SMART CONTRACT AUDIT REPORT

for

Spot Protocol

Prepared By: Xiaomi Huang

PeckShield

June 3, 2023

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	June 3, 2023	Xuxian Jiang	Final Release
1.0-rc1	May 22, 2023	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	June 3, 2023

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 (901435c)

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

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

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) [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	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	Possible Costly Vault Tokens From Improper Initialization	Time And State	Resolved
PVE-002	Low	Improved Precision Calculation in _computeRolloverAmt()	Numeric Errors	Resolved
PVE-003	Low	Accommodation of Non-ERC20-Compliant Tokens	Code Practices	Resolved
PVE-004	Informational	Improved Validations Logic in RouterV1	Code Practices	Resolved
PVE-005	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 Vault Tokens From Improper Initialization

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: RolloverVault
- Category: Time and State [6]
- CWE subcategory: CWE-362 [4]

Description

The Spot protocol allows users to deposit supported tranche tokens and get in return the perps to represent the overall share/ownership. It also has a RolloverVault that generates yield (from fees) by performing rollovers on PerpetualTranche. While examining the vault logic, we notice an issue that may unnecessarily make the vault token 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 collateral and get respective vault tokens in return. The issue occurs when the RolloverVault is being initialized under the assumption that it is empty.

```
    function deposit(uint256 amount) external override nonReentrant whenNotPaused
        returns (uint256) {
        uint256 totalSupply_ = totalSupply();
        uint256 notes = (totalSupply_ > 0) ? totalSupply_.mulDiv(amount, getTVL()) : (
            amount * INITIAL_RATE);

        underlying.safeTransferFrom(_msgSender(), address(this), amount);
        _syncAsset(underlying);

        _mint(_msgSender(), notes);
        return notes;
    }
```

Listing 3.1: RolloverVault::deposit()

```
    function getTVL() public override returns (uint256) {
        uint256 totalValue = 0;

        // The underlying balance
        totalValue += underlying.balanceOf(address(this));

        // The deployed asset value denominated in the underlying
        for (uint256 i = 0; i < _deployed.length(); i++) {
            ITranche tranche = ITranche(_deployed.at(i));
            uint256 trancheBalance = tranche.balanceOf(address(this));
            if (trancheBalance > 0) {
                (uint256 collateralBalance, uint256 trancheSupply) = tranche.
                    getTrancheCollateralization();
                totalValue += collateralBalance.mulDiv(trancheBalance, trancheSupply);
            }
        }

        // The earned asset (perp token) value denominated in the underlying
        uint256 perpBalance = perp.balanceOf(address(this));
        if (perpBalance > 0) {
            // The "earned" asset is assumed to be the perp token.
            // Perp tokens are assumed to have the same denomination as the underlying
            totalValue += perpBalance.mulDiv(IPerpetualTranche(address(perp)).
                getAvgPrice(), PERP_UNIT_PRICE);
        }

        return totalValue;
    }
```

Listing 3.2: `RolloverVault::getTVL()`

Specifically, when it is being initialized (line 282), the minted amount is based on the given `amount`, 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 `amount * INITIAL_RATE`. 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 vault tokens 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 vault 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 vault token 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 ensures that there is a decimal offset between the notes and underlying denominations. Also, the team plans to follow a guarded launch process and observe rollovers through the vault for 4 bond cycles (i.e., 28 days) before the public launch.

3.2 Improved Precision Calculation in _computeRolloverAmt()

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low

- Target: PerpetualTranche
- Category: Numeric Errors [8]
- CWE subcategory: CWE-190 [2]

Description

The lack of float support in Solidity may introduce another subtle, but troublesome issue: precision loss. In this section, we examine one possible precision loss source that stems from the rollover operations on PerpetualTranche.

