2018 32nd International Conference on Advanced Information Networking and Applications Workshops

Sniffing and Chaffing Network Traffic in Stepping-Stone Intrusion Detection

Jianhua Yang
TSYS School of Computer Science
Columbus State University
4225 University Ave. Columbus,
GA, 31907 USA
yang_jianhua@ColumbusState.edu

Yongzhong Zhang
College of Science and Technology
Shanghai Open University
288 Guoshun Rd., Shanghai, 200433
China
yzhang@shtvu.edu.cn

Robert King, Tim Tolbert
TSYS School of Computer Science
Columbus State University
4225 University Ave., Columbus,
GA 31907 USA
{king_robert,
Tolbert_tim}@ColumbusState.edu

***Abstract*— Since stepping-stones were widely used to launch attacks over the targets in the Internet, many approaches have been developed to detect stepping-stone intrusion. We found that most of the approaches need to sniff and analyze computer network traffic to detect stepping-stone intrusion. In this paper, we introduce how to make a code to sniff TCP/IP packet. But some intruders can evade detection using TCP/IP session manipulation, such as chaff-perturbation. In order to help researchers understand how a session is manipulated and develop more advanced approaches not only detecting stepping-stone intrusion, but also resisting intruders' manipulation, we present a tool Fragroute which can be used to inject meaningless packets into a TCP/IP session across the Internet.**

Keywords—Stepping-stone intrusion, Packet sniffing, Packet chaffing, Intrusion Detection, Network Traffic, Network Security

I. INTRODUCTION

Since the Internet was widely used in our daily life, more and more cyberattacks happen every day. To escape from being detected, most attackers use stepping-stone [1] to invade the computer hosts they are interested in. The attackers using stepping-stone to lunch their attacks are called stepping-stone intruders. The attacks are called stepping-stone intrusion. Since this type of attack was first introduced in the late 1990s, there have been many detection algorithms developed to detect stepping-stone intrusion. Stepping-stone intruders also try their best to make use of some advanced computer techniques to evade the detection.

S. Staniford-Chen, and L. T. Heberlein proposed to detect stepping-stone intruders by comparing the contents of the traffic from incoming and outgoing connections respectively of a host [2]. Obviously if the two connections of a computer host share the same/similar contents, the host might be used as a stepping-stone. This approach can be defeated if intruders use SSH/OpenSSH to make a connection which is encrypted. Instead of using computer network traffic contents, Y. Zhang and V. Paxson [1], and K. Yoda and H. Etoh [3] proposed different approaches respectively, but with the similar idea to detect stepping-stone intrusion by making use of time, packet sequence number, and packet size which could not be encrypted in a session established by using SSH/OpenSSH. Some other approaches were also proposed after 2000 to detect stepping-stone by using the count of network packets [4, 5], and the count of network traffic Round-Trip Time (RTT) [6]. All the above approaches proposed to detect stepping-stone intrusion compare an incoming connection of a host with any outgoing connection of the host to see if the two connections are in the same extended connection chain. Two connections belonging to the same connection chain are called relayed. If there were any relayed connection pair, the host would be used as a stepping-stone. These approaches are called host-based stepping-stone intrusion detection model since they use TCP/IP sessions to a host and the ones from the host. Even though many different methods proposed to detect stepping-stone intrusion may use computer network traffic in a slightly different way, they all need to sniff TCP/IP packets.

Different from host-based stepping-stone intrusion detection model, network-based stepping-stone intrusion detection model can bring down detection false-positive error. In this model, instead of checking the relation between incoming and outgoing connections of a host, it estimates the length of the connection chain from the detecting host which is called Sensor to the target host. Host-based stepping-stone intrusion detection approaches can incur false-positive detection error because there are some applications which use computer hosts as stepping-stones. But it is rare to find that more than three hosts are used as stepping-stones in legal applications. If the number of connections between a sensor and the target host is more than three, it is highly suspicious that the sensor is used by a stepping-stone intruder. The first method to detect stepping-stone intrusion by estimating the length of the connection chain from a sensor to the target was proposed by K.H. Yung [7]. Yung's idea is to use the RTT from the sensor to its adjacent host along the downstream connection as a base unit to gauge the length of the whole downstream connection chain. Another approach to detect stepping-stone intrusion through estimating the length of a downstream connection chain more accurately was proposed by J. Yang, S.-H.S. Huang, and M. D. Wan [8]. With this method, the number of connections included in a connection chain can be computed by using clustering-partitioning data mining approach. However, sniffing TCP/IP packets are needed in almost all the approaches to detect stepping-stone intrusion.

