Target-Based Fragmentation Reassembly

Author
Judy Novak

Revision 3.0
September 21, 2005

Prepared by:
Sourcefire, Incorporated
9770 Patuxent Woods Drive
Columbia, MD 21046

Introduction

In their landmark 1998 paper, "Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection,"[1] Thomas Ptacek and Timothy Newsham exposed some weaknesses in intrusion detection systems (IDS). The authors revealed that intrusion detection systems cannot be effective and accurate because they do not necessarily process, or perhaps, even *observe* network traffic exactly as the destination host that receives the message does. This flaw exists in several layers of evaluation and processing of the packets including the IP, protocol, and application layers. As an example of the problem, consider traffic that has overlapping fragments that are sent to a given host. Because different operating systems have unique methods of fragment reassembly, if an intrusion detection system uses a single "one size fits all" reassembly method, it may not reassemble and process the packets the same way the destination host does. An attack that successfully exploits these differences in fragment reassembly can cause the IDS to miss the malicious traffic and fail to alert.

Remarkably, seven years later, many of these problems still exist. One answer to the fragment reassembly dilemma is a savvy IDS that is aware of the operating system and applications on the hosts that the IDS protects. The term "target-based" has been coined to identify an intelligent IDS that is informed about hosts residing on the network and is capable of analyzing traffic sent to those hosts as the host itself analyzes the traffic. This does not solve all of the problems discussed by Ptacek and Newsham, but it certainly improves the accuracy of the IDS. This can eliminate false positives about irrelevant alerts such as a Windows exploit bound for a Unix host. As well, deliberately mangled packets do not dupe the IDS, since it processes those packets exactly as the receiving host does.

The open source IDS Snort has begun to implement target-based analysis with the frag3 preprocessor. Frag3 is able to reassemble overlapping fragments using the same policy as the destination host. A user configures the IDS to apply specific fragmentation reassembly policies for individual hosts or networks. Then, when the Snort sees overlapping fragments bound for any of these hosts, it knows the appropriate reassembly policy to apply—allowing both Snort and the destination host to reassemble the fragments identically. This successfully precludes evasion attacks that use overlapping fragments.

This paper discusses fragmentation attacks, the fragment reassembly policies identified by Vern Paxson and Umesh Shankar, how Snort frag3 can be used, and finally, how to implement two programs—one to assess the fragmentation reassembly policy used by a remote host and another to test whether or not an IDS can be evaded by an attack that employs overlapping fragments.

A Simple Fragmentation Attack

Suppose that an IDS has a signature that alerts on an attempted buffer overflow attack that occurs when an attacker supplies an overly long username value for FTP authentication. For example, Snort has the following rule to alert on such an attack:

```
alert tcp $EXTERNAL_NET any -> $HOME_NET 21 (msg:"FTP USER overflow attempt";
flow:to_server,established,no_stream; content:"USER"; nocase;
isdataat:100,relative; pcre:"/^USER\s[^\n]{100}/smi";)
```

This rule looks for traffic that originates outside of and bound for the protected network to the FTP command port. The alert fires when an established stream to port 21 begins with the content of "USER" with no linebreak in the following 100 bytes.

 ©Sourcefire, Inc. All rights reserved. January 21, 2005

Further, suppose an attacker wants to evade the IDS by sending wholly overlapping fragments that differ in content by a single character. The original fragment contains a content of "USER"; the overlapping fragment contains the content of "XSER". TCP/IP implementations elect to regard either the original or subsequent fragment as the valid one and discard the other. Windows hosts use a policy of accepting the first fragment and ignoring all subsequent overlapping ones. An IDS that considers the subsequent fragment the valid one will miss an attack that targets a Windows host because it sees the content of "XSER" instead of USER.

This is a simple and tidy illustration of overlapping fragments; thorough analysis must consider many more complications such as partially overlapping fragments and the location of the overlap of the fragments. Vern Paxson and Umesh Shankar conducted extensive fragmentation research involving overlapping fragments and determined that there are several different fragmentation reassembly policies used by different operating systems.

Paxson/Shankar Model

In the paper titled "Active Mapping: Resisting NIDS Evasion Without Altering Traffic,"[2] the authors, Vern Paxson and Umesh Shankar, include a section that discusses fragmentation techniques as a means of evading intrusion detection systems. They discovered that there are five different fragment reassembly methods in use by modern operating systems. They also developed a paradigm of overlapping fragments that tests all methods of reassembly and can be used to determine which operating systems use which reassembly techniques.

The model used in the Paxson/Shankar paper consists of a series of six fragments of varying offsets, content, and length. This particular model provides each of the following types of fragments:

- At least one fragment that is wholly overlapped by a subsequent fragment with an identical offset and length
- At least one fragment that is partially overlapped by a subsequent fragment with an offset greater than the original
- At least one fragment this is partially overlapped by a subsequent fragment with an offset less than the original

The following diagram depicts a series of IP fragments, labeled 1 through 6, that illustrate the Paxon/Shankar model. A fragment's number indicates its arrival order.

©Sourcefire, Inc. All rights reserved. January 21, 2005

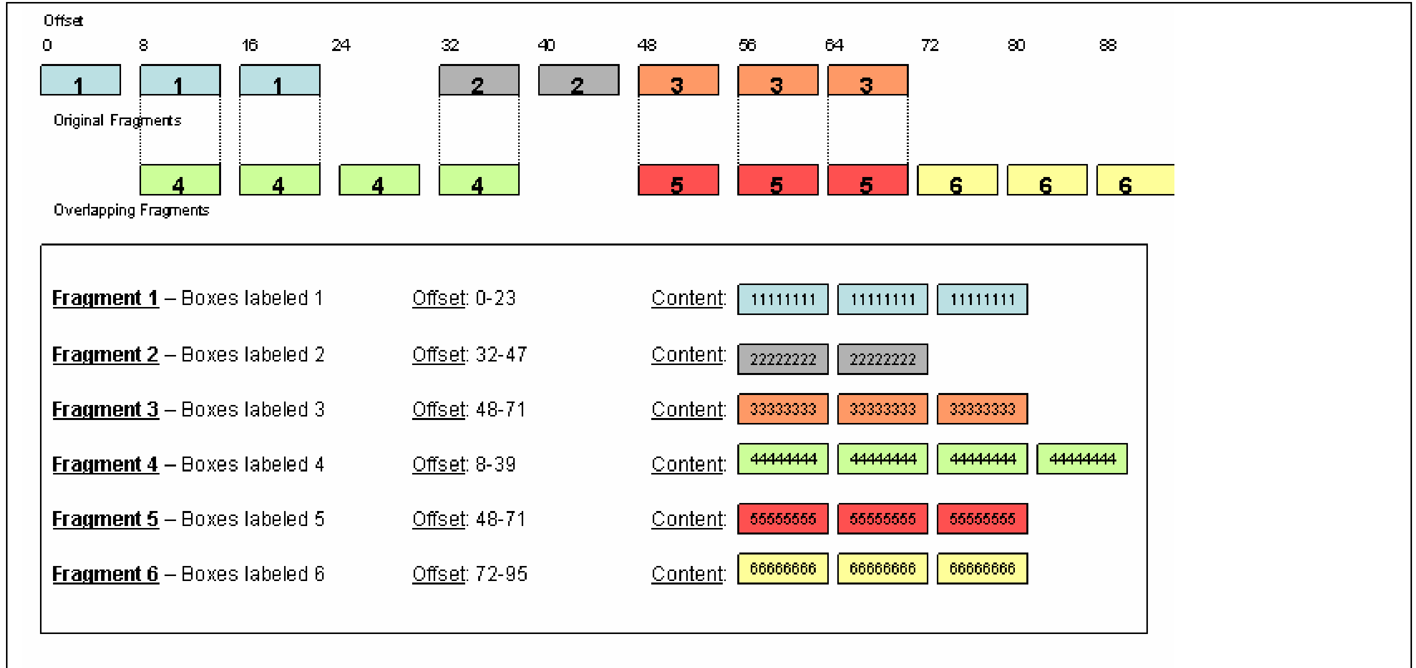

Figure 1 – Paxson/Shankar model

The fragment model used is at the top of the diagram; the legend below it explains the model. The offsets of each fragment are shown with small numbers above the model. Each fragment arrives in an IP datagram that contains an offset value in the IP header that represents the byte displacement of the fragment. The receiving host uses this value to properly reorder the fragments during reassembly. This offset is relative to the protocol header that precedes it. This discussion of the theory disregards the protocol header type and length, although they are a concern for implementation purposes. There are two depicted rows of fragments. The first row represents fragments that arrived first and are hereafter called original fragments. The second row represents fragments that arrived after the original fragments and are called subsequent fragments.

There are six different fragments numbered 1 through 6 in the diagram. Each colored box represents an 8-byte chunk of one of these six fragments. For instance, the first box with a label of 1 represents eight bytes of data starting at offset 0 from the end of the protocol header. The next two boxes with a label of 1 are associated with fragment 1 and also each represents 8 bytes of data. Therefore, fragment 1 consists of three 8-byte chunks of data beginning at offset 0 for 24 bytes. The content of each of these 8-byte chunks is 11111111. The other five fragments have similar content; for instance, fragment 2 has a content of 22222222. The fragments may have different offsets and lengths.

As explained before, fragments may wholly or partially overlap. In the model, fragment 5 wholly overlaps fragment 3. They both start at offset 48 and are 24 bytes long. They differ only in content, which is used to determine which fragment is used in the reassembly method for a particular operating system.

The model also requires subsequent fragments that overlap original fragments with offsets less than and greater than the original. This can be fulfilled using the same fragment, that is, a subsequent fragment with both an offset greater than one original fragment and an offset less than another original fragment. Fragment 4 (which has an offset of 8) satisfies the condition of both overlapping an original fragment with an offset greater than an original fragment (fragment 1, which has an offset of 0) and overlapping an original fragment with an offset less than the original fragment (fragment 2, which has an offset of 32).

 ©Sourcefire, Inc. All rights reserved. January 21, 2005

Each fragment differs in content so that when a receiving host uses unique reassembly methods, observers can discern which fragment is favored. The fragmentation policies employed by the Paxon/Shankar model favor individual fragment chunks based on two factors – offset and chronology. The offset pertains to the entire fragment offset; although a particular 8-byte chunk in a fragment may have its own offset, it is associated with the offset at the beginning of the entire fragment. Chronology denotes whether the fragment chunk is the original or subsequent at a given offset.

If you test the Paxson/Shankar model, you will observe five different reassembly policies:

- **BSD** favors an original fragment with an offset that is less than or equal to a subsequent fragment.
- **BSD-right** favors a subsequent fragment when the original fragment has an offset that is less than or equal to the subsequent one.
- **Linux** favors an original fragment with an offset that is less than a subsequent fragment.
- **First** favors the original fragment with a given offset.
- **Last** favors the subsequent fragment with a given offset.

The following figures show the fragments sent (for comparison purposes) and the reassembly of the fragments based on policy.

