The perils of the “real” client IP

Summary

This post ended up being incredibly ~~long~~ comprehensive. I’m afraid that many people won’t read enough to get everything that’s important, so here are the key points:

When deriving the “real client IP address” from the `X-Forwarded-For` header, use the rightmost IP in the list.

The leftmost IP in the XFF header is commonly considered to be “closest to the client” and “most real”, but it’s trivially spoofable. Don’t use it for anything even close to security-related.

When choosing the rightmost XFF IP, make sure to use the *last* instance of that header.

Using special “true client IPs” set by reverse proxies (like `X-Real-IP`, `True-Client-IP`, etc.) *can* be good, but it depends on the a) how the reverse proxy actually sets it, b) whether the reverse proxy sets it if it’s already present/spoofed, and c) how you’ve configured the reverse proxy (sometimes).

Any header not specifically set by your reverse proxy cannot be trusted. For example, you *must not* check the `X-Real-IP` header if you’re not behind Nginx or something else that always sets it, because you’ll be reading a spoofed value.

A lot of rate limiter implementations are using spoofable IPs and are vulnerable to rate limiter escape and memory exhaustion attacks.

If you use the “real client IP” anywhere in your code or infrastructure, you need to go check right now how you’re deriving it.

This is all explained in detail below, so keep reading. It’s a weird, scary, bumpy ride.

Introduction

The state of getting the “real client IP” using `X-Forwarded-For` and other HTTP headers is terrible. It’s done incorrectly, inconsistently, and the result is used inappropriately. This leads to security vulnerabilities in a variety of projects, and will certainly lead to more in the future.

After thinking about rate limiters for a while, I started worrying about their IPv6 handling. I wrote a post detailing how bad IPv6 rate limiting can and does lead to rate limiter escape and memory exhaustion. Then I moved on to worrying about how rate limiters determine what IP to rate-limit when they’re behind a load balancer (or any reverse proxy). As you’ll see, the situation is bad.

But this isn’t just about rate limiters. If you ever touch code that looks at the `X-Forwarded-For` header, or if you use someone else’s code that uses or gives you the “real client IP”, then you absolutely need to be savvy and wary. This post will help you get there.

NOTE: Portions of this are redacted as I'm trying to disclose responsibly to the affected projects. Those portions will be added in as that projects choose to make the issues public. (So check back later!)

It can't be that hard to get the real client IP, right?

There are many reasons why web services are interested in the IP address of their clients: geographical stats, geo-targeting, auditing, rate-limiting, abuse-blocking, session history, etc.

When a client directly connects to a server, the server can see the IP address of the client. If the client connects through one or more proxies (of any kind: forward, reverse, load balancer, API gateway, TLS offloading, IP access control, etc.), then the server only directly sees the IP address of the final proxy used by the client connection.

In order to pass the original IP address on to the server, there are several headers in common use:

`X-Forwarded-For` is a list of comma-separated IPs that gets appended to by each traversed proxy[1]. The idea is that the first IP (added by the first proxy) is the true client IP. Each subsequent IP is another proxy along the path. The last proxy's IP is *not* present (because proxies don't add their own IPs, and because it connects directly to the server so its IP will be directly available anyway). We're going to talk about this a lot, so it'll be abbreviated to "XFF".

`Forwarded` is the most official but seemingly least-used header. We look at it in more detail below, but it's really just a fancier version of XFF that has the same problems that we're about to discuss.

There are also special single-IP headers like `X-Real-IP` (Nginx), `CF-Connecting-IP` (Cloudflare), or `True-Client-IP` (Cloudflare and Akamai). We'll talk more about these below, but they're not the main focus of this post.

Pitfalls

Before talking about how to use XFF properly, we'll talk about the many ways that using `X-Forwarded-For` can go wrong.

First of all, and most importantly, you must always be aware that *any XFF IPs that were added (or appear to have been added) by any proxy not controlled by you are completely unreliable*. Any proxy could have added, removed, or modified the header any way it wants. The client as well could have initially set the header to anything it wants to get the spoof-ball rolling. For example, if you make this request to an AWS load balancer[2]…

```
curl -X POST https://my.load.balanced.domain/login -H "X-Forwarded-For: 1.2.3.4, 11.22.33.44"
```

…your server behind the load balancer will get this:

```
X-Forwarded-For: 1.2.3.4, 11.22.33.44, <actual client IP>
```

And this:

```
curl -X POST https://my.load.balanced.domain/login -H "X-Forwarded-For:
oh, hi,,127.0.0.1,,,,"
```

…will give you this:

```
X-Forwarded-For: oh, hi,,127.0.0.1,,,,, <actual client IP>
```

As you can see, everything already present is just passed through, unchanged and unvalidated. The final, actual IP is just appended to whatever is already there.

(In addition to curl'ing and custom clients, there is also at least one Chrome extension that lets you set the XFF header in browser requests. But *how* you can set the header doesn't really matter to us here, only that an attacker can do it.)

According to the HTTP/1.1 RFC (2616)[3]:

> Multiple message-header fields with the same field-name MAY be present in a message if and only if the entire field-value for that header field is defined as a comma-separated list [i.e., #(values)]. It MUST be possible to combine the multiple header fields into one "field-name: field-value" pair, without changing the semantics of the message, by appending each subsequent field-value to the first, each separated by a comma. The order in which header fields with the same field-name are received is therefore significant to the interpretation of the combined field value, and thus a proxy MUST NOT change the order of these field values when a message is forwarded.

That applies to XFF, as it is a comma-separated list. This can make getting the rightmost (or even leftmost) IP error-prone.

For example, Go has three ways to get a header value:

`http.Header.Get(headerName)` returns the first header value as a string.

`http.Header.Values(headerName)` returns a slice (array) of strings with the values of all instances of the header `headerName`. (`headerName` is canonicalized before lookup.)

`http.Header` is a `map[string][]string` and can be accessed directly. (The map keys are canonicalized header names.) This is similar to using `Values`.

So here's the attack:

Eve makes a request with *two* spoofed XFF headers.

Your reverse proxy adds Eve's true IP to the end of the *second* XFF header, per the RFC requirements.

You call `req.Header.Get("X-Forwarded-For")` and get the first header. You split it up and take the rightmost.

You have chosen a spoofed IP. You treat it as trustworthy. Bad things result.

Unlike Go, Twisted's method for getting a single header value returns the *last* value. (Why is there no standard, common, accepted behaviour for this?) This avoids the above attack, but it can cause a different (less likely) problem: If you're using the rightmost-ish algorithm (described below), you need to go backwards from the right looking for the first untrusted IP. But what if one of your reverse proxies has added a new header instead of appending (a valid thing to do, per the RFC)? Now the IP that you want is nowhere to be found in the last header – it's full of trusted reverse proxy IPs and the real IP is in a previous instance of the XFF header.