Some studies show that most of the approaches proposed to detect stepping-stone intrusion are vulnerable to intruders' manipulation, such as time-jittering, and chaff-perturbation. The idea to manipulate a TCP/IP session via time-jittering technique is to hold a packet in the session for a while before it is released. Chaff-perturbation technique is to inject some meaningless packets into a TCP/IP session to change some features of the session intentionally. The approach proposed by

978-1-5386-5395-1/18/$31.00 ©2018 IEEE
DOI 10.1109/WAINA.2018.00137

Y. Zhang, and V. Paxson [YZhang200] to detect stepping-stone intrusion detection by comparing the time-thumbprint of a TCP/IP session would not work if the session is time-jittered and chaffed. Correspondingly the approach [3] to detect stepping-stone intrusion by checking the deviation of two connections would also not work if the connections were manipulated by time-jittering and chaff-perturbation since the method uses the timestamp of each packet captured. Chaff-perturbation manipulation can make lots of other stepping-stone intrusion detection approaches unable to work, such as the approaches in [4, 5] using the count of TCP/IP packets. K.H. Yung's approach [7] using RTT to detect stepping-stone intrusion may resist time-jittering manipulation but still be affected by intruders' chaff-perturbation. The approaches proposed by J.Yang [6, 8] can resist intruders' time-jittering and chaff-perturbation manipulation, but only to a certain degree. If the chaff-rate is not bounded, the approaches [6, 8] may also be defeated.

From the above discussion, we understand that sniffing computer network traffic is important for stepping-stone intrusion detection. There are some existing tools, such as Wireshark, TCPDump, and Snort, can be used to capture TCP/IP packet, but in order to be seamlessly integrated into stepping-stone detection algorithm, most researchers need to make their own code to sniff computer network traffic on their own format. In this paper, we introduce how to make a code using Pcap package in C language to capture TCP/IP packet. To help researchers to develop approaches to not only detect stepping-stone intrusion, but also resist intruders' time-jittering and chaff-perturbation manipulation, we also present a tool used by the most of intruders to inject meaningless packets into a TCP session.

The paper is organized as the following. Making a code to sniff network traffic is introduced in Section II. In Section III, the tool to chaff network traffic will be presented. The whole paper is concluded in Section IV.

II. Sniffing Network Traffic

A. *Sniffing Network Traffic by Existing Tools*

A computer can exchange information through computer network in packet format. Depending on the type of communication, different types of packet might be applied. The typical one is TCP/IP packet family. As long as mention network traffic is mentioned in this paper, we always mean TCP/IP packets.

Capturing network traffic is very important. First it can help students to understand how network communication works, and know how information is encapsulated into packets, delivered, and routed to destination hosts. Second, in many research projects, such as stepping-stone intrusion detection, network traffic behavior were studied extensively. There are many tools developed to capture network traffic, such as Wireshark, Cain and Abel, Tcpdump, Kismet, Ettercap, Netstumbler, Dsniff, Ntop, Ngrep, EtherApe, NetworkMiner, P0f, insider, KisMAC, and etc. But among them, the most popular ones are Wireshark and Tcpdump. Wireshark is mainly used under Windows system, but Tcpdump works very well in Linux/Unix system.

Wireshark, also known as Ethereal, is an open-source multi-platform network protocol analyzer. This tool allows users to examine data from a live computer network connection. Wireshark has lots of powerful features including a rich display filter, as well as a capturing filter. Display filter allows users to capture all type of packets, but only allow the ones the user needs to be displayed. Capturing filter allows a certain type of packet to be captured, and displayed Wireshark supports hundreds of protocols and media types. This tool can be obtained from its official website: http://www.wireshark.org/download.html for free. But before installing Wireshark in your computer, please make sure if your computer supports LibPcap (Unix/Linux System) or WinPcap (Windows System).

When running Wireshark in your computer, you will be asked to select network interface. As long as the interface is selected, Wireshark can run to start monitoring the traffic passing through the interface. Wireshark has five major panels including command menus, display filter window, packet listing window, packet header detail window, and packet contents window in both ASCII and hexadecimal format. Among the two filters supported by Wireshark, display filter can be defined at any time, but capturing filter must be defined before sniffing starts.

Tcpdump is a command line network sniffer used to capture network packets. It has many options to allow a user to browse the packets dumped in terminal, create a pcap file and set up a filter to capture only required packets, and directly monitor the capturing of a remote system in any other Linux system. Tcpdump works on most Unix-like operating systems: Linux, Solaris, BSD, macOS, HP-UNIX, Android, and AIX. In the systems, tcpdump uses the libpcap library to capture packets. Tcpdump was originally written in 1987 by Van Jacobson, Craig Leres and Steven McCanne who were, at the time, working in the Lawrence Berkeley Laboratory Network Research Group. By the late 1990s there were numerous versions of tcpdump distributed as part of various operating systems, and numerous patches that were not well coordinated, so Bill Fenner created www.tcpdump.org in 1999. In some Unix-like operating systems, a user must have super-user privilege to use tcpdump because the packet capturing mechanisms on the systems require elevated privileges. In Linux, using "man tcpdump" can get its usage and tons of options.

