SMART CONTRACT AUDIT REPORT

PacNetwork Smart Contract

SEPTEMBER 2025

Contents

1. EXECUTIVE SUMMARY **4**
1.1 Methodology 4

2. FINDINGS OVERVIEW **7**
2.1 Project Info And Contract Address 7
2.2 Summary 7
2.3 Key Findings 8

3. DETAILED DESCRIPTION OF FINDINGS **9**
3.1 Emergency rescue function bypasses the pause mechanism 9
3.2 computeVaultAddress Lacks Salt Uniqueness Check, Allowing Silent State Corruption . . 10
3.3 Incorrect Return Value in Loop 12
3.4 txId Validation Lacks Cryptographic Binding to Action Parameters 14
3.5 Fund Loss Risk in setReserve due to Balance Overwriting 15
3.6 Commented non-zero amount requirement not enforced 16
3.7 Missing Zero-Address Validation for Input Array Elements 18
3.8 mintReward Access Control Mismatches Documentation 20
3.9 Typo in AddressFactory Error Name 22
3.10 Incorrect Comment for pacUSDHash 23
3.11 The comments are inconsistent with the contract 24

4. CONCLUSION **25**

5. APPENDIX **26**
5.1 Basic Coding Assessment 26
5.1.1 Apply Verification Control 26
5.1.2 Authorization Access Control 26
5.1.3 Forged Transfer Vulnerability 26
5.1.4 Transaction Rollback Attack 27
5.1.5 Transaction Block Stuffing Attack 27
5.1.6 Soft Fail Attack Assessment 27
5.1.7 Hard Fail Attack Assessment 28
5.1.8 Abnormal Memo Assessment 28
5.1.9 Abnormal Resource Consumption 28
5.1.10 Random Number Security 29
5.2 Advanced Code Scrutiny 29
5.2.1 Cryptography Security 29

5.2.2 Account Permission Control 29
5.2.3 Malicious Code Behavior 30
5.2.4 Sensitive Information Disclosure 30
5.2.5 System API 30

6. DISCLAIMER **31**

7. REFERENCES **32**

8. About Exvul Security **33**

1. EXECUTIVE SUMMARY

ExVul Web3 Security was engaged by **PacNetwork** to review smart contract implementation. The assessment was conducted in accordance with our systematic approach to evaluate potential security issues based upon customer requirement. The report provides detailed recommendations to resolve the issue and provide additional suggestions or recommendations for improvement.

The outcome of the assessment outlined in chapter 3 provides the system's owners a full description of the vulnerabilities identified, the associated risk rating for each vulnerability, and detailed recommendations that will resolve the underlying technical issue.

1.1 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [10] which is the gold standard in risk assessment using the following risk models:

- **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**: determine the overall criticality of the risk.

Likelihood can be: High, Medium and Low and impact are categorized into: High, Medium, Low, Informational. Severity is determined by likelihood and impact and can be classified into five categories accordingly: Critical, High, Medium, Low, Informational shown in table 1.1.

Likelihood \ IMPACT	Informational	Low	Medium	High
High	INFO	MEDIUM	HIGH	CRITICAL
Medium	INFO	LOW	MEDIUM	HIGH
Low	INFO	LOW	LOW	MEDIUM

Table 1.1 Overall Risk Severity

To evaluate the risk, we will be going through a list of items, and each would be labelled with a severity category. The audit was performed with a systematic approach guided by a comprehensive assessment list carefully designed to identify known and impactful security issues. If our tool or analysis does not identify any issue, the contract can be considered safe regarding the assessed item. For any discovered issue, we might further deploy contracts on our private test environment 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 1.2.

- **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.
- **Code and business security testing**: 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.

Category	Assessment Item
Basic Coding Assessment	• Apply Verification Control • Authorization Access Control • Forged Transfer Vulnerability • Forged Transfer Notification • Numeric Overflow • Transaction Rollback Attack • Transaction Block Stuffing Attack • Soft Fail Attack • Hard Fail Attack • Abnormal Memo • Abnormal Resource Consumption • Secure Random Number

Advanced Source Code Scrutiny	• Asset Security • Cryptography Security • Business Logic Review • Source Code Functional Verification • Account Authorization Control • Sensitive Information Disclosure • Circuit Breaker • Blacklist Control • System API Call Analysis • Contract Deployment Consistency Check • Abnormal Resource Consumption
Additional Recommendations	• Semantic Consistency Checks • Following Other Best Practices

Table 1.2: The Full List of Assessment Items

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [14], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development.

