EXVUL

SMART CONTRACT AUDIT REPORT

Mantle LSP

OCTOBER 2025

Contents

1. EXECUTIVE SUMMARY

ExVul Web3 Security was engaged by **Mantle LSP** 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
Mantle LSP	14/10/2025 - 20/10/2025	Solidity

Repository
https://github.com/mantle-lsp/contracts/pull/17

Commit Hash
1602a4a438b5f35998d17fd81dd624c1d3322361

2.2 Summary

Severity	Found
CRITICAL	0
HIGH	0
MEDIUM	6
LOW	5
INFO	1

2.3 Key Findings

Severity	Findings Title	Status
MEDIUM	managerAddress duplicate addition risk	Fixed
MEDIUM	borrow and repay Can Fail Due to Incorrect ETH Transfer	Fixed
MEDIUM	Allocation cap can be reduced below already allocated balance	Fixed
MEDIUM	Fees can be charged on principal during top-up	Fixed
MEDIUM	setLiquidityBuffer does not manage EXECUTOR_ROLE	Fixed
MEDIUM	Blocked accounts can stake	Acknowledge
LOW	Missing Zero-Address Validation in initialize	Acknowledge
LOW	Manager lifecycle can lock funds	Acknowledge
LOW	Deposit can bypass pause when no allocation is executed	Fixed
LOW	Deposit bounds can be set inconsistently	Acknowledge
LOW	Stakes and unstakes can be processed for zero value	Acknowledge
INFO	managerId is missing display check	Acknowledge

Table 2.3: Key Audit Findings

3. DETAILED DESCRIPTION OF FINDINGS

3.1 managerAddress duplicate addition risk

SEVERITY: MEDIUM **STATUS:** Fixed

PATH:

`src/liquidityBuffer/LiquidityBuffer.sol`

DESCRIPTION:

In the LiquidityBuffer.sol contract, addPositionManager allows a specific manager to add a PositionManagerConfig. There is no check whether the managerAddress already exists, which will cause different managerIds to correspond to the same managerAddress.

```
// src/liquidityBuffer/LiquidityBuffer.sol
function addPositionManager(
    address managerAddress,
    uint256 allocationCap
) external onlyRole(POSITION_MANAGER_ROLE) returns (uint256 managerId) {
    managerId = positionManagerCount;
    positionManagerCount++;

    positionManagerConfigs[managerId] = PositionManagerConfig({
        managerAddress: managerAddress,
        allocationCap: allocationCap,
        isActive: true
    });
    positionAccountants[managerId] = PositionAccountant({
        allocatedBalance: 0,
        interestClaimedFromManager: 0
    });

    totalAllocationCapacity += allocationCap;
    emit ProtocolConfigChanged(
        this.addPositionManager.selector,
        "addPositionManager(address,uint256)",
        abi.encode(managerAddress, allocationCap)
    );
```

```
}
```

IMPACT:

In getControlledBalance, repeated calculations may occur, resulting in unexpected results.

RECOMMENDATIONS:

Use a mapping to record whether the managerAddress already exists and verify it during execution.

```
+mapping(address => bool) public isRegisteredManager;
function addPositionManager(
    address managerAddress,
    uint256 allocationCap
) external onlyRole(POSITION_MANAGER_ROLE) returns (uint256 managerId) {
+   require(!isRegisteredManager[managerAddress], 'Manager already
    registered');
    managerId = positionManagerCount;
    positionManagerCount++;

    positionManagerConfigs[managerId] = PositionManagerConfig({
        managerAddress: managerAddress,
        allocationCap: allocationCap,
        isActive: true
    });
    positionAccountants[managerId] = PositionAccountant({
        allocatedBalance: 0,
        interestClaimedFromManager: 0
    });

+   isRegisteredManager[managerAddress] = true;
    totalAllocationCapacity += allocationCap;
    emit ProtocolConfigChanged(
        this.addPositionManager.selector,
        'addPositionManager(address,uint256)',
        abi.encode(managerAddress, allocationCap)
    );
}
```

3.2 borrow and repay Can Fail Due to Incorrect ETH Transfer

SEVERITY: MEDIUM **STATUS:** Fixed

PATH:

`src/liquidityBuffer/PositionManager.sol`

DESCRIPTION:

Both the borrow and repay functions in PositionManager use _safeTransferETH to send raw ETH to their caller. The designated caller (msg.sender) for both functions is the LiquidityBuffer contract, which is designed to reject raw ETH transfers.

IMPACT:

The borrow function is unusable. The repay function is fragile and will fail on any overpayment, disrupting automated debt management.