B. *Sniffing Network Traffic by Making Code*

The way to make a code to sniff computer network traffic is to use pacp (packet capture) package. Pcap consists of an application programming interface (API) for capturing network traffic. Unix-like systems implement pcap in the libpcap library, and Windows uses a port of libpcap known as WinPcap.

We take an example, capturing raw IP packets, to examine the steps to sniff packets by making a program under Linx/Unix-like system. There are four steps to follow in making code to sniff packets: 1) open a packet capture socket; 2) start packet capture loop; 3) parse and display packets; 4) Terminate program.

Open a packet capture socket: A socket is an endpoint for network communication that is identified in a program with a socket descriptor. Opening a packet capture socket involves a series of libpcap calls that are encapsulated in open_pcap_socket() function as shown in [9]. There are a couple of steps needed to open a packet capture socket. The first step is to select a network device using function pcap_lookupdev(). The second step is to open the network device selected for live capture using function pcap_open_live(). The third step is to call function pcap_lookupnet() to get the network address and subnet mask. The fourth step is to compile a packet capture filter by calling function pcap_compile(). The last step is to install the compiled packet filter program into the packet capture device. This causes libpcap to start collecting the packets with selected filter. The following sample code shows the four steps in opening a packet capture socket.

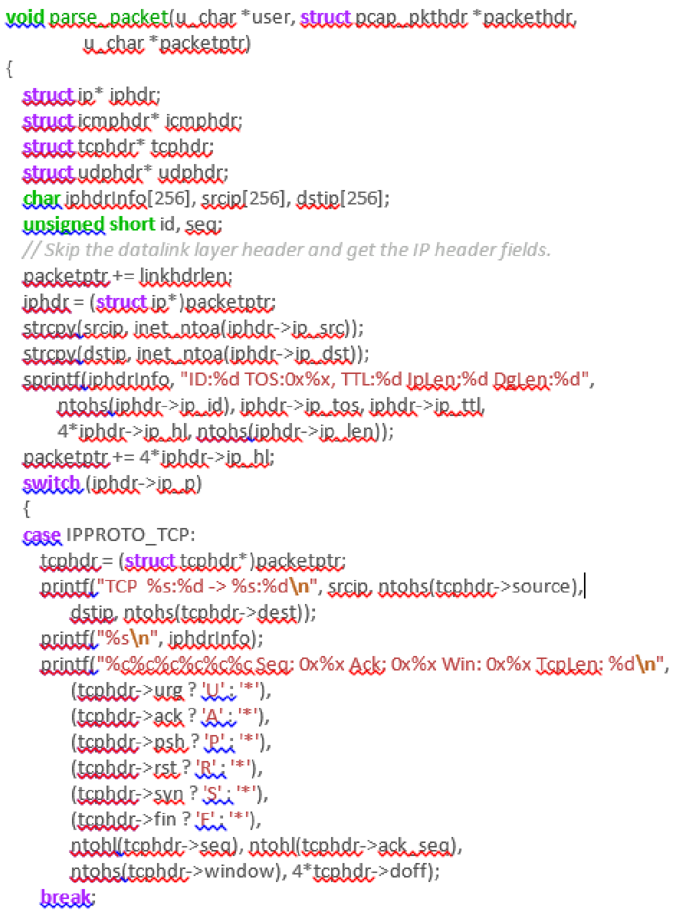
```
pcap_t* open_pcap_socket(char* device, const char* bpfstr)
{
  char errbuf[PCAP_ERRBUF_SIZE];
  pcap_t* pd;
  uint32_t srcip, netmask;
  struct bpf_program bpf;

  // If no network interface (device) is specified, get the first one.
  if (!*device && !(device = pcap_lookupdev(errbuf)))
  {
    printf("pcap_lookupdev(): %s\n", errbuf);
    return NULL;
  }
  // Open the device for live capture, as opposed to reading a packet
  // capture file.
  if ((pd = pcap_open_live(device, BUFSIZ, 1, 0, errbuf)) == NULL)
  {
    printf("pcap_open_live(): %s\n", errbuf);
    return NULL;
  }
  // Get network device source IP address and netmask.
  if (pcap_lookupnet(device, &srcip, &amp;netmask, errbuf) < 0)
  {
    printf("pcap_lookupnet: %s\n", errbuf);
    return NULL;
  }
  // Convert the packet filter expression into a packet filter binary.
  if (pcap_compile(pd, &bpf, (char*)bpfstr, 0, netmask))
  {
    printf("pcap_compile(): %s\n", pcap_geterr(pd));
    return NULL;
  }
  // Assign the packet filter to the given libpcap socket.
  if (pcap_setfilter(pd, &bpf) < 0)
  {
    printf("pcap_setfilter(): %s\n", pcap_geterr(pd));
    return NULL;
  }
  return pd;
}
```

