

WHITE PAPER

IP Fragment Reassembly with Scapy

Mark Baggett

IP Fragment Reassembly with scapy

GIAC (GCIA) Gold Certification

! "\$%&'() *&+(, *- -.##/(0%1234415-6*7089%6(
! : ; 7<%&'(=79+(> *??.&(

! 99. @#. : 'A"? . (1B#\$ (2412(

(! C<#&*9#(

(
D; . &0*@@7?-(EF(G&* -6. ?#<(9*?(C. ("<. : (CH(*##*9+. &<(#%(\$7: . (#\$. 7&(? . G*&7%" <(7?#. ?#7%?<(G&%6(7?#&" <7%?(: . #. 9#7%?(<H<#. 6(*?: (*?*0H<#<8(((D@. &*#7?-(<H<#. 6<(-7; . (@&. G. &. ?9. (#%(%; . &0*@@7?-(G&* -6. ?#<(C*<. : ("@%?(. 7#\$. &(#\$. (@%<7#7%?(7?#\$. (@*9+. #(%&(#\$. (#76. (%G(*&&7; *08(((! <(*&. <"0#(G&* -6. ?#. : (@*9+. #<(67-\$#(C. (&. *<<. 6C0. : (7?(%?. (%G(G7; . (: 7GG. &. ?#(I *H<8(((EG(#\$. (EJK(%&(#\$. (*?*0H<#: %(?%#(&. *<<. 6C0. (#\$. (@*9+. #<#\$. (<*6. (I *H(*<#\$. (#*&- . #(\$%<#/((*?(*##*9+(6 *H(<"99. . : (*?: (-%("?: . #. 9#. : 8(((>\$70. (<%6. (7?#&" <7%?(: . #. 9#7%?(<H<#. 6<(\$*; . (#. 9\$?7L". <(G&(: . *07?-(I 7#\$(\$\$. <. (*##*9+<#\$. &. (*&. (; . &H(G. I (#%0<(*; *70*C0. (#%(\$\$. (*?*0H<#(0%0%+(7?<7: . (#\$. (&. *<<. 6C0H(@&%9. <<(*?: (#&H(#%("?: . &<#*?: (#\$. (*##*9+. &Mk(7?#. ?#8(N\$7<(@*@. &(I 700(. 0@0%&. (\$%I (*?(*?*0H<#(9*?("<. (<9*@H(#%(&. *<<. 6C0. (#\$. (G&* -6. ?#. : (*##*9+(@*9+. #<(7?(*<7670*&(6*??. &(#%(P7?"0/(>7?: %I </() *97?#%<\$/(Q7<9%(&"#. &<(*?: (%#\$. &(%@. &*#7?-(<H<#. 6<(#%<. . (\$%I (. *9\$(%@. &*#7?-(<H<#. 6(I %"0: (7?#. &@&. #(#\$. (G&* -6. ?#. : (@*9+. #<8(((

1. Introduction

1.1. The Problem

Overlapping IP fragments can be used by attackers to hide their nefarious intentions from intrusion detection system and analysts. Packets fragmentation will be performed by a router when the size of a packet exceeds the link layers MTU of the upstream network. (Fall, Stevens, 2011) The receiving host is responsible for reassembling those fragmented packets and passing it up the TCP stack to the proper application. The RFCs are silent on the matter of what the receiving host is supposed to do when the fragments it receives are retransmitted or overlap one another. No guidance is given as to whether or not the host should favor the first transmitted fragment it receives, the second transmitted fragment or the last. Similarly, should it favor overlapping fragments with the lowest offset or the highest? As a result different operating systems handle overlapping fragments in different ways. This problem is illustrated by the paper Active Mapping: Resisting NIDS Evasion Without Altering Traffic by Umesh Shankar and Vern Paxson (Shankar & Paxson, 2003) and then further explained in Large t Based Fragmentation Assembly by Judy Novak (Novak, 2005).

