SMART CONTRACT AUDIT REPORT

for

Spool Protocol

Prepared By: Yiqun Chen

PeckShield

March 30, 2022

Document Properties

Client	Spool Protocol
Title	Smart Contract Audit Report
Target	Spool
Version	1.0
Author	Xuxian Jiang
Auditors	Stephen Bie, Patrick Lou, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	March 30, 2022	Xuxian Jiang	Final Release
1.0-rc1	February 5, 2022	Xuxian Jiang	Release Candidate #1

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Yiqun Chen
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the `Spool` protocol, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About Spool

`Spool Protocol` serves as the `DeFi middleware` that allows users to participate in a subset of yield generating protocols in a risk diversified, automatically managed, and efficient fashion. In particular, `Spool` offers a way to participate in multiple yield generators while maintaining proper diversification, managing risk appetite, and benefiting from economies of scale when it comes to rebalancing and compounding. A user simply deposits into an existing Spool (or makes their own) and in turn enjoys automated and optimized decision-making curated by the `Spool DAO`. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Spool

Item	Description
Name	Spool Protocol
Website	https://www.spool.fi/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	March 30, 2022

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note that Spool assumes a trusted DAO for the configuration of various trusted entities, which are not part of this audit.

- https://github.com/SpoolFi/spool-core.git (7d612cd)

And this is the commit ID after all fixes for the issues found in the audit have been checked in.

- https://github.com/SpoolFi/spool-core.git (d6ec9e3)

1.2 About PeckShield

PeckShield Inc. [8] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

1.3 Methodology

To standardize the evaluation, we define the following terminology based on the OWASP Risk Rating Methodology [7]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a checklist of items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.

- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.

- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [6], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered

Table 1.3: The Full Audit Checklist

Category	Checklist Items
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logic	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the implementation of the `Spool` protocol. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logic, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	1	■
Medium	3	■ ■ ■
Low	6	■ ■ ■ ■ ■ ■
Undetermined	1	■
Total	11	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 high-severity vulnerability, 3 medium-severity vulnerabilities, 6 low-severity vulnerabilities, and 1 recommendation with undetermined severity.

Table 2.1: Key Spool Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Improved Bitwise Operations in Helper Library	Coding Practices	Confirmed
PVE-002	Low	Enhanced Validation of Function Arguments	Coding Practices	Fixed
PVE-003	Low	Proper Function Type of RiskProviderRegistry::_setRisk()	Coding Practices	Fixed
PVE-004	Low	Accommodation of Non-ERC20-Compliant Tokens	Business Logic	Fixed
PVE-005	Medium	Incorrect Strategies Hash Update in _removeStrategyCalldata()	Business Logic	Fixed
PVE-006	Low	Proper Logic Of RewardDrip::updatePeriodFinish()	Business Logic	Fixed
PVE-007	Medium	Proper Fast Withdrawal Logic in _saveUserShares()	Business Logic	Fixed
PVE-008	High	Proper Reallocation Logic in _processWithdraw()	Business Logic	Fixed
PVE-009	Medium	Proper Strategy Removal Logic in notifyStrategyRemoved()	Business Logic	Fixed
PVE-010	Low	Proper CompoundStrategy Initialization	Business Logic	Fixed
PVE-011	Undetermined	Forced Investment Risk in Curve3poolStrategy	Business Logic	Fixed

Besides the identified issues, we emphasize that for any user-facing applications and services, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Improved Bitwise Operations in Helper Library

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Bitwise`
- Category: Coding Practices [4]
- CWE subcategory: CWE-563 [2]

Description

For gas efficiency, the `Spool` protocol has a bitwise library that facilitates a variety of bitwise manipulations, including bit fields extraction, bit packing, compression, and hashing. Our analysis shows the current bitwise library can be improved.