RECOMMENDATIONS:

Modify both functions to use the correct interaction pattern for sending ETH to LiquidityBuffer, as seen in the withdraw function. This involves calling a dedicated payable function on the LiquidityBuffer contract.

```
// In borrow():
-         _safeTransferETH(msg.sender, amount);
+         liquidityBuffer.receiveETHFromPositionManager{value: amount}();

// In repay():
-         if (msg.value > repayAmount) {
-             _safeTransferETH(msg.sender, msg.value - repayAmount);
-         }
+         if (msg.value > repayAmount) {
+             liquidityBuffer.receiveETHFromPositionManager{value:
    msg.value - repayAmount}();
+         }
```

3.3 Allocation cap can be reduced below already allocated balance

SEVERITY: MEDIUM **STATUS:** Fixed

PATH:

`src/liquidityBuffer/LiquidityBuffer.sol`

DESCRIPTION:

updatePositionManager allows lowering allocationCap below the current allocatedBalance, breaking the allocatedBalance <= allocationCap invariant. This can also cause totalAllocatedBalance to exceed totalAllocationCapacity.

```
function updatePositionManager(
        uint256 managerId,
        uint256 newAllocationCap,
        bool isActive
    ) external onlyRole(POSITION_MANAGER_ROLE) {
        if (managerId >= positionManagerCount) {
            revert LiquidityBuffer__ManagerNotFound();
        }

        PositionManagerConfig storage config =
   positionManagerConfigs[managerId];

        // Update total allocation capacity
        totalAllocationCapacity = totalAllocationCapacity -
   config.allocationCap + newAllocationCap;

        config.allocationCap = newAllocationCap;
        config.isActive = isActive;

        emit ProtocolConfigChanged(
            this.updatePositionManager.selector,
            ''updatePositionManager(uint256,uint256,bool)'',
            abi.encode(managerId, newAllocationCap, isActive)
        );
    }
```

IMPACT:

Breaks core accounting invariants. If totalAllocatedBalance exceeds totalAllocationCapacity, getAvailableCapacity() reverts due to an underflow, causing a Denial of Service on that view function.

RECOMMENDATIONS:

Enforce that the new allocation cap is not less than the currently allocated balance before updating.

```
+        // Prevent cap below current allocation
+        require(
+            newAllocationCap >=
    positionAccountants[managerId].allocatedBalance,
+            "cap < allocated"
+        );
         // Update total allocation capacity
         totalAllocationCapacity = totalAllocationCapacity -
    config.allocationCap + newAllocationCap;
```

3.4 Fees can be charged on principal during top-up

SEVERITY: MEDIUM **STATUS:** Fixed

PATH:

`src/liquidityBuffer/LiquidityBuffer.sol`

DESCRIPTION:

The topUpInterestToStaking function allows a privileged role to treat any ETH balance in the contract as interest. It does not distinguish between earned interest and principal that may be temporarily held in the contract.

```
// src/liquidityBuffer/LiquidityBuffer.sol
function topUpInterestToStaking(uint256 amount) external
    onlyRole(INTEREST_TOPUP_ROLE) {
     //[...]
     _topUpInterestToStakingAndCollectFees(amount);
}

function _topUpInterestToStakingAndCollectFees(uint256 amount) internal {
     //[...]
     uint256 fees = Math.mulDiv(feesBasisPoints, amount,
    _BASIS_POINTS_DENOMINATOR);
     //[...]
}
```

IMPACT:

This can lead to fees being incorrectly charged on stakers' principal, causing direct economic loss.

RECOMMENDATIONS:

Introduce a pendingInterest state variable to explicitly track earned interest. The top-up function should be constrained by this variable to ensure fees are only levied on actual profits.

```
//[...]
```

```
     uint256 public totalFeesCollected;
+    uint256 public pendingInterest;
//[...]
function _topUpInterestToStakingAndCollectFees(uint256 amount) internal {
-    uint256 fees = Math.mulDiv(feesBasisPoints, amount,
    _BASIS_POINTS_DENOMINATOR);
+    require(amount <= pendingInterest, 'exceeds pending interest');
+    pendingInterest -= amount;
+    uint256 fees = Math.mulDiv(feesBasisPoints, amount,
    _BASIS_POINTS_DENOMINATOR);
//[...]
}
//[...]
function _claimInterestFromManager(uint256 managerId) internal returns
    (uint256) {
//[...]
if (interestAmount > 0) {
//[...]
    positionAccountants[managerId].interestClaimedFromManager +=
    interestAmount;
    totalInterestClaimed += interestAmount;
+   pendingInterest += interestAmount;
    emit InterestClaimed(managerId, interestAmount);
//[...]
}
```

3.5 setLiquidityBuffer does not manage EXECUTOR_ROLE

SEVERITY: MEDIUM **STATUS:** Fixed

PATH:

`src/liquidityBuffer/PositionManager.sol`

DESCRIPTION:

setLiquidityBuffer updates the stored liquidityBuffer address but does not grant EXECUTOR_ROLE to the new address nor revoke it from the old one.

```
// src/liquidityBuffer/PositionManager.sol
function setLiquidityBuffer(address _liquidityBuffer) external
    onlyRole(MANAGER_ROLE) {
     liquidityBuffer = ILiquidityBuffer(_liquidityBuffer);
}
```

IMPACT:

The new LiquidityBuffer cannot call position functions.

