Forum:Hack Forums

Thread: [Tutorial] Understanding SSRF: Exploitation Mechanics and Prevention

Sift - *September 2, 2026, 2:13 pm*

SSRF: I tested the bypasses everyone posts, and half of them are dead

I've been reading up on SSRF and kept seeing the same list of blocklist bypasses copy-pasted everywhere. Decimal IPs, octal IPs, 127.1, the usual. So I built a small lab to actually run them instead of taking anyone's word for it.

Turns out most of them don't work any more. Here's what does.

SSRF for dummies

Say you're standing outside an office building. There's a keycard door and you don't have a card, so anything inside is off limits to you.

But there's a guy on reception who runs errands. You can walk up and say "go grab me the document at room 300" and off he goes, because he does have a card. He comes back, hands you the document, no questions asked.

Now you say "actually, go grab me whatever is in the server room". And he does. He's not checking whether you were allowed to ask, he's just fetching whatever address you gave him.

That's SSRF. The receptionist is the server. The keycard is being inside the network. You never got past the door, you just found someone who could walk through it for you and would do it on request.

The reason it works on webhooks is that fetching a URL you handed over is literally the receptionist's whole job description.

First, what SSRF actually is

You get the server to make a request somewhere it was never supposed to go.

What makes it dangerous isn't the request, it's where the request comes from. The attacker gains no new powers, they just borrow the server's spot on the network, and from there everything looks trusted. Your firewall is doing its job perfectly. The traffic is simply coming from inside the house.

That spot is worth a lot. From there you can reach internal admin panels, APIs that never bothered with auth because someone decided they were only reachable internally, and on any cloud box, the metadata service that hands out IAM credentials to anyone who asks.

Most modern websites use webhooks, and they're one of the most common places SSRF turns up. Think about what a webhook actually does: you hand it a URL and the server goes and fetches it. That's the feature. It's also the bug. Same deal with link previews, "import from URL", avatar

fetchers and PDF generators.

The vulnerable code

Code const r = await fetch(req.body.url);
That's it, that's the whole vulnerability. You pick the destination, the server brings the privileges.

Stuff worth pointing it at:
169.254.169.254 - cloud metadata. On IMDSv1 that's your IAM credentials, no auth needed.

127.0.0.1:whatever - all the internal services running without auth because they're "local only".

10.x, 192.168.x - port scan the internal network, using their server as the scanner.

file:///etc/passwd - it's not only an HTTP problem. Scheme matters.

Now the interesting part

First thing everyone reaches for is a blocklist:

Code const BLOCKED = ["localhost", "127.0.0.1", "169.254.169.254"];
if (BLOCKED.includes(new URL(url).hostname)) return res.status(403).end();
Then you go find a cheatsheet and it tells you to bypass it with encoded IPs. I set up three endpoints (vulnerable, blocklist, properly fixed) pointed at a fake internal service holding a secret, and threw everything at all three:

Code payload	vulnerable	blocklist	safe
plain loopback	LEAKED	blocked	blocked
hostname alias	LEAKED	blocked	blocked
decimal-encoded IP	LEAKED	blocked	blocked
octal-encoded IP	LEAKED	blocked	blocked
short form 127.1	LEAKED	blocked	blocked
DNS -> 127.0.0.1	LEAKED	LEAKED	blocked
302 redirect inward	LEAKED	LEAKED	blocked
legitimate request	ok	ok	ok

Look at the blocklist column, because all the encoding tricks got stopped. Not because the blocklist is any good, but because the URL parser quietly cleans them up before your check ever runs:

Code http://2130706433/ -> hostname: 127.0.0.1
http://0177.0.0.1/ -> hostname: 127.0.0.1
http://127.1/ -> hostname: 127.0.0.1
Every modern parser does this, whether that's Node, browsers, Go or Python, so by the time your code reads hostname it has already been turned back into 127.0.0.1.

Those payloads only still land on code running a regex against the raw string instead of parsing it properly, which is the real lesson hiding in there: parse it first, then check the parsed value. Never

pattern match on raw user input.

What actually got through

Two things, and both are invisible to a string check, which is exactly the point.

A normal-looking hostname that resolves to somewhere internal

Code http://localtest.me/ -> resolves to 127.0.0.1
Nothing about that string looks wrong, and that's the problem. Your blocklist compares text while DNS decides where the packet actually goes. Anyone can register a domain and point it at 127.0.0.1, or 10.0.0.5, or your metadata endpoint, and there's nothing in the hostname to give it away.

A redirect

Code http://attacker.com/go -> 302 -> http://169.254.169.254/latest/meta-data/
You validated the first URL and it passed. Nothing then looked at where it went afterwards, and since fetch follows redirects by default, off it goes.

That's the trouble with blocklists generally: they can only stop what someone remembered to write down, and there are always more ways to say "127.0.0.1" than anyone lists. The worse problem is that the endpoint now looks handled, so nobody goes back to it.

How to actually fix it

Allowlist the scheme. http and https, nothing else. Kills file://, gopher://, dict://.

Resolve the hostname yourself, then check the IP you got back. This is the important one. DNS turns a name you can't trust into an address you can actually evaluate.

Check every address it resolves to, not just the first. A host can hand back one public record and one private one.

Don't follow redirects. Or if you must, re-run all your checks on every hop. Following one blindly throws away everything above it.

Pin the connection to the IP you validated. Between your DNS lookup and your actual request, the record can change under you. That's DNS rebinding and pinning is the only thing that closes it.

And if you already know which hosts you're supposed to be calling, just allowlist those and drop everything else. Allowlists fail closed, blocklists fail open.

Check what you're actually going to connect to, at the moment you connect to it.

Every bypass here is the same shape underneath: a gap between the thing you checked and the thing you reached, whether that's a different spelling, a different DNS record, a redirect or a re-resolution. Close the gap and the whole category goes with it.

And test the payloads you read about. I nearly wrote up the decimal IP trick as gospel before I bothered running it.

Dotexe - *September 3, 2026, 5:55 am*

Agh, I miss such contents so much, unfortunately they are so rare nowadays. I definitely had fun reading the way you explained things, nicely done.

Sift - *September 3, 2026, 6:01 am*

(Sep 3, 2026 05:55 AM)Dotexe Wrote: [CITE] Agh, I miss such contents so much, unfortunately they are so rare nowadays. I definitely had fun reading the way you explained things, nicely done.
I think SSRF is a good vulnerability to check these days with many newer webapps utilizing Webhooks and often forget to whitelist URL's or block request sent to localhost or 127.0.0.1.

And think it's past time to attempt to bring more content into this section of the forum.

Dotexe - *September 3, 2026, 6:04 am*

(Sep 3, 2026 06:01 AM)Sift Wrote: [CITE] I think SSRF is a good vulnerability to check these days with many newer webapps utilizing Webhooks and often forget to whitelist URL's or block request sent to localhost or 127.0.0.1.

And think it's past time to attempt to bring more content into this section of the forum.
Absolutely, that's my still go to when it comes to web app pentesting.