SMART CONTRACT AUDIT REPORT

for

Pika Protocol

Prepared By: Yiqun Chen

PeckShield

December 24, 2021

Document Properties

Client	Pika Protocol
Title	Smart Contract Audit Report
Target	PikaPerpV2
Version	1.0
Author	Xuxian Jiang
Auditors	Xiaotao Wu, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	December 24, 2021	Xuxian Jiang	Final Release
1.0-rc	December 24, 2021	Xuxian Jiang	Release Candidate #1

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Yiqun Chen
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 Pika 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 Pika

Pika protocol is a decentralized perpetual swap exchange on Ethereum layer 2 with a number of features, including high leverage, deep liquidity, numerous assets for trade, as well as user-friendly composability with other DeFi systems. The protocol has a utility token, i.e., PIKA, which is designed to facilitate and incentivize the decentralized governance of the protocol. A portion of the protocol fees are distributed to PIKA stakers as reward. The protocol fees come from the liquidation reward and interest fees. Another portion of collected protocol fees will also be used to purchase PIKA token periodically, which will be then distributed to PIKA stakers. The basic information of audited contracts is as follows:

Table 1.1: Basic Information of PikaPerpV2

Item	Description
Name	Pika Protocol
Website	https://www.pikaprotocol.com
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	December 24, 2021

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

- https://github.com/PikaProtocol/PikaPerpV2.git (4a0965f)

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

- https://github.com/PikaProtocol/PikaPerpV2.git (b70c4fc)

1.2 About PeckShield

PeckShield Inc. [12] 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 [11]:

- 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 accordingly classified into four categories, 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) [10], 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. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

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 `Pika` protocol smart contracts. 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	1	■
Medium	1	■
Low	3	■ ■ ■
Informational	1	■
Total	6	

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 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 1 high-severity vulnerability, 1 medium-severity vulnerability, 3 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Consistency in maxExposure Calculation	Business Logic	Fixed
PVE-002	Low	Improved Function Argument Validation in PikaPerpV2	Coding Practices	Fixed
PVE-003	High	Potential Reentrancy Risk in closePositionWithId()	Time and State	Fixed
PVE-004	Low	Gas Optimization in Reward Distribution	Coding Practices	Fixed
PVE-005	Medium	Trust Issue of Admin Keys	Security Features	Mitigated
PVE-006	Informational	Removal Of Unused State/Code	Coding Practices	Fixed

Beside the identified issues, we emphasize that for any user-facing applications and services, 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 should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Consistency in maxExposure Calculation

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: PikaPerpV2
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

The Pika protocol has defined a number of protocol-wide risk parameters. For example, minMargin specifies the minimum margin amount accepted by the protocol. maxPositionMargin indicates the maximum margin amount for the active position. While reviewing the exposureMultiplier parameter, we notice the inconsistency of its enforcement in current implementation.

To elaborate, we show below the openPosition() routine. As the name indicates, the routine is designed to open a new position. Our analysis shows the new position price is calculated using uint256(vault.balance).mul(uint256(product.weight)).div(uint256(totalWeight)) (line 323) as the maxExposure of the new position. However, The closePosition() counterpart computes the price (line 420) with the uint256(vault.balance).mul(uint256(product.weight)).mul(exposureMultiplier).div(uint256(totalWeight)).div(10**4) as the maxExposure of the closed position. The inconsistency is suggested to be resolved.

```
297     // Opens position with margin = msg.value
298     function openPosition(
299         uint256 productId,
300         uint256 margin,
301         bool isLong,
302         uint256 leverage
303     ) external payable nonReentrant returns(uint256 positionId) {
304         // Check params
305         require(margin >= minMargin, "!margin");
306         require(leverage >= 1 * BASE, "!leverage");
307 
```

```
        // Check product
        Product storage product = products[productId];
        require(product.isActive, "!product-active");
        require(leverage <= uint256(product.maxLeverage), "!max-leverage");

        // Transfer margin plus fee
        uint256 tradeFee = _getTradeFee(margin, leverage, uint256(product.fee));
        IERC20(token).uniTransferFromSenderToThis((margin.add(tradeFee)).mul(tokenBase).
            div(BASE));
        pendingProtocolReward = pendingProtocolReward.add(tradeFee.mul(
            protocolRewardRatio).div(10**4));
        pendingPikaReward = pendingPikaReward.add(tradeFee.mul(pikaRewardRatio).div
            (10**4));
        pendingVaultReward = pendingVaultReward.add(tradeFee.mul(10**4 -
            protocolRewardRatio - pikaRewardRatio).div(10**4));

        // Check exposure
        uint256 amount = margin.mul(leverage).div(BASE);
        uint256 price = _calculatePrice(product.feed, isLong, product.openInterestLong,
            product.openInterestShort, uint256(vault.balance).mul(uint256(product.weight
                )).div(uint256(totalWeight)),
            uint256(product.reserve), amount);
        ...
    }
```

