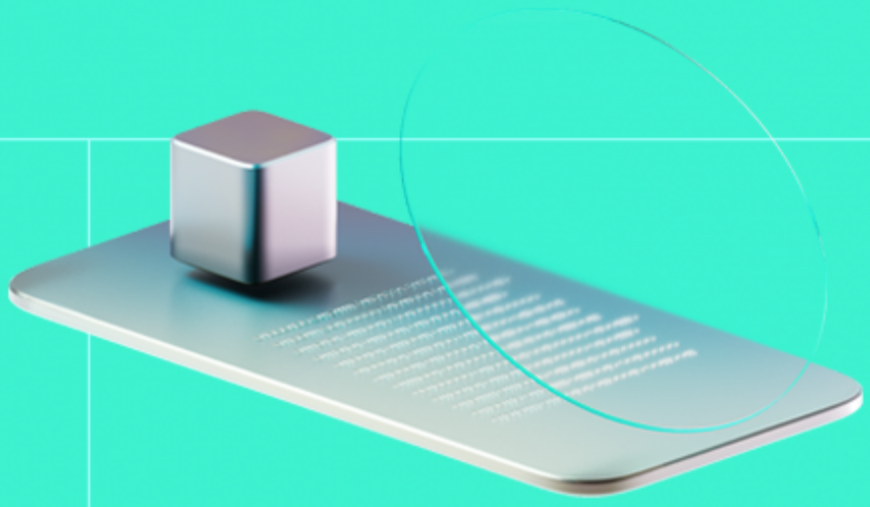
HACKEN

Smart Contract Code Review And Security Analysis Report



Customer: ECOMI

Date: 15/09/2026

We express our gratitude to the ECOMI team for the collaborative engagement that enabled the execution of this Smart Contract Security Assessment.

OMIStaking is a time-locked staking protocol on Base L2. The hardcoded OMI ERC-20 at `0x3792DBDD07e87413247DF995e692806aa13D3299` is locked into one position per wallet for a registered duration, and no on-chain yield or reward token is distributed.

Document

Name	Smart Contract Code Review and Security Analysis Report for ECOMI
Audited By	Khrystyna Tkachuk
Approved By	Kornel Światłowski
Website	https://www.ecomi.com/
Changelog	09/09/2026 - Preliminary Report
	15/09/2026 - Final Report
Platform	Base
Language	Solidity
Tags	Staking, ERC20, Upgradable, Centralization
Methodology	https://docs.hacken.io/methodologies/smart-contracts

Review Scope

Repository	https://github.com/ecomi-admin/omistaking-audit
Commit	34559f4
Final Commit	253edbc

Audit Summary

The system users should acknowledge all the risks summed up in the risks section of the report

2	2	0	0
Total Findings	Resolved	Accepted	Mitigated

Findings by Severity

Severity	Count
Critical	0
High	0
Medium	0
Low	1

Vulnerability	Severity	Status
F-2026-19275 - Missing Destination Validation in sweepStuckPenalties Reclassifies Tracked Penalties as Recoverable Surplus	Low	Fixed
F-2026-19277 - Redundant Imports	Info	Fixed

Documentation quality

- Functional requirements are sufficient.
  - Project overview is detailed.
  - All roles in the system are described.
  - Use cases are described.
  - Futures are described.
- Technical description is detailed.
  - Run instructions are provided.
  - Technical specification is provided.
  - NatSpec is sufficient.

Code quality

- The code leverages OpenZeppelin contracts and follows established patterns.
- The development environment is configured.

Test coverage

Code coverage of the project is **93.98%** (branch coverage).

- Deployment and basic user interactions are covered with tests.
- Negative cases for core guards are covered.
- Interactions by several users are not tested thoroughly.

Table of Contents

System Overview

The in-scope system is a single contract, **OMIStaking**, deployed behind a UUPS (ERC-1967) proxy. It inherits OpenZeppelin's upgradeable **Ownable2Step**, **Pausable**, **ReentrancyGuard**, and **UUPS** bases. A **TimelockController** owner, a 48-hour delay, and a pauser multisig are intended deployment wiring (per spec) and are not encoded in this bytecode.