In particular, we show below the related _computeRolloverAmt() routine. As the name indicates, this routine computes the amount of reserve tokens that can be rolled out for the given amount of tranches deposited. We notice that when the token out balance is not covered, the current logic re-calculates the values by initially setting tokenOutAmt to be tokenOutAmtRequested (lines 963-971). And the calculation may introduce possible precision loss.

```
925     function _computeRolloverAmt(
926         ITranche trancheIn,
927         IERC20Upgradeable tokenOut,
928         uint256 trancheInAmtAvailable,
929         uint256 tokenOutAmtRequested
930     ) private view returns (IPerpetualTranche.RolloverPreview memory) {
931         IPerpetualTranche.RolloverPreview memory r;

933         uint256 trancheInDiscount = computeDiscount(trancheIn);
934         uint256 trancheOutDiscount = computeDiscount(tokenOut);
935         uint256 trancheInPrice = computePrice(trancheIn);
936         uint256 trancheOutPrice = computePrice(tokenOut);
937         uint256 tokenOutBalance = _reserveBalance(tokenOut);
938         tokenOutAmtRequested = MathUpgradeable.min(tokenOutAmtRequested, tokenOutBalance
                );

940         if (trancheInDiscount == 0  trancheOutDiscount == 0  trancheInPrice == 0
                trancheOutPrice == 0) {
```

```
941             r.remainingTrancheInAmt = trancheInAmtAvailable;
942             return r;
943         }

945         r.trancheInAmt = trancheInAmtAvailable;
946         uint256 stdTrancheInAmt = _toStdTrancheAmt(trancheInAmtAvailable,
                trancheInDiscount);

948         // Basic rollover:
949         // (stdTrancheInAmt . trancheInPrice) = (stdTrancheOutAmt . trancheOutPrice)
950         uint256 stdTrancheOutAmt = stdTrancheInAmt.mulDiv(trancheInPrice,
                trancheOutPrice);
951         r.trancheOutAmt = _fromStdTrancheAmt(stdTrancheOutAmt, trancheOutDiscount);

953         // However, if the tokenOut is the mature tranche (held as naked collateral),
954         // we infer the tokenOut amount from the tranche denomination.
955         // (tokenOutAmt = collateralBalance * trancheOutAmt / matureTrancheBalance)
956         bool isMatureTrancheOut = _isMatureTranche(tokenOut);
957         r.tokenOutAmt = isMatureTrancheOut
958             ? tokenOutBalance.mulDiv(r.trancheOutAmt, _matureTrancheBalance)
959             : r.trancheOutAmt;

961         // When the token out balance is NOT covered:
962         // we fix tokenOutAmt = tokenOutAmtRequested and back calculate other values
963         if (r.tokenOutAmt > tokenOutAmtRequested) {
964             r.tokenOutAmt = tokenOutAmtRequested;
965             r.trancheOutAmt = isMatureTrancheOut
966                 ? _matureTrancheBalance.mulDiv(r.tokenOutAmt, tokenOutBalance)
967                 : r.tokenOutAmt;
968             stdTrancheOutAmt = _toStdTrancheAmt(r.trancheOutAmt, trancheOutDiscount);
969             stdTrancheInAmt = stdTrancheOutAmt.mulDiv(trancheOutPrice, trancheInPrice);
970             r.trancheInAmt = _fromStdTrancheAmt(stdTrancheInAmt, trancheInDiscount);
971         }

973         r.perpRolloverAmt = (stdTrancheOutAmt * trancheOutPrice).mulDiv(totalSupply(),
                _reserveValue());
974         r.remainingTrancheInAmt = trancheInAmtAvailable - r.trancheInAmt;
975         return r;
976     }
```

Listing 3.3: PerpetualTranche::_computeRolloverAmt()

Specifically, we notice the calculation of the resulting `stdTrancheInAmt = stdTrancheOutAmt.mulDiv(trancheOutPrice, trancheInPrice)` (line 969) takes a floor division, which apparently favors the trading user. The design here is better improved to take a ceiling dvision so that the computation is chosen in favor of the protocol. Note that the resulting precision loss may be just a small number, but it plays a critical role when certain boundary conditions are met.

Recommendation Revise the above calculations to better mitigate possible precision loss.

Status The issue has been fixed by this commit: `323acd9`.

