Server-Side Request Forgery (SSRF) Attacks - Part 1: The basics

Maxime Leblanc Follow
Aug 14, 2017 · 4 min read

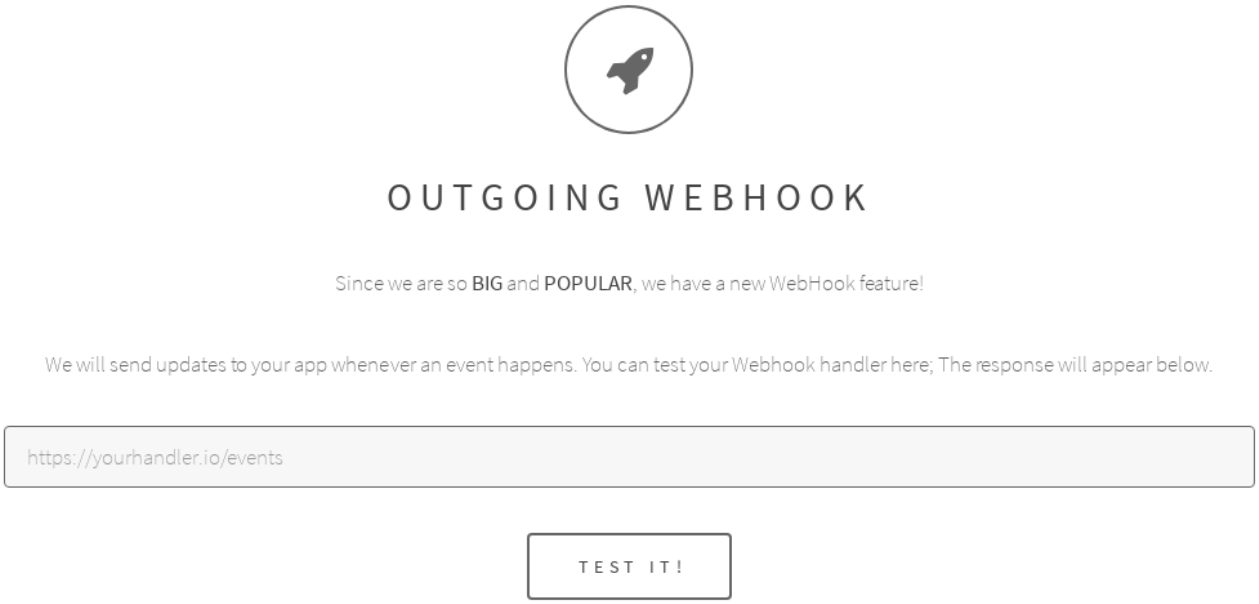
Hello world; For this article, I will introduce you to the notion of Server-Side Request Forgeries (SSRF), the server-side variant of it's better-known cousin, the Client-Side Request Forgery (CSRF). Now before I begin, I must say that this first publication will only cover the basics of the concept, in order to introduce more advanced techniques later-on, in my next SSRF stories. I must also give credit to Nicolas Grégoire (Hack in Paris, Hackfest) and to Orange Tsai (DEFCON) who both largely inspired the content of the present series and in a way triggered my interest for this technique. On my GitHub I have made an SSRF Lab available in order for you to try it out yourself. All my examples will be taken from there.

So what exactly is a Server-Side Request Forgery? It is simply a way to have a server make a network request on your behalf somewhere you shouldn't be able to. This is typically made possible because the web server usually has access to more resources than an external agent because the level of trust inside the perimeter of the local network is most likely higher than what is coming directly from the Internet (which, by the way, demonstrates the limits of the popular "defense in-depth" approach, also known as the "castle" approach to computer security).

SSRF attacks are not new, but trends are emerging and expose original attack surfaces. For the present story, we will suppose a shiny new Web App, which has become so popular that it has now begun to implement neat features such as a REST API and custom WebHooks.