Start packet capture loop: Libpcap provides three functions to capture packets: pcap_next(), pcap_dispatch(), and pcap_loop(). Since function pcap_next() can only grab one packet at the time to be called. So the programmer must call this function in a loop to receive multiple packets. The other two functions pcap_loop and pcap_dispatch() can loop automatically to receive multiple packets. Datalink type can be determined by calling pcap_datalink(), and then start packet capture. The following sample program uses pcap_loop() to sniff multiple packets. In this code, first to determine the datalink type by calling pcap_datalink(), and then start packet capture loop.

```
void capture_loop(pcap_t* pd, int packets, pcap_handler func)
{
  int linktype;
  // Determine the datalink layer type.
  if ((linktype = pcap_datalink(pd)) < 0)
  {
    printf("pcap_datalink(): %s\n", pcap_geterr(pd));
    return;
  }
  // Set the datalink layer header size.
  switch (linktype)
  {
  case DLT_NULL:
    linkhdrlen = 4;
    break;
  case DLT_EN10MB:
    linkhdrlen = 14;
    break;
  case DLT_SLIP:
  case DLT_PPP:
    linkhdrlen = 24;
    break;
  default:
    printf("Unsupported datalink (%d)\n", linktype);
    return;
  }
  // Start capturing packets.
  if (pcap_loop(pd, packets, func, 0) < 0)
    printf("pcap_loop failed: %s\n", pcap_geterr(pd));
}
```

Parse and display packets: The general technique for parsing packets is to set a character pointer to the beginning of the packet buffer then advance this pointer to a particular protocol header by the size in bytes of the header that precede it in the packet. The header can then be mapped to an IP, TCP, UDP, and ICMP header structure by casting the character pointer to a protocol specific structure pointer. A parse_packet() function starts off by defining pointers to IP, TCP, UDP and ICMP header structures. The packet pointer is advanced past the datalink header by the number of bytes corresponding to the datalink type determined in capture_loop(). Casting the packet pointer to struct tcphdr and struct udphdr pointers gives us access to TCP and UDP header fields respectively. And the struct icmphdr pointer enables us to display ICMP packet type and code along with the source and destination IP addresses. The following sample code shows the steps to parse and display packets, such as TCP packets which are used to detect stepping-stone intrusion.

Terminate Capturing: The last step is to terminate the packet capture by interrupt signals SIGNIT, SIGTERM, and SIGQUIT through calling function bailout() which displays the packet count, closes the packet capture socket then exits the program.



```
void parse_packet(u_char *user, struct pcap_pkthdr *packethdr,
          u_char *packetptr)
{
  struct ip* iphdr;
  struct icmphdr* icmphdr;
  struct tcphdr* tcphdr;
  struct udphdr* udphdr;
  char iphdrInfo[256], srcip[256], dstip[256];
  unsigned short id, seq;
  // Skip the datalink layer header and get the IP header fields.
  packetptr += linkhdrlen;
  iphdr = (struct ip*)packetptr;
  strcpy(srcip, inet_ntoa(iphdr->ip_src));
  strcpy(dstip, inet_ntoa(iphdr->ip_dst));
  sprintf(iphdrInfo, "ID:%d TOS:0x%x, TTL:%d IpLen:%d DgLen:%d",
      ntohs(iphdr->ip_id), iphdr->ip_tos, iphdr->ip_ttl,
      4*iphdr->ip_hl, ntohs(iphdr->ip_len));
  packetptr += 4*iphdr->ip_hl;
  switch (iphdr->ip_p)
  {
  case IPPROTO_TCP:
    tcphdr = (struct tcphdr*)packetptr;
    printf("TCP  %s:%d -> %s:%d\n", srcip, ntohs(tcphdr->source),
        dstip, ntohs(tcphdr->dest));
    printf("%s\n", iphdrInfo);
    printf("%c%c%c%c%c%c Seq: 0x%x Ack: 0x%x Win: 0x%x TcpLen: %d\n",
        (tcphdr->urg ? 'U' : '*'),
        (tcphdr->ack ? 'A' : '*'),
        (tcphdr->psh ? 'P' : '*'),
        (tcphdr->rst ? 'R' : '*'),
        (tcphdr->syn ? 'S' : '*'),
        (tcphdr->fin ? 'F' : '*'),
        ntohl(tcphdr->seq), ntohl(tcphdr->ack_seq),
        ntohs(tcphdr->window), 4*tcphdr->doff);
    break;
```

III. CHAFFING NETWORK TRAFFIC

As we discussed in the above, stepping-stone intrusion detection is a technology developed for purpose of network security and attempt to identify and isolate intrusions against computer systems. As intrusion detection technologies develop, so do intrusion detection countermeasures. One such countermeasure is known as packet injection or “chaffing”. While there are several tools available that can accomplish this task, we will be covering a typical one called Fragroute.

