

Forum:[Hack Forums](#)

Thread: [IDENTIFYING DoS-VULNERABLE ENDPOINTS](#)

Vinavil - January 12, 2026, 7:15 am

????????????????????????????????????

IDENTIFYING DoS-VULNERABLE ENDPOINTS

A Red Team Perspective

????????????????????????????????????

Guide by Vinavil

So, What's This All About?

When we talk about Denial of Service in web applications, we're not talking about flooding a server with millions of packets. That's old school. What we're looking for are those sneaky endpoints where a tiny request causes massive headaches server side. It's about finding the sweet spot where minimal effort translates into maximum server pain.

Unlike typical vulnerabilities that steal data or break into systems, DoS goes after availability. For most businesses, if their service is down, nothing else matters. That's what makes these findings so impactful during assessments.

What Actually Makes an Endpoint Vulnerable?

An endpoint becomes DoSable when there's a mismatch between what you put in and what the server has to do. We call this asymmetry, and it shows up in a few ways:

Computational Asymmetry You send something simple, the server does something expensive. Think regex matching gone wrong, heavy database queries, or anything involving crypto operations. Your request takes milliseconds to craft, but the server chews on it for seconds.

Bandwidth Asymmetry This is your classic amplification scenario. Small request in, massive response out. We'll dig deeper into this one because it's probably the most common and easiest to spot.

State Asymmetry The server has to hold onto resources like memory, connections, or file handles for your request. Meanwhile, you can just fire and forget. The server's stuck babysitting your abandoned requests.

Temporal Asymmetry You can spam requests instantly, but each one takes forever to process. Eventually the queue backs up and everything grinds to a halt.

Phase 1: Finding the Juicy Targets

Where Should You Look First?

Not every endpoint is worth your time. Here's where the good stuff usually hides:

? Search and Filter Functions

These are goldmines. Anytime you see a search box or filtering options, there's a database query behind it that you can potentially abuse. Look for anything accepting wildcards, sorting parameters, or letting you control how much data comes back. GraphQL endpoints are especially fun here because they're so flexible by design.

? Export and Report Generators

PDF exports, CSV downloads, report compilation. These are all computationally expensive. The server has to gather data, format it, maybe render it, and then send you a file. Lots of moving parts means lots of potential for abuse.

? File Processing

Anything that takes your upload and does something with it. Image resizing, document parsing, archive extraction. The beautiful thing here is that file complexity often matters more than file size. A tiny file can still wreck a server if it's crafted right.

? Auth and Crypto Stuff

Password hashing is designed to be slow. That's the whole point for security. But it also means registration and password reset endpoints can be abused. Token generation and certificate validation fall into the same bucket.

? Aggregation Endpoints

Dashboards that pull stats from everywhere, analytics that crunch numbers, batch operations that touch multiple systems. One request to these can fan out to dozens of backend calls.

Do Your Homework First

Before you start poking around, gather intel passively. Dig through JavaScript files for API patterns. Check response headers for rate limiting clues like X-RateLimit headers or Retry-After values. Poke at robots.txt and sitemaps for hidden endpoints.

If there's API documentation available, read it. Swagger specs are basically a roadmap telling you which endpoints lack auth, support batch operations, or have loose parameter validation.

Phase 2: Measuring the Amplification

This is where we get quantitative. There are two key numbers you want to track:

Bandwidth Amplification = Response Size ÷ Request Size

This tells you how efficiently your bandwidth converts into server resource consumption. A 50-byte request returning 500KB gives you 10,000x.

Computational Amplification = Server Processing Time ÷ Request Time

This captures the CPU cost. A request taking 100ms to send but forcing 10 seconds of processing gives you 100x computational amplification. Some endpoints return a tiny response but burn serious CPU getting there. If you only measure bytes, you'll miss these.

Both numbers matter. An endpoint returning a 200-byte error after 10 seconds of CPU time looks boring through the bandwidth lens but is devastating through the computational one. Always measure both.

