Security and Privacy in Computing (Project 2, Fall 2015) – Xiao Chong Chua

1. To get all the IP addresses of web servers, I wrote a script using the pyshark library in python.

   Filter(s) used:

   http.response

   This filter narrows down the packets to only HTTP responses because that would eliminate the need for me to worry about whether I should look at the source IP or destination IP to get the web server IP addresses, since responses are always FROM the server.

   Using the packets I got from the filter, my script pulled out the source IP addresses with the script and eliminated the duplicates before printing out the list of IP addresses (total of 36) as follows:

```
IP address(es) (36):
155.111.186.252
40.187.57.142
93.119.134.44
155.231.237.70
159.79.22.249
159.70.229.173
35.183.215.204
205.232.201.218
93.199.112.45
33.247.152.113
142.165.192.188
136.93.4.213
143.138.66.97
97.145.19.119
44.111.85.82
159.79.22.198
205.234.49.157
154.87.109.177
44.131.48.102
33.247.152.101
159.79.22.194
154.87.109.40
248.78.109.66
142.165.192.177
34.30.235.180
154.27.68.55
111.4.186.50
143.138.4.147
37.199.226.67
37.120.175.85
205.232.203.30
138.59.102.27
143.179.11.189
251.235.172.148
44.131.51.48
44.131.51.161
```

2. To get all the IP address of the host, I wrote a script using the pyshark library in python.

   Filter(s) used:

   http.request.full_uri contains "../../../"

This filter I put into pyshark's FileCapture() method identifies this "../../../" pattern in HTTP request packets' URL attribute and filters them out. A simple check on python shows that there are only 3 such packets.
Even though it is easy to manually check the 3 packets in the filtered list, I wrote a short script to eliminate duplicate IP addresses and print it out.

```
IP address(es) (1):
42.9.203.117
```

3. To get all the IP address of the host which is doing the brute force attack, I wrote a script using the pyshark library in python.

   Filter(s) used:

   ftp.request.command contains "USER" || ftp.request.command contains "PASS"

   Using this filter as a parameter for the pyshark FileCapture() method, the trace was filtered down to 17 packets because this only returns me the FTP packets that contain USER or PASS as a command. Using this list of packets I wrote a short script to count the number of times each IP address sent packets containing USER or PASS. I got the following output:

```
IP address(es) (3):
172.27.37.232 2
251.215.184.138 2
248.35.162.92 13
```

   From this I know that the IP address **248.35.162.92** made the most requests containing USER or PASS. I then do a manual check on wireshark GUI using "follow TCP stream" to see the packet that comes after the packet containing PASS, to check if it was success or failure. True enough, they were all failure login attempts, hence, I can conclude that this IP address is indeed doing a brute force attack.

4. To get all the unencrypted username and password, I wrote a script using the pyshark library in python.

   Filter(s) used (first part of script):

   telnet contains "USER" || telnet contains "PASS" || telnet contains "username" || telnet contains "password"

   ftp.request.command contains "USER" || ftp.request.command contains "PASS" || ftp.request.command contains "username" || ftp.request.command contains "password"

   http.request.method == POST contains "USER" || http.request.method == POST contains "PASS" || http.request.method == POST contains "username" || http.request.method == POST contains "password"

   pop.request.command contains "USER" || pop.request.command contains "PASS" || pop.request.command contains "username" || pop.request.command contains "password"

Using a script to dynamically construct these 4 filters, I feed them into the pyshark FileCapture() method to get 4 different lists of packets. These filters are different because different protocols deal with user logins differently. POP is similar to FTP, as explained in question 3. HTTP uses the POST request to do it, and telnet is doesn't have any special fields that I can filter by. The resulting lists give me an idea of which protocols I should be focusing my efforts on first.

```
http.request.method == POST 0
pop.request.command 0
ftp.request.command 17
telnet 3
```

From this output, I see that ftp has the most number of packets that I am interested in. From doing the previous question, I confirm this fact.