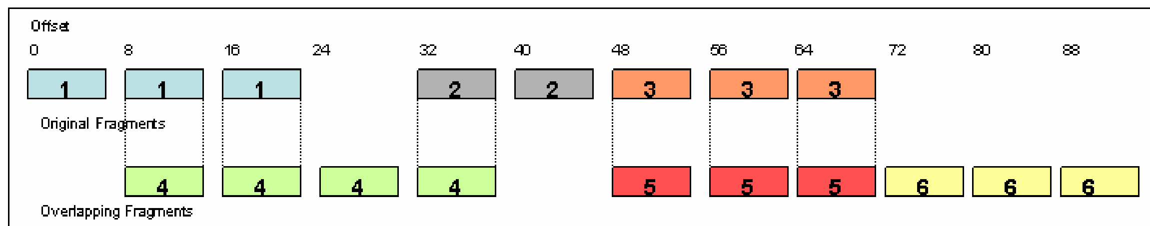

Figure 2 – Paxson/Shankar model

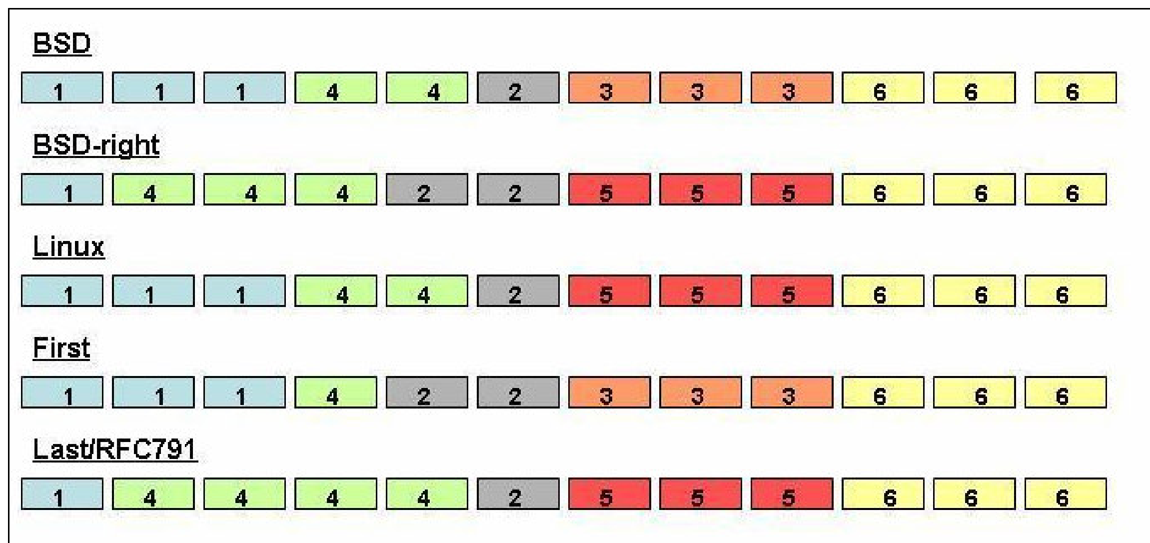

Figure 3 – Paxson/Shankar reassembly policies

Let's reexamine each policy to see how reassembly is performed.

©Sourcefire, Inc. All rights reserved. January 21, 2005

Policy	When…	Overlaps…	This fragment is favored…	Because…
BSD	subsequent fragment 4	original fragment 1	original fragment 1	fragment 1's offset of 0 is less than fragment 4's offset of 8
		original fragment 2	subsequent fragment 4	fragment 4's offset of 8 is less than fragment 2's offset of 32
	subsequent fragment 5	original fragment 3	original fragment 3	fragment 3's offset of 48 is equal to fragment 5's offset
BSD - Right	subsequent fragment 4	original fragment 1	subsequent fragment 4	fragment 1's offset of 0 is less than fragment 4's offset of 8
		original fragment 2	original fragment 2	fragment 2's offset of 32 is greater than fragment 4's offset of 8
	subsequent fragment 5	original fragment 3	subsequent fragment 5	fragment 5's offset of 48 is equal to fragment 3's offset
Linux	subsequent fragment 4	original fragment 1	original fragment 4	fragment 1's offset of 0 is less than fragment 4's offset of 8
		original fragment 2	subsequent fragment 4	fragment 4's offset of 8 is less than fragment 2's offset of 32
	subsequent fragment 5	original fragment 3	subsequent fragment 5	fragment 5's offset of 48 is equal to fragment 3's offset
First	subsequent fragment 4	original fragment 1	original fragment 1	fragment 1 is the original fragment
		original fragment 2	original fragment 2	fragment 2 is the original fragment
	subsequent fragment 5	original fragment 3	original fragment 3	fragment 3 is the original fragment
Last	subsequent fragment 4	original fragment 1	subsequent fragment 4	fragment 4 is the subsequent fragment
		original fragment 2	subsequent fragment 4	fragment 4 is the subsequent fragment
	subsequent fragment 5	original fragment 3	subsequent fragment 5	fragment 5 is the subsequent fragment

Paxson and Shankar implemented their model by placing overlapping fragments in the payload of an ICMP echo request and evaluating the payload of the ICMP echo reply. They discovered the unique reassembly policies used by different operating systems. Their implementation required the use of a different ICMP echo request per policy because ICMP echo requests have an ICMP checksum that is computed by using values found in the ICMP header and payload. The checksum is different for the reassembled payload because different fragments or partial fragments are used in the reassembly.

©Sourcefire, Inc. All rights reserved. January 21, 2005

One way to circumvent requiring five different fragmented echo requests, each with an ICMP checksum corresponding to a different policy, is to have fragment content that has the same ICMP checksum regardless of the policy used for reassembly. The checksum is computed by taking the one's complement of all 16-bit values and summing those values. This is a commutative operation where 16-bit fields can be swapped and the checksum remains the same.

```
x = 16-bit data field (header or payload)
checksum = ∑~x
```

An example of a fragment payload that can be used to yield an unchanging ICMP checksum is discussed later in the section "Implementation of the Paxson/Shankar Model."

Snort frag3 Preprocessor to the Rescue

The creator of Snort, Marty Roesch, has written an improved fragmentation preprocessor, frag3, that implements target-based fragmentation policies by allowing a user to identify the fragmentation reassembly method that is applied to a particular destination IP address or subnet. Assume that you have an entire subnet (10.4.10.x, for example) that consists only of Microsoft Windows hosts and you want to configure Snort to use the appropriate reassembly policy for these hosts. You need only enable the frag3 preprocessor in the configuration file and designate the appropriate fragmentation policy in an frag3 engine statement as follows:

```
preprocessor frag3_global:
preprocessor frag3_engine:  policy first, bind_to 10.4.10.0/24
```

This is fairly straightforward; the only unknown is the association of a fragmentation policy, in this case "first," with an operating system. The following table lists the fragmentation policies and associated platforms that Paxson and Shankar discovered.

Fragmentation Policy	Platforms
BSD-right	HP JetDirect
BSD	AIX 2, 4.3, 8.9.3, FreeBSD, HP-UX B.10.20, IRIX 4.0.5F, 6.2, 6.3, 6.4, NCD Thin Clients, OpenBSD, OpenVMS, OS/2, OSF1, SunOS 4.1.4, Tru64 Unix V5.0A,V5.1, Vax/VMS
Linux	Linux 2.x
First	HP-UX 11.00, MacOS (version unknown), SunOS 5.5.1,5.6,5.7,5.8, Windows (95/98/NT4/ME/W2K/XP/2003)
Last	Cisco IOS

Now any overlapping fragments that Snort sees destined for subnet 10.4.1.x are reassembled using the "first" fragmentation policy, so that Snort reassembles fragments destined to those hosts in precisely the same way as the Windows hosts themselves.

Let's see exactly how this works on some actual traffic, using a Snort rule that looks for a Windows NETBIOS exploit that can cause a buffer overflow.

```
alert tcp $EXTERNAL_NET any -> $HOME_NET 139 (msg:"NETBIOS SMB
SMB_COM_TRANSACTION Max Parameter and Max Count of 0 DOS Attempt";
flow:to_server,established; content:"|00|"; depth:1; content:"|FF|SMB%";
depth:5; offset:4; content:"|00 00 00 00|"; depth:4; offset:43; )
```

 ©Sourcefire, Inc. All rights reserved. January 21, 2005

The following tcpdump hexadecimal output shows an unfragmented attack packet that triggers the Snort rule. The shaded hexadecimal characters indicate the TCP payload that contains the content that Snort evaluates. The bold, underlined characters indicate the specific characters that the Snort rule examines.

```
08:59:38.543620 10.4.15.12.1115 > 10.4.10.28.139: P 1985349754:1985349853(99)
ack 1636263536 win 17241NBT Packet (DF)
0x0000       4500 008b 0db1 4000 8006 bf8c 0a04 0f0c
0x0010       0a04 0a1c 045b 008b 7656 087a 6187 6670
0x0020       5018 4359 33c9 0000 0000 005f ff53 4d42
0x0030       2500 0000 0000 0000 0000 0000 0000 0000
0x0040       0000 0000 0008 2404 0008 0000 0e13 0000
0x0050       0000 0000 0000 0000 0000 0000 0000 0013
0x0060       004c 0000 005f 0000 0020 005c 5049 5045
0x0070       5c4c 414e 4d41 4e00 6800 5772 4c65 6800
0x0080       4231 3342 577a 0001 00e0 ff
```

The six fragmented packets that follow represent an attacker's attempt to use overlapping fragments with a "first" reassembly policy. Only a host or an IDS capable of fragmentation reassembly based on the "first" policy can reassemble these fragments to yield the packet seen above. Not coincidentally, in the packets below, partial or entire fragment content that is discarded has an 8-byte value of 0xff ff ff ff ff ff ff ff.

Fragment 1

```
12:15:09.947614 10.4.15.12.1115 > 10.4.10.28.139: P 1985349754:1985349778(24)
ack 1636263536 win 17241 <eol>NBT Packet (frag 3505:48@0+)
0x0000       4500 0044 0db1 2000 8006 dfd3 0a04 0f0c
0x0010       0a04 0a1c 045b 008b 7656 087a 6187 6670
0x0020       6018 4359 23c5 0000 0000 0000 0000 005f
0x0030       ff53 4d42 2500 0000 0000 0000 0000 0000
0x0040       0000 0000
```

Fragment 2

```
12:15:09.948036 10.4.15.12 > 10.4.10.28: (frag 3505:16@56+)
0x0000       4500 0024 0db1 2007 8006 dfec 0a04 0f0c
0x0010       0a04 0a1c 0008 0000 0e13 0000 0000 0000
0x0020       0000 0000
```

Fragment 3

```
12:15:09.948412 10.4.15.12 > 10.4.10.28: (frag 3505:24@72+)
0x0000       4500 002c 0db1 2009 8006 dfe2 0a04 0f0c
0x0010       0a04 0a1c 0000 0000 0000 0013 004c 0000
0x0020       005f 0000 0020 005c 5049 5045
```

Fragment 4

```
12:15:09.948729 10.4.15.12 > 10.4.10.28: (frag 3505:32@32+)
0x0000       4500 0034 0db1 2004 8006 dfdf 0a04 0f0c
0x0010       0a04 0a1c ffff ffff ffff ffff ffff ffff
0x0020       ffff ffff 0000 0000 0008 2404 ffff ffff
0x0030       ffff ffff
```

Fragment 5

```
12:15:09.949156 10.4.15.12 > 10.4.10.28: (frag 3505:24@72+)
0x0000       4500 002c 0db1 2009 8006 dfe2 0a04 0f0c
0x0010       0a04 0a1c ffff ffff ffff ffff ffff ffff
```

 ©Sourcefire, Inc. All rights reserved. January 21, 2005

```
0x0020        ffff ffff ffff ffff ffff ffff
```

Fragment 6

```
12:15:09.949540 10.4.15.12 > 10.4.10.28: (frag 3505:27@96)
0x0000        4500 002f 0db1 000c 8006 ffdc 0a04 0f0c
0x0010        0a04 0a1c 5c4c 414e 4d41 4e00 6800 5772
0x0020        4c65 6800 4231 3342 577a 0001 00e0 ff
```

According to the Paxson/Shankar model, a host using the "first" policy reassembles the packets using the following scheme; the shaded numbers are the favored fragments:

```
111 22333
 4444 555666

111422333666
```

Translated, this means that the "first" policy uses all 24 bytes from Fragment 1, 8 bytes (offset 16-23) from Fragment 4, all 16 bytes from Fragment 2, all 24 bytes from Fragment 3, and finally, all remaining bytes from Fragment 6 for the reassembly process.