Listing 3.1: `PikaPerpV2::openPosition()`

Recommendation Resolve the above inconsistency in the `maxExposure` computation.

Status This issue has been fixed in the following commit: `b70c4fc`.

3.2 Improved Function Argument Validation in PikaPerpV2

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Coding Practices [7]
- CWE subcategory: CWE-628 [3]

Description

In the `PikaPerpV2` contract, the `addProduct()` function is used to add a new product with the associated parameters. To elaborate, we show below the related code snippet.

```
    function addProduct(uint256 productId, Product memory _product) external onlyOwner {

        Product memory product = products[productId];
        require(product.maxLeverage == 0, "!product-exists");
```

```
        require(_product.maxLeverage > 0, "!max-leverage");
        require(_product.feed != address(0), "!feed");
        require(_product.liquidationThreshold > 0, "!liquidationThreshold");

        products[productId] = Product({
        feed: _product.feed,
        maxLeverage: _product.maxLeverage,
        fee: _product.fee,
        isActive: true,
        openInterestLong: 0,
        openInterestShort: 0,
        interest: _product.interest,
        liquidationThreshold: _product.liquidationThreshold,
        liquidationBounty: _product.liquidationBounty,
        minPriceChange: _product.minPriceChange,
        weight: _product.weight,
        reserve: _product.reserve
        });
        totalWeight += _product.weight;

        emit ProductAdded(productId, products[productId]);

    }
```

Listing 3.2: `PikaPerpV2::addProduct()`

We notice that this function does not validate the given `productId`. If `productId` is equal to 0, the opened position may not be liquidated. Therefore, we suggest to add the following requirement, i.e., `require(productId>0)`.

In the same vein, there are number of other functions that can be similarly improved, including `updateProduct()`, `updateVault()`, `setProtocolRewardRatio()`, and `setPikaRewardRatio()`.

Recommendation Improve the above functions with additional validations.

Status This issue has been fixed in the following commit: `b70c4fc`.

3.3 Potential Reentrancy Risk in closePositionWithId()

- ID: PVE-003
- Severity: High
- Likelihood: High
- Impact:High
- Target: `PikaPerpV2`
- Category: Time and State [9]
- CWE subcategory: CWE-682 [4]

Description

A common coding best practice in Solidity is the adherence of `checks-effects-interactions` principle. This principle is effective in mitigating a serious attack vector known as `re-entrancy`. Via this particular attack vector, a malicious contract can be reentering a vulnerable contract in a nested manner. Specifically, it first calls a function in the vulnerable contract, but before the first instance of the function call is finished, second call can be arranged to re-enter the vulnerable contract by invoking functions that should only be executed once. This attack was part of several most prominent hacks in Ethereum history, including the `DAO` [14] exploit, and the recent `Uniswap/Lendf.Me` hack [13].

We notice there are several occasions the `checks-effects-interactions` principle is violated. Using the `PikaPerpV2` as an example, the `closePositionWithId()` function (see the code snippet below) is provided to externally call a token contract to transfer assets. However, the invocation of an external contract requires extra care in avoiding the above `re-entrancy`.

