SMART CONTRACT AUDIT REPORT

for

PRINT3R Protocol

Prepared By: Xiaomi Huang

PeckShield

July 10, 2024

Document Properties

Client	PRINT3R
Title	Smart Contract Audit Report
Target	PRINT3R
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	July 10, 2024	Xuxian Jiang	Final Release
1.0-rc	June 21, 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 source code of the PRINT3R 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 PRINT3R

PRINT3R is an innovative perpetual futures protocol designed to create permissionless trading markets. By addressing significant scaling issues found in existing platforms, PRINT3R leverages unique innovations, notably the use of Chainlink functions to securely perform essential computations off-chain. This allows users to trade a wide range of assets, from top 100 crypto tokens to the latest memecoin trends. Additionally, anyone can easily create a trading pool, similar to launching liquidity pools on current DEXs. Participants can also help secure the network and earn financial rewards by executing transactions, liquidating under-collateralized positions, or auto-deleveraging overheated markets. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of The PRINT3R

Item	Description
Name	PRINT3R
Website	https://print3r.xyz/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	July 10, 2024

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

- https://github.com/PRINT3Rxyz/V2.git (8d25bd0)

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

- https://github.com/PRINT3Rxyz/V2.git (c53d7a9)

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.

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.

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

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 implementation of the `PRINT3R` 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	2	■■
Medium	8	■■■■■■■■
Low	0	
Informational	0	
Total	10	

We have so far identified a list of potential issues. 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 2 high-severity vulnerabilities and 8 medium-severity vulnerabilities.

Table 2.1: Key PRINT3R Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Improper Fee Accumulation Logic in FeeDistributor	Business Logic	Resolved
PVE-002	High	Improper Total Funding Fee Calculation in Position	Coding Practices	Resolved
PVE-003	Medium	Incorrect New Token Addition Logic in Multi-Asset Market	Business Logic	Resolved
PVE-004	Medium	Revisited New Asset Support in PriceFeed	Business Logic	Resolved
PVE-005	Medium	Incorrect ADL Impact Calculation Logic in Execution	Business Logic	Resolved
PVE-006	Medium	Incorrect Average Price Update Logic in MarketUtils	Business Logic	Resolved
PVE-007	Medium	Trust Issue of Admin Keys	Security Features	Mitigated
PVE-008	Medium	Incorrect FundingRate And Velocity Update in Funding	Business Logic	Resolved
PVE-009	Medium	Revisited _getUniswapV3Price() Logic in Oracle	Business Logic	Resolved
PVE-010	High	Possible Liquidation of Healthy User Positions	Business Logic	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.

3 | Detailed Results

3.1 Improper Fee Accumulation Logic in FeeDistributor

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: FeeDistributor
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The PRINT3R protocol has a core FeeDistributor contract that is designed to accumulate and distribute protocol fees. In the process of examining current fee accumulation logic, we notice its implementation has a flaw that needs to be fixed.

In the following, we show the implementation of the affected routine – accumulateFees(). It has a rather straightforward logic in collecting the given fee amount (_wethAmount and _usdcAmount) and then updating the cumulative fee amount as well as the tokens per interval for distribution. While the cumulative fee amount is properly updated, the tokens per interval is not. Instead, the correct approach to update them are the following: accumulatedFees[vault].wethTokensPerInterval = (_wethAmount + wethRemaining)/ SECONDS_PER_WEEK and accumulatedFees[vault].usdcTokensPerInterval = (_usdcAmount + usdcRemaining)/ SECONDS_PER_WEEK (lines 75-76).

```
56     function accumulateFees(uint256 _wethAmount, uint256 _usdcAmount) external {
57         address vault = msg.sender;
58         if (!isVault[vault]) revert FeeDistributor_InvalidVault();

60         // Transfer in the WETH and USDC
61         IERC20(weth).safeTransferFrom(msg.sender, address(this), _wethAmount);
62         IERC20(usdc).safeTransferFrom(msg.sender, address(this), _usdcAmount);

64         // Get remaining rewards from last distribution period
65         (uint256 distributedWeth, uint256 distributedUsdc) = pendingRewards(vault);
66         uint256 wethRemaining = accumulatedFees[vault].wethAmount - distributedWeth;
67         uint256 usdcRemaining = accumulatedFees[vault].usdcAmount - distributedUsdc;
```

```
        // Accumulate the fees
        accumulatedFees[vault].wethAmount += _wethAmount;
        accumulatedFees[vault].usdcAmount += _usdcAmount;
        accumulatedFees[vault].lastDistributionTime = block.timestamp;

        // Set the Tokens per interval (week) for WETH and USDC
        accumulatedFees[vault].wethTokensPerInterval = _wethAmount + wethRemaining /
            SECONDS_PER_WEEK;
        accumulatedFees[vault].usdcTokensPerInterval = _usdcAmount + usdcRemaining /
            SECONDS_PER_WEEK;
        // Emit an event
        emit FeesAccumulated(vault, _wethAmount, _usdcAmount);
    }
```