To elaborate, we show below example bitwise-related helper routines, i.e., `get14BitUintByIndex()` and `set14BitUintByIndex()`. While the first `get14BitUintByIndex()` routine properly extracts the i^{th} $14bits$ in the given `bitwiseData` number, the second `set14BitUintByIndex()` routine has an implicit assumption that the current i^{th} $14bits$ in the given `bitwiseData` number is 0. Since this assumption may not always hold, there is a need to explicitly zero out the i^{th} $14bits$ before assigning the given `num14bit`. An example revision is shown as follows: `bitwiseData & ~((2^14-1)<< (14 * i))| (num14bit << (14 * i))`.

```
    // 14 bits is used for strategy proportions in a vault as FULL_PERCENT is 10_000
    function get14BitUintByIndex(uint256 bitwiseData, uint256 i) internal pure returns(
        uint256) {
        return (bitwiseData >> (14 * i)) & (16_383); // 16.383 is 2^14 - 1
    }

    function set14BitUintByIndex(uint256 bitwiseData, uint256 i, uint256 num14bit)
        internal pure returns(uint256) {
        return bitwiseData + (num14bit << (14 * i));
    }
```

Listing 3.1: `Bitwise::get14BitUintByIndex()/set14BitUintByIndex()`

The same improvement can also be applied to other routines, including `set16BitUintByIndex()` and `set24BitUintByIndex()`.

Recommendation Strengthen the current bitwise libraries to remove the above implicit assumption.

Status The issue has been confirmed.

3.2 Enhanced Validation of Function Arguments

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Coding Practices [4]
- CWE subcategory: CWE-563 [2]

Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The `Spool` protocol is no exception. Specifically, if we examine the `FeeHandler` contract, it has defined a number of protocol-wide risk parameters, e.g., `ecosystemFee` and `treasuryFee`. In the following, we show the corresponding routines that allow for their changes.

```
    function _setTreasuryFee(uint16 fee) private {
        require(fee <= MAX_TREASURY_FEE, "FeeHandler::_setTreasuryFee: Treasury fee too big");
        treasuryFee = fee;
        emit TreasuryFeeUpdated(fee);
    }

    /**
     * @notice Set ecosystem fee collector address
     *
     * @dev
     * Requirements:
     * - collector cannot be 0
     *
     * @param collector ecosystem fee collector address to set
     */
    function _setEcosystemCollector(address collector) private {
        require(collector != address(0), "FeeHandler::_setEcosystemCollector: Ecosystem Fee Collector address cannot be 0");
        ecosystemFeeCollector = collector;
        emit EcosystemCollectorUpdated(collector);
    }
```

Listing 3.2: A Number of `Setters` in `FeeHandler`

Our analysis shows the update logic on these fee parameters is properly guarded with various sanity checks. However, a number of other functions can benefit from similar validations. For example, the `SpoolDoHardWork::batchDoHardWork()` routine can be improved by further enforcing the given four arrays `stratIndexes`, `slippages`, `rewardSlippages`, and `allStrategies` share the same array length. The current logic only enforces the first two arrays have the same length.

```
    function batchDoHardWork(
        uint256[] memory stratIndexes,
        uint256[][] memory slippages,
        RewardSlippages[] memory rewardSlippages,
        address[] memory allStrategies
    )
        external
        onlyDoHardWorker
        verifyStrategies(allStrategies)
    {
        // update global index if this are first strategies in index
        if (_isBatchComplete()) {
            globalIndex++;
            doHardWorksLeft = uint8(allStrategies.length);
        }

        // check parameters
        require(reallocationIndex != globalIndex, "RLC");
        require(stratIndexes.length > 0 && stratIndexes.length == slippages.length, "BIPT");

        if (forceOneTxDoHardWork) {
            require(stratIndexes.length == allStrategies.length, "1TX");
        }
```

Listing 3.3: SpoolDoHardWork::batchDoHardWork()

The same improvement can also be applied to other routines, including `_batchDoHardWorkReallocation()` and `_depositRedistributedAmount()` in the same contract.

Recommendation Strengthen the validations on the above functions.

Status The issue has been fixed by this commit: `8b2e536`.

3.3 Proper Function Type of RiskProviderRegistry::_setRisk()

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `RiskProviderRegistry`
- Category: Coding Practices [4]
- CWE subcategory: CWE-563 [2]

Description

The `Spool` protocol is unique in allowing the approved risk providers to assess potential risks associated with current yielding strategies. In particular, the risk providers are able to set a risk score for the strategies in the range $[-10.0, 10.0]$. While reviewing the risk score assignment logic, we notice one helper routine can be improved.

