PUBLIC

Code Assessment

of the Spool V2 Smart Contracts

October 20, 2023

Produced for

by

Contents

1 Executive Summary

Dear all,

Thank you for trusting us to help Spool with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of Spool V2 according to Scope to support you in forming an opinion on their security risks.

Spool implements a system for meta-strategies where users invest in vaults that then collectively invest in strategies that interact with third-party DeFi systems.

The most critical subjects covered in our audit are functional correctness, access control, denial-of-service, precision of arithmetic operations, and reentrancy. Security regarding all the aforementioned subjects is good.

The general subjects covered are gas-efficiency, documentation, and error handling.

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 do not 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	1
• Code Corrected	1
High-Severity Findings	7
• Code Corrected	7
Medium-Severity Findings	5
• Code Corrected	5
Low-Severity Findings	20
• Code Corrected	18
• Specification Changed	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 Spool V2 repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

Private Repository

V	Date	Commit Hash	Note
1	24 April 2023	a42763a240ce924ead97ad7c1aab09655703bf33	Initial Version
2	24 May 2023	b63e530b83262016ecf4f0c77bedcd51e1e9e7bf	Second Version
3	28 June 2023	e2e65993b4209cac6a31a35fdf49b7c1b075bd36	Third Version
4	5 July 2023	c378d53b2b6641f166646d1fb330a448098899a7	Fourth Version
5	7 July 2023	09a7dcb3f946fa5ec8dcfd5fa462e2d37ff05b5c	Fifth Version

Public Repository

V	Date	Commit Hash	Note
1	18 October 2023	09a7dcb3f946fa5ec8dcfd5fa462e2d37ff05b5c	Fifth Version

For the solidity smart contracts, the compiler version `0.8.17` was chosen.

The files in scope are:

```
src/SmartVault.sol
src/MasterWallet.sol
src/guards/AllowlistGuard.sol
src/managers/ActionManager.sol
src/managers/RiskManager.sol
src/managers/DepositManager.sol
src/managers/StrategyRegistry.sol
src/managers/WithdrawalManager.sol
src/managers/GuardManager.sol
src/managers/AssetGroupRegistry.sol
src/managers/SmartVaultManager.sol
src/managers/UsdPriceFeedManager.sol
src/providers/ExponentialAllocationProvider.sol
src/providers/LinearAllocationProvider.sol
src/providers/UniformAllocationProvider.sol
src/libraries/ReallocationLib.sol
src/libraries/uint16a16Lib.sol
src/libraries/ArrayMapping.sol
src/libraries/uint128a2Lib.sol
src/libraries/SpoolUtils.sol
```

```
src/libraries/MathUtils.sol
src/interfaces/ISmartVaultManager.sol
src/interfaces/IMasterWallet.sol
src/interfaces/IRiskManager.sol
src/interfaces/IWithdrawalManager.sol
src/interfaces/IGuardManager.sol
src/interfaces/Constants.sol
src/interfaces/IDepositSwap.sol
src/interfaces/CommonErrors.sol
src/interfaces/IRewardManager.sol
src/interfaces/IStrategy.sol
src/interfaces/IDepositManager.sol
src/interfaces/ISmartVault.sol
src/interfaces/IAllocationProvider.sol
src/interfaces/IUsdPriceFeedManager.sol
src/interfaces/ISwapper.sol
src/interfaces/RequestType.sol
src/interfaces/IAction.sol
src/interfaces/IRewardPool.sol
src/interfaces/ISpoolAccessControl.sol
src/interfaces/IStrategyRegistry.sol
src/interfaces/IAssetGroupRegistry.sol
src/rewards/RewardPool.sol
src/rewards/RewardManager.sol
src/DepositSwap.sol
src/Swapper.sol
src/SmartVaultFactory.sol
src/external/interfaces/chainlink/AggregatorV3Interface.sol
src/external/interfaces/weth/IWETH9.sol
src/access/Roles.sol
src/access/SpoolAccessControl.sol
src/access/SpoolAccessControllable.sol
src/strategies/convex/Convex3poolStrategy.sol
src/strategies/convex/ConvexAlusdStrategy.sol
src/strategies/convex/ConvexStrategy.sol
src/strategies/curve/Curve3CoinPoolBase.sol
src/strategies/curve/Curve3poolStrategy.sol
src/strategies/curve/CurveAdapter.sol
src/strategies/curve/CurvePoolBase.sol
src/strategies/helper/StrategyManualYieldVerifier.sol
src/strategies/__BaseStrategy__.sol
src/strategies/AaveV2Strategy.sol
src/strategies/CompoundV2Strategy.sol
src/strategies/GhostStrategy.sol
src/strategies/IdleStrategy.sol
src/strategies/MorphoAaveV2Strategy.sol
src/strategies/MorphoCompoundV2Strategy.sol
src/strategies/MorphoStrategyBase.sol
src/strategies/NotionalFinanceStrategy.sol
src/strategies/REthHoldingStrategy.sol
src/strategies/SfrxEthHoldingStrategy.sol
src/strategies/StEthHoldingStrategy.sol
src/strategies/Strategy.sol
src/strategies/WethHelper.sol
src/strategies/YearnV2Strategy.sol
```

In Version 3, the following files were added to the scope:

```
src/libraries/PackedRange.sol
```

2.1.1 Excluded from scope

All other files. External protocols are assumed to be working correctly; hence, out-of-scope. Rebasing and Fee-taking tokens are assumed not to be used in the system. Note that the scope is limited to the presented code and does not include the integration of future, yet unknown, integrations and extensions.

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.

Spool implements Spool V2, a collection of smart contracts where users can deploy meta-strategies, called Smart Vaults (SVs), that then invest into strategies interacting with DeFi protocols. Users invest in SVs and receive Smart Vault Tokens (SVTs) in return while SVs then invest the user funds into strategies, receiving Strategy Share Tokens (SSTs) in return, that deposit into third-party protocols. To reduce gas costs, the number of interactions with external protocols is reduced by aggregating the SVs' funds and investing them all together with "Do Hard Work" (DHW) actions. Ultimately, an asynchronous process of depositing and withdrawing is implemented.

2.2.1 Smart Vaults

SVs are the layer at which users' investments are kept track of while being investors in strategies who themselves invest in third-party protocols. Given a specification, the SVs are deployed through the `SmartVaultFactory` that sets the required state for the SV according to the specification. Note that this should define the execution model of the SV and consists of several parameters.

1. Strategy-specific parameters
    1. Asset Group ID: must be a valid asset group ID.
    2. Strategies: all must be strategies of the same asset group ID and whitelisted. Limited to 16 strategies. Requires at least one strategy. Strategies must be unique.
2. Allocation-specific parameters:
    1. Fixed strategy allocation: fixed asset allocation among the strategies. If zero, this is irrelevant, since the parameters in 2. are used for computing allocations dynamically. If non-zero, the parameters in 2. are irrelevant.
    2. Risk tolerance: the parameter for weighting the contribution of APYs and risk scores. The higher the risk tolerance, the higher the APY should be weighted against risk scores when computing allocations.
    3. Risk provider: address providing risk scores for strategies in the risk manager.
    4. Allocation provider: provides the allocation of funds amongst different strategies of the SV.
3. Action Hook parameters (note that there is a maximum of 10 actions per request type)
    1. Actions: list of whitelisted action contracts.

2. Action Request Types: a list, including what type of actions these are (in which context they are called).

4. Guard Hooks parameters (note that there is a maximum of 10 actions per request type for each SV)
    1. Guards: a set of guard definitions per request type. A guard definition is defined by a guard contract, (string) method signature, an expected result value, the parameter types the guard is expecting, the parameters to be passed to the guard, and the operator to compare the expected value with the guard result. If no valid operator is used, the guard manager simply treats the return value as a boolean.
    2. Guard Request Types: list of what type of guards these are (in which context they are called).
5. Fee parameters
    1. Management fee percentage: at most 5%
    2. Deposit fee percentage: at most 5%
    3. Performance fee percentage: at most 20%
6. Other
    1. Name
    2. Flag defining whether the vault admin can redeem for users.
    3. The initial owner is set to the deployer.

Keep in mind that several parameters require a governance setup to be valid. See the corresponding sections. (Asset Group Registry, Risk Manager, Allocation Provider, Guards, and Actions)

Once an SV is deployed, the central entry point for SV-related actions is the `SmartVaultManager` (SVM) contract. It handles users' actions such as user deposits, redemptions, and SVT claims. Furthermore, it manages the communication with the `StrategyRegistry` which is the entry-point for DHW and, thus, the entry-point for SVs to indirectly interact with strategies.

The SVM orchestrates the strategies asynchronously. Namely, in so-called flush cycles, deposits and withdrawals are aggregated per smart vault. At the end of such a cycle (when the smart vault is flushed), the SVM will await the next DHW cycle, which should occur in the future. Hence, funds are awaited to be freed and to be deposited. Once, the DHW cycles of all underlying strategies end, the SVM can synchronize the result and, hence, finalize the deposits and withdrawals. Ultimately, SVM notifies strategies when flushing, while retrieving the results upon synchronizing.

More specifically, with `SmartVaultManager.deposit()` and `SmartVaultManager.redeem()` (or `SmartVaultManager.redeemFor()`) users register their actions to be applied in the next DHW cycle. The users receive deposit and withdrawal receipts in the form of an ERC-1155 (SV implements EIP-1155) as so-called D-NFTs and W-NFTs which will grant them the right to claim SVTs or the underlying funds, respectively, according to their deposited amounts and redeemed shares. Note that the deposit amount should be distributed with a valid ratio according to the allocations of the underlying strategies and the asset ratios in the strategies at the last executed DHW.

Once the previous flush cycle has been synchronized, an SV can be flushed again through `SmartVaultManager.flushSmartVault()` (or also optionally possible in deposits or withdrawals).

Smart Vault Manager notifies the strategies about aggregated deposits and withdrawals in an SV. These aggregated deposits and withdrawals are accumulated for the next DHW cycle. More specifically, this aggregation occurs in the `StrategyRegistry` contract. Note that a new flush cycle can start after the finished one has gotten synchronized. Meaning that new deposits and withdrawals can be made while the SV awaits the result of the strategies; however, no new flush can be performed until the SV's previous flush has not been finalized (synchronized). For deposits, a distribution of the underlying funds is computed according to the SV's strategy allocation and the asset ratios at the last DHW. Namely, using these it computes the asset allocations across strategies and the ideal deposits across assets. The ratio between these two essentially defines what share of a token deposit should go to a strategy.