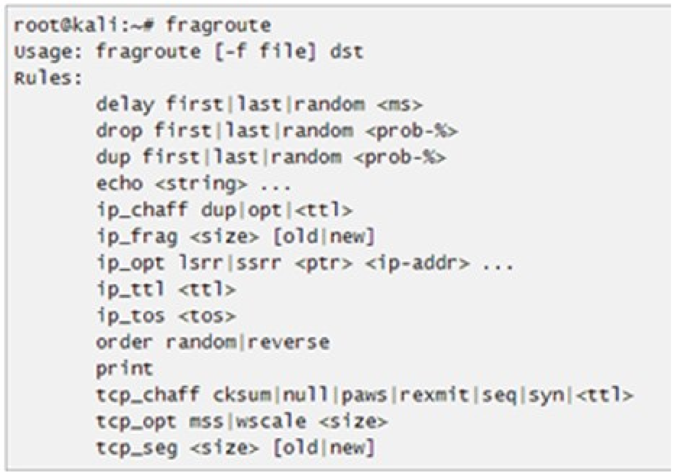
A. Introduction

Fragroute is a tool that was designed for the purpose of testing network intrusion detection systems and firewalls. It is a command line based application that intercepts, modifies, and rewrites network traffic that is destined for a specified host and provides options for capturing/monitoring data, reordering packets, injecting arbitrary packets of an arbitrary size / length into a data stream and evading detection. It is capable of implementing most of the attacks that are described in the paper [10] by Thomas H. Ptacek and Timothy N. Newsham from January 1998. This tool is supported on BSD, Linux, Solaris, and Windows 2000 platforms.

Fragroute uses a configuration file found in **/usr/local/etc/fragroute.conf** to specify certain rules and parameters that can be modified and changed as necessary. The following is the default configuration ruleset:

```
tcp_seg 4  new
tcp_chaff  paws
order  random
print
```

Other configuration options are shown in Figure 1 as the following:



```
root@kali:~# fragroute
Usage: fragroute [-f file] dst
Rules:
        delay first|last|random <ms>
        drop first|last|random <prob-%>
        dup first|last|random <prob-%>
        echo <string> ...
        ip_chaff dup|opt|<ttl>
        ip_frag <size> [old|new]
        ip_opt lsrr|ssrr <ptr> <ip-addr> ...
        ip_ttl <ttl>
        ip_tos <tos>
        order random|reverse
        print
        tcp_chaff cksum|null|paws|rexmit|seq|syn|<ttl>
        tcp_opt mss|wscale <size>
        tcp_seg <size> [old|new]
```

Figure 1: Fragroute configuration

delay - Delay the delivery of the first, last, or a randomly selected packet from the queue by *ms* milliseconds.
drop - Drop the first, last, or a randomly selected packet from the queue with a probability of prob-% percent.
dup - Duplicate the first, last, or a randomly selected packet from the queue with a probability of prob-% percent.
echo - echoes the string argument(s) to standard output.
ip_chaff - Interleave IP packets in the queue with duplicate IP packets containing different payloads, either scheduled for later delivery, carrying invalid IP options, or bearing short time-to-live values.
ip_frag - Fragment each packet in the queue into size-byte IP fragments, preserving the complete transport header in the first fragment. Optional fragment overlap may be specified as old or new, to favor newer or older data.
ip_opt - Add IP options to every packet, to enable loose or strict source routing. The route should be specified as list of IP addresses, and a bytewise pointer into them (e.g. the minimum ptr value is 4).
ip_ttl - Set the IP time-to-live value of every packet to ttl.
ip_tos - Set the IP type-of-service bits for every packet to tos.
order - Re-order the packets in the queue randomly, or in reverse.
print - Print each packet in the queue in tcpdump-style format.
tcp_chaff - Interleave TCP segments in the queue with duplicate TCP segments containing different payloads, either

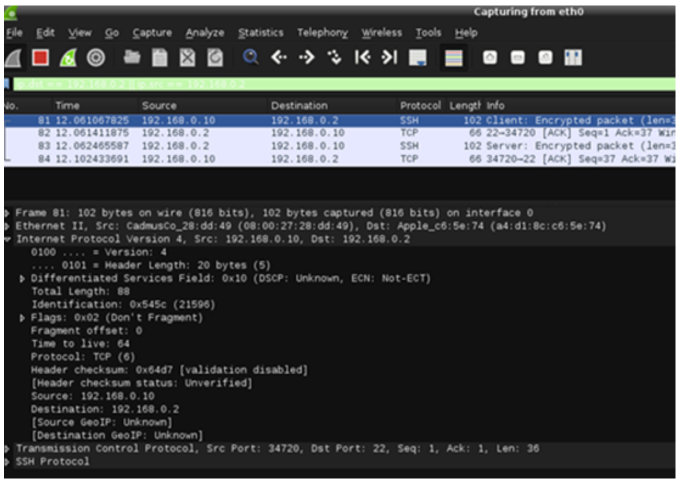
bearing invalid TCP checksums, null TCP control flags, older TCP timestamp options for PAWS elimination, faked retransmits scheduled for later delivery, out-of-window sequence numbers, requests to re-synchronize sequence numbers mid-stream, or short time-to-live values.

tcp_opt - Add TCP options to every TCP packet, to set the maximum segment size or window scaling factor.

tcp_seg - Segment each TCP data segment in the queue into size-byte TCP segments. Optional segment overlap may be specified as old or new, to favor newer or older data.