To elaborate, we show below the code snippet from the `setRisk()` function. As the name indicates, this function is designed to set the risk score of a strategy. After necessary validations, this function invokes the internal helper `_setRisk()` to actually set the risk score. It comes to our attention this helper `_setRisk()` is defined as `public`, which by design is supposed to be `internal`!

```
    function setRisk(address strategy, uint8 riskScore) public {
        require(
            isProvider(msg.sender),
            "RiskProviderRegistry::setRisk: Insufficient Privileges"
        );

        _setRisk(strategy, riskScore);
    }
```

Listing 3.4: RiskProviderRegistry :: setRisk()

```
    function _setRisk(address strategy, uint8 riskScore) public {
        require(riskScore <= MAX_RISK_SCORE, "RiskProviderRegistry::_setRisk: Risk score too big");

        _risk[msg.sender][strategy] = riskScore;

        emit RiskAssessed(msg.sender, strategy, riskScore);
    }
```

Listing 3.5: RiskProviderRegistry :: _setRisk()

Recommendation Redefine the function type of `_setRisk()` to be `internal`.

Status The issue has been fixed by this commit: `8b2e536`.

3.4 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: High
- Target: `Multiple Contracts`
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

Description

Though there is a standardized ERC-20 specification, many token contracts may not strictly follow the specification or have additional functionalities beyond the specification. In the following, we examine the `transfer()` routine and related idiosyncrasies from current widely-used token contracts.

In particular, we use the popular stablecoin, i.e., `USDT`, as our example. We show the related code snippet below. On its entry of `approve()`, there is a requirement, i.e., `require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)))`. This specific requirement essentially indicates the need of reducing the allowance to 0 first (by calling `approve(_spender, 0)`) if it is not, and then calling a second one to set the proper allowance. This requirement is in place to mitigate the known `approve()/transferFrom()` race condition (https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729).

```
/**
* @dev Approve the passed address to spend the specified amount of tokens on behalf
    of msg.sender.
* @param _spender The address which will spend the funds.
* @param _value The amount of tokens to be spent.
*/
function approve(address _spender, uint _value) public onlyPayloadSize(2 * 32) {

    // To change the approve amount you first have to reduce the addresses`
    //  allowance to zero by calling `approve(_spender, 0)` if it is not
    //  already 0 to mitigate the race condition described here:
    //  https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
    require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)));

    allowed[msg.sender][_spender] = _value;
    Approval(msg.sender, _spender, _value);
}
```

Listing 3.6: USDT Token **Contract**

Because of that, a normal call to `approve()` is suggested to use the safe version, i.e., `safeApprove()` , In essence, it is a wrapper around ERC20 operations that may either throw on failure or return false without reverts. Moreover, the safe version also supports tokens that return no value (and instead revert or throw on failure). Note that non-reverting calls are assumed to be successful. Similarly, there is a safe version of `transfer()` as well, i.e., `safeTransfe()`.

```
    /**
     * @dev Deprecated. This function has issues similar to the ones found in
     * {IERC20-approve}, and its usage is discouraged.
     *
     * Whenever possible, use {safeIncreaseAllowance} and
     * {safeDecreaseAllowance} instead.
     */
    function safeApprove(
        IERC20 token,
        address spender,
        uint256 value
    ) internal {
        // safeApprove should only be called when setting an initial allowance,
        // or when resetting it to zero. To increase and decrease it, use
        // 'safeIncreaseAllowance' and 'safeDecreaseAllowance'
        require(
            (value == 0)  (token.allowance(address(this), spender) == 0),
            "SafeERC20: approve from non-zero to non-zero allowance"
        );
        _callOptionalReturn(token, abi.encodeWithSelector(token.approve.selector,
            spender, value));
    }
```

Listing 3.7: `SafeERC20::safeApprove()`

In current implementation, if we examine the `_approveAndSwap()` routine in the `SwapHelper` contract, this routine is designed to perform token swaps. To accommodate the specific idiosyncrasy, there is a need to use `safeApprove()` (line 69).

```
55    function _approveAndSwap(
56        IERC20 from,
57        IERC20 to,
58        uint256 amount,
59        SwapData calldata swapData
60    ) internal virtual returns (uint256) {
61
62        // if there is nothing to swap, return
63        if(amount == 0)
64            return 0;
65
66        // if amount is not uint256 max approve unswap router to spend tokens
67        // otherwise rewards were already sent to the router
68        if(amount < type(uint256).max) {
69            from.safeApprove(address(uniswapRouter), amount);
70        } else {
71            amount = 0;
72        }
73        ...
74    }
```

Listing 3.8: `SwapHelper::_approveAndSwap()`

While the current implementation properly uses the `safeApprove()`, there is a need to approve twice: the first time resets the allowance to zero and the second time approves the intended amount. And this affects a number of routines in almost all supported strategies, including `YearnStrategy`, `IdleStrategy`, `MasterChefStrategyBase`, `Curve3poolStrategy`, and `HarvestStrategy`.