Once all DHWs for the strategies of an SV have been performed, the previous ("toSync") flush cycle can be synchronized through (`SmartVaultManager.syncSmartVault()` (or mandatorily through any other action at its beginning). The awaited results of the strategies are processed. Meaning, the underlying assets of strategies and shares of SSTs are claimed according to the state at DHW and the SV's deposits and withdrawals. Additionally, SVTs are preminted for all users according to the state when the flush occurred (and DHW-dependent state necessary for SV fees).

Once the deposit/withdrawal epoch is synchronized, users can redeem their D-NFTS and W-NFTs for SVTs and underlying tokens, respectively, with the corresponding functions `SmartVaultManager.claimSmartVaultTokens()` and `SmartVaultManager.claimWithdrawal()`.

To summarize, the process can be described in five steps:

1. Aggregate deposits and withdrawals on the SV level. Users receive D-NFTs and W-NFTs.
2. Register SV for the next DHW by flushing. Aggregates SV deposits and withdrawals per strategy.
3. SV awaits the results of its strategies' DHWs.
4. The SV can be synchronized. Based on the results of the DHW, an SV's shares of SSTs and underlying are claimed while SVTs are preminted for all users (unpacking aggregation at 2.).
5. Based on the results of 4., the aggregated user actions of 1. can be unpacked per user by redeeming the receipt tokens in return for SVTs or underlying tokens.

Note that the `SmartVaultManager` manages the flush cycles and hence the synchronization. It, however, commands the `DepositManager` and the `WithdrawalManager` contracts to manage the deposits and withdrawals. First, they store the SV's aggregated actions. Second, they communicate the deposits and withdrawals to the `StrategyRegistry` (which consequently aggregates for the strategies, see Strategies). Third, they process the deposit and withdrawal results. Last, they command the `SmartVault` contract to mint/burn its ERC-20/ERC-1155.

Note that there exists a non-asynchronous redeem function, `SmartVaultManager.redeemFast()`. Through the `WithdrawalManager`, it computes the amounts of SSTs claimable by the burned SVTs. These SSTs are then redeemed through the `StrategyRegistry` (which essentially batches calls to the strategies). The withdrawn amounts are then received by the user. The user can decide to retrieve the funds either `redeem()` or `redeemFast()`, depending on his personal preferences (e.g., gas cost, potentially unavailable funds in the underlying protocols, etc.).

Note that `ROLE_SPOOL_ADMIN` removes strategies from vaults (and optionally makes them unsupported by the system) by calling `removeStrategyFromVaults()`. It replaces the strategies in the SVs with a "ghost strategy" that is typically ignored when performing actions.

Recall that SVs can have dynamic strategy allocations. Changing the allocations is performed through a call to `reallocate()` by `ROLE_REALLOCATOR`. A rough description of the process:

1. Retrieves and sets new allocations from the risk manager according to the SV specification (synchronizes the smart vault if possible).
2. Computes per SV the deposits and withdrawals made to strategies in USD. Computes the total deposits in USD needed.
3. Computes per SV the shares to be redeemed for the withdrawals.
4. For each SV, it virtually distributes each withdrawal in USD to the strategies that need a deposit (based on the share they have of the total deposits needed). These results are aggregated to a matrix where `(i,j)` represents the total USD flow from `S_i` to `S_j`.
5. Matches between flows from `S_i` to `S_j` are computed. Essentially, `S_i` will return some shares `SST_j` to `S_j` as a "flow" that is equal in USD value to what `S_j` will return in `SST_i` to `S_i`.
6. However, unmatched amounts will exist which result in unmatched SSTs. These are redeemed. The resulting underlying funds are then distributed according to the unmatched USD amounts and

deposited into the strategies. This can be understood as `S_i` transferring (by converting first) `SST_j` shares to `S_j`.

7. Ultimately, each SV can claim `SST_j` shares from the withdrawals it made from `S_i` to deposit into `S_j` for the shares matched and unmatched (5. and 6.) according to the ratio of the value it deposited into `S_j` and the total value that has moved from `S_i` to `S_j`.

2.2.2 Strategies

The `StrategyRegistry` is notified by `DepositManager` and `WithdrawalManager` about how much they want to deposit to and withdraw from each strategy (`addDeposit()` and `addWithdrawal()`, respectively). With the aggregated deposits and withdrawals per strategy, the next DHW will deposit to and redeem from strategies. That is done by the `StrategyRegistry.doHardWork()` which batches the `doHardWork()` functions of each strategy. Note that, the strategies communicate the number of SSTs minted, the number of assets withdrawn, the yield percentage since the last DHW, the strategy value at post-DHW, and the total SSTs post-DHW. The registry then proceeds to finalize the DHW for each strategy

1. by storing the asset ratios of the strategies (required for computing the distribution when flushing SVs and for checking the deposit amounts to SVs),
2. by storing the asset exchange rates (required for claiming SSTs according to the deposits made when synchronizing),
3. by storing the assets withdrawn (required for splitting the underlying assets among SVs when synchronizing) and the unclaimed assets (assets not claimed yet - changes when SVs claim underlying assets. Required for strategy removals),
4. by storing the minted and total SSTs, the strategies' values, and the DHW timestamp,
5. by updating the total yield percentage accumulated over time,
6. and by updating the weighted APY according to the new yield generated and the weighting formula.

Further, other functions are used internally (e.g., `redeemFast()`, `claimWithdrawals()`, `removeStrategy()`). Besides these, the platform fees can be set with `setEcosystemFee()`, `setEcosystemFeeReceiver()`, `setTreasuryFee()` and `setTreasuryFeeReceiver()` by `ROLE_SPOOL_ADMIN`. Additionally, `ROLE_SPOOL_ADMIN` can set the emergency wallet with `setEmergencyWithdrawalWallet()` while the `ROLE_EMERGENCY_WITHDRAWAL_EXECUTOR` can initiate `emergencyWithdraw()` that batches `emergencyWithdraw()` calls to strategies (and optionally revokes their strategy status). The `ROLE_SPOOL_ADMIN` can add new strategies with `registerStrategy()`. Finally, the `ROLE_STRATEGY_APY_SETTER` can set a custom strategy APY with `setStrategyApy()` (in contrast to the running average APY computation in DHW).

2.2.3 Strategy

The contract `Strategy` provides Spool V2 with an interface to the external protocols. As mentioned in Strategies, it should implement the following functionalities:

1. `doHardWork()` to make the actual deposits to/withdrawals from the underlying protocol,
2. `emergencyWithdraw()` to be called when a strategy is defected,
3. An ERC-20 implementation to represent its SSTs. It should support releasing and claiming SSTs,
4. Functionality to burn SSTs of an SV and return its deposited assets,
5. Functionality to accumulate protocol rewards (if the underlying protocol distributes any),
6. Functionality to distribute SSTs.

Each underlying protocol has a specific API. Hence, for each protocol, a customised implementation of `Strategy` is devised. In what follows, we describe the strategies of each protocol.

2.2.3.1 Curve

The `Curve3poolStrategy` interacts with the Curve 3pool containing DAI, USDC and USDT. All three tokens must be deposited in roughly the same ratio the pool's balances are currently in. Deposited assets yield an LP token that is then deposited into the respective Gauge contract of the pool, which yields yet another LP token. The Gauge emits CRV token rewards that are redeemed and immediately swapped to the underlying tokens on each DHW run. These tokens are then deposited into the pool again.

2.2.3.2 Convex

The `Convex3poolStrategy` works similarly to the `Curve3poolStrategy`. The pool LP tokens are, however, deposited into Convex' `Booster` contract instead of the Curve Gauge. The `Booster` contract yields CVX tokens in addition to CRV tokens.

The `ConvexAlusdStrategy` also invests in the Curve 3pool. It then invests the LP tokens into the Curve alUSD pool which consists of the 3pool LP token and the alUSD token, creating a pool with 4 tokens in total. The strategy only invests the LP token and 0 alUSD. It also only withdraws the LP token and therefore consists of only the 3 original tokens. The alUSD pool LP token is then further invested into Convex' `Boooster` contract.

2.2.3.3 Aave V2

The `AaveV2Strategy` can be deployed to multiple instances, each with a different underlying token. The strategy has strictly only one underlying token per instance and can therefore only be used by SmartVaults that belong to an asset group consisting of this token. The strategy invests in an Aave v2 market as a supplier and does not perform borrowing on the supplied collateral. It does not handle any rewards.

2.2.3.4 Compound V2

The `CompoundV2Strategy` can be deployed to multiple instances, each with a different underlying token. The strategy has strictly only one underlying token per instance and can therefore only be used by SmartVaults that belong to an asset group consisting of this token. The strategy invests in a Compound v2 market as a supplier and does not perform borrowing on the supplied collateral. It redeems `COMP` rewards on every DHW run, converts them back to underlying and deposits them back into the protocol.

2.2.3.5 Idle

It implements an interface to Idle protocol, which facilitates users to optimise their asset allocations across different protocols. This protocol is governed by users holding the governance tokens.

`IdleStrategy` supports only one underlying asset. Hence, it can be deployed multiple times with different tokens. After receiving the underlying token, the strategy invests it in the Idle protocol. In return, the users, for each deposited token, receive `1/$IDLE Price`, as `IDLE` is a rebasing token and its value increases monotonically. Users having deposited to Idle receive their rewards of governance tokens, which can be traded against the underlying tokens and consequently be deposited into the protocol.

2.2.3.6 Morpho

Morpho is a P2P lending layer that uses another lending protocol for liquidity. Morpho contracts for two of these underlying lending protocols are used in Spool V2:

The `MorphoCompoundV2Strategy` can be deployed to multiple instances, each with a different underlying token. The strategy has strictly only one underlying token per instance and can therefore only be used by SmartVaults that belong to an asset group consisting of this token. The strategy invests in a Morpho Compound v2 market as a supplier and does not perform borrowing on the supplied collateral. It redeems `MORPHO` rewards on every DHW run, converts them back to underlying and deposits them back into the protocol.

The MorphoAaveV2Strategy can be deployed to multiple instances, each with a different underlying token. The strategy has strictly only one underlying token per instance and can therefore only be used by SmartVaults that belong to an asset group consisting of this token. The strategy invests in a Morpho Aave v2 market as a supplier and does not perform borrowing on the supplied collateral. It redeems MORPHO rewards on every DHW run, converts them back to underlying and deposits them back into the protocol.