How to Approach This

For each endpoint, measure everything. Request size including headers, response size, response time. Do this multiple times with different inputs to get a real picture.

What Do the Numbers Mean?

Here's a rough guide for interpreting bandwidth amplification:

1-5x Pretty normal. Most simple APIs fall here. Not exciting.

10-100x Now we're talking. Usually means list endpoints without proper pagination.

50-500x Getting spicy. Often search functions with wildcard support.

100-10,000x Jackpot territory. Export functions and download endpoints love to live here.

Unlimited GraphQL without query limits. The gift that keeps on giving.

And for computational amplification:

1-10x Normal range. Most endpoints process quickly.

10-100x Worth noting. Could tie up worker threads with moderate request volume.

100-1000x Dangerous. A handful of concurrent requests can saturate a thread pool.

1000x+ Critical. Single requests can cause visible degradation.

Don't Forget About Algorithmic Complexity

Beyond raw timing, watch for non-linear scaling. Try different inputs. Simple stuff, long strings, wildcards, regex patterns if they're supported. If you notice that doubling your input length quadruples the response time, you've probably found an algorithmic complexity issue. Those are beautiful because they don't even need many requests to cause problems.

Phase 3: The Vulnerability Patterns You'll Actually Find

ReDoS: When Regex Goes Wrong

Regular expressions can be dangerous beasts. When developers write patterns with nested quantifiers or overlapping alternatives, they create potential for catastrophic backtracking. The regex engine gets stuck exploring exponentially many paths through the pattern.

The telltale signs: you send a carefully crafted string and the response time explodes. Sometimes timeouts, sometimes errors. Classic vulnerable patterns include `(a+)+` or `(a|aa)+`. Anything where the engine can match the same input multiple ways.

Test any field that might use regex. Usernames, emails, search boxes, comment fields. Send strings designed to trigger backtracking and watch those response times.

Database Query Abuse

Databases are wonderful targets because they're doing the heavy lifting for most web apps.

Missing Indexes

Compare response times for queries returning lots of results versus almost nothing. If they take roughly the same time, the database is doing full table scans. That's a missing index, and every query is expensive regardless of results.

Wildcard Shenanigans

Leading wildcards are your friend. A search for `%something%` can't use indexes in most databases, forcing a full scan. Compare that to `something%` which uses indexes efficiently. If the endpoint lets you control wildcard placement, you've got a lever to pull.

Pagination Abuse

Huge offset values are brutal on databases. Page 100,000 means the database has to find and skip 99,999 pages of data first. Combine with sorting on non-indexed columns for maximum pain.

JOIN Explosions

Tables joined without proper constraints produce Cartesian products where row counts multiply. Any endpoint correlating data from multiple sources is worth investigating.

GraphQL: A Playground for DoS

GraphQL is amazing for developers but a nightmare security-wise. "Ask for exactly what you need" also means "ask for way more than the server can handle."

Depth Attacks

Got a user type with a friends field that returns more users? Ask for `user -> friends -> friends -> friends` fifty levels deep. Each level multiplies database queries. Without depth limits, this spirals fast.

Field Spam

Nothing stops you from requesting the same field a hundred times in one query. Each might trigger separate processing. Worth trying.

Alias Multiplication

The sneaky one. GraphQL aliases let you run the same expensive query multiple times in a single request by giving each a different name. Find an expensive resolver and alias it fifty times.

Introspection Queries

Built-in schema introspection can produce massive responses describing the entire API. Often overlooked when setting up rate limits because they seem like developer tools.

File Processing Nightmares

Compression Bombs

A zip file can have insane compression ratios. A 1MB file expanding to multiple gigabytes. If the server extracts uploads without checking decompressed size, you can fill disk or memory with one upload.

Pixel Bombs

Images have two sizes that matter: file size and dimensions. When decoded, memory usage depends on width × height × color depth. A tiny file with ridiculous dimensions can consume gigabytes of RAM when processed.