Imagine that we send the following 6 IP fragments that overlap in the following ways. For the sake of this discussion each fragment is 8 bytes in length which is the minimum size of a fragment. Also, to help keep things straight we will set the payload to be an eight ASCII 1s for packet 1, 2s for packet 2 and so on. Fragment 1 has an offset of zero and has a payload length of 24 bytes so that that fills fragment positions 0, 1 (offset 8) and 2 (offset 16). Fragment 2 begins at offset 24 and has a length of 16 bytes so that it fills fragment positions 4 and 5. Fragment 3 has an offset of 48, length of 24 bytes and fills fragment positions 6, 7 and 8. Fragment 4 has an offset of 8, a length of 32 and fills fragment positions 1 (offset 8), 2 (offset 16), 3 (offset 24) and 4 (offset 32) causing it to overlapping part of fragment positions 1 and 2. Fragment 5 has an offset of 48 and fills positions 6, 7 and 8 so that it perfectly overlaps fragment 3. Fragment 6 has an offset of 72 and fills fragment positions 9, 0xa and 0xb. Visually it would look like this:

(

Figure 1: 6 Fragmented Packets (Shankar & Paxson, 2003)(Novak, 2005)

Depending upon whether the reassembling host wants to favor the packets that arrive first or last or favor the packets with the lowest offset the fragments may end up in one of the 5 possible combinations. These combinations have been named First, Last, Linux, BSD and BSD-Right.

Reassembled using policy: First (Windows, SUN, MacOS, HPUX)

Reassembled using policy: Last/RFC791 (Cisco)

Reassembled using policy: Linux (Linux)

Reassembled using policy: BSD (AIX, FreeBSD, HPUX, VMS)

Reassembled using policy: BSD-Right (HP Jet Direct)

Figure 2: 5 Reassembly Methods (Shankar & Paxson, 2003)(Novak, 2005)

These inconsistencies allow attackers to put a malicious payload in an overlapped fragment. If the IDS and the host reassemble the packets differently the IDS will not see the packets, but the reassembling host will. Although many IDSs attempt to mitigate this risk by reassembling the packets in multiple ways, such as SNORT's frag3 preprocessor, the analyst is given very little insight to what happens inside the reassembly engine. This can lead to the analyst incorrectly dismissing an attack as an IDS false negative. Consider the following scenario. The attacker sends a crafted packet that contains both a Linux Exploit and a Windows Exploit to a vulnerable Windows target.

attacks.

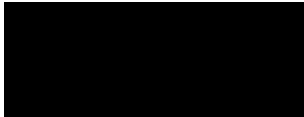


Figure 3: Views of the attacker, IDS and analyst

launched against their system?