2. FINDINGS OVERVIEW

2.1 Project Info And Contract Address

Project Name	Audit Time	Language
PacNetwork	04/09/2025 - 18/09/2025	Solidity

Repository
https://github.com/PacNetwork/evm-contracts

Commit Hash
84090e345d12bc6c46722558a481d00c350e0b32

2.2 Summary

Severity	Found
CRITICAL	0
HIGH	1
MEDIUM	4
LOW	3
INFO	3

2.3 Key Findings

Severity	Findings Title	Status
HIGH	Emergency rescue function bypasses the pause mechanism	Fixed
MEDIUM	computeVaultAddress Lacks Salt Uniqueness Check, Allowing Silent State Corruption	Fixed
MEDIUM	Incorrect Return Value in Loop	Fixed
MEDIUM	txId Validation Lacks Cryptographic Binding to Action Parameters	Acknowledge
MEDIUM	Fund Loss Risk in setReserve due to Balance Overwriting	Fixed
LOW	Commented non-zero amount requirement not enforced	Fixed
LOW	Missing Zero-Address Validation for Input Array Elements	Fixed
LOW	mintReward Access Control Mismatches Documentation	Fixed
INFO	Typo in AddressFactory Error Name	Fixed
INFO	Incorrect Comment for pacUSDHash	Fixed
INFO	The comments are inconsistent with the contract	Fixed

Table 2.3: Key Audit Findings

3. DETAILED DESCRIPTION OF FINDINGS

3.1 Emergency rescue function bypasses the pause mechanism

SEVERITY: HIGH **STATUS:** Fixed

PATH:

`contracts/v1/pacusd/PacUSD.sol`

DESCRIPTION:

There is an emergency handling function in PacUSD.sol. rescueTokens allows RESCUER_ROLE to force all tokens in the contract in an emergency, but this operation is not subject to the Pause function.

IMPACT:

When the contract is in a suspended state, rescueTokens can still be called normally.

RECOMMENDATIONS:

Add whenNotPaused constraint.

3.2 computeVaultAddress Lacks Salt Uniqueness Check, Allowing Silent State Corruption

SEVERITY: MEDIUM **STATUS:** Fixed

PATH:

`contracts/v1/factory/AddressFactory.sol`

DESCRIPTION:

computeVaultAddress function allows the owner to add new Vault addresses. However, it does not check if an incoming salt in the salts array has already been used. If the owner provides a used salt, the line saltIndexMap[salt] = count; will silently overwrite the index for that salt, causing the saltIndexMap to become inconsistent with the vaultAddresses, vaultImplAddresses, and vaultSalts arrays.

IMPACT:

While CREATE2 prevents deploying to the same address twice, the internal state of AddressFactory is corrupted in a prior, successful transaction. Subsequent attempts to deploy using the duplicated salt via MMFVaultDeployFactory will fail because the factory will read the wrong index from the corrupted saltIndexMap, causing the deployment transaction to revert. This breaks the deployment process for that salt and requires redeploying AddressFactory to fix.

RECOMMENDATIONS:

```
// contracts/v1/factory/AddressFactory.sol

+ error SaltAlreadyExists(bytes32 salt);

  ...

  function computeVaultAddress(
      address vaultFactoryAddress,
      bytes32[] memory salts
  ) external {
      if (msg.sender != owner) revert NotOwner();
      if (salts.length == 0) revert InvaildParams();
```

```
      uint256 count = vaultAddresses.length;
      uint256 length = salts.length;
      for (uint i; i < length; ++i) {
          bytes32 salt = salts[i];
+         // Ensure the salt has not been used before to prevent state
  inconsistency.
+         if (saltIndexMap[salt] != 0 || (vaultSalts.length > 0 &&
  vaultSalts[0] == salt)) {
+             revert SaltAlreadyExists(salt);
+         }
          (address vaultAddress, address vaultImplAddress) =
  _computeAddress(
              salt,
              vaultHash,
              vaultFactoryAddress
          );
          vaultAddresses.push(vaultAddress);
          vaultImplAddresses.push(vaultImplAddress);
          vaultSalts.push(salt);
          saltIndexMap[salt] = count;
          ++count;
      }
  }
```