Malicious Documents

PDFs, Word docs, and similar formats can contain external references that slow things down, deeply nested structures that break parsers, embedded scripts. If an app processes these, test with adversarial samples.

XML and JSON Parsing Issues

The Billion Laughs

This classic XML attack uses entity expansion to turn a few kilobytes into gigabytes. You define entities that reference other entities recursively, and boom. Exponential expansion. Most modern parsers ship with entity expansion limits enabled by default, so this is rarely exploitable on its own. That said, the broader XML-based attack surface is far from dead. WordPress is still vulnerable to XML-RPC call spam, and it powers roughly 40% of the web. Legacy systems and custom XML handling still show up more than you'd expect. Worth testing, but temper your expectations on modern stacks.

External Entities for DoS

Beyond data theft, XXE can cause DoS by referencing slow external resources, huge local files, or creating infinite loops.

Nesting Depth

Both XML and JSON parsers have limits on nesting depth. Push past those limits and you might crash the parser or exhaust memory. Test with increasingly deep structures to find the breaking point.

HTTP/2 Rapid Reset (CVE-2023-44487)

Arguably the biggest application-layer DoS development in recent years. HTTP/2 lets you open

multiple streams on a single connection. The Rapid Reset attack opens a stream and immediately sends a RST_STREAM frame to cancel it. The server still has to allocate resources to process the initial request before it gets the cancellation. Do this thousands of times per second on a single connection and you overwhelm a server while generating minimal network traffic.

This affected nearly every major web server. Nginx, Apache, IIS, Go's net/http, Node.js. The asymmetry is brutal: the client barely uses any resources while the server burns through CPU processing and cancelling streams it never actually serves.

When testing, check if the target has been patched and what HTTP/2 concurrent stream limits are in place. Many servers now cap the rate of stream resets per connection, but misconfigurations exist. The same concept applies to HTTP/3 and QUIC, though implementations vary.

DoS in Async and Event-Loop Architectures

The thread saturation model works for traditional threaded servers like Apache or Tomcat. But Node.js, Go, and async Python (FastAPI, aiohttp) use event loops or goroutines instead of thread-per-request. You can't exhaust a thread pool in the traditional sense, so the attack patterns shift.

Event Loop Blocking (Node.js)

Node runs on a single thread. Any synchronous operation that takes too long blocks everything. CPU-intensive JSON parsing, complex regex, or synchronous crypto will freeze the entire server for all users. One crafted request can stall every connection. Arguably worse than thread exhaustion because there's zero isolation between requests.

Goroutine Exhaustion (Go)

Go can spin up millions of goroutines, but each consumes memory. Trigger endpoints that hold goroutines open (slow reads, long-lived connections, hanging upstream calls) and you exhaust memory instead of a thread count. The server won't hit a concurrency limit, it'll just OOM.

Worker Pool Starvation (Async Python)

Frameworks like FastAPI offload blocking operations to thread pools. If the app runs synchronous database calls or file I/O in the async path, or the thread pool is undersized, you can starve it the same way you'd starve a traditional server.

The concurrency model determines what "exhaustion" looks like. Threaded servers: saturate threads. Event-loop: block the loop. Goroutines: consume memory. Identify the architecture during recon so you know which model applies. Check the Server header, look at response patterns, and test how the server handles slow clients. An event-loop server that degrades uniformly when you send one expensive request is a dead giveaway.

Phase 4: Dealing With Rate Limits

Finding Out What's Already There

First things first. Check response headers for the usual suspects. X-RateLimit-Limit tells you the maximum requests allowed. X-RateLimit-Remaining shows what's left in your window. X-RateLimit-Reset indicates when you get a fresh start. Retry-After tells you how long to wait.

No headers doesn't necessarily mean no limits, but it's a good sign limiting might be weak or absent. Test behaviorally by ramping up request rates until you hit 429 responses or notice degraded performance.

