SMART CONTRACT AUDIT REPORT

for

Spot Protocol

Prepared By: Xiaomi Huang

PeckShield

September 22, 2022

Document Properties

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

Version Info

Version	Date	Author(s)	Description
1.0	September 22, 2022	Xuxian Jiang	Final Release
1.0-rc1	September 15, 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 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 perpetual note backed by fully collateralized AmpleForth (AMPL) derivatives. Spot can fulfill many properties of modern day stablecoins but is not pegged to any particular value. Its price will likely float within a range similar to AMPL, SPOT can be considered as a derivative that strips away most of AMPL's supply volatility. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of The Spot Protocol

Item	Description
Issuer	Fragments, Inc.
Website	https://ampleforth.org
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	September 22, 2022

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 (6c1c28e)

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 (8fe1725)

1.2 About PeckShield

PeckShield Inc. [11] 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 [10]:

- 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 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

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

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) [9], 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	3	
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 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 3 low-severity vulnerabilities and and 1 informational recommendation.

Table 2.1: Key Spot Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Possible Costly Perp Tokens From Improper Initialization	Time And State	Resolved
PVE-002	Low	Improved Reentrancy Protection in PerpetualTranche	Time And State	Resolved
PVE-003	Informational	Improved Validations on Tranche Ratios in BondIssue	Code Practices	Resolved
PVE-004	Low	Trust on Admin Keys	Security Features	Mitigated

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 Possible Costly Perp Tokens From Improper Initialization

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `PerpetualTranche`
- Category: Time and State [6]
- CWE subcategory: CWE-362 [3]

Description

The `Spot` protocol allows users to deposit supported tranche tokens and get in return the perps to represent the overall share/ownership. While examining the perps calculation with the given deposits, we notice an issue that may unnecessarily make the perps extremely expensive and bring hurdles (or even causes loss) for later depositors.

To elaborate, we show below the `deposit()` routine, which is used for participating users to deposit the supported tranche tokens and get respective perps in return. The issue occurs when the `PerpetualTranche` is being initialized under the assumption that it is empty.

```
    function deposit(ITranche trancheIn, uint256 trancheInAmt) external override
        afterStateUpdate whenNotPaused {
        if (IBondController(trancheIn.bond()) != _depositBond) {
            revert UnacceptableDepositTranche(trancheIn, _depositBond);
        }

        // calculates the amount of perp tokens when depositing 'trancheInAmt' of
            tranche tokens
        uint256 perpAmtMint = _computeMintAmt(trancheIn, trancheInAmt);
        if (trancheInAmt == 0  perpAmtMint == 0) {
            revert UnacceptableMintAmt(trancheInAmt, perpAmtMint);
        }

        // calculates the fees to mint 'perpAmtMint' of perp token
        (int256 reserveFee, uint256 protocolFee) = feeStrategy.computeMintFees(
            perpAmtMint);
```

```
418         // transfers tranche tokens from the sender to the reserve
419         _transferIntoReserve(msg.sender, trancheIn, trancheInAmt);

421         // mints perp tokens to the sender
422         _mint(msg.sender, perpAmtMint);

424         // settles fees
425         _settleFee(msg.sender, reserveFee, protocolFee);

427         // updates & enforces supply cap and tranche mint cap
428         mintedSupplyPerTranche[trancheIn] += perpAmtMint;
429         _enforcePerTrancheSupplyCap(trancheIn);
430         _enforceTotalSupplyCap();
431     }
```

Listing 3.1: `PerpetualTranche::deposit()`

```
736     /// @dev Computes the perp mint amount for given amount of tranche tokens deposited
             into the reserve.
737     function _computeMintAmt(ITranche trancheIn, uint256 trancheInAmt) private view
            returns (uint256) {
738         uint256 totalSupply_ = totalSupply();
739         uint256 stdTrancheInAmt = _toStdTrancheAmt(trancheInAmt, computeDiscount(
                trancheIn));
740         uint256 trancheInPrice = computePrice(trancheIn);
741         uint256 perpAmtMint = (totalSupply_ > 0)
742             ? (stdTrancheInAmt * trancheInPrice * totalSupply_) / _reserveValue()
743             : (stdTrancheInAmt * trancheInPrice) / UNIT_PRICE;
744         return (perpAmtMint);
745     }
```

Listing 3.2: `PerpetualTranche::_computeMintAmt()`

Specifically, when it is being initialized (line 410), the minted amount is based on the given `trancheInAmt`, which could be manipulatable by the malicious actor. As this is the first deposit, the current total supply equals 0 and the return amount is computed as `perpAmtMint = (stdTrancheInAmt * trancheInPrice)/ UNIT_PRICE`. With that, the actor can further donate a huge amount of the underlying assets with the goal of making the perps extremely expensive.

