Global Information Assurance Certification Paper

GIAC Certified Intrusion Analyst (GCIA)
Practical Assignment
Version 3.4

Odis Richardson
Baltimore MD, 2003
March 24, 2004

Table of Contents :

Part 1 – The State of Intrusion Detection …..... 3
Analysis of Linux Kernel Packet FragmentationDefense..... 3
Part 2 – Network Detects 22
Detect2.1 22
BAD TRAFFIC bad frag bits 22
Detect 2.2 29
ShellCode x86 NOOP 29
Detect 2.3 35
RPC portmap request mountd..... 35
Part 3 - “ANALYZE THIS” 40

Part 1 – The State of Intrusion Detection
Introduction

This paper is focused on host system's defense for overlapping fragment attacks as it applies to the intrusion detection level. The major concentration of this attack is directed at the host level. Fragmentation is not a new problem, but still effective under the right circumstances. Reasons for using fragmentation are, to avoid intrusion detection filters, cause Denial of Service (DoS), and gather information for OS finger printing. This paper focuses on the host operating systems mechanism of defense against overlapping fragmentation attacks. Messages in the actual functions used to process fragmented packets are sent to the syslog for analysis. The messages are in real time as the host executes it's code for packet reassembly. This process can aid in protecting, preventing, and detecting against threats and attacks. This analysis of overlapping fragment attacks will show patterns, to better understand the fragmentation process on a host machine.

What is fragmentation? Fragmentation is the process allowing datagrams assembled as a packet to split into smaller packets for transmission from the source to the destination points. The packets are reassembled by the destination host and processed. The purpose of fragmentation is to insure that datagrams flow through all sorts of networks. In the IP header, fragmentation is identified by two fields. The fragment flag and offset information. The flag information is identified by bits, one bit for don't fragment and one bit for more fragments. The fragment offset field gives the position to place the fragment in the original datagram. The offset number is in units of 8 bytes expected by the host system.

The fragment attacks are meant to bypass routers, firewalls, and intrusion detection systems. Routers and firewalls are intermediate networks devices and are not meant to reassemble packets. Information in the IP header is used to determine where to direct the packet. This is the intent of routers and firewalls and will not identify the specific attack. Intrusion detection systems look further into the packet and analyze the IP header and data portion. The purpose is to match known attacks and threats with rules, filters, and or signatures. The intention of the fragment attack is to evade the intrusion detection system by splitting up the data so that the set rules are not matched. The intrusion detection system is able to identify overlapping fragment attacks, but does not protect or prevent them.

When datagrams are fragmented into packets the header information is kept in tack except for the fragment flags and offset, so that the host knows where to place the packet. In the reassembly stage the host creates a temporary packet with the fragmented datagrams using the the offsets for placement. Once the entire datagram is reassembled, the packet is processed by the host system. If overlapping occurs, the packet is not complete. This tests the strength of the fragmentation code of the host system to protect itself from this attack.

The purpose of generating messages in the fragment code is to

understand the reaction of the host, when reassembling overlapping fragment attacks. The operating system chosen for this task is Red Hat Linux 8.0 running kernel version vmlinuz-2.4.18-14, with source code available. The file:

/usr/src/linux-2.4.18-14/net/ipv4/ip_fragment.c

is the C code written to handle fragmented packets. After the initial review of the fragment code 'ip_fragment.c', messages are already sent to the kernel. The function used to send the message is 'printk()'. The next statement is an example taken from the fragment code, to send an error message to the kernel.

```
printk(KERN_ERR "ip_frag_create: no memory left !\n");
```

Further information on how to use the printk() function is in the manual for syslogd. The following is a list of message types to use for printk().

```
#define KERN_EMERG    "<0>"      /* system is unusable               */
#define KERN_ALERT    "<1>"      /* action must be taken immediately */
#define KERN_CRIT     "<2>"      /* critical conditions              */
#define KERN_ERR      "<3>"      /* error conditions                 */
#define KERN_WARNING  "<4>"  /* warning conditions               */
#define KERN_NOTICE   "<5>"      /* normal but significant condition */
#define KERN_INFO     "<6>"      /* informational                    */
#define KERN_DEBUG    "<7>"      /* debug□level messages */
```

I coded a series of printk() functions with my own message included. This messaging is added to each function in the fragment code to reveal a flow of logic. As the fragment code executed in real time, the messages will represent the function logic in words. An example of the printk() functions added is listed next:

```
printk(KERN_INFO "--- 1st function for ip_fragment.c\n");
```

A great deal of time and effort was spent on where to place comments in the source code. This was accomplished using the trial and error approach. This function taken from the fragment code is an example of the messages inserted and what they will produce.

```
static __inline__ unsigned int ipqhashfn(u16 id, u32 saddr, u32 daddr, u8 prot)
{
/*    ^=      bitwise exclusive OR (XOR).
*    >>      shift right. C ++ use of >>
*    ^       exclusive OR
*    &       Address of (Unary operator)
*/

                unsigned int h = saddr ^ daddr;
printk(KERN_INFO "\n");
printk(KERN_INFO "##3 :ipqhashfn(generate & return hash)\n");
                h ^= (h>>16)^id;
                h ^= (h>>8)^prot;
printk(KERN_INFO "hash : %i\n",h & (IPQ_HASHSZ - 1));
                return h & (IPQ_HASHSZ - 1);
}
```

These are the messages sent to the syslog generated by the printk functions.

```
Feb  5 19:03:32 localhost kernel:
Feb  5 19:03:32 localhost kernel: ##3 :ipqhashfn(generate & return hash)
Feb  5 19:03:32 localhost kernel: hash : 43
```

The printk functions sends the value of the calculated hash to the syslog for viewing. This is just one example of this technique to understand the actual processing of the fragmentation code. The entire file 'ip_fragment.c' with printk() messaging inserts are located in appendix A. After editing the fragment code with printk() functions, the file is too large to add to this paper. In order to execute the updated code, the the kernel must be re-compiled.

The Teardrop fragmentation exploit is explained in order to understand the combination of fragments sent to processed by the kernel. Teardrop is available at http:rootshell.com. The source code was compiled and executed on the attacker's box. The teardrop code sends 2 packets to the desired destination. The first packet is considered a normal packet. This packet includes a small amount of data with the fragment offset starting at 0. This causes no alarm. The second packet also has a small amount of data, but the fragment offset does not start after the data length of the first packet. The fragment offset over laps the first data portion by 12 bytes. This causes an error in the reassembly of the packet. Years ago this locked up Windows and Linux systems. The following is output from tcpdump version 3.7.2 of the teardrop packets to analyze the fragment offsets.

Command entered on the attacker's box to send teardrop exploit.

```
# teardop 10.0.0.2  10.0.0.10 1
```

Command on destination's box to show packet dump.

```
[root@localhost root]# tcpdump -vvv -X
tcpdump: listening on eth0
15:37:38.625204 10.0.0.2.65226 > 10.0.0.10.18461: [no cksum] udp 28 (frag 242:36@0+)
 (ttl 64, len 56)
0x0000   4500 0038 00f2 2000 4011 45b8 0a00 0002        E..8....@.E.....
0x0010   0a00 000a feca 481d 0024 0000 0000 0000        ......H..$......
0x0020   0000 0000 0000 0000 0000 0000 0000 0000        ................
0x0030   0000 0000 0000                                 ......
15:37:38.625323 10.0.0.2 > 10.0.0.10: udp (frag 242:4@24) (ttl 64, len 24)
0x0000   4500 0018 00f2 0003 4011 65d5 0a00 0002        E.......@.e.....
0x0010   0a00 000a feca 481d 0024 0000 0000 0000        ......H..$......
0x0020   0000 0000 0000 0000 0000 0000 0000            ..............
```

The fragment ID is 242 as indicated above. The first fragment has the data length of 36 bytes to place the fragment at offset 0. The fragment flags value is highlighted as well - 0x02. (.0.. - Don't frag bit, Off / ..1. - Fragments bit, On). The second packet also contains the proper ID of 242. The data length is only 4 bytes. The fragment flags indicate (.0.. - Don't frag bit, Off / ..0. - Fragments bit, Off). This is the last fragment. The fragment offset is 24, which is incorrect. This offset is fudged by the teardrop code. The 4 bytes of data will overlap the previous data starting by 12 bytes. This will cause an error in the reassembly code in the kernel.

The next fragment attack is Newbonk. To my surprise newbonk is still an effective Denial of Service (DoS) to the Linux operating system. As soon as newbonk starts, Linux suffers an DoS attack. The newbonk exploit program is available at http://www.rootshell.com. Newbonk sends two types of packets. The first is a harmless packet. The second packet has an offset which over laps the first packet, in which reassembly will never complete. As the kernel accepts the never ending stream of invalid packets for reassembly, this causes the DoS attack. The following is the attack command and tcpdump output to explain what newbonk is sending.

Attackers command: # newbonk 10.0.0.2 10.0.0.10
Tcpdump's Captured Packets : newbonk packets

```
13:35:43.479766 10.0.0.2.domain > 10.0.0.10.domain:
[no cksum]  0 [0q] (28) (frag 1109:36@0+) (ttl 255, len 56)
0x0000          4500 0038 0455 2000 ff11 8354 0a00 0002          E..8.U.....T....
0x0010          0a00 000a 0035 0035 0024 0000 0000 0000          .....5.5.$......
0x0020          0000 0000 0000 0000 0000 0000 0000 0000          ................
0x0030          0000 0000 0000                                   ......
13:35:43.479878 10.0.0.2 > 10.0.0.10: udp (frag 1109:4@8) (ttl 255, len 24)
0x0000          4500 0018 0455 0001 ff11 a373 0a00 0002          E....U.....s....
0x0010          0a00 000a 0035 0035 0024 0000 0000 0000          .....5.5.$......
0x0020          0000 0000 0000 0000 0000 0000                    ............
13:35:43.479993 10.0.0.2 > 10.0.0.10: udp (frag 1109:4@8) (ttl 255, len 24)
0x0000          4500 0018 0455 0001 ff11 a373 0a00 0002          E....U.....s....
0x0010          0a00 000a 0035 0035 0024 0000 0000 0000          .....5.5.$......
0x0020          0000 0000 0000 0000 0000 0000                    ............
13:35:43.480113 10.0.0.2 > 10.0.0.10: udp (frag 1109:4@8) (ttl 255, len 24)
0x0000          4500 0018 0455 0001 ff11 a373 0a00 0002          E....U.....s....
0x0010          0a00 000a 0035 0035 0024 0000 0000 0000          .....5.5.$......
0x0020          0000 0000 0000 0000 0000 0000                    ............
```

.
. packet repeated flooding kernel
.

The 36 bytes of data highlighted above in the first packet, starts at offset 0. The More Fragments bit (MF) is ON, with the fragment flag value of 0x02. This is OK. The second packet showing 4 bytes of data to start at offset 8 is incorrect. The fragment flag value is 0x00. No flags are set, so the kernel thinks it's the last packet. The second packet over writes the first packet by 24 bytes. To make matters worse, the second packet type never stops. It is sent continuously to the kernel. The kernel goes into a DoS to process this information. The newbonk is still effective, sending overlapping fragments.

The following section explains the control the kernel has to defend against overlapping fragment attacks. The messaging from teardrop and newbonk are real time as the the fragment code is executed. The teardrop messages are to the point and give a good view of the kernels processing. Newbonk messages show the vulnerability in the fragment code, the DoS attack. Newbonk identifies the problem explained, and the information to propose a solution. In this procedure the DoS attack is documented, providing information

to defend from such attacks. A number of messages are sent from the kernel to the syslog, which makes buffering an issue. The performance of the kernel is gaged with and without the added messages. Newer kernels may have a different reaction to the same attacks. Another kernel is tested, to find better results. Changes to the kernel are examined at the point where a loop appears in the fragment code, when receiving the newbonk attack. Firewalls and routers play a role in fragmentation as well, which can help this situation. An IDS level is determined for both the teardop and newbonk attack.