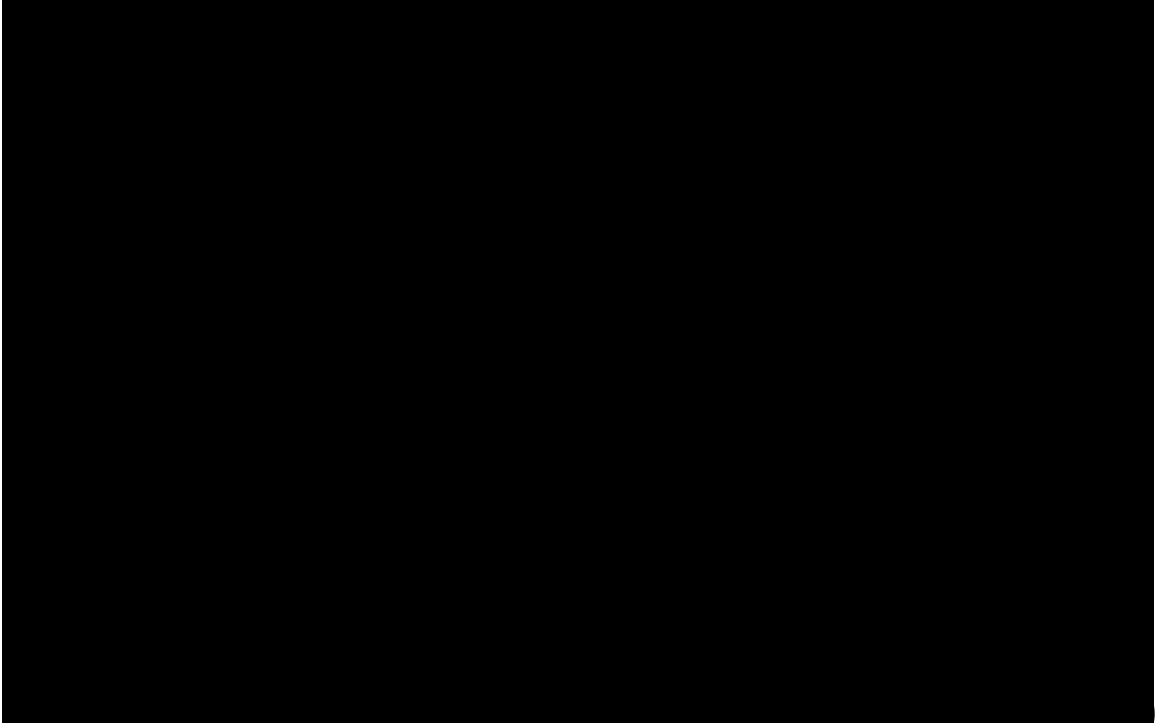


Figure 4: Wireshark uses BSD reassembly technique

1.2. One Possible Solution

Tools such as scapy and Python can be used to quickly reassemble packets in each of the differently combinations used by modern operating systems to get a better understanding of what the attacker may have intended to exploit. Over the next few pages we will examine how to recreate the reassembly engines as they are implemented by various operating systems. Then we as analysts, can use these techniques to peer behind the curtain and see how our reassembly engines would see that attackers packets. Understanding the techniques being used by the attacker will give us greater insight to the skill of our adversary and perhaps even help us identify attacks that our automated reassembly engines might overlook. (Shankar & Paxson, 2003)(Novak, 2005)

2. Writing a fragment reassembly engine

Writing the IP fragmentation engines in our TCP stack is no easy task. Fortunately for us, we do not have to deal with many of the difficulties the authors of those programs do. We don't have to worry about reassembly time-out, TTLs, memory management and other issues associated with the live transmission of data. We are

(

reading our packets from stored packet captures. This gives us another advantage over reassembling live packets in that we have all of the packets in our possession and can reorder them as needed before processing them. To reassemble the packets we will allocate a buffer in memory and then write each fragment to the buffer based allowing fragments to overwrite existing data. Using this method the last fragments that we process will overwrite any existing data in the buffer. To reassemble packets as each of the different reassembly policies we will just have to reorder our packet before we process them. For example, to process packets according to the `LAST/RFC791` policy we would just process packets from the first one we received until the last in chronological order. Since subsequent overlapping fragments will overwrite the previous packets we are favoring the LAST packet to arrive. To process packets according to the `FIRST` policy we process the same packets in reverse order filling the buffer with the last packet to arrive, then the 2nd to last etc. The first packet to arrive will overwrite any data that was written in the buffer by later packets thus favoring the FIRST packets.

2.1. Python and scapy data structures

Python's `StringIO` module provides us with a good data structure to use as our buffer for the reassembled fragments. We can use `StringIO`'s `seek()` method to set the location in the buffer to the fragments offset. Then we use the `write()` method to put our data in the buffer. After we have processed all the fragments we can use the `getvalue()` method to retrieve the contents of our completed fragment payload.

Scapy allows you to quickly and easily tear apart packets and get to the fields you are interested in. By following a variable containing a packet with `[protocol]` and `[field]` you can pull the contents of various fields from each packet. For example, to examine the IP ID field of a given packet we would simply address the variable containing the packet as `variablename[IP].id`. This tells scapy you want the value assigned to the `id` field in the `IP` layer of the packet. The field we are interested in is the `[IP].frag` which contains the fragment offset of the current fragment and the payload of each of the fragmented packets. The fragment offset will be the number of bytes into fragment chain that the payload bytes should be written. (Kozierok, 2005) The scapy

(
frag field is a fragment position not the byte offset. To get the byte offset you need to multiplying that number by 8 (8 bytes in the smallest fragment). Then using the StringIO.seek() method we place the pointer into the buffer at the location where the payload should be written. Using a FOR loop to step through each packet we have a simple reassembly engine.

2.2. The Last/RFC791 Policy

Let's look at the simplest reassembly policies Last/RFC791. This reassembly policy gives preference to fragment that appear later in a packet capture. Assume we have a list of all of the fragments that need to be reassembled. By processing the list of packets from the first to the last allowing the later to overwrite the earlier we follow the Last policy. When combined with scapy's ability to easily parse packets and extract fields like the fragment offset and payload we can write a very basic packet reassembly engine in just a few lines of Python code. The following code will take a list of fragments and assemble the payload according to the Last/RFC791 policy.

```
def rfc791(listoffragments):
    buffer=StringIO.StringIO()
    for pkt in listoffragments:
        buffer.seek(pkt[IP].frag*8)
        buffer.write(pkt[IP].payload)
    return buffer.getvalue()
```

(
Let's look at this code line by line. The first line uses the keyword def to define a new function called rfc791 which will be passed a single parameter. The parameter will be stored in a variable called listoffragments. As you might guess from the name the parameter will be a Python data structure called a list, and it will contain all of the fragments in a given fragment train. Notice that after the first line we begin indenting the code by 4 spaces. The indentation is very important to Python. It tells Python that each of those indented lines is part of the code block that makes up the rfc791 function we are defining. The second line will create a variable in memory called buffer which is of type StringIO. The variable buffer will be used to store all of the pieces of the fragment train. Next we start a for loop to step through each individual fragment inside of the fragment train. The for loop is followed by another group of indented lines. Again, the indentation is used to group lines of code into a code