Apparently, the interaction with the external contract (line 437) starts before effecting the update on internal states (e.g., lines 453-457), hence violating the principle. In this particular case, if the external contract has certain hidden logic that may be capable of launching `re-entrancy` via the very same `closePositionWithId()` function.

```
400     // Closes position from Position with id = positionId
401     function closePositionWithId(
402         uint256 positionId,
403         uint256 margin
404     ) public {
405         // Check params
406         require(margin >= minMargin, "!margin");
407
408         // Check position
409         Position storage position = positions[positionId];
410         require(msg.sender == position.owner, "!owner");
411
412         // Check product
413         Product storage product = products[uint256(position.productId)];
414
415         bool isFullClose;
416         if (margin >= uint256(position.margin)) {
417             margin = uint256(position.margin);
```

```
            isFullClose = true;
        }
        uint256 maxExposure = uint256(vault.balance).mul(uint256(product.weight)).mul(
            exposureMultiplier).div(uint256(totalWeight)).div(10**4);
        uint256 price = _calculatePrice(product.feed, !position.isLong, product.
            openInterestLong, product.openInterestShort,
            maxExposure, uint256(product.reserve), margin * position.leverage / 10**8);

        bool isLiquidatable;
        int256 pnl = _getPnl(position, margin, price);
        if (pnl < 0 && uint256(-1 * pnl) >= margin.mul(uint256(product.
            liquidationThreshold)).div(10**4)) {
            margin = uint256(position.margin);
            pnl = -1 * int256(uint256(position.margin));
            isLiquidatable = true;
        } else {
            // front running protection: if oracle price up change is smaller than
                threshold and minProfitTime has not passed, the pnl is be set to 0
            if (pnl > 0 && !_canTakeProfit(position, IOracle(oracle).getPrice(product.
                feed), product.minPriceChange)) {
                pnl = 0;
            }
        }

        uint256 totalFee = _updateVaultAndGetFee(pnl, position, margin, uint256(product.
            fee), uint256(product.interest));
        _updateOpenInterest(uint256(position.productId), margin.mul(uint256(position.
            leverage)).div(BASE), position.isLong, false);

        emit ClosePosition(
            positionId,
            position.owner,
            uint256(position.productId),
            price,
            uint256(position.price),
            margin,
            uint256(position.leverage),
            totalFee,
            pnl,
            isLiquidatable
        );

        if (isFullClose) {
            delete positions[positionId];
        } else {
            position.margin -= uint64(margin);
        }
    }
```

Listing 3.3: `PikaPerpV2::closePositionWithId()`

Recommendation Apply necessary reentrancy prevention by making use of the common `nonReentrant` modifier.

Status This issue has been fixed in the following commit: `b70c4fc`.

3.4 Gas Optimization in Reward Distribution

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `PikaPerpV2`
- Category: Coding Practices [7]
- CWE subcategory: CWE-628 [3]

Description

The `Pika` protocol makes use of the protocol fees collected from the liquidation reward and interest fees to facilitate and incentivize the decentralized governance of the protocol. While reviewing current reward distribution logic, we observe gas optimization opportunities.

To elaborate, we show below an example `distributePikaReward()` function. As the name indicates, this function is used to distribute rewards to `PIKA` stakers. It comes to our attention that this function can be revised to avoid repeated computations (lines 645, 646, and 648) as well as storage loads from the same location (lines 642 and 643).

```
640     function distributePikaReward() external returns(uint256) {
641         require(msg.sender == pikaRewardDistributor, "!distributor");
642         uint256 _pendingPikaReward = pendingPikaReward;
643         if (pendingPikaReward > 0) {
644             pendingPikaReward = 0;
645             IERC20(token).uniTransfer(pikaRewardDistributor, _pendingPikaReward.mul(
                    tokenBase).div(BASE));
646             emit PikaRewardDistributed(pikaRewardDistributor, _pendingPikaReward.mul(
                    tokenBase).div(BASE));
647         }
648         return _pendingPikaReward.mul(tokenBase).div(BASE);
649     }
```

Listing 3.4: `PikaPerpV2::distributePikaReward()`

An example revision is shown as follows. Note that two other functions `distributeProtocolReward()` and `distributeVaultReward()` can be similarly improved.

```
    function distributePikaReward() external returns(uint256) {
        require(msg.sender == pikaRewardDistributor, "!distributor");
        uint256 _pendingPikaReward = _pendingPikaReward.mul(tokenBase).div(BASE);
        if (pendingPikaReward > 0) {
            pendingPikaReward = 0;
            IERC20(token).uniTransfer(pikaRewardDistributor, pendingPikaReward);
```

```
            emit PikaRewardDistributed(pikaRewardDistributor, pendingPikaReward);
        }
        return pendingPikaReward;
    }
```

Listing 3.5: `PikaPerpV2::distributePikaReward()`

Recommendation Avoid repeated computation and storage loads to save gas consumption.

Status This issue has been fixed in the following commit: `b70c4fc`.