An extremely expensive perps can be very inconvenient to use as a small number of 1 `Wei` may denote a large value. Furthermore, it can lead to precision issue in truncating the perp tokens for deposited assets. If truncated to be zero, the deposited assets are essentially considered dust and kept by the pool without returning any pool tokens.

This is a known issue that has been mitigated in popular `Uniswap`. When providing the initial liquidity to the contract (i.e. when totalSupply is 0), the liquidity provider must sacrifice 1000 LP tokens (by sending them to $address(0)$). By doing so, we can ensure the granularity of the LP tokens is always at least 1000 and the malicious actor is not the sole holder. This approach may bring an additional cost for the initial liquidity provider, but this cost is expected to be low and acceptable.

Recommendation Revise current deposit logic to defensively calculate the perps amount when it is being initialized. An alternative solution is to ensure a guarded launch process that safeguards the first deposit to avoid being manipulated.

Status The issue has been resolved as the team will ensure in the deployment process that a deposit operation is performed when the supply is zero and before release.

3.2 Improved Reentrancy Protection in PerpetualTranche

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `PerpetualTranche`
- Category: Time and State [8]
- CWE subcategory: CWE-663 [4]

Description

A common coding best practice in Solidity is the adherence of `checks-effects-interactions` principle. This principle is effective in mitigating a serious attack vector known as `re-entrancy`. Via this particular attack vector, a malicious contract can be reentering a vulnerable contract in a nested manner. Specifically, it first calls a function in the vulnerable contract, but before the first instance of the function call is finished, second call can be arranged to re-enter the vulnerable contract by invoking functions that should only be executed once. This attack was part of several most prominent hacks in Ethereum history, including the `DAO` [13] exploit, and the recent `Uniswap/Lendf.Me` hack [12].

We notice there is an occasion where the `checks-effects-interactions` principle is violated. Using the `PerpetualTranche` as an example, the `redeem()` function (see the code snippet below) is provided to externally transfer redeemed tranches from the reserve to the sender. However, the invocation of an external contract requires extra care in avoiding the above `re-entrancy`.

Apparently, the interaction with the external contract (line 442) starts before effecting the update on internal states (line 446), hence violating the principle. In this particular case, if the external contract has certain hidden logic that may be capable of launching `re-entrancy` via the same entry function.

```
434     function redeem(uint256 perpAmtBurnt) external override afterStateUpdate
            whenNotPaused {
435         // gets the current perp supply
436         uint256 perpSupply = totalSupply();
437
438         // verifies if burn amount is acceptable
```

```
        if (perpAmtBurnt == 0  perpAmtBurnt > perpSupply) {
            revert UnacceptableBurnAmt(perpAmtBurnt, perpSupply);
        }

        // calculates share of reserve tokens to be redeemed
        (IERC20Upgradeable[] memory tokensOuts, uint256[] memory tokenOutAmts) = _computeRedemptionAmts(perpAmtBurnt);

        // calculates the fees to burn `perpAmtBurnt` of perp token
        (int256 reserveFee, uint256 protocolFee) = feeStrategy.computeBurnFees(perpAmtBurnt);

        // updates the mature tranche balance
        _updateMatureTrancheBalance((_matureTrancheBalance * (perpSupply - perpAmtBurnt)) / perpSupply);

        // settles fees
        _settleFee(msg.sender, reserveFee, protocolFee);

        // burns perp tokens from the sender
        _burn(msg.sender, perpAmtBurnt);

        // transfers reserve tokens out
        for (uint256 i = 0; i < tokensOuts.length; i++) {
            if (tokenOutAmts[i] > 0) {
                _transferOutOfReserve(msg.sender, tokensOuts[i], tokenOutAmts[i]);
            }
        }

        // enforces supply reduction
        uint256 newSupply = totalSupply();
        if (newSupply >= perpSupply) {
            revert ExpectedSupplyReduction(newSupply, perpSupply);
        }
    }
```

Listing 3.3: PerpetualTranche::redeem()

In the meantime, we should mention that the supported tokens in the protocol do implement rather standard ERC20 interfaces and their related token contracts are not vulnerable or exploitable for `re-entrancy`. However, it is important to take precautions in making use of `nonReentrant` to block possible `re-entrancy`.

Recommendation Apply necessary reentrancy prevention by utilizing the `nonReentrant` modifier to block possible `re-entrancy`. This suggestion is also applicable to other routines, including `deposit()` and `rollover()`.