(
 block. The two lines that follow the for loop will be executed repeatedly as part of the for loop for each individual fragment in our list `listoffragments`. The first time through the loop the variable `pkt` will contain the first fragmented packet in `listoffragments`. The second time through the loop it will contain the second fragmented packet in `listoffragments`. This will repeat for every packet in `listoffragments`. So, for every fragment in `listoffragments` we will execute these next two lines. The first one, `buffer.seek(pkt[IP].frag*8)` sets the pointer that will be used to write data in the buffer to the value that is contained in the scapy fragment position field of the current packet multiplied by eight. To convert a scapy fragment position number we multiply by 8 because each of these fragments will contain 8 bytes (64 bits). (Kozierok, 2005) Now that the pointer is set, the next line will write the payload of the fragment into the buffer at the location that was just set by the seek method. Once we have done that for all of the fragments we simply retrieve the contents of the buffer with the `getvalue()` method and return that from our function. (Python Software Foundation, 2012)

2.3. The FIRST Policy

To write the FIRST reassembly engine we can follow the exact same process we followed to favor the LAST packet, but process our packets in reverse order. In doing so the first shall be last and our packets will be assembled properly. Python lists make it very easy to process a list in reverse order. By simply adding `-1` to the end of our list of fragments we reverse the list. (Lutz, 2012) Now writing our FIRST reassembly engine is almost identical to rfc791.

```
def first(listoffragments):
    buffer=StringIO.StringIO()
    for pkt in listoffragments[::-1]:
        buffer.seek(pkt[IP].frag*8)
        buffer.write(pkt[IP].payload)
    return buffer.getvalue()
```

2.4. The MSD-Right Policy

(
 Our remaining 3 reassembly policies look at more than just the chronological order the fragments arrived in. They also take the fragment offset into consideration

(
 when deciding which fragment takes precedent. We need to reorder the packets so we process them based upon both the time they arrived and their offset according to the different reassembly engines. For the BSD-Right policy we need to process fragments in order by their fragment offset from lowest to highest. If two packets have the same offset then we allow the last one to arrive chronologically to overwrite the existing data. Since our fragments are already in chronological order, sorting the packets based on their fragment offset will line the packets up for the BSD policy. We can use the `sorted()` function to put the fragments in to order by fragment offset then by chronological order. We pass the `sorted` function two parameters. We will pass it the list we want to sort and a `key` function to `sorted()` and it returns a list that is sorted based on the key. In this case our key function is `lambda x:x[IP].frag` which tells `sorted()` to put them in fragment offset order.

```
def bsdright(listoffragments):
    buffer=StringIO.StringIO()
    for pkt in sorted(listoffragments, key=lambda x:x[IP].frag):
        buffer.seek(pkt[IP].frag*8)
        buffer.write(pkt[IP].payload)
    return buffer.getvalue()
```

2.5. The BSD policy

(
 BSD is simply BSD-Right in reverse. Processing the BSD-Right sorted fragments from last to first will cause the early fragments to overwrite the latter ones. We can take the same approach we used with FIRST and process the packets backwards by adding a `[-1]` to the end of our list of fragments. Because we want to process them in reverse order after they have been sorted, we add the `[::-1]` to the end of the `sorted` function. Now we will processing the packet the same way we did for BSD-Right but in reverse.

```
def bsd(listoffragments):
    buffer=StringIO.StringIO()
    for pkt in sorted(listoffragments, key=lambda \
x:x[IP].frag)[::-1]:
        buffer.seek(pkt[IP].frag*8)
        buffer.write(pkt[IP].payload)
    return buffer.getvalue()
```

(

(

2.6. The Linux policy

The Linux Policy also takes the fragment offset into consideration. It favors whatever packet has the lowest offset. By processing the packets in reverse fragment offset order we allow the lowest fragments to overwrite the highest. So we need to sort our packets with the highest fragment offset appearing first in the list. To reverse a sort is as simple as passing the parameter `reverse=True` to our sorted command. By applying a reverse sort to our fragments before processing them first to last we get a Linux reassembly policy.

```
def linux(listoffragments):
    buffer=StringIO.StringIO()
    for pkt in sorted(listoffragments, key= lambda x:x[IP].frag,\
reverse=True):
        buffer.seek(pkt[IP].frag*8)
        buffer.write(pkt[IP].payload)
    return buffer.getvalue()
```