Understanding How Limits Work

Different implementations have different weaknesses:

IP-based Tracks by your source IP. Easy to test, potentially bypassable depending on configuration.

User-based Only kicks in when you're logged in. Unauthenticated requests might be free game.

Token Bucket Lets you burst initially, then throttles. That initial burst might be all you need.

Sliding Window More consistent limiting. Harder to game with timing tricks.

Per-Endpoint Different limits for different routes. Some endpoints might be unprotected.

Bypassing Rate Limits

Lots of rate limiting implementations have holes, but effectiveness varies depending on the target's infrastructure.

? Header Spoofing (Situational)

If the backend trusts forwarded headers, you might be able to fake your source. Try X-Forwarded-For, X-Real-IP, X-Originating-IP, X-Client-IP, CF-Connecting-IP, True-Client-IP. Rotate through different values. However, modern CDNs and reverse proxies like Cloudflare, AWS CloudFront, and properly configured nginx strip or override these headers before your request reaches the app. This works against misconfigured setups and internal tools more than production infrastructure behind major providers. Always test it, but don't count on it.

? URL Tricks

Rate limiters have to normalize URLs to count requests properly, and they often do it wrong. Try trailing slashes, empty query strings, fragment identifiers, dot segments like `./` in the path, different casing, null bytes, or URL-encoded characters. Each variation might be counted separately.

? Method Switching

Some limiters only track specific HTTP methods. If GET is rate limited, try POST, PUT, PATCH, HEAD, OPTIONS. The same resource might be accessible through multiple methods with independent counters.

? Parameter Games

Adding random query parameters might make requests look unique to the limiter. Cache busters, version numbers, format specifiers. Anything that changes the URL signature.

? Session Rotation

If limits are tied to sessions, generate fresh ones. Clear cookies, grab new session IDs, look like a new visitor each time.

? Protocol Differences

HTTP/1.1, HTTP/2, and HTTP/3 might have separate limiting configurations. WebSocket connections to the same resource might bypass HTTP limits entirely.

Phase 5: Understanding Amplification In Depth

Why Amplification Matters So Much

Amplification is the whole game in application-layer DoS. It lets someone with limited resources take down something much bigger. The flavors:

? Bandwidth Amplification

An endpoint where a 50-byte request generates a 500KB response gives you 10,000x. In application-layer DoS, that response comes back to you, not to a third party like in network-layer reflection attacks. The value is about server resource consumption per connection: bandwidth, memory for buffering, and connection slots.

? Computational Amplification

This isn't about bytes. It's about CPU cycles. A request that takes 100ms to send but 10 seconds to process gives you 100x computational amplification. You can keep a server's workers completely saturated while barely breaking a sweat. For event-loop servers, even a single request hitting this ratio can degrade service for everyone.

? Storage Amplification

Requests that cause disk writes amplify storage consumption. Log injection, temporary file creation, cache pollution. All eat up server storage disproportionate to what you're sending.

? Chained Amplification

When applications make external requests in response to yours: webhooks, API calls, DNS lookups. One request triggering ten backend calls gives you another multiplier on top of everything else.

Running the Numbers

To figure out how feasible an attack would be, think about:

Worker Saturation How many workers/threads/connections can the server handle (guess if you have to), divided by average response time, gives you requests per second needed to exhaust capacity. Remember to account for the server's concurrency model.

Bandwidth Math Response size divided by request size tells you how efficiently your bandwidth translates to server resource consumption per connection.

Cost Analysis What would it cost an attacker in cloud resources versus what it costs the defender? Highly asymmetric costs favor the attacker.

Wrapping Up

Finding DoS vulnerabilities is about understanding asymmetry. Where can you invest a little and make the server invest a lot? Map out interesting targets, measure amplification factors (both bandwidth and computational), look for known vulnerability patterns, test rate limits and try to bypass them, then calculate real-world impact.

The key things to keep in mind:

Always calculate both amplification factors. Bandwidth and computational. They're your north star.