B. *Methodology*

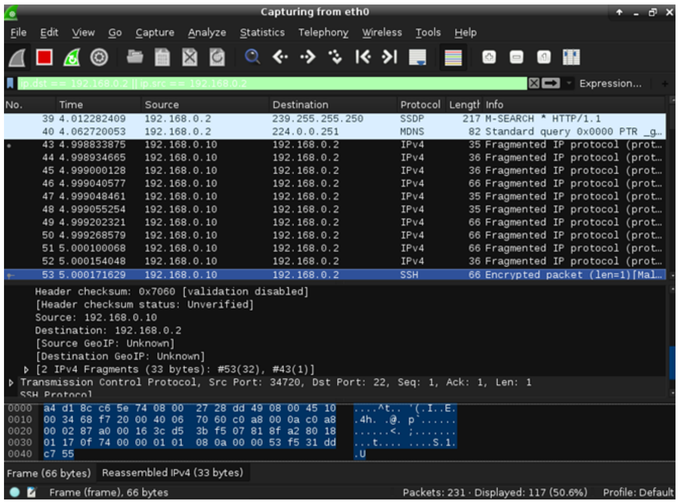
In our experimental setup, we have a Kali Linux machine and an Ubuntu Linux machine. When we connect to the Ubuntu machine from the Kali machine by using SSH, and enter a character in the terminal, such as “s”, we observe the traffic through Wireshark. Below shown in Figure 2 is an image of the communication between the two machines:


Capturing from eth0
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
ip.dst == 192.168.0.2 || ip.src == 192.168.0.2

```
No.   Time          Source        Destination   Protocol Lengtl Info
   81 12.061067825  192.168.0.10  192.168.0.2   SSH      102 Client: Encrypted packet (len=3
   82 12.061411875  192.168.0.2   192.168.0.10  TCP       66 22→34720 [ACK] Seq=1 Ack=37 Win
   83 12.062465587  192.168.0.2   192.168.0.10  SSH      102 Server: Encrypted packet (len=3
   84 12.102433691  192.168.0.10  192.168.0.2   TCP       66 34720→22 [ACK] Seq=37 Ack=37 Wi

Frame 81: 102 bytes on wire (816 bits), 102 bytes captured (816 bits) on interface 0
Ethernet II, Src: CadmusCo_28:dd:49 (08:00:27:28:dd:49), Dst: Apple_c6:5e:74 (a4:d1:8c:c6:5e:74)
Internet Protocol Version 4, Src: 192.168.0.10, Dst: 192.168.0.2
    0100 .... = Version: 4
    .... 0101 = Header Length: 20 bytes (5)
  Differentiated Services Field: 0x10 (DSCP: Unknown, ECN: Not-ECT)
    Total Length: 88
    Identification: 0x545c (21596)
  Flags: 0x02 (Don't Fragment)
    Fragment offset: 0
    Time to live: 64
    Protocol: TCP (6)
    Header checksum: 0x64d7 [validation disabled]
    [Header checksum status: Unverified]
    Source: 192.168.0.10
    Destination: 192.168.0.2
    [Source GeoIP: Unknown]
    [Destination GeoIP: Unknown]
Transmission Control Protocol, Src Port: 34720, Dst Port: 22, Seq: 1, Ack: 1, Len: 36
SSH Protocol
```

Figure 2: Wireshark Capture

To start Fragroute we enter fragroute <DestinationIP>, which in this case is 192.168.0.2, so we enter fragroute 192.168.0.2 using the default configuration and found in **/etc/fragroute.conf** the chaffed results as shown in Figure 3.