Filter(s) used (later parts of the script):

ftp.response.code == 230

ftp.response.code == 530

ftp.request.command contains "PASS"

My script's logic is as follows. Looking at ftp packets alone, I construct multiple filters to first get a list of FTP response packets that are successes and a list FTP response of packets that are failures. This gives me a "database" of success or failures I can check later to see if the login attempts were successful or not. Next, I filter out all the packets containing "PASS" from all FTP responses using **ftp.request.command contains "PASS"**. From these "PASS" packets I can able to obtain the acknowledgement packet's sequence number which is contained in one of the fields within the packet. This acknowledgement packet is important because it is where I would be able to find out if the packet that comes after the client sends the password shows a login success or failure. Once I find out all the acknowledgement packets that are successes, I store the TCP stream number, as well as the IP address of the client in this TCP stream. The lists are as follows:

```
IP addresses and TCP streams of interest:
['251.215.184.138', '172.27.37.232'] ['28', '55']
```

Using these client IP addresses and TCP streams, my script pulls out the usernames and passwords from my original list of all FTP request packets containing USER and PASS commands. This is done by pulling out the data within the packets using string manipulation, and storing them into a python dictionary. Finally, I print out my dictionary.

```
Login info found:
{'172.27.37.232': {'PASS': 'thisissosecure', 'USER': 'calrules'},
 '251.215.184.138': {'PASS': '', 'USER': 'anonymous'}}
```

Since the "anonymous" is a username that is in every server by default, we can ignore this. Thus the unencrypted login information we actually want are:

Username: calrules, Password: thisissosecure

5. To get all the IP address of the server with the oldest version of Apache server, I wrote a script using the pyshark library in python.

   Filter(s) used:

   http.server contains "Apache/"

   This filter means that I only want the packets containing "Apache/" in the HTTP layer's server field, to eliminate unnecessary packets for other server types. Tossing this filter into pyshark, the packets are narrowed down to 103. From this list of packets, I did a bit of simple string manipulation with my script to get obtain the version number from each packet, storing them into a list in order of the packets. Using this version number list, I am able to sort my packet list into the order of their Apache server versions from lowest to highest. Then I print out the host IP address and version number of the first packet of my sorted packet list.

```
Server with oldest version:
205.232.201.218 Apache/1.3.28 (Debian GNU/Linux)
```

6. To get all the IP addresses of DNS resolvers without port randomization, I wrote a script using the pyshark library in python.

   Filter(s) used:

   dns.flags == 0x0100

   Tossing this filter into pyshark, I get a list of 418 packets with DNS flag of 0x0100. This hex value of 0x0100 is equals to a 9 bit binary value of 000000100. The following screenshot from wireshark shows what it means.

```
Flags: 0x0100 Standard query
  0... .... .... .... = Response: Message is a query
  .000 0... .... .... = Opcode: Standard query (0)
  .... ..0. .... .... = Truncated: Message is not truncated
  .... ...1 .... .... = Recursion desired: Do query recursively
  .... .... .0.. .... = Z: reserved (0)
  .... .... ...0 .... = Non-authenticated data: Unacceptable
```

   Because we are looking at DNS resolvers, we want it to be a recursive DNS query, so that particular bit representing that is set as 1. The rest should be zero because of the nature of the queries we are looking at.

   Using this list of packets we have, my script simply dumps all the source IP addresses into a dictionary as keys, as well as every source port used for each of these IP addresses, at the same time eliminating duplicate source ports before I store them. From this I get a list of source IP addresses which map to every unique source port that they have used. Next, I just go through my dictionary and check all the IP

addresses to see which ones have only a single unique source port. Script output is as follows:

```
IP addresses of DNS resolvers without port randomization:
205.232.201.218
205.232.201.195
```

These are the IP addresses we want.

7. To get the IP addresses of the 2 TCP endpoints with the largest range of sequence number randomization, I wrote a script using the pyshark library in python.

   Filter(s) used:

   tcp.flags == 0x02

   The reason why I use this filter is because this hex number translates to 000000010 in binary, which represents the flags for a SYN packet. We only need to look at SYN packets because it is the packet that contains the initial sequence number (ISN) of every TCP connection. The list of packets we get from pyshark's FileCapture() method is 108.

   Using this list of SYN packets, my script goes through every packet to pull out the source IP address (client's), as well as the corresponding sequence numbers. These are stored in a dictionary with the IP address as the key, and the sequence numbers as lists tagged to each key.

   Now, using this dictionary, we loop through it to construct 2 separate lists. One list contains all the IP addresses, the other one contains all the ranges of all the respective sequence numbers for each IP address. The range is calculated by getting the difference between the largest and smallest sequence numbers of each IP address's list. Now, my script just sorts the lists in the order of lowest to highest sequence number ranges. The 2 IP addresses (last 2 elements of each list) we want which has the greatest randomization is printed out as output:

   ```
   Top 2 IP addresses with most random seq number ranges:
   251.215.153.250 3890239477
   205.232.201.195 243605121
   ```

   (the number following each IP address is the range calculated)

8. To get all the IP address of the host and destination of the traceroute path, I wrote a script using the pyshark library in python.

