CHECKLIST

Checklist: Hardening ML Model APIs Against Adversarial Attacks

A practical checklist for securing machine learning model APIs with layered controls for authentication, validation, rate limiting, output shaping, monitoring, and production readiness.

Checklist: Hardening ML Model APIs Against Adversarial Attacks

Use this checklist to assess and improve the security posture of an ML model API before exposing it to untrusted or production traffic.

---

1) Define the trust boundary

- [ ] Identify every client that can call the API.
- [ ] Classify callers as internal, partner, tenant, or public.
- [ ] Document what the API is allowed to do and what it must never do.
- [ ] Confirm whether the model influences security-sensitive decisions.
- [ ] Identify downstream systems that consume model outputs.
- [ ] Record acceptable failure modes and safe fallback behavior.

2) Minimize exposed surface area

- [ ] Remove unused endpoints, debug routes, and test hooks.

- [ ] Disable verbose stack traces in production.
- [ ] Avoid returning raw logits, embeddings, internal feature values, or full confidence vectors unless required.
- [ ] Replace detailed error messages with generic client-safe responses.
- [ ] Limit response fields to the minimum required by the business use case.
- [ ] Ensure explanations or reason codes are only exposed when strictly necessary.

3) Enforce authentication and authorization

- [ ] Require authentication for all non-demo endpoints.
- [ ] Use service-to-service credentials for internal callers.
- [ ] Apply tenant-scoped authorization where multi-tenancy exists.
- [ ] Validate that each caller may access only the models and data it is permitted to use.
- [ ] Rotate credentials and API keys on a defined schedule.
- [ ] Revoke unused credentials and stale access paths.

4) Apply request validation before inference

- [ ] Validate schema, data types, and required fields.
- [ ] Enforce field length, format, range, and encoding checks.
- [ ] Reject oversized payloads.
- [ ] Block unsupported fields instead of silently accepting them.
- [ ] Check for nulls, NaNs, infinities, and malformed values where relevant.
- [ ] Validate feature completeness before scoring.
- [ ] Quarantine or reject suspicious inputs rather than auto-correcting them blindly.
- [ ] Add model-aware validation for text, images, tabular features, or embeddings.

5) Reduce abuse with rate and quota controls

- [ ] Configure request rate limits appropriate to normal usage.
- [ ] Add burst limits and concurrency limits.
- [ ] Apply per-key, per-IP, and per-tenant throttles where appropriate.
- [ ] Set daily or hourly quotas for expensive inference paths.
- [ ] Separate limits for authentication failures, validation failures, and successful requests.
- [ ] Detect repeated probing patterns and trigger stricter controls.

6) Protect the service from resource exhaustion

- [ ] Set request and response size limits.
- [ ] Enforce inference timeouts.
- [ ] Add queue depth limits.
- [ ] Use circuit breakers for slow or failing dependencies.
- [ ] Fail closed when preprocessing or model execution exceeds safe thresholds.
- [ ] Prevent a small number of abusive requests from consuming all CPU, GPU, or memory capacity.

7) Shape outputs to limit leakage

- [ ] Return coarse categories or risk bands when sufficient.
- [ ] Round or bucket scores if exact precision is unnecessary.
- [ ] Avoid exposing intermediate reasoning that can help attackers iterate.
- [ ] Remove sensitive metadata from responses.
- [ ] Keep error responses consistent and non-revealing.
- [ ] Ensure output formatting does not reveal internal system behavior.

8) Add monitoring and anomaly detection

- [ ] Log request metadata, caller identity, response type, latency, and rejection reason.
- [ ] Record repeated failures, unusual payload sizes, and high-frequency query patterns.
- [ ] Monitor score distributions and output drift.
- [ ] Alert on spikes in validation errors, timeouts, or authentication failures.
- [ ] Track suspicious use of confidence values, embeddings, or explanations.
- [ ] Integrate anomaly detection for traffic and output behavior.
- [ ] Retain logs long enough to support investigation and incident response.

9) Test against realistic adversarial patterns

- [ ] Send malformed payloads and verify safe rejection.
- [ ] Test oversized requests and ensure limits trigger correctly.
- [ ] Repeatedly query boundary cases to check for probing resistance.
- [ ] Attempt high-frequency requests to validate throttling.
- [ ] Verify that responses do not reveal more detail after repeated failures.
- [ ] Simulate extraction attempts using systematic variation of inputs.
- [ ] Test denial-of-service conditions in a controlled environment.
- [ ] Validate behavior for prompt injection or tool-abuse scenarios if the system uses agents or tools.

10) Verify production safeguards

- [ ] Confirm authentication, rate limits, and quotas are enabled in production.
- [ ] Confirm logging and alerting are active before launch.
- [ ] Review rollback and disablement procedures.
- [ ] Verify safe fallback behavior if the model or dependencies fail.
- [ ] Document escalation paths for security incidents.
- [ ] Run a pre-production checklist review with engineering and security owners.
- [ ] Validate that any training-time hardening is paired with runtime controls.

11) Review model-specific risk factors

- [ ] Identify whether the model is exposed to untrusted public input.
- [ ] Determine whether the output affects access control, fraud, moderation, or safety decisions.
- [ ] Confirm whether confidence scores or embeddings are exposed.
- [ ] Assess whether the service can be queried at scale.
- [ ] Check whether training data may be inferable from output behavior.
- [ ] Evaluate whether adversarial training or other robustness techniques are warranted.

12) Decide on the right operating posture

Use a stronger hardening posture when:

- [ ] The API is exposed to untrusted clients.
- [ ] The model output influences security-sensitive decisions.
- [ ] The service can be queried repeatedly at scale.
- [ ] The API returns rich signals such as confidences, embeddings, or explanations.

A lighter posture may be acceptable only when:

- [ ] Access is tightly controlled.
- [ ] The service is fully internal.
- [ ] The model output has low operational impact.
- [ ] The calling application already validates inputs and constrains usage.

Quick pre-launch checklist

- [ ] Access is authenticated.

- [ ] Authorization is tenant-aware.
- [ ] Inputs are validated and bounded.
- [ ] Outputs are minimized.
- [ ] Rate limits and quotas are active.
- [ ] Timeouts and circuit breakers are configured.
- [ ] Logging and anomaly detection are enabled.
- [ ] Abuse tests have been run.
- [ ] Rollback and incident response steps are documented.
- [ ] The team has agreed on safe fallback behavior.

Minimal production standard

If you can only implement a few controls first, prioritize these in order:

- Authentication and authorization
- Strict request validation
- Rate limiting and quotas
- Output minimization
- Monitoring and anomaly detection
- Timeout and circuit breaker controls
- Abuse testing before production exposure

Notes

- Hardening is a layered control strategy, not a single feature.
- The API should limit what attackers can learn, how quickly they can learn it, and how much damage they can do.

- Runtime protections are as important as model robustness.
- When in doubt, reduce detail, reduce frequency, and detect abuse early.