Listing 3.1: `FeeDistributor::accumulateFees()`

Recommendation Improve the above-mentioned routine to properly accumulate fee and update tokens per internal for distribution.

Status This issue has been fixed by the following commit: `19a0ec17`.

3.2 Improper Total Funding Fee Calculation in Position

- ID: PVE-002
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: `Position`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1041 [1]

Description

As mentioned earlier, `PRINT3R` is a perpetual futures protocol that is designed to create permissionless trading markets. And the user positions are managed in a core `Position` contract. While analyzing the position-related funding fee collection, we notice the fee amount is incorrectly calculated.

In the following, we show the implementation of the related routine, i.e., `getTotalFundingFees()`. For a position, its funding fee is computed by multiplying the funding fee delta with the position size. It comes to our attention that the position size is maintained as a dollar amount. However, the multiplication makes use of the `percentageInt()` helper routine, which should be replaced with `percentageUsd()`.

```
    function getTotalFundingFees(MarketId _id, IMarket market, Data memory _position,
        uint256 _indexPrice)
        internal
        view
```

```
        returns (int256)
    {
        (, int256 nextFundingAccrued) = Funding.calculateNextFunding(_id, market,
            _position.ticker, _indexPrice);

        return _position.size.toInt256().percentageInt(nextFundingAccrued - _position.
            fundingParams.lastFundingAccrued);
    }
```

Listing 3.2: `Position::getTotalFundingFees()`

Moreover, the related `Borrow` contract shares another related issue in its `calculatePendingFees()`. Specifically, the resulting `pendingFees` should be further adjusted with the elapsed time duration as follows: `borrowRate.percentage(timeElapsed, SECONDS_PER_DAY)` (line 126).

```
113     function calculatePendingFees(MarketId _id, IMarket market, string calldata _ticker,
             bool _isLong)
114         public
115         view
116         returns (uint256 pendingFees)
117     {
118         uint256 borrowRate = market.getBorrowingRate(_id, _ticker, _isLong);
119
120         if (borrowRate == 0) return 0;
121
122         uint256 timeElapsed = block.timestamp - market.getLastUpdate(_id, _ticker);
123
124         if (timeElapsed == 0) return 0;
125
126         pendingFees = borrowRate * timeElapsed;
127     }
```

Listing 3.3: `Position::calculatePendingFees()`

Recommendation Revisit the above routine to properly compute a position's funding fee. Also, other related routines `_calculateFees()`, `_calculateAmountAfterFees()`, and `decreasePosition()` in `Execution` should also be improved for proper fee collection.

Status This issue has been fixed by the following commits: `19a0ec1` and `1abbd64`.

3.3 Incorrect New Token Addition Logic in Multi-Asset Market

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Market`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The `PRINT3R` protocol has a core `Market` contract to maintain market-wide accounting. By design, it supports the trading of multiple assets under the same liquidity. In the process of analyzing the multi-asset support, we notice the new token addition logic can be improved.