There might be a subtle, hypothetical attack possible here:

You have (at least) two reverse proxies that you trust.

The second of those reverse proxies doesn't like super long headers, so it creates a new one rather than appending if the XFF header is too long.

Eve knows this. And she wants to hide her IP from you.

Eve spoofs a long XFF in her request to you.

Your first reverse proxy adds her true IP to the XFF header.

Your second reverse proxy doesn't like how long that header is, so it creates a new one. The header value is the IP of the first reverse proxy.

Your server software gets the last header and it has only a single IP, belonging to your first reverse proxy.

What does your logic do? Use that IP? Treat it as special because it's private/trusted? Panic because it's impossible that this IP should be trusted?

Note that when I tested with a server behind AWS ALB I found that ALB had already concatenated the XFF headers. So that's good. I have no idea if other reverse proxies do the same, but I bet there's no real consistency.

The best thing to do is merge all of the XFF headers yourself.

(It is worthwhile asking – and checking – to make sure reverse proxies append to the correct header, because appending to the wrong header would wreck the trustworthiness of taking the rightmost. I have only checked AWS ALB and Cloudflare, and they're doing it right. If anyone discovers something doing it wrong, please let me know.)

[2022-03-04: I created a Go issue arguing for a change to the behaviour of `http.Header.Get`. Not with any real expectation of a change, but we'll see.]

Private IPs

Even in completely non-malicious scenarios, any of the XFF IPs – but especially the leftmost – may be a private/internal IP address. If the client first connects to an internal proxy, it may add the private IP of the client to the XFF header. This address is never going to be useful to you.

Splitting the IPs

Because `X-Forwarded-For` isn't an official standard, there's no formal spec for it. Most examples show the IP addresses comma-space (`", "`) separated, but the space isn't strictly required. (For example, the HTTP/1.1 RFC says that headers *like* XFF are simply "comma separated".) *Most* of the code I looked at splits by just comma and then trims the value, but at least one I found looks for comma-space.

When testing, it looks to me like AWS ALB uses comma-space when adding an IP, but Cloudflare uses only a comma.

Unencrypted data is always untrustworthy

This should go without saying, but if you're receiving HTTP-not-S requests, then anyone could have modified the headers before they got to you. This is worth mentioning because an interloper can't mess with the "rightmost-ish" approach (described below) because they can't mess with the IP of the final connection from the internet to your reverse proxy or server.

So just encrypt your traffic, okay?

Other headers (`X-Client-IP`, `True-Client-IP`) might be present and spoofed

Some reverse proxies remove any unexpected or unwanted headers, but some (like AWS ALB) don't. So an attacker can set headers like `X-Client-IP` and `True-Client-IP` straight through to your server. You need to not get fooled into using them if your reverse proxy didn't specifically set them for you.

Trying to learn about `X-Forwarded-For`

Trying to educate yourself about XFF is, unfortunately, also difficult.

MDN Web Docs are usually the gold standard for stuff like this, but the page about XFF doesn't mention these risks at all; it says "the right-most IP address is the IP address of the most recent proxy and the left-most IP address is the IP address of the originating client" with no caveat. The Wikipedia entry is much better: "Since it is easy to forge an X-Forwarded-For field the given information should be used with care. The right-most IP address is always the IP address that connects to the last proxy, which means it is the most reliable source of information."

[2022-03-09: Created an issue for the MDN documentation. 2022-03-19: I rewrote the page, PR'd it, and the change is live now. You can see a PDF of the original page here. Now to fix the `Forwarded` page…]

Other sources are similarly variable. Some say nothing whatsoever about the possibility of the header being spoofed or the presence of private addresses (1, 2, 3, 4, 5). Others do a pretty good job of mentioning the risks (6, 7, 8), but sometimes you have to read pretty deeply to get to the warnings.

Avoiding those pits

Let's make a few baseline statements:

Using an IP in the private address space as the “real” client IP is never the right choice[4].

Using a value that’s not actually an IP address is never the right choice.

In the absence of chicanery, the leftmost non-private, non-invalid IP is the closest we can come to the “real” client IP. (Henceforth, “leftmost-ish”.)

The only client IP that we can *trust* is the first one added by a (reverse) proxy that we control. (Henceforth, “rightmost-ish”.)

The leftmost-ish is usually going to be the most “real”, while the rightmost-ish is going to be the most trustworthy. So which IP should you use? It depends on what you’re going to do with it.

If you’re going to do something security-related, you need to use the IP you trust – the rightmost-ish. The obvious example here is rate-limiting. If you use the leftmost-ish IP for this, an attacker can just spoof a different XFF prefix value with each request and *completely avoid being limited*.

Additionally, they might be able to exhaust your server memory by forcing you to store too many individual entries – one for each fake IP. It may seem hard to believe that storing IP addresses in memory could lead to exhaustion – especially if they’re stored in a cache with finite time-to-live, but keep in mind:

The attacker won’t be limited to the 4 billion IPv4 addresses. They can use all the bazillion IPv6 addresses, if the limiter isn’t smart about prefixes.

Since many limiters don’t check for valid IPs, an attacker can use any random string it wants.

Also note that these strings can be *big*; for example, Go’s default header block size limit is 1MB. That means a single random-string “IP” can be almost 1MB. That means adding 1MB of memory usage *per request*.

It still won’t be feasible for all attackers and configurations, but it shouldn’t be dismissed without consideration.

Or an attacker can force you to rate-limit/block the IP addresses of other users. They can supply a real – but not their – IP address, and you’ll eventually be fooled into rate-limiting it. (And if you’re using the “real” IP for abuse reports, you might end up complaining about the wrong person.)

The downside to using the rightmost-ish IP for rate-limiting is that you might block a proxy IP that’s not actually a source of abuse but is just used by a bunch of different clients and you would have realized that if you’d just used the leftmost-ish instead. Yeah, well. That doesn’t seem super likely, and it’s still infinitely more acceptable than allowing attackers to trivially bypass your rate limiter and crash your server.

If you’re doing something not obviously security-related… Think hard about your use case. Let’s say you just want to do an IP-geolocation lookup for your stats. *Probably* the leftmost-ish IP is what you want. The vast majority of your users won’t be doing any header spoofing, and the geolocation of random internet proxies are no good to you, so you’ll *probably* get the best results with the IP closest to the user.

On the other hand, you might want to consider how many internet-proxy-using users you expect to have. Probably few enough that it won’t hurt your stats if you geolocate the wrong thing. And is there a way an

attacker could hurt you by purposely skewing your geo stats? Probably not, but take a moment to really think about it.