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related `approve()/transfer()/transferFrom()`.

Status The issue has been fixed as the allowance is reset to zero after the allowance is used.

3.5 Incorrect Strategies Hash Update in _removeStrategyCalldata()

- ID: PVE-005
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `Controller`
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

Description

The `Spool` protocol has a `Controller` contract that is the central point of the protocol for assessing the validity of various data in the system (i.e. supported strategy, vault etc.). While reviewing the current logic to remove an existing strategy, we notice its current logic needs to be corrected.

In particular, an issue stems from the need to properly update the `strategiesHash` since an existing strategy is being removed. However, the hash-update logic (line 599) fails to use the updated strategies to compute the hash. Instead, it still uses the stale list to compute and update the `strategiesHash`. Unfortunately, an incorrect `strategiesHash` may fail a variety of reallocation routines.

```
585     function _removeStrategyCalldata(address[] calldata allStrategies, address strategy)
            private {
586         uint256 lastEntry = allStrategies.length - 1;
587         address[] memory newStrategies = allStrategies[0:lastEntry];
588
589         for (uint256 i = 0; i < lastEntry; i++) {
590             if (allStrategies[i] == strategy) {
591                 strategies[i] = allStrategies[lastEntry];
592                 newStrategies[i] = allStrategies[lastEntry];
593                 break;
594             }
595         }
596
597         strategies.pop();
598
```

```
        _updateStrategiesHash(allStrategies);
    }
```

Listing 3.9: `Controller::_removeStrategyCalldata()`

Recommendation Correct the above logic in the update of `strategiesHash` when an existing strategy is being removed.

Status The issue has been fixed by this commit: `8b2e536`.

3.6 Proper Logic Of RewardDrip::updatePeriodFinish()

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `RewardDrip`
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

Description

The `Spool` protocol has flexible supports of multiple reward tokens and the support is mainly implemented in the `RewardDrip` contract. If we focus on one of its routine, i.e., `rewardPerToken()`, this routine is responsible for calculating the reward rate for each staked token and it is part of the `updateReward` modifier that would be invoked up-front for almost every public function in `RewardDrip` to update and use the latest reward rate.

Our analysis shows that this `rewardPerToken()` requires the computation of `timeDelta = lastTimeRewardApplicable(token)- config.lastUpdateTime` (line 55) to compute the latest reward rate where `lastTimeRewardApplicable(token)` is defined as `uint32(Math.min(block.timestamp, rewardConfiguration[token].periodFinish))`. It comes to our attention that the `periodFinish` state may be updated by a privileged account `owner`. We notice there is a lack of necessary validation of the given `timestamp` argument to configure the `periodFinish`. If the given `timestamp` argument is smaller than `rewardConfiguration[token].lastUpdateTime`, every invocation of `rewardPerToken()` will be unfortunately reverted, which may unfortunately lock up all staked funds!

```
    // End rewards emission earlier
    function updatePeriodFinish(IERC20 token, uint32 timestamp)
        external
        onlyOwner
        updateReward(token, address(0))
    {
        rewardConfiguration[token].periodFinish = timestamp;
    }
```

Listing 3.10: `RewardDrip::updatePeriodFinish()`

```
    function rewardPerToken(IERC20 token) public view returns (uint224) {
        RewardConfiguration storage config = rewardConfiguration[token];

        if (totalInstantDeposit == 0)
            return config.rewardPerTokenStored;

        uint256 timeDelta = lastTimeRewardApplicable(token) - config.lastUpdateTime;

        if (timeDelta == 0)
            return config.rewardPerTokenStored;

        return
            SafeCast.toUint224(
                config.rewardPerTokenStored +
                    ((timeDelta
                        * config.rewardRate)
                        / totalInstantDeposit)
            );
    }
```

Listing 3.11: `RewardDrip::rewardPerToken()`

Recommendation Add the necessary validation when there is a need to reconfigure the `periodFinish`.