Fragment 1

```
0000 005f ff53 4d42 2500 0000 0000 0000 0000 0000 0000 0000
```

Fragment 4

```
0000 0000 0008 2404
```

Fragment 2

```
0008 0000 0e13 0000 0000 0000 0000 0000
```

Fragment 3

```
0000 0000 0000 0013 004c 0000 005f 0000 0020 005c 5049 5045
```

Fragment 6

```
5c4c 414e 4d41 4e00 6800 5772 4c65 6800 4231 3342 577a 0001 00e0 ff
```

Combine the above fragment payloads and you see that the reassembled payload is identical to the original baseline unfragmented payload seen below.

```
08:59:38.543620 10.4.15.12.1115 > 10.4.10.28.139: P 1985349754:1985349853(99)
ack 1636263536 win 17241NBT Packet (DF)
0x0000        4500 008b 0db1 4000 8006 bf8c 0a04 0f0c
0x0010        0a04 0a1c 045b 008b 7656 087a 6187 6670
0x0020        5018 4359 33c9 0000 0000 005f ff53 4d42
0x0030        2500 0000 0000 0000 0000 0000 0000 0000
0x0040        0000 0000 0008 2404 0008 0000 0e13 0000
0x0050        0000 0000 0000 0000 0000 0000 0000 0013
0x0060        004c 0000 005f 0000 0020 005c 5049 5045
0x0070        5c4c 414e 4d41 4e00 6800 5772 4c65 6800
0x0080        4231 3342 577a 0001 00e0 ff
```

As demonstrated, this uniquely empowers Snort to handle attempted evasion tactics using overlapping fragments.

 ©Sourcefire, Inc. All rights reserved. January 21, 2005

Implementation of the Paxson/Shankar Model

Implementation of the Paxson/Shankar model requires an API that can send and read network traffic. Available Perl packages to accomplish this are Net::PcapUtils, Net::RawSock, and NetPacket. PcapUtils sniffs from an interface and captures traffic, RawSock writes IP datagrams to the network, and NetPacket decodes captured IP datagrams and encodes the same or modified datagrams to be passed to RawSock. You can use these packages to create an ICMP echo request with the appropriate fragmentation and content, write them to the network, and listen for an ICMP echo reply. Based on the reply, you can write additional Perl code to analyze the policy used for reassembly.

There are several technical challenges in implementing the model. First, a Linux 2.4 host running iptables prevents outgoing packets from being fragmented. You must configure iptables to allow fragments to a particular destination host or hosts. For instance, to send fragments to destination host 10.4.11.73, enter the command "iptables –A output –f –d 10.4.11.73 –j ACCEPT". Second, the most current version of NetPacket, 0.04, has an error in the code in module IP.pm that does not properly format the IP header fragment-related fields, MF flag and fragment offset, to create a fragmented packet. You must correct and recompile the code to modify those fields. See Appendix E for the patch. Finally, as previously discussed, you should generate a single set of fragments with an ICMP checksum that is the same regardless of the reassembly policy, rather than generate five different sets of fragments with an ICMP checksum unique to each reassembly policy.

One way to generate one set of fragments with the same ICMP checksum is to have each 8-byte chunk contain the same data, but in a different order. By changing the order of the content, each fragment can have a unique payload and be identified in the reassembly scheme. For the ICMP checksum to remain the same, you can change the content by swapping 16-bit fields only. Here is a scheme used:

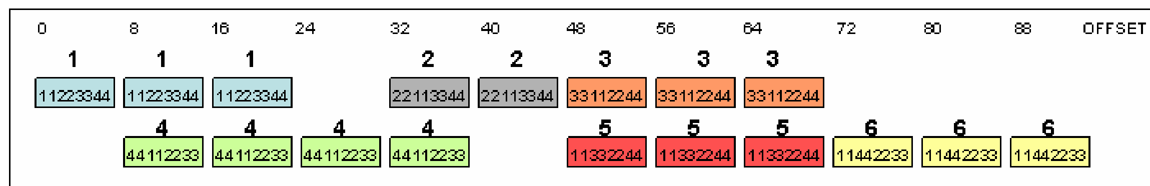

Figure 4 – Sample fragment payload for unchanging ICMP checksum

```
Fragment 1 8-byte content:  11223344
Fragment 2 8-byte content:  22113344
Fragment 3 8-byte content:  33112244
Fragment 4 8-byte content:  44112233
Fragment 5 8-byte content:  11332244
Fragment 6 8-byte content:  11442233
```

All contents are expressed in ASCII representation. Regardless of the fragmentation reassembly policy used, the twelve 8-byte chunks of reassembled data always has the same ICMP checksum value assuming all other ICMP fields (type, code, echo request ID, echo request sequence number) remain unchanged. As an example of a policy response, a BSD reassembled ICMP echo reply has a payload depicted as follows:

 ©Sourcefire, Inc. All rights reserved. January 21, 2005

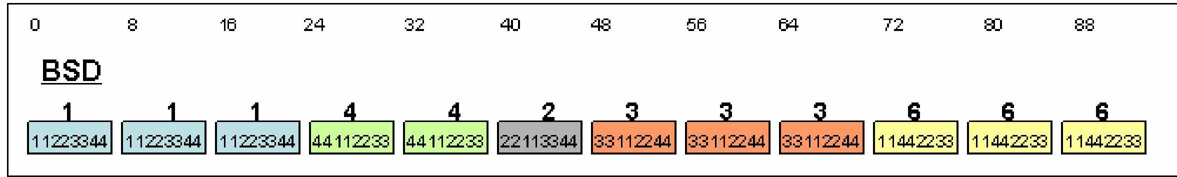

Figure 5 – BSD policy reassembly for sample fragment payload

The ASCII content reflected in the ICMP echo reply is:

```
11223344 11223344 11223344 44112233 44112233 22113344 33112244 33112244
33112244 11442233 11442233 11442233
```

This is not the only encoding scheme that maintains an unchanging ICMP checksum; it is just a simple one that is easily implemented.

As far as implementation details, you must generate a set of fragments using a first fragment that carries the appropriate IP header fragmentation flags, an ICMP header representing an ICMP echo request, and fragment 1 payload. Fashion subsequent fragments by manipulating the IP header fragmentation fields and data payloads. Remember that only the first fragment carries the protocol header followed by payload, while subsequent fragments contain payload only. Also, the fragment offsets of payloads are relative to the protocol header. Because the protocol header is an ICMP echo request, it has an 8-byte length and occupies fragment offset 0. All fragment offsets are relative to the ICMP header.

All fragment offsets detailed below are values that are placed in the IP header field for fragment offsets and must be multiplied by 8 to compute the true offset. For example, a fragment offset value of 1 in the IP header fragment offset field indicates an actual fragment offset of 8. The technical explanation for representing offset values in the IP header as an eighth of their real value is based on the number of bits allocated for the fragment offset in the IP header. Thirteen bits are available for the fragment offset value, however, an IP datagram length is allocated 16 bits. Theoretically, though not often seen, a fragment offset can be greater than 2^{13} or 8192. The multiplication factor of 8 comes from 13 bits being 3 bits less than 16 bits ($2^3 = 8$).

The details of each fragment follow:

Fragment	Fragment Offset*	MF Flag	ICMP Header	ASCII Payload
1	0	1	Type = 08 Code = 00 Checksum – Applies to all fields in ICMP header and all fragment payloads combined ID (2 bytes) = any value Seq (2 bytes) = any value	112233441122334411223344
2	4	1		2211334422113344
3	6	1		331122443311224433112244
4	1	1		44112233441122334411223344112233
5	6	1		113322441133224411332244

©Sourcefire, Inc. All rights reserved. January 21, 2005

6	9	0		114422331144223311442233

*The above fragment offset values are relative to the end of the ICMP header.

Creating a More Generic Program to Implement the Model for all Protocols

The implementation of fragmentation of an ICMP echo request payload for operating system identification provides a good foundation for understanding not only patterns for reassembly identification, but also what is required to fragment a TCP or UDP packet that contains real data. In fact, you can test your IDS/IPS with a program that uses different policies to fragment traffic that your IDS/IPS normally flags or blocks. This shows you if your IDS/IPS is susceptible to evasion by fragmentation. In their landmark paper, "Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection," Ptacek and Newsham exposed the problems presented by fragmentation overlap:

> "Overlap resolution is further complicated by the fact that data from conflicting fragments is used differently depending on their positions. In some situations, conflicts are resolved in favor of the new data. In others, the old data is preferred and the new data is discarded. An IDS that does this incorrectly is vulnerable to evasion attacks."[2]

The discussion that follows explains the technical details required for a program to read a pcap file containing attack traffic, fragment the payload(s), and write the fragments back to the network. The same API Perl packages used in the implementation of the ICMP echo request fragments are necessary for this more generic program. You can use the same PcapUtils to read the packets from the dump file as well as from the network interface.

Additional functionality required for this code is:

- The need to solicit user input for the fragmentation policy to be implemented
- The need to deal with protocol headers other than an ICMP echo request
- Preparation to write bogus traffic to the network where the destination IP address does not necessarily exist

There are new challenges presented by these additional requirements. First, the fragmentation model depends on having the fragmented payload at a 0-offset relative to the end of the protocol header. This works with an 8-byte ICMP header. Since fragments must fall on 8-byte boundaries (as implemented by the offset value in the IP header), the end of the protocol header must fall on an 8-byte boundary. If not, the fragmentation scheme used gets skewed by having fragments that may or may not be lumped with the protocol header itself.

For example, assume you have a TCP header that is 20 bytes long – standard for a TCP packet that has no TCP options. In this case, there are 4 bytes of TCP header that must be lumped with the payload for it to fall on an 8-byte boundary. However, the first fragment has 4 bytes of TCP header and 20 bytes of payload data instead of 24 bytes of payload data that is used in the model. To resolve this problem, you can pad the TCP header with a 4-byte TCP EOL option. This is essentially a NOOP in the TCP options that does not change the packet or its interpretation in any way other than to pad a TCP header to fall on an 8-byte boundary, if required. If you do this, though, you must recompute the TCP checksum to include the new data. This problem is not present in UDP since it has a standard 8-byte header. ICMP headers that do not fall on 8-byte boundaries have no easy fix. However, there are few, if any, instances of ICMP packets being used in attacks (other than fragmentation attacks such as the "ping of death").

 ©Sourcefire, Inc. All rights reserved. January 21, 2005

The RawSock program is not ideal for this application since it really does not create a “raw socket.” Instead, it requires a MAC address for the destination IP of any packet that it writes. This is not a problem when the destination host either is within the netmask of the sending host and alive, or is not within the netmask of the sending host. In the former case, an ARP request is issued and an ARP reply is returned. In the latter case, the MAC address of the router is used. However, if the destination host in the processed pcap record is within the netmask, but is not alive, no packet is ever written. To avoid this problem, you must add bogus MAC addresses to the ARP cache of the sending host for destination hosts that require them.

Another problem was exposed in the NetPacket encode code when a packet was sent that had an IP datagram length of less than 46 bytes. If an IP datagram packet sent over Ethernet has a length of 46 bytes, it must be zero-padded to be 46 bytes. When such a packet is decoded, the IP datagram length reflects the true length of the datagram. However, when encoded by the program IP.pm, the length mistakenly includes the zero padding. For example, if a TCP ACK packet contains a 20-byte IP header and 20-byte TCP header, it must be zero-padded to 46 bytes. The IP total datagram length should be 40 bytes. But, when such a packet is encoded in the IP.pm code, the IP datagram length is recomputed to be 46 bytes. This makes TCP think that it has 6 bytes of TCP data. The statement that recomputes the IP header length has to be bypassed when rewriting unaltered or padded packets. Instead of using the supplied IP.pm for encoding, the good code was placed in a local subroutine to avoid the problem.

A final consideration for this program is the need for filler bytes for fragment content that is overwritten. While the ICMP echo request program simply writes the same set of fragments and analyzes the response for fragmentation policy, it does not need to consider those replaced fragments or partial fragments. This new program takes payload data and shapes it for a given fragmentation policy. In other words, it needs to take data from the pre-fragmented payload only when a given fragment is used in the reassembly policy. If a fragment is overwritten in a given policy, the fragment or partial fragment content must be populated with some kind of filler data. The filler data used in this program is an 8-byte chunk of 0xFFFFFFFF. This can be readily identified in the packet fragment as filler and it is very unlikely that true payload has this filler.