So *be careful* when writing your "GetRealClientIP(request)" function. Make sure it has a big warning comment about how it should be used. Or maybe write two functions: "GetUntrustworthyRealClientIP(request)" and "GetTrustworthyButLessRealClientIP(request)". Which are horrible names. Maybe just pass a flag instead. Anyway, the point is to prevent the callers of your functions from having any confusion about the nature of the result.

Also be careful when using the results of that function. It's easy to write code that gets the leftmost-ish IP to do some geo lookup and then later decide you also need to do rate limiting… so you might as well use the same "realClientIP" variable! Oops. This might be a good time to make wrong code look wrong.

And remember that the final proxy IP – or the address of the client if it's connecting directly – is *not* in the XFF header. You need to look at your request connection information for that. (`http.Request.RemoteAddr` in Go, the `REMOTE_ADDR` environment variable for many CGI servers, etc.)

Algorithms

When reading this, remember that the final proxy IP is not in the XFF list – it's the `RemoteAddr`. Also note that `RemoteAddr` might have the form `ip:port`, depending on your platform (like it does in Go) – sure sure to only use the IP part.

First: collect all of the IPs

Make a single list of all the IPs in all of the `X-Forwarded-For` headers.[5] Also have the `RemoteAddr` available.

Second: decide what your security needs are

Default to using the rightmost-ish approach. Only use the leftmost-ish if you have to, and make sure you do so carefully.

Leftmost-ish: the closest to the "real IP", but utterly untrustworthy

If your server is directly connected to the internet, there might be an XFF header or there might not be (depending on whether the client used a proxy). If there *is* an XFF header, pick the leftmost IP address that is a *valid*, *non-private* IPv4 or IPv6 address. If there is no XFF header, use the `RemoteAddr`.

If your server is behind one or more reverse proxies, pick the leftmost XFF IP address that is a *valid, non-private* IPv4 or IPv6 address. (If there's no XFF header, you need to fix your network configuration problem *right now*.)

And never forget the security implications!

Rightmost-ish: the only useful IP you can trust

If your server is directly connected to the internet, the XFF header cannot be trusted, period. Use the `RemoteAddr`.

If your server is behind one or more reverse proxies and not directly accessible from the internet, you need to know either the IP addresses of those reverse proxies or the number of them that the request will pass through. We'll call these the "trusted proxy IPs" and "trusted proxy count". (Using "trusted proxy IPs" is preferable, for reasons described in the "network architecture changes" section.)

The trusted proxy IPs or trusted proxy count will tell you how far from the right of the XFF header you need to check before you find the first IP that doesn't belong to one of your reverse proxies. This IP was added by your first trusted proxy and is therefore the only IP you can trust. Use it.

(Notice that I'm not saying "valid, non-private IP" here. It is tempting to do so, just to be extra safe, and I won't blame you if you do, but if you can't trust your own reverse proxy to add the proper IP, then you have a bigger problem.)

Again, if you're behind one or more reverse proxies and there's no XFF header, you need to immediately figure out how people are connecting to your server directly.

Tentative variation: rightmost non-private IP

If all of your reverse proxies are in the same private IP space as your server, I *think* it's okay to use the rightmost non-private IP rather than using "trusted proxy IPs" or "trusted proxy count". This is equivalent to adding all private IP ranges to your "trusted proxy IPs" list.

An example where this *doesn't* work is if you're behind an external reverse proxy service, like Cloudflare – it's not in your private address space.

Falling into those pits

Let's look at real-world examples!

Warning: I got a little carried away here. I was only intending to look at a couple of projects that I was familiar with, but the hit-rate of dangerous-use-of-leftmost was so high that I just kept searching. (And there were some interesting and educational aspects even when it was done right.)

(If a tool or service isn't mentioned here, it's because I either didn't look at it or couldn't find enough information about it. I included all successes as well as failures.)

Cloudflare, Nginx, Apache

Let's start with some good news.

Cloudflare adds the `CF-Connecting-IP` header to all requests that pass through it; it adds `True-Client-IP` as a synonym for Enterprise users who require backwards compatibility. The value for these headers is a single IP address. The fullest description of these headers that I could find makes it *sound* like they are just using the leftmost XFF IP, but the example was sufficiently incomplete that I tried it out myself. Happily, it looks like they're actually using the rightmost-ish.

Nginx offers a not-enabled-by-default module that adds the `X-Real-IP` header. This is also a single IP. When properly and fully configured[6], it also uses the rightmost IP that isn't on the "trusted" list. So, the rightmost-ish IP. Also good.

Similarly, when configured to look at `X-Forwarded-For`, Apache's mod_remoteip picks the rightmost untrusted IP to set into `REMOTE_ADDR`.

Akamai

Akamai does very wrong things, but at least warns about it. Here is the documentation about how it handles `X-Forwarded-For` and `True-Client-IP` (original emphasis):

> `X-Forwarded-For` header is the default header proxies use to report the end user IP that is requesting the content. However, this header is often overwritten by other proxies and is also overwritten by Akamai parent servers and thus are not very reliable.
>
> The `True-Client-IP` header sent by Akamai does not get overwritten by proxy or Akamai servers and will contain the IP of the client when sending the request to the origin.
>
> `True-Client-IP` is a self provisioned feature enabled in the Property Manager.
>
> *Note that if the `True-Client-IP` header is already present in the request from the client it will not be overwritten or sent twice. It is not a security feature.*
>
> The connecting IP is appended to `X-Forwarded-For` header by proxy server and thus it can contain multiple IPs in the list with comma as separator. `True-Client-IP` contains only one IP. If the the end user uses proxy server to connect to Akamai edge server, `True-Client-IP` is the first IP from in `X-Forwarded-For` header. If the end user connects to Akamai edge server directly, `True-Client-IP` is the connecting public IP seen by Akamai.

The relevant bits are "`True-Client-IP` is the first IP from in `X-Forwarded-For` header" and "if the True-Client-IP header is already present in the request from the client it will not be overwritten". So `True-Client-IP` is either the leftmost XFF IP or keeps the original value spoofed by the client. Just the worst possible thing.

However, there is also the sentence "It is not a security feature." Well, that's certainly true. Does that warning make it okay? What's the chance that there aren't a ton of Akamai users out using `True-Client-IP` for security-related purposes?

(I'm not sure how to interpret the above when it says that the XFF header is "overwritten by Akamai parent

servers". Does it mean "appended to" when it says "overwritten"? Or is Akamai actually blowing away the existing header value? That would be against the spirit of XFF.)

Fastly

Fastly adds the `Fastly-Client-IP` header with a single IP value. I *think* it's using the rightmost-ish XFF IP:

> Essentially, `Fastly-Client-IP` is the non-Fastly thing that is making the request to Fastly.