RECOMMENDATIONS:

Revoke the role from the old address and grant it to the new one.

3.6 Blocked accounts can stake

SEVERITY: MEDIUM **STATUS:** Acknowledge

PATH:

`src/METH.sol`, `src/Staking.sol`

DESCRIPTION:

Blocklist checks in METH.sol are only on _transfer, not _mint. A blocked address can successfully stake() and receive mETH, but the subsequent unstakeRequest() will fail because it calls transferFrom, which is blocked.

```
// src/METH.sol
function _transfer(address from, address to, uint256 amount) internal
    override notBlocked(from, to) {
     super._transfer(from, to, amount);
}

// src/Staking.sol
function _unstakeRequest(...) internal returns (uint256) {
     // ...
     SafeERC20Upgradeable.safeTransferFrom(mETH, msg.sender,
    address(unstakeRequestsManager), methAmount);
     // ...
}
```

IMPACT:

A user on the blocklist can deposit ETH, but their resulting mETH becomes permanently locked as it can neither be transferred nor unstaked, leading to a loss of funds.

RECOMMENDATIONS:

Prevent blocked addresses from staking at the entry point by adding a check in Staking.stake().

3.7 Missing Zero-Address Validation in initialize

SEVERITY: LOW **STATUS:** Acknowledge

PATH:

`Not specified`

DESCRIPTION:

initialize stores critical addresses without zero checks, so misconfigured deployment bricks staking roles and integrations.

IMPACT:

Passing address(0) for any dependency breaks staking or governance until a full redeploy.

RECOMMENDATIONS:

Add zero address check.

3.8 Manager lifecycle can lock funds

SEVERITY: LOW **STATUS:** Acknowledge

PATH:

`src/liquidityBuffer/LiquidityBuffer.sol`

DESCRIPTION:

Managers can be deactivated with nonzero allocations and inbound returns from inactive managers are rejected, leading to stuck funds. updatePositionManager can also deactivate without zeroing allocations.

```
// src/liquidityBuffer/LiquidityBuffer.sol
function togglePositionManagerStatus(uint256 managerId) external
    onlyRole(POSITION_MANAGER_ROLE) {
     if (managerId >= positionManagerCount) {
         revert LiquidityBuffer__ManagerNotFound();
     }
     PositionManagerConfig storage config =
    positionManagerConfigs[managerId];
     config.isActive = !config.isActive;
}
```

IMPACT:

Funds stuck until reactivation.

RECOMMENDATIONS:

Enforce zero allocation on deactivation and accept returns from any known manager.

3.9 Deposit can bypass pause when no allocation is executed

SEVERITY: LOW **STATUS:** Fixed

PATH:

`src/liquidityBuffer/LiquidityBuffer.sol`

DESCRIPTION:

The depositETH() function lacks a pause check at its entry point. The check is only performed conditionally within _allocateETHToManager. If shouldExecuteAllocation is set to false, this check is bypassed, allowing deposits to occur during a protocol pause.

```
// src/liquidityBuffer/LiquidityBuffer.sol
function depositETH() external payable onlyRole(LIQUIDITY_MANAGER_ROLE) {
    _receiveETHFromStaking(msg.value);
    if (shouldExecuteAllocation) {
        _allocateETHToManager(defaultManagerId, msg.value);
    }
}
```

IMPACT:

The pause mechanism behaves inconsistently, which can lead to operational confusion during an emergency.

RECOMMENDATIONS:

Add a pause check at the beginning of depositETH to ensure the pause is always enforced, regardless of the allocation strategy.

```
function depositETH() external payable onlyRole(LIQUIDITY_MANAGER_ROLE) {
+    if (pauser.isLiquidityBufferPaused()) revert
   LiquidityBuffer__Paused();
     _receiveETHFromStaking(msg.value);
     if (shouldExecuteAllocation) {
         _allocateETHToManager(defaultManagerId, msg.value);
```

```
        }
}
```

3.10 Deposit bounds can be set inconsistently

SEVERITY: LOW **STATUS:** Acknowledge

PATH:

`src/Staking.sol`

DESCRIPTION:

setMinimumDepositAmount and setMaximumDepositAmount do not enforce min ≤ max, allowing misconfiguration of bricks validator initiation.

```
// src/Staking.sol
function setMinimumDepositAmount(uint256 minimumDepositAmount_) external
    onlyRole(STAKING_MANAGER_ROLE) {
     minimumDepositAmount = minimumDepositAmount_;
}
function setMaximumDepositAmount(uint256 maximumDepositAmount_) external
    onlyRole(STAKING_MANAGER_ROLE) {
     maximumDepositAmount = maximumDepositAmount_;
}
```

IMPACT:

All deposits can be blocked depending on ordering of admin operations; operational DoS.