2.2.3.7 Notional

Notional is a lending protocol for fixed-rate, fixed-term loans. While fixed-term lending might require interaction from time to time, Notional also allows liquidity provisioning via so-called nTokens that provide liquidity to all markets and roll-over loans automatically. Spool V2 invests in these nTokens.

The NotionalFinanceStrategy can be deployed to multiple instances, each with a different underlying token. The strategy has strictly only one underlying token per instance and can therefore only be used by SmartVaults that belong to an asset group consisting of this token. The strategy invests into a nTokens for a certain market which is used to provide liquidity to multiple pools. The investments can either be net-lending or net-borrowing depending on the current market. nToken investments yield Note token rewards that are reinvested on every DHW run.

2.2.3.8 Rocket Pool

Rocket Pool gathers ETH from the staking users to spin up the ETH validators. Hence, users can only deposit ETH to Rocket Pool and in return receive rETH. As Ethereum validators receive rewards, they pay fees to the protocol. As a result, for a certain amount of deposited ETH, fees will be accumulated and the price of rETH held by users against ETH increases. Spool V2, however, does not directly deposit to Rocket Pool, but trades ETH with rETH through Uniswap and Balancer.

It is worth mentioning, that as all deposited ETH in Spool V2 are converted to WETH, when interacting with Rocket Pool, they should again be wrapped and unwrapped.

2.2.3.9 Frax

Frax Ether is an ETH staking derivative to generate yields. Staked ETH in Frax comes in two forms, either frxETH or sfrxETH. When a user deposits to Frax by calling frxETHMinter.submitAndDeposit(), frxETH gets minted. Holding frxETH on its own is not eligible for staking yield. Therefore, frxEthMinter locally exchanges frxETH against sfrxETH. sfrxETH accrues the staking yield of Frax ETH validators. While sfrxETH is a rebasing token, the exchange rate of frxETH per sfrxETH increases.

Spool V2 devises another option for obtaining sfrxETH, as well as depositing to frxEthMinter, which is exchanging ETH on Curve to receive frxETH and deposit it to sfrxEthToken contract to receive the respective amount of sfrxETH. When withdrawing, the only option is to redeem sfrxETH for frxETH and exchange it on Curve to get ETH back.

2.2.3.10 Lido

Lido is a liquid staking pool. It acts as an ERC-20 token, which represents staked ETH, namely stETH. Although stETH tokens are pegged by deposited ETH, they yield fees and the market exchange rate between stETH and ETH increases with more ETH being deposited.

Spool V2 can receive stETH either by sending ETH to the submit function of Lido or exchanging its ETH on Curve to obtain stETH. When redeeming, it sells stETH on Curve and receives ETH. Like Frax, during redeeming the only possible way is to trade stETH against ETH on Curve.

2.2.3.11 Yearn

Yearn interfaces a Yearn Token Vault, which holds one and only one underlying token. Yearn, similar to Spool V2, can be connected to multiple strategies to maximize its yield. It has a monitoring mechanism, named `Yearn Watch`, which monitors the health of underlying strategies and if necessary, reallocates the deposited funds amongst them. Apart from this algorithm, to maximize the yield, Yearn has no reward tokens.

When a user deposits to Yearn, it can theoretically revert if the total deposited value reaches a pre-set limit. Minting `yvToken` is based on the free funds in the system (outstanding debts included). Upon withdrawing, not enough tokens might be present in the Yearn Token Vault. In this case, Yearn has to withdraw from the underlying strategies, which could potentially cause loss.

2.2.4 Guards

On SV deployment, the `GuardManager` contract receives a set of guards and request types through `setGuards()` from the SV factory. This deploys a "storage contract" per request type that contains all guard definitions.

Guards support different request types (`TransferSVTs`, `TransferNFT`, `BurnNFT`, `Deposit`, and `Withdrawal`). The view function `runGuards()` runs guards of a certain type. Note that this requires encoding calldata according to the guard definition.

Users should carefully set up these guards, given the creation of the calldata.

Note that the only guard present in the codebase is the `AllowlistGuard` where the smart vault-specific role `ROLE_GUARD_ALLOWLIST_MANAGER` can add and remove users from a whitelist with `addToAllowlist()` and `removeFromAllowlist()`.

2.2.5 Actions

Upon the SV deployment, the `ActionManager` contract receives a set of actions and request types through `setAction()` from the SV factory. These, however, need to be whitelisted. Only `ROLE_SPOOL_ADMIN` can whitelist actions with `whitelist()`.

The only request types that will be executed are `Deposit` and `Withdrawal`. The former will be executed before the underlying assets are actually deposited into SV, while the latter will be executed when withdrawals are claimed before the funds are sent. Note that this happens only through Spool V2 internal calls to `runActions()`.

2.2.6 Asset Group Registry

The `AssetGroupRegistry` contract defines supported groups of assets (at least one item). To create an asset group, `ROLE_SPOOL_ADMIN` can register a list of **ordered** assets through `registerAssetGroup()`, which means that the asset group will not contain duplicate assets. Note that each group can have only one ID. Further, the assets added to groups must be whitelisted which the `ROLE_SPOOL_ADMIN` can do through `allowToken()` or `allowTokenBatch()`.

2.2.7 Risk Manager

On SV deployment, the `RiskManager` contract stores the SV's risk provider, risk tolerance, and allocation provider with `setRiskProvider`, `setRiskTolerance()`, and `setAllocationProvider()`. The risk manager receives risk scores from `ROLE_RISK_PROVIDER` (note these are risk provider based) through the call `setRiskScores()`. These risk scores are in the range from 1 to 100.

Note that deployers can specify `STATIC_RISK_PROVIDER` as the risk provider which allows having a static risk score of one for every strategy.

The `calculateAllocation()` computes an SV's relative strategy allocation. It forwards the strategies, their risk scores and APYs, and the SV's risk tolerance to the allocation provider that computes the allocation. Note that it is validated that the allocation sums up to 100%.

2.2.8 Allocation Provider

The allocation provider devises `calculateAllocation()` for the instances of `RiskManager`. It practically calculates the allocation of SV's funds to the strategies, and ultimately the underlying protocols. In the current state of Spool V2 three different policies for allocation provider is implemented, which we are going to cover next.

2.2.8.1 Uniform Allocation Provider

Oblivious to the risk tolerance and APYs of the underlying strategies, this allocation provider evenly distributes funds to the strategies. Due to rounding errors, the calculated allocations probably do not sum up to 100%, hence, it collects the dust by assigning the difference to the first strategy.

2.2.8.2 Linear Allocation Provider

Given a risk score, it first finds `riskWeight` and `apyWeight`. The higher the risk score, the higher `apyWeight` and the lower `riskWeight`. It calculates the allocation to a given strategy as

```
normalizedApy = (uint256(data.apys[i]) * MULTIPLIER) / apySum;
uint256 normalizedRisk = (MULTIPLIER - (data.riskScores[i] * MULTIPLIER) / riskSum) / (data.apys.length - 1);
allocations[i] = normalizedApy * apyWeight + normalizedRisk * riskWeight;
```

It is worth mentioning, that if APY of a strategy is negative, `normalizedApy` is set to 0; hence, the allocation would be computed solely concerning risk factors. Lower risk scores lead to higher `normalizedRisk`, while higher `apy` lead to higher `normalizedApy`.

2.2.8.3 Exponential Allocation Provider

It calculates the allocation to each strategy according to the following formula:

$$allocations[i] = \frac{2^{apy[i]^{riskTolerance}}}{riskScore[i]}$$

As seen through this formula, higher risk tolerances as well as higher strategy APY's increase the allocation, while risk score scales down the allocation.

2.2.9 USD Price Feed Manager

Given the pool of potentially distinct assets in an asset group, the USD price feed manager helps unify the value of the underlying tokens to USD as a denomination unit. The USD price feed manager is a wrapper around Chainlink price feeds. `ROLE_SPOOL_ADMIN` can define the required parameters per asset with `setAsset()` so that it works correctly with the Spool V2.

2.2.10 Master Wallet

The master wallet is the contract that holds underlying tokens. The funds deposited, waiting to be invested into strategies, and the funds withdrawn from strategies, waiting to be claimed, are put into the `MasterWallet`. It can only be used by `ROLE_MASTER_WALLET_MANAGER` (e.g., withdrawal manager and strategy registry for transfers, governance for giving approvals).

2.2.11 Access Control

All access control is centrally managed in `SpoolAccessControl`.

The default administrator role is equal to ROLE_SPOOL_ADMIN. Special role administrators are the ADMIN_ROLE_STRATEGY and ADMIN_ROLE_SMART_VAULT_ALLOW_REDEEM that are the admins for giving out ROLE_STRATEGY and ROLE_SMART_VAULT_ALLOW_REDEEM roles.

It expands on the OpenZeppelin access control library and expands it by implementing

1. grantSmartVaultRole(): either the governance or the vault admin can give out custom vault-specific roles (see example in Guards). revokeSmartVaultRole() revokes the role from an address. renounceSmartVaultRole() and renounceSmartVaultRole() to renounce from a smart vault role.
2. grantSmartVaultOwnership(): used by factory to give ROLE_SMART_VAULT_ADMIN to the deployer.
3. pause() and unpause(): to pause and unpause the system (e.g., pauses SV manager and DHW).

2.2.12 Swapper

The Swapper contract acts as a wrapper around DEXs. The ROLE_SPOOL_ADMIN can (dis-)allow exchange addresses for performing swaps with updateExchangeAllowlist(). The swap() function, swaps the tokens-in against the tokens-out and sends the tokens-out to the receiver. To swap, the tokens-in need to be transferred to the swapper, before the swap() function is called. It batches multiple calls to whitelisted exchanges according to the swap information (arbitrary calls possible). Unused funds are returned to the receiver.

Note that no slippage protection is included here and that it is expected that exchanges are implementing slippage protection. In general, it is not guaranteed that the swap info does not swap to other tokens rather than the tokens-out.

2.2.13 Deposit Swap

The DepositSwap contract is a peripheral contract that implements the function swapAndDeposit(). It pulls funds from the caller (supporting native ETH) and swaps them through the swapper according to the swap information to the assets of the smart vault. When the swap is complete, the funds are deposited into the smart vault, and the remaining (known of) assets are sent back to the caller.