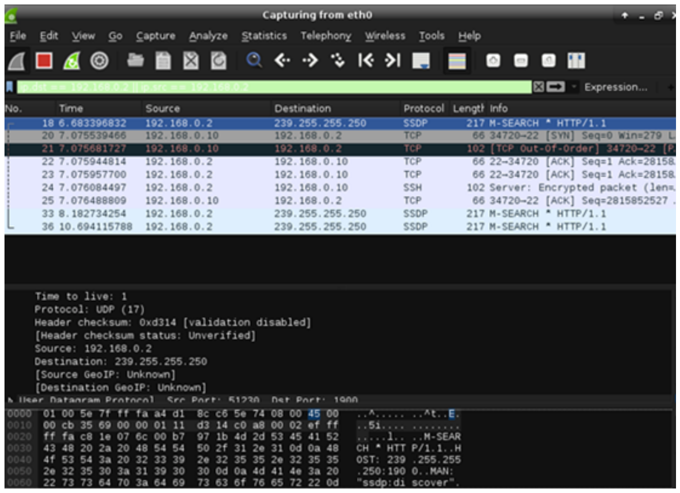

Capturing from eth0
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
ip.dst == 192.168.0.2 || ip.src == 192.168.0.2
Expression...

```
No.   Time         Source        Destination      Protocol Lengtl Info
   39 4.012282409  192.168.0.2   239.255.255.250  SSDP     217 M-SEARCH * HTTP/1.1
   40 4.062720053  192.168.0.2   224.0.0.251      MDNS      82 Standard query 0x0000 PTR _g…
   43 4.998833875  192.168.0.10  192.168.0.2      IPv4      35 Fragmented IP protocol (prot…
   44 4.998934665  192.168.0.10  192.168.0.2      IPv4      36 Fragmented IP protocol (prot…
   45 4.999000128  192.168.0.10  192.168.0.2      IPv4      36 Fragmented IP protocol (prot…
   46 4.999040577  192.168.0.10  192.168.0.2      IPv4      66 Fragmented IP protocol (prot…
   47 4.999048461  192.168.0.10  192.168.0.2      IPv4      35 Fragmented IP protocol (prot…
   48 4.999055254  192.168.0.10  192.168.0.2      IPv4      35 Fragmented IP protocol (prot…
   49 4.999202321  192.168.0.10  192.168.0.2      IPv4      66 Fragmented IP protocol (prot…
   50 4.999268579  192.168.0.10  192.168.0.2      IPv4      66 Fragmented IP protocol (prot…
   51 5.000100068  192.168.0.10  192.168.0.2      IPv4      66 Fragmented IP protocol (prot…
   52 5.000154048  192.168.0.10  192.168.0.2      IPv4      36 Fragmented IP protocol (prot…
   53 5.000171629  192.168.0.10  192.168.0.2      SSH       66 Encrypted packet (len=1)[Mal…

    Header checksum: 0x7060 [validation disabled]
    [Header checksum status: Unverified]
    Source: 192.168.0.10
    Destination: 192.168.0.2
    [Source GeoIP: Unknown]
    [Destination GeoIP: Unknown]
  [2 IPv4 Fragments (33 bytes): #53(32), #43(1)]
Transmission Control Protocol, Src Port: 34720, Dst Port: 22, Seq: 1, Ack: 1, Len: 1
SSH Protocol

0000  a4 d1 8c c6 5e 74 08 00  27 28 dd 49 08 00 45 10   ....^t.. '(.I..E.
0010  00 34 68 f7 20 00 40 06  70 60 c0 a8 00 0a c0 a8   .4h. .@. p`......
0020  00 02 87 a0 00 16 3c d5  3b f5 07 81 8f a2 80 18   ......<. ;.......
0030  01 17 0f 74 00 00 01 01  08 0a 00 00 53 f5 31 dd   ...t.... ....S.1.
0040  c7 55                                              .U

Frame (66 bytes)  Reassembled IPv4 (33 bytes)
Frame (frame), 66 bytes          Packets: 231 · Displayed: 117 (50.6%)   Profile: Default
```

Figure 3: Chaff with Fragroute

This outputs the segments of all TCP data to a host into forward-overlapping 4-byte segments (favoring newer data), interleaves with overwriting, random chaff segments bearing older timestamp options for PAWS elimination, reorders randomly, and prints to standard output.

In our experiments, we used the tcp_chaff configuration. When changing the configuration file to contain only tcp_chaff syn, we get the following as shown in Figure 4.


Capturing from eth0
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
ip.dst == 192.168.0.2 || ip.src == 192.168.0.2
Expression...

```
No.   Time          Source        Destination      Protocol Lengtl Info
   18 6.683396832   192.168.0.2   239.255.255.250  SSDP     217 M-SEARCH * HTTP/1.1
   20 7.075539466   192.168.0.10  192.168.0.2      TCP       66 34720→22 [SYN] Seq=0 Win=279 L
   21 7.075681727   192.168.0.10  192.168.0.2      TCP      102 [TCP Out-Of-Order] 34720→22 [P
   22 7.075944814   192.168.0.2   192.168.0.10     TCP       66 22→34720 [ACK] Seq=1 Ack=28158
   23 7.075957700   192.168.0.2   192.168.0.10     TCP       66 22→34720 [ACK] Seq=1 Ack=28158
   24 7.076084497   192.168.0.2   192.168.0.10     SSH      102 Server: Encrypted packet (len=
   25 7.076488809   192.168.0.10  192.168.0.2      TCP       66 34720→22 [ACK] Seq=2815852527 .
   33 8.182734254   192.168.0.2   239.255.255.250  SSDP     217 M-SEARCH * HTTP/1.1
   36 10.694115788  192.168.0.2   239.255.255.250  SSDP     217 M-SEARCH * HTTP/1.1

    Time to live: 1
    Protocol: UDP (17)
    Header checksum: 0xd314 [validation disabled]
    [Header checksum status: Unverified]
    Source: 192.168.0.2
    Destination: 239.255.255.250
    [Source GeoIP: Unknown]
    [Destination GeoIP: Unknown]
User Datagram Protocol, Src Port: 51230, Dst Port: 1900

0000  01 00 5e 7f ff fa a4 d1  8c c6 5e 74 08 00 45 00   ..^..... ..^t..E.
0010  00 cb 35 69 00 00 01 11  d3 14 c0 a8 00 02 ef ff   ..5i.... ........
0020  ff fa c8 1e 07 6c 00 b7  97 1b 4d 2d 53 45 41 52   .....l.. ..M-SEAR
0030  43 48 20 2a 20 48 54 54  50 2f 31 2e 31 0d 0a 48   CH * HTT P/1.1..H
0040  4f 53 54 3a 20 32 33 39  2e 32 35 35 2e 32 35 35   OST: 239 .255.255
0050  2e 32 35 30 3a 31 39 30  30 0d 0a 4d 41 4e 3a 20   .250:190 0..MAN:
0060  22 73 73 64 70 3a 64 69  73 63 6f 76 65 72 22 0d   "ssdp:di scover".
```

