SMART CONTRACT AUDIT REPORT

for

ZetaEarn Protocol

Prepared By: Xiaomi Huang

PeckShield

January 30, 2024

Document Properties

Client	ZetaEarn
Title	Smart Contract Audit Report
Target	ZetaEarn
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	January 30, 2024	Xuxian Jiang	Final Release
1.0-rc	January 28, 2024	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 ZetaEarn 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 could potentially be improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About ZetaEarn

ZetaEarn serves as a liquid staking protocol for the ZetaChain PoS (Proof of Stake) blockchain. It provides users with the ability to stake their ERC20 ZETA tokens and instantly receive a representation of their stake in the form of stZETA tokens, eliminating the need to maintain staking infrastructure. Users will earn staking rewards and retain control over their stZETA tokens. ZETA tokens will be delegated amongst validators who have successfully registered and been approved within the ZetaEarn on ZetaChain protocol. The basic information of audited contracts is as follows:

Table 1.1: Basic Information of ZetaEarn

Item	Description
Name	ZetaEarn
Website	https://zetaearn.com/
Type	Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	January 30, 2024

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

- https://github.com/zetaearn/zetaearn_contract.git (ca6d6ee)

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

- https://github.com/zetaearn/zetaearn_contract.git (206bc0b)

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 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) [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. 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 `ZetaEarn` 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	0	
Medium	1	■
Low	4	■ ■ ■ ■
Informational	0	
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 may involve 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 medium-severity vulnerability and 4 low-severity vulnerabilities.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Suggested Adherence of The Checks-Effects-Interactions Pattern	Time and State	Resolved
PVE-002	Low	Improved Initialization Logic in StZETA	Coding Practices	Resolved
PVE-003	Low	Possible DoS in delegate() With EJECTED/UNSTAKED operators	Business Logic	Resolved
PVE-003	Low	Improved Parameter Validation in Registry	Coding Practices	Resolved
PVE-004	Medium	Trust Issue of Admin Keys	Security Features	Resolved

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 Suggested Adherence of The Checks-Effects-Interactions Pattern

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `ValidatorOperator`
- Category: Time and State [8]
- CWE subcategory: CWE-663 [3]

Description

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

We notice there are occasions where the `checks-effects-interactions` principle is violated. Using the `ValidatorOperator` as an example, the `unstakeClaimTokens()` function (see the code snippet below) is provided to externally call a contract to execute unstake operations. However, the invocation of an external contract requires extra care in avoiding the above `re-entrancy`. For example, the interaction with the external contract (line 262) start before effecting the update on internal state (line 264), hence violating the principle. In this particular case, if the external contract has certain hidden logic that may be capable of launching `re-entrancy` via the same entry function.

```
255     /// @notice unstake claim
256     /// @param unbondNonce unbond nonce
257     /// @return amount amount
258     function unstakeClaimTokens(uint256 unbondNonce) external override isStZETA returns(
            uint256) {
```

```
        // get user unbond info
        DelegatorUnbond memory unbond = unbonds_new[msg.sender][unbondNonce];
        // according to unbond info to unstake claim
        uint256 amount = _unstakeClaimTokens(unbond);
        // delete unbonds_new[msg.sender][unbondNonce];
        delete unbonds_new[msg.sender][unbondNonce];

        return amount;
    }
```

Listing 3.1: `ValidatorOperator::unstakeClaimTokens()`

Recommendation Apply necessary reentrancy prevention by following the `checks-effects-interactions` principle and/or utilizing the necessary `nonReentrant` modifier to block possible `re-entrancy`.

Status This issue has been fixed by the following commit: `206bc0b`.

3.2 Improved Initialization Logic in StZETA

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `StZETA`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [1]

Description

The `ZetaEarn` protocol provides users with the ability to stake their `ERC20 ZETA` tokens and instantly receive a representation of their stake in the form of `stZETA` tokens. While examining the `StZETA`'s' construction and initialization logic, we notice current implementation can be improved.

In the following, we use the `StZETA` contract as an example and shows its initialization routine. It comes to our attention that the `initialize()` routine directly calls the parent contracts' initialization routines in the following forms: `__AccessControl_init_unchained()` (line 126), `__Pausable_init_unchained()` (line 127), and `__ERC20_init_unchained("Staked ZETA", "stZETA")` (line 128). Note they may be instead revised as `__AccessControl_init()`, `__Pausable_init()`, and `__ERC20_init("Staked ZETA", "stZETA")` respectively so that the coding practice follows better the call convention when initializining an upgradeable proxy contract.

```
118     function initialize(
119         address _dao,
120         address _insurance,
121         address _oracle,
122         address _nodeOperatorRegistry,
```

```
        address _unStZETA,
        uint256 _currentEpoch
    ) external override initializer {
        __AccessControl_init_unchained();
        __Pausable_init_unchained();
        __ERC20_init_unchained("Staked ZETA", "stZETA");

        // Set roles
        _grantRole(DEFAULT_ADMIN_ROLE, msg.sender);
        _grantRole(DAO, _dao);
        _grantRole(ORACLE_ROLE, _oracle);
        _grantRole(PAUSE_ROLE, msg.sender);
        _grantRole(UNPAUSE_ROLE, _dao);

        // Set addresses
        dao = _dao;
        insurance = _insurance;
        oracle = _oracle;
        nodeOperatorRegistry = INodeOperatorRegistry(_nodeOperatorRegistry);
        unStZETA = IUnStZETA(_unStZETA);

        // Set fee distribution
        entityFees = FeeDistribution(25, 50, 25);
        // Set threshold
        submitThreshold = 10 ** 10;
        // Set maximum submit threshold
        submitMaxThreshold = 10 ** 34;
        // Set protocol fee
        protocolFee = 10;
        // Set delegation lower bound
        delegationLowerBound = 0;
        // Set current epoch
        currentEpoch = _currentEpoch;
        // Set epoch delay
        epochDelay = 5;
        // version number
        version = "1.0.3";
    }
```