Identify the server's concurrency model early. Thread-based, event-loop, or goroutine-based changes what exhaustion looks like.

Rate limits exist but they're often weaker than they look. Bypass techniques are situational though, especially header spoofing.

GraphQL deserves special attention because flexibility is a double-edged sword.

File processing and export functions are high-value targets.

Don't sleep on protocol-level attacks like HTTP/2 Rapid Reset.

Numbers matter more than gut feelings when assessing impact.

Think about theoretical impact to understand how much damage you can actually cause.

Now go find some vulnerable endpoints.

Chorozon - January 12, 2026, 7:27 am

Interesting post , good work OP

Vinavil - January 12, 2026, 7:39 am

(Jan 12, 2026 08:27 AM)Chorozon Wrote: [\[CITE\]](#) Interesting post , good work OP
Tysm :)

s1r10us - January 27, 2026, 9:57 am

Very nice post. Thanks

Vinavil - January 27, 2026, 2:18 pm

(Jan 27, 2026 10:57 AM)s1r10us Wrote: [\[CITE\]](#) Very nice post. Thanks
no issue, if you have any specific question you can ask, i'll be glad to help!

passerino - February 4, 2026, 5:57 pm

(Jan 27, 2026 10:57 AM)s1r10us Wrote: [\[CITE\]](#) Very nice post. Thanks
thank
so if one finds GraphQL exposed in a site is it a good sign??

Vinavil - February 5, 2026, 3:11 am

(Feb 4, 2026 06:57 PM)passerino Wrote: [\[CITE\]](#) thank
so if one finds GraphQL exposed in a site is it a good sign??
it's a good sign in the sense that it gives you something to poke at, but it's not a vuln by itself. what you wanna look for is stuff like introspection being enabled, no query depth limits, no rate limiting,

batching allowed without restrictions etc. that's what actually makes it exploitable. if it's properly hardened then there's not much you can do with it. so yeah good starting point but it all depends on how it's configured

WinGV - February 5, 2026, 12:23 pm

Hey Vinavil, solid write-up overall, enjoyed reading through it. The asymmetry framework is a great way to structure this and the GraphQL section in particular is on point. Just wanted to share a few things that caught my eye in case you want to tighten it up:

The amplification formula could be a tad more complete. You cover computational cost well throughout the article (hashing, ReDoS, query abuse), but the actual formula you give is purely bandwidth-based. Someone following it literally might dismiss an endpoint that returns a 200-byte error after 10 seconds of CPU time because the size ratio looks boring. Maybe worth defining a second formula for time-based amplification, or just broadening the definition so it matches the rest of the content.

The bandwidth math example is a bit off. You mention needing only 100Kbps to generate 1Gbps against a target, but that model applies to reflected amplification (DNS/NTP reflection etc.) where a third party sends the response to the victim. In app-layer DoS, that fat response comes back to you, not to someone else. Might confuse people since you specifically said this isn't about network-layer flooding.

Header spoofing for rate limit bypass is getting less reliable every year. It still works against misconfigured setups, sure, but Cloudflare, AWS CloudFront, modern nginx configs all strip or override X-Forwarded-For and friends before your request hits the app. Worth mentioning that this is situational rather than presenting it as a go-to technique.

Billion Laughs is mostly a historical footnote at this point. Pretty much every modern XML parser ships with entity expansion limits on by default now. Still worth a test, but calling it something people would "be surprised" to find exploitable is a stretch in 2026.

Might be worth adding a section on HTTP/2 Rapid Reset (CVE-2023-44487). It was arguably the biggest app-layer DoS development in recent years, affected nearly every major web server, and fits perfectly with your theme of small effort / big impact. Feels like a missing piece.

Same goes for async/event-loop architectures (Node, Go, async Python). The thread saturation model you describe works for traditional threaded servers, but modern stacks handle concurrency differently and the attack patterns change with them.

