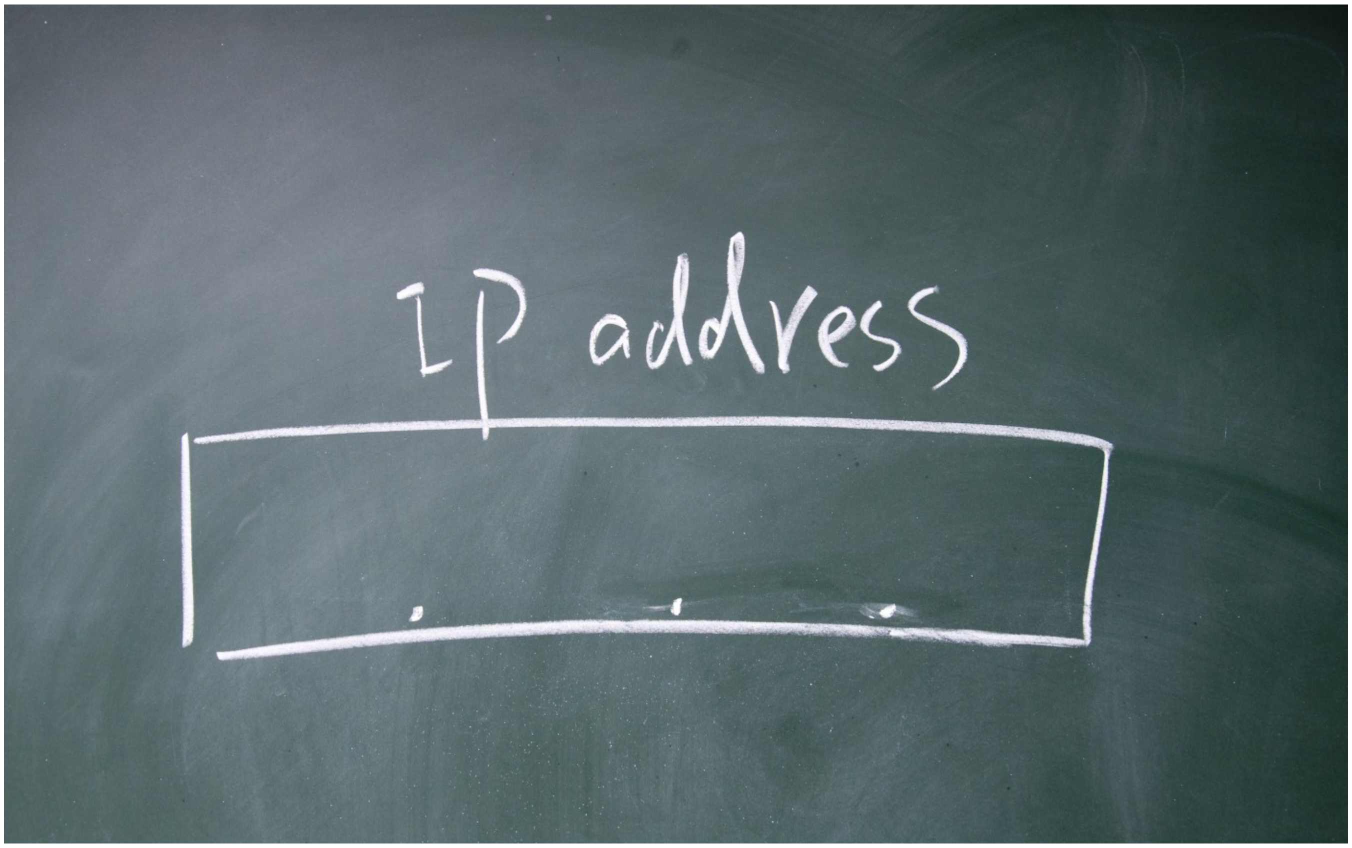


Server-Side Request Forgery (SSRF) Attacks — Part 2: Fun with IPv4 addresses

Maxime Leblanc Follow
Aug 28, 2017 · 4 min read

Hello again; This is the second part of my article about Server-Side Request Forgery attacks (SSRF). Now, in order to give you a little bit of context, I strongly suggest that you take a look at Part 1 of this article, at least to be aware of the Webhook feature we are hacking. Again, I must give credit to Nicolas Grégoire (Hack in Paris, Hackfest) and to Orange Tsai (DEFCON) who both strongly inspired this article (*Part 2* in particular extensively uses Nicolas' techniques, I hope he takes it as a tribute rather than plagiarism if he ever sees this 🙂). I also remind you that all the code I am writing about is available on my GitHub's SSRF Lab. This Part 2 in particular is available as a Docker-Compose script that should be runnable as-is with its network.

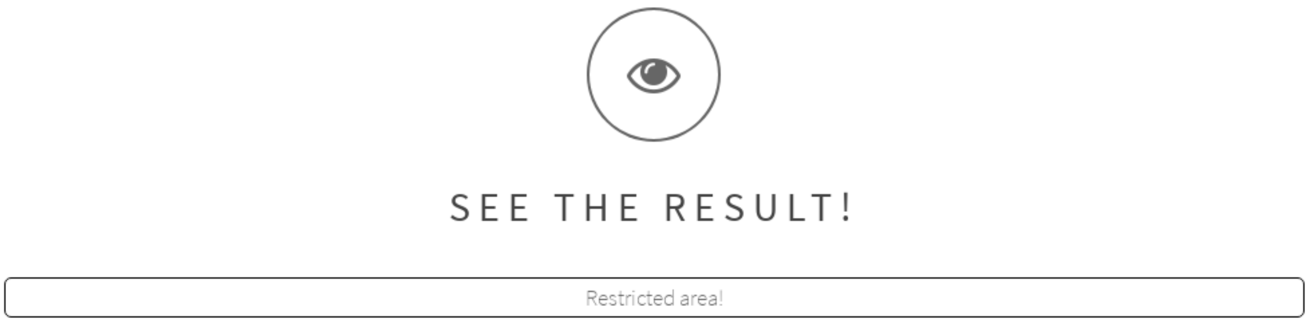
Let's re-use Part 1's super popular application:

So, let's say that you are aware of the possibility of SSRF attacks in your new WebHook feature and have decided to secure it in a way that restricts requests to `http://` and prohibits requests to `10.0.0.3` , which hosts an internal private service the outside world shouldn't see. You decide to do this using two regexes, one insuring that the URL begins with `http[s]://` and another one restricting the specific IP of our secret server:

```
if (preg_match('#^https?://#i', $handler) !== 1) {
  echo "Wrong scheme! You can only use http or https!";
  die();
} else if(preg_match('#^https?://10.0.0.3#i', $handler) === 1) {
  echo "Restricted area!";
  die();
}
```

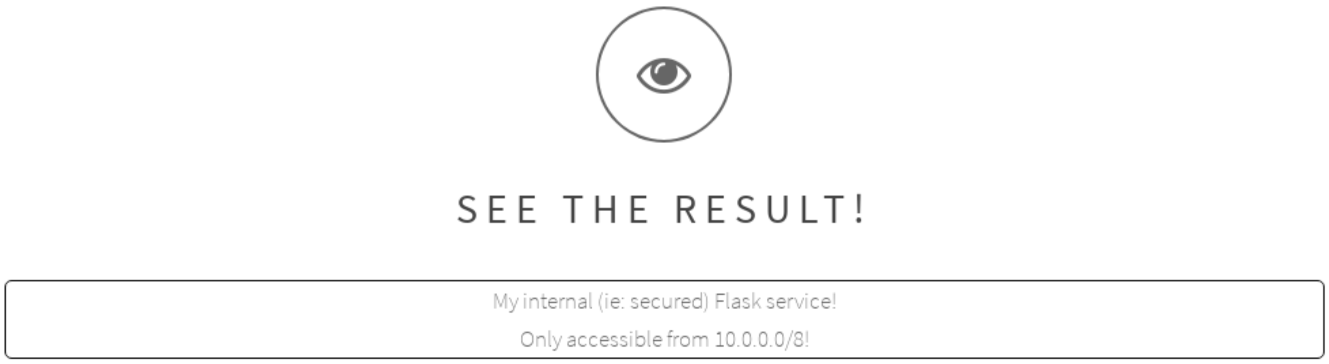
Disclaimer: This is a bad way to secure your server, ***please*** don't do this. Also note that the WebHook uses a simple PHP `curl` function, nothing too exotic.

We will leave `scheme://` hacking for Part 3. What about a request to this intriguing server at `http://10.0.0.3` ?



Restricted area!

Damn, how could-we get around this somehow without DNS or such? Let's try something funky, I will explain later; Let's try a request for `http://167722163` instead:



My internal (ie: secured) Flask service!
Only accessible from 10.0.0.0/8!

Wait, what happened there? Remember that an IPv4 address is just the four bytes relative to the OSI Layer 3 (IP) in the network stream. We usually express it as four numbers for convenience, but the integer of this value is also entirely valid. Let's examine the address in more details:

```
String value: 10.0.0.3
Binary:       00001010 . 00000000 . 00000000 . 00000011
Hexadecimal:  0A.00.00.03
Integer:      167772163
```

Note that in order to convert to the decimals we know, all network addresses in TCP/IP use Big-Endian representation. This way, our hack was to give the WebHook a valid representation of the address we were trying to reach, and the `curl` backend understood it.

Now, what else can-we try? There are surely a number of ways that 32 bits can be represented; One of the most obvious would be hexadecimal: `http://0x0A000003`. And guess what? It works, it gets around the regex filter and gets interpreted as an IPv4 address by `php-curl` like a charm!

Another less-known variant of number literals in the computer world are octal representations: In a number of languages, if the least-significant position in a number is zero, it gets interpreted as base-8 instead of our human-friendly base-10 decimals. So if you write `020` instead of `20`, your computer will most-likely think you mean 16 in our everyday decimal system. We can also use this to trick the regex filters: `http://01200000003` is the octal representation of our `10.0.0.3` IP address, and yes, it works with `php-curl`.

Conclusion

From there, we can try all kinds of different approaches; I don't know how the dots are supposed to be handled in an IP parser, but separating the "dot-parts" in hexadecimal also works: `http://0x0A.0x00.0x00.0x03` is indeed also a valid request, along with `http://012.00.00.03`. Nicolas Grégoire also pointed-out at the Hack in Paris that some NodeJS application servers could also overflow bytes, in such a way that `http://265.0.0.3` could also work for our `http://10.0.0.3` target, (because 265 is too big for 8 bits, it "resets" at 255 and becomes 10); I was not able to reproduce this on my setup, but if you have any idea, leave a comment 😉.

Finally, I just wanted to point-out that regex network filtering is not the way to go when it comes to securing your internal servers. The bestway is to use networking utilities that actually understand the TCP/IP logic and all the subtleties that the network eators had in mind when implementing those standards.

Happy Hacking!

Thanks to Laurent Jalbert-Simard.

Technology