2.7. Testing the code

To test the code we can reassemble various fragmented packets samples from internet. We can craft our own fragmented packets using tools such as fragroute and fragrouter. We can also craft our own packets using scapy. The following section of code will generate the six packet fragments outlined in the introduction with the offsets specified in the Shankar/Paxson and Novak papers.

```
def genfragments():
    pkts=scapy.plist.PacketList()
    pkts.append(IP(flags="MF",frag=0)/("1"*24))
    pkts.append(IP(flags="MF",frag=4)/("2"*16))
    pkts.append(IP(flags="MF",frag=6)/("3"*24))
    pkts.append(IP(flags="MF",frag=1)/("4"*32))
    pkts.append(IP(flags="MF",frag=6)/("5"*24))
    pkts.append(IP(frag=9)/("6"*24))
    return pkts
```

(

Now we can pass that fragment train off to each of our functions to see how it puts humpty back together. Passing the results of `genfragments()` to the `first()` function will generate our fragment test pattern, then reassemble the packets using the

(

first policy. By calling each of the reassembly engines we can see if our results match those outlines in paper.

```
print "Reassembled using policy: First"
print first(genfragments())
print "Reassembled using policy: Last/RFC791"
print rfc791(genfragments())
print "Reassembled using policy: Linux"
print linux(genfragments())
print "Reassembled using policy: BSD"
print bsd(genfragments())
print "Reassembled using policy: BSD-Right"
print bsdright(genfragments())
```

(

Running our script we would get the following result and we can see that indeed our reassembled packets do match what is expected from each of the reassembly engines.

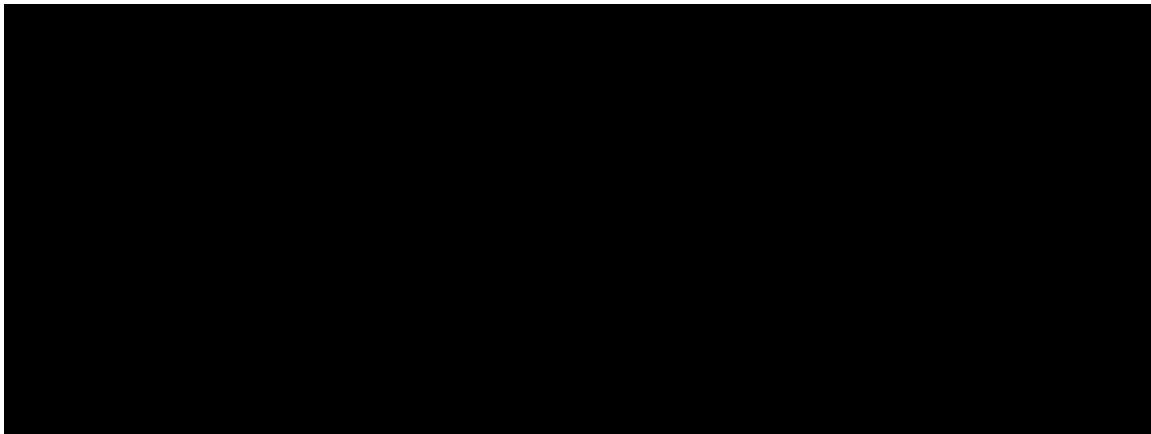


Figure 5: Output of running a simple fragment generator and reassembly engine

2.8. Extending the code

With the basic reassembly engines completed we can turn our attention to making the code user friendly and useful. One possible use would be to place each of the reassembly routines into a modularize script so you can `import` it into your existing scapy sessions to reassemble payloads as needed. Another application of this code would be to add functionality that extracts fragmented packets from pcap files then reassembles them using each of the 5 different policies so the analyst can see how the packets would be interpreted by different operating systems. The extended application could also allow the analyst to write reconstructed packets to disk. Although the review of that source code is beyond the scope of this paper, I have produced that tool for you and provided it in the appendix of this paper. Let's look at how we can use that

(application to analyze fragmented packets generated by fragroute and scapy. The extended version of this script will read the packets from disk, find all fragmented packets, prompt the user and ask if they want to process a given fragment train and display the reassembled packet on the screen or write the payload to disk. As this script is updated the latest source code will be available for download at the address <http://baggett-scripts.googlecode.com/svn/trunk/reassembler/>. The version that was used for testing in this paper is included in the Appendix. First let's take a look at the options that are available to the program. We can see the options by passing `--help` as an option to the script.

