SMART CONTRACT AUDIT REPORT

for

Spot Protocol

Prepared By: Xiaomi Huang

PeckShield

March 14, 2024

Document Properties

Client	Fragments, Inc.
Title	Smart Contract Audit Report
Target	Spot
Version	1.0
Author	Xuxian Jiang
Auditors	Jason Shen, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	March 14, 2024	Xuxian Jiang	Final Release
1.0-rc1	March 10, 2024	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 SPOT 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 SPOT

SPOT is a decentralized, inflation resistant store of value designed to be resilient in all market conditions. It has no reliance on centralized custodians, liquidations, or lenders of last resort. It has no feedback loops, no dependence on continual growth, and is free from bank runs. The system bends safely rather than breaking catastrophically in extreme market scenarios, and can forever resume its function without bailouts. SPOT can be held directly or rotated in as an alternative collateral asset to USDC within existing systems. It uses AMPL as the underlying unit of account, Buttonwood Tranche for collateral preparation, and onchain governance through the FORTH DAO. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of The SPOT Protocol

Item	Description
Client	Fragments, Inc.
Website	https://ampleforth.org
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	March 14, 2024

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit.

- https://github.com/ampleforth/spot.git (eda092c)

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

- https://github.com/ampleforth/spot.git (01b01a4)

1.2 About PeckShield

PeckShield Inc. [9] 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 OWASP Risk Rating Methodology [8]:

- 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 List of Check Items

Category	Check Item
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 list of check 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) [7], 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.

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 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.

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 Logics	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.

2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the design and implementation of the `Spot` 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 logics, 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	0	
Low	4	■ ■ ■ ■
Informational	1	■
Total	5	

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 that 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 4 low-severity vulnerabilities and and 1 informational recommendation.

Table 2.1: Key Spot Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Improved Constructor Logic in BondIssuer	Coding Practices	
PVE-002	Informational	Strengthened Validation on Function Arguments in RouterV2	Coding Practices	
PVE-003	Low	Possible Inconsistency in Vault Fee Calculation	Code Practices	
PVE-004	Low	Revisited previewDeposit() Logic in BondHelpers	Business Logic	
PVE-005	Low	Trust on Admin Keys	Security Features	

Besides recommending specific countermeasures to mitigate these issues, we also emphasize that 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 need to kick in at the very moment when the contracts are being deployed in mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Improved Constructor Logic in BondIssuer

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: BondIssuer
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

To facilitate possible future upgrade, a number of key contracts (e.g., PerpetualTranche and BondIssuer) are instantiated as a proxy with actual logic contracts in the backend. While examining the related contract construction and initialization logic, we notice current construction can be improved.

In the following, we shows its initialization routine. We notice its constructor does not have any payload. With that, it can be improved by adding the following statement, i.e., _disableInitializers ();. Note this statement is called in the logic contract where the initializer is locked. Therefore any user will not able to call the initialize() function in the state of the logic contract and perform any malicious activity. Note that the proxy contract state will still be able to call this function since the constructor does not effect the state of the proxy contract.

```
    function init(
        uint256 maxMaturityDuration_,
        uint256[] memory trancheRatios_,
        uint256 minIssueTimeIntervalSec_,
        uint256 issueWindowOffsetSec_
    ) public initializer {
        __Ownable_init();
        updateMaxMaturityDuration(maxMaturityDuration_);
        updateTrancheRatios(trancheRatios_);
        updateIssuanceTimingConfig(minIssueTimeIntervalSec_, issueWindowOffsetSec_);
    }
```

Listing 3.1: BondIssuer::initialize()

Recommendation Improve the above-mentioned constructor routines in all existing upgradeable contracts, including `BondIssuer`, `FeePolicy`, `PerpetualTranche`, and `RolloverVault`.

Status This issue has been fixed by this commit: `b78a882`.

3.2 Strengthened Validation on Function Arguments in RouterV2

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `RouterV2`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

To facilitate the user interaction, the `Spot` protocol provides a `RouterV2` contract to dry-run and batch multiple operations. While examining one specific function `trancheAndDeposit()`, we notice it can benefit from improved validation.

Specifically, this function is proposed to tranche the collateral by using the current deposit bond and then depositing individual tranches to mint perp tokens. It currently takes three arguments and the first two are `perp` and `bond`. And there is an implicit assumption that the given `bond` needs to be the active `deposit bond` of `perp`. As a result, we can either validate this assumption (by enforcing `require( bond == perp.getDepositBond())`) or simply obtain the `bond` directly from `perp` (without passing it as an argument).

```
    function trancheAndDeposit(IPerpetualTranche perp, IBondController bond, uint256
        collateralAmount) external {
        // If deposit bond does not exist, we first issue it.
        if (address(bond).code.length <= 0) {
            perp.updateState();
        }

        BondTranches memory bt = bond.getTranches();
        IERC20Upgradeable collateralToken = IERC20Upgradeable(bond.collateralToken());

        // transfers collateral & fees to router
        collateralToken.safeTransferFrom(msg.sender, address(this), collateralAmount);

        // approves collateral to be tranched
        _checkAndApproveMax(collateralToken, address(bond), collateralAmount);

        // tranches collateral
        bond.deposit(collateralAmount);
```

```
        ...
    }
```

Listing 3.2: `RouterV1::trancheAndDeposit()`

Recommendation Validate the given `bond` argument in the above routine to be the `deposit bond` of the given `perp`.

