ExVul

SMART CONTRACT AUDIT REPORT

Aimonica EVM Smart Contract

AUGUST 2025

Contents

1. EXECUTIVE SUMMARY

ExVul Web3 Security was engaged by **Aimonica** 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
Aimonica EVM	13/08/2025 - 14/08/2025	Solidity

Repository
https://github.com/Aimonica-Brands/aimonica-core-evm.git

Commit Hash
1f8dbcfbb742ded989451d889beb095cc28e9cb1

2.2 Summary

Severity	Found
CRITICAL	0
HIGH	0
MEDIUM	2
LOW	1
INFO	1

2.3 Key Findings

Severity	Findings Title	Status
MEDIUM	Emergency Unstake Time Validation Bypass	Fixed
MEDIUM	Missing UUPS Upgrade Pattern Implementation	Fixed
LOW	Fee-on-Transfer Token Mistake Handling and Missing SafeERC20	Fixed
INFO	Gas-Inefficient Field Ordering in Stake Struct Increases Gas Costs	Fixed

Table 2.3: Key Audit Findings

3. DETAILED DESCRIPTION OF FINDINGS

3.1 Emergency Unstake Time Validation Bypass

SEVERITY: MEDIUM **STATUS:** Fixed

PATH:

`AimStaking.sol::emergencyUnstake()`

DESCRIPTION:

The emergencyUnstake function omits time validation, enabling users to bypass lockup restrictions and incur high emergency fees after the lockup period has ended. Unlike the standard unstake function which properly validates lockup completion, emergency unstake proceeds without temporal checks.

The vulnerable code section:

```
function emergencyUnstake(uint256 stakeId) external nonReentrant {
    Stake storage userStake = stakes[stakeId];
    require(userStake.user == msg.sender, "Not stake owner");
    require(userStake.status == StakeStatus.Active, "Stake not active");

    userStake.status = StakeStatus.EmergencyUnstaked;

    uint256 amountToUnstake = userStake.amount;
    // Apply emergency unstake fee if applicable
    if (emergencyUnstakeFeeRate > 0 && feeWallet != address(0)) {
        uint256 fee = (amountToUnstake * emergencyUnstakeFeeRate) / 10000;
        if (fee > 0) {
            IERC20(userStake.stakingToken).transfer(feeWallet, fee);
        }
        amountToUnstake -= fee;
    }

    // Transfer the remaining amount back to the user
    IERC20(userStake.stakingToken).transfer(msg.sender, amountToUnstake);

    emit EmergencyUnstaked(stakeId, msg.sender, userStake.amount);
}
```

IMPACT:

Users can be charged excessive emergency unstake fees after lockup expiration, violating the intended emergency withdrawal design and causing unnecessary financial losses. The vulnerability contradicts documented behavior and creates inconsistent fee structures.

RECOMMENDATIONS:

Add lockup period validation to prevent emergency unstake after lockup expiration:

```
function emergencyUnstake(uint256 stakeId) external nonReentrant {
    Stake storage userStake = stakes[stakeId];
    require(userStake.user == msg.sender, "Not stake owner");
    require(userStake.status == StakeStatus.Active, "Stake not active");
+   require(block.timestamp < userStake.unlockedAt, "Lockup period ended,
   use regular unstake");

    userStake.status = StakeStatus.EmergencyUnstaked;
    // ... rest of the function
}
```

3.2 Missing UUPS Upgrade Pattern Implementation

SEVERITY: MEDIUM **STATUS:** Fixed

PATH:

`scripts/deployProxy.ts`

DESCRIPTION:

The README claims the contract uses UUPS pattern, but the actual implementation and deployment scripts use Transparent Proxy (the default for Hardhat's upgrades plugin). The contract is missing the required UUPS upgrade pattern implementation.

```
// scripts/deployProxy.ts
import { ethers, upgrades } from "hardhat";
const contract = await upgrades.deployProxy(factory, initializeArgs, {
    initializer: "initialize" });
```

According to OpenZeppelin documentation, the default proxy kind is "transparent" unless explicitly specified as "uups".

IMPACT:

The deployment scripts use OpenZeppelin's default Transparent Proxy pattern instead of UUPS, contradicting the README documentation. While upgrades are still possible, this creates governance model inconsistencies and potential confusion for developers and users.

RECOMMENDATIONS:

Option 1: Implement proper UUPS upgrade pattern:

```
+ import "
    @openzeppelin/contracts-upgradeable/proxy/utils/UUPSUpgradeable.sol";

- contract AimStaking is AccessControlEnumerableUpgradeable,
    ReentrancyGuardUpgradeable {
+ contract AimStaking is AccessControlEnumerableUpgradeable,
    ReentrancyGuardUpgradeable, UUPSUpgradeable {
```

```
    function initialize(address admin) external initializer {
+         __UUPSUpgradeable_init();
+         __AccessControlEnumerable_init();
         __ReentrancyGuard_init();
         _grantRole(DEFAULT_ADMIN_ROLE, admin);
         _grantRole(MANAGER_ROLE, admin);
         feeWallet = admin;
     }

+     function _authorizeUpgrade(address newImplementation) internal
    override onlyRole(DEFAULT_ADMIN_ROLE) {}

+     uint256[50] private __gap; // Storage gap for future upgrades
}
```

And update deployment script:

```
- const contract = await upgrades.deployProxy(factory, initializeArgs, {
    initializer: "initialize" });
+ const contract = await upgrades.deployProxy(factory, initializeArgs, {
    initializer: "initialize", kind: "uups" });
```

Option 2: Update README to reflect Transparent Proxy usage by removing UUPS claims from documentation and clarifying Transparent Proxy pattern usage.