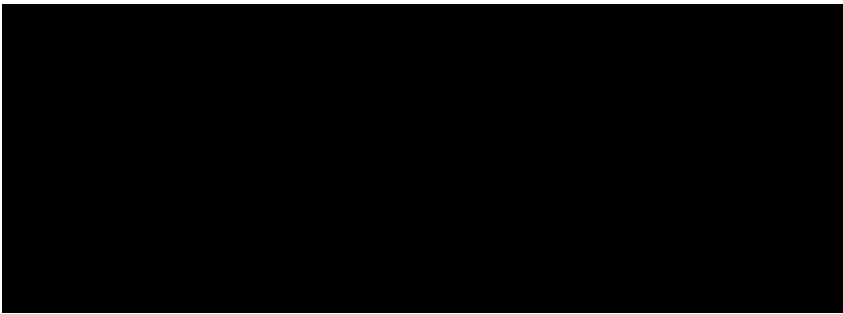


Figure 6 : Help for reassembler.py

Here you can see we can pass it `-r` to read a pcap from disk and process it. We can use `-w` if we want it to write the payloads to disk instead of printing to the screen. Finally, `-p` is specified if we want to specify the filename prefix to use when using the `-w` option to create payload files on disk. There is also the `-d` option which will generate a fragmented packet stream then decode it. The `-i` option is used to quickly gain an understanding of what each of the fragmentation engines does. If we really want to test the application we need some fragmented packets to test with. For that we can use a tool like fragroute or fragrouter to generate our packets.

Although several fragmentation combinations were tested, here is one example of how we can create fragmented packets to test the software. First we create a fragroute configuration file that tells fragroute how to break our packets down. Here is an example of a fragroute configuration file.

(

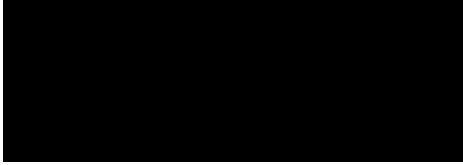


Figure 7: Fragroute Configuration File

This configuration file tells fragroute to break the data into 8 byte fragments then insert duplicate packets with random payloads. Then transmit the fragments in random order. Note that the `chaff_dup` creates duplicate overlapping fragments. There are no partially overlapping fragments. The fragments perfectly overlap other fragments. All of the engines will reassemble the payloads as they to packets 3 and 5 from our overlapped packet test pattern. Therefore, we expect the `IRST` and `BSD` patterns to reassemble packet one way and the other engines to assemble them the other way.

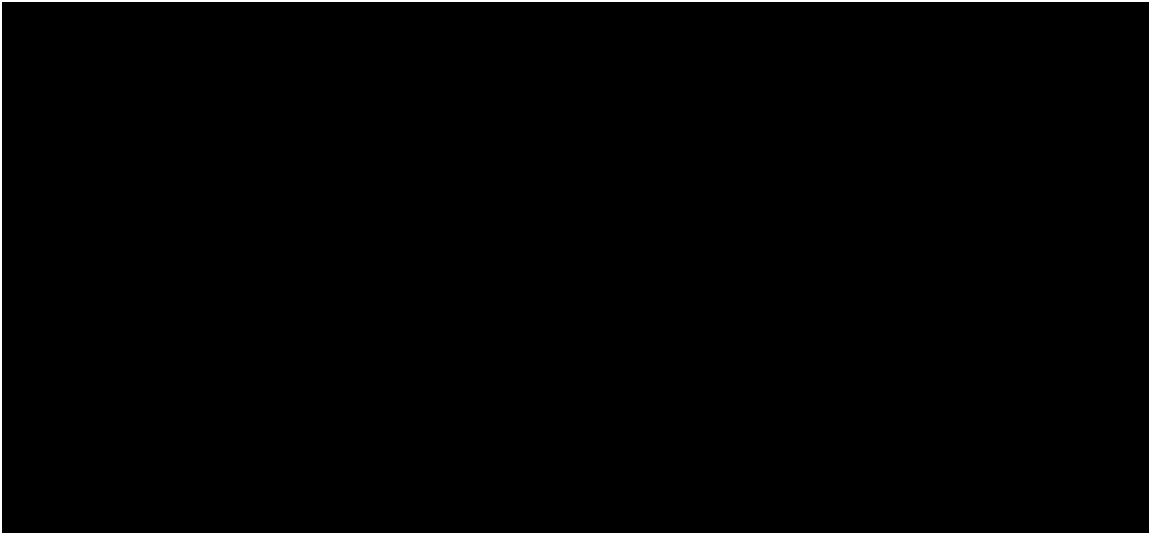


Figure 8: Results of assembling fragroute generated packets

As expected First and BSD reassemble the packets one way and the other engines see a totally different payload. In this case First and BSD see the real payload and the other engines all see random garbage that was created by fragroute. Now, let's test to see if our code that writes the payloads to disk with an optional prefix works properly.