Teardrop

The following section involves the teardrop attack. The messages provided in real time trace of kernels reaction to teardrop fragments. The functions called are summarized and a listing of functions to follow the fragment codes logic. All messages are produces with the printk command, using the KERN_INFO parameter to send the message to the syslog. In order to gather this information from the syslog, the program 'tail' is used.

```
# tail -f -n 0 /var/log/messages > /home/teardrop.msg

*** Note Start of 1st Packet
Feb  5 19:03:32 localhost kernel: --- 1st function for ip_fragment.c
Feb  5 19:03:32 localhost kernel: ##17 ip_defrag( Process an incoming IP datagram fragment )
Feb  5 19:03:32 localhost kernel: Start, clean up memory
Feb  5 19:03:32 localhost kernel: sysctl_ipfrag_low_thresh : 196608
Feb  5 19:03:32 localhost kernel: sysctl_ipfrag_high_thresh : 262144
Feb  5 19:03:32 localhost kernel: Lookup or Create queue header
Feb  5 19:03:32 localhost kernel:
Feb  5 19:03:32 localhost kernel: ##3 :ipqhashfn(generate & return hash)
Feb  5 19:03:32 localhost kernel: hash : 43
Feb  5 19:03:32 localhost kernel:
Feb  5 19:03:32 localhost kernel: ##14 :ip_find( Find the correct entry in the incomplete
Feb  5 19:03:32 localhost kernel: ##                datagrams queue or this IP datagram,
Feb  5 19:03:32 localhost kernel: ##                and create new one, if nothing is found.
Feb  5 19:03:32 localhost kernel: hash : 43
Feb  5 19:03:32 localhost kernel: qp : 00000000
Feb  5 19:03:32 localhost kernel: loops if qp != 0
Feb  5 19:03:32 localhost kernel: for(qp = ipq_hash[hash]; qp; qp = qp->next)
Feb  5 19:03:32 localhost kernel: match not found, first fragment.
Feb  5 19:03:32 localhost kernel:
Feb  5 19:03:32 localhost kernel: ##13 :ip_frag_create( Add an entry to the 'ipq' queue for
Feb  5 19:03:32 localhost kernel: ##                    a newly received IP datagram )
Feb  5 19:03:32 localhost kernel:
Feb  5 19:03:32 localhost kernel: ##6 frag_alloc_queue()
Feb  5 19:03:32 localhost kernel: ##13 returned
Feb  5 19:03:32 localhost kernel: qp->protocol : 17
Feb  5 19:03:32 localhost kernel: Initialize a timer for this entry.
Feb  5 19:03:32 localhost kernel:
Feb  5 19:03:32 localhost kernel: ##12 ip_frag_intern()
Feb  5 19:03:32 localhost kernel: With SMP race we have to recheck hash table,
Feb  5 19:03:32 localhost kernel:   because such entry could be created on other
Feb  5 19:03:32 localhost kernel:   cpu, while we promoted read lock to write lock.
Feb  5 19:03:32 localhost kernel: check for CONFIG_SMP
Feb  5 19:03:32 localhost kernel: CONFIG_SMP is defined.
Feb  5 19:03:32 localhost kernel: hash : 43
Feb  5 19:03:32 localhost kernel: qp : 00000000
Feb  5 19:03:32 localhost kernel: loops if qp != 0
Feb  5 19:03:32 localhost kernel: for(qp = ipq_hash[hash]; qp; qp = qp->next)
```

```
Feb  5 19:03:32 localhost kernel: match not found, first fragment.
Feb  5 19:03:32 localhost kernel:
Feb  5 19:03:32 localhost kernel: ##15 :ip_frag_queue( Add new segment to existing queue.)
Feb  5 19:03:32 localhost kernel: qp->last_in : 0
Feb  5 19:03:32 localhost kernel: COMPLETE : 4
Feb  5 19:03:32 localhost kernel: (qp->last_in & COMPLETE) : 0
Feb  5 19:03:32 localhost kernel: if (qp->last_in & COMPLETE ) goto err
Feb  5 19:03:32 localhost kernel:
Feb  5 19:03:32 localhost kernel: flags : 2000
Feb  5 19:03:32 localhost kernel: Determine the position of this fragment.
Feb  5 19:03:32 localhost kernel: offset : 0
Feb  5 19:03:32 localhost kernel: skb->len : 56
Feb  5 19:03:32 localhost kernel: ihl : 20
Feb  5 19:03:32 localhost kernel: end = offset + skb->len - ihl;
Feb  5 19:03:32 localhost kernel: end = 36
Feb  5 19:03:32 localhost kernel:
Feb  5 19:03:32 localhost kernel: Is this the final fragment?
Feb  5 19:03:32 localhost kernel: flags : 2000
Feb  5 19:03:32 localhost kernel: IP_MF : 2000
Feb  5 19:03:32 localhost kernel: (flags & IP_MF) : 2000
Feb  5 19:03:32 localhost kernel: if ((flags & IP_MF ) == 0) process final fragment
Feb  5 19:03:32 localhost kernel: No, This is this the not the final fragment.
Feb  5 19:03:32 localhost kernel:
Feb  5 19:03:32 localhost kernel: end &= ~7;
Feb  5 19:03:32 localhost kernel: end : 32
Feb  5 19:03:32 localhost kernel: qp->len : 0
Feb  5 19:03:32 localhost kernel: 32 bits beyond end
Feb  5 19:03:32 localhost kernel: if (end > qp->len)
Feb  5 19:03:32 localhost kernel:
Feb  5 19:03:32 localhost kernel: Check --> Some bits beyond end = corruption
Feb  5 19:03:32 localhost kernel: qp->last_in : 0
Feb  5 19:03:32 localhost kernel: LAST_IN : 1
Feb  5 19:03:32 localhost kernel: (qp->last_in & LAST_IN) : 0
Feb  5 19:03:32 localhost kernel: if (qp->last_in & LAST_IN) goto err;
Feb  5 19:03:32 localhost kernel:
Feb  5 19:03:32 localhost kernel: end : 32
Feb  5 19:03:32 localhost kernel: offset : 0
Feb  5 19:03:32 localhost kernel: if (end == offset ) goto err
Feb  5 19:03:32 localhost kernel:
Feb  5 19:03:32 localhost kernel: if (pskb_pull(skb, ihl) == NULL) goto err;
Feb  5 19:03:32 localhost kernel: if (pskb_trim(skb, end-offset)) goto err
Feb  5 19:03:32 localhost kernel:
Feb  5 19:03:32 localhost kernel: Find out which fragments are in front and at the back
Feb  5 19:03:32 localhost kernel:   of us in the chain of fragments so far.
Feb  5 19:03:32 localhost kernel: We must know where to put this fragment, right?
Feb  5 19:03:32 localhost kernel:
Feb  5 19:03:32 localhost kernel: next : 00000000
Feb  5 19:03:32 localhost kernel: for(next = qp->fragments; next != NULL; next = next->next)
Feb  5 19:03:32 localhost kernel:
Feb  5 19:03:32 localhost kernel: We found where to put this one.
Feb  5 19:03:32 localhost kernel: Check for overlap with preceding fragment, and, if needed,
Feb  5 19:03:32 localhost kernel:   align things so that any overlaps are eliminated.
Feb  5 19:03:32 localhost kernel:
Feb  5 19:03:32 localhost kernel: check for preceding fragment
Feb  5 19:03:32 localhost kernel: if (prev)
Feb  5 19:03:32 localhost kernel:
Feb  5 19:03:32 localhost kernel: Insert this fragment in the chain of fragments.
Feb  5 19:03:32 localhost kernel: ##17 returned
Feb  5 19:03:32 localhost kernel: ? call ip frag reassembly
Feb  5 19:03:32 localhost kernel: No, Will Not call ip_frag_reasm();
Feb  5 19:03:32 localhost kernel:
Feb  5 19:03:32 localhost kernel: ##8 ipg_put()
Feb  5 19:03:32 localhost kernel:
*** Note Start of 2nd Packet
```

```
Feb  5 19:03:32 localhost kernel: --- 1st function for ip_fragment.c
Feb  5 19:03:32 localhost kernel: ##17 ip_defrag( Process an incoming IP datagram fragment )
Feb  5 19:03:32 localhost kernel: Start, clean up memory
Feb  5 19:03:32 localhost kernel: sysctl_ipfrag_low_thresh : 196608
Feb  5 19:03:32 localhost kernel: sysctl_ipfrag_high_thresh : 262144
Feb  5 19:03:32 localhost kernel: Lookup or Create queue header
Feb  5 19:03:32 localhost kernel:
Feb  5 19:03:32 localhost kernel: ##3 :ipqhashfn(generate & return hash)
Feb  5 19:03:32 localhost kernel: hash : 43
Feb  5 19:03:32 localhost kernel:
Feb  5 19:03:32 localhost kernel: ##14 :ip_find( Find the correct entry in the incomplete
Feb  5 19:03:32 localhost kernel: ##                datagrams queue or this IP datagram,
Feb  5 19:03:32 localhost kernel: ##                and create new one, if nothing is found.
Feb  5 19:03:32 localhost kernel: hash : 43
Feb  5 19:03:32 localhost kernel: qp : c8c84600
Feb  5 19:03:32 localhost kernel: loops if qp != 0
Feb  5 19:03:32 localhost kernel: for(qp = ipq_hash[hash]; qp; qp = qp->next)
Feb  5 19:03:32 localhost kernel: for loop qp : c8c84600
Feb  5 19:03:32 localhost kernel: compare id, saddr ,daddr, protocol for match
Feb  5 19:03:32 localhost kernel:   in the ipq_hash table for previous fragment
Feb  5 19:03:32 localhost kernel: match !, found existing fragment
Feb  5 19:03:32 localhost kernel:
Feb  5 19:03:32 localhost kernel: ##15 :ip_frag_queue( Add new segment to existing queue.)
Feb  5 19:03:32 localhost kernel: qp->last_in : 2
Feb  5 19:03:32 localhost kernel: COMPLETE : 4
Feb  5 19:03:32 localhost kernel: (qp->last_in & COMPLETE) : 0
Feb  5 19:03:32 localhost kernel: if (qp->last_in & COMPLETE ) goto err
Feb  5 19:03:32 localhost kernel:
Feb  5 19:03:32 localhost kernel: flags : 0
Feb  5 19:03:32 localhost kernel: Determine the position of this fragment.
Feb  5 19:03:32 localhost kernel: offset : 24
Feb  5 19:03:32 localhost kernel: skb->len : 24
Feb  5 19:03:32 localhost kernel: ihl : 20
Feb  5 19:03:32 localhost kernel: end = offset + skb->len - ihl;
Feb  5 19:03:32 localhost kernel: end = 28
Feb  5 19:03:32 localhost kernel:
Feb  5 19:03:32 localhost kernel: Is this the final fragment?
Feb  5 19:03:32 localhost kernel: flags : 0
Feb  5 19:03:32 localhost kernel: IP_MF : 2000
Feb  5 19:03:32 localhost kernel: (flags & IP_MF) : 0
Feb  5 19:03:32 localhost kernel: if ((flags & IP_MF ) == 0) process final fragment
Feb  5 19:03:33 localhost kernel: Yes, This is this the final fragment.
Feb  5 19:03:33 localhost kernel:
Feb  5 19:03:33 localhost kernel: If we already have some bits beyond end or have different end,
Feb  5 19:03:33 localhost kernel:   the segment is corrrupted.
Feb  5 19:03:33 localhost kernel: if (end < qp->len ||
Feb  5 19:03:33 localhost kernel:     ((qp->last_in & LAST_IN) && end != qp->len)) goto err;
Feb  5 19:03:33 localhost kernel: Instruction order:
Feb  5 19:03:33 localhost kernel: ( (qp->last_in & LAST_IN) : 0
Feb  5 19:03:33 localhost kernel:      &&
Feb  5 19:03:33 localhost kernel: end != qp->len : 28 != 32 )
Feb  5 19:03:33 localhost kernel:      OR
Feb  5 19:03:33 localhost kernel: end < qp->len : 28 < 32
Feb  5 19:03:33 localhost kernel:    goto err;
Feb  5 19:03:33 localhost kernel: ##17 returned
Feb  5 19:03:33 localhost kernel: ? call ip frag reassembly
Feb  5 19:03:33 localhost kernel: No, Will Not call ip_frag_reasm();
Feb  5 19:03:33 localhost kernel:
Feb  5 19:03:33 localhost kernel: ##8 ipg_put()
Feb  5 19:04:02 localhost kernel:
Feb  5 19:04:02 localhost kernel: ##11 :ip_expire(A fragment queue timed out.
Feb  5 19:04:02 localhost kernel:                   Kill it and send an ICMP reply.
Feb  5 19:04:02 localhost kernel:
Feb  5 19:04:02 localhost kernel: ##9 ip_q_kill(Kill ipq entry. It is not destroyed immediately,
```

```
Feb  5 19:04:02 localhost kernel:               because caller (and someone more)
Feb  5 19:04:02 localhost kernel:               holds reference count.)
Feb  5 19:04:02 localhost kernel:
Feb  5 19:04:02 localhost kernel: ##2 ipq_unlink(lock & unlock for __ipq_unlink()
Feb  5 19:04:02 localhost kernel:
Feb  5 19:04:02 localhost kernel: ##1 ipq_unlink(unlink ipq struct from linklist)
Feb  5 19:04:02 localhost kernel:
Feb  5 19:04:02 localhost kernel: Send an ICMP 'Fragment Reassembly Timeout' message.
Feb  5 19:04:02 localhost kernel: ICMP_TIME_EXCEEDED, ICMP_EXC_FRAGTIME
Feb  5 19:04:02 localhost kernel:
Feb  5 19:04:02 localhost kernel:
Feb  5 19:04:02 localhost kernel: ##8 ipg_put()
Feb  5 19:04:02 localhost kernel:
Feb  5 19:04:02 localhost kernel: ##7 ip_frag_destroy(complete destruction of fragments - ipq
Feb  5 19:04:02 localhost kernel:
Feb  5 19:04:02 localhost kernel: ##4 :ipqhashfn(generate & return hash)
Feb  5 19:04:02 localhost kernel: :frag_kfree_skb()
Feb  5 19:04:02 localhost kernel:
Feb  5 19:04:02 localhost kernel: ##5 frag_free_queue(free the fragment queue)
```