However:

> The value is not protected from modification at the edge of the Fastly network, so if a client sets this header themselves, we will use it. If you want to prevent this [you need to do some additional configuration].

So, by default `Fastly-Client-IP` is trivially spoofable. Again, it seems highly likely that there are a lot of people using its default behaviour for security-related purposes and making themselves vulnerable to attack.

Azure

Azure Front Door adds the `X-Azure-ClientIP` and `X-Azure-SocketIP` headers. They are described like so:

> `X-Azure-ClientIP`: Represents the client IP address associated with the request being processed. For example, a request coming from a proxy might add the `X-Forwarded-For` header to indicate the IP address of the original caller.
>
> `X-Azure-SocketIP`: Represents the socket IP address associated with the TCP connection that the current request originated from. A request's client IP address might not be equal to its socket IP address because the client IP can be arbitrarily overwritten by a user.

So, `X-Azure-ClientIP` is the leftmost-ish XFF IP and `X-Azure-SocketIP` is the rightmost-ish.

That's reasonably good, but I think it could be a lot clearer. The only warning about `X-Azure-ClientIP` is a subtle hint in the description of the *other* header. I also hand-wavingly feel that the name of the less-secure header is more appealing than the more-secure one, and is probably leading many people into the wrong choice.

go-chi/chi

Chi is a Go HTTP router and provides a RealIP middleware and a rate limiter. The RealIP middleware has this comment:

> You should only use this middleware if you can trust the headers passed to you (in particular, the two [three, actually] headers this middleware uses), for example because you have placed a reverse proxy like HAProxy or nginx in front of Chi. If your reverse proxies are configured to pass along arbitrary header values from

> the client, or if you use this middleware without a reverse proxy, malicious clients will be able to make you very sad (or, depending on how you're using RemoteAddr, vulnerable to an attack of some sort).

Which is a pretty good warning, right? Almost.

Let's take this opportunity to talk about abusing `X-Real-IP`, `True-Client-IP`, etc. For example, AWS ALB "[passes] along arbitrary header values from the client" and, indeed, if you don't realize that you will end up "very sad". Because a request like this…

```
curl -X POST https://my.load.balanced.domain/login -H "X-Forwarded-For:1.1.1.1" -H "X-Real-IP:2.2.2.2" -H "True-Client-IP:3.3.3.3"
```

…results in your server getting these headers:

```
X-Forwarded-For: 1.1.1.1, <actual client IP>
X-Real-IP: 2.2.2.2
True-Client-IP: 3.3.3.3
```

`chi.middleware.RealIP`'s logic goes like: "use the `True-Client-IP`; if that doesn't exist, use the `X-Real-IP`; if that doesn't exist, use `X-Forwarded-For`". So it falls victim to header spoofing.

But, as we've learned, the `chi.middleware.RealIP` warning also isn't good enough when it comes to `X-Forwarded-For` *because you can never, ever trust all of it*[7]. In the `RealIP` code, the logic I just paraphrased actually ends with "use the leftmost XFF IP address". And we're now suitably scared of using the leftmost XFF IP. (It also does not check that the leftmost "IP" is valid and non-private.)

So `chi.middleware.RealIP` falls firmly into the "only safe for non-security use" category. And you *must* be aware of its header preference order and what your reverse proxy does or doesn't set and let through. In short, it's hard to recommend it.

Chi's rate limiter has identical logic for obtaining the IP address and doesn't have the same warning. So that's bad, for the reasons we've discussed – the non-XFF headers could be spoofed, the XFF header could be spoofed, the IP can be garbage, the rate limiter could be bypassed, your memory could be exploded. The best way to use it is to not use its "real IP" logic and instead write your own "GetTrustworthyClientIP(request)" and pass that to its "rate limit by arbitrary keys" feature.

Both RealIP and httprate are both using Go's `http.Header.Get` to get the XFF header. As discussed above, this means that switching to taking the rightmost-ish IP wouldn't be sufficient, as an attacker could force the wrong header to be used.

Chi's rate limiter is also the one instance I found of the XFF list being split by comma-space instead of just comma. I think that's wrong.

[2022-03-03: Disclosed to maintainer via email. 2022-03-04: Maintainer requested that I make an issue.]

didip/tollbooth

The Tollbooth HTTP rate limiter is better, but you still need to be aware of what it's doing in order to use it properly.

Its README says the order in which it looks for the "real" client IP address is…

> By default it's: "RemoteAddr", "X-Forwarded-For", "X-Real-IP"
> If your application is behind a proxy, set "X-Forwarded-For" first.

Strangely, that default order isn't actually the default everywhere. If you call `limiter.New()` that *is* the default. But if you call `tollbooth.NewLimiter()` – "a convenience function to limiter.New" – the order is `"X-Forwarded-For", "X-Real-IP", "RemoteAddr"`. Which is an important difference!

It doesn't make clear that it doesn't support `CF-Connecting-IP`, `True-Client-IP`, or any other arbitrary header – it just silently skips them if you add them (returning empty string, leading to over-limiting).

This is a general problem with Tollbooth – it "fails open": if it can't find an IP to use it doesn't rate limit. So if the user of the library misconfigures it (by trying to use an unsupported header, or even making a letter-case mistake in a supported on), it won't rate limit at all and will give no indication of this. In a comment on the PR to fix this I give my opinion on how it should be have, but this is a pretty tricky problem.

Tollbooth also suffers from the "multiple headers" problem – it uses Go's `r.Header.Get`. So, even though it's counting from the rightmost, which is good, it's doing so with the first header, which is bad.

Speaking of counting from the rightmost… If Tollbooth is configured to count very far from the right (like, `lmt.SetForwardedForIndexFromBehind(1000)`), then it will effectively take the leftmost IP (it uses a minimum index of zero). In that case, it should be doing "leftmost-ish" IP validation.

There are two more things that bug me about tollbooth's design. The first is that the ostensible default order has Go's `http.Request.RemoteAddr` first. My understanding is that that field should never be empty (for an HTTP server, rather than client), so the rest of the list is guaranteed to be ignored. So why have a list? Also, if your server is behind a reverse proxy, `RemoteAddr` will be your proxy's IP, which is useless.

The second thing that bugs me is going to get its very own section…

[2022-03-03: Disclosed to maintainer via email. 2022-03-04: Maintainer created a PR to fix it. Ongoing discussion there.]

A default list of places to look for the client IP makes no sense

Where you should be looking for the "real" client IP is very specific to your network architecture and use case. A default configuration encourages blind, naïve use and will result in incorrect and potentially dangerous behaviour more often than not.

If you're using Cloudflare you want `CF-Connecting-IP`. If you're using `ngx_http_realip_module`, you want `X-Real-IP`. If you're behind AWS ALB you want the

rightmost-ish `X-Forwarded-For` IP. If you're directly connected to the internet, you want `RemoteAddr` (or equivalent). And so on.