Each wallet may hold one time-locked `StakePosition`. Principal is pulled on entry or top-up, may be re-locked after maturity or extended to a longer registered tier, and is always returned to `msg.sender` on exit. Early exit applies a bounded penalty that is forwarded to `penaltyAddress`, or accrued as `stuckPenalties` if that transfer fails. Pause blocks new locks and lock changes; withdrawals remain available. On-chain stake floors and top-up cooldowns are not enforced. Lifecycle events are the surface for off-chain XP accounting (per spec). The owner configures penalty parameters, registered tiers, and the pauser, recovers surplus OMI and any non-OMI ERC-20s, and authorizes upgrades. Ownership cannot be renounced.

Files in Scope

- **OMIStaking_v5.sol**: Single UUPS implementation compiled as **OMIStaking**. Holds per-wallet OMI lock positions and exposes `stake`, `topUp`, `withdraw`, `earlyWithdraw`, `restake`, `extendLockPeriod`, `pauser` halt of new entries, and `owner` configuration including surplus-capped `recoverTokens` and `sweepStuckPenalties`.

Privileged roles

- **owner** (inherited from **Ownable2StepUpgradeable**): Gated by `onlyOwner`. Authorizes configuration, token recovery, UUPS upgrades, pauser assignment, fallback pause control, and two-step ownership transfer. The on-chain role is `owner`.
  - Can call `setPenaltyAddress` to update the early-exit penalty destination. Zero address, `address(this)`, and `OMI_TOKEN` are rejected.
  - Can call `setEarlyWithdrawPenaltyBps` to set the early-exit penalty rate in basis points, bounded by `MIN_PENALTY_BPS` (500) and `MAX_PENALTY_BPS` (2,000).
  - Can call `setPauser` to replace the `pauser` address. Zero address is rejected.
  - Can call `addTier` to register a new lock duration. Duration must be greater than 0, at most `MAX_TIER_DURATION` (720 days), and not already registered. The registered count is capped at `MAX_TIERS` (20).
  - Can call `removeTier` to deregister a lock duration. Active stakes in that tier are unaffected. The last remaining tier cannot be removed.
  - Can call `recoverTokens` to transfer a requested amount of an ERC-20 to a chosen destination. For OMI, only the surplus above `totalStakedOMI + stuckPenalties` is recoverable. For other tokens, the requested amount is transferred and `safeTransfer` reverts if the balance is insufficient.
  - Can call `sweepStuckPenalties` to transfer accumulated `stuckPenalties` OMI to a chosen destination and reset `stuckPenalties` to 0.

    - Can call `pause` to halt `stake`, `topUp`, `restake`, and `extendLockPeriod`. `withdraw` and `earlyWithdraw` remain available. `onlyPauser` allows pauser or owner.
    - Can call `unpause` to resume `stake`, `topUp`, `restake`, and `extendLockPeriod`. `onlyPauser` allows pauser or owner.
    - Can call `transferOwnership` to nominate a new owner as `pendingOwner`.
    - Can call `renounceOwnership`, which is overridden and always reverts. Ownership cannot be renounced.
    - Can call `upgradeToAndCall` (inherited from **UUPSUpgradeable**, gated by `_authorizeUpgrade` with `onlyOwner`) to replace the proxy implementation.
- **pendingOwner** (inherited from **Ownable2StepUpgradeable**): Completes a two-step ownership transfer after `transferOwnership`.
    - Can call `acceptOwnership` to become `owner`.
- **pauser**: Fast-path emergency pause. Stored in `pauser`. `onlyPauser` allows this address or `owner`.
    - Can call `pause` to halt `stake`, `topUp`, `restake`, and `extendLockPeriod`. `withdraw` and `earlyWithdraw` remain available.
    - Can call `unpause` to resume `stake`, `topUp`, `restake`, and `extendLockPeriod`.
- **address(this)** (self-call trampoline, not an admin role): Restricted to `msg.sender == address(this)`. Not an administrative role.
    - Can call `_forwardPenalty` to transfer OMI to a destination as the try/catch trampoline used by `earlyWithdraw`. Any other caller reverts.