This is the corresponding ICMP message sent to the source address 10.0.0.2 (attacker) from the ip_fragment.c code.

```
16:06:16.906477 10.0.0.10 > 10.0.0.2: icmp: ip reassembly time exceeded for
10.0.0.2 > 10.0.0.10: [|udp] (frag 242:36@0+) (ttl 64, len 56) [tos 0xc0]  (ttl 64, id 42452, len 80)
0x0000   45c0 0050 a5d4 0000 4001 c00d 0a00 000a        E..P....@.......
0x0010   0a00 0002 0b01 f1f4 0000 0000 4500 0038        ............E..8
0x0020   00f2 2000 4011 45b8 0a00 0002 0a00 000a        ....@.E.........
0x0030   df61 2384 0024                                 .a#..$
```

The purpose for using '##' in the printk messages, is to list the order of functions called. This also sets up the format for explanation of the above messages. The functions are numbered as they appear in the code. The order of the numbers is a guide, which will help in comparing fragments.

```
[root@localhost root]# grep '##' /home/teardrop.kern
Feb  5 19:03:32 localhost kernel: ##17 ip_defrag( Process an incoming IP datagram fragment )
Feb  5 19:03:32 localhost kernel: ##3 :ipqhashfn(generate & return hash)
Feb  5 19:03:32 localhost kernel: ##14 :ip_find( Find the correct entry in the incomplete
Feb  5 19:03:32 localhost kernel: ##                datagrams queue or this IP datagram,
Feb  5 19:03:32 localhost kernel: ##                and create new one, if nothing is found.
Feb  5 19:03:32 localhost kernel: ##13 :ip_frag_create( Add an entry to the 'ipq' queue for
Feb  5 19:03:32 localhost kernel: ##                        a newly received IP datagram )
Feb  5 19:03:32 localhost kernel: ##6 frag_alloc_queue()
Feb  5 19:03:32 localhost kernel: ##13 returned
Feb  5 19:03:32 localhost kernel: ##12 ip_frag_intern()
Feb  5 19:03:32 localhost kernel: ##15 :ip_frag_queue( Add new segment to existing queue.)
Feb  5 19:03:32 localhost kernel: ##17 returned
Feb  5 19:03:32 localhost kernel: ##8 ipg_put()
Feb  5 19:03:32 localhost kernel: ##17 ip_defrag( Process an incoming IP datagram fragment )
Feb  5 19:03:32 localhost kernel: ##3 :ipqhashfn(generate & return hash)
Feb  5 19:03:32 localhost kernel: ##14 :ip_find( Find the correct entry in the incomplete
Feb  5 19:03:32 localhost kernel: ##                datagrams queue or this IP datagram,
Feb  5 19:03:32 localhost kernel: ##                and create new one, if nothing is found.
Feb  5 19:03:32 localhost kernel: ##15 :ip_frag_queue( Add new segment to existing queue.)
Feb  5 19:03:33 localhost kernel: ##17 returned
Feb  5 19:03:33 localhost kernel: ##8 ipg_put()
Feb  5 19:04:02 localhost kernel: ##11 :ip_expire(A fragment queue timed out.
Feb  5 19:04:02 localhost kernel: ##9 ip_q_kill(Kill ipq entry. It is not destroyed immediately,
Feb  5 19:04:02 localhost kernel: ##2 ipq_unlink(lock & unlock for __ipq_unlink()
Feb  5 19:04:02 localhost kernel: ##1 ipq_unlink(unlink ipq struct from linklist)
Feb  5 19:04:02 localhost kernel: ##8 ipg_put()
```

Feb 5 19:04:02 localhost kernel: ##7 ip_frag_destroy(complete destruction of fragments - ipq
Feb 5 19:04:02 localhost kernel: ##4 :ipqhashfn(generate & return hash)
Feb 5 19:04:02 localhost kernel: ##5 frag_free_queue(free the fragment queue

*** Note Start of 1st Packet
--- 1st function for ip_fragment.c
##17 ip_defrag(Process an incoming IP datagram fragment)

This is the entry point to 'ip_fragment.c'. The fragment cache limits are set here. The memory size of 256K * 1024 is the high_thresh. The low_thresh is 192K * 1024. The byte count from reading the data from &ip_frag_mem is compared to the high_thresh value. A queue header for the data is either located or created.

##3 :ipqhashfn(generate & return hash)

The fragment code generates its own hash value, to use for data verification when processing segments. The hash value is 43.

##14 :ip_find(Find the correct entry in the incomplete
datagrams queue for this IP datagram,
and create new one, if nothing is found.

This is the loop where the code looks for other matching datagrams or create a new datagram in the queue. The queue pointer qp is valued at 0. The loop will not run, this indicates the first datagram. The match is not found and the ip_frag_create function is called.

##13 :ip_frag_create(Add an entry to the 'ipq' queue for
a newly received IP datagram)

This function makes an entry placing the IP datagram into the queue.

##6 frag_alloc_queue()

This function checks memory for the sizeof structure ipq. If ok, this structure is added to the ip_frag_mem address.

##13 returned

Return comment to know that we returned back to this function. The frag_alloc_queue is check to see if it is equal to NULL. If so this generates an out_of_memory error. The following fields are set in the (qp->) structure from the (iph->) structure.

```
qp->protocol = iph->protocol;
qp->last_in = 0;
qp->id = iph->id;
qp->saddr = iph->saddr;
qp->daddr = iph->daddr;
qp->len = 0;
qp->meat = 0;
qp->fragments = NULL;
qp->iif = 0;
```

The timer is set for this entry, which is the time limit for reassembling the datagrams. The timer is the controlling factor in stopping the reassembly process.

```
nit_timer(&qp->timer);
qp->timer.data = (unsigned long) qp;          /* pointer to queue          */
```

qp->timer.function = ip_expire; /* expire function */

##12 ip_frag_intern()

Due to SMP another cpu could have created an entry in the hash table. The hash value 43 is still in tact. The queue pointer value at this time is 0. The loop will not run which means there won't be a match for the first datagram.

##15 :ip_frag_queue(Add new segment to existing queue.)

This function is one of the longer functions with more checks on the datagram received. The first check is comparing the qp->last = 0 to constant labeled 'COMPLETE' set to 4 . If the statement 'if(qp->last & COMPLETE)' is true, then the code will jump to the err tag. The result is not true, so code follows through. The fragment flags value is shown as '2000'. The fragment offset = 0, total length = 56, and the IP header length = 20. The variable 'end' is set by the statement 'end = offset+skb->len – ihl;'. The variable 'end' equals 36. The variable 'end' is the expected next fragment offset value.

The fragment flags are checked to see if this is the final fragment. The flags value is 0x02 and constant for more fragments 'IP_MF' also equals 0x02. The instruction to compare the two values is 'if((flags & IP_MF) == 0) { /* process final fragment */ }'. This if statement does not equate to 0, and the code processes the { NOT final fragment code }. The next statement is 'if (end&7) '. If true the variable 'end' is altered, to 'end &= ~7;'. 'end' is now equal to 32. The variable 'end' = 32 is tested to be greater than qp->len which is 0.

If some of the bits are beyond the the end, this would mean corruption. The value of p->last_in which is 0, and LAST_IN which is 1 are compared in the if statement 'if (p->last_in & LAST_IN) goto err;'. The if statement equals 0, and p->last is set to 32. If the variable 'end' is equal to 'offset' then an error is triggered. The next issue is to find out which fragments are in front or behind this fragment in the chain of fragments. This tell us where to put this fragment. Variable 'qp->fragments' = 0, so there are no other fragments to compare this one to. This is the first fragment. The fragment is now inserted in the chain of fragments.

##17 returned

Control is returned to this function which is the first function executed. This if statement determines if the function 'ip_frag_reassembly' is called. 'if (qp->last_in == (FIRST_IN|LAST_IN) && qp->meat == qp->len)'. The statement is not true, and the code continues.

##8 ipg_put()

This is the last function run in ip_fragment.c for the first fragment. There is a final test in the if statement:
'if (atomic_dec_and_test(&ipq->refcnt)) ip_frag_destroy(ipq);' The function ip_frag_destroy is ##7, and no message appears for this packet. The fragment has passed this test and is accepted in the kernel.

*** Note Start of 2nd Packet

*** Note Functions with the same results will have the same text as above.

Feb 5 19:03:32 localhost kernel: --- 1st function for ip_fragment.c
##17 ip_defrag(Process an incoming IP datagram fragment)

This is the entry point to 'ip_fragment.c'. The fragment cache limits are set here. The memory size of 256K * 1024 is the high_thresh. The low_thresh is 192K * 1024. The byte count from reading the data from &ip_frag_mem is compared to the high_thresh value. A queue header for the data is either located or created.

##3 :ipqhashfn(generate & return hash)

The fragment code generates its own hash value, to use for data verification when processing segments. The hash value is 43.

##14 :ip_find(Find the correct entry in the incomplete
datagrams queue or this IP datagram,
and create new one, if nothing is found.

This is the loop where the code looks for other datagrams or create a new datagram in the queue. The queue pointer qp is valued at c8c84600. The loop compares ID, source address, destination address, and protocol for match. The existing fragment was matched. This is at least the second fragment.

##15 : ip_frag_queue(Add new segment to existing queue.)

The value of qp->last at this time is 2 and 'COMPLETE' is defined as 4. The statement 'if(qp->last & COMPLETE)' is 0, the code continues. The fragment flag is now 0x00. This information is confirmed in the tcpdump output above. The fragment offset = 24, total length = 24, and the IP header length = 20. The variable 'end' is set by the statement 'end = offset+skb->len – ihl;'. The variable 'end' equals 28. This is the exploit of teardrop.

The fragment flags are checked to see if this is the final fragment. The flags value is 0x00 and the constant 'IP_MF' equals 0x02. The instruction to compare the two values is 'if((flags & IP_MF) == 0) { /* process final fragment */ }' The if statement is true, and the process final fragment code is executed.

The next check is to see if we have some bits beyond end or have a different end, then the segment is corrupted. The if statement for this check is 'if (end < qp->len || ((qp->last_in & LAST_IN) && end != qp->len)) goto err;'. Variables included: Note, the qp->len is different than the skb->len.
'if (28 < 32 || ((2 & LAST_IN) && 28 != 32)) goto err;'. The if statement is false and the code continues.

##8 ipg_put()

This is the last function run in ip_fragment.c for the first fragment. There is a final test in the if statement:
'if (atomic_dec_and_test(&ipq->refcnt)) ip_frag_destroy(ipq);' The function ip_frag_destroy is ##7, and no message appears for this packet. The fragment has passed this test and is accepted in the kernel.

##11 :ip_expire(A fragment queue timed out.
Kill it and send an ICMP reply.

The fragment queue timed out trying to reassemble the the

overlapping of fragments sent by teardrop..

##9 ip_q_kill(Kill ipq entry. It is not destroyed immediately,
because caller (and someone more)
holds reference count.)
The del_timer() function is called to delete the timer structure.

##2 ipq_unlink(lock & unlock for __ipq_unlink()
A lock is placed for the ip fragment. Once complete the fragment is unlinked. A unlock finished the process.

##1 ipq_unlink(unlink ipq struct from linklist)
The 'ip_frag_nqueues' variable is decremented by 1;

##11 Returned
Two statistical functions are called :
IP_INC_STATS_BH(IpReasmTimeout);
IP_INC_STATS_BH(IpReasmFails);
The code kills the fragment and sends an ICMP reply:
icmp_send(head, ICMP_TIME_EXCEEDED, ICMP_EXC_FRAGTIME, 0);

##8 ipg_put()
This is the last function run in ip_fragment.c for the first fragment. There is a final test in the if statement:
'if (atomic_dec_and_test(&ipq->refcnt)) ip_frag_destroy(ipq);' The function ip_frag_destroy is ##7, and this is the next function.

