set to a role other than accountant. And be given no other permissions.`

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	2

- Disproportional Unrealized Loss on Redemption Code Corrected
- Inconsistent Debt Accounting on Withdrawal From Strategies Code Corrected

Low-Severity Findings	8

- Add Self as a Strategy Code Corrected
- Incorrect Return Type of Decimals Code Corrected
- Incorrect Return Value Code Corrected
- Incorrect and Missing Specification Specification Changed
- Missing Event upon Role Change Code Corrected
- Non ERC-4626 Compliant Functions Code Corrected
- Unchecked Profit Max Unlock Time Code Corrected
- Unprotected Sweep Function Code Corrected

6.1 Disproportional Unrealized Loss on Redemption

Correctness Medium Version 1 Code Corrected

CS-YVV3-014

If `total_idle` is insufficient to fulfill a user's withdrawal, `_redeem()` attempts to retrieve assets from the strategies a user defined or overridden by the `queue_manager`. Should a queried strategy have unrealized loss, the user will take part of the unrealized loss. However, the user may take the loss in a disproportional way as shown in the code.

- First, the user's share of the unrealized loss is computed based on `assets_to_withdraw`.
- Afterwards, `assets_to_withdraw` is capped by its upper bound.

```
unrealised_losses_share: uint256 = self._assess_share_of_unrealised_losses(strategy, assets_to_withdraw)
if unrealised_losses_share > 0:
    # User now "needs" less assets to be unlocked (as he took some as losses)
    assets_to_withdraw -= unrealised_losses_share
    requested_assets -= unrealised_losses_share
    # NOTE: done here instead of waiting for regular update of these values because it's a rare case
    # (so we can save minor amounts of gas)
    assets_needed -= unrealised_losses_share
    curr_total_debt -= unrealised_losses_share

# After losses are taken, vault asks what is the max amount to withdraw
assets_to_withdraw = min(assets_to_withdraw, min(self.strategies[strategy].current_debt, IStrategy(strategy).maxWithdraw(self)))
```

If `assets_to_withdraw` is restricted to `strategy.maxWithdraw(self)`, the user will cover more than his proportional share of the loss. In addition, the updated `current_debt` of this strategy as well as the vault's total debt will diverge from the real debt because `unrealised_losses_share` has been overestimated.

```
current_debt: uint256 = self.strategies[strategy].current_debt
new_debt: uint256 = current_debt - (assets_to_withdraw + unrealised_losses_share)

# Update strategies storage
self.strategies[strategy].current_debt = new_debt
```

Code corrected:

When `max_withdraw` is the limiting factor for `assets_to_withdraw`, the unrealised loss the user takes is now adjusted proportionally. As a result, the user no longer bears more than their fair share of the loss, and the update to `current_debt` is done using the correct value.

```
# If max withdraw is limiting the amount to pull, we need to adjust the portion of
# the unrealized loss the user should take.
if max_withdraw < assets_to_withdraw - unrealised_losses_share:
    # How much would we want to withdraw
    wanted: uint256 = assets_to_withdraw - unrealised_losses_share
    # Get the proportion of unrealised comparing what we want vs. what we can get
    unrealised_losses_share = unrealised_losses_share * max_withdraw / wanted
    # Adjust assets_to_withdraw so all future calcultations work correctly
    assets_to_withdraw = max_withdraw + unrealised_losses_share
```