In the following, we show the implementation of the related routine, i.e., `addToken()`. As the name indicates, this routine is used to dynamically add a new token and accordingly support the share reallocation among supported tokens. However, it comes to our attention that the new token's pool is initialized (line 133) after the pool share reallocation (line 131). This is incorrect as the pool share allocation should be performed after the new token pool initialization.

```
112     function addToken(
113         MarketId _id,
114         Pool.Config calldata _config,
115         string memory _ticker,
116         bytes calldata _newAllocations,
117         bytes32 _priceRequestKey
118     ) external onlyPoolOwner(_id) {
119         Pool.GlobalState storage state = globalState[_id];

121         if (!state.isMultiAsset) revert Market_SingleAssetMarket();
122         if (state.assetIds.length() >= MAX_ASSETS) revert Market_MaxAssetsReached();
123         bytes32 assetId = keccak256(abi.encode(_ticker));
124         if (state.assetIds.contains(assetId)) revert Market_TokenAlreadyExists();

126         Pool.validateConfig(_config);

128         if (!state.assetIds.add(assetId)) revert Market_FailedToAddAssetId();
129         state.tickers.push(_ticker);

131         _reallocate(_id, _newAllocations, _priceRequestKey);

133         Pool.initialize(marketStorage[_id][assetId], _config);
134     }
```

Listing 3.4: `Market::addToken()`

Recommendation Revise the above routine by initializing the new token pool before the pool share re-allocation.

Status This issue has been fixed by the following commit: `19a0ec17`.

3.4 Revisited New Asset Support in PriceFeed

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `PriceFeed`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

As mentioned earlier, the `Market` support in `PRINT3R` allows for the trading of multiple assets under the same liquidity. With that, the related oracle is required to dynamically add new token to query token prices. Our analysis shows the current oracle needs to be improved when adding a new token.

In the following, we show the implementation of the related routine – `supportAsset()`. For the new token, it basically maintains the correct mapping from the new token to the related pricing strategy (line 167). However, it forgets to maintain the related token decimals, i.e., `tokenDecimals[_ticker] = _tokenDecimals`. The lack of the new token's decimals will make the base unit of queried token price unavailable and possibly revert the oracle operation.

```
159     function supportAsset(string memory _ticker, SecondaryStrategy calldata _strategy,
            uint8 _tokenDecimals)
160         external
161         onlyRoles(_ROLE_0)
162     {
163         bytes32 assetId = keccak256(abi.encode(_ticker));
164         if (assetIds.contains(assetId)) return; // Return if already supported
165         bool success = assetIds.add(assetId);
166         if (!success) revert PriceFeed_AssetSupportFailed();
167         strategies[_ticker] = _strategy;
168         emit AssetSupported(_ticker, _tokenDecimals);
169     }
```

Listing 3.5: `PriceFeed::supportAsset()`

Recommendation Improve the above-mentioned routine to properly maintain the decimals for the new token asset.

Status This issue has been fixed by the following commit: `19a0ec17`.

3.5 Incorrect ADL Impact Calculation Logic in Execution

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Execution`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The `PRINT3R` protocol has a core `Execution` contract that is designed to execute user orders and update user positions. A specific order is named `ADL`, which aims to automatically de-leverage the user position if the position's profit reaches the protocol-specified threshold. The `ADL` execution needs to adjust the execution price for the affected positions within specific boundaries to maintain market health. Our analysis shows the current approach to calculate the execution price is incorrect.

In the following, we show the implementation of the related routine, i.e., `_executeAdlImpact()`. We notice the use of `percentage()` to calculate acceleration factor `accelerationFactor` (line 751), which should be revised as below: `accelerationFactor = (_pnlToPoolRatio - TARGET_PNL_FACTOR).percentage (PRECISION, TARGET_PNL_FACTOR)`. Similarly, the pool impact needs to be corrected as `pnlImpact = pnlImpact.percentage(PRECISION, _poolUsd)` (line 755).

```
743     function _executeAdlImpact(
744         uint256 _indexPrice,
745         uint256 _averageEntryPrice,
746         uint256 _pnlBeingRealized,
747         uint256 _poolUsd,
748         uint256 _pnlToPoolRatio,
749         bool _isLong
750     ) private pure returns (uint256 impactedPrice) {
751         uint256 accelerationFactor = (_pnlToPoolRatio - TARGET_PNL_FACTOR).percentage(
                TARGET_PNL_FACTOR);

753         uint256 pnlImpact = _pnlBeingRealized * accelerationFactor / PRECISION;

755         uint256 poolImpact = pnlImpact.percentage(_poolUsd);

757         if (poolImpact > PRECISION) poolImpact = PRECISION;

759         // Calculate the minimum profit price for the position, where profit = 5% of
                position (average entry price +- 5%)
760         uint256 minProfitPrice = _isLong
761             ? _averageEntryPrice + (_averageEntryPrice.percentage(MIN_PROFIT_PERCENTAGE)
                    )
762             : _averageEntryPrice - (_averageEntryPrice.percentage(MIN_PROFIT_PERCENTAGE)
                    );
```

```
        uint256 priceDelta = (_indexPrice.absDiff(minProfitPrice) * poolImpact) /
            PRECISION;

        if (_isLong) impactedPrice = _indexPrice - priceDelta;
        else impactedPrice = _indexPrice + priceDelta;
    }
```