##7 ip_frag_destroy(complete destruction of fragments - ipq
Complete destruction of ipq. Releases all fragment data and the queue descriptor.

##4 :ipqhashfn(generate & return hash)
Gives back ip_frag_memory and frees the skb buffer.

##5 frag_free_queue(free the fragment queue)
Frees ipq and qp buffers.

This is the ICMP message sent to the source address 10.0.0.2 (attacker) from the ip_fragment.c code, when the reassembly timer expired. This is the only response from the kernel to this attack.

```
16:06:16.906477 10.0.0.10 > 10.0.0.2: icmp: ip reassembly time exceeded for
10.0.0.2 > 10.0.0.10: [|udp] (frag 242:36@0+) (ttl 64, len 56) [tos 0xc0]  (ttl 64, id 42452, len 80)
0x0000   45c0 0050 a5d4 0000 4001 c00d 0a00 000a        E..P....@.......
0x0010   0a00 0002 0b01 f1f4 0000 0000 4500 0038        ............E..8
0x0020   00f2 2000 4011 45b8 0a00 0002 0a00 000a        ....@.E.........
0x0030   df61 2384 0024                                 .a#..$
```

The characters '##' are used in the grep command to exclude just the functions called. With less detail it is easier to follow the flow of functions.

```
[root@localhost root]# grep '##' /home/teardrop.kern
Feb  5 19:03:32 localhost kernel: ##17 ip_defrag( Process an incoming IP datagram fragment )
Feb  5 19:03:32 localhost kernel: ##3 :ipqhashfn(generate & return hash)
Feb  5 19:03:32 localhost kernel: ##14 :ip_find( Find the correct entry in the incomplete
```

```
Feb  5 19:03:32 localhost kernel: ##                datagrams queue or this IP datagram,
Feb  5 19:03:32 localhost kernel: ##                and create new one, if nothing is found.
Feb  5 19:03:32 localhost kernel: ##13 :ip_frag_create( Add an entry to the 'ipq' queue for
Feb  5 19:03:32 localhost kernel: ##                      a newly received IP datagram )
Feb  5 19:03:32 localhost kernel: ##6 frag_alloc_queue()
Feb  5 19:03:32 localhost kernel: ##13 returned
Feb  5 19:03:32 localhost kernel: ##12 ip_frag_intern()
Feb  5 19:03:32 localhost kernel: ##15 :ip_frag_queue( Add new segment to existing queue.)
Feb  5 19:03:32 localhost kernel: ##17 returned
Feb  5 19:03:32 localhost kernel: ##8 ipg_put()
Feb  5 19:03:32 localhost kernel: ##17 ip_defrag( Process an incoming IP datagram fragment )
Feb  5 19:03:32 localhost kernel: ##3 :ipqhashfn(generate & return hash)
Feb  5 19:03:32 localhost kernel: ##14 :ip_find( Find the correct entry in the incomplete
Feb  5 19:03:32 localhost kernel: ##                datagrams queue or this IP datagram,
Feb  5 19:03:32 localhost kernel: ##                and create new one, if nothing is found.
Feb  5 19:03:32 localhost kernel: ##15 :ip_frag_queue( Add new segment to existing queue.)
Feb  5 19:03:33 localhost kernel: ##17 returned
Feb  5 19:03:33 localhost kernel: ##8 ipg_put()
Feb  5 19:04:02 localhost kernel: ##11 :ip_expire(A fragment queue timed out.
Feb  5 19:04:02 localhost kernel: ##9 ip_q_kill(Kill ipq entry. It is not destroyed immediately,
Feb  5 19:04:02 localhost kernel: ##2 ipq_unlink(lock & unlock for __ipq_unlink()
Feb  5 19:04:02 localhost kernel: ##1 ipq_unlink(unlink ipq struct from linklist)
Feb  5 19:04:02 localhost kernel: ##8 ipg_put()
Feb  5 19:04:02 localhost kernel: ##7 ip_frag_destroy(complete destruction of fragments - ipq
Feb  5 19:04:02 localhost kernel: ##4 :ipqhashfn(generate & return hash)
Feb  5 19:04:02 localhost kernel: ##5 frag_free_queue(free the fragment queue
```

There is an entry point to function ##17 and a hash value is generate in function ##3. The queue is checked for previous IP datagrams. If a matching IP datagram is found , a segment is added to the queue and accepted by the kernel. If no, memory is allocated for the queue and the IP datagram is added. Then the segment is added to the queue. The kernel accepts this fragment and exits the fragment code.

The kernel has already provided a defense against the teardrop attack. This supports the kernel has a role to play in intrusion detection. This role continues as fragment and various other attacks test the kernel's stability. By the existence of the protections in the kernel, stopping intrusions was already considered.

The Newbonk overlapping fragment attack creates a different response from the kernel. Newbonk is still successful in causing a DoS attack with this version of the kernel. The printk() messages are also sent to the syslog, with unexpected results. The buffer size plays a role in receiving the proper messages in order to track the progress of the kernel. Once the messaging is cleared up the DoS loop in the fragment code is evident. Since the syslogd could not send all of the messages properly, segments of the log are shown. The kernel also loops in processing the same type of packet sent by newbonk. The following is the log output at the start of the newbonk attack. The start of the log is missing and it is unclear how many messages are missing. Alternate buffer sizes are tested to hopefully capture all of the messages. The performance will suffer with the over head of many messages going to the syslog. Another kernel is compared to determine it's reaction to newbonk as well. Changes to the kernel are reviewed to fix the problem.

The following is a listing generated by the altered kernel to show the process of overlapping fragmentation by the newbonk exploit. All messages are produces with the printk command, using the KERN_INFO parameter to send the message to the syslog. In order to gather this information from the syslog, the program 'tail' is used.

```
# tail -f -n 0 /var/log/messages > /home/newbonk.msg

Feb 27 12:54:17 localhost kernel: Instruction order:
Feb 27 12:54:17 localhost kernel: ( (qp->last_in & LAST_IN) : 1
Feb 27 12:54:17 localhost kernel:      &&
Feb 27 12:54:17 localhost kernel: end != qp->len : 12 != 12 )
Feb 27 12:54:17 localhost kernel:      OR
Feb 27 12:54:17 localhost kernel: end < qp->len : 12 < 12
Feb 27 12:54:17 localhost kernel:    goto err;
Feb 27 12:54:17 localhost kernel: end : 12
Feb 27 12:54:17 localhost kernel: offset : 8
Feb 27 12:54:17 localhost kernel: if (end == offset ) goto err
Feb 27 12:54:17 localhost kernel:
Feb 27 12:54:17 localhost kernel: if (pskb_pull(skb, ihl) == NULL) goto err;
Feb 27 12:54:17 localhost kernel: if (pskb_trim(skb, end-offset)) goto err
Feb 27 12:54:18 localhost kernel:
Feb 27 12:54:22 localhost kernel: Find out which fragments are in front and at the back
Feb 27 12:54:22 localhost kernel:   of us in the chain of fragments so far.
Feb 27 12:54:22 localhost kernel: We must know where to put this fragment, right?
Feb 27 12:54:22 localhost kernel:
Feb 27 12:54:22 localhost kernel: next : c5c46280
Feb 27 12:54:22 localhost kernel: for(next = qp->fragments; next != NULL; next = next->next)
Feb 27 12:54:22 localhost kernel:
Feb 27 12:54:22 localhost kernel: Check offsets
Feb 27 12:54:22 localhost kernel: FRAG_CB(next)->offset : 8
Feb 27 12:54:22 localhost kernel: offset : 8
Feb 27 12:54:22 localhost kernel: if (FRAG_CB(next)->offset >= offset) break;
Feb 27 12:54:22 localhost kernel: ## Bingo ------ found Bad offset
Feb 27 12:54:22 localhost kernel:
Feb 27 12:54:22 localhost kernel: We found where to put this one.
Feb 27 12:54:22 localhost kernel: Check for overlap with preceding fragment, and, if needed,
Feb 27 12:54:22 localhost kernel:   align things so that any overlaps are eliminated.
Feb 27 12:54:22 localhost kernel:
Feb 27 12:54:22 localhost kernel: check for preceding fragment
Feb 27 12:54:22 localhost kernel: if (prev)
Feb 27 12:54:22 localhost kernel: ## overlap is 4 bytes
Feb 27 12:54:22 localhost kernel: ## Old fragmnet is completely overridden with new one drop it.
Feb 27 12:54:22 localhost kernel: ## free it.
Feb 27 12:54:22 localhost kernel:
Feb 27 12:54:22 localhost kernel: ##4 :ipqhashfn(generate & return hash)
Feb 27 12:54:22 localhost kernel: :frag_kfree_skb()
Feb 27 12:54:22 localhost kernel: ## Fix - NewBonk, Call here.
Feb 27 12:54:22 localhost kernel:
Feb 27 12:54:22 localhost kernel: Insert this fragment in the chain of fragments.
Feb 27 12:54:22 localhost kernel: ##17 returned
Feb 27 12:54:22 localhost kernel: ? call ip frag reassembly
Feb 27 12:54:22 localhost kernel: No, Will Not call ip_frag_reasm();
Feb 27 12:54:22 localhost kernel:
Feb 27 12:54:22 localhost kernel: ##8 ipg_put()
Feb 27 12:54:22 localhost kernel:
Feb 27 12:54:22 localhost kernel: --- 1st function for ip_fragment.c
Feb 27 12:54:22 localhost kernel: ##17 ip_defrag( Process an incoming IP datagram fragment )
Feb 27 12:54:22 localhost kernel: Start, clean up memory
Feb 27 12:54:22 localhost kernel: sysctl_ipfrag_low_thresh : 196608
Feb 27 12:54:22 localhost kernel: sysctl_ipfrag_high_thresh : 262144
```

```
Feb 27 12:54:22 localhost kernel: Lookup or Create queue header
Feb 27 12:54:22 localhost kernel:
Feb 27 12:54:22 localhost kernel: ##3 :ipqhashfn(generate & return hash)
Feb 27 12:54:22 localhost kernel: hash : 8
Feb 27 12:54:22 localhost kernel:
Feb 27 12:54:22 localhost kernel: ##14 :ip_find( Find the correct entry in the incomplete
Feb 27 12:54:22 localhost kernel: ##                datagrams queue or this IP datagram,
Feb 27 12:54:22 localhost kernel: ##                and create new one, if nothing is found.
Feb 27 12:54:22 localhost kernel: hash : 8
Feb 27 12:54:22 localhost kernel: qp : c48c7b80
Feb 27 12:54:22 localhost kernel: loops if qp != 0
Feb 27 12:54:22 localhost kernel: for(qp = ipq_hash[hash]; qp; qp = qp->next)
Feb 27 12:54:22 localhost kernel: for loop qp : c48c7b80
Feb 27 12:54:22 localhost kernel: compare id, saddr ,daddr, protocol for match
Feb 27 12:54:22 localhost kernel:   in the ipq_hash table for previous fragment
Feb 27 12:54:22 localhost kernel: qp->id : 21764 == id : 21764
Feb 27 12:54:22 localhost kernel: qp->saddr : 10.0.0.2 == saddr : 10.0.0.2
Feb 27 12:54:22 localhost kernel: qp->daddr : 10.0.0.10 == daddr : 10.0.0.10
Feb 27 12:54:22 localhost kernel: qp->protocol : 17 == protocol : 17
Feb 27 12:54:22 localhost kernel: match !, found existing fragment
Feb 27 12:54:22 localhost kernel:
Feb 27 12:54:22 localhost kernel: ##15 :ip_frag_queue( Add new segment to existing queue.)
Feb 27 12:54:22 localhost kernel: qp->last_in : 1
Feb 27 12:54:22 localhost kernel: COMPLETE : 4
Feb 27 12:54:22 localhost kernel: (qp->last_in & COMPLETE) : 0
Feb 27 12:54:22 localhost kernel: if (qp->last_in & COMPLETE ) goto err
Feb 27 12:54:22 localhost kernel:
Feb 27 12:54:22 localhost kernel: flags : 0
Feb 27 12:54:22 localhost kernel: Determine the position of this fragment.
Feb 27 12:54:22 localhost kernel: offset : 8
Feb 27 12:54:23 localhost kernel: skb->len : 24
Feb 27 12:54:23 localhost kernel: ihl : 20
Feb 27 12:54:23 localhost kernel: end = offset + skb->len - ihl;
Feb 27 12:54:23 localhost kernel: end = 12
Feb 27 12:54:23 localhost kernel:
Feb 27 12:54:23 localhost kernel: Is this the final fragment?
Feb 27 12:54:23 localhost kernel: flags : 0
Feb 27 12:54:23 localhost kernel: IP_MF : 2000
Feb 27 12:54:23 localhost kernel: (flags & IP_MF) : 0
Feb 27 12:54:23 localhost kernel: if ((flags & IP_MF ) == 0) process final fragment
Feb 27 12:54:23 localhost kernel: Yes, This is this the final fragment.
Feb 27 12:54:23 localhost kernel:
```

In the above listing, highlights are used to note the points of interest. The first message is not function ##17, which is the entry point to the fragment code. The message buffer sizes does not seem adequate for the numerous messages generated. It is not known as to how many messages are missing. The Variable FRAG_CB(next)->offset equals 8. The variable 'offset' is 8 as well. The comment 'Bingo --- found Bad offset' is an original comment located in the fragment code, to note the overlapping of fragments. This fragment overlaps by 4 bytes. The original comment 'Old fragment is completely overridden with new one drop it' is self explanatory.