The vulnerable application

Of course, you want your users to be able to test their WebHook handlers before publishing them. In this case, you will need to give them a nice debugging interface:



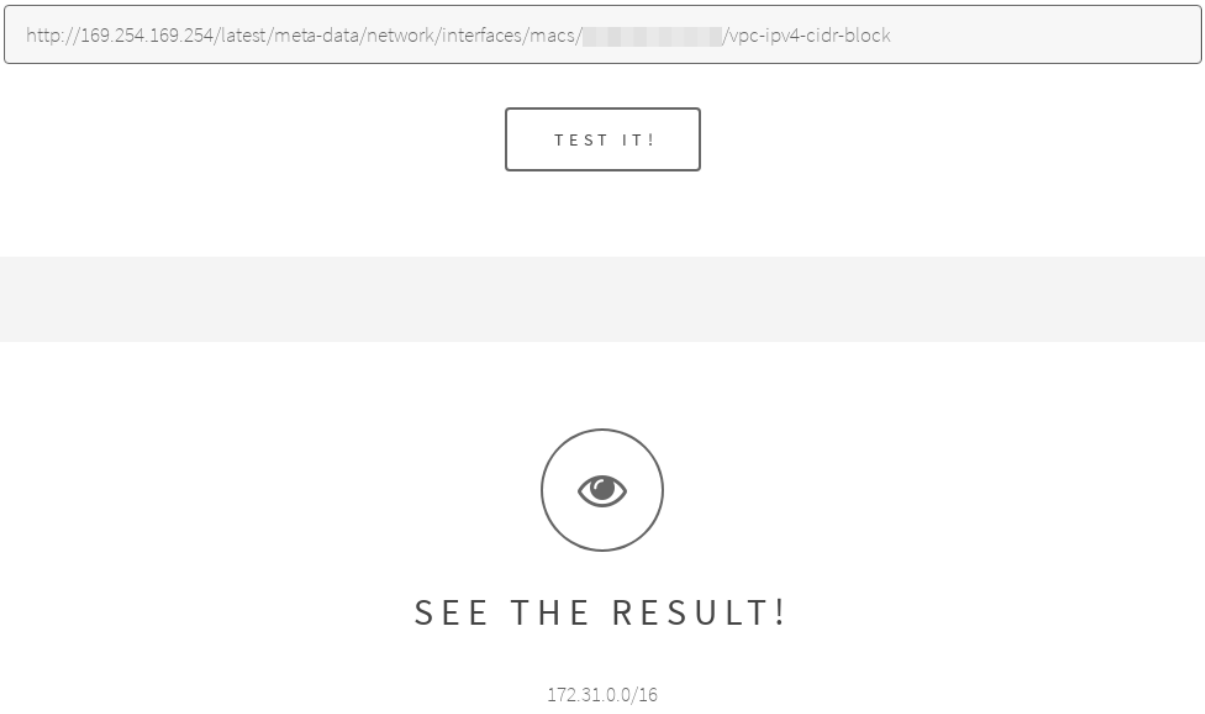
So if your user has a REST API listening on *https://yourhandler.io/events*, he or she will receive a test event and have some debugging information, such as the response content and status code from its server.

But we can exploit that.

If we are in the position of an attacker, we now have a server willing to make HTTP requests on our behalf to an arbitrary location, and give us the response it got! The thing is, it does not make sure the URL we input is actually an *external* Internet address.

What if instead of an event handler, we input something like *http://127.0.0.1:8080*? Well, we now have a port scanner on the host itself, beyond the Firewall or Security Group and even if the services only listen on *localhost*! Of course, we could also try to scan local addresses in the popular 10.0.0.0/8 subnet. But how could-we know the actual subnet of the vulnerable host?