Potential Risks

- **Scope Definition and Security Guarantees**: The audit source-file scope is limited to `contracts/OMIStaking_v5.sol` (**OMIStaking**). Other repository Solidity includes the testnet mirror **OMIStakingTest**, **OMIEmberMonthTest**, **OMIPassTest**, **TestOMI**, **MockBlacklistOMI**, **OMIStakingTestLegacyV6**, `_OZImports.sol`, and Foundry mocks, handlers, and invariant tests. Comments in **OMIStaking** defer TimelockController and multisig wiring to out-of-scope deploy scripts such as `contracts-deploy/scripts/deployMainnet.cjs`. Defects or mis-wiring in those out-of-scope artifacts can still determine who `owner` and `pauser` are and how upgrades are scheduled, without being covered by this review.
- **System Reliance on External Contracts**: Every value-moving path in **OMIStaking** depends on the immutable ERC-20 at `OMI_TOKEN` (`0x3792DBDD07e87413247DF995e692806aa13D3299`). `stake` and `topUp` pull with `safeTransferFrom`; `withdraw`, `earlyWithdraw`, `_forwardPenalty`, `recoverTokens`, and `sweepStuckPenalties` push with `safeTransfer`; `_omiSurplus` reads `balanceOf`. Comments require that token to behave as a standard, non-fee, non-rebasing, non-denylisting ERC-20 (named as OptimismMintableERC20). A revert or denylist on that token can block `stake` and `topUp` and, because the user payout in `withdraw` and `earlyWithdraw` is not inside try/catch, can also block the always-exit path.
- **External Precompile or Infrastructure Dependency**: **OMIStaking** is documented for Base L2, where comments state that true OMI burns are unavailable. `penaltyAddress` is described as routing early-exit OMI to a bridge or custodian for an L1 burn. `earlyWithdraw` forwards the penalty through `_forwardPenalty`; if that destination reverts, the user still forfeits `(total * earlyWithdrawPenaltyBps) / BPS_DENOMINATOR` and the tokens are added to `stuckPenalties` until `sweepStuckPenalties`. Penalty-burn completion and general liveness therefore depend on Base infrastructure and on the off-contract penalty destination, neither of which this implementation controls.
- **Coarse-Grained Authorization Model**: A single **owner** (inherited from **Ownable2StepUpgradeable**) gates `setPenaltyAddress`, `setEarlyWithdrawPenaltyBps`, `setPauser`, `addTier`, `removeTier`, `recoverTokens`, `sweepStuckPenalties`, `_authorizeUpgrade`, and `transferOwnership` on **OMIStaking**. `pause` and `unpause` are split to `onlyPauser`, but that `pauser` address is assigned by the same owner via `setPauser`. Compromise of `owner` therefore covers economic parameters, rescue of surplus and stuck penalties, pauser rotation, and replacement of UUPS logic in one role.
- **Absence of on-chain timelock and multi-signature controls**: **OMIStaking** gates configuration and upgrades with `onlyOwner` and `onlyPauser` and encodes no delay, schedule, confirmation count, or signature threshold. Comments describe `owner` as an OpenZeppelin TimelockController with a 48-hour delay and a multisig as timelock proposer, canceller, executor, and live `pauser`, yet they also state that the source guarantees nothing about role assignment. `initialize` stores `_owner` and `_pauser` as ordinary addresses, and `setEarlyWithdrawPenaltyBps`, `setPenaltyAddress`, `addTier`, `removeTier`, `setPauser`, `recoverTokens`, `sweepStuckPenalties`, and `_authorizeUpgrade` take effect as soon as the owner call succeeds. If `owner` is a single account, that account can raise the early-exit cost to 20%, retarget `penaltyAddress`, and replace UUPS logic in one transaction, while `pause` and `unpause` already execute immediately for `pauser` or `owner`.
- **Treasury or Rescue Authority**: `recoverTokens` (`onlyOwner`, `nonReentrant`) transfers an ERC-20 to a caller-chosen `destination`. When `token == OMI_TOKEN`, `amount` must be less than or equal to `_omiSurplus` (balance minus `totalStakedOMI` minus `stuckPenalties`); for any other token, the requested `amount` is transferred with no surplus cap, and `safeTransfer` reverts if the balance is insufficient.