Status The issue has been resolved in the following PRs: `94` and `95`.

3.3 Improved Validations on Tranche Ratios in BondIssue

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: BondIssue
- Category: Coding Practices [7]
- CWE subcategory: CWE-1126 [1]

Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The Spot protocol is no exception. Specifically, if we examine the BondIssue contract, it has defined a number of protocol-wide risk parameters, such as minIssueTimeIntervalSec and trancheRatios. In the following, we show the corresponding constructor that initializes these parameters.

```
    constructor(
        IBondFactory bondFactory_,
        uint256 minIssueTimeIntervalSec_,
        uint256 issueWindowOffsetSec_,
        uint256 maxMaturityDuration_,
        address collateral_,
        uint256[] memory trancheRatios_
    ) {
        bondFactory = bondFactory_;
        minIssueTimeIntervalSec = minIssueTimeIntervalSec_;
        issueWindowOffsetSec = issueWindowOffsetSec_;
        maxMaturityDuration = maxMaturityDuration_;

        collateral = collateral_;
        trancheRatios = trancheRatios_;

        lastIssueWindowTimestamp = 0;
    }
```

Listing 3.4: BondIssue::constructor()

These parameters define various aspects of the protocol operation and maintenance and need to exercise extra care when configuring or updating them. Our analysis shows the above logic on these parameters can be improved by applying more rigorous sanity checks. Based on the current implementation, certain corner cases may lead to an undesirable consequence. For example, an unlikely mis-configuration of trancheRatios may break the implicit assumption of the total sum of being 1000, hence adversely affecting the protocol.

Recommendation Validate these system-wide parameters to ensure they fall in an appropriate range.

Status The issue has been resolved in the following PR: `96`.

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [5]
- 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 updateBondIssuer(IBondIssuer bondIssuer_) public onlyOwner {
        if (address(bondIssuer_) == address(0)) {
            revert UnacceptableReference();
        }
        if (address(_reserveAt(0)) != bondIssuer_.collateral()) {
            revert InvalidCollateral(bondIssuer_.collateral(), address(_reserveAt(0)));
        }
        bondIssuer = bondIssuer_;
        emit UpdatedBondIssuer(bondIssuer_);
    }

    /// @notice Update the reference to the fee strategy contract.
    /// @param feeStrategy_ New strategy address.
    function updateFeeStrategy(IFeeStrategy feeStrategy_) public onlyOwner {
        if (address(feeStrategy_) == address(0)) {
            revert UnacceptableReference();
        }
        feeStrategy = feeStrategy_;
        emit UpdatedFeeStrategy(feeStrategy_);
    }

    /// @notice Update the reference to the pricing strategy contract.
    /// @param pricingStrategy_ New strategy address.
    function updatePricingStrategy(IPricingStrategy pricingStrategy_) public onlyOwner {
        if (address(pricingStrategy_) == address(0)) {
            revert UnacceptableReference();
        }
        if (pricingStrategy_.decimals() != PRICE_DECIMALS) {
            revert InvalidStrategyDecimals(pricingStrategy_.decimals(), PRICE_DECIMALS);
```

```
        }
        pricingStrategy = pricingStrategy_;
        emit UpdatedPricingStrategy(pricingStrategy_);
    }
```

Listing 3.5: Example Privileged Operations in `PerpetualTranche`

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 will be managed by a multisig and eventually handed over to `Forth DAO` control.

4 Conclusion

In this audit, we have analyzed the design and implementation of the Spot protocol, which is a perpetual note backed by fully collateralized AmpleForth (AMPL) derivatives. Spot can fulfill many properties of modern day stablecoins but is not pegged to any particular value. Its price will likely float within a range similar to AMPL, SPOT can be considered as a derivative that strips away most of AMPL's supply volatility. 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-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition'). https://cwe.mitre.org/data/definitions/362.html.

[4] MITRE. CWE-663: Use of a Non-reentrant Function in a Concurrent Context. https://cwe.mitre.org/data/definitions/663.html.

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

[6] MITRE. CWE CATEGORY: 7PK - Time and State. https://cwe.mitre.org/data/definitions/361.html.

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

[8] MITRE. CWE CATEGORY: Concurrency. https://cwe.mitre.org/data/definitions/557.html.

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

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

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

[12] PeckShield. Uniswap/Lendf.Me Hacks: Root Cause and Loss Analysis. https://medium.com/@peckshield/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09.

[13] David Siegel. Understanding The DAO Attack. https://www.coindesk.com/understanding-dao-hack-journalists.