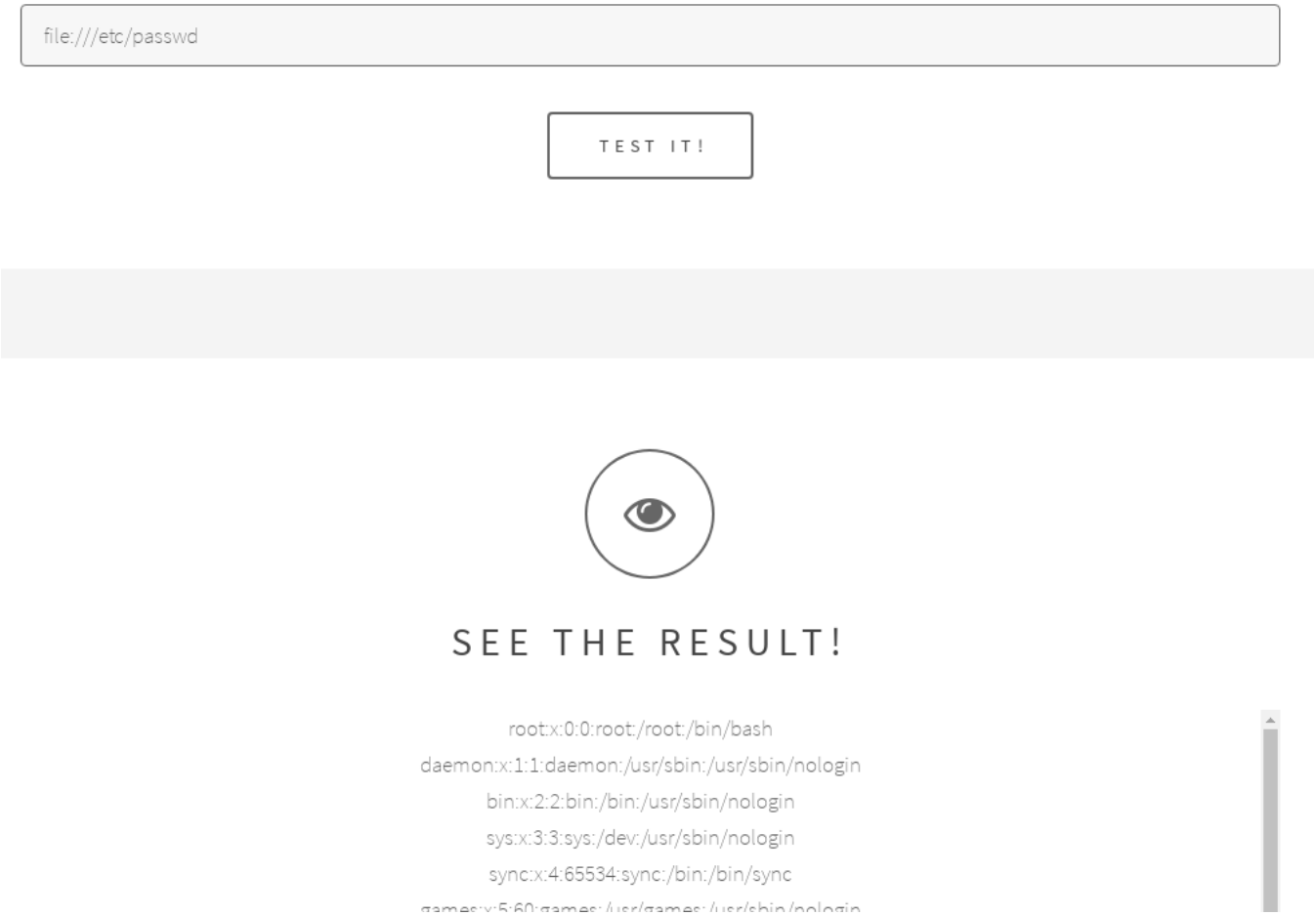
Well, let's says that a little WHOIS request tells you that the public IP address of the vulnerable Web Application is part of the AWS infrastructure, why not try to just ask the AWS metadata service?



And we now know the address space of the VPC for our Web App. The metadata service can be full of other very useful information, but what if we were not restricted to the Web? Let's remember the structure of a URL:

scheme://user:pass@host:port/path?query=value#fragment

The previous demonstrations abuse the *host* and *port* parts of a URL. But we can also play with the *scheme* part: By replacing *http* or *https* by *file*, we get a free pass on the host's filesystem (or at least what the *www-data* user can see, under Ubuntu). Let's test it:



Yep, it works! We are now able to dig through configuration files and source code files, looking for database credentials, as an example.

Conclusion

If your application makes requests to external resources, make sure they stay ***external*** under all circumstances. I can already hear you say that in this story's case, a simple regex filtering on the user's input would have done the trick. My next publications should convince you that things are not so simple, and URL validation is harder than one might think. Network-level egress filtering might be one of the only ways to protect your application from this, combined with strict *scheme://* white-listing.

Until then, happy hacking!