Listing 3.2: `StZETA::initialize()`

Recommendation Improve the above-mentioned initialization routine. Note the improvement is also applicable to other initialization routines in `UnStZETA` and `NodeOperatorRegistry`.

Status This issue has been fixed by the following commit: `206bc0b`.

3.3 Possible DoS in delegate() With EJECTED/UNSTAKED operators

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: StZETA
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

The ZetaEarn protocol has the core StZETA contract that allows users to directly stake ZETA tokens. The staked tokens will be later delegated to authorized NodeOperators. While examining the related delegate logic, we notice current implementation need to be improved.

Specifically, we show below the implementation of the related routine, i.e., delegate(). It validates the amount of available staked tokens, retrieves the stake information of all active node operators, and calls the actual delegate operation upon active node operations. Our analysis shows that the retrieval of the stake information of all active node operators may be reverted, hence causing unexpected denial-of-service. The reason is that the invoked helper getValidatorsDelegationAmount() is coded to revert the call if the related node operator is in EJECTED or UNSTAKED states.

```
    function delegate() external override whenNotPaused nonReentrant onlyRole(
        ORACLE_ROLE) {
        // Store totalBuffered and reservedFunds temporarily
        uint256 ltotalBuffered = totalBuffered;
        uint256 lreservedFunds = reservedFunds;
        // Check if totalBuffered is greater than delegationLowerBound + reservedFunds
        _require(
            ltotalBuffered > delegationLowerBound + lreservedFunds,
            "Amount lower than minimum"
        );
        // Check if the balance of the current contract is greater than totalBuffered
        _require(
            address(this).balance >= ltotalBuffered,
            "Balance lower than Buffered"
        );

        // The total amount to delegate is equal to totalBuffered - reservedFunds
        uint256 amountToDelegate = ltotalBuffered - lreservedFunds;

        // Get the stake information of all active node operators
        (
            INodeOperatorRegistry.ValidatorData[] memory validators,
            ,
            uint256[] memory operatorRatios,
            uint256 totalRatio,
```

```
        ) = nodeOperatorRegistry.getValidatorsDelegationAmount();
        // Get the length of validators
        uint256 validatorsOperatorLength = validators.length;
        // Remainder, the remaining amount of money
        uint256 remainder;
        // Actual delegated amount
        uint256 amountDelegated;
        // Temporary variable for the amount delegated by the validator used
        uint256 validatorAmountDelegated;
        // Iterate through all validators
        for (uint256 i = 0; i < validatorsOperatorLength; i++) {
            // Get the current validator's ratio
            uint256 operatorRatio = operatorRatios[i];
            // Calculate the delegated amount for the current validator
            validatorAmountDelegated = (amountToDelegate * operatorRatio) / totalRatio;
            // If the delegated amount for the current validator is 0, skip
            if (validatorAmountDelegated == 0) continue;
            // Delegate the tokens
            IValidatorOperator(validators[i].operatorAddress).delegate{value:
                validatorAmountDelegated}();
            // Update the actual delegated amount
            amountDelegated += validatorAmountDelegated;
        }
        // Remainder, the remaining amount of money
        remainder = amountToDelegate - amountDelegated;
        // update totalBuffered
        totalBuffered = remainder + lreservedFunds;
        // Emit the event
        emit DelegateEvent(amountDelegated, remainder);
    }
```

Listing 3.3: `StZETA::delegate()`

Recommendation Revisit the above logic to avoid the unnecessary revert when the called node operator is the states of `EJECTED` or `UNSTAKED` .

Status This issue has been resolved as the team confirms it is part of intended design.