The following example shows how the payload must be fashioned per policy. The notation substr(payload, *x*, *y*) indicates that the program should extract *y* bytes of payload beginning at offset *x*. If no *y* value is given, the program should extract all remaining data from the payload. The variable $F represents the value 0xFFFFFFFF.

Frag Payload	BSD	BSD-right	Linux	First	Last
2	$F + substr(pay, 40, 8)	substr(pay,32,16)	$F + substr(pay, 40, 8)	substr(pay,32,16)	$F + substr(pay, 40, 8)
3	substr(pay, 48, 24)	$F + $F + $F	$F + $F + $F	substr(pay, 48, 24)	$F + $F + $F
4	$F + $F + substr(pay, 24, 16)	substr(pay, 8, 24) + $F	$F + $F + substr(pay,24,16)	$F + $F + substr(pay, 24, 8) + $F	substr(pay, 8, 32)
5	$F + $F + $F	substr(pay,48,24)	substr(pay,48,24)	$F + $F + $F	substr(pay,48,24)
6	substr(pay, 72)	substr(pay, 72)	substr(pay, 72)	substr(pay, 72)	substr(pay, 72)

The details of each fragment follow:

Fragment	Fragment Offset*	MF Flag	Protocol Header	Payload

 ©Sourcefire, Inc. All rights reserved. January 21, 2005

1	0	1	Pertinent protocol header values Checksum – Applies to all fields in protocol header (and pseudo-header for TCP and UDP) and all fragment payloads combined	Policy (BSD, Linux, etc.) fragment 1
2	4	1		Policy (BSD, Linux, etc.) fragment 2
3	6	1		Policy (BSD, Linux, etc.) fragment 3
4	1	1		Policy (BSD, Linux, etc.) fragment 4
5	6	1		Policy (BSD, Linux, etc.) fragment 5
6	9	0		Policy (BSD, Linux, etc.) fragment 6

*The above fragment offset values are relative to the end of the protocol header.

Actual offset = (protocol header length/8) + fragment offset

According to the Paxon/Shankar model, you must find a minimum number of bytes of payload to fragment. While 94 bytes takes care of all fragments, fragment 6 is handled the same way for all models. A minimum number of 80 bytes is used to create at least some bytes of fragment 6. All packets, whether fragmented or not, should be read from the pcap file and written to the network.

©Sourcefire, Inc. All rights reserved. January 21, 2005

Beyond the Paxson/Shankar Model

The Paxon/Shankar model is not comprehensive enough since it does not test all different combinations of fragment placement. For instance, it never examines a subsequent fragment with a starting offset greater than the original fragment and an ending offset less than the original fragment. Windows and Solaris hosts reassemble this particular example differently and no longer are strictly a "first" policy since the subsequent fragment is the honored one. Consequently, Steve Sturges and Judy Novak, both of Sourcefire, developed a new all-encompassing model to deal with all possible fragmentation combinations.

The table below offers all possible scenarios of fragmentation overlap. We must consider all combinations of the placement of an original fragment compared to an overlapping subsequent fragment. As important, it is necessary to compare the placement of a subsequent fragment with an original fragment. From the table, it is apparent that the Paxson/Shankar model missed several different combinations of overlapping fragments. The "proposed model" by Sturges and Novak extends the original Paxson/Shankar model to include these missing cases. If you recall, the Paxson/Shankar model has original fragments 1, 2, and 3 followed by subsequent fragments 4, 5, and 6. The Sturges/Novak extension adds original fragments 3.1, 3.2, 3.3, 3.4, 3.5 and 3.6 that immediately follow original fragments 1-3 from the Paxson/Shankar model. It also adds subsequent fragments 7, 8, 9, 10, and 11 that follow subsequent fragments 4-6.

	Original Fragment vs. Subsequent Fragment		**Subsequent Fragment vs. Original Fragment**	
	Current Model	**Proposed Model**	**Current Model**	**Proposed Model**
Starts before, ends after	No coverage	Fragments 3.3 & 8	No coverage	Fragments 6 & 3.1
Starts before, ends before	Fragments 1 & 4	Fragments 1 & 4	Fragments 4 & 2	Fragments 4 & 2
Starts before, ends same	No coverage	Fragments 3.4 & 9	No coverage	Fragments 7 & 3.2
Starts same, ends after	No coverage	Fragments 3.5 & 10	No coverage	Fragments 11 & 3.6
Starts same, ends before	No coverage	Fragments 3.6 & 11	No coverage	Fragments 10 & 3.5
Starts same, ends same	Fragments 3 & 5	Fragments 3 & 5	Fragments 5 & 3	Fragments 5 & 3
Starts after, ends after	Fragments 2 & 4	Fragments 2 & 4	Fragments 4 & 1	Fragments 4 & 1
Starts after, ends before	No coverage	Fragments 3.1 & 6	No coverage	Fragments 8 & 3.3
Starts after, ends same	No coverage	Fragments 3.2 & 7	No coverage	Fragments 3.4 & 9

©Sourcefire, Inc. All rights reserved. January 21, 2005

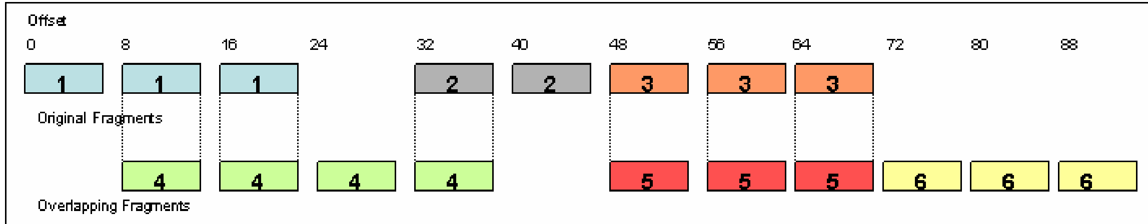

Figure 6 – Original Paxson/Shankar Model

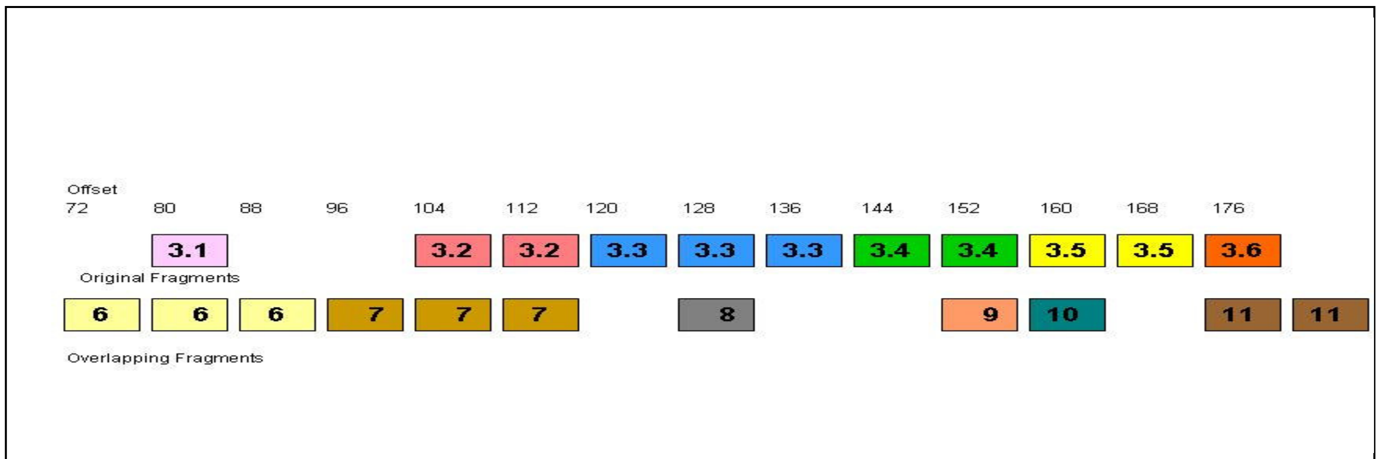

Figure 7 – Sturges/Novak Model Addition

Revisiting the Paxson/Shankar model, the five different reassembly policies are:

- **BSD** favors an original fragment with an offset that is less than or equal to a subsequent fragment.
- **BSD-right** favors a subsequent fragment when the original fragment has an offset that is less than or equal to the subsequent one.
- **Linux** favors an original fragment with an offset that is less than a subsequent fragment.
- **First** favors the original fragment with a given offset.
- **Last** favors the subsequent fragment with a given offset.

The new model yields seven different reassembly policies and a modification to the BSD-right model:

- **BSD** favors an original fragment with an offset that is less than or equal to a subsequent fragment.
- **BSD-right** favors a subsequent fragment when the original fragment has an offset that is less than or equal to the subsequent one **except** when the original fragment ends at the same or greater offset than the subsequent fragment. In this case, BSD-right favors the original fragment.
- **Linux** favors an original fragment with an offset that is less than a subsequent fragment.
- **First** favors the original fragment with a given offset.
- **Windows** favors the original fragment **except** if a subsequent fragment offset begins before the original fragment and ends after the original fragment. In this case, Windows favors the subsequent fragment.

©Sourcefire, Inc. All rights reserved. January 21, 2005

- **Solaris** favors the original fragment **except** if a subsequent fragment offset begins before the original fragment and ends at an offset equal to or greater than the original fragment. In this case, Solaris favors the subsequent fragment.
- **Last** favors the subsequent fragment with a given offset.

The new model reassembly policies are as follows:

BSD-right:

<1><4><4><4><2><2><5><5><5><6><6><6><7><7><7><3_3><3_3><3_3><3_4><3_4><3_5><3_5><11><11>

BSD:

<1><1><P1><4><4><2><3><3><3><6><6><6><7><7><7><3_3><3_3><3_3><3_4><3_4><3_5><3_5><3_6><11>

Linux:

<1><1><1><4><4><2><5><5><5><6><6><6><7><7><7><3_3><3_3><3_3><3_4><3_4><10><3_5><11><11>

First:

<1><1><1><4><2><2><3><3><3><6><3.1><6><7><3_2><3_2><3_3><3_3><3_3><3_4><3_4><3_5><3_5><3_6><11>

Windows:

<1><1><1><4><2><2><3><3><3><6><6><6><7><3_2><3_2><3_3><3_3><3_3><3_4><3_4><3_5><3_5><3_6><11>

Solaris:

<1><1><1><4><2><2><3><3><3><6><6><6><7><7><7><3_3><3_3><3_3><3_4><3_4><3_5><3_5><3_6><11>

Last:

<1><4><4><4><4><2><5><5><5><6><6><6><7><7><7><3_3><8><3_3><3_4><9><10><3_5><11><11>

The "first" and "last" policies are theoretical as we did not have access to any operating system that responds as expected. The "last" policy used to apply to Cisco IOS versions, but it appears that current versions of Cisco IOS do not respond to any overlapping fragments. We speculate that they simply drop these packets, a wise policy for all operating systems.

In addition to the enhanced overlapping fragmentation model, Steve Sturges wondered how different operating systems respond to a series of related fragments that have more than one fragment that purports to be the last fragment. He constructed a model that uses four fragments, two with a more fragments flag set, two with the more fragments flag not set. These fragments are sent in different order to analyze how a given operating systems responds.

The following tests use fragments labeled P1, P2, P3, and P4. Fragments P1 and P3 have the more fragments flag set, fragments P2 and P4 do not. Fragment P1 falls at offset 0 and contains an ICMP echo request header and a

©Sourcefire, Inc. All rights reserved. January 21, 2005

payload content of "AABBCCDD". Fragment P2 is at offset 16 since it follows the 8-byte ICMP header and the 8-byte payload of P1. It has a payload of "BBCCDDAA". Fragment P3 follows at offset 24 with a payload of "CCDDAABB". Finally, P4 falls at offset 32 with a payload of "DDCCBBAA". Each payload is unique so if a host responds with an ICMP echo reply, the ICMP echo reply payload identifies how the host reassembles the fragments.