3.3 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `RolloverVault`, `RouterV1`
- Category: Coding Practices [7]
- CWE subcategory: CWE-1126 [1]

Description

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

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

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

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

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

Listing 3.4: USDT Token **Contract**

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

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

Listing 3.5: `SafeERC20::safeApprove()`

In current implementation, if we examine the `RouterV1::_checkAndApproveMax()` routine that is designed to check if the spender has sufficient allowance. If not, there is a need to approve the maximum possible amount. To accommodate the specific idiosyncrasy, there is a need to make use of `safeApprove()` twice: the first time resets the allowance to be 0 and the second time sets the intended amount (line 307).

```
300     function _checkAndApproveMax(
301         IERC20Upgradeable token,
302         address spender,
303         uint256 amount
304     ) private {
305         uint256 allowance = token.allowance(address(this), spender);
306         if (allowance < amount) {
307             token.safeApprove(spender, type(uint256).max);
308         }
309     }
```

Listing 3.6: `RouterV1::_checkAndApproveMax()`

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

Status The issue has been fixed by this commit: `89b50a5`.

3.4 Improved Validations Logic in RouterV1

- ID: PVE-004
- Severity: Informational
- Likelihood: N/A
- Impact: N/A

- Target: RouterV1
- Category: Coding Practices [7]
- CWE subcategory: CWE-1126 [1]

Description

To facilitate the user interaction, the Spot protocol provides a RouterV1 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 four 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 or simply obtain the bond directly from perp (without passing it as an argument).

```
    function trancheAndDeposit(
        IPerpetualTranche perp,
        IBondController bond,
        uint256 collateralAmount,
        uint256 feePaid
    ) external afterPerpStateUpdate(perp) {
        BondTranches memory bt = bond.getTranches();
        IERC20Upgradeable collateralToken = IERC20Upgradeable(bond.collateralToken());
        IERC20Upgradeable feeToken = perp.feeToken();

        // transfers collateral & fees to router
        collateralToken.safeTransferFrom(msg.sender, address(this), collateralAmount);
        if (feePaid > 0) {
            feeToken.safeTransferFrom(msg.sender, address(this), feePaid);
        }

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

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

Listing 3.7: RouterV1::trancheAndDeposit()

Recommendation Validate the given bond argument in the above routine to be the deposit bond of the given perp. Note another routine trancheAndRollover() shares the same issue.

Status The issue has been resolved as the perp contract validates whether the tranches entering the system are part of the current deposit bond.

3.5 Trust Issue of Admin Keys

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: Multiple Contracts
- Category: Security Features [5]
- CWE subcategory: CWE-287 [3]

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 pause() public onlyKeeper {
        _pause();
    }

    /// @notice Unpauses deposits, withdrawals and rollovers.
    /// @dev NOTE: ERC-20 functions, like transfers will always remain operational.
    function unpause() public onlyKeeper {
        _unpause();
    }

    /// @notice Updates the reference to the keeper.
    /// @param newKeeper The address of the new keeper.
    function updateKeeper(address newKeeper) public virtual onlyOwner {
        address prevKeeper = keeper;
        keeper = newKeeper;
        emit UpdatedKeeper(prevKeeper, newKeeper);
    }

    /// @notice Updates the authorized roller set.
    /// @dev CAUTION: If the authorized roller set is empty, all rollers are authorized.
    /// @param roller The address of the roller.
    /// @param authorize If the roller is to be authorized or unauthorized.
    function authorizeRoller(address roller, bool authorize) external onlyOwner {
        if (authorize && !_rollers.contains(roller)) {
            _rollers.add(roller);
        } else if (!authorize && _rollers.contains(roller)) {
            _rollers.remove(roller);
        } else {
            return;
        }

        emit UpdatedRollerAuthorization(roller, authorize);
    }

    /// @notice Update the reference to the bond issuer contract.
    /// @param bondIssuer_ New bond issuer address.
    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_);
    }
```

Listing 3.8: 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 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-190: Integer Overflow or Wraparound. https://cwe.mitre.org/data/definitions/190.html.

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

[4] MITRE. CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition'). https://cwe.mitre.org/data/definitions/362.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: Numeric Errors. https://cwe.mitre.org/data/definitions/189.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.