CS-070 · API Security

API Rate Limiting

Controlling how many requests a client can make to protect APIs from abuse, overload, and brute force.

Executive Summary

API rate limiting controls how many requests a given client can make in a period of time, so a single caller cannot overwhelm a service or grind through attacks unchecked. It protects availability, blunts brute force and scraping, and helps control cost. Effective rate limiting combines the right algorithm, clear client keys, sensible limits, and helpful responses when a limit is reached.

What It Is

Rate limiting is a control that caps the number of requests a client may send to an API within a defined window, and throttling slows or queues requests that exceed a threshold. The API tracks usage per identifier, such as an API key, an authenticated user, or an address, and rejects or delays requests once the limit is hit. It is both a reliability control, protecting a service from being overwhelmed, and a security control, making brute force, credential stuffing, and data scraping far slower and more expensive for an attacker. Related mechanisms include quotas, which cap usage over longer periods such as a day or month, and burst allowances, which permit short spikes while holding a steady average.

Why It Matters

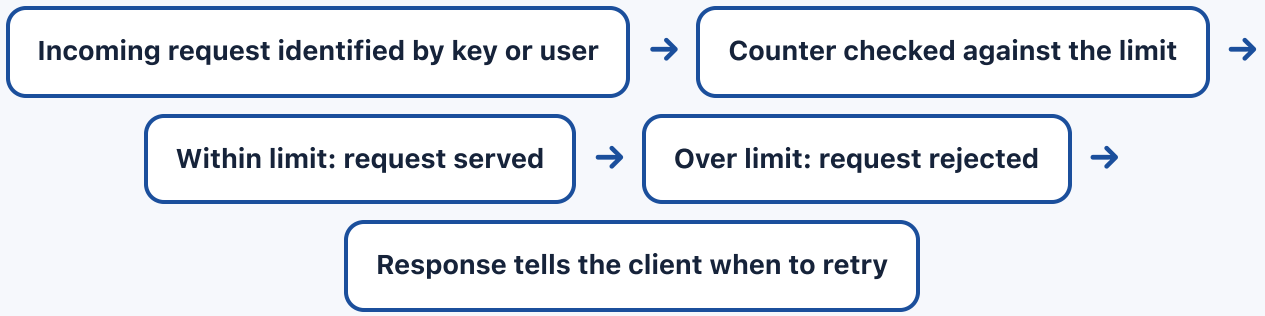
Without rate limiting, a single misbehaving or malicious client can degrade a service for everyone, and an attacker can attempt thousands of password guesses or extract large volumes of data through an otherwise legitimate endpoint. Rate limiting turns these attacks from fast and cheap into slow and costly, which is often enough to stop them. It also protects downstream systems and controls the cost of paid or metered dependencies. For engineers and security

professionals, rate limiting is a foundational defense that appears in the OWASP API Security Top 10, because the absence of it amplifies almost every other API risk, from brute force on authentication to bulk extraction through weak authorization.

How It Works

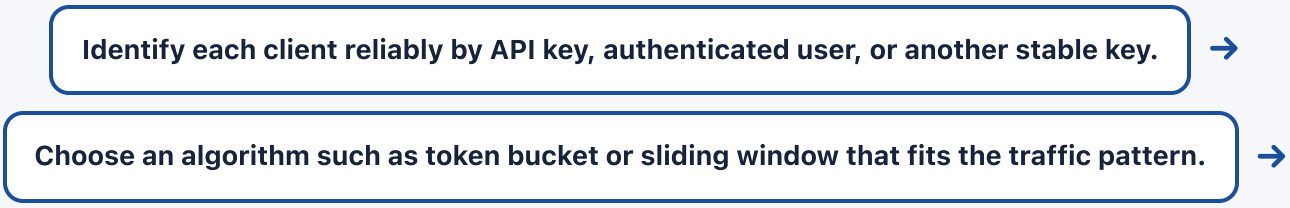
Rate limiting depends on identifying the client and counting its requests against a limit. Several algorithms are common. A fixed window counts requests in set intervals, which is simple but can allow a burst at the edges of two windows. A sliding window smooths this by considering a rolling period. A token bucket gives each client a bucket of tokens that refills at a steady rate, allowing short bursts while enforcing an average, and a leaky bucket processes requests at a constant rate and drops overflow. When a client exceeds its limit, a well-designed API returns a clear status indicating too many requests, along with information telling the client how long to wait before retrying. In distributed systems the counters must be shared across servers, often in a fast central store, so the limit holds no matter which server handles a request. Limits are typically layered: stricter on sensitive endpoints such as login, looser on read-only public data.