Status This issue has been fixed by this commit: `9cd15ab`.

3.3 Possible Inconsistency in Vault Fee Calculation

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `FeePolicy`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

The SPOT protocol has a unique `FeePolicy` mechanism that aims to balance the demand for holding perp tokens with the demand for holding vault tokens such that the total collateral in the vault supports rolling over all mature collateral backing perps. While reviewing the dynamic fee calculation, we notice certain inconsistency can be resolved.

To elaborate, we show two related routines, i.e., `computeVaultMintFeePerc()` and `computeVaultBurnFeePerc()`. As their names indicate, the first routine computes the vault mint fee and the second one computes the vault burn fee. The vault mint fee is computed with the step function (i.e., `_stepFnFeePerc()`) while the vault burn fee is relatively static without the step function computation. For consistency, we would suggest to make use of the step function for the vault burn fee calculation.

```
    function computeVaultMintFeePerc(uint256 drPre, uint256 drPost) external view
        override returns (uint256) {
        return _stepFnFeePerc(drPre, drPost, 0, vaultMintFeePerc);
    }

    /// @inheritdoc IFeePolicy
    function computeVaultBurnFeePerc(uint256 /*drPre*/, uint256 /*drPost*/) external
        view override returns (uint256) {
        return vaultBurnFeePerc;
    }
```

Listing 3.3: `FeePolicy::computeVaultMintFeePerc()/computeVaultBurnFeePerc()`

Recommendation Revise the above-mentioned routines for improved consistency.

Status The issue has been fixed by this commit: `01b01a4`.

3.4 Revisited previewDeposit() Logic in BondHelpers

- ID: PVE-004
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `BondHelpers`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

To interacting with the `ButtonWood`'s `Bond` contract, the `Spot` protocol provides a `BondHelpers` library contract. In the process of examining one specific interaction function to estimate the amount of tranches minted when a given amount of collateral is deposited into the bond, we notice it has an implicit assumption that can be explicitly enforced.

To elaborate, we show below the implementation of this library function, i.e., `previewDeposit()`. Specifically, this function estimates the amount of tranches that will be minted when a given amount of collateral is deposited. It currently takes two arguments: `BondController` and `collateralAmount`. It comes to our attention that there is an implicit assumption, i.e., it assumes the given `BondController` is not mature yet. In other words, if `BondController` is mature, we need to return empty `tranchesOut` or simply revert the calculation.

```
    function previewDeposit(IBondController b, uint256 collateralAmount) internal view
        returns (TokenAmount[] memory) {
        BondTranches memory bt = getTranches(b);
        TokenAmount[] memory tranchesOut = new TokenAmount[](2);

        uint256 totalDebt = b.totalDebt();
        uint256 collateralBalance = IERC20Upgradeable(b.collateralToken()).balanceOf(
            address(b));

        uint256 seniorAmt = collateralAmount.mulDiv(bt.trancheRatios[0],
            TRANCHE_RATIO_GRANULARITY);
        if (collateralBalance > 0) {
            seniorAmt = seniorAmt.mulDiv(totalDebt, collateralBalance);
        }
        tranchesOut[0] = TokenAmount({ token: bt.tranches[0], amount: seniorAmt });

        uint256 juniorAmt = collateralAmount.mulDiv(bt.trancheRatios[1],
            TRANCHE_RATIO_GRANULARITY);
        if (collateralBalance > 0) {
            juniorAmt = juniorAmt.mulDiv(totalDebt, collateralBalance);
        }
        tranchesOut[1] = TokenAmount({ token: bt.tranches[1], amount: juniorAmt });
```

```
        return tranchesOut;
    }
```

Listing 3.4: `BondHelpers::previewDeposit()`

Recommendation Revisit the above routine to validate the given `BondController` is not mature yet.

Status The issue has been fixed by this commit: `f486c0a`.

