SMART CONTRACT AUDIT REPORT

for

Spool Strategies (Arbitrum)

Prepared By: Xiaomi Huang

PeckShield

February 26, 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	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	February 26, 2022	Xuxian Jiang	Final Release
1.0-rc	January 12, 2022	Xuxian Jiang	Release Candidate #1

Contact

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

Name	Xiaomi Huang
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 Arbitrum deployment 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. This audit covers the new deployment on Arbitrum as well as related new strategies. 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	February 26, 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 (86c0127)

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 (ce1b503)

1.2 About PeckShield

PeckShield Inc. [7] 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 [6]:

- 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.

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

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) [5], 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 1.4: 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 `Arbitrum` deployment 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	0	
Medium	1	■
Low	2	■ ■
Informational	1	■
Total	4	

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 medium-severity vulnerability, 2 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key Spool Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Improper Vault Proportion Update Upon Strategy Removal	Business Logic	Resolved
PVE-002	Low	Consistency of Max128Bit-Based Storage Reads And Writes	Coding Practices	Resolved
PVE-003	Informational	Improved Logic in AbracadabraMetapoolStrategy/Curve2poolStrategy	Coding Practices	Resolved
PVE-004	Low	Incorrect MANTISSA Initialization in BalancerStrategy	Business Logic	Resolved

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 Improper Vault Proportion Update Upon Strategy Removal

- ID: PVE-001
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `Vault`
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

The `Spool` protocol has a user-facing `Vault` contract to support user deposits and withdrawals. The user deposits will be redirected to back-end `strategies` for yields. The `strategies` may be dynamically added or removed from the supported `Vault`. While analyzing the removal logic of a current `strategy`, we notice the current implementation needs to be corrected.

To elaborate, we show below the related code snippet of the related `notifyStrategyRemoved()` routine, which will be called to notify a vault a strategy was removed from `Spool`. Specifically, when an active strategy is removed, there is a need to dynamically reallocate the funds from the removed strategy to other strategies. The dynamic reallocation is specified with the proportions for each strategy. Our analysis shows the resulting `newProportions` is not correct since it needs to start as 0, instead of being initialized to the stale `_proportions` (line 647)!

```
607     function notifyStrategyRemoved(
608         address[] memory vaultStrategies,
609         uint256 i
610     )
611         external
612         reallocationFinished
613         verifyStrategies(vaultStrategies)
614         hasStrategies(vaultStrategies)
615         redeemVaultStrategiesModifier(vaultStrategies)
616     {
617         require(
618             i < vaultStrategies.length &&
```

```
            !controller.validStrategy(vaultStrategies[i]),
            "BSTR"
        );

        uint256 lastElement = vaultStrategies.length - 1;

        address[] memory newStrategies = new address[](lastElement);

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

            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(j, propJ);
                    newProportionsLeft -= propJ;
                }

                newProportions = newProportions.set14BitUintByIndex(lastNewElement,
                    newProportionsLeft);

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

```

```
        _updateStrategiesHash(newStrategies);
        emit StrategyRemoved(i, vaultStrategies[i]);
    }
```

Listing 3.1: `Vault::notifyStrategyRemoved()`

Recommendation Correct the above logic to calculate the new reallocation `newProportions` when an active strategy is being removed.

Status This issue has been fixed in the following commit: `f4052ab`.

3.2 Consistency of Max128Bit-Based Storage Reads And Writes

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Max128Bit`
- Category: Coding Practices [3]
- CWE subcategory: CWE-1041 [1]

Description

The `Spool` protocol has a specific `Max128Bit` library that is proposed to handle setting zero value in a storage word as `uint128` max value. Specifically, the purpose is to avoid resetting a storage word to the zero value. The reasoning here is that the gas cost of re-initializing the value is the same as setting the word originally. With that, if a word is to be set to zero, the protocol sets it to `uint128` max.

This library should only be used to read or write directly from storage. To facilitate the operations, it provides two main routines, i.e., `get()` and `set()`. As the names indicate, the first routine is used when there is a need to load a word from storage and the second routine is used when there is a need to write a word to storage.

```
    function get(uint128 a) internal pure returns(uint128) {
        return (a == ZERO) ? 0 : a;
    }

    function set(uint128 a) internal pure returns(uint128){
        return (a == 0) ? ZERO : a;
    }
```