3.3 Incorrect Return Value in Loop

SEVERITY: MEDIUM **STATUS:** Fixed

PATH:

`contracts/v1/factory/MMFVaultDeployFactory.sol`

DESCRIPTION:

The deployContracts function in MMFVaultDeployFactory is designed to deploy multiple MMFVault proxy contracts within a for loop. However, the function is declared to return a single address. In each iteration of the loop, the mmfVaultProxy return variable is overwritten. As a result, the function only returns the address of the very last contract deployed in the loop.

IMPACT:

This leads to a loss of contract references, requiring off-chain event parsing to recover them, and can cause serious integration failures.

RECOMMENDATIONS:

```
// contracts/v1/factory/MMFVaultDeployFactory.sol

-   function deployContracts(
-       address[] memory mmfTokenAddresses,
-       address[] memory pricerAddresses,
-       bytes32[] memory mmfVaultSalts,
-       address admin,
-       address upgrader
-   ) external returns (address mmfVaultProxy) {
+   function deployContracts(
+       address[] memory mmfTokenAddresses,
+       address[] memory pricerAddresses,
+       bytes32[] memory mmfVaultSalts,
+       address admin,
+       address upgrader
+   ) external returns (address[] memory mmfVaultProxies) {
```

```
        if (msg.sender != owner) revert NotOwner();
        // ...
        uint256 length = mmfTokenAddresses.length;
+       mmfVaultProxies = new address[](length);
        // ...
        for (uint i; i < length; ++i) {
            // ...
-           mmfVaultProxy = Create2.deploy(
+           address mmfVaultProxy = Create2.deploy(
                0,
                salt,
                abi.encodePacked(
                    type(ERC1967Proxy).creationCode,
                    abi.encode(mmfVaultImpl, "")
                )
            );
            // ...
            // (Initialization logic)
            // ...
+           mmfVaultProxies[i] = mmfVaultProxy;
            emit ContractsDeployed(msg.sender, mmfVaultProxy);
        }
    }
```

3.4 txId Validation Lacks Cryptographic Binding to Action Parameters

SEVERITY: MEDIUM **STATUS:** Acknowledge

PATH:

`contracts/v1/pacusd/PacUSD.sol`

DESCRIPTION:

The critical flaw is that the execution step only validates the txId's existence and status, but does not cryptographically verify that the amount and to/from parameters passed to the function match the parameters for which the txId was originally approved.

```
function mintByTx(
    bytes32 txId,
    uint256 amount,
    address to
) external onlyMinter whenNotPaused nonReentrant notBlocklisted(to) {
    uint256 status = _mintTxs[txId];
    if (status != TX_STATE_AVAILABLE) revert TxIdInvalid(txId, status);
    if (to == address(0)) revert ZeroAddress();

    _mintTxs[txId] = TX_STATE_EXECUTED;
    _mint(to, amount);
    emit Mint(to, amount);
}
```

IMPACT:

An operational error or bug within a Minter contract (e.g., MMFVault) could cause it to call mintByTx or burnByTx with incorrect parameters. This could damage the token's backing, cause loss of funds, or disrupt the protocol's economic balance.

RECOMMENDATIONS:

It is recommended to enforce a cryptographic binding between the txId and its associated business parameters within the PacUSD contract.

3.5 Fund Loss Risk in setReserve due to Balance Overwriting

SEVERITY: MEDIUM **STATUS:** Fixed

PATH:

`contracts/v1/staking/PacUSDStaking.sol`

DESCRIPTION:

The setReserve function is designed to change the RESERVE address and transfer any pending rewards from the old reserve address to the new one. However, it does so by overwriting the new address's balance instead of adding to it.

IMPACT:

This implementation can lead to a direct loss of funds. If the newly designated reserve address is also an existing staker with a pending reward balance in the rewardBalances mapping, their original reward balance will be completely overwritten by the balance of the old reserve address.