Status The issue has been fixed by this commit: `d11bbaf`.

3.7 Proper Fast Withdrawal Logic in _saveUserShares()

- ID: PVE-007
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `FastWithdraw`
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

Description

The `Spool` protocol has a `FastWithdraw` contract that allows to withdraw user shares without the need to wait for the `DoHardWork` functions to be executed. The logic is to transfer the related withdrawal share to the `FastWithdraw` contract and the user can claim them later. Note that the performance fee is still paid to the vault where the shares where initially taken from.

While analyzing the logic in the `FastWithdraw` contract, we notice there is a need to save user strategy shares that are being transfered from the vault – as shown in the following routine. However, the logic blindly overwrites the internal state `vaultWithdraw.proportionateDeposit`, which may cause an issue if the user makes multiple consecutive fast withdraw requests!

```
177     function _saveUserShares(
178         address[] calldata vaultStrategies,
179         uint128[] calldata sharesWithdrawn,
180         uint256 proportionateDeposit,
181         IVault vault,
182         address user
183     ) private {
184         VaultWithdraw storage vaultWithdraw = userVaultWithdraw[user][vault];

186         vaultWithdraw.proportionateDeposit = proportionateDeposit;

188         for (uint256 i = 0; i < vaultStrategies.length; i++) {
189             vaultWithdraw.userStrategyShares[vaultStrategies[i]] = sharesWithdrawn[i];
190         }
191     }
```

Listing 3.12: `FastWithdraw::_saveUserShares()`

Recommendation Properly revise the above routine to increase the state `vaultWithdraw.proportionateDeposit` by the given amount `proportionateDeposit` (line 186).

Status This issue has been fixed in the following commit: `ff3f376`.

3.8 Proper Reallocation Logic in _processWithdraw()

- ID: PVE-008
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: `SpoolDoHardWork`
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

Description

For gas efficiency, the `Spool` protocol has a guarded `DoHardWork` process to interact with external protocols. In particular, this process aggregates many actions together to act in an optimized manner with special considerations for flexible support of external protocols and reduced gas cost. While analyzing the reallocation-related logic, we notice the current implementation needs to be improved.

To elaborate, we show below the related `_processWithdraw()` function. As the name indicates, this function is used to withdraw assets from the current set of strategies. After necessary optimization on the concurrent deposits and withdraws, this routine further redistributes the withdrawn assets to other strategies for immediate deposits. However, our analysis shows that it incorrectly provides the arguments for the redistribution. Specifically, it takes four arguments and the last one is `withdrawData.reallocationProportions[withdrawData.stratIndexes[stratIndex]]`, which should be `withdrawData.reallocationProportions[stratIndex]`!

```
    function _processWithdraw(
        ReallocationWithdrawData memory withdrawData,
        address[] memory allStrategies,
        PriceData[] memory spotPrices
    ) private {
        ReallocationShares memory reallocation = _optimizeReallocation(withdrawData,
            spotPrices);

        // go over withdrawals
        for (uint256 i = 0; i < withdrawData.stratIndexes.length; i++) {
            uint256 stratIndex = withdrawData.stratIndexes[i];
            address stratAddress = allStrategies[stratIndex];
            Strategy storage strategy = strategies[stratAddress];
            require(!strategy.isInDepositPhase, "SWP");

            uint128 withdrawnReallocationRecieved;
            {
                uint128 sharesToWithdraw = reallocation.totalSharesWithdrawn[stratIndex]
                     - reallocation.optimizedShares[stratIndex];

                ProcessReallocationData memory processReallocationData =
                    ProcessReallocationData(
                    sharesToWithdraw,
                    reallocation.optimizedShares[stratIndex],
                    reallocation.optimizedWithdraws[stratIndex]
                );

                // withdraw reallocation / returns non-optimized withdrawn amount
                withdrawnReallocationRecieved = _doHardWorkReallocation(stratAddress,
                    withdrawData.slippages[stratIndex], processReallocationData);
            }

            // redistribute withdrawn to other strategies
            _depositRedistributedAmount(
                // withdrawData.stratIndexes[stratIndex],
                reallocation.totalSharesWithdrawn[stratIndex],
                withdrawnReallocationRecieved,
                reallocation.optimizedWithdraws[stratIndex],
                allStrategies,
                withdrawData.reallocationProportions[withdrawData.stratIndexes[
                    stratIndex]]
            );

            _updatePending(stratAddress);

            strategy.isInDepositPhase = true;
        }
    }
```

Listing 3.13: `SpoolDoHardWork::_processWithdraw()`

Recommendation Correct the above routine for proper reallocation of withdrawn assets to current strategies.