2.2.14 Rewards

The reward contract offers the possibility to add extra incentives to smart vaults. Namely, the vault administrator or the Spool admin can add a reward with addToken() for an SV. Given a number of reward tokens, the reward token, and an end timestamp of the reward release schedule, a configuration for an SV can be created. These tokens are expected to be made claimable by ROLE_REWARD_POOL_ADMIN in cycles (according to the release rate defined) in an off-chain component by computing a Merkle Tree and publishing its root with addTreeRoot() (option to update the root updateTreeRoot() available). Users can claim rewards with claim(). Note that, it collects all the rewards up to the cycle specified for a given SV and incentive token (implies that each user's total rewards claimable must be monotonically increasing). Consider that an incentive period can be extended for the same duration with extra tokens with extendRewardEmission(), and that only non-blacklisted and non-underlying (per SV) tokens are supported as incentives. Once a reward has been fully released (end time has passed), it can be removed to create space for new rewards since at most 6 incentive programs per SV can be active.

Only ROLE_SPOOL_ADMIN can (un-)blacklist tokens for an SV with forceRemoveReward() and removeFromBlacklist().

2.2.15 Roles & Trust Model

Main contracts such as Smart Vault, Smart Vault Factory, Strategy Registry, and Asset Group Registry are deployed behind a beacon proxy. The proxy admin (assumed to be appropriately chosen, e.g., a timelocked or limited multisig) is fully trusted to act honestly and correctly at all times.

We assume a correct deployment and consequently a sound assignment of roles.

There are several roles present that refer to smart contracts of the system. These are expected to match the right contracts and are trusted.

- `ROLE_SMART_VAULT_INTEGRATOR`: It deploys and adds SVs to the Spool. However, users can choose whether to interact with an SV or not. It should be assigned to the SmartVaultFactory.
- `ROLE_MASTER_WALLET_MANAGER`: In a sane deployment, this role is granted to trusted core contracts namely SmartVaultManager, StrategyRegistry, DepositManager, and WithdrawalManager.
- `ROLE_SMART_VAULT_MANAGER`: Granted to SmartVaultManager, DepositManager and WithdrawalManager; hence, trusted.
- `ROLE_STRATEGY_REGISTRY`: StrategyRegistry holds this role and is assumed to be trusted.
- `ROLE_ALLOCATION_PROVIDER`: Each contract holding this role should be fully trusted, as it can be queried as an allocation provider.
- `ROLE_STRATEGY`: To be valid, a strategy should have this role. It should be trusted, as it acts as an interface for the underlying protocol.
- `ADMIN_ROLE_STRATEGY`: Taken as fully trusted. Expected to be the strategy registry.
- `ADMIN_ROLE_SMART_VAULT_ALLOW_REDEEM`: Fully trusted, as it can assign the aforementioned role to users. Expected to be the SV factory.

Other roles are privileged roles on the system level that can execute some privileged actions:

- `ROLE_SPOOL_ADMIN`: Fully trusted as it has the highest privilege in the ecosystem (e.g., manipulate the USD price feed, assign other roles).
- `ROLE_EMERGENCY_WITHDRAWAL_EXECUTOR`: Can withdraw funds in case of emergency to the emergency wallet. Assumed to be fully trusted and to use its powers only in case of emergency.
- `ROLE_STRATEGY_APY_SETTER`: Trusted as APYs directly affect allocations. Expected to only provide meaningful values if necessary.
- `ROLE_PAUSER` and `ROLE_UNPAUSER`: These roles can pause/unpause the system. Hence, they should be trusted, otherwise, users' funds can be trapped in the system.
- `ROLE_REALLOCATOR`: An address holding this role is privileged as it can call `reallocate()` and provide the reallocation parameters which should be set correctly. Fully trusted since bad reallocation parameters could be provided.
- `ROLE_DO_HARD_WORKER`: Fully trusted, as it is capable of calling `doHardWork()`. Similar, to the above.
- `ROLE_RISK_PROVIDER`: The smart contract having this role should be trusted and using untrusted risk providers should be avoided, as it feeds in risk scores for each strategy which ultimately affects the allocation of an SV.
- `ROLE_REWARD_POOL_ADMIN`: Fully trusted for the incentive mechanism.

Further, some roles are specific to a smart vault:

- `ROLE_SMART_VAULT_ADMIN`: Generally trusted. Can assign smart vault-specific roles (e.g., for the allow list guard). Also, can redeem for users (to the users).
- `ROLE_GUARD_ALLOWLIST_MANAGER`: This role is in charge of maintaining allow lists for an SV; hence, capable of bricking the vault. Therefore, it should be fully trusted if it is used.

Last, some roles describe the properties of smart vaults:

- `ROLE_SMART_VAULT_ALLOW_REDEEM`: Fully trusted. A malicious user holding this role can redeem the assets of other users and block them from receiving the planned yields.

External Users: Untrusted and could act maliciously.

We assume the users holding privileged roles, e.g., doHardWorker and reallocator, calculate the optimal and correct slippages off-chain, before feeding them into the system. We further assume that the admin does not add malicious tokens to the `AssetGroupRegistry`.

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	1

- Read-only Reentrancy Acknowledged

5.1 Read-only Reentrancy

Design Low Version 1 Acknowledged

CS-SpoolV2-024

It can be possible to construct examples where certain properties of the SV mismatch reality. For example, during reallocations, a temporary devaluation of SVTs occurs due to SSTs being released. Due to reentrancy possibilities, certain values retrieved could be inaccurate (e.g. SV valuation).

Acknowledged:

While the read-only reentrancy does directly affect on the protocol, it could affect third parties. Spool replied:

```
The mentioned view functions are not intended to be used while the
reallocation is in progress.
```

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	1

- Lack of Access Control in recoverPendingDeposits() Code Corrected

High-Severity Findings	7

- DOS Synchronization by Dividing With Zero Redeemed Shares Code Corrected
- DOS on Deposit Synchronization Code Corrected
- Donation Attack on SST Minting Code Corrected
- Donation Attack on SVT Minting Code Corrected
- Flushing Into Ongoing DHW Leading to Loss of Funds Code Corrected
- No Deposit Due to Reentrancy Into redeemFast() Code Corrected
- Wrong Slippage Parameter in Curve Deposit Code Corrected

Medium-Severity Findings	5

- Curve LP Token Value Calculation Can Be Manipulated Code Corrected
- Deposits to Vault With Only Ghost Strategies Possible Code Corrected
- Ghost Strategy Disables Functionality Code Corrected
- Inconsistent Compound Strategy Value Code Corrected
- Strategy Value Manipulation Code Corrected

Low-Severity Findings	19

- Distribution to Ghost Strategy Code Corrected
- Lack of Access Control for Setting Extra Rewards Code Corrected
- Wrong Error IdleStrategy.beforeRedeemalCheck() Code Corrected
- Access Control Not Central to Access Control Contract Specification Changed
- Asset Decimal in Price Feed Code Corrected
- Bad Event Emissions Code Corrected
- Broken Conditions on Whether Deposits Have Occurred Code Corrected
- Deposit Deviation Can Be Higher Than Expected Code Corrected
- Inconsistent Handling of Funds on Strategy Removal Code Corrected
- Misleading Constant Name Code Corrected
- Missing Access Control in Swapper Code Corrected
- Missing Event Fields Code Corrected
- No Sanity Checks on Slippage Type Code Corrected
- Precision Loss in Notional Finance Strategy Code Corrected
- Redemption Executor Code Corrected

- State Inconsistencies Possible Code Corrected
- Unused Functions Code Corrected
- Unused Variable Code Corrected
- Validation of Specification Code Corrected

Informational Findings	9

- Reverts Due to Management Fee Code Corrected
- Simplifying Performance Fees Code Corrected
- Strategy Removal for an SV Possible That Does Not Use It Code Corrected
- Errors in NatSpec Specification Changed
- Distinct Array Lengths Code Corrected
- Gas Optimizations Code Corrected
- Nameless ERC20 Code Corrected
- NFT IDs Code Corrected
- Tokens Can Be Enabled Twice Code Corrected

6.1 Lack of Access Control in `recoverPendingDeposits()`

Security Critical Version 3 Code Corrected

CS-SpoolV2-039

`DepositManager.recoverPendingDeposits()` has no access control (instead of being only callable by the SV manager). Thus, it allows arbitrary users to freely specify the arguments passed to the function. Ultimately, funds from the master wallet can be stolen.

Code corrected:

Access control was added. Now, only `ROLE_SMART_VAULT_MANAGER` can access the function.

6.2 DOS Synchronization by Dividing With Zero Redeemed Shares

Security High Version 1 Code Corrected

CS-SpoolV2-001

`_sharesRedeemed` describes the SSTs redeemed by an SV. That value could be zero due to rounding. Hence,

```
uint256 withdrawnAssets =
                _assetsWithdrawn[strategy][dhwIndex][j] * strategyShares[i] / _sharesRedeemed[strategy][dhwIndex];
```

could be a division by zero.

Consider the following scenario:

1. Many deposits are made to an SV.

2. The attacker makes a 1 SVT wei withdrawal.
3. The attacker flushes the SV.
4. The redeemed SSTs are computes as `strategyWithdrawals[i] = strategyShares * withdrawals / totalVaultShares`. `strategyShares` corresponds to the shares held by the SV. Hence if the SV's balance of SSTs is lower than the total supply of SSTs (recall, the withdrawal is 1), the shares to be withdrawn is 0.
5. The withdrawal manager passes it to the strategy registry which then stores these values in `_sharesRedeemed`.
6. No other SV tries to withdraw.
7. The division reverts on synchronization.

Ultimately, funds will be locked and SVs could be DOSed.

Code corrected:

Now, in every iteration of the loop in `StrategyRegistry.claimWithdrawals()`, it is checked whether the strategy shares to be withdrawn from the SV (`strategyShares`) are non-zero. In the case of `strategyShares` being zero, the iteration is skipped. If not the case, `_sharesRedeemed > 0` will hold. That is because it is the sum of all SV withdrawals. In other words, `strategyShares_SV > 0 => _sharesRedeemed > 0`.

6.3 DOS on Deposit Synchronization

Security | High | Version 1 | Code Corrected

CS-SpoolV2-002

After the DHW of an SV's to-sync flush cycle, the SV must be synced. The deposit manager decides, based on the value of the deposits at DHW, how many of the minted SSTs will be claimable by the SV. It is computed as follows:

```
result.sstShares[i] = atDhw.sharesMinted * depositedUsd[0] / depositedUsd[1];
```

The `depositedUsd` has the total deposit of the vault in USD at index zero while at index 1 the total deposits of all SVs are aggregated.