None of this takes away from the core content, the methodology is sound and the vulnerability patterns section is very useful. Just some tweaks that would make it more accurate for today's landscape that goes along with your initial post. Nice work!

Vinavil - February 6, 2026, 5:44 am

(Feb 5, 2026 01:23 PM)WinGV Wrote: [\[CITE\]](#) Hey Vinavil, solid write-up overall, enjoyed reading through it. The asymmetry framework is a great way to structure this and the GraphQL section in particular is on point. Just wanted to share a few things that caught my eye in case you want to tighten it up:

The amplification formula could be a tad more complete. You cover computational cost well throughout the article (hashing, ReDoS, query abuse), but the actual formula you give is purely bandwidth-based. Someone following it literally might dismiss an endpoint that returns a 200-byte error after 10 seconds of CPU time because the size ratio looks boring. Maybe worth defining a

second formula for time-based amplification, or just broadening the definition so it matches the rest of the content.

The bandwidth math example is a bit off. You mention needing only 100Kbps to generate 1Gbps against a target, but that model applies to reflected amplification (DNS/NTP reflection etc.) where a third party sends the response to the victim. In app-layer DoS, that fat response comes back to you, not to someone else. Might confuse people since you specifically said this isn't about network-layer flooding.

Header spoofing for rate limit bypass is getting less reliable every year. It still works against misconfigured setups, sure, but Cloudflare, AWS CloudFront, modern nginx configs all strip or override X-Forwarded-For and friends before your request hits the app. Worth mentioning that this is situational rather than presenting it as a go-to technique.

Billion Laughs is mostly a historical footnote at this point. Pretty much every modern XML parser ships with entity expansion limits on by default now. Still worth a test, but calling it something people would "be surprised" to find exploitable is a stretch in 2026.

Might be worth adding a section on HTTP/2 Rapid Reset (CVE-2023-44487). It was arguably the biggest app-layer DoS development in recent years, affected nearly every major web server, and fits perfectly with your theme of small effort / big impact. Feels like a missing piece.

Same goes for async/event-loop architectures (Node, Go, async Python). The thread saturation model you describe works for traditional threaded servers, but modern stacks handle concurrency differently and the attack patterns change with them.

None of this takes away from the core content, the methodology is sound and the vulnerability patterns section is very useful. Just some tweaks that would make it more accurate for today's landscape that goes along with your initial post. Nice work!

Hey, really appreciate the thorough feedback.

You're right that the amplification formula only looks at bandwidth. I talk about computational cost throughout the piece but never actually pin it down the same way. Adding a simple time-based ratio alongside the size-based one would round that out nicely.

And yeah, the bandwidth math, that 100Kbps to 1Gbps example is reflection logic, which doesn't really apply at the application layer where the heavy response comes back to you, not some third party. I should've framed it around thread and connection saturation, since that's what actually matters here.

On header spoofing, fair, it's definitely less reliable against proper setups these days. I still think it's worth testing though, because you'd be surprised how many apps are sitting behind a misconfigured reverse proxy that just trusts whatever headers it gets. Especially internal tools and smaller deployments. But "situational" is a better way to frame it than making it sound like a default go-to.

Billion Laughs I'm keeping in, but I'll admit the "you'd be surprised" angle is a bit of a stretch. Most modern parsers handle it out of the box. That said, XML-based attack surface isn't totally dead. WordPress still gets hit with XML-RPC call spam, and it runs like 40% of the web. Legacy systems and custom XML handling still show up more than people expect. "Worth testing, rarely exploitable" is probably the more honest take.

HTTP/2 Rapid Reset is a fair call-out, that one fits the asymmetry theme perfectly. Same with the async and event-loop stuff, the thread saturation model I used is pretty old-school and doesn't account for how Node or Go handle concurrency. Both would've been solid additions.

I'm going to go back and work these points into the post. Better to address them directly in the guide than leave them buried in the comments.

Thanks again for the review, this is the kind of feedback that actually makes the work better <3