Status This issue has been fixed in the following commit: `ff3f376`.

3.9 Proper Strategy Removal Logic in notifyStrategyRemoved()

- ID: PVE-009
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `Vault`
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

Description

The `Spool` is a middleware that connects users to existing and new `yield generators` and `yield optimizers`. Accordingly, the protocol has the flexible support of adding and removing external strategies. While reviewing the current strategy-removal logic, we notice the current implementation is flawed.

To elaborate, we show below the related function, i.e., `notifyStrategyRemoved()`. When an existing strategy needs to be removed, there is a need to adjust the deposit proportions accordingly. However, the proportions-adjusting logic has a mis-calculation at line 610. The new proportions need to be updated with the iteration for all remaining strategies (`j`), instead of the one being removed (`i`). In other words, it does not update the proportions as intended!

```
568     function notifyStrategyRemoved(
569         address strat,
570         address[] memory vaultStrategies,
571         uint256 i
572     )
573         external
574         verifyStrategies(vaultStrategies)
575         hasStrategies(vaultStrategies)
576         redeemVaultStrategiesModifier(vaultStrategies)
577     {
578         require(vaultStrategies[i] == strat, "NSTR");

580         uint256 lastElement = vaultStrategies.length - 1;
581         address[] memory newStrategies = new address[](lastElement);

583         if (lastElement > 0) {
584             for (uint256 j; j < lastElement; j++) {
585                 newStrategies[j] = vaultStrategies[j];
586             }

588             if (i < lastElement) {
```

```
                newStrategies[i] = vaultStrategies[lastElement];
            }

            uint256 _proportions = proportions;
            uint256 proportionsLeft = FULL_PERCENT - _proportions.get14BitUintByIndex(i)
                ;
            if (lastElement > 1 && proportionsLeft > 0) {
                if (i == lastElement) {
                    _proportions = _proportions.reset14BitUintByIndex(i);
                } else {
                    uint256 lastProportion = _proportions.get14BitUintByIndex(
                        lastElement);
                    _proportions = _proportions.reset14BitUintByIndex(i);
                    _proportions = _proportions.set14BitUintByIndex(i, lastProportion);
                }

                uint256 newProportions = _proportions;

                uint256 lastNewElement = lastElement - 1;
                uint256 newProportionsLeft = FULL_PERCENT;
                for (uint256 j; j < lastNewElement; j++) {
                    uint256 propJ = _proportions.get14BitUintByIndex(j);
                    propJ = (propJ * FULL_PERCENT) / proportionsLeft;
                    newProportions = newProportions.set14BitUintByIndex(i, propJ);
                    newProportionsLeft -= propJ;
                }

                newProportions = newProportions.set14BitUintByIndex(lastNewElement,
                    newProportionsLeft);

                proportions = newProportions;
            } else {
                proportions = FULL_PERCENT;
            }
        } else {
            proportions = 0;
        }

        _updateStrategiesHash(newStrategies);
    }
```

Listing 3.14: `CollSurplusPool::setAddresses()`

Recommendation Properly adjust the proportions when there is a need to remove an existing strategy.

Status This issue has been fixed in the following commit: `dfdfd3f`.

3.10 Proper CompoundStrategy Initialization

- ID: PVE-010
- Severity: Low
- Likelihood: Medium
- Impact: Low
- Target: `CompoundStrategy`
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

Description

As mentioned earlier, the `Spool` protocol supports a number of external protocol for integration. While reviewing a specific one `CompoundStrategy`, we notice the strategy is not properly constructed.