3.5 Trust Issue of Admin Keys

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [6]
- CWE subcategory: CWE-287 [1]

Description

In the `Pika` 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., product creation, parameter setting, and reward configuration). It also has the privilege to regulate or govern the flow of assets for margining among the involved components.

With great privilege comes great responsibility. Our analysis shows that the `owner` account is indeed privileged. In the following, we show representative privileged operations in the `Pika` protocol.

```
    function setDistributors(
        address _protocolRewardDistributor,
        address _pikaRewardDistributor,
        address _vaultRewardDistributor,
        address _vaultTokenReward
    ) external onlyOwner {
        protocolRewardDistributor = _protocolRewardDistributor;
        pikaRewardDistributor = _pikaRewardDistributor;
        vaultRewardDistributor = _vaultRewardDistributor;
        vaultTokenReward = _vaultTokenReward;
    }

    function setProtocolRewardRatio(uint256 _protocolRewardRatio) external onlyOwner {
        require(_protocolRewardRatio <= 10000, "!too-much");
        protocolRewardRatio = _protocolRewardRatio;
        emit ProtocolRewardRatioUpdated(protocolRewardRatio);
    }
```

```
    function setPikaRewardRatio(uint256 _pikaRewardRatio) external onlyOwner {
        require(_pikaRewardRatio <= 10000, "!too-much");
        pikaRewardRatio = _pikaRewardRatio;
        emit PikaRewardRatioUpdated(pikaRewardRatio);
    }
```

Listing 3.6: Various `Setters` in `PikaPerpV2`

We emphasize that current privilege assignment is necessary and required for proper protocol operation. However, if the privileged `owner` account is a plain EOA account, this may be worrisome and pose counter-party risk to the protocol users. Note that a multi-sig account could greatly alleviate this concern, though it is still far from perfect. Specifically, a better approach is to eliminate the administration key concern by transferring the role to a community-governed DAO. In the meantime, a timelock-based mechanism can also be considered as mitigation.

Moreover, it should be noted that current contracts have the support of being deployed behind a proxy. And there is a need to properly manage the proxy-admin privileges as they fall in this trust issue as well.

Recommendation Promptly transfer the `owner` privilege to the intended `DAO`-like governance contract.

Status This issue has been confirmed and partially mitigated with a new timelock account. The team further confirms the plan to eventually move the admin key under DAO control.

3.6 Redundant State/Code Removal

- ID: PVE-006
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Multiple Contracts`
- Category: Coding Practices [7]
- CWE subcategory: CWE-563 [2]

Description

The `Pika` protocol makes good use of a number of reference contracts, such as `ERC20`, `SafeBEP20`, `SafeMath`, and `Address`, to facilitate its code implementation and organization. For example, the `PikaStaking` smart contract has so far imported at least five reference contracts. However, we observe the inclusion of certain unused code or the presence of unnecessary redundancies that can be safely removed.

For example, if we examine closely the `PikaStaking` contract, there is a storage state `rewardTokenDecimal` that is defined, but not used. In addition, the `PikaPerpV2` contract defines a state `nextStakeId`, which is also not used either.

```
contract PikaStaking is ReentrancyGuard, Pausable {

    using SafeERC20 for IERC20;
    using Address for address payable;

    address public owner;
    address public pikaPerp;
    address public rewardToken;
    address public stakingToken;
    uint256 public rewardTokenDecimal;
    ...
    }
```

Listing 3.7: The `PikaStaking` Contract

Recommendation Consider the removal of the redundant state (or code) with a simplified, consistent implementation.

Status This issue has been fixed in the following commit: `b70c4fc`.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Pika` protocol, which is a decentralized perpetual swap exchange on Ethereum layer 2 with a number of features, including high leverage, deep liquidity, numerous assets for trade, as well as user-friendly composability with the entire DeFi system. 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-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[2] MITRE. CWE-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.html.

[3] MITRE. CWE-628: Function Call with Incorrectly Specified Arguments. https://cwe.mitre.org/data/definitions/628.html.

[4] MITRE. CWE-682: Incorrect Calculation. https://cwe.mitre.org/data/definitions/682.html.

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

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

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

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

[9] MITRE. CWE CATEGORY: Error Conditions, Return Values, Status Codes. https://cwe.mitre.org/data/definitions/389.html.

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

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

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

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

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