   Filter(s) used:

   udp && ip.ttl == 1

   udp && ip.ttl == 2

   …

   udp && ip.ttl == 5

Using python, I iterated through the numbers 1 to 5 for the above filter and got 5 lists of packets from pyshark. The reason for this is because when doing a traceroute the host sends multiple UDP packets containing increasing TTL (time to live) numbers, starting from 1 to 64 (which is the default TTL value for linux machines). This is because the every time the packet reaches a router on the path, it will decrease the TTL value by one, and if this value reaches 0, the packet will "die", and inform the host of the IP address of where it ended. This allows the host to eventually get all the IP addresses of the routers/machines that are on the path between the start and the end. I simply want to check the and see how many matches of similar host IP addresses sending such packets for the first 5 packets to see if I can narrow it down enough.

My script then does a count for the number of packets belonging to each IP address pair (source and destination) for each TTL value using a loop through all the packets we have. The IP address pairs and counts for each TTL are stored in a dictionary.

When this is done, I print out the dictionary as follows:

```
IP addresses and number of packets containing TTLs 1 to 5:
{'172.18.74.109 172.25.130.244': {1: 2},
 '172.29.115.68 176.95.3.38': {1: 1},
 '172.31.250.21 172.25.130.244': {1: 2},
 '205.232.201.10 188.183.25.203': {1: 155},
 '205.232.201.11 188.183.25.203': {1: 165},
 '248.86.240.130 159.79.23.208': {1: 30, 2: 30, 3: 30, 4: 30, 5: 30}}
```

From this output, I can see an obvious pattern for the host IP address 248.86.240.130 and destination IP address 159.79.23.208, since it has exactly 30 packets for each value of TTL checked. We can conclude that these are the host and destination IP address for the traceroute.

9. To get the IP address of the server that has a cross-site scripting vulnerability and the evidence of reflected XSS, I wrote a script using the pyshark library in python.

   Filter(s) used:

   http.request.full_uri contains "%3Cscript%3E" || http.request.full_uri contains "<script>"

   Using this filter with pyshark, I am able to narrow down my packets that are attempting XSS attacks in the trace. This is because in a reflected XSS, the script is sent as a HTTP request packet, with the URL containing the script. The script will most definitely have the script tag "<script>" or "%3Cscript%3E" (%3C and %3E represents < and > in HTML character encoding). This list contains 18 packets.

   Using this list of packets, my script pulls out the TCP stream number, IP address (src and dst), and script string in each packet to form a dictionary to be used later. This maps TCP streams to IP address pairs and the XSS script string.

Next, using this dictionary, I do individual queries using pyshark that looks like the following:

tcp.stream eq 13 && http.response.code == 200

This allows the script to pull out the successful http response packets of each TCP stream I am interested in, which is related to each of the XSS attack attempts.
This is because I want to see if the HTTP response that comes after the XSS attempt is success or not, so that we know a bit more about whether each attack was successful or not. My script will reference to this list of list of HTTP response success packets for each TCP stream later.

For each of the of the lists in this list, we confirm that there is only one response packet, and for each of these responses, we check the HTML data contained within it to see if the script string appears within the HTML data. This is because if the script string appears multiple times in the HTML data, it most likely means that the script got "reflected" back to the user in the webpage. This is evidence for reflected XSS attack.
Using this logic, my script checks each of these responses, and prints out the number of instances that each script string appears in their respective HTTP response webpage data, together with their corresponding server/client IP addresses. The output is as follows:

```
Script contents with list index, ip server/client, http response, number of times the scri
pt appears in html:
alert('struts_sa_surl_xss.nasl')
index 0, 159.79.22.249 251.215.153.250, http OK, 0
alert(123456789)
index 1, 44.111.85.82 251.215.76.253, http OK, 11
document.cookie=%22testhzlg=9267;%22
index 2, 159.79.22.249 251.215.153.250, http OK, 0
alert('struts_sa_surl_xss.nasl')
index 3, 159.79.22.249 251.215.153.250, http OK, 0
alert('struts_sa_surl_xss.nasl')
index 4, 159.79.22.249 251.215.153.250, http OK, 0
foo
index 5, 159.79.22.249 251.215.153.250, http OK, 0
cross_site_scripting.nasl
index 6, 159.79.22.249 251.215.153.250, http OK, 0
```

From this output we can see that for the HTTP response packet sent by the server IP address of 44.111.85.82, there were 11 instances of the script string found on the webpage. I further confirm this by looking at the packet data directly on Wireshark GUI. Hence, this is the server that is vulnerable to reflected XSS.

Note: My scripts were submitted in a single ipython notebook file (.ipynb). I asked Christina, she said it was OK, as long the code can be read. I chose this format because the outputs can be stored together with the file. So you can see the outputs even without running the code.