To calculate `result.sstShares[i]` the following condition should be met:

```
/// deposits = _vaultDeposits[parameters.smartVault][parameters.bag[0]][0];
deposits > 0 && atDhw.sharesMinted > 0
```

which means that the first asset in the asset group had to be deposited and that at least one SST had to be minted. Given very small values and front-running DHWs with donations that could be achieved. Ultimately, a division-by-zero could DOS the synchronization.

Consider the following scenario:

1. Only withdrawals occur on a given strategy.
2. An attacker sees a DHW incoming for that strategy.
3. The attacker frontruns the transaction and makes a minor deposit so that `deposits > 0` holds. Additionally, the `assetToUsdCustomPriceBulk()` should return `0` which is possible due to rounding. See the following code in `UsdPriceFeedManager.assetToUsdCustomPrice`:

```
assetAmount * price / assetMultiplier[asset];
```

Under the condition that `assetAmount * price` is less than `assetMultiplier` (e.g. 1 wei at 0.1 USD for a token with 18 decimals), that will return 0.

4. Additionally, the attacker donates an amount so that `Strategy.doHardWork()` so that 1 wei SST will be minted (note that the Strategy mints based on the balances and does not receive the amount that were deposited).
5. Finally, DHW is entered and succeeds with 1 minted share.
6. The vault must sync. However, it reverts due to `depositedUsd[1]` being calculated as `0`.

Ultimately, an attacker could cheaply attack multiple SVs under certain conditions.

Code corrected:

`deposits > 0` has been replaced by checking whether there are any deposits made to any of the underlying assets. Additionally, a condition skips the computation (and some surrounding ones) in case the deposited value is zero.

6.4 Donation Attack on SST Minting

Security | High | Version 1 | Code Corrected

CS-SpoolV2-003

The SSTs are minted on DHW and based on the existing value. However, it is possible to donate (e.g. aTokens to the Aave strategy) to strategies so that deposits are minting no shares.

A simple attack may cause a loss in funds. Consider the following scenario:

1. A new strategy is deployed.
2. 1M USD is present for the DHW (value was zero since it is a new strategy).
3. An attacker donates 1 USD in underlying of the strategy (e.g. aToken).
4. DHW on the strategies happens.``usdWorth[0]`` will be non-zero. Hence, the `depositShareEquivalent` will be computed using multiplication with total supply which is 0. Ultimately, no shares will be minted.

Ultimately, funds could be lost.

An attacker could improve on the attack for profit.

1. A new strategy is deployed.
2. An attacker achieves to mint some shares.
3. The attacker redeems the shares fast so that only 1 SST exists.
4. Now, others deposit 1M USD.
5. The attacker donates 1M + 1 USD in yield-bearing tokens to the strategy.
6. No shares are minted due to rounding issues since the `depositSharesEquivalent` and the `withdrawnShares` are zero.

The deposits will increase the value of the strategy so that the attacker profits.

Ultimately, funds could be stolen.

Code corrected:

While the total supply of SSTs is less than `INITIAL_LOCKED_SHARES`, the shares are minted at a fixed rate. `INITIAL_LOCKED_SHARES` are minted to the address `0xdead` so that a minimum amount of shares is enforced. That makes such attacks much more expensive.

6.5 Donation Attack on SVT Minting

Security | High | Version 1 | Code Corrected

CS-SpoolV2-004

The SVTs that are minted on synchronization are minted based on the existing value at the flush. However, it is possible to donate to SVs so that deposits are minting no shares.

A simple attack may cause a loss in funds. Consider the following scenario:

1. A new SV is deployed.
2. 1M USD is flushed (value was zero since it is a new vault).
3. An attacker, holding some SSTs (potentially received through platform fees), donates 1 USD in SSTs (increases the vault value to 1 USD). Frontruns DHW.
4. DHW on the strategies happens.
5. The SV gets synced. The synchronization does not enter the branch of `if (totalUsd[1] == 0)` since the value is 1 USD. The SVTs are minted based on the total supply of SVTs which is zero. Hence, zero shares will be minted.
6. The depositors of the fund receive no SVTs.

Ultimately, funds could be lost.

An attacker could improve on the attack for profit.

1. A new SV is deployed.
2. An attacker achieves to mint some shares.
3. The attacker redeems the shares fast so that only 1 SVT exists.
4. Now, others deposit 1M USD, and the deposits are flushed.
5. The attacker donates 1M + 1 USD in SSTs to the strategy.
6. Assume there are no fees for the SV for simplicity. Synchronization happens. The shares minted for the deposits will be equal to `1 * 1M USD / (1M + 1 USD)` which rounds down to zero.

The deposits will increase the value of the vault so that the attacker profits.

Finally, consider that an attack could technically also donate to the strategy before the DHW so that `totalStrategyValue` is pumped.

Code corrected:

While the total supply of SSTs is less than `INITIAL_LOCKED_SHARES`, the shares are minted at a fixed rate. `INITIAL_LOCKED_SHARES` are minted to the address `0xdead` so that a minimum amount of shares is enforced. That makes such attacks much more expensive.

6.6 Flushing Into Ongoing DHW Leading to Loss of Funds

Security High Version 1 Code Corrected

CS-SpoolV2-005

The DHW could be reentrant due to the underlying protocols allowing for reentrancy or the swaps being reentrant. That reentrancy potential may allow an attacker to manipulate the perceived deposit value in `Strategy.doHardWork()`.

Consider the following scenario:

1. DHW is being executed for a strategy. The deposits are 1M USD. Assume that for example the best off-chain computed path is taken for swaps. An intermediary token is reentrant.
2. The strategy registry communicated the provided funds and the withdrawn shares for the DHW index to the strategy.
3. Funds are swapped.
4. The attacker reenters a vault that uses the strategy and flushes 1M USD. Hence, the funds to deposit and shares to redeem for the DHW changed even though the DHW is already running.
5. The funds will be lost. However, the loss is split among all SVs.
6. However, the next DHW will treat the assets as deposits made by SVs. An attacker could maximize his profit by depositing a huge amount and flushing to the DHW index where the donation will be applied. Additionally, he could try flushing all other SVs with small amounts. The withdrawn shares will be just lost.

To summarize, flushing could be reentered to manipulate the outcome of DHW due to bad inputs coming from the strategy registry.

Code corrected:

Reentrancy protection has been added for this case.

6.7 No Deposit Due to Reentrancy Into `redeemFast()`

Security High Version 1 Code Corrected

CS-SpoolV2-006

The DHW could be reentrant due to the underlying protocols allowing for reentrancy or the swaps being reentrant. That reentrancy potential may allow an attacker to manipulate the perceived deposit value in `Strategy.doHardWork()`.

Consider the following scenario:

1. DHW is executed for a strategy. The deposits are 1M USD. Assume that for example the best off-chain computed path is taken for swaps. An intermediary token is reentrant.
2. DHW checks the value of the strategy, which is 2M USD and fully controlled by the attacker's SV.
3. The DHW swaps the incoming assets. The attacker takes control of the execution.
4. The attacker redeems 1M USD with `redeemFast()`. The strategy's value drops to 1M USD.
5. DHW proceeds, a good swap is made and the funds are deposited into the protocol.
6. DHW retrieves the new strategy value which is now 2M USD.

7. The perceived deposit is now 0 USD due to 2. and 6. However, the actual deposit was 1M USD.

Ultimately, the deposit made is treated as a donation to the attacker since zero shares are minted.

Similarly, such attacks are possible when redeeming SSTs with `redeemStrategyShares()`.

Also, the attack could occur in regular protocol interactions if the underlying protocol has reentrancy possibilities (e.g. protocol itself has a swapping mechanism). In such cases, the reallocation could be vulnerable due to similar reasons in `depositFast()`.

Code corrected:

Reentrancy protection has been added for this case.

6.8 Wrong Slippage Parameter in Curve Deposit

Correctness | High | Version 1 | Code Corrected

CS-SpoolV2-007

`Curve3CoinPoolBase._depositToProtocol()` calculates an offset for the given `slippage` array. This offset is then passed - without the actual array - into the function `_depositToCurve()`. The `add_liquidity()` function of the Curve pool is then called with this offset parameter, setting the slippage to always either 7 or 10:

```
uint256 slippage;
if (slippages[0] == 0) {
    slippage = 10;
} else if (slippages[0] == 2) {
    slippage = 7;
} else {
    revert CurveDepositSlippagesFailed();
}

_depositToCurve(tokens, amounts, slippage);
```

```
pool.add_liquidity(curveAmounts, slippage);
```

DHW calls can be frontrun to extract almost all value of this call.

Code corrected:

`Curve3CoinPoolBase._depositToProtocol()` now passes the correct value of the `slippages` array to `_depositToCurve()`.

6.9 Curve LP Token Value Calculation Can Be Manipulated

Correctness | Medium | Version 1 | Code Corrected

CS-SpoolV2-008

`Curve3CoinPoolBase._getUsdWorth()` and `ConvexAlusdStrategy._getTokenWorth()` calculate the value of available LP tokens in the following way:

```
for (uint256 i; i < tokens.length; ++i) {
    usdWorth += priceFeedManager.assetToUsdCustomPrice(
        tokens[i], _balances(assetMapping.get(i)) * lpTokenBalance / lpTokenTotalSupply, exchangeRates[i]
    );
}
```

This is problematic as the pool exchanges tokens based on a curve (even though it is mostly flat). Consider the following scenario (simplified for 2 tokens):

- The pool's current A value is 2000.
- The pool holds 100M of each token.
- The total LP value according to the given calculation is 200M USD.
- A big trade (200M) changes the holdings of the pool in the following way:
    - 300M A token
    - ~160 B token
- The total LP value according to the given calculation is now ~300M USD.

A sandwich attack on `StrategyRegistry.doHardWork()` could potentially skew the value of a strategy dramatically (although an enormous amount of tokens would be required due to the flat curve of the StableSwap pool). This would, in turn, decrease the number of shares all deposits in this DHW cycle receive, shifting some of this value to the existing depositors.

All in all, an attacker must hold a large position on the strategy, identify a DHW that contains a large deposit to the strategy and then sandwich attack it with a large amount of tokens. The attack is therefore rather unlikely but has a critical impact.

Code corrected:

The Curve and Convex strategies now contain additional slippage checks for the given Curve pool's token balances (and also the Metapool's balances in the case of `ConvexAlusdStrategy`) in `beforeDepositCheck`. As this function is always called in `doHardWork`, the aforementioned sandwich attack can effectively be mitigated by correctly set slippages. It is worth noting that these slippages can be set loosely (to prevent the transaction from failing) as some less extreme fluctuations cannot be exploited due to the functionality of the underlying Curve 3pool.