3.3 Fee-on-Transfer Token Mistake Handling and Missing SafeERC20

SEVERITY: LOW **STATUS:** Fixed

PATH:

`AimStaking.sol::stake(), unstake()`

DESCRIPTION:

Uses direct IERC20.transfer/transferFrom calls without SafeERC20, and fails to account for fee-on-transfer tokens. This can lead to accounting inconsistencies and potential insolvency when dealing with tokens that charge fees on transfer.

```
// Stake - no accounting for actual received amount
IERC20(stakingTokenAddress).transferFrom(msg.sender, address(this),
    amount);

// Unstake - direct transfer without SafeERC20
IERC20(userStake.stakingToken).transfer(feeWallet, fee);
IERC20(userStake.stakingToken).transfer(msg.sender, amountToUnstake);
```

IMPACT:

Fee-on-transfer tokens cause accounting mismatches where the contract promises to return more tokens than it actually holds, leading to potential insolvency and fund locking due to silent transfer failures.

RECOMMENDATIONS:

Use SafeERC20 for all token transfers and use the actual received amount for accounting:

```
+ import ”
    @openzeppelin/contracts-upgradeable/token/ERC20/utils/SafeERC20Upgradeable.sol”;

contract AimStaking is AccessControlEnumerableUpgradeable,
    ReentrancyGuardUpgradeable {
+     using SafeERC20Upgradeable for IERC20;
```

```
    function stake(uint256 amount, uint256 durationInDays, bytes32
        projectId) external nonReentrant {
-         IERC20(stakingTokenAddress).transferFrom(msg.sender,
    address(this), amount);
+         uint256 beforeBalance =
    IERC20(stakingTokenAddress).balanceOf(address(this));
+         IERC20(stakingTokenAddress).safeTransferFrom(msg.sender,
    address(this), amount);
+         uint256 actualReceived =
    IERC20(stakingTokenAddress).balanceOf(address(this)) - beforeBalance;

        // Use actualReceived instead of amount for accounting
        stakes[stakeId] = Stake({
            // ...
-           amount: amount,
+           amount: actualReceived,
            // ...
        });
    }

    function unstake(uint256 stakeId) external nonReentrant {
        // ...
-         IERC20(userStake.stakingToken).transfer(feeWallet, fee);
-         IERC20(userStake.stakingToken).transfer(msg.sender,
    amountToUnstake);
+         IERC20(userStake.stakingToken).safeTransfer(feeWallet, fee);
+         IERC20(userStake.stakingToken).safeTransfer(msg.sender,
    amountToUnstake);
    }
}
```

3.4 Gas-Inefficient Field Ordering in Stake Struct Increases Gas Costs

SEVERITY: INFO **STATUS:** Fixed

PATH:

`AimStaking.sol::Stake struct`

DESCRIPTION:

The fields within the Stake struct are not ordered according to gas optimization principles. Types smaller than 32 bytes are separated by full 32-byte types. This layout prevents the Solidity compiler from packing multiple smaller variables into a single 32-byte storage slot, leading to unnecessary space waste.

```
struct Stake {
    uint256 stakeId;      // 32 bytes
    address user;         // 20 bytes
    uint256 amount;       // 32 bytes
    bytes32 projectId;    // 32 bytes
    address stakingToken; // 20 bytes
    uint256 stakedAt;     // 32 bytes
    uint256 duration;     // 32 bytes
    uint256 unlockedAt;   // 32 bytes
    StakeStatus status;   // 1 byte
}
```

IMPACT:

This inefficient storage layout results in higher transaction fees for users performing stakes and wastes on-chain storage space.

RECOMMENDATIONS:

Reorder struct fields to optimize storage packing:

```
struct Stake {
    // Full 32-byte fields
```

```
    uint256 stakeId;
    uint256 amount;
    bytes32 projectId;
    uint256 stakedAt;
    uint256 duration;
    uint256 unlockedAt;

    // Packable fields smaller than 32 bytes
    address stakingToken; // Occupies one slot

    address user;         // Packed with status in one slot
    StakeStatus status;
}
```

This optimization packs the `address user` (20 bytes) and `StakeStatus status` (1 byte) into a single 32-byte storage slot, reducing gas costs for struct operations.

4. CONCLUSION

In this audit, we thoroughly analyzed **Aimonica EVM** 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

Contact

Website
www.exvul.com

Email
contact@exvul.com

Twitter
@EXVULSEC

Github
github.com/EXVUL-Sec