It is clear that the kernel has some share of responsibility in defending against fragmentation. I have indicated where to change the fragment code to place a fix for the loop. The proposal for the fix is pointed out later. The next point of interest is the fragment flags variable is 0. No fragment bits are turned on. The kernel thinks this is the last fragment. Newbonk sends this fragment continuously and the fragment code never notices that it's the same fragment repeated. This is where the DoS loop occurs.

Using grep to show the function order at execution time, reveals the loop in a readable format. A segment of the output is listed below.

```
[root@localhost root]# grep '##' /home/newbonk.grep

Feb 27 13:33:48 localhost kernel: ##17 ip_defrag( Process an incoming IP datagram fragment )
Feb 27 13:33:48 localhost kernel: ##3 :ipqhashfn(generate & return hash)
Feb 27 13:33:48 localhost kernel: ##14 :ip_find( Find the correct entry in the incomplete
Feb 27 13:33:48 localhost kernel: ##                datagrams queue or this IP datagram,
Feb 27 13:33:48 localhost kernel: ##                and create new one, if nothing is found.
Feb 27 13:33:48 localhost kernel: ## Bingo ------ found Bad offset
Feb 27 13:33:48 localhost kernel: ## overlap is 4 bytes
Feb 27 13:33:48 localhost kernel: ## Old fragmnet is completely overridden with new one drop it.
Feb 27 13:33:48 localhost kernel: ## free it.
Feb 27 13:33:48 localhost kernel: ##4 :ipqhashfn(generate & return hash)
Feb 27 13:33:48 localhost kernel: ## Fix - NewBonk, Call here.
Feb 27 13:33:48 localhost kernel: ##17 returned
Feb 27 13:33:48 localhost kernel: ##8 ipg_put()
Feb 27 13:33:48 localhost kernel: ##17 ip_defrag( Process an incoming IP datagram fragment )
Feb 27 13:33:48 localhost kernel: ##3 :ipqhashfn(generate & return hash)
Feb 27 13:33:48 localhost kernel: ##14 :ip_find( Find the correct entry in the incomplete
Feb 27 13:33:48 localhost kernel: ##                datagrams queue or this IP datagram,
Feb 27 13:33:48 localhost kernel: ##                and create new one, if nothing is found.
Feb 27 13:33:48 localhost kernel: ##15 :ip_frag_queue( Add new segment to existing queue.)
Feb 27 13:33:48 localhost kernel: ## Bingo ------ found Bad offset
Feb 27 13:33:48 localhost kernel: ## overlap is 4 bytes
Feb 27 13:33:48 localhost kernel: ## Old fragmnet is completely overridden with new one drop it.
Feb 27 13:33:48 localhost kernel: ## free it.
Feb 27 13:33:48 localhost kernel: ##4 :ipqhashfn(generate & return hash)
Feb 27 13:33:48 localhost kernel: ## Fix - NewBonk, Call here.
Feb 27 13:33:48 localhost kernel: ##17 returned
Feb 27 13:33:48 localhost kernel: ##8 ipg_put()
Feb 27 13:33:48 localhost kernel: ##17 ip_defrag( Process an incoming IP datagram fragment )
Feb 27 13:33:48 localhost kernel: ##3 :ipqhashfn(generate & return hash)
Feb 27 13:33:48 localhost kernel: ##14 :ip_find( Find the correct entry in the incomplete
Feb 27 13:33:48 localhost kernel: ##                datagrams queue or this IP datagram,
Feb 27 13:33:48 localhost kernel: ##                and create new one, if nothing is found.
Feb 27 13:33:48 localhost kernel: ##15 :ip_frag_queue( Add new segment to existing queue.)
Feb 27 13:33:48 localhost kernel: ## Bingo ------ found Bad offset
Feb 27 13:33:48 localhost kernel: ## overlap is 4 bytes
Feb 27 13:33:48 localhost kernel: ## Old fragmnet is completely overridden with new one drop it.
Feb 27 13:33:48 localhost kernel: ## free it.
Feb 27 13:33:48 localhost kernel: ##4 :ipqhashfn(generate & return hash)
Feb 27 13:33:48 localhost kernel: ## Fix - NewBonk, Call here.
Feb 27 13:33:48 localhost kernel: ##17 returned
Feb 27 13:33:48 localhost kernel: ##8 ipg_put()
```

The above grep output list 3 sets of loops which causes the DoS to the host system. The fragment code found the bad offset, and knows that the fragment overlaps by 4 bytes. The message to free the IP datagram indicates that the there is not a build up of incoming fragments. I noted where I think the insert should go to fix the loop. More control is needed to determine that IP datagram is repeated.

In function ##15, I made a comment to insert the change to the kernel to stop the loop. After using atomic and timer functions in order to effect the reassembly time, it was unsuccessful. Causing a time out error seemed to be the safest way to let the kernel use it's own code to solve the problem. So far I

was unable to trigger the timer. The fix is possible, but will take a lot more research of the kernel source code to safely stop the loop.

Conditions to consider :
- A count of IP datagrams within a certain time period.
- Check if the offset was already used.
- Check the data length for unrealistic sizes.
- Dynamically drop the repeated fragments for a period of time before reassembly, by matching the address, offset, and ID.

It is clear that thought was already given to this problem by freeing the IP datagram according to the messages. If the host is the last line of defense for this attack, further protections are needed. I believe this to be an ongoing process, and will gain more attention as user's report vulnerabilities.

An attempt was made to find the correct buffer size to capture the start of the messaging. The printk() source code is located at '/usr/src/linux-2.4.18-14/kernel/printk.c'. The constant LOG_BUF_LEN is defined in the source code. The current value used is 16,384 bytes. The LOG_BUF_LEN is increased twice to 65,536 and 131,072 bytes. The kernel re-compiled and booted successfully. The same result occurred with the messages missing at the start. The increase of this buffer did not solve this problem It is possible that this is not the buffer in need of a larger size.

The logging of messages added to the fragment does effect the performance of the system. The CPU usage goes directly to 100%, as soon as the attack is started. This is not a practical feature to use to monitor fragment attacks. This is an informational feature developed to understand more about the kernels role in defending against overlapping fragment attacks. A newer kernel was also tested using teardrop and newbonk. The next kernel test is kernel version 2.4.25. This kernel was re-compiled and booted successfully. The same results occurred. The fragmentation code did not change with respect to overlapping fragmentation. Kernel version 2.6.3 was also re-compiled for testing. The kernel did not boot successfully. An error occurred involving ext 3. The fragment source code for this version was reviewed and no changes where made to the section in question.

RFC1858 covers security issues when dealing with overlapping IP fragments. Blocking these attacks is desirable because they can compromise a host, or tie up all of it's internal resources. Prevention is possible by adopting better strategy in the router's IP filtering code. The following are points for the prevention of overlapping fragment attacks.

- enforce a minimum fragment offset for fragments that have non-zero offsets, it can prevent overlaps in filter parameter regions of the transport headers.
- In the case of TCP, this minimum is sixteen octets, to ensure that the TCP flags field is never contained in a non-zero-offset fragment.

- Minimum MTU on a link should be 68 bytes.
- A general algorithm, for ensuring that filters work in the face of both the tiny fragment attack and the overlapping fragment attack is:

  IF FO=1 and PROTOCOL=TCP then DROP PACKET

Is the host the right place to defend against fragment attacks? My answer is YES. It's already in the kernel code. The complete responsibility does not fall on the host systems alone to handle these attacks. Firewalls and routers can help in this situation as discussed next. If these intermediate devices are not configured properly the burden still falls on the kernel. Since the kernel is already looking for fragment errors, it is obvious that the developers reflect this the source code.

Firewalls and routers also share a role in defending against fragment attacks. The firewall and routers do not reassembly the packet. That is not there function. They have the capability of identifying fragmented packets with some additional information. At this stage, the fragmented packet usually dropped so the host never receives it. It is a potential problem, that does not need to get worse. Blocking fragmented packets can also stop legitimate traffic form getting through.

Iptables available in the linux kernel. It is a packet filtering administration tool. Iptables is able to setup, maintain, ans inspect the tables of IP packet filtering rules in the linux kernel. One of the rule specifications is ' f, or fragment'. This rule refers to the second and further fragmented packets, which be dropped. In this case only the first packet would get through. Next is an example of Iptables statements, related to dropping fragmentation before it get to the host:

```
# Drop All Fragments  #
iptables -A INPUT -i eth0  -f  -j LOG --log-level debug --log-prefix "IPTABLES \ FRAGMENTS: "
iptables -A INPUT -i eth0  -f  -j DROP
```

The above statements are filtering incoming packets on device eth0. The rule specification ' -f ' is highlighted. The first statement sends a message to the log to inform the administrator that fragmented packets are reaching this system. The second statement then drops this packet, so that it never reaches the internal network. This is another defense mechanism that the kernel performs.

Cisco Systems provides features to protection against fragments in the Cisco secure PIX firewall and Router software. The command is called 'sysopt security fragguard' which enables the IP Frag Guard feature. The Frag Guard feature enforces two additional checks, in addition to the recommended checks in RFC 1858. The first is that an IP fragment must be associated with an already seen valid initial IP fragment. This just means that the initial fragment is required. The second check is that IP fragments are rated 100 full IP fragmented packets per second to each internal host. This means that the PIX will process

1200 packet fragments per second.

Cisco also uses the Access Control List (ACL) to identify fragmented packets at the IP level. The statement below is an example of how a fragmented packet is identified:

access-list 101 deny ip any host 151.36.23.3 fragments

This statement gives a result of true of false in order to control the flow of this packet. The ACL statement is used to block forwarding of this packet as desired. Since the router's role is to route packets, detailed analysis of packets will use needed resources. The primary job of the firewall is to block packets, and then route them.

```
[root@localhost etc]# snort -i eth1 -v -A full -s /home/snort/alert.msg -c snortbck.conf -l
/home/snort
[**] (spp_frag2) Teardrop attack [**]
02/28-17:53:30.221863 10.0.0.2 -> 10.0.0.10
UDP TTL:255 TOS:0x0 ID:1109 IpLen:20 DgmLen:24
Frag Offset: 0x0001   Frag Size: 0x0003
=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=
root@localhost snort-1.9.1]# grep 'Teardrop' -r rules
rules/dos.rules:alert udp $EXTERNAL_NET any -> $HOME_NET any (msg:"DOS Teardrop
attack"; id:242; fragbits:M; reference:cve,CAN-1999-0015;
reference:url,www.cert.org/advisories/CA-1997-28.html; reference:bugtraq,124;
classtype:attempted-dos; sid:270; rev:2;)
```

What is the IDS level :

Criticality : 2 : The target system is the host where packets are reassembled.

Lethality : 3 : Not lethal, no damage is done. The system clears up as soon as the attack ends.

System Counter : 4 : The kernel defense is adequate, even though it loops.

Network Counter : 4 : The network can stop fragments at firewalls and routers.

(2 + 3) - (4 + 4) = -3

Conclusion

How does the host defend against overlapping fragment attacks. The next logical place is to review the fragmentation source code available. Kernel version 2.4.18-14 is used for this purpose. By adding the printk functions to the fragment source code, messages may be sent to the syslog to trace the logic of the fragment re-assembly functions. The Teardrop attack generates understandable messages which shows the logic of the kernels fragmentation code. This is an overlapping fragment attack, which sends only two IP fragments. By viewing only the functions called, the messaging shows that it is

clear that the kernel is prepared for this attack. Another overlapping fragment attack generates different results. The Newbonk attack sends a normal IP fragment at first. The second packet overlaps the first packet and is sent repeatedly to the host system. This causes a DoS attack on the receiving host, as it tries to reassemble the same packet over and over again. The fragment code does not recognize that this is the same packet repeated. After reviewing the fragmentation code I determined where to insert the fix to stop the loop causing the DoS. I was unable to trigger the reassembly timer to stop the loop. No functions where added do to unknown characteristics of the kernel. More time is needed to research the fragmentation and other source code to provide a safe solution.