6.10 Deposits to Vault With Only Ghost Strategies Possible

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

CS-SpoolV2-009

Governance can remove strategies from vaults. It happens by replacing the strategy with the ghost strategy. However, if an SV has only ghost strategies, deposits to it are still possible (checking the deposit ratio always works since the ideal deposit ratio is 0 or due to the "one-token" mechanics). However, flushing would revert. User funds could unnecessarily be lost. Similarly, redemptions would be possible. Additionally, synchronization could occur if the ghost strategy is registered (which should not be the case).

Code corrected:

The case was disallowed by making a call to the newly implemented function `_nonGhostVault`, which also gets called when redeeming and flushing. Hence, depositing to, redeeming and flushing from a ghost vault is disabled.

6.11 Ghost Strategy Disables Functionality

Correctness | Medium | Version 1 | Code Corrected

CS-SpoolV2-010

Governance can remove strategies from SVs by replacing them with the ghost strategy. This may break `redeemFast()` on SVs due to `StrategyRegistry.redeemFast()` trying to call `redeemFast()` on the ghost strategy.

Code corrected:

The iteration is skipped in case the current strategy is the ghost strategy. Hence, the function is not called on the ghost strategy anymore.

6.12 Inconsistent Compound Strategy Value

Correctness | Medium | Version 1 | Code Corrected

CS-SpoolV2-011

`CompoundV2Strategy` calculates the yield of the last DHW epoch with `exchangeRateCurrent()` which returns the supply index up until the current block:

```
uint256 exchangeRateCurrent = cToken.exchangeRateCurrent();

baseYieldPercentage = _calculateYieldPercentage(_lastExchangeRate, exchangeRateCurrent);
_lastExchangeRate = exchangeRateCurrent;
```

On the other hand, `_getUsdWorth()` calculates the value of the whole strategy based on the output of `_getcTokenValue()` which in turn calls Compound's `exchangeRateStored()`:

```
if (cTokenAmount == 0) {
    return 0;
}

// NOTE: can be outdated if noone interacts with the compound protocol for a longer period
return (cToken.exchangeRateStored() * cTokenAmount) / MANTISSA;
```

This behavior has been acknowledged with a comment in the code. However, it can become problematic in the following scenario:

- The compound protocol did not have interaction over a longer period.
- A user has deposited into a `SmartVault` that contains the `CompoundV2Strategy`.
- In the `doHardWork()` call, the strategy's `_compound` function does not deposit to the protocol (i.e. the index is not updated in Compound). This can happen in the following cases:
    - No COMP rewards have been accrued since the last DHW.
    - The `ROLE_DO_HARD_WORKER` role has not supplied a `SwapInfo` to the strategy's `_compound` function.

In this case, the following line in `Strategy.doHardWork()` relies on outdated data:

```
usdWorth[0] = _getUsdWorth(dhwParams.exchangeRates, dhwParams.priceFeedManager);
```

`usdWorth[0]` is then used to determine the number of shares minted for the depositors of this DHW epoch:

```
mintedShares = usdWorthDeposited * totalSupply() / usdWorth[0];
```

Since some interest is missing from this value, the depositors receive more shares than they are eligible for, giving them instant gain.

Code corrected:

`_getcTokenValue()` now retrieves the current exchange rate instead of the stale one.

6.13 Strategy Value Manipulation

Security | Medium | Version 1 | Code Corrected

CS-SpoolV2-043

`SmartVaultManager.redeemFast()` allows users to directly redeem their holdings on the underlying protocols of the strategies in a vault. The function calls to `Strategy.redeemFast()` in which the `totalUsdValue` of the respective strategy is updated.

This value can be manipulated in several ways:

- If the given Chainlink oracle for one of the assets is not returning a correct value, the user can provide `exchangeRateSlippages` that would allow these false exchange rates to be used.
- If the strategy's correct value calculation depends on slippage values to be non-manipulatable, the strategy's value can be changed with a sandwich attack as there is no possibility to enforce correct behavior (see, for example, Curve LP token value calculation can be manipulated). Furthermore, this sandwich attack is particularly easy to perform as the user is in control of the call that has to be sandwiched (i.e., all calls can be performed in one transaction).

A manipulated strategy value is problematic for `SmartVaultManager.reallocate()` because the `totalUsdValue` is used to compute how much value is moved/matched between strategies.

Note: This issue was disclosed by the Spool team during the review process of this report.

Code corrected:

`reallocate()` now computes the value of strategies directly, rather than relying on `totalUsdValue` (which is now completely removed from the codebase),

6.14 Distribution to Ghost Strategy

Correctness | Low | Version 4 | Code Corrected

CS-SpoolV2-040

`DepositManager._distributeDepositSingleAsset` assigns all dust to the first strategy in the given array. There are no checks present to ensure that this strategy is not the Ghost strategy.

Code corrected:

The code has been adjusted to add dust to the first strategy with a deposit.

6.15 Lack of Access Control for Setting Extra Rewards

Correctness Low Version 3 Code Corrected

CS-SpoolV2-041

`setExtraRewards()` has no access control. However, an attacker could set the extra rewards to false for a long time. Then, after their SV's first deposit to the strategy, could set it to true, so that they receive more compounded yield than they should have received.

Code corrected:

The code has been corrected.

6.16 Wrong Error `IdleStrategy.beforeRedeemalCheck()`

Correctness Low Version 3 Code Corrected

CS-SpoolV2-028

The range-check in `IdleStrategy.beforeRedeemalCheck()` reverts with the `IdleBeforeDepositCheckFailed` error. However, `IdleBeforeRedeemalCheckFailed` would be the suiting error.

Code corrected:

The correct error is used.

6.17 Access Control Not Central to Access Control Contract

Correctness Low Version 1 Specification Changed

CS-SpoolV2-012

The specification defines that access control should be centralized in `SpoolAccessControl`:

```
All access control is handled centrally via SpoolAccessControl.sol.
```

However, the factory as an `UpgradeableBeacon` implements access control for changing implementation which does not use the central access control contract.

Specification changed:

The documentation has been clarified:

```
Access control is managed on three separate levels:

    - All privileged access pertaining to usage of the platform is handled
      through SpoolAccessControl.sol, which is based on OpenZeppelin's
      AccessControl smart contract

    - Core smart contracts upgradeability is controlled through
      OpenZeppelin's ProxyAdmin.sol

    - SmartVault upgradeability is controlled using OpenZeppelin's
      UpgradeableBeacon smart contract
```

Hence, the access control for upgrading the beacons is now accordingly documented.

6.18 Asset Decimal in Price Feed

Design Low Version 1 Code Corrected

CS-SpoolV2-013

The asset decimals are given as an input parameter in `setAsset()`. Although being cheaper than directly querying `ERC20.decimals()`, it is more prone to errors. Fetching the asset decimals through the `ERC20` interface could reduce such risks.

Code corrected:

`ERC20.decimals()` is now called to fetch the underlying asset decimals.

6.19 Bad Event Emissions

Correctness Low Version 1 Code Corrected

CS-SpoolV2-014

In `StrategyRegistry.redeemFast()`, the `StrategySharesFastRedeemed()` is emitted. The `assetsWithdrawn` parameter of the event will be set to `withdrawnAssets` on every loop iteration. However, that does not correspond to the assets withdrawn from a strategy but corresponds to the assets withdrawn up to the strategy `i`.

Code corrected:

The event takes now `strategyWithdrawnAssets` as a parameter.

6.20 Broken Conditions on Whether Deposits Have Occurred

Correctness Low Version 1 Code Corrected

CS-SpoolV2-015

In `DepositManager.flushSmartVault()`, the condition `_vaultDeposits[smartVault][flushIndex][0] == 0` checks whether at least one wei of the first token in the asset group has been deposited. However, the condition may be imprecise as it could technically be possible to create deposits such that the deposit of the first asset could be zero while the others are non-zero. A similar check is present in `DepositManager.syncDepositsSimulate()` during deposit synchronization.

Note that this would lead to deposits not being flushed and synchronized (ultimately ignoring them). While the user will receive no SVTs for very small deposits in general, the deposits here would be completely ignored. Further, this behavior becomes more problematic for rather large asset groups (given the `checkDepositRatio()` definition).

Code corrected:

The checks have been improved to consider the summation of `_vaultDeposits[smartVault][flushIndex]` to all assets rather than only considering the first asset in the group.

6.21 Deposit Deviation Can Be Higher Than Expected

Design | Low | Version 1 | Code Corrected

CS-SpoolV2-016

The deviation of deposits could be higher than expected due to the potentially exponential dropping relation between the first and last assets. Note that the maximum deviation is the one from the minimum ideal-to-deposit ratio to the maximum ideal-to-deposit ratio. Ultimately, given the current implementation, this maximum deviation could be violated.

Code corrected:

The following mechanism has been implemented. First, a reference asset is found with an ideal weight non-zero (first one found). Then, other assets are compared to that asset. Ultimately, each ratio is in the range of the reference asset.

6.22 Inconsistent Handling of Funds on Strategy Removal

Design | Low | Version 1 | Code Corrected

CS-SpoolV2-018

When a strategy is removed from the strategy registry, the unclaimed assets by SVs are sent to the emergency wallet. However, the funds flushed and unflushed underlying tokens are not (similarly the minted shares are not).

Code corrected:

Consistency was reevaluated. The corner case of an SV with non-flushed deposited assets was handled by introducing a recovery function, namely `DepositManager.recoverPendingDeposits()`. The other cases were specified as intended.

6.23 Misleading Constant Name

Design Low Version 1 Code Corrected

CS-SpoolV2-019

In `SfrxEthHoldingStrategy` the constant `CURVE_ETH_POOL_SFRXETH_INDEX` is used to determine the coin ID in an ETH/frxETH Curve pool. Since the pool trades frxETH instead of sfrxETH, the naming of the constant is misleading.

Code corrected:

Spool has changed `CURVE_ETH_POOL_SFRXETH_INDEX` to `CURVE_ETH_POOL_FRXETH_INDEX`.

6.24 Missing Access Control in Swapper

Security Low Version 1 Code Corrected

CS-SpoolV2-020

The `Swapper.swap()` function can be called by anyone. If a user accidentally sends funds to the swapper or if it was called with a misconfigured `SwapInfo` struct, the remaining funds can be sent to an arbitrary address by anyone.

