SMART CONTRACT AUDIT REPORT

for

Spot Protocol

Prepared By: Xiaomi Huang

PeckShield

May 28, 2025

Document Properties

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

Version Info

Version	Date	Author(s)	Description
1.0	May 28, 2025	Xuxian Jiang	Final Release
1.0-rc1	May 24, 2025	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	May 28, 2025

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 (25a4544)

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 (037d37b)

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 getTwapTick() Logic in UniswapV3PoolHelpers	Business Logic	Resolved
PVE-002	Informational	Simplified Rollover Tokens Retrieval in PerpetualTranche	Coding Practices	Resolved
PVE-003	Low	Enhanced Validation of Single-Side Mint Calculation in BillBroker	Business Logic	Resolved
PVE-004	Low	Trust on Admin Keys	Security Features	Resolved

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.

PeckShield

3 Detailed Results

3.1 Improved getTwapTick() Logic in UniswapV3PoolHelpers

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `UniswapV3PoolHelpers`
- Category: Business Logic [6]
- CWE subcategory: CWE-770 [3]

Description

The `SPOT` protocol has a library contract with helper functions to work with `UniswapV3`-based pools. While examining the specific helper function to calculate the `Time-Weighted Average Price (TWAP)` tick from a `UniswapV3` pool, we notice an issue that needs to be addressed.

In the following, we show the implementation of this specific function, i.e., `getTwapTick()`. It has a rather straightforward logic in calculating the time-weighted average price. However, it can be improved when the difference from the two `tickCumulatives` can not be fully divided by `twapDuration`. In this case, we can adjust the result by rounding to negative infinity.

```
    function getTwapTick(
        IUniswapV3Pool pool,
        uint32 twapDuration
    ) internal view returns (int24) {
        uint32[] memory secondsAgo = new uint32[](2);
        secondsAgo[0] = twapDuration;
        secondsAgo[1] = 0;
        (int56[] memory tickCumulatives, ) = pool.observe(secondsAgo);
        return int24((tickCumulatives[1] - tickCumulatives[0]) / twapDuration);
    }
```

Listing 3.1: `UniswapV3PoolHelpers::getTwapTick()`

Recommendation Improve the above-mentioned routine to handle the case when the result is not fully dividable.

Status This issue has been resolved as the team strictly follows the examples in the Uniswap Oracle docs: `here`.

3.2 Simplified Rollover Tokens Retrieval in PerpetualTranche

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

Description

The `Spot` protocol has a core `PerpetualTranche` contract that is in essence a perpetual note ERC-20 token contract, backed by buttonwood tranches. In the process of examining its logic to retrieve the list of reserve tokens which are up for rollover, we notice its implementation may be improved.

To elaborate, we show below the implementation of this specific function, i.e., `getReserveTokensUpForRollover()`. While it properly computes the list of reserve tokens for rollover, we notice the final step of recreating a smaller array with just the tokens up for rollover can be optimized. Basically, we can simply resize the very first array via the following statement, i.e., `assembly mstore(activeRolloverTokens, numTokensUpForRollover)`. After that, we can then return `activeRolloverTokens`.

```
    function getReserveTokensUpForRollover() external override afterStateUpdate returns (IERC20Upgradeable[] memory) {
        uint8 reserveCount = uint8(_reserves.length());
        IERC20Upgradeable[] memory activeRolloverTokens = new IERC20Upgradeable[](reserveCount);

        // We count the number of tokens up for rollover.
        uint8 numTokensUpForRollover = 0;

        // If any underlying collateral exists it can be rolled over.
        IERC20Upgradeable underlying_ = _reserveAt(0);
        if (underlying_.balanceOf(address(this)) > 0) {
            activeRolloverTokens[0] = underlying_;
            numTokensUpForRollover++;
        }

        // Iterating through the reserve to find tranches that are ready to be rolled out.
        for (uint8 i = 1; i < reserveCount; ++i) {
            IERC20Upgradeable token = _reserveAt(i);
            if (_isTimeForRollout(ITranche(address(token)))) {
                activeRolloverTokens[i] = token;
                numTokensUpForRollover++;
```

```
            }
        }

        // We recreate a smaller array with just the tokens up for rollover.
        IERC20Upgradeable[] memory rolloverTokens = new IERC20Upgradeable[](
            numTokensUpForRollover);
        uint8 j = 0;
        for (uint8 i = 0; i < reserveCount; ++i) {
            if (address(activeRolloverTokens[i]) != address(0)) {
                rolloverTokens[j++] = activeRolloverTokens[i];
            }
        }

        return rolloverTokens;
    }
```

Listing 3.2: `PerpetualTranche::getReserveTokensUpForRollover()`

Recommendation Improve the above routine by avoiding the creation of a smaller array with the same set of reserve tokens for rollover.