`sweepStuckPenalties` sets `stuckPenalties` to zero and transfers that OMI to any non-zero `destination`, not necessarily `penaltyAddress`. Under this implementation those paths cannot reduce the staked principal floor, but they do let the owner extract surplus OMI, foreign tokens, and failed-forward penalties once `onlyOwner` is satisfied.

- **Single Points of Failure and Control**: Operational control of **OMIStaking** concentrates on **owner** and **pauser**. Via `onlyPauser`, `pauser` or `owner` can `pause` and halt `stake`, `topUp`, `restake`, and `extendLockPeriod`; `withdraw` and `earlyWithdraw` remain callable, and `pauser` has no token-transfer function. The owner additionally changes penalty parameters, rotates `pauser`, recovers surplus, sweeps `stuckPenalties`, and authorizes UUPS upgrades. Custody of those two addresses cannot be verified from this contract; owner compromise plus `_authorizeUpgrade` is the path that can rewrite accounting and move principal, while pauser compromise can freeze new entries and lock extensions only.
- **Centralized Governance Mechanisms**: The registered tier set, `earlyWithdrawPenaltyBps`, `penaltyAddress`, and `pauser` are not voted by OMI holders. They are written by **owner** through `addTier`, `removeTier`, `setEarlyWithdrawPenaltyBps`, `setPenaltyAddress`, and `setPauser`. Holders of active `StakePosition` entries have no on-chain veto other than `earlyWithdraw` at the then-current penalty or waiting until `unlockTime` for `withdraw`.
- **Flexibility and Risk in Contract Upgrades**: **OMIStaking** inherits **UUPSUpgradeable** and authorizes implementation changes only in `_authorizeUpgrade`. Comments state that UUPS is the sole admin path that can reach user principal, because `recoverTokens` is bounded by `_omiSurplus`. A replacement implementation can alter `stakes`, `totalStakedOMI`, or token transfers and extract locked OMI.
- **Incompatibility with Pending State During Upgrade**: Each wallet may hold a live `StakePosition` (`amount`, `tierDuration`, `startTime`, `unlockTime`, `lastTopUpTime`) whose OMI remains in the contract until `withdraw` or `earlyWithdraw`. Locks can last up to `MAX_TIER_DURATION` (720 days). An upgrade that reorders that struct, removes those exit functions, or changes how `totalStakedOMI` is interpreted can leave in-flight positions unable to exit under the new logic.
- **Lack of Emergency Downgrade Mechanism**: **OMIStaking** has no rollback, implementation snapshot, or previous-version registry. Recovering from a faulty upgrade requires another `onlyOwner` UUPS upgrade to prior bytecode. If the broken implementation disables `_authorizeUpgrade` or bricks `owner`, that recovery path is unavailable from this contract.'
- **Off-chain XP accounting and UI-owned floors**: **OMIStaking** distributes no on-chain rewards; comments state that an off-chain XP engine consumes `Staked`, `ToppedUp`, `Withdrawn`, `Restaked`, and `LockExtended`, and must classify exits by `Withdrawn.earlyExit` rather than `penaltyAmount`. `stake` and `topUp` require only `amount > 0`, so business floors and top-up pacing described as UI-owned are bypassable by a direct call, and dust stakes can enter `totalStakedOMI` with no on-chain qualifying threshold. `removeTier` leaves existing positions with a stored `tierDuration` that may be absent from `getValidTiers`. Engine downtime or mis-parsing does not trap OMI, but it can mis-allocate off-chain XP.
- **L2 penalty transfer is not an L1 burn**: Early-exit value is not burned inside **OMIStaking**. `earlyWithdraw` transfers the penalty to `penaltyAddress` via `_forwardPenalty`, and comments describe that address as a bridge or custodian path for an L1 burn of Base OptimismMintableERC20 OMI. This contract does not verify a burn, a burn amount, or that `penaltyAddress` is a bridge. Divergence between L2 penalty receipts and any L1 supply reduction cannot be detected or reconciled by **OMIStaking**.