Code corrected:

Spool has introduced a new function `_isAllowedToSwap`, which checks if the caller to `Swapper.swap()` holds `ROLE_STRATEGY` or `ROLE_SWAPPER` role. `ROLE_SWAPPER` must now additionally be assigned to the `DepositSwap` contract.

6.25 Missing Event Fields

Design Low Version 1 Code Corrected

CS-SpoolV2-021

The events `PoolRootAdded` and `PoolRootUpdated` of `IRewardPool` do not include added root (and previous root in the case of `PoolRootUpdated`).

Code corrected:

The code has been adapted to include the added root.

6.26 No Sanity Checks on Slippage Type

Correctness Low Version 1 Code Corrected

CS-SpoolV2-022

Some functions do not verify the value in slippages[0]. Some examples are:

1. `IdleStrategy._emergencyWithdrawImpl` does not check if `slippages[0] == 3`.
2. `IdleStrategy._compound` does not check if `slippages[0] < 2`.

Code corrected:

All relevant functions now check that `slippages[0]` has the expected value and revert otherwise.

6.27 Precision Loss in Notional Finance Strategy

Correctness | Low | Version 1 | Code Corrected

CS-SpoolV2-023

`NotionalFinanceStrategy._getNTokenValue()` calculates the value of the strategy's nToken balance in the following way:

```
(nTokenAmount * uint256(nToken.getPresentValueUnderlyingDenominated()) / nToken.totalSupply())
    * underlyingDecimalsMultiplier / NTOKEN_DECIMALS_MULTIPLIER;
```

`nToken.getPresentValueUnderlyingDenominated()` returns values similar or notably smaller than `nToken.totalSupply`. On smaller amounts of nToken balances, precision is lost in this calculation.

Code corrected:

The implementation of `_getNTokenValue()` has been changed to the following:

```
(nTokenAmount * uint256(nToken.getPresentValueUnderlyingDenominated()) * _underlyingDecimalsMultiplier)
    / nToken.totalSupply() / NTOKEN_DECIMALS_MULTIPLIER;
```

All divisions are now performed after multiplications, ensuring that precision loss is kept to a minimum.

6.28 Redemption Executor

Correctness | Low | Version 1 | Code Corrected

CS-SpoolV2-025

Redemptions will enter `WithdrawalManager._validateRedeem()` that will run `Withdrawal` guards with the redeemer as the executor. However, when called through `SmartVaultManager.redeemFor()` the actual executor is a user with `ROLE_SMART_VAULT_ALLOw_REDEEM`. This address is neither sent through `RedeemBag` nor `RedeemExtras`. In this case, `WithdrawalManager._validateRedeem()` runs the guards with the executor being set as the redeemer.

Code corrected:

The executor is now more accurately handled.

6.29 State Inconsistencies Possible

Correctness | Low | Version 1 | Code Corrected

CS-SpoolV2-042

`SmartVaultManager.redeemFast()` allows users to redeem their holdings directly from underlying protocols. In contrast to `StrategyRegistry.doHardWork()`, users can set the slippages for withdrawals themselves which could potentially lead to users setting slippages that do not benefit them.

This is problematic because the amount of shares actually redeemed in the underlying protocol is not accounted for. Since some protocols redeem on a best-effort basis, fewer shares may be redeemed than requested (this is, for example, the case in the `YearnV2Strategy`). If this happens, and the user sets wrong slippages, the protocol burns all SVTs the user requested but does not redeem all the respective shares of the underlying protocol leading to an inconsistency that unexpectedly increases the value of the remaining SVTs.

Code corrected:

The code for the Yearn V2 strategy has been adapted to check for full redeemals.

6.30 Unused Functions

Design | Low | Version 1 | Code Corrected

CS-SpoolV2-026

The following functions of `MasterWallet` are not used:

1. `approve`
2. `resetApprove`

Code corrected:

These functions have been removed.

6.31 Unused Variable

Design | Low | Version 1 | Code Corrected

CS-SpoolV2-044

1. `DepositSwap.swapAndDeposit()` takes an input array of `SwapInfo`, which contains `amountIn`. This function however takes an input array of `inAmounts`.
2. The mapping `DepositManager._flushShares` is defined as internal and its subfield `flushSvtSupply` is never read.
3. `WithdrawalManager._priceFeedManager` is set but never used.

Code corrected:

The code has been adapted to remove the unused variables.

6.32 Validation of Specification

Design | Low | Version 1 | Code Corrected

CS-SpoolV2-027

The specification of an SV is validated to ensure that the SV works as the deployer would expect it. However, some checks could be missing. Examples of such potentially missing checks are:

1. Request type validation for actions: Only allow valid request types for action (some request types are not used for some actions).
2. If static allocations are used, specifying a risk provider, a risk tolerance or an allocation provider may not be meaningful as they are not stored. Similarly, if only one strategy is used it could be meaningful to enforce a static allocation.
3. Static allocations do not enforce the 100% rule that the allocation providers enforce. For consistency, such a property could be enforced.

Code corrected:

The code has been adapted to enforce stronger properties on the specification.

6.33 Distinct Array Lengths

Informational | Version 1 | Code Corrected

CS-SpoolV2-029

Some arrays that are iterated over jointly can have distinct lengths which lead to potentially unused values and a result different from what was expected due to human error or a revert.

Examples of a lack of array length checks in the specification when deploying an SV through the factory are:

1. `actions` and `actionRequestTypes` in `ActionManager.setActions()` may have distinct length. Some request-type values may remain unused.
2. Similarly, this holds for guards.
3. In the strategy registry's `doHardWork()`, the base yields array could be longer than the strategies array.
4. In `assetToUsdCustomPriceBulk()` the array lengths could differ. When used internally, that will not be the case while when used externally that could be the case. The semantics of this are unclear.
5. `calculateDepositRatio()` and `calculateFlushFactors()` in `DepositManager` are similar to 4.

Code corrected:

The missing checks in 1-3 have been added. However, for 4-5 which are view functions, Spool decided to keep as is.

6.34 Errors in NatSpec

Informational | Version 1 | Specification Changed

At several locations, the NatSpec is incomplete or missing. The following is an incomplete list of examples:

1. `IGuardManager.RequestContext`: not all members of the struct are documented.
2. `IGuardManager.GuardParamType`: not all items of the `enum` are documented.
3. `_stateAtDhw` has no NatSpec.
4. `IDepositManager.SimulateDepositParams`: documentation line of `bag` mentions `oldTotalSVTs` along with `flush index` and `lastDhwSyncedTimestamp`.
5. `StrategyRegistry._dhwAssetRatios`: is a mapping to the asset ratios, as the name suggests; however, the spec mentions exchange rate.
6. `StrategyRegistry._updateDhwYieldAndApy()`: it only updates APY and not the yield for a given `dhwIndex` and `strategy`.
7. `RewardManager.addToken()`: callable only by either `DEFAULT_ADMIN_ROLE` or `ROLE_SMART_VAULT_ADMIN` of an SV and not "reward distributor" as mentioned in the specification.

Specification changed:

The NatSpec was improved. Naming of `StrategyRegistry._updateDhwYieldAndApy()` was changed to `_updateApy()`.

6.35 Gas Optimizations

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

CS-SpoolV2-031

Some parts of the code could be optimized in terms of gas usage. Reducing gas costs may improve user experience. Below is an incomplete list of potential gas inefficiencies:

1. `claimSmartVaultTokens()` could early quit if the claimed NFT IDs are claimed. Especially, that may be relevant in cases in the redeem functions where a user can specify W-NFTs to be withdrawn.
2. The `FlushShares` struct has a member `flushSvtSupply` that is written when an SV is flushed. However, that value is never used and hence the storage write could be removed to reduce gas consumption.
3. `swapAndDeposit()` queries the token out amounts with `balanceOf()`. `Swapper.swap()` returns the amounts. However, the return value is unused.
4. `RewardManager()` inherits from `ReentrancyGuardUpgradeable`. It further is initializable, initializing only the reentrancy guard state. However, reentrancy locks are not used.
5. The constructor of `SmartVaultFactory` checks whether the implementation is `0x0`. However, in `UpgradeableBeacon` an `isContract()` check is made.
6. In `redeemFast()` the length of the NFT IDs and amounts is ensured to be equal. However, in `DepositManager.claimSmartVaultTokens()` the same check is made.
7. In the internal function `SmartVaultManager._redeem()`, the public method `flushSmartVault()` is used. The `_onlyRegisteredSmartVault()` check will be performed twice.
8. `IStrategy.doHardwork()` could return the `assetRatio()` with the DHW info so that a staticcall to `IStrategy.assetRatio()` in `StrategyRegistry.doHardwork()` is not needed.

9. In `_validateRedeem()` the balance of the redeemer is checked. However, that check is made when the SVTs are transferred to the SV.
10. The input argument `vaultName_` in `SmartVault.initialize` can be defined as calldata.
11. `SmartVault.transferFromSpender()` gets called only by `WithdrawalManager` with `spender` equal to `from`.
12. `SmartVault.burnNFT()` checks that the owner has enough balance to burn. The same condition is later checked as it calls into `_burnBatch`.
13. The struct `SmartVaultSpecification` in `SmartVaultFactory` has an inefficient ordering of elements. For example, by moving `allowRedeemFor` below `allocationProvider` its storage layout decreases by one slot.
14. The struct `IGuardManager.GuardDefinition` shows an inefficient ordering.
15. Where `ReallocationLib.doReallocation()` computes `sharesToRedeem`, it can replace `totals[0] - totals[1]` with `totalUnmatchedWithdrawals`.
16. `SmartVaultManager._simulateSync()` increments the memory variable `flushIndex.toSync` which is neither used later nor returned as a return value.
17. `SmartVaultManager._redeem()` calls `flushSmartVault`. However, the internal function `_flushSmartVault` could directly be called.
18. `SmartVaultManager._redeem()` accesses the storage variable `_flushIndexes[bag.smartVault]` twice. It could be cached and reused once.
19. `StrategyRegistry.doHardWork()` reads `_assetsDeposited[strategy][dhwIndex][k]` twice. Similar to the issue above, it could be cached.
20. `UsdPriceFeedManager.assetToUsdCustomPriceBulk()` could be defined as external.
21. `WithdrawalManager.claimWithdrawal()` can be defined as an external function.
22. `RewardManager.forceRemoveReward()` eventually removes `rewardConfiguration[smartVault][token]`, which is already removed in `_removeReward()`.
23. `RewardPool.claim()` can simply set `rewardsClaimed[msg.sender][data[i].smartVault][data[i].token]` to `data[i].rewardsTotal`.
24. `SmartVaultManager._simulateSyncWithBurn()` can fetch `fees` after checking all DHWs are completed.
25. Strategies are calling `AssetGroupRegistry.listAssetGroup` in multiple functions. The token addresses could instead be cached in the strategy the avoid additional external calls.
26. `REthHoldingStrategy._emergencyWithdrawImpl()` reverts if `slippages[0] != 3`. This check can be accomplished at the very beginning of the function.
27. `REthHoldingStrategy._depositInternal()` can have an early return if `amounts[0] < 0.01 ETH`. It is mentioned in its documentations, that the smallest deposit value should be `0.01 ETH`
28. The input parameter `strategyName_` of `SfrxEthHoldingStrategy.initialize()` can be defined as calldata.
29. `Strategy` calls `_swapAssets` and then loads the balances of each token again. Since `_swapAssets` is not used in all of the strategies, the subsequent `balanceOf` calls by checking if `_swapAssets` actually performed any actions.