Larger buffer sizes where used as not all the messaging made it to the syslog during the newbonk attack. This proved to be unsuccessful. A newer version of the kernel was compiled tested as well. I got the same results, and viewing the fragmentation source code showed no changes to the reassembly section in question. RFC 1858 contains security checks for over lapping fragments. Some vendors have included these checks and additional ideas into their code. Firewalls and Routers have some capability to check for fragmented packets. Examples where given for Iptables, Cisco PIX, and ACL statements, as they related to fragmentation. Snort rules consider the Newbonk attack the same as the Teardrop attack. This is correct because they are both overlapping fragment attacks. The IDS severity level was determined to be -3. This turns out not to be a major alarm.

The course of intrusion detection is growing in 3 areas. The intrusion detection systems such as snort, intermediate systems such as firewalls and routers, and the third at the host level.

Part 2 – Network Detects

Detect 2.1

Source of Trace

The following network detection was taken from the log file 2002.10.12 located at http://www.incidents.org/logs/Raw. The logs were saved in the Tcpdump binary format. The following detect is in file 2002.10.12. The log information was generated from Snort which read the file 2002.10.12 as input.

```
[**] [1:1322:4] BAD TRAFFIC bad frag bits [**]
[Classification: Misc activity] [Priority: 3]
11/11-19:13:30.936507 213.105.102.161 -> 207.166.185.174
TCP TTL:48 TOS:0x0 ID:1314 IpLen:20 DgmLen:1468 DF MF
Frag Offset: 0x0000   Frag Size: 0x05A8
```

Detect was Generated By

This detect was generated by the Snort Intrusion Detection System version snort-1.9.1 Snort produced the alerts when reading the Tcpdump binary logs using the -r option, and writing the alerts to a specified log directory using the -l option. The following command was used : snort -A full -v -r /home/snort-

1.9.1/etc/raw/2002.10.12 -c /home/snort-1.9.1/etc/snort.conf -l /home/snort-1.9.1/etc/log/10.12, which produced the alert log file.

The packet triggered the alert, because the fragment flag value is 0x6. This means that the don't fragment (DF) bit is on and the more fragments (MF) bits is also on for this packet. The snort rule checks the more fragments bit and then checks the don't fragment bit. If both are on, this alert is triggered.

Rule :
alert ip $EXTERNAL_NET any -> $HOME_NET any (msg:"BAD-TRAFFIC bad frag bits"; fragbits:MD; sid:1322; classtype:misc-activity; rev:6;)

Probability Source Address was Spoofed

The source address is not spoofed. The intention of this packet is to get the host to reply with an ICMP error message (type 3, code 5). The source address needs this response for reconnaissance. If there was a hardware malfunction on a firewall or router, I would expect many more alerts.

Description of Attack

This is a reconnaissance attack which probes the host in order to obtain a response to determine the type of operating system used. Once the desired information is received the attacker could escalate the level of attack on that network and/or host. This malformed packet is crafted for a response, by intentionally sending irregular fragment flag indicators.

This packet is irregular because the don't fragment (DF) bit is on and the more fragments (MF) bit is on. The don't fragment bit indicates to the host reassembling the packet, that there are no fragments needed to complete this packet. At the same time the more fragment (MF) bit tells the host, that more fragments are needed to complete the re-assembly of this packet. This confusion is answered with the ICMP error message to the source address.

As an example of how and why the ICMP message is sent, the kernel for the linux host in Part 1 of this paper is used. The source code 'ip_fragment.c' re-assembles packets the the host system. Informational messages are inserted into each function, with messages directed to the syslog. The function printk() is used to send messages to the kernel. The messages will show the real time logic of the fragment reassembly of the host. I was able to craft this packet as well, which is explained later. The packet dump is shown in the next section. Refer to Part 1 of this paper for a detailed explanation for the adjusted kernel and it's fragmentation process. The following is the syslog output from the host receiving the bad frag bits packet:

```
Mar  1 00:19:54 localhost kernel: --- 1st function for ip_fragment.c
Mar  1 00:19:54 localhost kernel: ##17 ip_defrag( Process an incoming IP datagram fragment )
Mar  1 00:19:54 localhost kernel: ##3 :ipqhashfn(generate & return hash)
Mar  1 00:19:54 localhost kernel: hash : 43
Mar  1 00:19:54 localhost kernel: ##14 :ip_find( Find the correct entry in the incomplete
Mar  1 00:19:54 localhost kernel: ##                datagrams queue or this IP datagram,
Mar  1 00:19:54 localhost kernel: ##                and create new one, if nothing is found.
Mar  1 00:19:54 localhost kernel: match not found, first fragment.
Mar  1 00:19:54 localhost kernel: ##13 :ip_frag_create( Add an entry to the 'ipq' queue for
```

```
Mar  1 00:19:54 localhost kernel: ##                    a newly received IP datagram )
Mar  1 00:19:54 localhost kernel: ##6 frag_alloc_queue()
Mar  1 00:19:54 localhost kernel: ##13 returned
Mar  1 00:19:54 localhost kernel: Initialize a timer for this entry.
Mar  1 00:19:54 localhost kernel: match not found, first fragment.
Mar  1 00:19:54 localhost kernel: ##15 :ip_frag_queue( Add new segment to existing queue.)
Mar  1 00:19:54 localhost kernel: flags : 6000
Mar  1 00:19:54 localhost kernel: IP_MF : 2000
Mar  1 00:19:54 localhost kernel: No, This is this the not the final fragment.
Mar  1 00:19:54 localhost kernel: We found where to put this one.
Mar  1 00:19:54 localhost kernel: Check for overlap with preceding fragment, and, if needed,
Mar  1 00:19:54 localhost kernel:   align things so that any overlaps are eliminated.
Mar  1 00:19:54 localhost kernel: Insert this fragment in the chain of fragments.
Mar  1 00:19:54 localhost kernel: ##17 returned
Mar  1 00:19:54 localhost kernel: ? call ip frag reassembly
Mar  1 00:19:54 localhost kernel: No, Will Not call ip_frag_reasm();
Mar  1 00:19:54 localhost kernel: ##8 ipg_put()
Mar  1 00:19:54 localhost kernel: ##11 :ip_expire(A fragment queue timed out.
Mar  1 00:19:54 localhost kernel:                  Kill it and send an ICMP reply.
Mar  1 00:19:54 localhost kernel: ##9 ip_q_kill(Kill ipq entry. It is not destroyed immediately,
Mar  1 00:19:54 localhost kernel:               because caller (and someone more)
Mar  1 00:19:54 localhost kernel:               holds reference count.)
Mar  1 00:19:54 localhost kernel: ##2 ipq_unlink(lock & unlock for __ipq_unlink()
Mar  1 00:19:54 localhost kernel: ##1 ipq_unlink(unlink ipq struct from linklist)
Mar  1 00:19:54 localhost kernel: Send an ICMP 'Fragment Reassembly Timeout' message.
Mar  1 00:19:54 localhost kernel: ICMP_TIME_EXCEEDED, ICMP_EXC_FRAGTIME
Mar  1 00:19:54 localhost kernel: ##8 ipg_put()
Mar  1 00:19:54 localhost kernel: ##7 ip_frag_destroy(complete destruction of fragments - ipq
Mar  1 00:19:54 localhost kernel: ##4 :ipqhashfn(generate & return hash)
Mar  1 00:19:54 localhost kernel: :frag_kfree_skb()
Mar  1 00:19:54 localhost kernel: ##5 frag_free_queue(free the fragment queue)
```

The messages to note about crafted packet are highlighted. Function # 17 is the entry point to the fragmentation code. The host immediately tries to find another matching packet. The match is not found, this is the first packet. The packet is added to the queue. The fragment flag is set to 0x6. This indicates both the DF and MF flags are on. The host believes this is not the final packet. This packet is inserted in the expected chain of fragments. Function #8 means that the host has received the packet. Notice the time, immediately function # 11 is executed. Function # 11 is the fragment queue timeout, and send the ICMP reply to the source address. The ICMP reply is expected. The fact that the host's queue timer stops immediately could aid in OS fingerprinting

To test other operating system's response to this packet Cisco and Windows systems are used. The Cisco system is a PIX 506 Firewall, version 6.0 . The packet was sent to the inside interface ethernet 1. The default configuration was not changed, just eth 1 is activated. This is the trusted interface and ethernet 0 is the un-trusted interface.

```
Tcpdump output of captured packets.
Attacker's IP Address : 10.0.0.2
PIX          IP Address : 10.0.0.6
21:12:32.150303 10.0.0.2.2419 > 10.0.0.6.6623: [no cksum] udp 28
 (frag 242:36@0+) (ttl 64, len 56)
0x0000   4500 0038 00f2 6000 4011 05bc 0a00 0002        E..8..`.@.......
0x0010   0a00 0006 0973 19df 0024 0000 0000 0000        .....s...$......
0x0020   0000 0000 0000 0000 0000 0000 0000 0000         ................
0x0030   0000 0000 0000
```

Notice that the PIX did not reply to the at all to the bad packet. This non-response provides information for fingerprinting.

When testing Windows XP against the crafted packet, the results are different. The following is the tcpdump output of the captured packets.

Attacker's IP Address : 10.0.0.2 Windows XP IP Address : 10.0.0.3

```
19:51:37.131317 10.0.0.2.58154 > 10.0.0.3.7510: [no cksum] udp 28
(frag 242:36@0+) (ttl 64, len 56)
0x0000          4500 0038 00f2 6000 4011 05bf 0a00 0002          E..8..`.@.......
0x0010          0a00 0003 e32a 1d56 0024 0000 0000 0000          .....*.V.$......
0x0020          0000 0000 0000 0000 0000 0000 0000 0000          ................
0x0030          0000 0000 0000                                   ......

19:52:42.007130 10.0.0.3 > 10.0.0.2: icmp: ip reassembly time exceeded for 10.0.0.2 > 10.0.0.3:
[|udp] (frag 242:36@0+) (ttl 64, len 56) (ttl 128, id 197, len 56)
0x0000   4500 0038 00c5 0000 8001 25fc 0a00 0003        E..8......%.....
0x0010   0a00 0002 0b01 f459 0000 0000 4500 0038        .......Y....E..8
0x0020   00f2 6000 4011 05bf 0a00 0002 0a00 0003        ..`.@...........
0x0030   e32a 1d56 0024                                 .*.V.$
```

Windows XP responded with the expected ICMP error message. The difference is the time. The ICMP message was sent approximately 5 seconds after receiving the bad packet. This again is useful information for fingerprinting this operating system.

Attack Mechanism

In order to craft a packet with both the DF and MF bits on, I used Teardrop.c located at http:rootshell.com. Teardrop.c is a short and to the point fragment exploit program. I made just 3 changes to successfully re-create the bad frag bits packet.

1) Added IP_DFMF to the defines

```
#define IP_MF   0x2000  /* More IP fragment en route */
 --- Added IP_DFMF
#define IP_DFMF 0x6000  /* Don't Fragment & More IP fragment en route */
```

2) replace IP_MF with IP_DFMF

```
*((u_short *)p_ptr) |= FIX(IP_DFMF);  /* IP frag flags and offset */
```

3) Delete the code sending the second packet.

I renamed the code from teardrop to fragbits

Compile command : gcc -O2 fragbits.c -o fragbits

source IP : 10.0.0.10, destination IP : 10.0.0.2

Execute comand : fragbits 10.0.0.10 10.0.0.2

Tcpdump captured the bad frag bit packet and ICMP reply:

```
[root@localhost root]# tcpdump -i eth1 -vvv -X
tcpdump: listening on eth1
00:31:27.781613 10.0.0.10.26750 > 10.0.0.2.31054: [no cksum] udp 28
(frag 242:36@0+) (ttl 64, len 56)
```

```
0x0000   4500 0038 00f2 6000 4011 05b8 0a00 000a        E..8..`.@.......
0x0010   0a00 0002 687e 794e 0024 0000 0000 0000        ....h~yN.$......
0x0020   0000 0000 0000 0000 0000 0000 0000 0000        ................
0x0030   0000 0000 0000                                 ......
```

00:31:28.494049 10.0.0.2 > 10.0.0.10: icmp: ip reassembly time exceeded for 10.0.0.10 > 10.0.0.2: [|udp] (frag 242:36@0+) (ttl 64, len 56) [tos 0xc0] (ttl 64, id 230, len 80)

The ICMP error messaging from different operating systems (OS), and other devices vary. The following is a list of some of the information obtainable from ICMP error messages:

- The ICMP error message includes at least 8 bytes of the offending datagram, several OS and network devices can echo more than 8 bytes of data.
- The offending packets IP header may be altered by the OS stack implementation.
- Some OS may add or subtract 20 bytes from the IP total length field, some may echo correctly.
- Fragmentation flags and offset values bit order may change with ICMP error messages.
- Some OS may miscalculate, zero out or return the correct checksum of the offending packet.
- Router and Hosts may ignore the last bit field in the Type of Service (TOS) byte.
- Some OS may set the don't fragment (DF) bit in error quoting.
- The TTL for ICMP has 2 separate values. 1) for ICMP query messages and 2) ICMP reply messages.