- **Requested-amount escrow versus live token balances**: `stake` and `topUp` add the caller-supplied `amount` to `stakes[msg.sender].amount` and `totalStakedOMI` before `safeTransferFrom` of that same figure, not a measured balance delta. `_omiSurplus` compares `IERC20(OMI_TOKEN).balanceOf(address(this))` to `totalStakedOMI + stuckPenalties`. Comments require OMI to have no fee-on-transfer and no rebase, and they state that `OMI_TOKEN` must not be repointed in a future upgrade without reworking accounting. A token that violated those assumptions would desync the solvency floor from redeemable OMI.

Findings

Vulnerability Details

F-2026-19275 - Missing Destination Validation in sweepStuckPenalties Reclassifies Tracked Penalties as Recoverable Surplus - Low

Description:

The **OMIStaking** records early-exit penalty OMI that failed to forward to `penaltyAddress` via `stuckPenalties`. The bucket is excluded from `_omiSurplus` so it cannot be recovered as accidental surplus. The `setPenaltyAddress` bans `address(this)` and `OMI_TOKEN` as `penaltyAddress`, because a transfer to either destination succeeds without delivering value to a real sink.

The same two destinations remain legal on `sweepStuckPenalties`. Only a non-zero destination is required, and `stuckPenalties` is cleared before the transfer:

```
function sweepStuckPenalties(address destination) external onlyOwner nonReentrant {
    require(destination != address(0), "OMIStaking: zero destination");
    uint256 amount = stuckPenalties;
    require(amount > 0, "OMIStaking: nothing to sweep");

    stuckPenalties = 0;
    IERC20(OMI_TOKEN).safeTransfer(destination, amount);
    emit StuckPenaltiesSwept(destination, amount);
}
```

Once `earlyWithdraw` has credited the bucket, `sweepStuckPenalties` may be called with `address(this)` or `OMI_TOKEN`. A self-transfer leaves the token balance unchanged while `stuckPenalties` becomes 0, so `_omiSurplus` rises by the swept amount and a later `recoverTokens` of that OMI succeeds. A transfer to `OMI_TOKEN` credits the token contract, from which this ABI has no extract path. `StuckPenaltiesSwept` reports a completed routing in both cases.

The same wei can already be swept to a controlled wallet. Destinations already classified as dangerous break the stuck-versus-surplus split: tracked penalty OMI is reclassified as recoverable surplus, or it is parked inside the token contract. User principal is not reachable.

Assets:

- OMIStaking_v5.sol [github.com/ecomi-admin/omistaking-audit]

Status: Fixed

Classification

Impact Rate: 2/5

Likelihood Rate: 3/5

Exploitability: Dependent

Complexity: Simple

Severity: Low

Recommendations

Remediation: Apply the existing three-address predicate to every owner-supplied OMI destination:

```
function _requireValidOmiDestination(address dest) internal view {
    require(
        dest != address(0) && dest != address(this) && dest != OMI_TOKEN,
        "OMIStaking: invalid destination"
    );
}
```

Replace the zero-address check in `sweepStuckPenalties` with `_requireValidOmiDestination(destination)`. When `token` equals `OMI_TOKEN`, apply the same check in `recoverTokens`.

Resolution: Fixed in `a625805`.

The `sweepStuckPenalties` function in **OMIStaking** now calls `_requireValidOmiDestination` in place of the zero-address-only check. The helper reverts when `destination` is `address(0)`, `address(this)`, or `OMI_TOKEN`. The OMI branch of `recoverTokens` applies the same helper before surplus is transferred. Checks-effects-interactions order and the `nonReentrant` guard on both admin paths remain unchanged, and no storage slot was added.

```
function _requireValidOmiDestination(address dest) internal view {
    require(
        dest != address(0) && dest != address(this) && dest != OMI_TOKEN,
        "OMIStaking: invalid destination"
    );
}
```

F-2026-19277 - Redundant Imports - Info

Description:

The **OMIStaking** is upgradeable contract and it explicitly imports **Initializable** even though it is already pulled in transitively by every OpenZeppelin upgradeable base they inherit. **UUPSUpgradeable**, **Ownable2StepUpgradeable**, **PausableUpgradeable**, and **ReentrancyGuardUpgradeable** all inherit from **Initializable**, so the dedicated import statement adds nothing to the compilation unit and only duplicates a symbol that is already in scope.