3.4 Improved Parameter Validation in Registry

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `NodeOperatorRegistry`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [1]

Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The `ZetaEarn` protocol is no exception. Specifically, if we examine the `NodeOperatorRegistry` contract, it provides the setters to register protocol-wide handlers, such as `operatorAddress` and associated `rewardAddress`. In the following, we show the corresponding routines that allow for their changes.

```
    function setRewardAddress(address _newRewardAddress)
        external
        override
        whenNotPaused {
        // only old reward address can call this function
        require(_newRewardAddress != msg.sender, "Invalid reward address");
        address operatorAddress = validatorRewardAddressToOperatorAddress[msg.sender];
        address oldRewardAddress = validatorOperatorAddressToRewardAddress[
            operatorAddress];
        require(oldRewardAddress == msg.sender, "Unauthorized");
        require(_newRewardAddress != address(0), "Invalid reward address");

        validatorOperatorAddressToRewardAddress[operatorAddress] = _newRewardAddress;
        validatorRewardAddressToOperatorAddress[_newRewardAddress] = operatorAddress;
        delete validatorRewardAddressToOperatorAddress[msg.sender];

        emit SetRewardAddress(operatorAddress, oldRewardAddress, _newRewardAddress);
    }
```

Listing 3.4: NodeOperatorRegistry::setRewardAddress()

These parameters define various aspects of the protocol operation and maintenance and need to exercise extra care when configuring or updating them. Our analysis shows the update logic on these parameters can be improved by applying more rigorous sanity checks. For example, we can add the following requirement to the above routine, i.e., `require(validatorRewardAddressToOperatorAddress[_newRewardAddress] == address(0))`.

Recommendation Validate any changes regarding these system-wide parameters to ensure the changes are expected and fall in an appropriate mapping or range.

Status This issue has been fixed by the following commit: `206bc0b`.

3.5 Trust Issue of Admin Keys

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Multiple Contracts
- Category: Security Features [5]
- CWE subcategory: CWE-287 [2]

Description

The ZetaEarn protocol has a privileged account (with the DEFAULT_ADMIN_ROLE role) that plays a critical responsibility in governing and regulating the protocol-wide operations (e.g., manage node operators, adjust fees, and configure protocol-wide risk parameters). It also has the privilege to control or govern the flow of assets among various protocol components. In the following, we examine the privileged account and related privileged accesses in current contracts.

```
    function setFees(uint8 _daoFee, uint8 _operatorsFee, uint8 _insuranceFee) external
        override onlyRole(DAO) {
        // Check if the sum of fees is equal to 100
        _require(
            _daoFee + _operatorsFee + _insuranceFee == 100,
            "sum(fee)!=100"
        );
        entityFees.dao = _daoFee;
        entityFees.operators = _operatorsFee;
        entityFees.insurance = _insuranceFee;

        emit SetFees(_daoFee, _operatorsFee, _insuranceFee);
    }

    /// @notice Allow setting new DaoAddress.
    /// @param _newDaoAddress new DaoAddress.
    function setDaoAddress(address _newDaoAddress) external override onlyRole(DAO) {
        address oldDAOAddress = dao;
        dao = _newDaoAddress;
        emit SetDaoAddress(oldDAOAddress, _newDaoAddress);
    }

    /// @notice Allow setting new OracleAddress.
    /// @notice Only the DAO can call this function.
    /// @param _newOracleAddress new OracleAddress.
    function setOracleAddress(address _newOracleAddress) external override onlyRole(DAO)
         {
        address oldOracleAddress = oracle;
        oracle = _newOracleAddress;
        emit SetOracleAddress(oldOracleAddress, _newOracleAddress);
    }
```

```
    /// @notice Set a new protocol fee.
    /// @param _newProtocolFee - The new protocol fee, in percentage.
    function setProtocolFee(uint8 _newProtocolFee)
        external
        override
        onlyRole(DAO) {
        // Check if the protocol fee is greater than 0 and less than or equal to 100
        _require(
            _newProtocolFee > 0 && _newProtocolFee <= 100,
            "Invalid protocol fee"
        );
        uint8 oldProtocolFee = protocolFee;
        protocolFee = _newProtocolFee;

        emit SetProtocolFee(oldProtocolFee, _newProtocolFee);
    }

    /// @notice Allow setting new InsuranceAddress.
    /// @notice Only the DAO can call this function.
    /// @param _newInsuranceAddress new InsuranceAddress.
    function setInsuranceAddress(address _newInsuranceAddress)
        external
        override
        onlyRole(DAO) {
        insurance = _newInsuranceAddress;
        emit SetInsuranceAddress(_newInsuranceAddress);
    }
```

Listing 3.5: Example Privileged Operations in `StZETA`

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 admin role to the intended `DAO`-like governance contract. And activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status This issue has been resolved as the admin role will be controlled through a CA multi-signature wallet (similar to `Gnosis Safe`).

4 Conclusion

In this audit, we have analyzed the design and implementation of the ZetaEarn protocol, which serves as a liquid staking protocol for the ZetaChain PoS (Proof of Stake) blockchain. It provides users with the ability to stake their ERC20 ZETA tokens and instantly receive a representation of their stake in the form of stZETA tokens, eliminating the need to maintain staking infrastructure. Users will earn staking rewards and retain control over their stZETA tokens. ZETA tokens will be delegated amongst validators who have successfully registered and been approved within the ZetaEarn on ZetaChain protocol. 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-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

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

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

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

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

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

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

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

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

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

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

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

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