There's *never* a time when you're okay with just falling back across a big list of header values that have nothing to do with your network architecture. That's going to bite you.

Even Tollbooth defaulting to using the rightmost XFF IP can be problematic. If your server is behind two layers of reverse proxies, then you'll be looking at the IP of your first proxy instead of the client's IP.[8] (What will probably happen is that you will rate-limit your proxy almost immediately, nothing gets through, and then you fix your config. But it still would have been better to be forced to think about the correct configuration in the first place.)

So, even though I know it's not very user friendly, I don't think that rate-limiting libraries should have any default at all, and instead should *require* explicit configuration.

ulule/limiter

Another Go rate limiter middleware. By default it doesn't look at the XFF header, but if enabled it uses the leftmost XFF IP. The option is called "TrustForwardHeader", but *you can never trust the XFF header*. So it falls victim to rate limit escape, etc.

It also uses Go's `http.Header.Get`, so if it switches to rightmost-ish it will need to change how it gets the XFF header.

When `TrustForwardHeader` is true it first looks for XFF and then falls through to `X-Real-IP` and finally uses `RemoteAddr`. But as we just saw, "a default list of places to look for the client IP makes no sense".

It returns `net.ParseIP(ip)` rather than just the raw string. This seems good at first, but `net.ParseIP` returns nil if the parse fails. So in the case of a garbage string, ulule/limiter doesn't check for the failure, returns nil, and then, as far as I can tell, uses `"<nil>"` as the "IP" key. (I'm surprised that it doesn't panic, but I don't think it does.) The way this logic works makes memory exhaustion more difficult, but it might be achievable using valid IPv6 addresses.

[2022-03-04: Disclosed to maintainer via email. 2022-03-05: PR has been created with fixes (mostly documentation warnings). It's public, so I'm un-redacting this. 2022-03-17: The PR was merged. I don't love the changes, but it sounds like the next major version will address the shortcomings.]

sethvargo/go-limiter

This is yet another Go rate limiter middleware. If its `httplimit.IPKeyFunc` is configured to look at the `X-Forwarded-For` header (which is given as an example in its comment), it will *use the whole header* as the rate limit key. That's almost worse than taking the leftmost IP.

The way to work around this would be to avoid its "real IP" logic and create your own `KeyFunc` that extracts the correct IP for your network architecture.

If the library can't find the configured header(s), it falls through to `RemoteAddr`. But, again, I don't think default fallbacks are good.

The library also uses Go's `http.Header.Get()`.

[2022-03-04: Disclosed to maintainer via email. 2022-03-05: Maintainer indicated by email that I could un-redact this.]

[REDACTED]

Pending disclosure

Let's Encrypt

It looks like Let's Encrypt is using Nginx with `X-Real-IP`. If it's configuration is good (I don't think the config files are in GitHub), then it should be using rightmost-ish.

[REDACTED]

Pending disclosure

Jetty

It looks like the Jetty web server uses the leftmost XFF IP address. I haven't dug far enough in to see what it's used for (or how it's exposed), but that's a dangerous start.

Express

Express is a NodeJS web framework. The default configuration ignores the XFF header, but it's possible to configure it to use the leftmost or a rightmost-ish XFF IP. The setting that uses the leftmost has this warning:

> When setting to true, it is important to ensure that the last reverse proxy trusted is removing/overwriting all of the following HTTP headers: X-Forwarded-For, X-Forwarded-Host, and X-Forwarded-Proto otherwise it may be possible for the client to provide any value.

So that's pretty good. (But still a footgun that I'm sure someone will fall victim to.)

Traefik

Traefik is a "cloud native network stack". Its rate limiter is configured to use the "trusted proxy count" version of rightmost-ish. So that's good.

phpList

phpList is an "open source newsletter and email marketing software". It uses the leftmost XFF IP. I can't quite tell what it's used for, but it's something surrounding login.

IIS

I could find anything to suggest that Microsoft IIS processes the XFF header, but an official support blog post entitled "How to use X-Forwarded-For header to log actual client IP address?" says:

> If you see multiple IP addresses in X-Forwarded-For column, it means the client went through more than one network device. Each network device adds their own IP to the end of the value. The left-most IP address is the actual client IP address. Others belong to network devices the client go through.

Which is a *woefully incomplete* statement. I fear for the 97,641 people who read that post.

Tor

Tor is an anonymity network. They have recently realized that they have a control server that is both directly connected to the internet *and* behind a reverse proxy and are using XFF to give them trustworthy IPs, so they're vulnerable to spoofing. It looks like they're working towards limiting the public-ness of the interface, or doing better verification of CDN connections, or both.

(Because I looked into it, I'll mention that it looks like they're *not* falling victim to the "multiple headers" pitfall. It looks like they use Twisted and call `request.getHeader` to get the XFF value. The Twisted source for that method indicates that it returns the *last* matching header. That could cause problems if you need the Nth-from-the-right header, but I think it's fine in this case.)

gorilla/handlers.ProxyHeaders

[Section added 2022-03-27. u/Genesis2001 asked about this on Reddit, so I looked at the code and figured I should add some comments here.]

Gorilla is a Go web toolkit. It's most known for its router, gorilla/mux. It has a `ProxyHeaders` middleware for handling XFF (that is intended for general consumption, not just for gorilla/mux users).

`ProxyHeaders` (source) is deficient in a number of ways, but at least it has a warning that the user's first reverse proxy must strip out the headers being checked before adding them back in. So it's good that it has that warning, but that requirement means that a) it won't be usable for a lot of users, and b) it will be misused by a lot of users.

Let's touch on the problems that are legitimately mitigated by stripping the headers:

It's taking the leftmost XFF and `Forwarded` values.

It's checking `X-Forwarded-For` and then `X-Real-IP` and then `Forwarded`. So it has the "default list" problem.

It's using `X-Forwarded-Host` to replace `r.Host`. So that's a new spoofable thing.

It's using `X-Forwarded-Proto`-then-`X-Forwarded-Scheme`-then-`Fowarded` to replace `r.URL.Scheme`. Another new spoofable thing.

It's using comma-space to parse `X-Forwarded-For`, contrary to RFC 2616. So it can set `r.RemoteAddr` to an IP like "1.1.1.1,10.1.1.1", etc. It's also not trimming the result, and I *think* the LWS rules of RFC 2616 mean that there can be spaces before the comma, so `ProxyHeaders` can also end up with strings like `"<space>1.1.1.1<space>"`.

It also doesn't support any single-IP headers besides `X-Real-IP`, which limits its general utility.

(And I keep wondering why stripping XFF at the first reverse proxy makes sense. If you have that much control – e.g., if you're using Nginx – you should instead just set `X-Real-IP` and let XFF behave the way it's intended to.)