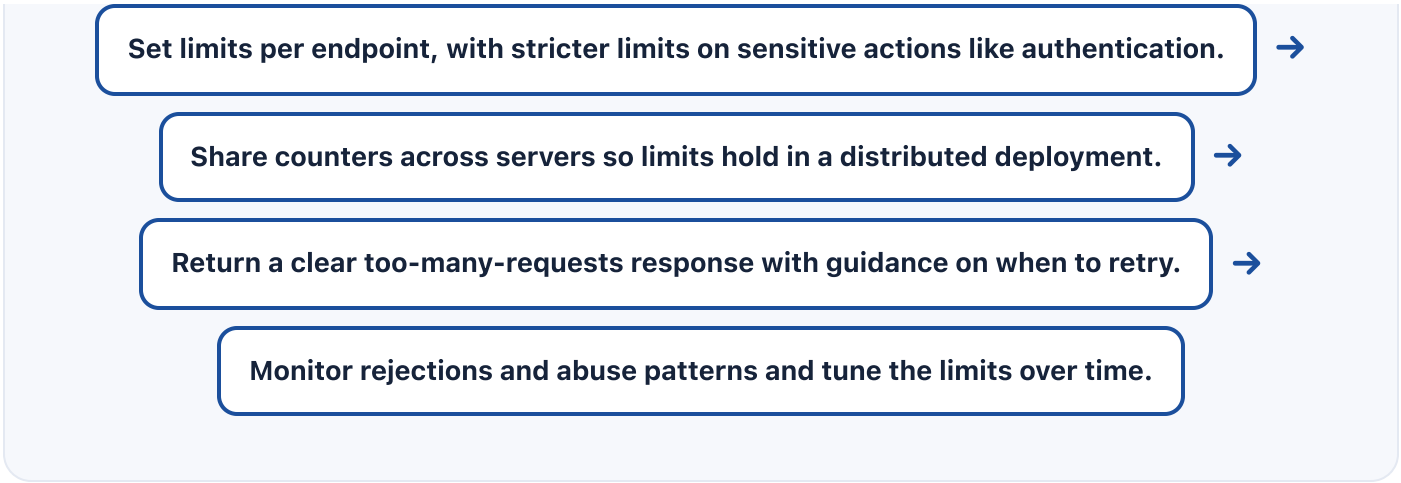
Architecture Diagram



Each request is identified, counted against the client's limit, and either served or rejected with a retry hint.

Visual Workflow





Common Attacks

- Brute force and credential stuffing against login endpoints when limits are missing
- Data scraping that extracts large volumes through a legitimate endpoint
- Denial of service that exhausts capacity with a flood of requests
- Distributing requests across many identities to stay under a per-client limit
- Abusing expensive endpoints to drive up cost or exhaust downstream resources

Common Mistakes

- Applying no limit at all on sensitive endpoints such as login and password reset
- Limiting only by address, which is easy to rotate or shared behind proxies
- Using a single global limit instead of tighter limits on sensitive actions
- Keeping counters per server so the limit is effectively multiplied in a cluster
- Returning an unhelpful error with no retry guidance, breaking well-behaved clients
- Leaking whether an account exists through different responses to rate-limited attempts

Best Practices

- Rate limit every API, with stricter limits on authentication and other sensitive endpoints
- Identify clients by a stable key such as an authenticated user or API key, not address alone
- Choose an algorithm suited to the traffic, such as token bucket for bursty clients
- Enforce limits in a shared store so they hold across all servers
- Return the standard too-many-requests status with clear retry guidance

- Combine rate limiting with quotas, monitoring, and account lockout defenses

Quick Checklist

- [x] Rate limits applied to all endpoints, tightest on login and password reset
- [x] Clients identified by a stable key, not address alone
- [x] Appropriate algorithm selected for the traffic pattern
- [x] Counters shared across servers in a central store
- [x] Too-many-requests responses include retry guidance
- [x] Rejections and abuse patterns monitored and limits tuned

Recommended Tools

API gateway
Enforces rate limits and quotas centrally in front of services

Reverse proxy
Applies request throttling at the network edge

In-memory data store
Holds shared counters for fast, distributed limit checks

Web application firewall (WAF)
Detects and blocks abusive request patterns

Industry Standards

OWASP API Security Top 10
Names unrestricted resource consumption and lack of rate limiting as key API risks

IETF guidance on the too-many-requests status
Defines the standard response for exceeding a rate limit

NIST SP 800-63B
Recommends throttling and rate limiting to resist online guessing of credentials

Career Relevance

Rate limiting is a routine responsibility for backend engineers, API security engineers, and application security engineers who design resilient, abuse-resistant endpoints. Security engineers use it to defend authentication and reduce scraping, and reliability and platform teams rely on it to protect availability. Understanding it is also useful for AI governance professionals, since rate limits protect AI service endpoints from abuse and runaway cost.

Interview Questions

- Why is rate limiting both a reliability control and a security control?
- Compare the token bucket and sliding window approaches to rate limiting.
- How would you rate limit a login endpoint to slow brute force without harming real users?
- Why is limiting by address alone often insufficient?
- How do you enforce a consistent rate limit across many servers?

Related Certifications

OWASP-aligned application security training

CompTIA Security+ (foundational)

Cloud and platform engineering certifications (vendor-neutral concepts)

Further Reading

- OWASP API Security Project
- NIST SP 800-63B Digital Identity Guidelines
- OWASP Cheat Sheet Series

Key Takeaways

- Rate limiting caps how many requests a client can make in a period of time.
- It protects availability and slows brute force, scraping, and abuse.
- Apply the tightest limits to sensitive endpoints such as login.
- Identify clients by a stable key and share counters across all servers.

- Return the standard too-many-requests status with clear retry guidance.

FAQ

Related Careers

RELATED ROLES

Application Security Engineer

Api Security Engineer

Backend Engineer

Security Engineer

RELATED CERTIFICATIONS

OWASP-aligned application security training

CompTIA Security+ (foundational)

Cloud and platform engineering certifications (vendor-neutral concepts)

CURRENT OPENINGS

Live roles are on the job board.

Browse all jobs

GUIDES & SALARY

Career guides

Role roadmaps

Salary data

Career tools

SUGGESTED LEARNING PATH

1. Ground the basics with CS-001 Cybersecurity
2. Study this sheet: **API Rate Limiting**
3. Go deeper: API Gateways
4. Go deeper: REST API Security

5. Validate it: work toward **OWASP-aligned application security training**
6. Find the role: browse current openings

RELATED SHEETS

CS-071 API Gateways | CS-066 REST API Security | CS-068 OAuth 2.0 | CS-062 Authentication | CS-057 OWASP Top 10