Figure 4: Chaffed screenshot

Here we can see a chaff packet is created after the SYN. Wireshark catches it as an out-of-order packet.

Fragroute is powerful and versatile. Creating and using a configuration with a combination of the options can lead to more obfuscated network traffic. Using tcp_chaff with other options can result in the crafting of packets that could be utilized in stepping-stone intrusion attacks.

IV. Conclusion

In order to minimize the risk to be detected and captured, more and more intruders tend to exploit stepping-stone to launch their attacks. Since there are many sophisticated stepping-stone intrusion detection approaches proposed after 2000, so many intruders evade the detection by manipulating TCP/IP sessions. In this paper, we first present how to make a code to sniff computer network traffic since most approaches need to intercept TCP/IP packets to detect stepping-stone intrusion. Unlike using packet sniffing tools, self-making code to sniff network packet can be easily integrated into different detecting approaches. We second introduce a tool Fragroute to chaff a connection. Session manipulation can help intruders escape from detection. So knowing how to manipulate a TCP/IP session can help researchers develop more advanced detection methods not only detecting stepping-stone intrusion, but also resisting intruders’ manipulation.

References

[1] [YZang2000]Y. Zhang, and V. Paxson, “Detecting Stepping-Stones,” Proc. of the 9th USENIX Security Symposium, Denver, CO, pp. 67-81, August 2000.

[2] S. Staniford-Chen, and L. T. Heberlein, “Holding Intruders Accountable on the Internet,” Proc. IEEE Symposium on Security and Privacy, Oakland, CA, pp. 39-49, 1995.

[3] K. Yoda, and H. Etoh, “Finding Connection Chain for Tracing Intruders,” Proc. 6th European Symposium on Research in Computer Security, Toulouse, France, Lecture Notes in Computer Science, vol. 1985, 2000, pp. 31-42.

[4] A. Blum, D. Song, and S. Venkataraman, “Detection of Interactive Stepping-Stones: Algorithms and Confidence Bounds,” Proc. of International Symposium on Recent Advance in Intrusion Detection, Sophia Antipolis, France, pp. 20-35, September 2004.

[5] T. He, L. Tong, “Detecting encrypted stepping-stone connections,” In: Proceedings of IEEE Transaction on signal processing, vol. 55, No. 5, 2007, pp. 1612-1623.

[6] J. Yang, and Y. Zhang, “RTT-based Random Walk Approach to Detect Stepping-Stone Intrusion,” Proc. of 29th IEEE International Conference on Advanced Information Networking and Applications, Gwangju, South Korea, pp.558-563, March 2015.

[7] K. H. Yung, “Detecting Long Connecting Chains of Interactive Terminal Sessions,” Proc. of International Symposium on Recent Advance in Intrusion Detection (RAID), Zurich, Switzerland, pp.1-16, October 2002.

[8] J. Yang, S.-H.S. Huang, M. D. Wan, "A Clustering-Partitioning Algorithm to Find TCP Packet Round-Trip Time for Intrusion Detection," Proceedings of 20th IEEE International Conference on Advanced Information Networking and Applications (AINA 2006), Vienna, Austria, Vol. 1, pp. 231-236, April 2006.

[9] Vic Hargrave, Develop a packet sniff with Libpcap, https://vichargrave.github.io/articles/2012-12/develop-a-packet-sniffer-with-libpcap, Accessed at March 4th, 2017.

[10] T. H. Ptacek, and T. N. Newsham, “Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection” , January 1998, SF298 Report “http://www.dtic.mil/dtic/tr/fulltext/u2/a391565.pdf”, accessed at Dec. 3rd, 2017.