RECOMMENDATIONS:

Validate the relationship between both setters.

```
function setMinimumDepositAmount(uint256 minimumDepositAmount_) external
    onlyRole(STAKING_MANAGER_ROLE) {
-     minimumDepositAmount = minimumDepositAmount_;
+     require(minimumDepositAmount_ <= maximumDepositAmount, 'min>max');
+     minimumDepositAmount = minimumDepositAmount_;
}
function setMaximumDepositAmount(uint256 maximumDepositAmount_) external
    onlyRole(STAKING_MANAGER_ROLE) {
```

```
-     maximumDepositAmount = maximumDepositAmount_;
+     require(maximumDepositAmount_ >= minimumDepositAmount, 'max<min');
+     maximumDepositAmount = maximumDepositAmount_;
}
```

3.11 Stakes and unstakes can be processed for zero value

SEVERITY: LOW **STATUS:** Acknowledge

PATH:

`src/Staking.sol`

DESCRIPTION:

The conversion functions ethToMETH and mETHToETH can round down to zero if the input amount is small. The stake and _unstakeRequest functions do not prevent these zero-value conversions from proceeding if the user provides a minimum expected return of 0.

```
// src/Staking.sol
function stake(uint256 minMETHAmount) external payable {
    // ...
    uint256 mETHMintAmount = ethToMETH(msg.value);
    if (mETHMintAmount < minMETHAmount) {
        revert StakeBelowMinimumMETHAmount(mETHMintAmount, minMETHAmount);
    }
    unallocatedETH += msg.value;
    mETH.mint(msg.sender, mETHMintAmount);
}

// src/Staking.sol
function _unstakeRequest(uint128 methAmount, uint128 minETHAmount)
   internal returns (uint256) {
    // ...
    uint128 ethAmount = uint128(mETHToETH(methAmount));
    if (ethAmount < minETHAmount) {
        revert UnstakeBelowMinimumETHAmount(ethAmount, minETHAmount);
    }

    unstakeRequestsManager.create({requester: msg.sender, mETHLocked:
   methAmount, ethRequested: ethAmount});
    SafeERC20Upgradeable.safeTransferFrom(mETH, msg.sender,
   address(unstakeRequestsManager), methAmount);
    // ...
}
```

IMPACT:

A user can lose their deposited funds (ETH during stake, mETH during unstake) in exchange for nothing. This is effectively a donation to the protocol. It also creates unnecessary on-chain events with zero amounts.

RECOMMENDATIONS:

Reject any conversion that results in a zero amount in the stake and _unstakeRequest functions.

3.12 managerId is missing display check

SEVERITY: INFO **STATUS:** Acknowledge

PATH:

`src/liquidityBuffer/LiquidityBuffer.sol`

DESCRIPTION:

getInterestAmount is a view function that can query Interest by passing in a managerId. There is no explicit check whether the managerId is initialized. If not, the default value will be returned.

```
function getInterestAmount(uint256 managerId) public view returns
    (uint256) {
     PositionManagerConfig memory config =
    positionManagerConfigs[managerId];
     // Get current underlying balance from position manager
     IPositionManager manager = IPositionManager(config.managerAddress);
     uint256 currentBalance = manager.getUnderlyingBalance();

     // Calculate interest as: current balance - allocated balance
     PositionAccountant memory accounting = positionAccountants[managerId];

     if (currentBalance > accounting.allocatedBalance) {
         return currentBalance - accounting.allocatedBalance;
     }

     return 0;
}
```

IMPACT:

If managerId does not exist, the default value address(0) will be used, causing the transaction to revert.

RECOMMENDATIONS:

Explicitly verify managerId.

```
function getInterestAmount(uint256 managerId) public view returns
    (uint256) {
+    if (managerId >= positionManagerCount) {
+        revert LiquidityBuffer__ManagerNotFound();
+    }
     PositionManagerConfig memory config =
    positionManagerConfigs[managerId];
     // ...
}
```

4. CONCLUSION

In this audit, we thoroughly analyzed **Mantle LSP** 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

% … Appendix content continues as per template …

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.

EXVUL

Contact

Website
www.exvul.com

Email
contact@exvul.com

Twitter
@EXVULSEC

Github
github.com/EXVUL-Sec

ExVul