[2022-03-27: Created issue and pull request.]

Quickies

[2022-03-29: Added this section.]

I keep finding more examples of code making mistakes. They help to drive home the points I'm trying to make, but they would bloat this post if I gave a section to every one. Instead I'm going to write some short bullets.

sebest/xff: Claims to be a `Forwarded` (RFC 7239) parser but is instead an XFF parser. It uses a leftmost-non-private algorithm, but doesn't document the risks (though there is a 6-year-old issue). The private IP ranges are incomplete. (Created issues.)

pbojinov/request-ip: Falls back between XFF and many single-IP headers. Uses leftmost for XFF. Has no warning about dangers. Claims to support `Forwarded`, but just returns the whole value. Has issues and PRs to address some of this, but project seems abandoned.

mo7zayed/reqip: Based on the previous one, with all the same shortcomings.

stanvit/go-forwarded: Takes the absolute rightmost, which is better than leftmost but still not right for all configurations. (Created an issue.)

Advanced and theoretical pitfalls and attacks

I've talked a lot about two attacks on rate limiters: avoiding being limited and exhausting server memory. I've done this because rate limiters are what led me to this topic and because causing a map of IPs to fill memory was an obvious danger in many implementations.

But rate limiters are only one "security-related" use of `X-Forwarded-For`, and there are more, cooler possibilities for badness! They're harder to find or reproduce, but they should be fun to speculate on…

Server behind reverse proxy *and* directly connectable

This was briefly mentioned in the "algorithms" section, but is worth repeating.

Generally speaking, if your server is behind one or more reverse proxies, there are one or more rightmost IPs in the XFF header that you can trust. The “rightmost-ish” algorithm is predicated on that. But if your server can *also* be connected to directly from the internet, that is no longer true.

With some experimentation, an attacker can craft an XFF header to look exactly like the one you expect from your reverse proxy:

Attacker gets her IP limited/blocked by your server.

Attacker crafts XFF header so that the rightmost of it has different IPs in the private space, and different counts of those IPs.

Continue until the limit/block unexpectedly disappears.

Now you’re using an untrusted XFF IP and don’t realize it. Rate limiter escape, memory exhaustion, etc.

One way to mitigate this is to check the `RemoteAddr` to make sure it belongs to your reverse proxy before you try to use the XFF header.

Re-fronting attack

Thanks to Ryan Gerstenkorn for sending me his blog post about this.

If a) your backend is in-house or otherwise externally accessible, and b) it’s fronted by a CDN, and c) you trust the IP addresses/ranges of your CDN, then you may be vulnerable to another class of attack.

With AWS CloudFront, it’s possible for an attacker to create a distribution that points to your origin. Now requests are coming to your origin from trusted IPs, but from a distribution not owned by you. But the real beauty/horror of this is that the attacker can use Lambda@Edge to modify the `Host` header so that you can’t tell that a different hostname was used to access your origin, and can also modify the `X-Forwarded-For` header to be whatever the attacker wants.

So your “trusted” reverse proxy IPs become untrusted and can lie to you about the client IP. This can be used to bypass your rate limiter, IP-based access control, etc.

The proper way to address this is to also verify that it’s *your* CDN distribution talking to you. This will usually involve a shared secret or client certificate.

Note that Gerstenkorn verified that this works for AWS CloudFront. I checked Cloudflare and found that it doesn’t work there: Cloudflare’s “Transform Rules” won’t let you “set” the XFF header, and if you “delete” the header, only the pre-existing header is deleted and a new one is added with the actual IP. And it’s similar when attempting to leverage Workers – the actual client IP is still appended to the XFF header after any other manipulation.

Always do strong verification of your CDN! Maybe there are other Cloudflare headers that are important to you and aren’t as protected as XFF. Or maybe you’re using some other CDN. And this general class of attacks might apply to third-party WAFs, etc., depending on how they’re configured.

Many trusted reverse proxy IPs

This is more of a “difficulty” than a pitfall or attack, but this is as good a place as any to fit it in.

In the rightmost-ish algorithm, the “trusted proxy IPs” list might be very large and might involve ranges rather than specific IPs. For example:

There might be a pool of reverse proxies that feed into your server.

That pool might scale out with load. So your trusted IPs will have to include whatever range that pool uses.

You might use a large external service, like Cloudflare. So you’ll need to “trust” all of their very large set of IPs.

Especially in that last case – and *especially* if you’re accepting IPv6 connections – you can’t just have a big list of single IPs that you check. You’ll need to check a list of ranges.

Network architecture changes

So, you’ve set everything up perfectly. Your configuration is exactly right and you’re picking the correct “client IP” every time. Great. It runs quietly and flawlessly for so long that you forget all about it.

And then you change your network architecture.

The scenarios with the less-bad result are when you’re using the rightmost-ish approach and you add a new level of reverse proxy.

You were accepting connections directly from the internet, so you were using `RemoteAddr` for rate-limiting. Then you added a load balancer. Now you’re rate-limiting your load balancer.

You were using a single reverse proxy. You were using the rightmost XFF IP – the one that gets added by that proxy – for rate-limiting. Then you added another level of reverse proxy. Now you’re rate-limiting one of your reverse proxies (whichever is first in the chain).

You had a complex setup of internal reverse proxies. You were rate-limiting by rightmost-ish XFF IP, with your whole internal IP range on the “trusted proxy” list. Then you added Cloudflare in front of it all. Now you’re rate-limiting Cloudflare.

Those are “less bad” because they don’t introduce security flaws, but they’re still going to result in a near-complete inability to process requests.

The “much more bad” scenarios tend to occur when removing reverse proxy levels, and introduce vulnerabilities that you won’t notice.

You were using a single reverse proxy. You were using the rightmost XFF IP – the one that gets added by that proxy – for rate-limiting. Then you decide that you don’t need that extra proxy level and instead you connect your server directly to the internet. But now no part of the XFF is trustworthy and you’re vulnerable to spoofing.

You were using Cloudflare in front of AWS ALB and relying on its `CF-Connecting-IP` header. Then you decide to save some money, drop Cloudflare, and now have requests go directly to ALB. So now you're vulnerable to `CF-Connecting-IP` spoofing.

You were using two levels of reverse proxy in front of your server. You were using the rightmost-ish XFF IP with a "trusted proxy count" of 2 for rate-limiting – so you were always taking the second-from-the-right IP. You decide to remove a level of reverse proxy. Now you're vulnerable to spoofing because the second-from-the-right XFF IP is no longer trustable.

Depending on your logic, it might take an attack before you notice that you're vulnerable.