3.5 Trust Issue of Admin Keys

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

In the `Spot` protocol, there is a special administrative account, i.e., `owner`. This `owner` account plays a critical role in governing and regulating the system-wide operations (e.g., configure various settings and execute privileged operations). It also has the privilege to control or govern the flow of assets within the protocol contracts. In the following, we examine the privileged account and the related privileged accesses in current contracts.

```
    function updateTargetSubscriptionRatio(uint256 targetSubscriptionRatio_) external
        onlyOwner {
        if (targetSubscriptionRatio_ < TARGET_SR_LOWER_BOUND  targetSubscriptionRatio_ >
             TARGET_SR_UPPER_BOUND) {
            revert InvalidTargetSRBounds();
        }
        targetSubscriptionRatio = targetSubscriptionRatio_;
    }

    /// @notice Updates the deviation ratio bounds.
    /// @param deviationRatioBoundLower_ The new lower deviation ratio bound as fixed
        point number with {DECIMALS} places.
    /// @param deviationRatioBoundUpper_ The new upper deviation ratio bound as fixed
        point number with {DECIMALS} places.
    function updateDeviationRatioBounds(
        uint256 deviationRatioBoundLower_,
        uint256 deviationRatioBoundUpper_
    ) external onlyOwner {
        if (deviationRatioBoundLower_ > ONE  deviationRatioBoundUpper_ < ONE) {
```

```
            revert InvalidDRBounds();
        }
        deviationRatioBoundLower = deviationRatioBoundLower_;
        deviationRatioBoundUpper = deviationRatioBoundUpper_;
    }

    /// @notice Updates the perp mint fee parameters.
    /// @param perpMintFeePerc_ The new perp mint fee ceiling percentage
    ///        as a fixed point number with {DECIMALS} places.
    function updatePerpMintFees(uint256 perpMintFeePerc_) external onlyOwner {
        if (perpMintFeePerc_ > ONE) {
            revert InvalidPerc();
        }
        perpMintFeePerc = perpMintFeePerc_;
    }

    /// @notice Updates the perp burn fee parameters.
    /// @param perpBurnFeePerc_ The new perp burn fee ceiling percentage
    ///        as a fixed point number with {DECIMALS} places.
    function updatePerpBurnFees(uint256 perpBurnFeePerc_) external onlyOwner {
        if (perpBurnFeePerc_ > ONE) {
            revert InvalidPerc();
        }
        perpBurnFeePerc = perpBurnFeePerc_;
    }

    /// @notice Update the parameters determining the slope and asymptotes of the sigmoid fee curve.
    /// @param p Lower, Upper and Growth sigmoid parameters are fixed point numbers with {DECIMALS} places.
    function updatePerpRolloverFees(RolloverFeeSigmoidParams calldata p) external onlyOwner {
        // If the bond duration is 28 days and 13 rollovers happen per year,
        // perp can be inflated or enriched up to ~13% annually.
        if (p.lower < -int256(SIGMOID_BOUND)  p.upper > int256(SIGMOID_BOUND)  p.lower > p.upper) {
            revert InvalidSigmoidAsymptotes();
        }
        perpRolloverFee.lower = p.lower;
        perpRolloverFee.upper = p.upper;
        perpRolloverFee.growth = p.growth;
    }

    /// @notice Updates the vault mint fee parameters.
    /// @param vaultMintFeePerc_ The new vault mint fee ceiling percentage
    ///        as a fixed point number with {DECIMALS} places.
    function updateVaultMintFees(uint256 vaultMintFeePerc_) external onlyOwner {
        if (vaultMintFeePerc_ > ONE) {
            revert InvalidPerc();
        }
        vaultMintFeePerc = vaultMintFeePerc_;
    }
```

```
    /// @notice Updates the vault burn fee parameters.
    /// @param vaultBurnFeePerc_ The new vault burn fee ceiling percentage
    ///        as a fixed point number with {DECIMALS} places.
    function updateVaultBurnFees(uint256 vaultBurnFeePerc_) external onlyOwner {
        if (vaultBurnFeePerc_ > ONE) {
            revert InvalidPerc();
        }
        vaultBurnFeePerc = vaultBurnFeePerc_;
    }
```

Listing 3.5: Example Privileged Operations in `FeePolicy`

We understand the need of the privileged functions for proper contract operations, but at the same time the extra power to these privileged accounts may also be a counter-party risk to the contract users. Therefore, we list this concern as an issue here from the audit perspective and highly recommend making these privileges explicit or raising necessary awareness among protocol users.

Recommendation Promptly transfer the privileged account to the intended `DAO`-like governance contract. All changes to privileged operations may need to be mediated with necessary timelocks. Eventually, activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status The issue has been mitigated as the team clarifies that the `owner` account is currently a 2/4 `DAO` multisig. The ownership will eventually be handed off to `ForthDAO` governance + timelock the same as the `AMPL` contracts.

PeckShield

4 Conclusion

In this audit, we have analyzed the design and implementation of the SPOT protocol, which is a decentralized, inflation resistant store of value designed to be resilient in all market conditions. The system bends safely rather than breaking catastrophically in extreme market scenarios, and can forever resume its function without bailouts. SPOT can be held directly or rotated in as an alternative collateral asset to USDC within existing systems. It uses AMPL as the underlying unit of account, Buttonwood Tranche for collateral preparation, and onchain governance through the FORTH DAO. The current code base is clearly organized and those identified issues are promptly confirmed and fixed.

Meanwhile, 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-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

[2] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

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

[4] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

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

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

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

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

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