	Win2k/WinXP	**FreeBSD 5.1/ OpenBSD 3.6**	**Solaris 10**	**Linux 2.2**	**Linux 2.4**	**Cisco IOS 12**
P1-P2-P3-P4	P1-P2	P1-P2	P1-P2	P1-P2	P1-P2	P1-P2
P2-P4-P1-P3	P1-P2-P3-P4	P1-P2-P3-P4	P1-P2-P4	P1-P2-P3-P4	P1-P2	P1-P2
P4-P2-P1-P3	No reply	P1-P2-P3-P4	P1-P2-P3-P4	No reply	No reply	P1-P2-P3
P4-P2-P1	No reply	No reply	No reply	No reply	No reply	P1-P2
P3-P2-P4-P1	P1-P2-P3-P4	P1-P2-P3-P4	P1-P2-P3-P4	P1-P2-P3-P4	No reply	P1-P2-P3-P4
P1-P3-P4-P2	No reply	P1-P2-P3-P4	P1-P2-P3-P4	No reply	No reply	P1-P2-P3-P4
P1-P3-P2-P4	P1-P2-P3-P4	P1-P2-P3-P4	P1-P2-P3-P4	No reply	No reply	P1-P2-P3-P4

These tests present an additional implementation challenge. Unlike the overlapping fragments, where checksums are an issue, but remedied by using a unique 8-byte fragment payload with the same checksum; it is not possible to use one ICMP checksum for all possible outcomes of multi-last fragment reassembly. This is because different operating systems accept or reject particular fragments whereas with overlapping fragments, a particular operating system selects either an original or subsequent fragment. The problem is that we do not know beforehand which or how many fragments a given operating system considers acceptable and whose fragment payloads are computed in the ICMP checksum. In order to deal with this new situation, tests are run multiple times, each with a different ICMP checksum appropriate for different reassembly methods.

As with fragmentation overlap, there are many unique methods of reassembling multiple last fragments. These differences once again reinforce the need for target-based reassembly. Employing multiple last fragments is another means of IDS or IPS evasion if the target host and IDS/IPS reassembly is not identical. The new discoveries concerning target-based fragmentation reassembly from the new fragmentation overlap model and multiple last fragments are incorporated in the Snort frag3 preprocessor. Snort and the target host reassemble these mutant fragments identically when Snort is configured to use the correct target-based fragmentation policy for the destination host.

As an aside, NetPacket is the software that was used to implement the original Paxson/Shankar model. Several months later, when we began research on the extended fragment overlap and multi-last fragment models, a much-improved python API called scapy was available. It is written by Philippe Biondi and it makes crafting unconventional packets almost trivial, especially when compared with the quirks discovered in NetPacket.

 ©Sourcefire, Inc. All rights reserved. January 21, 2005

Summary

This paper discusses target-based fragmentation reassembly in theory and in application using the Snort frag3 preprocessor. It clarifies the terse explanation given for fragmentation evasion in the Paxson/Shankar paper. It also demonstrates how fragments are reassembled by using a Snort rule with unfragmented and overlapping fragments. Finally, it discusses several practical uses for implementing the model and code presented. Understanding this theory can be particularly helpful.

It may help you identify remote operating systems through observation of the reassembly policy used by the examined host. This is best implemented using the ICMP echo request with a unique payload that, when reflected in the returned ICMP echo response, can determine the fragmentation reassembly policy. Use of other services or protocols for fragmentation policy identification is impractical because few services or protocols simply echo back what is sent. They may require additional input or authentication and may give no indication of the reassembly employed. Successful identification of a specific reassembly policy can validate results of other active or passive scanners.

You can use the discoveries about fragmentation overlaps and multiple last fragments to better understand how attackers can evade your IDS/IPS, and be more prepared to mitigate threats. As Ptacek and Newsham noted, an intrusion detection system that is not aware of the reassembly policy used by a destination host cannot possibly know how to perform the reassembly appropriately.

©Sourcefire, Inc. All rights reserved. January 21, 2005

References

[1] Thomas H. Ptacek and Timothy N. Newsham, ”Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection”, January 1998.

[2] Umesh Shankar and Vern Paxson, “Active Mapping: Resisting NIDS Evasion Without Altering Traffic”, 2003.

©Sourcefire, Inc. All rights reserved. January 21, 2005

Appendices

Appendix A - Perl packages used from search.cpan.org

```
NetPacket-0.04
Net-Pcap-0.03
Net-PcapUtils-0.01
Net-RawSock-1.0
```

Note: All source code is copyright of Sourcefire, Inc. and distributed under the GPL.

Appendix B - Fragmented ICMP echo request code

```
#!/usr/bin/perl

use Net::PcapUtils;
use Net::RawSock;
use NetPacket::IP;
use NetPacket::ICMP;

if ($#ARGV < 1) {
   print "Need to supply destination IP and source IP\n";
   exit;
   }

$dst  = $ARGV[0];
$src   = $ARGV[1];

$BSD =   "111442333666";
$BSDr =  "144422555666";
$linux = "111442555666";
$first = "111422333666";
$last =  "144442555666";



if (($pid = fork()) < 0) {
  die  "cannot fork, dying";
}
else {
  if ($pid == 0) {
    &sniff_reply;
    exit(0);
  }
}

sleep 2;
$all_pay  =
"0000112233441122334411223344112233441122334411223344112233441122334411223344112233441122334411223344"
;

$off   = 0;
$flgs  = 1;
$payload = "0000112233441122334411223344";
```

 ©Sourcefire, Inc. All rights reserved. January 21, 2005

```
$first_frag = 1;

&create_packet($off, $flgs, $payload,$first_frag);

$off   = 40;
$flgs  = 1;
$payload = "2211334422113344";
$first_frag = 0;

&create_packet($off, $flgs, $payload,$first_frag);

$off   = 56;
$flgs  = 1;
$payload =  "331122443311224433112244";
$first_frag = 0;

&create_packet($off, $flgs, $payload,$first_frag);

$off   = 16;
$flgs  = 1;
$payload =  "44112233441122334411223344112233";
$first_frag = 0;

&create_packet($off, $flgs, $payload,$first_frag);

$off   = 56;
$flgs  = 1;
$payload =  "113322441133224411332244";
$first_frag = 0;

&create_packet($off, $flgs, $payload,$first_frag);

$off   = 80;
$flgs  = 0;
$payload =  "114422331144223311442233";
$first_frag = 0;

&create_packet($off, $flgs, $payload,$first_frag);


sub create_packet {
  ## Create IP
  my $ip = NetPacket::IP->decode('');

  $off  = $_[0];
  $flgs = $_[1];
  $pay  = $_[2];
  $ff   = $_[3];

  ## Init IP

  $ip->{ver}        = 4;
  $ip->{hlen}       = 5;

  $temp = $ip->{hlen} &0x0f;
  $ver_len = $temp | (($ip->{ver} << 4) & 0xf0);

  if ($ff) {
    $len = 24 + length($pay);
  }
  else {
    $len = 20 + length($pay);
  }

  $ip->{tos}        = 0;
  $ip->{id}         = 0x1d1d;
  $ip->{ttl}        = 0x5a;
  $ip->{proto}      = 1;
  $ip->{src_ip}     = $src;
```

 ©Sourcefire, Inc. All rights reserved. January 21, 2005

```
  $ip->{dst_ip}    =  $dst;

  $offset = $flgs << 13;
  $offset = $offset | (($off >> 3) & 0x1fff);
  $cksum = 0;

  $src_ip = gethostbyname($src);
  $dst_ip = gethostbyname($dst);

  $pack_ip = pack(c, $ver_len) . pack(c, $ip->{tos}) . pack(n, $len) .
             pack(n, $ip->{id}) . pack(n, $offset) . pack(c, $ip->{ttl}).
             pack(c, $ip->{proto}) . pack(n, $cksum) .
             pack(a4a4, $src_ip, $dst_ip);

  ## Create ICMP
  my $icmp = NetPacket::ICMP->decode('');

  ## Init ICMP
  $icmp->{type}     = 8;
  $icmp->{code}     = 0;
  $icmp->{data}     = $pay;

  ## Assemble
  if ($ff) {
    $cksum = &checksum($icmp->{type}, $icmp->{code}, $all_pay);
    $icmp = pack("ccna*", $icmp->{type}, $icmp->{code}, $cksum, $all_pay);
  }
  else {
    $icmp = pack("a*", $pay);
  }

  $pkt = $pack_ip . $icmp;


  ## Write to network layer
  Net::RawSock::write_ip($pkt);
}


sub checksum {

  $type = $_[0];
  $code = $_[1];
  $data = $_[2];

# Put the packet together for checksumming
  $zero = 0;

  $packet = pack("CCna*", $type, $code,
                 $zero, $data);

  $csum = NetPacket::htons(NetPacket::in_cksum($packet));
  return($csum);
}



# Load modules
use Net::PcapUtils;
use NetPacket::Ethernet qw(:strip);
use NetPacket::ICMP;
use NetPacket::IP qw(:strip);


sub sniff_reply {

# Start sniffin in promisc mode
Net::PcapUtils::loop(\&sniffit,
                     Promisc => 1,
```

 ©Sourcefire, Inc. All rights reserved. January 21, 2005

```
                      FILTER => 'icmp',
                      SNAPLEN => 1500,
                      DEV => 'eth0');
}

# Callback
sub sniffit
{

  my ($args,$header,$packet,$i,$l) = @_;
  $ip = NetPacket::IP->decode(eth_strip($packet));
  $icmp = NetPacket::ICMP->decode($ip->{data});

  if ($icmp->{type} eq 0) {
    print "$ip->{src_ip} --> $ip->{dest_ip}:$icmp->{type}\n";
    $idata = substr($icmp->{data},4,$len);
    $idata = substr($icmp->{data},4);

    print "Actual ICMP response data is $idata\n";

    $idata = extract_response($idata);

    print "Manipulated ICMP response is $idata\n";

   if ($idata == $BSD) {
     print "Matched BSD\n";
   }
   else {
     if ($idata == $BSDr) {
       print "Matched BSDr\n";
     }
     else {
       if ($idata == $linux) {
         print "Matched Linux\n";
       }
         else {
           if ($idata == $first) {
             print "Matched First/Windows\n";
           }
           else {
             if ($idata == $last) {
               print "Matched Last\n";
             }
             else {
               print "Matched Nothing\n";
             }
           }
         }
       }
     }
 exit(0);
   }
}

sub extract_response {

  $icmp = $_[0];

  for ($i = 0; $i < 97; $i = $i+8)  {
      $temp = substr($icmp,$i,1);
      if ($temp == 1) {
        if (substr($icmp, $i+2, 1) == 3) {
          $temp = 5;
        }
        else {
          if (substr($icmp, $i+2, 1) == 4) {
            $temp = 6;
          }
        }
```

 ©Sourcefire, Inc. All rights reserved. January 21, 2005

```
      }
    $string = $string . $temp;
  }

  return($string);
}
```


©Sourcefire, Inc. All rights reserved. January 21, 2005

Appendix C - Code to read a pcap file and fragment payload and write to network

```
#!/usr/bin/perl

#---------------------------------------------------------------------------
#
# This code reads a correctly formatted pcap file and fragments all records
# that have a data payload of 80 bytes or more. The fragmentation is done
# according to the policy that the user supplies in the command line:
# bsd, bsdr, linux, first, last.
#
# The fragmentation policies and paradigms used are from Vern Paxson's
# paper on fragmentation and the different ways that various operating
# systems reassemble a complicated fragment with overlapping
# (entire, left, and/or right edges) fragments.
#
# The model that he uses and employed in this code requires a minimum of
# 80 bytes of payload in a packet to account for all fragments.
#
# The output created is written back on the wire. It can be captured
# using tcpdump with appropriate arguments. The purpose of this program
# is to be able to test Snort's frag3 preprocessor using actual exploits
# and fragmented pcap files.
#
#
#    Judy Novak - Nov. 2004
#----------------------------------------------------------------------------

use Net::RawSock;
use Net::PcapUtils;
use NetPacket::Ethernet qw(:strip);
use NetPacket::IP qw(:strip);
use NetPacket::TCP;
use NetPacket::UDP;
use NetPacket::ICMP;


if ($#ARGV < 1)  {
   print "\nNeed to supply the following parameters in this order:\n";
   print "  1) Fragmentation policy to use (bsd, bsdr, linux, first, last)\n";
   print "  2) Name of the pcap file\n\n";
   exit;
}

my($ip, $tcp, $udp, $icmp, $protohdr_len, $pkt, $ip_proto, $incount, $outcount);

$incount = 0;
$outcount = 0;

$policy =    $ARGV[0];
$dump_file = $ARGV[1];

if ($policy ne "bsd" and $policy ne "bsdr" and $policy ne "linux" and $policy ne "last" and $policy ne
"first") {
   print "Invalid fragmentation policy.\n";
   print "Must be: bsd, bsdr, linux, first, last.\n";
   exit;
}

$min_frag_length = 80;
$frag_flag = 0;


# ARP entries can be a problem for the RawSock routine since it
```