6.2 Inconsistent Debt Accounting on Withdrawal From Strategies

Correctness **Medium** **Version 1** **Code Corrected**

CS-YVV3-006

If `total_idle` is insufficient to fulfill the redemption, `_redeem()` attempts to retrieve assets from the strategies. Should a queried strategy have an unrealized loss, the user has to take a part of this loss, which is regarded as realized and deducted from `curr_total_debt`. At the end of the loop, `self.total_debt` is updated to `curr_total_debt`.

```
# CHECK FOR UNREALISED LOSSES
# If unrealised losses > 0, then the user will take the proportional share and realise it
# (required to avoid users withdrawing from lossy strategies)
# NOTE: assets_to_withdraw will be capped to strategy's current_debt within the function
# NOTE: strategies need to manage the fact that realising part of the loss can mean the realisation of 100% of the loss !!
```

```
#  (i.e. if for withdrawing 10% of the strategy it needs to unwind the whole position, generated losses might be bigger)
unrealised_losses_share: uint256 = self._assess_share_of_unrealised_losses(strategy, assets_to_withdraw)
if unrealised_losses_share > 0:
        # User now "needs" less assets to be unlocked (as he took some as losses)
        assets_to_withdraw -= unrealised_losses_share
        requested_assets -= unrealised_losses_share
        # NOTE: done here instead of waiting for regular update of these values because it's a
        # rare case (so we can save minor amounts of gas)
        assets_needed -= unrealised_losses_share
        curr_total_debt -= unrealised_losses_share

# After losses are taken, vault asks what is the max amount to withdraw
assets_to_withdraw = min(assets_to_withdraw, min(self.strategies[strategy].current_debt, IStrategy(strategy).maxWithdraw(self)))

# continue to next strategy if nothing to withdraw
if assets_to_withdraw == 0:
    continue
```

However, in case the strategy with unrealized loss reports 0 on `maxWithdraw()`, it will jump to the next iteration and skip the following code which updates the strategy-specific debt (`strategies.current_debt`). Consequently, the sum of all `strategies.current_debt` will exceed `self.total_debt` and result in an accounting inconsistency.

```
current_debt: uint256 = self.strategies[strategy].current_debt
new_debt: uint256 = current_debt - (assets_to_withdraw + unrealised_losses_share)

# Update strategies storage
self.strategies[strategy].current_debt = new_debt
# Log the debt update
log DebtUpdated(strategy, current_debt, new_debt)
```

Code corrected:

The updated code ensures accurate accounting before proceeding to the next loop iteration when it is not possible to withdraw funds from a strategy:

1. If funds are simply locked, the users share of the loss to cover is zero and all accounting is correct.
2. If the strategy has a complete loss, the user realiszes this loss and the strategies debt is updated.

6.3 Add Self as a Strategy

Design | Low | Version 1 | Code Corrected

CS-YVV3-012

The vault should not add itself as a strategy. Otherwise, `update_debt` will revert when funds are to be deposited into the strategy, as the recipient of the shares cannot be the vault itself.

Code corrected:

In the updated code it is no longer possible to add the vault itself as a strategy.

6.4 Incorrect Return Type of Decimals

Correctness | Low | Version 1 | Code Corrected

CS-YVV3-009

`decimals()` of contract `VaultV3` returns an `uint256` which does not comply with the ERC20 standard where an `uint8` is returned.

Code corrected:

The type of the return value has been changed to `uint8` which is compliant with the specification.

6.5 Incorrect Return Value

Correctness | Low | Version 1 | Code Corrected

CS-YVV3-011

`mint()` returns the calculated amount of assets to deposit, instead of the actual amount of assets deposited. If a user mints shares which converted into assets equal `max(uint256)`, `self._deposit()` considers this a "magic value" and will only deposit the user's balance. `mint()` however will return `max(uint256)` and not the actual amount of assets deposited.

The same issue exists for `withdraw()` when the amount of assets converted to shares equals `max(uint256)`.

The possibility of these scenarios depends on the exchange rate between shares and assets. The caller might rely on the returned values for further calculations or decision-making processes, which could lead to unintended consequences due to the discrepancy in the returned and actual deposited or withdrawn assets.

Code corrected:

Yearn has removed the ability to pass MAX_UINT as a "magic value" to use the full balance.

6.6 Incorrect and Missing Specification

Correctness | Low | Version 1 | Specification Changed

CS-YVV3-010

In contract `VaultV3`, `mint()` returns the amount of assets deposited instead of shares according to its specification. In addition, the specifications of `withdraw()` and `redeem()` are missing.

Specification changed:

The specification of `mint()` has been corrected. Specification has been added for `withdraw()` and `redeem()`.

6.7 Missing Event upon Role Change

Design | Low | Version 1 | Code Corrected

CS-YVV3-013

In contrast to other sections of the code, role management functions (with the exception of `accept_role_manager`) do not emit events upon these important state changes. Emitting events would enable external parties to observe these important state changes more easily.