Listing 3.2: `Max128Bit::get()/set()`

However, our analysis shows that this library is not used consistently. For example, in the following `BaseStrategy` contract, when there is a need to load from the storage `strategy.pendingUser.deposit` (lines 273 – 274), the `get()` routine is used. However, when the same storage is written, the `set()` routine is not used (line 275).

```
256     function emergencyWithdraw(address recipient, uint256[] calldata data) external
            virtual override {
257         uint256 balanceBefore = underlying.balanceOf(address(this));
258         _emergencyWithdraw(recipient, data);
259         uint256 balanceAfter = underlying.balanceOf(address(this));
260 
261         uint256 withdrawnAmount = 0;
262         if (balanceAfter > balanceBefore) {
263             withdrawnAmount = balanceAfter - balanceBefore;
264         }
265 
266         Strategy storage strategy = strategies[self];
267         if (strategy.emergencyPending > 0) {
268             withdrawnAmount += strategy.emergencyPending;
269             strategy.emergencyPending = 0;
270         }
271 
272         // also withdraw all unprocessed deposit for a strategy
273         if (strategy.pendingUser.deposit.get() > 0) {
274             withdrawnAmount += strategy.pendingUser.deposit.get();
275             strategy.pendingUser.deposit = 0;
276         }
277 
278         if (strategy.pendingUserNext.deposit.get() > 0) {
279             withdrawnAmount += strategy.pendingUserNext.deposit.get();
280             strategy.pendingUserNext.deposit = 0;
281         }
282 
283         // if strategy was already processed in the current index that hasn't finished
                yet,
284         // transfer the withdrawn amount
285         // reset total underlying to 0
286         if (strategy.index == globalIndex && doHardWorksLeft > 0) {
287             uint256 withdrawnReceived = strategy.batches[strategy.index].
                    withdrawnReceived;
288             withdrawnAmount += withdrawnReceived;
289             strategy.batches[strategy.index].withdrawnReceived = 0;
290 
291             strategy.totalUnderlying[strategy.index].amount = 0;
292         }
293 
294         if (withdrawnAmount > 0) {
295             // check if the balance is high enough to withdraw the total withdrawnAmount
296             if (balanceAfter < withdrawnAmount) {
297                 // if not withdraw the current balance
298                 withdrawnAmount = balanceAfter;
299             }
300 
301             underlying.safeTransfer(recipient, withdrawnAmount);
```

```
        }
    }
```

Listing 3.3: `BaseStrategy::emergencyWithdraw()`

Recommendation Be consistent when the `Max128Bit` library is used.

Status This issue has been fixed in the following commit: `35e8451`.

3.3 Improved Logic in AbracadabraMetapoolStrategy/Curve2poolStrategy

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Multiple Contracts`
- Category: Coding Practices [3]
- CWE subcategory: CWE-1041 [1]

Description

`Spool` is a decentralized asset management protocol that participates in multiple yield generators. While examining a specific strategy, i.e., `AbracadabraMetapoolStrategy`, we notice a possible improvement in removing an extra validation check.

To elaborate, we show blow the related `_claimStrategyReward()` routine. Specifically, this routine is used to claim strategy rewards. It comes to our attention that when new rewards are claimed (line 125), there is no need to validate the following requirement: `rewardTokenAmount > 0` (line 127). The reason is that this requirement is guaranteed to be true! Note another strategy `Curve2poolStrategy` shares the same issue.

```
118     function _claimStrategyReward() internal override returns(uint128) {
119         (
120             uint256 rewardTokenAmount,
121             bool didClaimNewRewards
122         ) = farmHelper.claimReward(true);
123
124
125         if (didClaimNewRewards) {
126             Strategy storage strategy = strategies[self];
127             if (rewardTokenAmount > 0) {
128                 strategy.pendingRewards[address(rewardToken)] += rewardTokenAmount;
129             }
130         }
131
132         return SafeCast.toUint128(strategies[self].pendingRewards[address(rewardToken)])
            ;
```

```
    }
```

Listing 3.4: `AbracadabraMetapoolStrategy::_claimStrategyReward()`

Recommendation Remove the extra validation on the return `rewardTokenAmount` in the above two strategies.

Status This issue has been fixed in the following commit: `9c2a005`.

3.4 Incorrect MANTISSA Initialization in BalancerStrategy

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `BalancerStrategy`
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

The `Spool` protocol has developed a `BalancerStrategy` to support the interaction with the `Balancer` protocol. This strategy has an internal variable `MANTISSA` that is designed to convert the LP to the underlying token. While reviewing the related initialized logic, we notice the current implementation can be improved.

In particular, we show below this related constructor with the following initialization: `MANTISSA = 10 ** uint(pDecimals + (pDecimals - uDecimals))` (line 67). The initialization is suggested to revise as `MANTISSA = 10 ** uint(18 + (pDecimals - uDecimals))` (line 67). By doing so, it captures the conversion between LP and the underlying token. Fortunately, the current `pDecimals` is always 18, which ensures the correctness of the execution of current logic, though semantically confusing or misleading.

```
    constructor(
        IStablePool _pool,
        IERC20Metadata _underlying,
        uint256 _nCoin,
        address _self
    )
        NoRewardStrategy(_underlying, 1, 1, 1, false, _self)
    {
        require(address(_pool) != address(0), "BalancerStrategy::constructor: Pool
            address cannot be 0");
        vault = IBalancerVault(_pool.getVault());
        poolId = _pool.getPoolId();
        (IAsset[] memory _assets,,) = vault.getPoolTokens(poolId);

        require(address(_underlying) == address( _assets[_nCoin] ), "BalancerStrategy::
            constructor: Underlying address and nCoin invalid");
```

```
    
        pool = _pool;
        nCoin = _nCoin;

        // we derive the underlying amount from BPT token amount; the mantissa
        // is used to convert (see _lpToCoin()).
        // BPT and underlying token decimals may differ, so we handle that here.
        int uDecimals = int(int8(_underlying.decimals()));
        int pDecimals = int(int8(_pool.decimals()));
        MANTISSA =  10 ** uint(pDecimals + (pDecimals - uDecimals));
    }
```

Listing 3.5: `BalancerStrategy::constructor()`

Recommendation Improve the above `MANTISSA` initialization in a meaningful way.

Status This issue has been fixed in the following commit: `db20ebb`.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Arbitrum` deployment of the `Spool` protocol, which serves as the `DeFi middleware` and allows users to participate in a subset of yield generating protocols in a risk diversified, automatically managed, and efficient fashion. This audit covers the new deployment on `Arbitrum` as well as related new strategies. 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] MITRE. CWE-1041: Use of Redundant Code. https://cwe.mitre.org/data/definitions/1041.html.

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

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

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

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

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

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