Listing 3.6: `Execution::_executeAdlImpact()`

Recommendation Improve the above-mentioned routine to properly adjust the execution price for an `ADL` order.

Status This issue has been fixed by the following commit: `19a0ec17`.

3.6 Incorrect Average Price Update Logic in MarketUtils

- ID: PVE-006
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `MarketUtils`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The `PRINT3R` protocol has a key `Positions` contract that allows the user to create or adjust his/her trading positions. While examining the current position-related logic, we notice the price adjustment of an increased position can be improved.

To elaborate, we show below the code snippet from the related `calculateWeightedAverageEntryPrice()` routine from `MarketUtils`. As the name indicates, this routine computes the next average price when a position is adjusted with `_sizeDelta` (line 380). Specifically, for current position of `_prevPositionSize` with its `_prevAverageEntryPrice`, if it is increased by `_sizeDelta` with the latest mark price `_indexPrice`, the next average price is currently computed as `(_prevPositionSize * _prevAverageEntryPrice + _sizeDelta * _indexPrice)/(prevPositionSize + _sizeDelta)`, which needs to be revised as `(_prevPositionSize + _sizeDelta)/(_prevPositionSize / _prevAverageEntryPrice + _sizeDelta / _indexPrice)`.

```
377     function calculateWeightedAverageEntryPrice(
378         uint256 _prevAverageEntryPrice,
379         uint256 _prevPositionSize,
380         int256 _sizeDelta,
381         uint256 _indexPrice
382     ) internal pure returns (uint256) {
383         if (_sizeDelta <= 0) {
384             // If full close, Avg Entry Price is reset to 0
385             if (_sizeDelta == -_prevPositionSize.toInt256()) return 0;
```

```
            // Else, Avg Entry Price doesn't change for decrease
            else return _prevAverageEntryPrice;
        }

        // Increasing position size
        uint256 newPositionSize = _prevPositionSize + _sizeDelta.abs();

        uint256 numerator = (_prevAverageEntryPrice * _prevPositionSize) + (_indexPrice
            * _sizeDelta.abs());

        uint256 newAverageEntryPrice = numerator / newPositionSize;

        return newAverageEntryPrice;
    }
```

Listing 3.7: `MarketUtils::calculateWeightedAverageEntryPrice()`

Recommendation Revise the above routine to properly compute the next average price when a position is increased.

Status This issue has been fixed by the following commit: `c1aed195`.