(

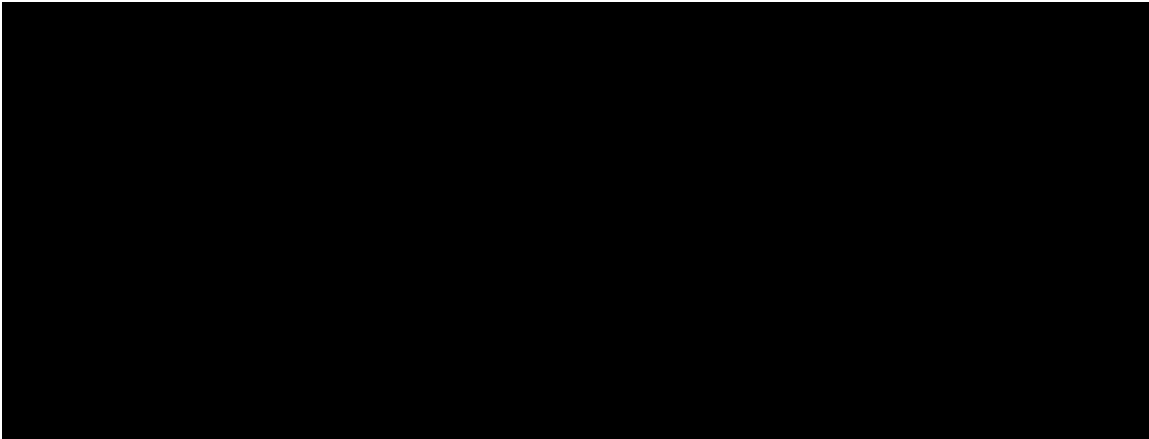


Figure 9: Writing reassembled payloads to disk

It works!! This should be suitable for extracting binary payloads such as shell code and other exploits which may not behave well when displayed to the screen. Now, we as analysts can get the same insight to fragmented packets that our IDS engine may already have and make better informed decisions about the threat posed by a given attack

3. Conclusions

The use of overlapping IP fragments for IDS evasion has been around since the 20th century. While some IDSs reduce the risk of false negatives through various reassembly mechanisms, the IDS Analyst is often blind to these attacks and left to trust the technology is not overlooking the threat. However, with tools such as Python, scapy and a little elbow grease the analyst can see exactly what malicious activities are being launched against their organization. By making use of these tools and techniques analysts are less likely to incorrectly dismiss an IDS generated true positive as a false positive.

(

4. References

Shankar, U., & Paxson, V. (2003). Active mapping: Resisting nids evasion without altering traffic. Retrieved April 29, 2012 from <http://www.icir.org/vern/papers/activemap-oak03.pdf>

Novak, J. (2005, April). Target-based fragmentation reassembly. Retrieved April 29, 2012 from http://www.snort.org/assets/165/target_based_frag.pdf

Fall, K., & Stevens, W. R. (2011). TCP/IP illustrated . (2nd ed., Vol. 1, p. 148). Ann Arbor, Michigan: Pearson Education Inc.

Lutz, M. (2012). Python pocket reference. (4th ed., p. 16). North Sebastopol, CA: O'Reilly Media Inc.

Seitz, J. (2009). Grey Hat Python. San Francisco, CA: No Starch Press.

Python Software Foundation (2012). Python Online Documentation. Retrieved from <http://www.python.org/doc/>

Kozierok, C. (2005). The TCP Guide. (p. 374). San Francisco, CA: No Starch Press.

(

```

print "Reassembled using policy: Last/RFC791 (Cisco)"
print rfc791(genjudyfrags())

print "Reassembled using policy: Linux (Umm.. Linux)"
print linux(genjudyfrags())

print "Reassembled using policy: BSD (AIX, FreeBSD, HPUX, VMS)"
print bsd(genjudyfrags())

print "Reassembled using policy: BSD-Right (HP Jet Direct)"
print bsdright(genjudyfrags())

