

Forum:[Hack Forums](#)

Thread: [Explanation of JSON based CSRF gaps in practical examples](#)

Mr8n - October 22, 2018, 7:32 am

Today we will be talking about one of the gaps we often encounter as two breakthrough labs for Web applications, the JSON Based CSRF. Web applications often rely on JSON (JavaScript Object Notation) to send and receive requests from and to the server, and these web applications may be vulnerable to a vulnerability such as CSRF. In this case, exploiting the gap is more complex than exploiting it in normal situations.

Let's say, for example, that we were testing a site hack, and discovered the following request:

PHP Code

```
<?php
```

```
POST /edit/1/username HTTP/1.1
```

Host: example.com

Content-Type: application/json

Content-Length: 19

```
{"username":"test"}
```

All that the request does is send the new username and its test value to the server. It may be clear to you that there are no types of tokens in the application, allowing us to perform CSRF attacks and change the username of the victim without knowing if a page is being visited under our control. The next step is to prepare the HTML file for the attack, but how do we do it in normal cases.

Parameters and their values are sent in the form of parameter = value. The corresponding HTML code is as follows:

PHP Code

```
<?php
```

```
<input type="hidden" name="username" value="test" />
```

But in this case it is sent as {"parameter": "value"}, what can we do?

first attempt :

We can try to manipulate the HTML code to send a request similar to the previous request, but with some differences as follows:

The previous code uses form to make the request, and it should be noted that the value of the enctype for the existing form is text / plain, in order to prevent the browser from doing the URL encoding of the input before sending the request.

In the form input has no value, but the name value is {"username": "test"}, now to send the request in the browser and see what happens (you can see the request sent by using any proxy tool such as Burp Suite):

```
PHP Code
<?php
POST /edit/1/username HTTP/1.1
```

Host: example.com

Content-Type: text/plain

Content-Length: 24

```
={"username":"hacked"}
```

What was specifically done? And why is there a = after the payload sender?

As we specified the form in the HTML code, the request was sent and Content-Type is specified as text / plain, and we see that the payload is sent without URL encoding. But we sent a = sign, because we specified the payload sent as a name value in the input. What the browser does is take the name value followed by a = and then follow the value value, which is empty in this case.

Now, we can say that the attack succeeded when the following two conditions were met:

The web application does not verify that the value of the Content-Type header is application / json

The JSON parser ignores excessive values in the content of the request, such as the = sign in this case

second attempt:

Fortunately, there is a more practical solution than the previous solution, let's look at the modified HTML code below:

In the modified code we have changed the value of name and value in a way that allows us to send a valid JSON value, while allowing us to exploit the gap correctly and successfully. The following is the request sent to the server in this case:

```
PHP Code
<?php
POST /edit/1/username HTTP/1.1
```

Host: example.com

Connection: close

Content-Type: text/plain

Content-Length: 39

```
{"username":"hacked","ignorable":"="}
```

What was done in this case?

As mentioned earlier, the browser sends the value of name then = then the value of the value. In this case, all we did was add an extra value by ignoring the "JSON" in the sender. This method does not work in all cases, since the addition of new parameters to the request sent to the server must be

allowed by the web application, and the values of these parameters should be ignored in the backend code.

You may now be thinking of doing the same attack, but by using XMLHttpRequest, let's look at this and see what happens. We will modify the HTML code to contain the following code:

In this case, a Javascript code was written that would:

Create a new object of its kind called XMLHttpRequest and its name is xhr, this object is simply what sends the request to the server later

Modify xhr by changing the value attribute withCredentials to true, in which case cookies will be sent to the infected site with the sent request

Use the open function to send a POST request to the specified URL

Add a request header to the request whose name is Content-Type and its application / json value until the original request is accurately replicated

Finally, use the send function to actually send the request by converting the payload to the JSON object by using the JSON.stringify function and then passing the resulting payload to the above send function

Finally, the request is sent automatically when you open the page in any browser, now we will open the page and we use Burp Suite or any proxy tool so we can see the sent request:

Why was an OPTIONS request sent instead of a POST request?

When a request is sent to a new method in the new browsers using XMLHttpRequest, the value of the Content-Type header is first confirmed. If it is not equal to text / plain, the OPTIONS request is sent before the original request. This type of request is known as pre-flight requests, Usually as a stage of ascertaining the merit of the site (in this context Origin) is called to send this request.

If the server response contains the pre-flight request on the Access-Control-Allow-Origin header and its value is * (ie origin) or its value is the domain sent to the preflight-request (in our case it is null because we sent it from our PC Not from server), the original request is sent only in this case. The original request is ignored if this is not the case.

So what if we can do if the answer is not the answer?

As we did in the previous example, in this case the request must be sent after changing the Content-Type header to text / plain and then sending the request again, the following image is the request after we changed the header already mentioned:

As we see, the POST request was sent as we wanted and without sending a pre-flight request as in the previous example. The success of the attack in this case depends on all dependence on the two conditions mentioned earlier in the previous example.

Mr.Mr8n

Thanks ?

Ramb0x - *October 22, 2018, 7:50 am*

Do you have an example of webpage like this ?

SlowMehMo - *October 22, 2018, 11:39 am*

Good read, very interesting, I wish more posts in this section was like this, you know, clear thoughts and details, spending time on making your post clear, understandable and informative, instead of "OMG THIS GUY SHOT ME HOW TO STEAL HIS PAYPAL?" and from new member as well, that's refreshing.