 ©Sourcefire, Inc. All rights reserved. January 21, 2005

```
# seems to require a MAC address for any destination IP that falls
# in the current netmask. It sends packets just fine if the
# destination IP address MAC address is in the current ARP cache or
# the destination IP is outside of the netmask. Just in case a
# destination IP address belongs to a host in the netmask that is
# not currently up, a bogus ARP entry is assigned. These are deleted
# after all records have been read/written. However, they may hang
# around as "incomplete" in when doing "arp -a". Eventually, they
# time out.

$mac_n = 0;
$mac_beg = "aa:bb:cc:dd:ee:";
$fake_arp;

&arp_cache;

## Open and read the user-supplied pcap file.

$pcap_t = Net::Pcap::open_offline($dump_file, \$err);

if (!defined($pcap_t)) {
    die("Net::Pcap::open_offline of $dump_file returned error $err");
}

&read_pkts;

if ($frag_flag == 0)  {
  print "No packets found in file $dump_file that were at least $min_frag_length long - no fragments
generated.\n";
}

print "Number of packets read in is $incount\n";
print "Number of packets written out is $outcount\n";

if ($outcount < $incount) {
  print "Number of records written out less than read in.\n";
  print "There may be a problem with the input file or heaven forbid - this program\n";
}

# If any bogus arp entries have been made, delete entries

if ($mac_n > 0) {
  &arp_del_entries;
}

sub read_pkts {

Net::Pcap::loop($pcap_t, -1, \&process_pkt, "");

}

#  Process the packets read from the user-supplied pcap file
#  Decode the packet's IP header and determine the protocol
#  found in the packet (tcp, udp, or icmp). Decode the protocol
#  and determine if the data payload is the minimum length in
#  order to fragment it according to Paxson's model. If it is
#  not long enough, write it back to the network unchanged.
#  Otherwise, send it to the fragmentation process.

#  NetPacket supplies an IP encode routine along with the IP
#  decode routine. However, there is a bug when it recomputes
#  the IP datagram length when a packet is received, decoded,
#  and encoded if the length is less than 46 bytes. The original
#  packet has the correct datagram length, but if there is zero
#  padding because the minimum length of an IP datagram over
#  Ethernet is 46, the encoding miscalculates the datagram length
#  and includes the padding. My routine leaves the header length,
#  but for some reason, the padding is no longer included in the
#  packet. This should not affect generating a pcap for use with
```

 ©Sourcefire, Inc. All rights reserved. January 21, 2005

```
#  Snort, but this datagram may have problems being sent to an
#  actual destination. This is not the anticipated use for this
#  program.

sub process_pkt
{

  my ($args,$header,$packet,$i,$l) = @_;
  $incount++;
  $ip = NetPacket::IP->decode(eth_strip($packet));

  &arp_entry($ip->{src_ip});
  &arp_entry($ip->{dest_ip});

  if ($ip->{proto} == 6) {
    $ip_proto = "tcp";
    $tcp = NetPacket::TCP->decode($ip->{data});
    $len = length($tcp->{data});
    if ($len <= $min_frag_length) {
       $ip_pkt = &encode_IP;
       $outcount++;
       Net::RawSock::write_ip($ip_pkt);
    }
    else {
      $frag_flag = 1;
      $outcount++;
      &frag_it;
    }
  }
  else {
    if ($ip->{proto} == 17) {
      $ip_proto = "udp";
      $udp = NetPacket::UDP->decode($ip->{data});
      $len = length($udp->{data});
      if ($len <= $min_frag_length) {
         $ip_pkt = &encode_IP;
         $outcount++;
         Net::RawSock::write_ip($ip_pkt);
      }
      else {
         $frag_flag = 1;
         $outcount++;
         &frag_it;
      }
    }
    else {
      if ($ip->{proto} == 1) {
        $ip_proto = "icmp";
        $icmp = NetPacket::ICMP->decode($ip->{data});
        $len = length($icmp->{data});
        if ($len <= $min_frag_length) {
          $ip_pkt = &encode_IP;
          $outcount++;
          Net::RawSock::write_ip($ip_pkt);
        }
        else {
          $frag_flag = 1;
          $outcount++;
          &frag_it;
        }
      }
      else {
        print "Unsupported IP protocol found in packet $ip->{proto}.\n";
      }
    }
  }
}
```


©Sourcefire, Inc. All rights reserved. January 21, 2005

```
# Fragment the packet.

sub frag_it {

#  Need to compute the length of the protocol header so that accurate fragment
#  offsets can be used later.

#  Also, TCP can possibly present a problem since the header length may
#  not be evenly divisible by 8 (the size of a fragment). This messes
#  up the generic (for all protocols) fragment overlap calculations. The
#  solution to this is to add a 4-byte EOL (essentially a NOP) TCP option
#  at the end of the TCP header to pad the TCP header and make the length
#  evenly divisible by 8. This requires recalculating the TCP checksum
#  since the original packet TCP checksum will no longer be valid with
#  the addition of the TCP option.

  if ($ip->{proto} == 17) {
    $protohdr_len = 8;
    $all_pay = $udp->{data};
  }
  else {
    if ($ip->{proto} == 6) {
      $protohdr_len = $tcp->{hlen} * 4;
      if (($protohdr_len % 8) == 4) {
        $protohdr_len = $protohdr_len + 4;
        $tcp->{hlen} = $tcp->{hlen} + 1;
        $tcp->{options} = $tcp->{options} . "\x00\x00\x00\x00";
      }
      $all_pay = $tcp->{data};
    }
    else {
      if ($ip->{proto} == 1) {
        $protohdr_len = $ip->{len} - ($ip->{hlen} * 4) - length($icmp->{data});
        if ($icmp->{type} == 8) {
          $protohdr_len = $protohdr_len + 4;
          $all_pay = substr($icmp->{data}, 4);
        }
        else {
          $all_pay = $icmp->{data};
        }
      }
    }
  }

# Set a "filler" fragment that is used for fragment content when a totally
# overlapped/replaced fragment is required in the payload. While actual
# payload content can be used, this is content is a more distinct placeholder.

  $F = "\xff\xff\xff\xff\xff\xff\xff\xff";

#  Each policy must fragment the payload according to Paxson's model. This
#  requires extracting the correct number of bytes from the correct offset
#  in the payload and using filler fragments where they are replaced.

  if ($policy eq "bsd") {
    $payload1 = substr($all_pay, 0, 24);
    $payload2 = $F . substr($all_pay, 40, 8);
    $payload3 = substr($all_pay, 48, 24);
    $payload4 = $F . $F . substr($all_pay, 24, 16);
    $payload5 = $F . $F . $F;
    $payload6 = substr($all_pay, 72);
  }
  else {
    if ($policy eq "bsdr") {
      $payload1 = substr($all_pay, 0, 8) . $F . $F;
      $payload2 = substr($all_pay, 32, 16);
      $payload3 = $F . $F . $F;
      $payload4 = substr($all_pay, 8, 24) . $F;
      $payload5 = substr($all_pay, 48, 24);
```

 ©Sourcefire, Inc. All rights reserved. January 21, 2005

```
      $payload6 = substr($all_pay, 72);
    }
    else {
      if ($policy eq "linux") {
        $payload1 = substr($all_pay, 0, 24);
        $payload2 = $F . substr($all_pay, 40, 8);
        $payload3 = $F . $F . $F;
        $payload4 = $F . $F . substr($all_pay, 24, 16);
        $payload5 = substr($all_pay, 48, 24);
        $payload6 = substr($all_pay, 72);
      }
      else {
        if ($policy eq "first") {
          $payload1 = substr($all_pay, 0, 24);
          $payload2 = substr($all_pay, 32, 16);
          $payload3 = substr($all_pay, 48, 24);
          $payload4 = $F . $F . substr($all_pay, 24, 8) . $F;
          $payload5 = $F . $F . $F;
          $payload6 = substr($all_pay, 72);
        }
        else {
          if ($policy eq "last") {
            $payload1 = substr($all_pay, 0, 8) . $F . $F;
            $payload2 = $F . substr($all_pay, 40, 8);
            $payload3 = $F . $F . $F;
            $payload4 = substr($all_pay, 8, 32);
            $payload5 = substr($all_pay, 48, 24);
            $payload6 = substr($all_pay, 72);
          }
        }
      }
    }
  }

# Create the 6 sets of fragments in Paxson's model with the
# appropriate fragment offset, MF flag, and determine whether
# or not this is the first fragment.

# Create fragment 1

  $off   = 0;
  $flgs  = 1;
  $first_frag = 1;

  &create_packet($off, $flgs, $payload1, $first_frag);

# Create fragment 2

  $off   = $protohdr_len + 32;
  $flgs  = 1;
  $first_frag = 0;

  &create_packet($off, $flgs, $payload2, $first_frag);

# Create fragment 3

  $off   = $protohdr_len + 48;
  $flgs  = 1;
  $first_frag = 0;

  &create_packet($off, $flgs, $payload3, $first_frag);

# Create fragment 4

  $off   = $protohdr_len + 8;
  $flgs  = 1;
  $first_frag = 0;

  &create_packet($off, $flgs, $payload4, $first_frag);
```

 ©Sourcefire, Inc. All rights reserved. January 21, 2005

```
# Create fragment 5

  $off   = $protohdr_len + 48;
  $flgs  = 1;
  $first_frag = 0;

  &create_packet($off, $flgs, $payload5, $first_frag);

# Create fragment 6

  $off   = $protohdr_len + 72;
  $flgs  = 0;
  $first_frag = 0;

  &create_packet($off, $flgs, $payload6, $first_frag);
}

sub create_packet {

  $off =  $_[0];
  $flgs = $_[1];
  $pay  = $_[2];
  $ff   = $_[3];

## Create IP

# Format the IP header. Code take from NetPacket, but wanted
# more control than NetPacket offered - especially since it
# had errors in the code the created the fragmentation flags in
# IP header.

  $temp = $ip->{hlen} & 0x0f;
  $ver_len = $temp | (($ip->{ver} << 4) & 0xf0);

# The IP datagram length for the first fragment will include the
# payload header length as well as the IP header and payload lengths.
# Subsequent fragments do not have a protocol header - just data.

  if ($ff) {
    $len = $ip->{hlen} * 4 + $protohdr_len + length($pay);
  }
  else  {
    $len = $ip->{hlen} * 4 + length($pay);
  }

  $offset = $flgs << 13;
  $offset = $offset | (($off >> 3) & 0x1fff);
  $cksum = 0;
  $src_ip = gethostbyname($ip->{src_ip});
  $dest_ip = gethostbyname($ip->{dest_ip});

  $pack_ip = pack(c,$ver_len) . pack(c,$ip->{tos}) . pack(n,$len) . pack(n,$ip->{id}) . pack(n,
$offset) .
        pack(c,$ip->{ttl}) . pack(c,$ip->{proto}) . pack(n, $cksum) .
        pack(a4a4, $src_ip, $dest_ip);

  if (($ip->{hlen} > 5) and ($ip->{options} ne "")) {
    $pack_ip = $pack_ip . pack('a*', $ip->{options});
  }

  ## Create protocol part of packets

  ## Create UDP

  if ($ip_proto eq "udp") {

    if ($ff) {
      $len       = length($all_pay) + 8;
```