3.7 Trust Issue of Admin Keys

- ID: PVE-007
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

In the `PRINT3R` protocol, there is a special `owner` account that plays a critical role in governing and regulating the protocol-wide operations (e.g., assign roles, manage price oracles, configure parameters, and execute various privileged operations). Our analysis shows that the `owner` account and other privileged roles need to be scrutinized. In the following, we examine these privileged accounts and their related privileged accesses in current contracts.

```
    function setRewardTracker(address _rewardTracker) external onlyOwner {
        rewardTracker = IGlobalRewardTracker(_rewardTracker);
    }

    function setFeedValidators(
        address _chainlinkFeedRegistry,
        address _pyth,
        address _uniV2Factory,
```

```
        address _uniV3Factory
    ) external onlyOwner {
        feedRegistry = FeedRegistryInterface(_chainlinkFeedRegistry);
        pyth = IPyth(_pyth);
        uniV2Factory = IUniswapV2Factory(_uniV2Factory);
        uniV3Factory = IUniswapV3Factory(_uniV3Factory);
    }

    function setDefaultConfig(Pool.Config memory _defaultConfig) external onlyOwner {
        defaultConfig = _defaultConfig;
        emit DefaultConfigSet();
    }

    function updatePriceFeed(IPriceFeed _priceFeed) external onlyOwner {
        priceFeed = _priceFeed;
    }

    function updateMarketFees(uint256 _marketCreationFee, uint256 _marketExecutionFee,
        uint256 _priceSupportFee)
        external
        onlyOwner
    {
        marketCreationFee = _marketCreationFee;
        marketExecutionFee = _marketExecutionFee;
        priceSupportFee = _priceSupportFee;
    }

    /// @dev - Merkle Trees used as whitelists for all valid Pyth Price Feed Ids and
        Stablecoin Addresses
    /// These are used for feed validation w.r.t secondary strategies
    function updateMerkleRoot(bytes32 _stablecoinMerkleRoot) external onlyOwner {
        stablecoinMerkleRoot = _stablecoinMerkleRoot;
    }

    function updateFeeDistributor(address _feeDistributor) external onlyOwner {
        feeDistributor = IFeeDistributor(_feeDistributor);
    }

    function updatePositionManager(address _positionManager) external onlyOwner {
        positionManager = IPositionManager(_positionManager);
    }

    /// @dev  withdrawableAmount = balance - reserved incentives
    function withdrawCreationTaxes() external onlyOwner {
        uint256 withdrawableAmount = address(this).balance - (marketExecutionFee *
            requests.length());

        SafeTransferLib.safeTransferETH(payable(msg.sender), withdrawableAmount);
    }
```

Listing 3.8: Example Privileged Operations in `MarketFactory`

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.

In the meantime, the above contract makes use of the proxy contract to allow for future upgrades. The upgrade is a privileged operation, which also falls in this trust issue on the admin key.

Recommendation Promptly transfer the privileged accounts 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 confirms the use of a multi-sig account as the admin.

3.8 Incorrect FundingRate And Velocity Update in Funding

- ID: PVE-008
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Funding`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The `PRINT3R` protocol has a `Funding` library contract that is designed to facilitate the funding-related calculations. In the process of examining current update logic of funding rates, we notice its implementation has a flaw that needs to be fixed.

In the following, we show the implementation of the affected routine – `updateState()`. It has a rather straightforward logic in updating the given pool's `fundingRate`, `fundingAccruedUsd`, and `fundingRateVelocity`. Note the `fundingRateVelocity` update should come after the `fundingRate` update. However, current implementation incorrectly updates `fundingRateVelocity` before updating `fundingRate`.

```
    function updateState(
        MarketId _id,
        IMarket market,
        Pool.Storage storage pool,
        string calldata _ticker,
        uint256 _indexPrice,
        int256 _sizeDelta,
        bool _isLong
    ) internal {
```

```
        int256 nextSkew = _calculateNextSkew(_id, market, _ticker, _sizeDelta, _isLong);

        pool.fundingRateVelocity =
            getCurrentVelocity(market, nextSkew, pool.config.maxFundingVelocity, pool.
                config.skewScale).toInt64();

        (pool.fundingRate, pool.fundingAccruedUsd) = calculateNextFunding(_id, market,
            _ticker, _indexPrice);
    }
```

Listing 3.9: `Funding::updateState()`

Recommendation Improve the above-mentioned routine to properly update various pool's states.

Status This issue has been fixed by the following commit: `19a0ec17`.

3.9 Revisited _getUniswapV3Price() Logic in Oracle

- ID: PVE-009
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Oracle`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The `PRINT3R` protocol has a core `Oracle` contract that provides a reliable approach to query the prices of supported assets. While examining the `UniswapV3`-based price support, we notice it is incorrectly implemented.