For more details on ICMP error messaging see reference :
http://www.sys-security.com/archive/articles/login.pdf

Correlations

The RFC 1858 explains the overlapping fragment attack well. This is a valuable reference to this paper.
http://www.ietf.org/rfc/rfc1858.txt

This detect of the same alert, has a similar severity level and differs in evidence of targeting. Their logs showed questionable activities.
http://cert.uni-stuttgart.de/archive/intrusions/2002/11/msg00026.html

This detect reports spoofing of the source address, with active targeting of some sort. The Severity level is similar.
http://cert.uni-stuttgart.de/archive/intrusions/2002/10/msg00222.html

This practical shares the same low severity, and includes an obvious exploit for Windows systems.
http://www.giac.org/practical/GCIA/Donald_Gregory_GCIA.pdf

Subj: "LOGS: GIAC GCIA Version 3.4 Practical,BAD TRAFFIC bad frag bits, Odis Richardson"
Date: 3/23/2004 10:13:53 PM Eastern Standard Time
From: ORich17@aol.com
To: intrusions@incidents.org

No Questions asked.

Evidence of Active Targeting

The files used for analysis are log files 2002.10.1 – 2002.10.18 located at http://www.incidents.org/logs/Raw. To gather information for just this alert, I created my own conversion for snort's alert file data to a mysql ready format. The snort's alert file only includes the "BAD-TRAFFIC bad frag bits" alerts to isolate this information. The following table are the listings from the isolated Bad Fragments alerts.

Bad Fragments Bits Alerts for Destination 207.166.*

Source IP	Destination IP	Count (distinct dst)	Alerts
80.4.97.69	207.166.206.194	3	14
80.7.188.43	207.166.110.24	3	10
213.105.82.236	207.166.26.90	2	8
80.7.2.30	207.166.185.153	2	7
213.105.191.213	207.166.252.145	4	6
62.255.6.16	207.166.79.202	1	6
213.105.102.161	207.166.185.174	1	1

The 2 companies found are New Age Consulting Service (dst) and NTLI Network Management Centre (src). This is not active targeting from an attacker, 3 source addresses are from the same company in Great Britain. Maybe a MTU setting was mis configured at a low number, and a router added the MF bit during routing. The following are the whois for the destination and 3 source addresses:

Destination Address : 207.166.206.194
New Age Consulting Service NACS-BLK-1 (NET-207-166-192-0-1)
207.166.192.0 - 207.166.223.255
New Age Consulting Service, Inc. NACS-DIALUP-LORAIN (NET-207-166-206-0-1)
207.166.206.0 - 207.166.206.255

Source Address : 80.4.97.69
inetnum: 80.4.96.0 – 80.4.103.255 netname: NTL descr: NTL Glasgow - CABLE HEADEND country: GB admin-c: NNMC1-RIPE tech-c: NNMC1-RIPE status: ASSIGNED PA mnt-by: AS5089-MNT changed: hostmaster@ntli.net 20011016 changed: hostmaster@ntli.net 20020815 source: RIPE

Source Address : 213.105.82.236
inetnum: 213.105.80.0 – 213.105.91.255 netname: NTL descr: NTL Internet - Luton

site
country: GB admin-c: NNMC1-RIPE tech-c: NNMC1-RIPE status: ASSIGNED PA
mnt-by: AS5089-MNT changed: hostmaster@ntli.net 20010108
changed: hostmaster@ntli.net 20020815 changed: hostmaster@ntli.net 20030415
source: RIPE

Source Address : 62.255.6.16
inetnum: 62.255.0.0 – 62.255.31.255 netname: NTL descr: NTL Internet
descr: Bristol site country: GB admin-c: NNMC1-RIPE tech-c: NNMC1-RIPE
status: ASSIGNED PA mnt-by: AS5089-MNT changed: hostmaster@ntli.net 20011011
changed: hostmaster@ntli.net 20020815 source: RIPE

Severity

(Target's Criticality + Lethality of Attack) - (System Defense + NetworkDefense)

Target's Criticality – The target is most likely the host and is not extremely critical. : 2

Lethality of Attack – If the ICMP message is sent to the source, no damage is done to the host. : 1

System Defense – The host defense mechanism is strong, because the packet is recognized and the host may or may not reply : 4

Network Defense – The network firewalls and routers can block all fragmented packets, so the host never gets this packet. : 4

The severity for this detection is : 3 – 8 = -5

Defense Recommendation

My response is in two fold. Stop the fragmented packets from coming in and don't let the ICMP messages get out.

Iptables available in the linux kernel. It is a packet filtering administration tool. Iptables is able to setup, maintain, ans inspect the tables of IP packet filtering rules in the linux kernel. One of the rule specifications is ' f, or fragment'. Since the more fragment (MF) bit is on, the packet is matched and dropped. The following is an example of Iptables statement, related to dropping fragmentation before it get to the host:

```
# Drop All Fragments  #
iptables -A INPUT -i eth0 -f -j LOG --log-level debug --log-prefix "IPTABLES \
   FRAGMENTS: "
iptables -A INPUT -i eth0 -f -j DROP
```

The above statements filters incoming packets on device eth0. The rule specification ' -f ' is highlighted. The first statement sends a message to the log to inform the administrator that fragmented packets are reaching this system. The second statement then drops this packet, so that it never reaches the

internal network. This is another defense mechanism that the kernel performs.

Cisco also uses the Access Control List (ACL) to identify fragmented packets at the IP level. The statement below is an example of how a fragmented packet is identified:

```
access-list 102 deny ip any host 207.166.206.194  fragments
```

This statement gives a result of true or false in order to control the flow of this packet. The ACL statement is used to block forwarding of this packet as desired. Since the router's role is to route packets, detailed analysis of packets will use needed resources. The primary job of the firewall is to block unwanted packets, then route the packets to there destination.

By default most networks and hosts machine will return the proper ICMP message, in reaction to the malformed packet. The way to defend your network is to dis-allow outgoing ICMP messages. The attacker will not gain any information with this specific packet because a variety if not all operating systems have this capability.

Ex : Iptables – Linux
```
# Drop all outgoing ICMP for type 3
iptables -A OUTPUT -p icmp -s 0/0 --icmp-type 3 -j DROP
```

Ex : CISCO Pix – Firewall
```
# Deny all unreachable messages
icmp deny any unreachable outside
```

Multiple Choice Question

Question : How many bits represent the fragment flags in the IP header?

1) 2
2) 3
3) 4
4) 8

Answer : # 2) 3 bits

Detect 2.2
Source of Trace

The following network detection was taken from the log file 2002.10.11 located at http://www.incidents.org/logs/Raw. The logs were saved in the Tcpdump binary format. The following detect is in file 2002.10.11. The log information was generated from Snort which read the file 2002.10.11 as input.

```
[**] [1:648:5] SHELLCODE x86 NOOP [**]
[Classification: Executable code was detected] [Priority: 1]
11/01-02:56:01.116507 63.111.48.133:80 -> 207.166.87.157:63390
TCP TTL:113 TOS:0x0 ID:47730 IpLen:20 DgmLen:1500 DF
***A**** Seq: 0x5195B19D  Ack: 0x3E10F9DA  Win: 0xFFB4  TcpLen: 20
```

[Xref => arachnids 181]

Detect was Generated By

This detect was generated by the Snort Intrusion Detection System version
snort-1.9.1 Snort produced the alerts when reading the Tcpdump binary logs using the -r option, and writing the alerts to a specified log directory using the -l option. The following command was used : snort -A full -v -r /home/snort-1.9.1/etc/raw/2002.10.11 -c /home/snort-1.9.1/etc/snort.conf -l /home/snort-1.9.1/etc/log/10.11, which produced the alert log files.

The properties in this packet that triggered the alert, are a series of '9090...' in the data section of the packet. Snort checks the data content for '9090...', and the alert triggered when the data match is true.

Rule:
alert ip $EXTERNAL_NET any -> $HOME_NET $SHELLCODE_PORTS (msg:"SHELLCODE x86 NOOP"; content: "|90 90 90 90 90 90 90 90 90 90 90 90 90 90|"; depth: 128; reference:arachnids,181; classtype:shellcode-detect; sid:648; rev:5;)

Probability Source Address was Spoofed

The source address is not spoofed. Supporting reasons follow. The files used to analyze this alert are log files 2002.10.1 – 2002.10.18 located at http://www.incidents.org/logs/Raw. To gather information about this alert, I created my own conversion for snort's alert file data to mysql ready format.

The following mysql table listing supports my reasoning, that the source address is not spoofed. Notice that there are 548 alerts for the same source and destination address. Other source addresses triggered the same alert to the destination address. This is traffic, not an attacker. This is also a TCP connection which requires the correct source address for the 3-way handshaking to complete.

Alert count for Source IP

Source IP	Destinatio IP	Count
63.111.48.133	207.166.87.157	548
66.220.44.31	207.166.87.157	54
207.188.7.150	207.166.87.157	39
152.3.183.67	207.166.87.157	38
129.118.2.10	170.129.50.120	34

Description of Attack

This attack is presented as a buffer overflow exploit. The attacker is sending machine language code for the host computer to execute as commands. The idea is to overwrite the return pointer for the stack, then execute your command. The series of '9090' are used to move the return pointer to different addresses in the stack.

This attack comes from a series of NOP instructions directed at the Intel x86 architecture. A NOP instruction means no operation is performed when the instruction is read. The attacker is trying to take advantage of non-secured coding practices in order to execute arbitrary code. The NOP instruction allows the attacker to load an address space with numerous NOPs followed by the code to execute. When the NOPs are executed, this is referred to as sledding into the attackers shellcode.

Functions like strcpy(), strcat(), sprintf() do not have protection for boundaries. It is possible to overflow the buffer used by these functions in order to write arbitrary data to the address space of the service. If you can find the location where to execute the newly written data, its possible to execute your shellcode on that system.

To support my reasoning that this is a false positive, I used tcpdump to visually analyze the packet data. I noticed that the series of '9090' are not very long. The series also differs as to the occurrences in the packet data. The '9090' series appears once in the first packet capture and twice in the second packet capture. The '9090' series is highlighted in the following dump.

```
# tcpdump -vvv -X -r 2002.10.11  src host 63.111.48.133 and dst host 207.166.87.157
results :
03:22:49.406507 63.111.48.133.http > 207.166.87.157.63222: . [bad tcp cksum b5b5!]
2299596358:2299597818(1460) ack 3459736803 win 65460 (DF) (ttl 113, id 41940, len 1500,
bad cksum 135a!)
0x0000          4500 05dc a3d4 4000 7106 135a 3f6f 3085                 E.....@.q..Z?o0.
0x0010          cfa6 579d 0050 f6f6 8911 0e46 ce37 64e3                 ..W..P.....F.7d.
0x0020          5010 ffb4 8ef7 0000 c705 f47a 4100 0000                 P..........zA...
0x0030          0000 83c8 ff5b c390 9090 9090 9090 9090                 .....[..........
0x0040          9090 9090 9056 8b74 2408 8b46 0ca8 8374                 .....V.t$..F...t
    .             .     .     .     .     .     .     .                 .
0x05b0          4424 3483 c40c 3c0a 7404 c606 0d46 8b6c                 D$4...<.t....F.l
0x05c0          2418 3bfd 0f82 31ff ffff 2b74 2424 8bee    $.;...1...+t$$..
0x05d0          8bc5 5f5e 5d5b 83c4 0cc3 8b44              .._^][.....D

03:22:49.526507 63.111.48.133.http > 207.166.87.157.63222: . [bad tcp cksum b5b5!]
8760:10220(1460) ack 1 win 65460 (DF) (ttl 113, id 41951, len 1500, bad cksum 134f!)
0x0000          4500 05dc a3df 4000 7106 134f 3f6f 3085                 E.....@.q..O?o0.
0x0010          cfa6 579d 0050 f6f6 8911 307e ce37 64e3                 ..W..P....0~.7d.
0x0020          5010 ffb4 fd3e 0000 006a 0056 ff15 a8a2 P....>...j.V....
0x0030          4100 81fd 6048 4100 740f a1e4 7e41 0055                 A...`HA.t...~A.U
0x0040          6a00 50ff 155c a241 005f 5e33 c05d c390                 j.P..\.A._^3.]..
0x0050          9090 9090 9090 9090 9090 9090 9056 8b74                 .............V.t
    .             .     .     .     .     .     .     .                 .
0x0230          4100 750a 6a10 e882 feff ff83 c404 5ec3 A.u.j.........^.
0x0240          9090 9090 9090 9090 9090 9090 9051 8b0d                 .............Q..
0x0250          8068 4100 538b 5c24 0c55 5657 894c 2410                 .hA.S.\$.UVW.L$.
    .             .     .     .     .     .     .     .                 .
0x05b0          25ff 0000 0003 f03b 7424 1472 bd33 c05f                 %......;t$.r.3._
0x05c0          5e5d 5bc3 8d04 168d 9ff8 0000 003b c373                 ^][..........;.s
0x05d0          092b ca89 0789 4f04 eb09 892f              .+....O..../
```

