
Server Side Template Injection

Author :
Christopher RALEY

Last Revised: July 27, 2019

Contents

1	Penetration Test Environment	2
1.1	Introduction	2
1.2	Pre-installed Packages	2
2	Reconnaissance	3
2.1	Scanning	3
2.2	Exploring the Web Application	4
2.3	Template Engines	5
3	Exploitation	7
3.1	Exploring Objects	7
3.2	Getting to Object	8
3.3	Viewing Available Classes	9
3.4	Remote Code Execution	10

1 Penetration Test Environment

1.1 Introduction

The following tutorial will go through the process of exploiting *server side template injection*. This is found on web applications that generally rely on templates. For our environment we will be going through a web application that is set up using Flask as the Python web framework and Jinja2 which acts as our template.

The machine that we will be using in order to exploit the injection vulnerability will be Kali Linux. Make sure that it is connected to the same network that our machines are connected to.

Estimated Time: 15 minutes

1.2 Pre-installed Packages

There are no additional packages that need to be installed. You just need your Kali machine.

2 Reconnaissance

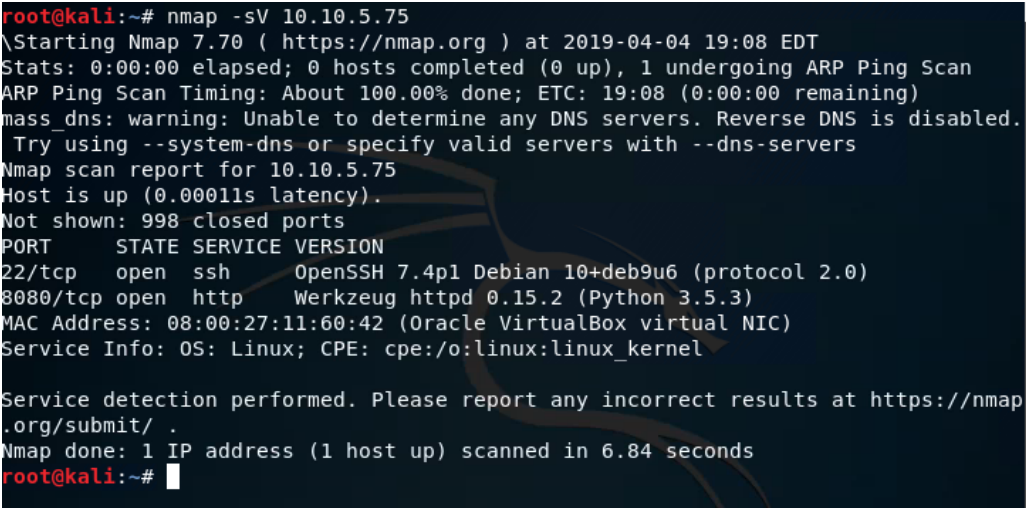
2.1 Scanning

Let's get an idea of how the environment that we are trying to exploit is set up. The machine is held on 10.10.5.75.

Let's do a quick scan on that IP to see what ports are available along with their versions. We will use *nmap* on a new terminal session on Kali:

```
nmap -sV 10.10.5.75
```

You should be able to see the following appear:



```
root@kali:~# nmap -sV 10.10.5.75
Starting Nmap 7.70 ( https://nmap.org ) at 2019-04-04 19:08 EDT
Stats: 0:00:00 elapsed; 0 hosts completed (0 up), 1 undergoing ARP Ping Scan
ARP Ping Scan Timing: About 100.00% done; ETC: 19:08 (0:00:00 remaining)
mass_dns: warning: Unable to determine any DNS servers. Reverse DNS is disabled.
Try using --system-dns or specify valid servers with --dns-servers
Nmap scan report for 10.10.5.75
Host is up (0.00011s latency).
Not shown: 998 closed ports
PORT      STATE SERVICE VERSION
22/tcp    open  ssh      OpenSSH 7.4p1 Debian 10+deb9u6 (protocol 2.0)
8080/tcp   open  http      Werkzeug httpd 0.15.2 (Python 3.5.3)
MAC Address: 08:00:27:11:60:42 (Oracle VirtualBox virtual NIC)
Service Info: OS: Linux; CPE: cpe:/o:linux:linux_kernel

Service detection performed. Please report any incorrect results at https://nmap
.org/submit/ .
Nmap done: 1 IP address (1 host up) scanned in 6.84 seconds
root@kali:~#
```

Figure 1: Nmap Scan of 10.10.5.75

From here it looks like that there are only 2 ports that are available to us: 22 and 8080.

It's nice that we have port 22 available, however, since we don't have any credentials available it is best to move on to the next port.

Typically we would associate 8080 with something like Apache Tomcat or something similar, so it's probably best if we start going there in a browser.