```

PROGRAM 2 Extending the code to reassemble fragments from disk

```

from scapy.all import *
import StringIO
from optparse import OptionParser
import os
import sys

def rfc791(fragmentsin):
    buffer=StringIO.StringIO()
    for pkt in fragmentsin:
        buffer.seek(pkt[IP].frag*8)
        buffer.write(pkt[IP].payload)
    return buffer.getvalue()

def first(fragmentsin):
    buffer=StringIO.StringIO()
    for pkt in fragmentsin[::-1]:
        buffer.seek(pkt[IP].frag*8)
        buffer.write(pkt[IP].payload)
    return buffer.getvalue()

def bsdright(fragmentsin):
    buffer=StringIO.StringIO()
    for pkt in sorted(fragmentsin, key= lambda x:x[IP].frag):
        buffer.seek(pkt[IP].frag*8)
        buffer.write(pkt[IP].payload)
    return buffer.getvalue()

def bsd(fragmentsin):
    buffer=StringIO.StringIO()
    for pkt in sorted(fragmentsin, key=lambda x:x[IP].frag)[::-1]:
        buffer.seek(pkt[IP].frag*8)
        buffer.write(pkt[IP].payload)
    return buffer.getvalue()

def linux(fragmentsin):
    buffer=StringIO.StringIO()
    for pkt in sorted(fragmentsin, key= lambda x:x[IP].frag, reverse=True):
        buffer.seek(pkt[IP].frag*8)
        buffer.write(pkt[IP].payload)
    return buffer.getvalue()

def genjudyfrags():
    pkts=scapy.plist.PacketList()
    pkts.append(IP(flags="MF",frag=0)/("1"*24))
    pkts.append(IP(flags="MF",frag=4)/("2"*16))
    pkts.append(IP(flags="MF",frag=6)/("3"*24))
    pkts.append(IP(flags="MF",frag=1)/("4"*32))
    pkts.append(IP(flags="MF",frag=6)/("5"*24))
    pkts.append(IP(frag=9)/("6"*24))
    return pkts

```

```
(
def processfrags(fragmenttrain):
    print "Reassembled using policy: First (Windows, SUN, MacOS, HPUX)"
    print first(fragmenttrain)
    print "Reassembled using policy: Last/RFC791 (Cisco)"
    print rfc791(fragmenttrain)
    print "Reassembled using policy: Linux (Umm.. Linux)"
    print linux(fragmenttrain)
    print "Reassembled using policy: BSD (AIX, FreeBSD, HPUX, VMS)"
    print bsd(fragmenttrain)
    print "Reassembled using policy: BSD-Right (HP Jet Direct)"
    print bsdright(fragmenttrain)

def writefrags(fragmenttrain):
    fileobj=open(options.prefix+"-first","w")
    fileobj.write(first(fragmenttrain))
    fileobj.close()
    fileobj=open(options.prefix+"-rfc791","w")
    fileobj.write(rfc791(fragmenttrain))
    fileobj.close()
    fileobj=open(options.prefix+"-bsd","w")
    fileobj.write(bsd(fragmenttrain))
    fileobj.close()
    fileobj=open(options.prefix+"-bsdright","w")
    fileobj.write(bsdright(fragmenttrain))
    fileobj.close()
    fileobj=open(options.prefix+"-linux","w")
    fileobj.write(linux(fragmenttrain))
    fileobj.close()

def main():
    parser=OptionParser(usage='%prog [OPTIONS]')
    parser.add_option('-r','--read',default="",help='Read the specified packet
capture',dest='pcap')
    parser.add_option('-d','--demo',action='store_true', help='Generate classic fragment
test patter and reassemble it.')
    parser.add_option('-w','--write',action='store_true', help='Write 5 files to disk
with the payloads.')
    parser.add_option('-p','--prefix',default='reassembled', help='Specify the prefix for
file names')

    if (len(sys.argv)==1):
        parser.print_help()
        sys.exit()

    global options, args
    (options,args)=parser.parse_args()

    if options.demo:
        processfrags(genjudyfrags())

    if not os.path.exists(options.pcap):
        print "Packet capture file not found."
        sys.exit(2)

    packets=rdpcap(options.pcap)
    filtered=[a for a in packets if a[IP].flags==1 or a[IP].frag > 0]

    if len(filtered)==0:
        print "No fragments in packet capture file."
        sys.exit(2)

    unipids={}
    for a in filtered:
        unipids[a[IP].id]='we are here'

    for ipid in unipids.keys():
        print "Packet fragments found. Collecting fragments now."
        fragmenttrain = [ a for a in filtered if a[IP].id == ipid ]
        processit = raw_input("Reassemble packets between hosts "+str(a[0][IP].src)+" and
"+str(a[0][IP].dst)+"? [Y/N]")
```

```
(  
    if str(processit).lower()=="y":  
        if options.write:  
            writefrags(fragmenttrain)  
        else:  
            processfrags(fragmenttrain)  
  
if __name__ == '__main__':  
    main()
```