RECOMMENDATIONS:

Change the assignment operation to an addition to ensure the old reserve's rewards are added to the new reserve's existing balance, preventing any loss of funds.

```
// contracts/v1/staking/PacUSDStaking.sol
function setReserve(address reserve) external onlyRole(RESERVE_SET_ROLE) {
    if (reserve == address(0)) revert ZeroAddress();
    uint256 accumulated = rewardBalances[RESERVE];
    rewardBalances[RESERVE] = 0;
    RESERVE = reserve;
-   rewardBalances[RESERVE] = accumulated;
+   rewardBalances[RESERVE] += accumulated;
    emit ReserveSet(reserve);
}
```

3.6 Commented non-zero amount requirement not enforced

SEVERITY: LOW **STATUS:** Fixed

PATH:

`contracts/v1/pacusd/PacUSD.sol`

DESCRIPTION:

In PacUSD.sol, the mintFee function requires amount to be non-zero, but the problem arises because the _mint() function mistakenly assumes that it will automatically check the amount.

```
/**
 * @notice Mints PacUSD tokens specifically for fee distribution purposes
 * @dev This function is restricted to accounts with the `onlyMinter`
    role, ensuring only authorized contracts (e.g., MMFVault)
 *      can mint tokens for fee-related use cases. It includes core
    security checks (zero-address prevention, pause state,
 *      blocklist validation) and follows non-reentrant design to avoid
    reentrancy attacks.
 * @param amount The quantity of PacUSD tokens to mint (must be non-zero,
    though zero check may be handled by underlying `_mint` logic)
 * @param to The recipient address that will receive the minted PacUSD
    tokens (fee receiver, typically a designated account)
 */
function mintFee(
    uint256 amount,
    address to
) external onlyMinter whenNotPaused notBlocklisted(to) nonReentrant {
    if (to == address(0)) revert ZeroAddress();
    _mint(to, amount);
    emit MintFee(to, amount);
}
```

IMPACT:

Developers or users may judge security when they see inconsistencies between opinions and reality.

RECOMMENDATIONS:

```
function mintFee(
    uint256 amount,
    address to
) external onlyMinter whenNotPaused notBlocklisted(to) nonReentrant {
    if (to == address(0)) revert ZeroAddress();
+   if (amount == 0) revert ZeroAmount();
    _mint(to, amount);
    emit MintFee(to, amount);
}
```

3.7 Missing Zero-Address Validation for Input Array Elements

SEVERITY: LOW **STATUS:** Fixed

PATH:

contracts/v1/factory/MMFVaultDeployFactory.sol, contracts/v1/factory/StakingDepl

DESCRIPTION:

Both MMFVaultDeployFactory and StakingDeployFactory accept address arrays as parameters (mmfTokenAddresses, pricerAddresses) but do not validate the elements of these arrays for zero-addresses before proceeding with contract deployment.

```
for (uint i; i < length; ++i) {
    bytes32 salt = mmfVaultSalts[i];
    uint256 index = addressFactory.saltIndexMap(salt);
    address mmfTokenAddress = mmfTokenAddresses[i];
    address pricerAddress = pricerAddresses[i];
    // --------------------
    // Deploy MMFVault implementation contract
    // --------------------
    address mmfVaultImpl = Create2.deploy(
        0, // Gas value (0 for default)
        salt, // Deployment salt for determinism
        type(MMFVault).creationCode // Bytecode of MMFVault
    );

    // ...
```

IMPACT:

Contract deployment via CREATE2 is one of the most gas-intensive operations in the EVM. According to its rules, even when a transaction reverts, the gas fees for all operations performed up to the point of failure are still consumed. This results in wasted gas for the user.

RECOMMENDATIONS:

Add a zero-address check inside the loop to validate the addresses.