In the following, we show the related implementation in the `_getUniswapV3Price()`. This routine has two issues. The first one is the lack of differentiation of two different feed types – `FeedType.UNI_V30` and `FeedType.UNI_V31` – in the final token price calculation. In particular, while `indexToken` and `stableToken` are properly identified, the related `baseUnit` should be computed based on `indexToken`, not `stableToken`. Also, the second issue is that the current price is only applicable for the `FeedType.UNI_V30` case, not `FeedType.UNI_V31`.

```
    function _getUniswapV3Price(IPriceFeed.SecondaryStrategy memory _strategy) private
        view returns (uint256 price) {
        if (_strategy.feedType != IPriceFeed.FeedType.UNI_V30 && _strategy.feedType !=
            IPriceFeed.FeedType.UNI_V31) {
            revert Oracle_InvalidReferenceQuery();
        }
        IUniswapV3Pool pool = IUniswapV3Pool(_strategy.feedAddress);
```

```
        (uint160 sqrtPriceX96,,,,,,) = pool.slot0();

        address indexToken;
        address stableToken;
        if (_strategy.feedType == IPriceFeed.FeedType.UNI_V30) {
            indexToken = pool.token0();
            stableToken = pool.token1();
        } else {
            indexToken = pool.token1();
            stableToken = pool.token0();
        }

        (bool successStable, uint256 stablecoinDecimals) = _tryGetAssetDecimals(IERC20(
            stableToken));
        if (!successStable) revert Oracle_InvalidAmmDecimals();

        uint256 baseUnit = 10 ** stablecoinDecimals;
        UD60x18 numerator = ud(uint256(sqrtPriceX96)).powu(2).mul(ud(baseUnit));
        UD60x18 denominator = ud(2).powu(192);

        // Scale and return the price to 30 decimal places
        price = unwrap(numerator.div(denominator)) * (10 ** (PRICE_DECIMALS -
            stablecoinDecimals));
    }
```

Listing 3.10: `Oracle::_getUniswapV3Price()`

Recommendation Improve the above-mentioned routine to properly compute the `UniswapV3`-based price

Status This issue has been fixed by the following commit: `ab28cf4`.

3.10 Possible Liquidation of Healthy User Positions

- ID: PVE-010
- Severity: High
- Likelihood: Medium
- Impact: Medium
- Target: `TradeEngine`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The `PRINT3R` protocol has a core `TradeEngine` contract that is designed to execute user orders and update user positions. In particular, when a position is under water, the position may be liquidated. Our analysis on current liquidation logic indicates that a healthy user position may also be liquidated.

In the following, we show the implementation of the related routine – `liquidatePosition()`. As the name indicates, this routine is used to liquidate a user position. However, it comes to our attention that the given user position is not validated to meet the liquidation condition. As a result, a healthy user position may also be liquidated.

```
    function liquidatePosition(MarketId _id, bytes32 _positionKey, bytes32 _requestKey,
        address _liquidator)
        external
        onlyRoles(_ROLE_4)
        nonReentrant
    {
        IVault vault = market.getVault(_id);

        Position.Data memory position = tradeStorage.getPosition(_id, _positionKey);

        if (position.user == address(0)) revert TradeEngine_PositionDoesNotExist();

        uint48 requestTimestamp = priceFeed.getRequestTimestamp(_requestKey);
        Execution.validatePriceRequest(priceFeed, _liquidator, _requestKey);

        Execution.Prices memory prices =
            Execution.getTokenPrices(priceFeed, position.ticker, requestTimestamp,
                position.isLong, false);

        // No price impact on Liquidations
        prices.impactedPrice = prices.indexPrice;

        _updateMarketState(_id, prices, position.ticker, position.size, position.isLong,
             false);

        Position.Settlement memory params =
            Position.createLiquidationOrder(position, prices.collateralPrice, prices.
                collateralBaseUnit, _liquidator);

        _decreasePosition(_id, vault, params, prices);

        _liquidatePositionEvent(_id, _positionKey, position, prices.indexPrice, params.
            request.input.collateralDelta);
    }
```

Listing 3.11: `TradeEngine::liquidatePosition()`

Recommendation Improve the above-mentioned routine to ensure only a under-water user position can be liquidated.

Status This issue has been fixed by the following commits: `3e8f46c` and `1abbd64`.

4 Conclusion

In this audit, we have analyzed the design and implementation of the PRINT3R protocol, which is an innovative perpetual futures protocol designed to create permissionless trading markets. By addressing significant scaling issues found in existing platforms, PRINT3R leverages unique innovations, notably the use of Chainlink functions to securely perform essential computations off-chain. This allows users to trade a wide range of assets, from top 100 crypto tokens to the latest memecoin trends. Additionally, anyone can easily create a trading pool, similar to launching liquidity pools on current DEXs. Participants can also help secure the network and earn financial rewards by executing transactions, liquidating under-collateralized positions, or auto-deleveraging overheated markets. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

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-1041: Use of Redundant Code. https://cwe.mitre.org/data/definitions/1041.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.