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

contract OMIStaking is
    Initializable,
    Ownable2StepUpgradeable,
    PausableUpgradeable,
    ReentrancyGuardUpgradeable,
    UUPSUpgradeable
{
    ...
}
```

Additionally, **OMIStaking** contract uses plain import of **IERC20** alongside **SafeERC20**:

```
import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";

contract OMIStaking is
    Initializable,
    Ownable2StepUpgradeable,
    PausableUpgradeable,
    ReentrancyGuardUpgradeable,
    UUPSUpgradeable
{
    using SafeERC20 for IERC20;
    ...
}
```

SafeERC20 already relies on **IERC20**, so the explicit **IERC20** import in **OMIStaking** appears redundant from a code-maintenance perspective. Keeping both imports increases verbosity and can make the dependency surface look larger than it actually is.

This may lead to reduced code clarity and maintainability, as redundant import can make the contract appear more complex than necessary and may create minor confusion for reviewers or future maintainers.

Separately, these imports use the unqualified, global form (`import "...File.sol";`) rather than named imports. Global imports bring every top-level symbol from the target file into the current namespace, which obscures which symbols are actually used, increases the risk of identifier collisions, and makes the dependency surface harder to reason about during review. This is purely a code-quality and maintainability concern with no direct security impact.

Assets:

- OMIStaking_v5.sol [github.com/ecomi-admin/omistaking-audit]

Status: Fixed

Classification

Impact Rate: 1/5

Likelihood Rate: 5/5

Exploitability: Independent

Complexity: Simple

Severity: Info

Recommendations

Remediation:

- Remove redundant and unnecessary plain import of **IERC20** to simplify the codebase and improve readability.
- Remove the redundant **Initializable** import; the symbol remains available through the inherited upgradeable bases, and the contracts may still list **Initializable** in their inheritance order without the standalone import.
- Additionally, convert the remaining global imports to explicit named imports using the `{}` syntax, so contract declares exactly the modules it consumes. For example:

```
import { UUPSUpgradeable, Initializable } "@openzeppelin/contracts-upgradeable/proxy/utils/UUPSUpgradeable.sol";
import { Ownable2StepUpgradeable }"@openzeppelin/contracts-upgradeable/access/Ownable2StepUpgradeable.sol";
import { PausableUpgradeable }"@openzeppelin/contracts-upgradeable/utils/PausableUpgradeable.sol";
```

