Forwarded Header Sabotage | adam-p

We all know by now that the leftmost values in the `X-Forwarded-For` header can be spoofed and only the rightmost IPs – added by your own reverse proxies – can be trusted. The `Forwarded` header (RFC 7239, 2014) has that same problem, and a new one: If the header is parsed correctly, an attacker can *sabotage the whole header*.

Let's take a quick trip to understanding how that can happen and how complicated `Forwarded` parsing can get. (Think about how you'd parse the header as we go.)

## Syntax

A simple `Forwarded` header might look like this:

```
Forwarded: for=1.1.1.1, For=2.2.2.2
```

Here's what a header looks like with an IPv6 value:

```
Forwarded: for=1.1.1.1, FOR="[2001:db8:cafe::17]"
```

Colons and square brackets are not allowed in a "token", so the IPv6 address needs to be quoted. But that means we could have:

```
Forwarded: host="with,comma=equals;semicolon";for=1.1.1.1
```

So now you can't just split by comma – you need to be aware of quoted strings as well.

But characters can also be escaped, so this is also legal:

```
Forwarded: ext="escaped\"quote";for=1.1.1.1
```

The blessed semicolon-separated parameter names in an entry are "for", "by", "host", and "proto" (case-insensitive). There is allowance for "extensions" using other tokens.

Some more legal things:

Anything can be escaped, including backslashes: `\\`. So don't just delete all them all.

Any amount of whitespace around the commas and semicolons.

There can be multiple instances of the header, and they must be considered a single list, top to bottom.

And some illegal things:

Can't have whitespace around the equal sign.

Can't have disallowed characters in parameter names (and not quotable).

Can't have disallowed characters in a parameter value, if not quoted. (Which mean, for example, that an unquoted IPv6 address is illegal.)

IPv6 addresses must have square brackets.

Backslash escaping is only allowed in quoted strings.

There is only one single library I've found that actually correctly parses the header: github.com/lpinca/forwarded-parse.[1] Everything else just does what you were probably thinking after the first couple of steps above:

Split by comma.

Trim whitespace.

Split by semicolon.

Trim the quotes off the value.

Done.

Hilariously, this half-assed, RFC-violating parsing is resistant to sabotage and proper parsing is not.

## Sabotage!

The `Forwarded` header is unique. It is the only header that:

Has untrusted values at the start and trusted values at the end.

Is official and specified.

This combination leads to its susceptibility to sabotage, where the whole header – including the trusted part – needs to be discarded because of chicanery in the untrusted part.

The RFC doesn't (that I can find) provide any special instructions about salvaging the rest of the header if a single entry ("forwarded-element") has a syntax error. So, in theory, the whole header needs to be thrown if a spoofer adds, say, `f*r=` instead of `for=`.

The sabotage is even more fun with an unclosed double-quote:

```
Forwarded: for="1.1.1.1, for=2.2.2.2, for=3.3.3.3
```

It's illegal to have an unclosed quote, so the whole thing is immediately garbage. But even if you wanted to salvage the header… Where do you close the quote? What do you salvage and discard?

## Why is `X-Forwarded-For` not sabotage-able?

Because there's no spec! People just split by comma, trim, and that's it. Your trusted reverse proxy will add `", 1.1.1.1"` and you don't really need to care about what comes before that. (Unless you want the leftmost-ish value, but then you're in the danger zone regardless.)

## Mitigations

"Half-assed, RFC-violating parsing" is the most obvious. If you're using a rightmost-ish value, you should know if your reverse proxies are going to be quoting things, escaping, etc. – and they probably aren't. So do a simple comma-splitting and throw away the stuff on the left.

Doing simple splitting means that you could end up with total garbage in your leftmost values – from spoofing or from valid-but-complicated values. You could probably make your parser more-complex-but-still-not-RFC-compliant by trying to handle quotes and escaping, without discarding everything in the case of bad data. Perhaps your deviation rule could be "no commas allowed in quotes or escaped; they always signal a new entry". Or just don't use a leftmost value.

(Note that differences in parsing at different points – reverse proxies, server, etc. – could result in parser mismatch vulnerabilities.)

At the reverse proxy level, the obvious mitigation is to discard any existing `Forwarded` headers and start fresh, so there are only trusted, well-formed values. If you don't like the idea of discarding potentially valuable forensic information, maybe your reverse proxy could move the previous header value into some new `X-` header before starting fresh.

A variation on unconditionally discarding the `Forwarded` is to check for validity and, if it fails, discard or replace with `for=unknown` (which is also per spec). (I asked lpinca and this is what he prefers.)

Another possibility is to have your reverse proxy not discard the existing `Forwarded` headers but to add a new one. *Maybe* the separate headers could be sanely interpreted separately. But this violates RFC 2616, which says "It MUST be possible to combine the multiple header fields into one "field-name: field-value" pair, without changing the semantics of the message".

So, I can't see how it's possible to adhere to the spec *and* retain the existing header value without falling victim to sabotage.

## Conclusion

I wrote a half-assed, RFC-violating `Forwarded` parser. Then I decided I wanted it to be "correct" and started looking for better implementations.

Along the way I found an Nginx forum conversation from 2017 about adding `Forwarded` support. That conversation is interesting and brought the sabotage potential to my attention, but there's no resolution. It just kind of trails off into hopelessness. (Interestingly, there was strong resistance to discarding the header and thereby losing information.)

Because my `Forwarded` parser is in a project that I am hoping will be a reference implementation for getting the "real" client IP, I really wanted the parser itself to be a reference implementation. But, as disappointing as it is, it seems like being spec-compliant is the wrong move. I also can't assume that a user of the library has the ability to tweak their reverse proxy handling of `Forwarded` (I mean, I could state it as a requirement for use of the library, but that's limiting and error-prone). Documenting the shortcomings seems about as good as it gets.

(Now I have to figure out how to summarize this in the MDN `Forwarded` page update that I'm on the hook for…)

## Addendum

[2022-04-03: Added this section.]

David Moles pointed out that Nginx has instructions for enabling `Fowarded` handling. Part of that is this amazing regex that should be used for validation:

```
^(,[ \t]*)*([!#$%&'*+.^_`|~0-9A-Za-z-]+=([!#$%&'*+.^_`|~0-9A-Za-
z-]+|"([\t \x21\x23-\x5B\x5D-\x7E\x80-\xFF]|\\[\t \x21-\x7E
\x80-\xFF])*"))?(;([!#$%&'*+.^_`|~0-9A-Za-z-]+=([!#$%&'*+.^_`|~0-9A-Za-
z-]+|"([\t \x21\x23-\x5B\x5D-\x7E\x80-\xFF]|\\[\t \x21-\x7E
\x80-\xFF])*"))?)*([ \t]*,([ \t]*([!#$%&'*+.^_`|~0-9A-Za-z-]+=([!#$%&
'*+.^_`|~0-9A-Za-z-]+|"([\t \x21\x23-\x5B\x5D-\x7E\x80-\xFF]|\\[\t
\x21-\x7E\x80-\xFF])*"))?(;([!#$%&'*+.^_`|~0-9A-Za-z-]+=([!#$%&
'*+.^_`|~0-9A-Za-z-]+|"([\t \x21\x23-\x5B\x5D-\x7E\x80-\xFF]|\\[\t
\x21-\x7E\x80-\xFF])*"))?)*)?)*$
```

David and Tim McCormack (via email) both suggested that maybe the `Forwarded` header could be parsed backwards. I initially didn't think that would be much better that just splitting by comma an parsing each pice, but I've come around to the idea. It allows stricter RFC adherence (quoted commas) while still allowing salvaging of rightmost good values.