2.2 Exploring the Web Application

Once we open up a browser we can do a quick query in the address bar to *10.10.5.75:8080* (note that we have the colon along with 8080 in order to indicate which port we are trying to communicate with). We are then greeted by the following website:

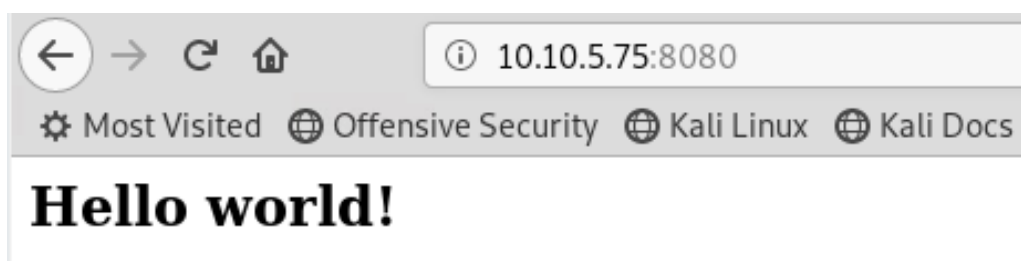


Figure 2: Website on Port 8080

It looks like that we don't have much to go off from here since viewing the page source doesn't give us an inherent details of the implementation.

Let's start by trying to see if there are any queries that we can manipulate or determine. From the message it looks like they are looking for some sort of name query.

We can find that if we type the following:

```
10.10.5.75:8080/?name=bob
```

The web application changes such that the following appears:

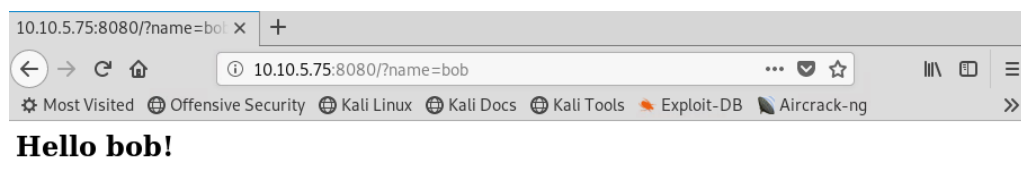


Figure 3: Manipulating the Query

It looks like that we have a lead here! Based on what we have gotten it looks like the website isn't rendered by any simple HTML, PHP, or JavaScript. It's common to see many sites use web applications based off of frameworks such as Django or Flask so let's try to test to see if we can find this out.

2.3 Template Engines

Something that these frameworks are vulnerable to (if sanitization is not implemented) are SSTI's. A common test to find these injections is by appending "{ {} }" to a query. Let's try to input the following:

```
10.10.5.75:8080/?name=bob.{{ 7*'7' }}
```

This is a common way to determine which template engine that the web application is using.

The following appears:

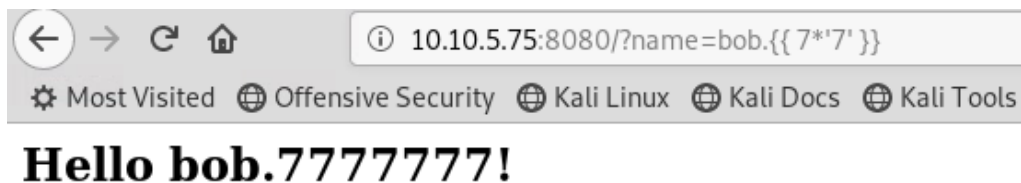


Figure 4: Testing Injection

It turns out that we got 7 iterations of 7's printed out. After looking up how template engines interact we can determine that *Jinja2* is the one that does this.

Note: This is important as something like Twig (another template engine) would have printed out 49 instead

3 Exploitation

3.1 Exploring Objects

We are now ready to get into the nitty gritty side of server side template injection.

Now that we are able to get the version of the template engine, we can use that to our advantage. In Jinja2, we can use the fact that everything is inheriting the Object class (this is similar to object-oriented programming languages in that at the root of all classes there lies the Object class).

There are a couple of functions that we should get acquainted with first:

`__class__`

`__mro__`

`__subclasses__()`

The class function returns the type of whatever you append it to. For example, if you were to do the following: `"hi".__class__` you will get type string.

The mro function will allow you to go up one level with respect to inheritance. For example, if you were to call this function on a string then you will get type Object.

The subclasses function returns an array of all classes that can be inherited by the current object.

Putting this all together, we would then be able to inject an empty string where we can then use the mro function which will allow us to access the Object class. From here we can use the subclasses function which will enable us to see whatever class that is available to us within that Python environment. We learn to use the **Popen** class which will allow us to do remote code execution.

3.2 Getting to Object

Let's take this step by step:



Figure 5: Empty String

Here we are using an empty string and using the class function to make sure that we get the string type. Since it appears on the screen we can continue:



Figure 6: Object Class

Now we build off of the last command by using the mro function which allows us to access the object class.