In the following, we use the `constructor()` of the `CompoundStrategy` contract. The `constructor()` routine adds a requirement in validating `address(_underlying)!= _cToken.underlying()` (line 35). The purpose is to ensure that the underlying asset for yields matches the underlying token in the integrated `cToken`. However, it comes to our attention the validation should be performed as follows: `require(address(_underlying)== _cToken.underlying()`. In other words, it currently performs the contrary requirement!

```
24      constructor(
25          IERC20 _comp,
26          ICErc20 _cToken,
27          IComptroller _comptroller,
28          IERC20 _underlying
29      )
30          BaseStrategy(_underlying, 1, 0, 0, 0, false)
31          ClaimFullSingleRewardStrategy(_comp)
32      {
33          require(address(_cToken) != address(0), "CompoundStrategy::constructor: Token
                address cannot be 0");
34          require(address(_comptroller) != address(0), "CompoundStrategy::constructor:
                Comptroller address cannot be 0");
35          require(address(_underlying) != _cToken.underlying(), "CompoundStrategy::
                constructor: Underlying and cToken underlying do not match");
36          cToken = _cToken;
37          comptroller = _comptroller;
38      }
```

Listing 3.15: `CompoundStrategy::setAddresses()`

Recommendation Properly validate the consistency of the underlying assets for the `CompoundStrategy` support.

Status This issue has been fixed in the following commit: `dfdfd3f`.

3.11 Forced Investment Risk in Curve3poolStrategy

- ID: PVE-011
- Severity: Undetermined
- Likelihood: N/A
- Impact: N/A
- Target: `Curve3poolStrategy`, `ConvexSharedStrategy`
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

Description

`Spool` is a decentralized asset management protocol which shifts crypto investment from normal users to professionals. The investment subsystem is inspired from the `yearn.finance` framework and thus shares similar architecture with `vaults`, `controller`, and `strategies`.

While examining a specific `Curve3poolStrategy` implementation, we notice the `lpToken` share computation may suffer from a potential force investment risk that has been exploited in earlier hacks, e.g., `yDAI` [9] and `BT.Finance` [1]. To elaborate, we show blow the related `Curve3poolStrategy::_lpToCoin()` routine.

Specifically, this routine is used to compute the value for the given amount of `lpToken`. Unfortunately, the computation has an implicit assumption that the current `Curve` pool is balanced. In other words, when the pool is being manipulated to be highly imbalanced, the computed `lpToken` share computation may be manipulated to force the investment to a faulty strategy!

```
    function _lpToCoin(uint256 lp) internal view returns (uint128) {
        if (lp == 0)
            return 0;

        uint256 lpToCoin = pool.calc_withdraw_one_coin(ONE_LP_UNIT, nCoin);

        uint256 result = (lp * lpToCoin) / ONE_LP_UNIT;

        return SafeCast.toUint128(result);
    }
```

Listing 3.16: `Curve3poolStrategy::_lpToCoin()`

Fortunately, it comes to our attention that the reallocation task is guarded or can be invoked by whitelisted entities that are assumed to be trustworthy. However, we need to emphasize the risk here: If the configured strategy blindly invests the deposited funds into an imbalanced `Curve` pool, the strategy will not result in a profitable investment. In fact, earlier incidents (yDAI and BT hacks [9, 1]) have prompted the need of a guarded call to the reallocation operations. From another perspective, we do need to stay alert on the potential frontrunning or `MEV` that may still be able to exploit the reallocation for profit.

Recommendation Ensure the reallocation task can only be called via a trusted entity. And take extra care in ensuring the vault assets will not be blindly deposited into a faulty strategy (that is currently not making any profit).

Status This issue has been fixed in the following commit: `a9654cf`.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Spool` protocol, which serves as the `DeFi middleware` that allows users to participate in a subset of yield generating protocols in a risk diversified, automatically managed, and efficient fashion. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Moreover, we need to emphasize that `Solidity-based` smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

References

[1] BT Finance. BT.Finance Exploit Analysis Report. https://btfinance.medium.com/bt-finance-exploit-analysis-report-a0843cb03b28.

[2] MITRE. CWE-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.html.

[3] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

[4] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

[5] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

[6] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

[7] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[8] PeckShield. PeckShield Inc. https://www.peckshield.com.

[9] PeckShield. The yDAI Incident Analysis: Forced Investment. https://peckshield.medium.com/the-ydai-incident-analysis-forced-investment-2b8ac6058eb5.