3.8 mintReward Access Control Mismatches Documentation

SEVERITY: LOW **STATUS:** Fixed

PATH:

`contracts/v1/pacusd/PacUSD.sol`

DESCRIPTION:

The Natspec documentation for the mintReward function explicitly states that it is callable by an account with APPROVER_ROLE. However, the function's implementation uses the onlyMinter modifier, restricting access to minters.

```
/**
 * @notice Mints reward tokens to a specified address.
 * @dev Only callable by an account with APPROVER_ROLE when not paused,
    with reentrancy protection.
 *      Reverts if the recipient is the zero address or blocklisted.
    Emits a MintReward event.
 * @param amount The amount of tokens to mint as a reward.
 * @param to The address to receive the reward tokens.
 */
function mintReward(
    uint256 amount,
    address to
) external onlyMinter whenNotPaused notBlocklisted(to) nonReentrant {
    if (to == address(0)) revert ZeroAddress();
    _mint(to, amount);
    emit MintReward(to, amount);
}
```

IMPACT:

This inconsistency does not lead to direct fund loss but harms the clarity, maintainability, and integration of the contract.

RECOMMENDATIONS:

Align the documentation with the code's implementation.

3.9 Typo in AddressFactory Error Name

SEVERITY: INFO **STATUS:** Fixed

PATH:

`contracts/v1/factory/AddressFactory.sol`

DESCRIPTION:

The custom error InvaildParams is misspelled. It should be InvalidParams.

IMPACT:

This has no effect on contract logic but affects code professionalism and consistency.

RECOMMENDATIONS:

```
contract AddressFactory {
  error NotOwner();
- error InvaildParams();
+ error InvalidParams();
```

3.10 Incorrect Comment for pacUSDHash

SEVERITY: INFO **STATUS:** Fixed

PATH:

`contracts/v1/factory/AddressFactory.sol`

DESCRIPTION:

In the AddressFactory contract, the state variable pacUSDHash has a misleading natspec comment.

IMPACT:

It harms code clarity and maintainability.

RECOMMENDATIONS:

```
// contracts/v1/factory/AddressFactory.sol
-    // @dev Salt used for deterministic deployment of MMFVault
+    // @dev Hash of the PacUSD contract bytecode
     bytes32 pacUSDHash;
```

3.11 The comments are inconsistent with the contract

SEVERITY: INFO **STATUS:** Fixed

PATH:

`contracts/v1/pacusd/PacUSD.sol`

DESCRIPTION:

In PacUSD.sol, _authorizeUpgrade allows the owner to upgrade the contract to a new address, but the function can only be called if ADMIN_ROLE is described in the comment. Here ADMIN_ROLE refers to admin, and owner is upgrader.

```
/**
 * @notice Authorizes an upgrade to a new contract implementation.
 * @dev Implements UUPS upgradeability, only callable by an account with
    ADMIN_ROLE.
 * @param newImpl The address of the new contract implementation.
 */
function _authorizeUpgrade(address newImpl) internal override onlyOwner {
    if (newImpl == address(0)) revert ZeroAddress();
}
```

IMPACT:

May cause misunderstandings among developers.

RECOMMENDATIONS:

Modify according to project needs.

4. CONCLUSION

In this audit, we thoroughly analyzed **PacNetwork** smart contract implementation. The problems found are described and explained in detail in Section 3. The problems found in the audit have been communicated to the project leader. We therefore consider the audit result to be **PASSED**.

To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

5. APPENDIX

5.1 Basic Coding Assessment

5.1.1 Apply Verification Control

Description	The security of apply verification
Result	Not found
Severity	CRITICAL

5.1.2 Authorization Access Control

Description	Permission checks for external integral functions
Result	Not found
Severity	CRITICAL

5.1.3 Forged Transfer Vulnerability

Description	Assess whether there is a forged transfer notification vulnerability in the contract
Result	Not found
Severity	CRITICAL

5.1.4 Transaction Rollback Attack

Description	Assess whether there is transaction rollback attack vulnerability in the contract
Result	Not found
Severity	CRITICAL

5.1.5 Transaction Block Stuffing Attack

Description	Assess whether there is transaction blocking attack vulnerability
Result	Not found
Severity	CRITICAL

5.1.6 Soft Fail Attack Assessment

Description	Assess whether there is soft fail attack vulnerability
Result	Not found
Severity	CRITICAL

5.1.7 Hard Fail Attack Assessment

Description	Examine for hard fail attack vulnerability
Result	Not found
Severity	CRITICAL

5.1.8 Abnormal Memo Assessment

Description	Assess whether there is abnormal memo vulnerability in the contract
Result	Not found
Severity	CRITICAL