Code corrected:

Events have been added to `set_role()`, `set_open_role()` and `close_open_role()`. Note that `transfer_role_manager()` does not emit an event, an event is emitted upon the completion of the role transfer in `accept_role_manager()` only.

6.8 Non ERC-4626 Compliant Functions

Correctness | Low | Version 1 | Code Corrected

CS-YVV3-007

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

In addition, `maxWithdraw()` assumes a full withdrawal is possible if `queue_manager` is set regardless of the unrealized loss. This conflicts with the specification which reads:

```
MUST NOT be higher than the actual maximum that would be accepted (it should underestimate if necessary)
```

Besides, `convertToShares()` does not distinguish the following cases when `total_assets` is 0:

- This is the first deposit where price per share is 1.
- The vault is dead where there are shares remaining but no assets. The price per share is 0 because further deposit would revert in `_issue_shares_for_amount`.

This would be misleading for external contracts to see a non-zero value when using `convertToShares()` but fail on `deposit()`.

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 diluted 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.

The full specification can be found here: https://eips.ethereum.org/EIPS/eip-4626

Code corrected:

The code has been changed so that the deposit limit is set to 0 when the vault is shutdown, thus `maxDeposit()` would return 0 in this case. `convertToShares()` has been adjusted to distinguish the case when the vault is dead. The potentially misleading return value of `previewWithdraw()` is acknowledged.

Yearn also acknowledged the risk of `maxWithdraw()` and states:

```
It is deemed acceptable for maxWithdraw() to not take into account unrealized
losses. Since this would be very gas intensive for a function potentially used
on chain, and is not possible to accurately account for vaults that allow custom
withdraw queues.
```

The external system is expected to exercise caution with the features of this contract during their integration.

6.9 Unchecked Profit Max Unlock Time

Design Low Version 1 Code Corrected

CS-YVV3-015

In contract `VaultV3`, `profit_max_unlock_time` is not checked at initialization. A faulty value may lead to unexpected behaviors. In case `profit_max_unlock_time==0`, the profit of the vault will be locked forever. In case `profit_max_unlock_time` is too large, the weighted average computation of `new_profit_locking_period` may revert, which blocks `process_report()` as a consequence.

Code corrected:

`profit_max_unlock_time` is now checked in the vault constructor and setter ensuring that it is greater than 0 and less than 1 year.

```
# Must be > 0 so we can unlock shares
assert profit_max_unlock_time > 0 # dev: profit unlock time too low
# Must be less than one year for report cycles
assert profit_max_unlock_time <= 31_556_952 # dev: profit unlock time too long
```

6.10 Unprotected Sweep Function

Security Low Version 1 Code Corrected

CS-YVV3-008

`sweep()` is not protected by the reentrancy guard. If trusted roles misbehave it's possible to sweep assets of the vault at a time when the value of `total_idle` is stale. No direct issue has been uncovered, however this permits excessive access which may introduce unnecessary risks.

- In `deposit()`, one could reenter by calling `sweep()` in the hook of `erc20_safe_transfer_from()` only if the weird underlying token calls back to the sender after transferring the token.
- Another case is that a strategy reenters `sweep()` when `update_debt()` calls `withdraw()` on the strategy. As the balance withdrawn is determined based on the delta of the actual balance, this shouldn't have any negative impact, apart from potentially spurious events.

Code corrected:

A guard has been added for extra safety.

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 Loose Token Decimal Restriction

Informational **Version 1**

CS-YVV3-003

The vault's share token has the same token decimal as the underlying token. The underlying token decimal is restricted (<= 38) in the `VaultV3` constructor.

Token decimals are only for user representation and front-end interfaces. At the smart contract level, all balances maintain token decimal precision. Overflows could potentially occur if a token permits sufficiently large balances, leading to an overflow when these balances are multiplied. Importantly, this issue is unrelated to decimals, so the check in the constructor cannot prevent it.

Note that we are not aware of any meaningful token with this behavior, this is more a theoretical consideration.

Yearn understand that overflows are still possible no matter the token decimal value used. The check was updated, it now only ensures that the decimal value does not exceed an `uint8`. Legitimate vaults with a normal underlying token will not trigger any overflows.