The takeaway here is pretty obvious – "when network architecture changes, configuration that depends on that architecture also needs to change" – but it can be very difficult to remember to update every dependent config file when you tweak something that "should" work and does appear to work. (Maybe there are fancy network-management tools that automatically push config changes in response to architecture changes? Would be cool.)

The "trusted proxy count" variation of the rightmost-ish algorithm is especially brittle to adding or removing reverse proxies. The "trusted proxy count" (especially with a big internal range) or "rightmost non-private IP" variations can better cope with changing the number of internal proxies (as long as you're not going to zero).

Even worse: *Third-party* network architecture changes

Take a look at the Cloudflare IP list. Notice down at the bottom that there have been two times where Cloudflare removed IPs from the list.

Imagine you had those IPs on your trusted list. Imagine you didn't realize they were removed. *Now who owns those IPs?* Whoever it is can put whatever they want in the XFF, pass it on to your trusted proxy chain, and you'll use it as the "real" IP. Rate limit escape and memory exhaustion.

…The answer to that question is that Cloudflare still owns the IPs (I checked ARIN). But the point isn't about Cloudflare and those particular IP ranges. *Any* CDN or reverse proxy service with a trusted IP list could change their list and cause problems.

(Edit: A reader shared the email Cloudflare sent when they most recently changed their IP set. He pointed out that even though Cloudflare still owns the IPs, they should be considered untrusted. The email mentions that there's an API to get Cloudflare's IPs, which is good. Anyway, my original point wasn't just about Cloudflare, so it remains unchanged.)

`X-Forwarded-For` parser mismatch

This is inspired by JSON interoperability vulnerabilities. These occur when different levels of code or architecture interpret JSON in different ways. So if the JSON parser at one level deals with, say, duplicate object keys by taking the first key and another level deals with it by taking the last key, you can have a

problem. (E.g., an attacker passes a `"username"` value along with a matching password, but then also passes another `"username"` value. If your auth check uses the first username and the business logic uses the second, you're going to access the wrong user data.)

If there's one thing that's certainly true of the XFF header it's that there's an abundance of ways of "interpreting" it. Let's recap some of them and add a fun new one:

Which position? Leftmost? Rightmost? Nth-from-rightmost? And so on.

What value is acceptable? Don't validate at all? Only IPs? Only non-private IPs?

How to split up the list of IPs? Comma? Comma-space?

How to handle multiple XFF headers?

New: How to handle weird characters? What if there's a null byte? Or some other control character? What if there's a UTF-8 sequence?

Any difference in the answers to any of those question marks can result in a mismatch between parsers.

I wish I had a slam-dunk example scenario for this, but I don't. Here are some hand-wavy ones:

You block access to your service to requests from, say, Antarctica. You have a reverse proxy at one level that grabs an XFF IP and checks that. At another level, you have a reverse proxy that grabs a different XFF IP and collects geolocation statistics. You get confused about why you seem to have users connected from Antarctica. (One of them is doing it wrong, but this isn't enough to tell you which.)

At one reverse proxy level, you check a user's incoming IP address against your DB to make sure it's acceptable for that user. At another reverse proxy level, you update that DB. If there's a mismatch, you'll end up too permissive, too restrictive, or both.

More generally… At one reverse proxy level you use the XFF header to determine the client's IP. Allowing the request to proceed is an attestation that the client IP is acceptable for further processing. At a later reverse proxy level, the client IP is again derived from the XFF header and treated as trusted data because it is implicitly attested to by the previous level.[9] A difference between the two levels in XFF parsing introduces a vulnerability.

RFC 7239: Forwarded HTTP Extension, June 2014

After considering comma-vs-comma-space and other parser mismatches, you're probably thinking, "There should be an RFC to concretely specify this." And there is, kind of.

RFC 7239 specifies the `Forwarded` header. Its purpose is to replace and improve `X-Forwarded-For`. The big thing it addresses is that `X-Forwarded-For` (a list of client and proxy IP addresses), `X-Forwarded-Host` (the hostname requested by the client), `X-Forwarded-Proto` (the protocol used by the client; e.g., "https"), and `X-Forwarded-By` (the proxy IP address) are all separate-but-related headers. It becomes very easy to for an intermediary proxy to mess up the relationship between them. To

address this, the `Forwarded` header includes all that information in a single list.

Does the `Forwarded` header fix the security issues that `X-Forwarded-For` has? Not even a little. It can be misused in exactly the same ways that XFF can.

Does it have wide adoption? Not that I can see. It gets mentioned in documentation occasionally, but I don't remember it actually being checked in any of the code I read.

Okay, does the RFC at least make clear how it should be used and not be used? Well, there is this section:

> ##### 8.1. Header Validity and Integrity
>
> The "Forwarded" HTTP header field cannot be relied upon to be correct, as it may be modified, whether mistakenly or for malicious reasons, by every node on the way to the server, including the client making the request.
>
> One approach to ensure that the "Forwarded" HTTP header field is correct is to verify the correctness of proxies and to whitelist them as trusted. This approach has at least two weaknesses. First, the chain of IP addresses listed before the request came to the proxy cannot be trusted. Second, unless the communication between proxies and the endpoint is secured, the data can be modified by an attacker with access to the network.

And that's it.[10]

That warning is strictly true, but it's not very helpful and could be clearer. Would you read those five sentences and then think, "Now I thoroughly understand the danger! It's perfectly clear to me how to use this header in a secure manner."? I wouldn't.

I feel like it should be the responsibility of this RFC not only to specify how to *create* the header but also how to correctly *consume* it.

(Bonus: The RFC adds a variation on IPv6 parsing – quotation marks: "Note that as ":" and "[]" are not valid characters in "token", IPv6 addresses are written as "quoted-string"." E.g., `"[2001:db8:cafe::17]:4711"`.)

Conclusions

Well, that was exhausting. I didn't start out with the intention of writing a dissertation on this godforsaken header.

I have avoided giving this definition until now, just in case anyone skims the article and sees it, but I think we're ready for it:

```
X-Forwarded-For: <client>, <proxy1>, <proxy2>
```

That's what you'll see on basically every page that describes the header. Is it any wonder that misuse of `X-Forwarded-For` is so prevalent?

Let’s summarize some of the things we’ve learned, the wisdom we’ve gained, and the opinions we’ve formed:

Danger on the left, trust on the right. There are, maybe, some situations where the leftmost-ish XFF IP can be used, but only very carefully. If there’s any doubt, use the rightmost-ish.

Any header – or any part of any header – that wasn’t set by your own reverse proxy is fundamentally untrustworthy.

Setting a special header to a helpful value but also just letting it through if it’s already set is a terrible idea. (See: Akamai, Fastly.)

If there’s no universally sane default, there should be no default. And “get the real IP” is a case where there’s no sane default. (See: Chi, Tollbooth.)