5.1.9 Abnormal Resource Consumption

Description	Examine whether abnormal resource consumption in contract processing
Result	Not found
Severity	CRITICAL

5.1.10 Random Number Security

Description	Examine whether the code uses insecure random number
Result	Not found
Severity	CRITICAL

5.2 Advanced Code Scrutiny

5.2.1 Cryptography Security

Description	Examine for weakness in cryptograph implementation
Result	Not found
Severity	HIGH

5.2.2 Account Permission Control

Description	Examine permission control issue in the contract
Result	Not found
Severity	MEDIUM

5.2.3 Malicious Code Behavior

Description	Examine whether sensitive behavior present in the code
Result	Not found
Severity	MEDIUM

5.2.4 Sensitive Information Disclosure

Description	Examine whether sensitive information disclosure issue present in the code
Result	Not found
Severity	MEDIUM

5.2.5 System API

Description	Examine whether system API application issue present in the code
Result	Not found
Severity	LOW

6. DISCLAIMER

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability) set forth in the Services Agreement, or the scope of services, and terms and conditions provided to the Company in connection with the Agreement. This report provided in connection with the Services set forth in the Agreement shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Agreement. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes without ExVul's prior written consent.

This report is not, nor should be considered, an "endorsement" or "disapproval" of any particular project or team. This report is not, nor should be considered, an indication of the economics or value of any "product" or "asset" created by any team or project that contracts ExVul to perform a security assessment. This report does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors, business, business model or legal compliance.

This report should not be used in any way to make decisions around investment or involvement with any particular project. This report in no way provides investment advice, nor should be leveraged as investment advice of any sort. This report represents an extensive assessing process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk. ExVul's position is that each company and individual are responsible for their own due diligence and continuous security. ExVul's goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies, and in no way claims any guarantee of security or functionality of the technology we agree to analyze.

7. REFERENCES

[1] MITRE. CWE-191: Integer Underflow (Wrap or Wraparound). https://cwe.mitre.org/data/definitions/191.html.

[2] MITRE. CWE-197: Numeric Truncation Error. https://cwe.mitre.org/data/definitions/197.html.

[3] MITRE. CWE-400: Uncontrolled Resource Consumption. https://cwe.mitre.org/data/definitions/400.html.

[4] MITRE. CWE-440: Expected Behavior Violation. https://cwe.mitre.org/data/definitions/440.html.

[5] MITRE. CWE-684: Protection Mechanism Failure. https://cwe.mitre.org/data/definitions/693.html.

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

[7] MITRE. CWE CATEGORY: Behavioral Problems. https://cwe.mitre.org/data/definitions/438.html.

[8] MITRE. CWE CATEGORY: Numeric Errors. https://cwe.mitre.org/data/definitions/189.html.

[9] MITRE. CWE CATEGORY: Resource Management Errors. https://cwe.mitre.org/data/definitions/399.html.

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

8. About Exvul Security

Premier Security for the Web3 Ecosystem

ExVul is a premier Web3 security firm committed to forging a secure and trustworthy decentralized ecosystem. Our elite team consists of security veterans from world-leading technology and blockchain security firms, including Huawei, YBB Captical, Qihoo 360, Amber, ByteDance, MoveBit, and PeckShield. Team member Nolan is ranked as a top-40 whitehat on Immunefi and is the platform's sole All-Star in the APAC region.

Our expertise covers the full spectrum of Web3 security. We conduct **meticulous smart contract audits**, having fortified thousands of projects on chains like Evm, Solana, Aptos, Sui etc. Our **Blockchain Protocol Audits** secure the core infrastructure of L1/L2 by uncovering deep-seated vulnerabilities. We also offer **comprehensive wallet audits** to protect user assets and provide **proactive web3 pentest**, enabling partners to neutralize threats before they strike.

Trusted by industry leaders, ExVul is the security partner for **OKX, Bitget, Cobo, Infini, Stacks, Aptos, Sui, CoreDAO, Sei** etc.

Contact

Website
www.exvul.com

Email
contact@exvul.com

Twitter
@EXVULSEC

Github
github.com/EXVUL-Sec