 ©Sourcefire, Inc. All rights reserved. January 21, 2005

```
      $pack_udp = pack("nnnna*", $udp->{src_port}, $udp->{dest_port},
                 $len, $udp->{cksum}, $pay);
    }
    else {
      $pack_udp = pack("a*", $pay);
    }

    my $pkt = $pack_ip . $pack_udp;
    Net::RawSock::write_ip($pkt);
  }
  else  {
    ## Create TCP

    if ($ip_proto eq "tcp") {

      if ($ff) {
        $len       = length($all_pay) + $protohdr_len;

        $tmp = $tcp->{hlen} << 12;
        $tmp = $tmp | (0x0f00 & ($tcp->{reserved} << 8));
        $tmp = $tmp | (0x00ff & $tcp->{flags});

        $cksum     = &tcp_checksum;

        $pack_tcp = pack('nnNNnnnna*a*',
             $tcp->{src_port}, $tcp->{dest_port}, $tcp->{seqnum},
             $tcp->{acknum}, $tmp, $tcp->{winsize}, $cksum,
             $tcp->{urg}, $tcp->{options}, $pay);
      }
      else {
        $pack_tcp = pack("a*", $pay);
      }
      my $pkt = $pack_ip . $pack_tcp;
      Net::RawSock::write_ip($pkt);
    }
    else  {
      ## Create ICMP

      if ($ip_proto eq "icmp") {

        if ($ff) {
          $len       = length($all_pay) + $protohdr_len;
          if ($icmp->{type} != 8) {
            $icmp = pack("ccna*", $icmp->{type}, $icmp->{code}, $icmp->{cksum}, $pay);
          }
          else {
            $type8_hdr  = substr($icmp->{data}, 0, 4);
            $pay = $type8_hdr . $pay;
            $icmp = pack("ccna*", $icmp->{type}, $icmp->{code}, $icmp->{cksum}, $pay);
          }
        }
        else {
          $icmp = pack("a*", $pay);
        }
        my $pkt = $pack_ip . $icmp;
        Net::RawSock::write_ip($pkt);
      }
    }
  }
}

sub tcp_checksum {

    $zero = 0;
    $proto = 6;
    $tcplen = ($tcp->{hlen} * 4) + length($tcp->{data});

    $tmp = $tcp->{hlen} << 12;
```

 ©Sourcefire, Inc. All rights reserved. January 21, 2005

```
    $tmp = $tmp | (0x0f00 & ($tcp->{reserved} << 8));
    $tmp = $tmp | (0x00ff & $tcp->{flags});

    # Pack pseudo-header for tcp checksum

    $src_ip = gethostbyname($ip->{src_ip});
    $dest_ip = gethostbyname($ip->{dest_ip});

    $packet = pack('a4a4nnnnNNnnnna*a*',
            $src_ip, $dest_ip, $proto, $tcplen,
            $tcp->{src_port}, $tcp->{dest_port}, $tcp->{seqnum},
            $tcp->{acknum}, $tmp, $tcp->{winsize}, $zero,
            $tcp->{urg}, $tcp->{options}, $tcp->{data});

    $cksum = NetPacket::htons(NetPacket::in_cksum($packet));
    return($cksum);
}

sub encode_IP {

    # create a zero variable
    $zero = 0;

    # adjust the length of the packet
    # Next statement commented out because it hoses the length
    # when padding for 46 byte minimum limit

#    $self->{len} = ($self->{hlen} * 4) + length($self->{data});

    $tmp = $ip->{hlen} & 0x0f;
    $tmp = $tmp | (($ip->{ver} << 4) & 0xf0);

    $offset = $ip->{flags} << 13;
    $offset = $offset | (($ip->{foffset} >> 3) & 0x1fff);

    # convert the src and dst ip
    $src_ip = gethostbyname($ip->{src_ip});
    $dest_ip = gethostbyname($ip->{dest_ip});

    # construct header to calculate the checksum
    $hdr = pack('CCnnnCCna4a4a*', $tmp, $ip->{tos},$ip->{len},
         $ip->{id}, $offset, $ip->{ttl}, $ip->{proto},
         $zero, $src_ip, $dest_ip, $ip->{options});

    $ip->{cksum} = NetPacket::htons(NetPacket::in_cksum($hdr));

    # make the entire packet
    $packet = pack('CCnnnCCna4a4a*a*', $tmp, $ip->{tos}, $ip->{len},
         $ip->{id}, $offset, $ip->{ttl}, $ip->{proto},
         $ip->{cksum}, $src_ip, $dest_ip, $ip->{options},
         $ip->{data});

    return($packet);

}

sub arp_cache {

  $n = 0;

  open(ARP, "arp -an|");

  while($arp = <ARP>) {
    chop($arp);
    @arp_rec = split(" ",$arp);
    $pos = index($arp_rec[3], ":");
    if ($pos > -1) {
      $arp_rec[1] =~ s/\(//;
      $arp_rec[1] =~ s/\)//;
```

 ©Sourcefire, Inc. All rights reserved. January 21, 2005

```
      $IP[$n] = $arp_rec[1];
      $n++;
    }
  }
}

sub arp_entry {

  $search_IP = $_[0];
  $max = $#IP;

  for ($i = 0; $i <= $max; $i++) {
    if ($IP[$i] eq $search_IP){
     return;
    }
  }

  for ($i = 0; $i < $mac_n; $i++) {
    if ($fake_arp[$i] eq $search_IP){
     return;
    }
  }

  if (length($mac_n) >= 2) {
    $mac = $mac_beg . $mac_n;
  }
  else {
    $mac = $mac_beg . "0" . $mac_n;
  }
  $fake_arp[$mac_n] = $search_IP;
  $cmd =`arp -s $search_IP $mac temp`;
  $mac_n++;
}

sub arp_del_entries {

  for ($i = 0; $i < $mac_n; $i++) {
    $cmd = `arp -d $fake_arp[$i]`;
  }
}
```

 ©Sourcefire, Inc. All rights reserved. January 21, 2005

Appendix D – Sample output from tcpdump from ICMP echo request/reply with BSD policy

```
13:21:58.646304 10.4.11.45 > 10.4.12.16: icmp: echo request (frag 7453:32@0+)
0x0000   4500 0034 1d1d 2000 5a01 f867 0a04 0b2d     E..4....Z..g...-
0x0010   0a04 0c10 0800 161e 3030 3030 3131 3232     ........00001122
0x0020   3333 3434 3131 3232 3333 3434 3131 3232     3344112233441122
0x0030   3333 3434                                   3344

13:21:58.649060 10.4.11.45 > 10.4.12.16: (frag 7453:16@40+)
0x0000   4500 0024 1d1d 2005 5a01 f872 0a04 0b2d     E..$....Z..r...-
0x0010   0a04 0c10 3232 3131 3333 3434 3232 3131     ....221133442211
0x0020   3333 3434                                   3344

13:21:58.650301 10.4.11.45 > 10.4.12.16: (frag 7453:24@56+)
0x0000   4500 002c 1d1d 2007 5a01 f868 0a04 0b2d     E..,....Z..h...-
0x0010   0a04 0c10 3333 3131 3232 3434 3333 3131     ....331122443311
0x0020   3232 3434 3333 3131 3232 3434               224433112244

13:21:58.651765 10.4.11.45 > 10.4.12.16: (frag 7453:32@16+)
0x0000   4500 0034 1d1d 2002 5a01 f865 0a04 0b2d     E..4....Z..e...-
0x0010   0a04 0c10 3434 3131 3232 3333 3434 3131     ....441122334411
0x0020   3232 3333 3434 3131 3232 3333 3434 3131     2233441122334411
0x0030   3232 3333                                   2233

13:21:58.652644 10.4.11.45 > 10.4.12.16: (frag 7453:24@56+)
0x0000   4500 002c 1d1d 2007 5a01 f868 0a04 0b2d     E..,....Z..h...-
0x0010   0a04 0c10 3131 3333 3232 3434 3131 3333     ....113322441133
0x0020   3232 3434 3131 3333 3232 3434               224411332244

13:21:58.653737 10.4.11.45 > 10.4.12.16: (frag 7453:24@80)
0x0000   4500 002c 1d1d 000a 5a01 1866 0a04 0b2d     E..,....Z..f...-
0x0010   0a04 0c10 3131 3434 3232 3333 3131 3434     ....114422331144
0x0020   3232 3333 3131 3434 3232 3333               223311442233

13:21:58.654003 10.4.12.16 > 10.4.11.45: icmp: echo reply
0x0000   4500 007c 7182 0000 4001 ddba 0a04 0c10     E..|q...@.......
0x0010   0a04 0b2d 0000 1e1e 3030 3030 3131 3232     ...-....00001122
0x0020   3333 3434 3131 3232 3333 3434 3131 3232     3344112233441122
0x0030   3333 3434 3434 3131 3232 3333 3434 3131     3344441122334411
0x0040   3232 3333 3232 3131 3333 3434 3333 3131     2233221133443311
0x0050   3232 3434 3333 3131 3232 3434 3333 3131     2244331122443311
0x0060   3232 3434 3131 3434 3232 3333 3131 3434     2244114422331144
0x0070   3232 3333 3131 3434 3232 3333               223311442233
```


©Sourcefire, Inc. All rights reserved. January 21, 2005

Appendix E - Patch to IP.pm to write calculated fragment flags and offset bytes

```
--- IP.pm      2003-05-21 09:16:40.000000000 -0400
+++ IP.good.pm 2004-11-30 14:15:16.000000000 -0500
@@ -217,14 +217,14 @@

     # construct header to calculate the checksum
     $hdr = pack('CCnnnCCna4a4a*', $tmp, $self->{tos},$self->{len},
-          $self->{id}, $self->{offset}, $self->{ttl}, $self->{proto},
+          $self->{id}, $offset, $self->{ttl}, $self->{proto},
          $zero, $src_ip, $dest_ip, $self->{options});

     $self->{cksum} = NetPacket::htons(NetPacket::in_cksum($hdr));

     # make the entire packet
     $packet = pack('CCnnnCCna4a4a*a*', $tmp, $self->{tos},$self->{len},
-          $self->{id}, $self->{foffset}, $self->{ttl}, $self->{proto},
+          $self->{id}, $offset, $self->{ttl}, $self->{proto},
          $self->{cksum}, $src_ip, $dest_ip, $self->{options},
          $self->{data});
```

 ©Sourcefire, Inc. All rights reserved. January 21, 2005

Appendix E - Code to generate multiple last fragments (uses scapy)

```
#-----------------------------------------------------------------
#
# This script sends an ICMP echo request with a payload containing
# fragments with unique content.  The fragments are sent in
# varying order and fragments 2 and 4 both have a MF value of 0,
# indicating that they are both the last fragment.
#
# Since it is not predictable which fragments the receiving host
# will accept/reassemble, the ICMP checksum becomes an issue.
# Ideally, there would be some way to have one checksum that
# remains the same regardless of the number of or exact
# fragments accepted.  Instead, every ICMP echo request is sent 4
# different times each with a different checksum to accomodate a
# receiving host acccepting 1,2,3, or all 4 fragments.  As with
# other fragment tests, each fragment has the same checksum.  This
# allows us to ignore, for checksum sake, the exact fragments
# that the receiving host accepts.
#
# To run this program:
#   1) Call this program supplying it the destination IP and source IP:
#      python multilastfrag.py "10.4.11.73" "10.4.12.23"
#
# Note:  This is the revised version where the ICMP echo request
#        and first fragment are in offset 0 (not just the ICMP
#        echo request).
#
# To analyze output, run tcpdump before starting the program
# with the destination IP and icmp filter and write to a file.
# Then dump the file and analyze the output:
#
# tcpdump -s0 -w /tmp/mulilastfrag.cisco.pcap 'host 10.4.11.73 and icmp'
#
# tcpdump -r /tmp/multilastfrag.cisco.pcap -nnX > /tmp/out
#
# J. Novak - 8/11/05
#
#-----------------------------------------------------------------
#!/usr/bin/python


def frag(dip, tnum):


  P1 = "AABBCCDD"
  P2 = "BBCCDDAA"
  P3 = "CCDDAABB"
  P4 = "DDAABBCC"

  p1_offset = 0
  p2_offset = 2
  p3_offset = 3
```