Code corrected:

While not every improvement has been implemented, gas consumption has been reduced.

6.36 NFT IDs

Informational | Version 1 | Code Corrected

CS-SpoolV2-032

The NFT IDs are in the following ranges:

- D-NFTs: `[1, 2**255 - 2]`
- W-NFTs: `[2**255 + 1, 2**256 - 2]`

Note that the ranges could be technically increased. Further, in theory, there could be many more withdrawals than deposits. The sizes do not reflect that. However, in practice, a scenario with such a large number of redemptions does not seen to be realistic. Additionally, `getMetaData()` will return deposit meta data for ID `0` and `2**255 - 1`. However, these are not valid deposit NFT IDs. Similarly, the function returns metadata for invalid withdrawal NFTs. However, these remain empty. Last, technically one could input such IDs for burn (using 0 shares burn). Similarly, one could burn others' NFTs (0 amounts).

Ultimately, the effects of this may create confusion.

Code corrected:

The range of valid NFT-IDs has been increased.

6.37 Nameless ERC20

Informational | Version 1 | Code Corrected

CS-SpoolV2-033

The SVT ERC-20 does not have a name. Specifying a name may help third-party front-ends (e.g. Etherscan) to display useful information to users for a better user experience.

Code corrected:

The SVT now has a name and symbol for its ERC-20. Additionally, the ERC-1155 has a URI now.

6.38 Reverts Due to Management Fee

Informational | Version 1 | Code Corrected

CS-SpoolV2-035

An SV can have a management fee that is computed as

```
totalUsd[1] * parameters.fees.managementFeePct * (result.dhwTimestamp - parameters.bag[1])
                / SECONDS_IN_YEAR / FULL_PERCENT;
```

It could be the case that more than one year has passed between the two timestamps. Ultimately the condition

```
parameters.fees.managementFeePct * (result.dhwTimestamp - parameters.bag[1]) > SECONDS_IN_YEAR * FULL_PERCENT
```

could hold if at least around 20 years have passed. That would make the fee greater than the total value.

Ultimately,

```
result.feeSVTs = localVariables.svtSupply * fees / (totalUsd[1] - fees);
```

could revert.

Code corrected:

The code was corrected by limiting the dilution of SVTs so that the subtraction cannot revert.

6.39 Simplifying Performance Fees

Informational | Version 1 | Code Corrected

CS-SpoolV2-036

The performance fees could further be simplified to

```
strategyUSD * interimYieldPct / (1 + interimYieldPct * (1-totalPlatformFees))
```

which is equivalent to the rather complicated computations made in the current implementation.

Code improved:

The readability of the code has been improved by simplifying the computation.

6.40 Strategy Removal for an SV Possible That Does Not Use It

Informational | Version 1 | Code Corrected

CS-SpoolV2-037

The event `StrategyRemovedFromVaults` gets emitted for a strategy even if the SV does not use the strategy.

Code corrected:

The event is now emitted per vault that uses the strategy. Furthermore, the name of this event has been changed to `StrategyRemovedFromVault`.

6.41 Tokens Can Be Enabled Twice

Informational | Version 1 | Code Corrected

CS-SpoolV2-038

In `AssetGroupRegistry`, the same token can be allowed multiple times. Although it does not make any difference, regarding the internal state, it emits an event of `TokenAllowed` again.

Code corrected:

The event is not emitted anymore in such cases.

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 Packed Arrays With Too Big Values Could DOS the Contract

Informational Version 1 Risk Accepted

CS-SpoolV2-034

The packed array libraries could technically DOS the system due to reverts on too high values. For storing DHW indexes this is rather unlikely given the expectation that it will be only called every day or two (would generally require many DHWs). It is also expected that the withdrawn strategy shares will be less than or equal to `uint128.max`. Though theoretically speaking DOS on flush is possible, the conditions on the practical example are very unlikely.

Risk accepted:

Spool replied:

```
We agree that theoretically packed arrays could overflow and revert, however,
we did some calculations and this should never happen in practice.
```

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 Bricked Smart Vaults

Note Version 1

Some Smart Vaults may be broken when they are deployed.

An example of such broken SVs could be that a malicious SV owner could deploy a specification with guards that allow deposits but disallow withdrawals (e.g. claiming SVT). Moreover, the owner may deploy a specification that is seemingly safe from the user's perspective while then maliciously changing the behaviour of the guard (e.g. removing from the allow list, upgrading the guard).

Another example could be where transfers between users could be allowed while the recipient could be blocked from redemption.

Similarly, actions or other addresses could be broken.

Users, before interacting with an SV, should be very carefully studying the specification. Similarly, deployers should be knowledgeable about the system so that they can create proper specifications to not create bricked vaults by mistake.

8.2 Curve Asset Ratio Slippage

Note Version 1

Curve strategies return the current balances of the pool in their `assetRatio()` functions. These ratios are cached once at the end of each DHW. For all deposits occurring during the next DHW epoch, the same ratios are used although the ratios on the pool might change during that period. It is therefore possible, that the final deposit to the protocol incurs a slight slippage loss.

Given the size and parameters of the pools, this cost should be negligible in most cases.

8.3 DOS Potential for DHWs Due to External Protocols

Note Version 1

DHWs could be blocked in case external protocols cannot accept or return funds. For example, if Aave v2 or Compound v2 have 100% utilization, DHWs could be blocked if withdrawals are necessary. This can in turn prolong the time until deposits earn interest and become withdrawable again.

8.4 ERC-1155 `balanceOf()`

Note Version 1

The `balanceOf()` function of the SV's ERC-1155 returns `1` if the user has any balance. The standard defines that the function should return the balance which in this case is defined as the "fractional

balance". Depending on the interpretation of EIP-1155, this could still match the standard. However, such a deviation from the "norm" could break integrations.

8.5 Management Fee Considerations

Note Version 1

Users should be aware that the management fee is not taken based on the vault value at the beginning of the flush cycle but at the end of it (hence, including the potential yield of strategies, however not including fresh deposits).

8.6 Ordering of Swaps in Reallocations and Swaps

Note Version 1

The privileged user doing reallocation or swaps (e.g. the one holding `ROLE_DO_HARD_WORKER`) should take an optimal path when performing the swaps, as depositing to/withdrawing from a strategy changes its value.

Also, note that some strategies could be affected more by bad trades due to the swaps being performed in the order of the strategies. For example:

1. `depositFast()` to the first strategy happens. The swap changes the price in the DEX.
2. `depositFast()` to the second strategy happens. The swap works at a worse price than the first strategy.

Ultimately, some deposits could have worse slippage.

8.7 Price Aggregators With More Than 18 Decimals

Note Version 1

Setting price aggregators with more than 18 decimals will revert in `UsdPriceFeedManager.setAsset()`. Such are not supported by the system.

8.8 Public Getter Functions

Note Version 1

Users should be aware that some public getters provide only meaningful results with the correct input values (e.g. `getClaimedVaultTokensPreview()`). When used internally, it is ensured that the inputs are set such that the results are meaningful.

8.9 Reentrancy Potential

Note Version 1

While reentrancy protection was implemented in Version 2 of the code, some potential for reentrancy-based attacks may still exist. However, it highly depends on the underlying strategies. Future unknown strategies could introduce vulnerable scenarios.

An example could be a strategy that swaps both on compounding and on deposits in DHW. If it is possible to manipulate the USD value oracle of the strategy (e.g. similar to Curve), then one could effectively generate a scenario that creates 0-deposits or "bypasses" the pre-deposit/redeemal checks.

8.10 Reward Pool Updates

Note Version 1

The `ROLE_REWARD_POOL_ADMIN` should be very careful, when updating the root of a previous cycle (if necessary), as it could break the contract for certain users.

8.11 Slippage Loss in alUSD Strategy

Note Version 1

`ConvexAlusdStrategy` never invests alUSD into the corresponding Curve pool. This can result in a slight slippage loss due to unbalanced deposits. Both deposits and withdrawals are subject to this problem.

The loss is negligible up to a certain amount of value deposited/withdrawn. After that, there is no limit though. At the time this report was written, a withdrawal of 10M LP tokens to 3CRV incurs a loss of roughly 25%.

8.12 Special Case: Compound COMP Market

Note Version 1

Compound v2 currently has an active market for the COMP token. In this case, deposits to the `CompoundV2Strategy` would be absorbed by the `_compound()` function if a `compoundSwapInfo` has been set for the strategy. The correct handling is therefore completely dependent on the role `ROLE_DO_HARD_WORKER` and is not enforced on-chain.

8.13 Unsupported Markets

Note Version 1

Some markets of the supported protocols in Spool V2's strategies might be problematic:

- Aave markets in which the aToken has different decimals than the underlying. While this is not the case for any aToken currently deployed, Aave does not guarantee that this will be the case in the future.
- Compound supports fee-taking tokens. If such a market would be integrated into Spool V2, it could be problematic as the `CompoundV2Strategy._depositToCompoundProtocol()` does not account for the return value of Compound's `mint()` function.
- Compound's cETH market is unsupported due to it requiring support for native ETH and hence having a different interface than other cTokens.

8.14 Value of the alUSD Strategy's Metapool LP Token Overvalued

Note Version 1

The Curve metapool that is used in the `ConvexAlusdStrategy` allows to determine the value of LP tokens, if only one of the 2 underlying tokens is withdrawn, with the function `calc_withdraw_one_coin()`. This is used in the strategy to determine the value of one token which is then scaled up by the actual LP token amount.

The function, however, does not linearly scale with the amount of LP tokens due to possible slippage loss with higher amounts. The LP tokens are therefore overvalued.