Defaults should *not* be insecure. There shouldn’t need to be extra configuration to avoid shooting yourself in the foot. (See: Fastly.)

If you know a function or value is dangerous (spoofable, etc.), put that in your documentation for it in big red letters. Don’t just coyly hint at it. (See: Azure, etc.)

Good specifications (i.e., RFCs) should tell you how to consume a value, not just how to produce it. And if there are different ways to consume that value that make sense in different situations, it should give you the necessary information – with sufficient clarity – to help you make that choice. A reference implementation would also help. [2022-03-24: I wrote a library that I hope will become that reference implementation.]

Inconsistency in security implementations is bad. Pick a tool or cloud service that I didn’t cover here to check for XFF behaviour. Can you guess beforehand what you’ll find with any certainty? You can’t, and that’s bad.

If you’re creating a security tool, product, or feature, you need to have as thorough an understanding of the problem space as possible. Even if you start with solid off-the-shelf components (like a token bucket library and expirable cache for a rate limiter), you still need to be fully aware of, for example, the nature and meaning of a header that’s being used as input. Otherwise you run the risk of introducing a new vulnerability (or two, etc.).

If you’re taking untrusted input – like a header value – and putting it into an ever-growing, pretty-long-lived, in-memory structure (like a rate limiter map), you need to be incredibly careful. (Especially if that structure isn’t *behind* a rate limiter and instead *is* the rate limiter.)

Whenever possible, read the code for your dependencies. It’s hard and a ton of work, but bad security surprises can be worse.

I have avoided saying that you should only use the rightmost-ish XFF IP and never, ever the leftmost. But, seriously, just don’t use it.

Discussion

Comment and discuss at Hacker News.

There have been some interesting comments on HN and Reddit. I'll share some tidbits here.

HN commenter scottlamb pointed out that Rust's method to return a single header value also returns the value of the first such header. The commenter checked and discovered that they were using it wrong.

AWS ELB/ALB has an option to make XFF even worse

HN commenter nickjj brought the AWS ELB/ALB "client port preservation" option to my attention. If enabled, the client port number is appended to the IP added to XFF. Turning that option on will a) violate the de facto standard form of the header, and b) mess up a lot of IP parsing code.

(And if the IP suddenly starts failing, then what? Does the rate limiter logic keep moving to the right until it finds a good IP? If done wrong, that could lead to using untrusted values. Does the rate limiter instead give up? And do what? Fail open? Fail closed? Panic? In a comment on the didip/tollbooth PR for this I talk more about this.)

HN commenter terom said:

> highly recommended to just override the entire XFF header with a single value at the appropriate point in your stack, if at all possible

Which is good advice and I didn't really say in the post. If you have the ability to use one of the "good" single-IP headers, or add your own at your first proxy, that's much better than messing around with XFF.

(The reason I didn't really talk about the custom header is like: I was mostly writing for people who are trying to use what's available rather than doing a lot of proxy tinkering. Or something.)

Go's `net/http/httputil.ReverseProxy` XFF behaviour being re-examined

Right now, `httputil.ReverseProxy` appends the client IP to the XFF header. It looks like they are considering either replacing the existing XFF header by default or adding options to append to, overwrite, or preserve the existing header.

My gut feeling is that the initial more-knobs-to-turn suggestion in the issue is better than the limited-and-awkward thing it seems to be turning into. (I guess I'll express my opinion there.)

After being prompted to look at `httputil.ReverseProxy` and Caddy, I started thinking more about overwriting the `X-Forwarded-For` header (which they both do). I had previously given the idea only a footnote[7], but it deserves more consideration.

The idea is this: In a multi-reverse-proxy scenario, the first proxy replaces any existing XFF header(s) with one containing only `RemoteAddr`. All subsequent proxies (configured to trust the previous proxies) then append to the fresh XFF header.

This approach has an obvious nice property: There are no untrusted values in the XFF list. You can't

possibly choose a spoofed value. But there are also aspects that I don't like.

First of all, I think that it teaches bad XFF hygiene and introduces the possibility of mistakes leading to spoofing vulnerability. Because the XFF header is being replaced by the first proxy, the instructions for use become "use the leftmost". But what if you then swap out your first proxy that doesn't remove the XFF and instead appends to it? Spoofed!

Secondly, there's no configuration-simplicity gain with this approach. Trusted proxies still need to be configured for all proxies but the first. If you're doing that, you can use the rightmost-ish approach. And the rightmost-ish approach is more robust: Is your XFF overwritten? It works! Is your XFF list appended to? It works! Is your XFF list mostly spoofed? It works! And you're helping people to understand the right way to think about the XFF header.

Thirdly, there are still legitimate uses for the leftmost-ish XFF IP (albeit with a lot of warnings attached). If your only mode of operation is to overwrite the XFF header, then you utterly deny those use cases.

Finally, hand-wavingly, I think this violates the spirit of the headers. XFF is supposed to be a comma-separated list of all the IPs involved. Which overwriting it negates. I think that, instead, `X-Real-IP` should be used, set by the first proxy. No need to hijack XFF for this effectively-single-IP purpose.

However, I wouldn't fight to the death against overwriting the XFF. It's still an improvement over the dismal widespread-ness of append-and-use-leftmost.

Envoy's XFF documentation is really something

HN commenter jrockway pointed me at the Envoy Proxy documentation for XFF use. It's not exactly generally educational, but I think it's a really good effort at making sure that Envoy users don't shoot themselves in the foot.

Real-world examples of doing it wrong

"Flask apps on Heroku susceptible to IP spoofing" (2013-05-12). Via eli on HN. It was using the leftmost.

"For many years, a very prominent computer science journal used XFF for guarding access — if you set it to an IP of some well-known universities, you'd be able to download all you want." (HN)

"I remember the source code for a certain simple web app would check if X-Forwarded-For matched a certain IP as a way of granting admin powers. I spoofed it. It was sort of the first "hack" I ever did." (Reddit)

"This was really helpful, I feel good after fixing something I didn't know was even a problem 🤗" (Reddit)

Acknowledgements

Thanks to Rod Hynes for proofreading and providing feedback. All mistakes are mine, of course.

Thanks to Psiphon Inc. for giving me the time to work on this. And employing me.

TODO

Nodejs collapses XFF https://nodejs.org/api/http.html#messageheaders https://old.reddit.com/r/programming/comments/t7lxeb/the_perils_of_the_real_client_ip_or_all_the_wrong/hzkg18l/

add note about single-value header Get being combined list

all projects: if deciding to use leftmost, check for valid/non-private

finish reference implementation

probably add some diagrams

rethink hyphenating rate-limit* (right now I'm not doing it for nouns but am doing it for verbs, and I can't decide what's right)

AWS ALB returns 463 if there are more than 30 XFF IP addresses