7.2 Updating Queue_Manager

Informational **Version 1**

CS-YVV3-004

The `queue_manager` smart contract defines the withdrawal sequence for a vault. Whenever a new strategy is added, the vault informs the `queue_manager` by calling `queue_manager.new_strategy(address strategy)`.

The queue manager for the vault can be updated using `set_queue_manager()`. Note that the new queue manager is not informed about all existing strategies of the vault; in this case the queue manager must be configured correctly manually.

7.3 yv<Asset_Symbol> Not Enforced

Informational **Version 1**

CS-YVV3-005

The system specification requires the shares to be named yv<Asset_Symbol>. Note that this isn't enforced by the code, the share name can be freely defined when deploying a new Vault.

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 Debt Rebalanced in a Linear Way

Note Version 1

During `update_debt()`, if the target debt value cannot be reached given the vault and strategy specific limitations on the idle and debt, it will not revert. Instead, it will rebalance the debt to the closest value towards the target. This behavior assumes it is always better to be closer to the target. However, the assumption may not always be true for different strategies.

8.2 No User Protection on Shares Redemption

Note Version 1

If during redemption funds must be pulled from a strategy at a loss, the user must cover his share of this not yet realized loss. Additionally, in case the call to `strategy.withdraw()` results in less than the requested assets, the user takes the full loss.

Unaware users may redeem their shares for less of the underlying than they expect. There is no protection e.g. in form of a parameter which allows the user to specify the minimum amount of underlying to receive / shares to be burned he tolerates before the transaction should revert.

Yearn states:

```
It is expected that off chain users interact with the vaults through an ERC-4626
router which has logic to set minimums and slippage tolerance for deposits and
withdrawals. And on chain users can either use the router or set their own limits.
```

8.3 Queue Manager Can Pause Withdrawals From Strategies

Note Version 1

A faulty or malicious `queue_manager` with `should_override` enabled can pause users' withdrawals from strategies by: (1) directly revert. (2) return a non-existing strategy. `queue_manager` must be properly configured and trusted if enabled.

In Version 2 the `should_override` option has been removed so users can always bypass the `queue_manager` if a customized withdraw queue is specified. Otherwise, the withdraw queue will be queried from the `queue_manager`.

```
if queue_manager != empty(address):
    if len(_strategies) == 0:
        _strategies = IQueueManager(queue_manager).withdraw_queue(self)
```

8.4 Race Condition on Withdrawal From Strategies

Note **Version 1**

If `total_idle` is insufficient to fulfill the redemption, `_redeem()` attempts to retrieve assets from the strategies. Should a queried strategy have unrealized loss, the user has to take a part of this loss. In case the vault has global unrealized loss, users may engage in a race to withdraw from the optimal strategies.

- In case the `queue_manager` is disabled, users will race to withdraw from the strategies without unrealized loss. As a consequence, the tardy users will take more unrealized loss.
- In case the `queue_manager` is enabled, withdrawals may be biased across all strategies depending on the actual construction of the `withdraw_queue`.

Users will only share the unrealized loss of a strategy in a fair way if it is reported by the `REPORTING_MANAGER`.

8.5 Tokens With a Blacklist

Note **Version 1**

Tokens such as USDC maintain a blacklist that prohibits the transfer of tokens to and from the addresses listed on it. Assuming a vault utilizes such a token, a blacklisted address would be unable to be the recipient when funds are withdrawn. If a strategy is blacklisted, withdrawal of allocated funds would be impossible. Furthermore, if a vault itself is blacklisted, the withdrawal of all deposited funds would be prevented.

8.6 Trade-off in Profits Distribution

Note **Version 1**

All profits getting paid to vault depositors are retroactive:

- New joiners of a vault will share part of the locked profits accumulated before they entered.
- The locked profits generated by their deposits will be forfeited upon their withdrawals.

This is a trade-off to improve the gameability and avoid intensive gas to track specific accounts for the time they deposit. As long as the profits are distributed slowly and continuously, no whales are expected to game the system by deposit right before a profit harvest and realize full gains.

8.7 User-Selected Strategies

Note **Version 1**

If no queue manager is configured, when idle funds are insufficient for a withdrawal, users can specify which strategies should be used to retrieve funds.

This could enable users to substantially interfere with the planned allocation of assets, necessitating frequent intervention from the debt_manager.