WordPress REST API Vulnerability: Violating Security’s Rule Zero

Lori MacVittie, 2017-07-02

It's an API economy. If you don't have an API you're already behind. APIs are the fuel driving organizations' digital transformation. We've all heard something similar to these phrases in the past few years. And while they look like marketing, they taste like truth. Because APIs really are transforming organizations and have taken hold as the de facto method of integration - internally, externally, laterally, and vertically. APIs enable mobile apps and things, web apps and middleware to communicate, collaborate, and extricate digital gold (that's data, by the way).

So when there’s a vulnerability discovered in an API, it turns heads. Especially if it’s a vulnerability that is easily exploitable (this one is) and affects a significant number of publicly accessible resources (it does).

> #WordPress REST API vulnerability is already abused in defacement campaigns https://t.co/5i9pYKW4tH by @danielcid

If you’re running WordPress versions **4.7.0** or **4.7.1** *you are vulnerable.* **Stop reading this and go patch it right now. If you can’t for some reason and you’ve got a BIG-IP ASM you can apply these rules right now to protect vulnerable sites. I’m not kidding. I’ll wait right here.**

Okay, now that we’ve got *that* out of the way, I want to talk about APIs and security for a moment because there seems to be some general misunderstandings about REST APIs and security that this vulnerability just happens to illustrate perfectly.

HTTP: The Foundation on which REST APIs are Built

REST stands for Representational State Transfer. Don’t worry too much about what *that* means, because what you really need to understand is that it’s an architectural style. If you were going to list it along with other methods of achieving the same thing (transfer of data between two endpoints) you might list: RPC, CORBA, and SOAP.

A REST API call is an HTTP request where the URI endpoint is typically indistinguishable from a web URI. The request looks the same.

Really, which of these two URIs is a call to an API: `GET /w/thing/snowmobile/id` `GET /a/thing/snowmobile/id`

See what I mean? No difference from outside. There’s no standard out there that requires a REST API contain some identifying attribute or HTTP header that makes it different than a traditional web request. The *Content-Type* suggested by the JSON specification is `application/json` but like the Pirate Code, those are more guidelines than actual rules. Thus, one cannot rely solely on Content-Type to identify a REST API from a standard HTTP request. In fact, good old **x-www-form-urlencoded** is often used to invoke API calls from clients.

For example, here’s an HTTP request to an Express-based API I’m working on (for fun, cause I still do that) using Postman:

```
GET /api/user/1 HTTP/1.1
Host: 192.168.0.57:8080
Content-Type: application/x-www-form-urlencoded
```

```
Cache-Control: no-cache
Postman-Token: 6d2247a1-9923-4774-6b86-7ba334bd497e
```

And here’s one that does a POST, to send data to the API endpoint:

```
POST /api/login HTTP/1.1
Host: 192.168.0.57:8080
Content-Type: application/x-www-form-urlencoded
Cache-Control: no-cache
Postman-Token: 32a29369-5d9d-6952-7de6-8aa4782d694d

name=Webmistress&passwd=xxxx
```

Now, these are API calls. And I’m betting that you noticed a couple of things:

1. The REST API uses **HTTP** verbs like GET, POST, PUT, and DELETE.
2. The URI is – and hold on to your hats now – a standard **HTTP** URI.
3. The Content-Type does not help us determine whether this is a request to an API or a traditional web app.

These first two points are really important because it’s this basic foundation that lets us breathe (a little) easier when it comes to securing APIs. Because we already know a whole lot about HTTP and the myriad ways in which it can be exploited.

REST has become the de facto standard for communication because it’s far simpler than its predecessors and it relies solely on well-understood protocols and platforms, primarily HTTP. Building a REST API, then, means using HTTP to define a set of interfaces through which mobile apps, things, and web apps will communicate. These interfaces, in aggregate, comprise an API. And basically, they’re a set of URI endpoints invoked via HTTP and executed on by a server-side application*.

The fact that the architectural design is REST, and it’s an API, is irrelevant. The same code could have been written to support a traditional web-based system, because the problem isn’t with the API or REST, it’s *with the code that’s processing the input provided*. The WordPress vulnerability is not a vulnerability in its API, per se, but in the way the API *implementation* handles standard, well-understood HTTP mechanisms.

API implementations are often accomplished via frameworks (like Express, can you tell I’m a fan?) that take the tedium out of splitting apart the URI and distilling the paths into “routes” that are then used to call the appropriate function. In addition, this code is responsible for grabbing the query parameters (everything that comes after the ? in a URI, including the key-value pairs) and stuffing them into variables that can be used to do things like update databases, retrieve forum posts, and insert new content into the system.

In this case, the back-end code for the API improperly handles those variables (including some poorly considered authorization logic), failing to properly validate and sanitize it. In Dungeons and Dragons we have the concept of “Rule Zero” and it comes before every other rule out there. It is foundational. There is a similar Rule Zero in security, and it is this:

THOU SHALT NOT TRUST USER INPUT. EVER.

It is Rule Zero that has been violated and thus introduces this vulnerability to WordPress.

Now, you can certainly dig into the details and walk through the code (I like the type-casting logic that really makes this vulnerability work and the lazy sanitization attempt) but the point here is that REST API security relies a lot on the same, well-understood WEB security long preached by OWASP and every other security vendor out there. In fact, one could say it starts with strong, basic web security, the foundation of which is built upon rule zero.

Yes, there are other concerns with APIs with respect to security, and REST introduces some of them because it eschews state. That means it requires a more active or external means of authentication and authorization. And many security folks are still getting up to speed on JSON, as are many of the security services inserted into the data path between endpoints that inspect, scan, and scrub app layer data.

But if you're trying to figure out where to start securing APIs, one of the best places is back to the beginning by laying down a strong set of secure coding practices that begins with Security's Rule Zero.

* Increasingly server-side applications are microservices and in the case of APIs, serverless (Function as a Service) is also gaining traction. And still they rely on endpoint invocation via an HTTP URI. Just sayin. The more things change, the more the stay the same.

F5 Networks, Inc. | 401 Elliot Avenue West, Seattle, WA 98119 | 888-882-4447 | f5.com

F5 Networks, Inc.	F5 Networks	F5 Networks Ltd.	F5 Networks
Corporate Headquarters	Asia-Pacific	Europe/Middle-East/Africa	Japan K.K.
info@f5.com	apacinfo@f5.com	emeainfo@f5.com	f5j-info@f5.com