 ©Sourcefire, Inc. All rights reserved. January 21, 2005

```
p4_offset = 4

flag1 = 1
flag2 = 0
flag3 = 1
flag4 = 0

x = 0

while (x < 4):
    id = random.randint(1,65535)
    ip=IP(src=sip, dst=dip, id=id, flags=1, frag=0)

    if (x == 0):
        icmp=ICMP(type=8, code=0, chksum=0xcbd3)
    elif (x == 1):
        icmp=ICMP(type=8, code=0, chksum=0xd6de)
    elif (x == 2):
        icmp=ICMP(type=8, code=0, chksum=0xe1e9)
    else:
        icmp=ICMP(type=8, code=0, chksum=0xecf4)

    ip.proto=1
    ip.frag=p1_offset
    ip.flags = flag1
    ip.id = id
    f1=ip/icmp/P1

    ip.proto=1
    ip.frag=p2_offset
    ip.flags = flag2
    ip.id = id
    f2=ip/P2

    ip.proto=1
    ip.frag=p3_offset
    ip.flags = flag3
    ip.id = id
    f3=ip/P3

    ip.proto=1
    ip.frag=p4_offset
    ip.flags = flag4
    ip.id = id
    f4=ip/P4


    if (tnum == 1):
        send(f1)
        send(f2)
        send(f3)
        send(f4)
    elif (tnum == 2):
        send(f2)
        send(f4)
```


©Sourcefire, Inc. All rights reserved. January 21, 2005

```
            send(f1)
            send(f3)
        elif (tnum == 3):
            send(f4)
            send(f2)
            send(f1)
            send(f3)
        elif (tnum == 4):
            send(f4)
            send(f2)
            send(f1)
        elif (tnum == 5):
            send(f3)
            send(f2)
            send(f4)
            send(f1)
        elif (tnum == 6):
            send(f1)
            send(f3)
            send(f4)
            send(f2)
        elif (tnum == 7):
            send(f1)
            send(f3)
            send(f2)
            send(f4)

        x = x + 1

import random
import time
import sys
from scapy import  IP, TCP, ICMP, Ether, Raw, send, sniff, sr1
dip = sys.argv[1]
sip = sys.argv[2]

frag(dip, 1)
frag(dip, 2)
frag(dip, 3)
frag(dip, 4)
frag(dip, 5)
frag(dip, 6)
frag(dip, 7)
```

 ©Sourcefire, Inc. All rights reserved. January 21, 2005

Appendix F - Code to implement extended Sturges/Novak Model (uses scapy)

```
#-----------------------------------------------------------------------------
#
# This script creates an ICMP echo request with a payload consisting of 9
# sets of fragments with different overlap patterns.  This is similar to
# the Paxson/Shankar model, but as Steve discovered, a wholly overlapped
# first fragment (preceded by a gap between previous fragment) will be
# superseded by an overlapping fragment for Windows hosts. Additionally
# there is a fragment 3.7 that is overlapped by fragment 7 that begins
# at an offset 8 bytes less than 3.7 and overlaps it for 16 bytes. And
# the former BSD-right policy changes because of this model.
#
# To run this program:
#  2) Run this program supplying it the destination and source IP's:
#     python: python newmodel.py "10.4.11.73" "10.4.12.23"
#
# Note: This is a revised version where the ICMP echo request header
#       and the first fragment offset are in the 0-offset fragment.
#
# Before running this, have tcpdump ready to collect the results to be
# analyzed by another script readnewmodel.py.  For instance, if the
# target destination IP is 10.4.11.73:
#
#   tcpdump -s0 -w /tmp/reply.pcap 'icmp[0] = 0 and host 10.4.11.73'
#
# This will collect all ICMP echo request and write them to file
# /tmp/reply.pcap.  This will be input to the program readnewmodel.py.
#
# Note that if the target host is Solaris, you must change the MF on
# fragment 3.7 to be 0.  All other OS's are okay with MF = 1 on this
# and fragment 7.
#
# J.Novak - 8/11/05
#
#-----------------------------------------------------------------------------
#!/usr/bin/python

def frag(dip):

    P1   = "AABBCCDD"
    P2   = "BBCCDDAA"
    P3   = "CCDDAABB"
    P3_1 = "DDAABBCC"
    P3_2 = "AACCBBDD"
    P3_3 = "CCBBAADD"
    P3_4 = "CCBBDDAA"
    P3_5 = "DDBBCCAA"
    P3_6 = "DDCCAABB"
    P4   = "AADDBBCC"
    P5   = "BBAACCDD"
    P6   = "BBDDAACC"
```

 ©Sourcefire, Inc. All rights reserved. January 21, 2005

```
P7   = "CCAABBDD"
P8   = "DDAABBCC"
P9   = "DDBBAACC"
P10  = "AABBDDCC"
P11  = "AACCDDBB"

ip=IP(src=sip, dst=dip, flags=1, frag=0)
icmp=ICMP(type=8, code=0, chksum=0xeef6)

p1_offset = 0
p2_offset = 5
p3_offset = 7
p3_1_offset = 11
p3_2_offset = 14
p3_3_offset = 16
p3_4_offset = 19
p3_5_offset = 21
p3_6_offset = 23
p4_offset = 2
p5_offset = 7
p6_offset = 10
p7_offset = 13
p8_offset = 17
p9_offset = 20
p10_offset = 21
p11_offset = 23

frag1 = P1 * 3
frag2 = P2 * 2
frag3 = P3 * 3
frag3_1 = P3_1
frag3_2 = P3_2 * 2
frag3_3 = P3_3 * 3
frag3_4 = P3_4 * 2
frag3_5 = P3_5 * 2
frag3_6 = P3_6
frag4 = P4 * 4
frag5 = P5 * 3
frag6 = P6 * 3
frag7 = P7 * 3
frag8 = P8
frag9 = P9
frag10 = P10
frag11 = P11 * 2

ip.proto=1
ip.frag=p1_offset
ip.flags = 1
f1=ip/icmp/frag1

ip.proto=1
ip.frag=p2_offset
ip.flags = 1
f2=ip/frag2
```

 ©Sourcefire, Inc. All rights reserved. January 21, 2005

```
ip.proto=1
ip.frag=p3_offset
ip.flags = 1
f3=ip/frag3

ip.proto=1
ip.frag=p3_1_offset
ip.flags = 1
f3_1=ip/frag3_1

ip.proto=1
ip.frag=p3_2_offset
ip.flags = 1
f3_2=ip/frag3_2

ip.proto=1
ip.frag=p3_3_offset
ip.flags = 1
f3_3=ip/frag3_3

ip.proto=1
ip.frag=p3_4_offset
ip.flags = 1
f3_4=ip/frag3_4

ip.proto=1
ip.frag=p3_5_offset
ip.flags = 1
f3_5=ip/frag3_5

ip.proto=1
ip.frag=p3_6_offset
ip.flags = 1 #Must be set to 0 for Solaris, other OS's can be 0 or 1
f3_6=ip/frag3_6

ip.proto=1
ip.frag=p4_offset
ip.flags = 1
f4=ip/frag4

ip.proto=1
ip.frag=p5_offset
ip.flags = 1
f5=ip/frag5

ip.proto=1
ip.frag=p6_offset
ip.flags = 1
f6=ip/frag6

ip.proto=1
ip.frag=p7_offset
ip.flags = 1
f7=ip/frag7
```


©Sourcefire, Inc. All rights reserved. January 21, 2005

```
    ip.proto=1
    ip.frag=p8_offset
    ip.flags = 1
    f8=ip/frag8

    ip.proto=1
    ip.frag=p9_offset
    ip.flags = 1
    f9=ip/frag9

    ip.proto=1
    ip.frag=p10_offset
    ip.flags = 1
    f10=ip/frag10

    ip.proto=1
    ip.frag=p11_offset
    ip.flags = 0
    f11=ip/frag11

    send(f1)
    send(f2)
    send(f3)
    send(f3_1)
    send(f3_2)
    send(f3_3)
    send(f3_4)
    send(f3_5)
    send(f3_6)
    send(f4)
    send(f5)
    send(f6)
    send(f7)
    send(f8)
    send(f9)
    send(f10)
    send(f11)

import random
import time
import sys
from scapy import  IP, TCP, ICMP, Ether, Raw, send, sniff, sr1

dip = sys.argv[1]
sip = sys.argv[2]


frag(dip)
```


©Sourcefire, Inc. All rights reserved. January 21, 2005

Appendix G - Code to read ICMP echo reply from newmodel.py extended Sturges/Novak Model (uses scapy)

```
#-----------------------------------------------------------
#
# This program analyzes the ICMP echo reply payloads from
# running newmodel.py.  It will print how the ICMP echo
# reply reassembled the overlapping fragments.  Input to
# this program is the file /tmp/reply.pcap.
#
# J. Novak - 8/11/05
#
#-----------------------------------------------------------
#!/usr/bin/python

def readit(filein):

  import sys
  from scapy import rdpcap, IP, TCP, Ether, Raw, ICMP
  records = rdpcap(filein)
  l = len(records)
  x=0

  P1      = "AABBCCDD"
  P2      = "BBCCDDAA"
  P3      = "CCDDAABB"
  P3_1    = "DDAABBCC"
  P3_2    = "AACCBBDD"
  P3_3    = "CCBBAADD"
  P3_4    = "CCBBDDAA"
  P3_5    = "DDBBCCAA"
  P3_6    = "DDCCAABB"
  P4      = "AADDBBCC"
  P5      = "BBAACCDD"
  P6      = "BBDDAACC"
  P7      = "CCAABBDD"
  P8      = "DDAABBCC"
  P9      = "DDBBAACC"
  P10     = "AABBDDCC"
  P11     = "AACCDDBB"


  while (x < l):
      if (records[x].haslayer(IP)):
          i=records[x].getlayer(IP)
          if (i.haslayer(ICMP)):
              icmp=i.getlayer(ICMP)
              if (icmp.haslayer(Raw)):
                  p=icmp.getlayer(Raw)
                  pay = str(p)
                  offset = 0
                  fstring = ""
```

 ©Sourcefire, Inc. All rights reserved. January 21, 2005

```
                while (offset <= 192):
                    frag = pay[offset:offset + 8]
                    if (frag == P1):
                        fstring = fstring + "<P1>"
                    elif (frag == P2):
                        fstring = fstring + "<P2>"
                    elif (frag == P3):
                        fstring = fstring + "<P3>"
                    elif (frag == P3_1):
                        fstring = fstring + "<P3_1>"
                    elif (frag == P3_2):
                        fstring = fstring + "<P3_2>"
                    elif (frag == P3_3):
                        fstring = fstring + "<P3_3>"
                    elif (frag == P3_4):
                        fstring = fstring + "<P3_4>"
                    elif (frag == P3_5):
                        fstring = fstring + "<P3_5>"
                    elif (frag == P3_6):
                        fstring = fstring + "<P3_6>"
                    elif (frag == P4):
                        fstring = fstring + "<P4>"
                    elif (frag == P5):
                        fstring = fstring + "<P5>"
                    elif (frag == P6):
                        fstring = fstring + "<P6>"
                    elif (frag == P7):
                        fstring = fstring + "<P7>"
                    elif (frag == P8):
                        fstring = fstring + "<P8>"
                    elif (frag == P9):
                        fstring = fstring + "<P9>"
                    elif (frag == P10):
                        fstring = fstring + "<P10>"
                    elif (frag == P11):
                        fstring = fstring + "<P11>"

                    offset = offset + 8

                print "Test results: %s" % (fstring)
        x=x+1

readit("/tmp/reply.pcap")
```


©Sourcefire, Inc. All rights reserved. January 21, 2005