```
import { ReentrancyGuardUpgradeable }"@openzeppelin/contracts-upgradeable/utils/ReentrancyGuardUpgradeable.sol";
import { SafeERC20, IERC20}"@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";
```

Resolution: Fixed in `eb714b9`.

Standalone imports of `Initializable.sol` and `IERC20.sol` were removed from **OMIStaking**. `Initializable` is now a named import from `UUPSUpgradeable.sol`, and `IERC20` is a named import from `SafeERC20.sol`. The remaining OpenZeppelin bases (`Ownable2StepUpgradeable`, `PausableUpgradeable`, `ReentrancyGuardUpgradeable`) use the same named-import form, and the contract still lists `Initializable` in its inheritance list. Function bodies, storage layout, and the external ABI are unchanged by this import refactor.

Disclaimers

Hacken Disclaimer

The smart contracts given for audit have been analyzed based on best industry practices at the time of the writing of this report, with cybersecurity vulnerabilities and issues in smart contract source code, the details of which are disclosed in this report (Source Code); the Source Code compilation, deployment, and functionality (performing the intended functions).

The report contains no statements or warranties on the identification of all vulnerabilities and security of the code. The report covers the code submitted and reviewed, so it may not be relevant after any modifications. Do not consider this report as a final and sufficient assessment regarding the utility and safety of the code, bug-free status, or any other contract statements.

While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only — we recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contracts.

English is the original language of the report. The Consultant is not responsible for the correctness of the translated versions.

As part of Hacken's ongoing quality assurance process, we may conduct re-audits of select projects. These re-audits are performed independently from the original audit and are intended solely for internal quality control and improvement. Updated reports resulting from such re-audits will be shared privately with the respective clients and may be published on the Hacken website only with their explicit consent.
The sole authoritative source for finalized and most up-to-date versions of all reports remains the Audits section at https://hacken.io/audits/.

Technical Disclaimer

Smart contracts are deployed and executed on a blockchain platform. The platform, its programming language, and other software related to the smart contract can have vulnerabilities that can lead to hacks. Thus, the Consultant cannot guarantee the explicit security of the audited smart contracts.

Appendix 1. Definitions

Severities

When auditing smart contracts, Hacken is using a risk-based approach that considers **Likelihood**, **Impact**, **Exploitability** and **Complexity** metrics to evaluate findings and score severities.

Reference on how risk scoring is done is available through the repository in our Github organization:

hknio/severity-formula

Severity	Description
Critical	Critical vulnerabilities are usually straightforward to exploit and can lead to the loss of user funds or contract state manipulation.
High	High vulnerabilities are usually harder to exploit, requiring specific conditions, or have a more limited scope, but can still lead to the loss of user funds or contract state manipulation.
Medium	Medium vulnerabilities are usually limited to state manipulations and, in most cases, cannot lead to asset loss. Contradictions and requirements violations. Major deviations from best practices are also in this category.
Low	Major deviations from best practices or major Gas inefficiency. These issues will not have a significant impact on code execution.

Potential Risks

The "Potential Risks" section identifies issues that are not direct security vulnerabilities but could still affect the project's performance, reliability, or user trust. These risks arise from design choices, architectural decisions, or operational practices that, while not immediately exploitable, may lead to problems under certain conditions. Additionally, potential risks can impact the quality of the audit itself, as they may involve external factors or components beyond the scope of the audit, leading to incomplete assessments or oversight of key areas. This section aims to provide a broader perspective on factors that could affect the project's long-term security, functionality, and the comprehensiveness of the audit findings.

Appendix 2. Scope

The scope of the project includes the following smart contracts from the provided repository:

Scope Details	
Repository	https://github.com/ecomi-admin/omistaking-audit
Commit	34559f44d3704c4e927db58ede4bf76cd8bf8e21
Final Commit	253edbc68687a099c282a220a34299286c9cb39b
Whitepaper	-
Requirements	./docs; NatSpec
Technical Requirements	./docs; NatSpec

Asset	Type
OMIStaking_v5.sol [github.com/ecomi-admin/omistaking-audit]	Smart Contract

Appendix 3. Additional Valuables

Additional Recommendations

The smart contracts in the scope of this audit could benefit from the introduction of automatic emergency actions for critical activities, such as unauthorized operations like ownership changes or proxy upgrades, as well as unexpected fund manipulations, including large withdrawals or minting events. Adding such mechanisms would enable the protocol to react automatically to unusual activity, ensuring that the contract remains secure and functions as intended.

To improve functionality, these emergency actions could be designed to trigger under specific conditions, such as:

- Detecting changes to ownership or critical permissions.
- Monitoring large or unexpected transactions and minting events.
- Pausing operations when irregularities are identified.

These enhancements would provide an added layer of security, making the contract more robust and better equipped to handle unexpected situations while maintaining smooth operations.

Frameworks and Methodologies

This security assessment was conducted in alignment with recognised penetration testing standards, methodologies and guidelines, including the NIST SP 800-115 – Technical Guide to Information Security Testing and Assessment, and the Penetration Testing Execution Standard (PTES), These assets provide a structured foundation for planning, executing, and documenting technical evaluations such as vulnerability assessments, exploitation activities, and security code reviews. Hacken's internal penetration testing methodology extends these principles to Web2 and Web3 environments to ensure consistency, repeatability, and verifiable outcomes.