3.3 Viewing Available Classes

Let's now see what classes are available to us within this environment:

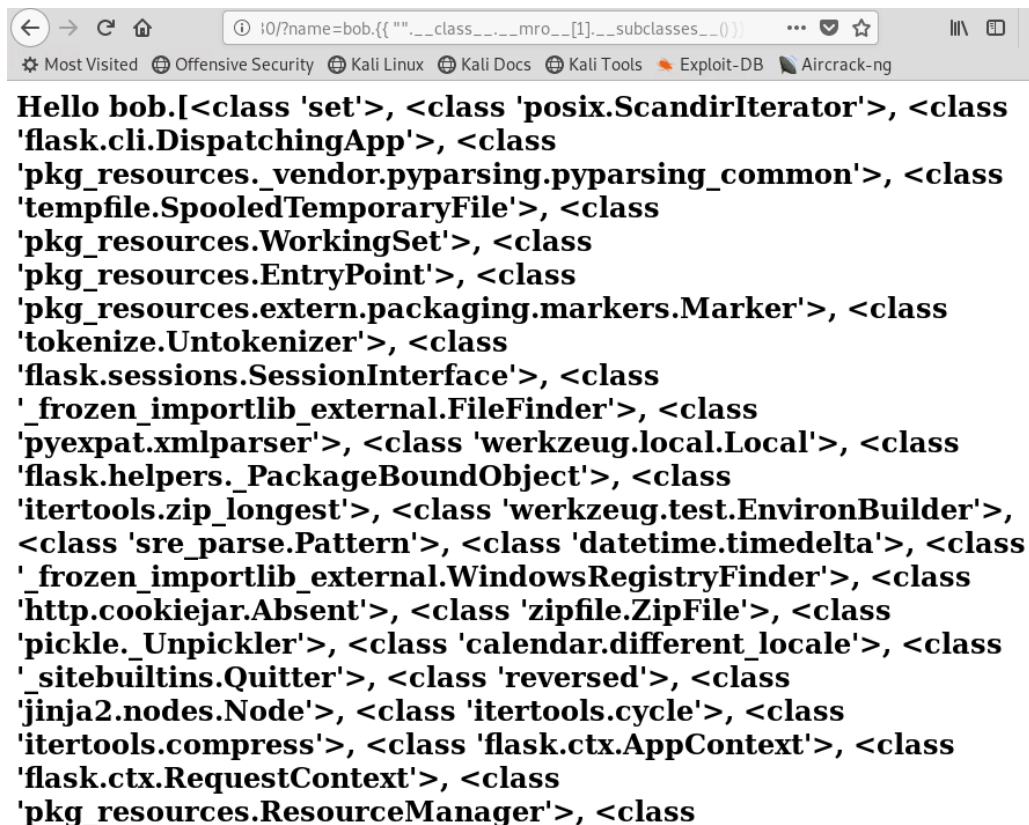


Figure 7: All Classes

Note: We use the index 1 in order to choose the Object class instead of the string class since we were dealing with an array

Looks like that there are many different classes we can use.

Recall that we will be looking for something called **Popen**:

```
'werkzeug.utils.HTMLBuilder'>, <class  
'pkg_resources._vendor.pyparsing.OnlyOnce'>, <class  
'operator.attrgetter'>, <class 'subprocess.Popen'>, <class  
'werkzeug.middleware.shared_data.SharedDataMiddleware'>,  
<class 'http.cookiejar.Cookie'>, <class
```

Figure 8: Finding Popen

Since we are dealing with a very large array of classes, we will need to find the correct index of the Popen. After through some trial-and-error we are able to find that the index of Popen is 466.

Important: The indexes will always vary depending on the Python environment, hence, the same web app might have Popen at a different index on a different machine

We can now utilize functions under the Popen class. There are many functions that can be used.

Here is a reference for the full documentation:

<https://docs.python.org/3/library/subprocess.html>

3.4 Remote Code Execution

We can now create a new Popen object which will allow us to enter in commands:

```
...[466]('whoami',shell=True,stdout=-1).communicate()[0].strip()
```

This command allows us to open up a shell and run the "whoami" command which will tell us the name of the user on the machine.

Here is the result of running the command altogether:

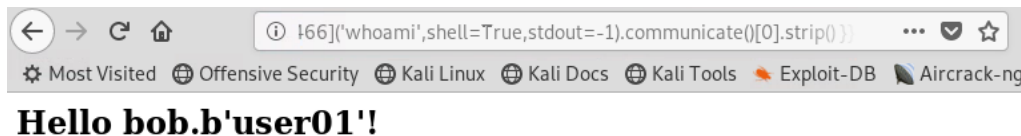


Figure 9: Remote Code Execution

Now we can run any commands underneath the user of the 'user01'! We have successfully injected code which allow allows for remote code execution.