Status This issue has been fixed by this commit: `c2ea591`.

3.3 Enhanced Validation of Single-Side Mint Calculation in BillBroker

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

Description

The `SPOT` protocol has a unique `BillBroker` contract to act as an intermediary between parties who want to borrow and lend. With that, it has the natural support for liquidity providers to add/remove liquidity. Our analysis shows that current single-side liquidity addition may require certain pre-condition of runtime contract status.

To elaborate, we show one example routine, i.e., `computeMintAmtWithUSD()`. As the name indicates, this routine computes the amount of `LP` tokens minted when the given number of `USD` tokens are deposited. However, the invocation requires that the pool is at the status of having under-weight `USD`, which is currently not enforced. Similarly, another function `computeMintAmtWithPerp()` requires the under-weight `Perp` condition.

```
    function computeMintAmtWithUSD(
        uint256 usdAmtIn,
        ReserveState memory s
    ) public view returns (uint256 mintAmt) {
        if (usdAmtIn <= 0) {
            return 0;
        }

        uint256 totalSupply_ = totalSupply();
        uint256 valueIn = s.usdPrice.mulDiv(usdAmtIn, usdUnitAmt);
        uint256 totalReserveVal = (s.usdPrice.mulDiv(s.usdBalance, usdUnitAmt) +
            s.perpPrice.mulDiv(s.perpBalance, perpUnitAmt));
        if (totalReserveVal == 0 || totalSupply_ == 0) {
            return 0;
        }

        // Compute mint amount.
        mintAmt = valueIn.mulDiv(totalSupply_, totalReserveVal);

        // A single sided deposit is a combination of swap and mint.
        // We first calculate the amount of usd swapped into perps and
        // apply the swap fee only for that portion.
        // The mint fees are waived, because single sided deposits
        // push the pool back into balance.
        uint256 postOpUsdBal = s.usdBalance + usdAmtIn;
        uint256 postOpUsdClaim = postOpUsdBal.mulDiv(mintAmt, totalSupply_ + mintAmt);
        uint256 percOfAmtInSwapped = ONE.mulDiv(usdAmtIn - postOpUsdClaim, usdAmtIn);
        uint256 feeFactor = computeUSDToPerpSwapFeeFactor(
            assetRatio(s),
            assetRatio(_updatedReserveState(s, postOpUsdBal, s.perpBalance))
        );
        mintAmt =
            mintAmt.mulDiv(percOfAmtInSwapped, ONE).mulDiv(TWO - feeFactor, ONE) +
            mintAmt.mulDiv(ONE - percOfAmtInSwapped, ONE);
    }
```

Listing 3.3: `BillBroker::computeMintAmtWithUSD()`

Recommendation Revise the above-mentioned routines for improved validation of current `USD/Perp` weights.

Status This issue has been fixed by this commit: `515c8b7`.

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- 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 updateVault(IRolloverVault vault_) external onlyOwner {
        vault = vault_;
    }

    /// @notice Updates the reference to the keeper.
    /// @param keeper_ The address of the new keeper.
    function updateKeeper(address keeper_) public onlyOwner {
        keeper = keeper_;
    }

    /// @notice Update the reference to the bond issuer contract.
    /// @param bondIssuer_ New bond issuer address.
    function updateBondIssuer(IBondIssuer bondIssuer_) public onlyOwner {
        if (bondIssuer_.collateral() != address(_reserveAt(0))) {
            revert UnexpectedAsset();
        }
        bondIssuer = bondIssuer_;
    }

    /// @notice Update the reference to the fee policy contract.
    /// @param feePolicy_ New strategy address.
    function updateFeePolicy(IFeePolicy feePolicy_) public onlyOwner {
        if (feePolicy_.decimals() != PERC_DECIMALS) {
            revert UnexpectedDecimals();
        }
        feePolicy = feePolicy_;
    }

    /// @notice Update the maturity tolerance parameters.
    /// @param minTrancheMaturitySec_ New minimum maturity time.
    /// @param maxTrancheMaturitySec_ New maximum maturity time.
    function updateTolerableTrancheMaturity(
```

```
        uint256 minTrancheMaturitySec_,
        uint256 maxTrancheMaturitySec_
    ) public onlyOwner {
        if (minTrancheMaturitySec_ > maxTrancheMaturitySec_) {
            revert UnacceptableParams();
        }
        minTrancheMaturitySec = minTrancheMaturitySec_;
        maxTrancheMaturitySec = maxTrancheMaturitySec_;
    }
```

Listing 3.4: 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 resolved 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.

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.

PeckShield

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-770: Allocation of Resources Without Limits or Throttling. https://cwe.mitre.org/data/definitions/770.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.