Attack Mechanism

The bases of this attack is for the x86 architecture. Other

architectures for shellcode exploits are Sparc, MIPS, and PPC. The elements of the attack vary to comply with the requirements of the different architectures. There are 9 types of alerts in the snort shellcode rules file, related to x86. Highlighted are the particular content each rule is looking for. This suggest that the shellcode attacks for the x86, has at least 9 versions.

```
alert ip (msg:"SHELLCODE x86 setgid 0"; content: "|b0b5 cd80|")
alert ip (msg:"SHELLCODE x86 setuid 0"; content: "|b017 cd80|")
alert ip (msg:"SHELLCODE x86 NOOP"; content: "|90 90 90 90 90 90 90 90 90 90 90 90 90 90|")
alert ip (msg:"SHELLCODE x86 stealth NOOP";   content: "|eb 02 eb 02 eb 02|")
alert ip (msg:"SHELLCODE x86 unicode NOOP"; content: "|9000900090009000|";)
alert ip (msg:"SHELLCODE linux shellcode"; content:"|90 90 90 e8 c0 ff ff ff|/bin/sh";)
alert ip (msg:"SHELLCODE x86 inc ebx NOOP"; content:"|43 43 43 43 43 43 43 43 43 43 43 43
43 43 43 43 43 43 43 43 43 43 43|";)
alert ip (msg:"SHELLCODE x86 NOOP"; content:"|61 61 61 61 61 61 61 61 61 61 61 61 61 61
61 61 61 61 61 61|";)
alert ip (msg:"SHELLCODE x86 EB OC NOOP"; content:"|EB 0C EB 0C EB 0C EB 0C EB 0C EB
0C EB 0C EB 0C|";)
```

Since this is a false positive alert from snort, I successfully triggered the alert. This is a JPEG file sent fro the attacker's machine to the host. The method of transfer is NFS. NFS is the Network File System which shares files across a network. NFS runs is UDP or TCP. NFS is using UDP on my local network, which appears in the header information. The difference in this false alarm created is the content match of '4343'.

```
snort -i eth1 -v -X -l log/noop/create -c snort-noop.conf
Rule:
alert ip $EXTERNAL_NET any -> $HOME_NET $SHELLCODE_PORTS (msg:"SHELLCODE x86
inc ebx NOOP"; content:"|43 43 43 43 43 43 43 43 43 43 43 43 43 43 43 43 43 43 43 43 43 43
43|"; classtype:shellcode-detect; sid:1390; rev:3;)

Alert:
[**] SHELLCODE x86 inc ebx NOOP [**]
03/07-13:37:26.487349 10.0.0.2:800 -> 10.0.0.10:2049
UDP TTL:64 TOS:0x0 ID:6127 IpLen:20 DgmLen:4284
Len: 4264
0x0000: 00 40 F4 53 31 47 00 50 BA CB 44 F8 08 00 45 00  .@.S1G.P..D...E.
0x0010: 10 BC 17 EF 00 00 40 11 3E 37 0A 00 00 02 0A 00  ......@.>7......
0x0020: 00 0A 03 20 08 01 10 A8 60 EC A8 E0 6D A8 00 00  ... ....`...m...
.                .                .                .                .
0x0180: 43 2D 26 2D 43 43 43 43 43 43 43 43 43 43 43 43  C-&-CCCCCCCCCCCC
0x0190: 43 43 43 43 43 43 43 43 43 43 43 43 43 43 43 43  CCCCCCCCCCCCCCCC
0x01A0: 43 43 43 43 43 43 43 43 43 43 43 43 43 43 43 43  CCCCCCCCCCCCCCCC
0x01B0: 43 43 43 43 43 43 FF C0 00 11 08 03 20 02 23 03  CCCCCC...... .#.
.                .                .                .                .
0x10A0: 30 EC 1B 39 E1 56 1C 03 72 31 DB BA B1 98 A7 38  0..9.V..r1.....8
0x10B0: 59 E8 16 99 46 66 12 40 E7 D8 2A A4 C0 F5 1C F4  Y...Ff.@..*.....
0x10C0: 0B 41 1B 41 27 0E F9 59 9D 65                    .A.A'..Y.e
```

Correlations

Snort shellcode rules as being a False Alarms
http://thing.fwsystems.com/build/sburns/Kyle_Haugsness_GCIA.doc

Also talks about shellcode rules as False / Alarm
http://www.giac.org/practical/chris_kuethe_gcia.html

Snort high false positive alert / Exploit code
http://www.giac.org/practical/Wade_Dauphinee_GCIA.doc

Evidence of Active Targeting

There is no active targeting. The shellcode alerts in snort can cause many false positives: The x86 NOP can found in daily traffic. All it requires is a data that contains the same content, and the rule will trigger. JPG, GIF, PNG, PICT or Document (MSWord etc.) will trigger this rule.

In the mysql out put from the noop table, there is a count of 548 alerts with the same source and destination address. This is not active targeting.

Alert count for Source IP

Source IP	Destinatio IP	Count
63.111.48.133	207.166.87.157	548
66.220.44.31	207.166.87.157	54
207.188.7.150	207.166.87.157	39
152.3.183.67	207.166.87.157	38
129.118.2.10	170.129.50.120	34

In the mysql out put from the noop table, the minutes and seconds are too close together and fit a pattern. I doubt an attacker would bother to keep this pattern.

Alert listing for Source and Destination IP

Date	Hour	Min.	Sec.	Source IP	Destination IP
11/01/04	2	56	1.12	63.111.48.133	207.166.87.157
11/01/04	2	56	1.72	63.111.48.133	207.166.87.157
11/01/04	2	56	1.73	63.111.48.133	207.166.87.157
11/01/04	2	56	1.91	63.111.48.133	207.166.87.157
11/01/04	3	11	1.92	63.111.48.133	207.166.87.157
11/01/04	3	11	2.98	63.111.48.133	207.166.87.157
11/01/04	3	11	3.5	63.111.48.133	207.166.87.157
11/01/04	3	11	3.85	63.111.48.133	207.166.87.157
11/01/04	3	41	19.91	63.111.48.133	207.166.87.157
11/01/04	3	41	19.96	63.111.48.133	207.166.87.157
11/01/04	3	41	19.98	63.111.48.133	207.166.87.157
11/01/04	3	41	20	63.111.48.133	207.166.87.157

The IP addresses relate to software companies communicating. This

is not active targeting for an shellcode attack. False alarms are generated between the companies exchanging pictures or documents.

IP Address – Whois Information				
Source IP		Destination IP		Count
63.111.48.133	UUNET Technologies, Inc.	207.166.87.157	I-Link Worldwide Inc	548
66.220.44.31	Swiftcomm, Inc	207.166.87.157	I-Link Worldwide Inc	54
207.188.7.150	RealNetworks, Inc.	207.166.87.157	I-Link Worldwide Inc	39

63.111.48.133
Results:
UUNET Technologies, Inc. UUNET63 (NET-63-64-0-0-1) 63.64.0.0 - 63.127.255.255
Sybari Software, Inc. UU-63-111-48-128 (NET-63-111-48-128-1) 63.111.48.128 - 63.111.48.159

66.220.44.31
Swiftcomm, Inc SWIFTCOMM-1 (NET-66-220-32-0-1) 66.220.32.0 - 66.220.63.255
3 Jane 3JANE-SWIFT (NET-66-220-44-6-1) 66.220.44.6 – 66.220.44.63

207.188.7.150
OrgName: RealNetworks, Inc. OrgID: REAL Address: 2601 Elliott Ave City: Seattle
StateProv: WA PostalCode: 98121 Country: US

207.166.87.157
OrgName: I-Link Worldwide Inc OrgID: ILKW Address: 13751 S Wadsworth Park Dr, Suite 200
City: Draper StateProv: UT PostalCode: 84020 Country: US
NetRange: 207.166.64.0 - 207.166.111.255 CIDR: 207.166.64.0/19, 207.166.96.0/20
NetName: I-LINK3 NetHandle: NET-207-166-64-0-1 Parent: NET-207-0-0-0-0
NetType: Direct Allocation NameServer: NS.I-LINK.NET NameServer: NS1.I-LINK.NET
Comment: ADDRESSES WITHIN THIS BLOCK ARE NON-PORTABLE

Severity

(Target's Criticality + Lethality of Attack) - (System Defense + NetworkDefense)

Criticality : 2 : The target is a host computer or work station, not considered critical.

Lethality : 1 : This is normal traffic, and a false alarm. It is not lethal.

System Defense : 4 : not necessary, the traffic could be blocked if needed

Network Defense : 4 : not necessary, the traffic could be blocked if needed

Severity : (2 + 1) - (4 + 4) = -5

Defense Recommendation

Determine if this NOP was part of an attack or simply part of an innocent stream of data.
Adjust shellcode rules to produce less false positives.

Multiple Choice Question

Which pointer are shellcode NOP buffer overflow attacks designed to over write?

1) Stack pointer
2) Instruction pointer
3) Return pointer
4) Heap pointer

Answer : # 3) Return Pointer

Detect 2.3
Source of Trace

The following network detection was taken from the log file 2002.10.18 located at http://www.incidents.org/logs/Raw. The logs were saved in the Tcpdump binary format. The following detect is in file 2002.10.18. The log information was generated from Snort which read the file 2002.10.18 as input.

Snort reported this alert 16 times.
[**] [1:579:2] RPC portmap request mountd [**]
[Classification: Decode of an RPC Query] [Priority: 2]
11/17-21:40:58.696507 153.33.24.3:965 -> 170.129.113.233:111
UDP TTL:113 TOS:0x0 ID:18078 IpLen:20 DgmLen:84
Len: 64

Detect was Generated By

This detect was generated by the Snort Intrusion Detection System version
snort-1.9.1 Snort produced the alerts when reading the Tcpdump binary logs using the -r option, and writing the alerts to a specified log directory using the -l option. The following command was used : snort -A full -v -r /home/snort-1.9.1/etc/raw/2002.10.18 -c /home/snort-1.9.1/etc/snort.conf -l /home/snort-1.9.1/etc/log/10.18, which produced the alert log files.

The packet triggered the alert because the mountd program number '100005' is 0x01 86 A5 00 00. The rule is looking for this content to match.

Rule(s):
alert tcp $EXTERNAL_NET any -> $HOME_NET 111 (msg:"RPC portmap request mountd"; content:"|01 86 A5 00 00|";offset:40;depth:8; reference:arachnids,13; classtype:rpc-portmap-decode; flow:to_server,established; sid:1266; rev:4;)

alert udp $EXTERNAL_NET any -> $HOME_NET 111 (msg:"RPC portmap request mountd"; content:"|01 86 A5 00 00|";offset:40;depth:8; reference:arachnids,13; classtype:rpc-portmap-decode; sid:579; rev:2;)

**] RPC portmap request mountd [**]
11/17-21:40:58.696507 153.33.24.3:965 -> 170.129.113.233:111
UDP TTL:113 TOS:0x0 ID:18078 IpLen:20 DgmLen:84
Len: 64
0x0000: 00 00 0C 04 B2 33 00 03 E3 D9 26 C0 08 00 45 00 .....3....&...E.

```
0x0010: 00 54 46 9E 00 00 71 11 35 6C 99 21 18 03 AA 81  .TF...q.5l.!....
0x0020: 71 E9 03 C5 00 6F 00 40 D1 CA 48 C8 05 B5 00 00  q....o.@..H.....
0x0030: 00 00 00 00 00 02 00 01 86 A0 00 00 00 02 00 00  ................
0x0040: 00 03 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0x0050: 00 00 00 01 86 A5 00 00 00 03 00 00 00 11 00 00  ................
0x0060: 00 00                                            ..
```

Probability Source Address was Spoofed

The source address is not spoofed. The alert triggers 16 times with the same source address. To make this attack or probe work, a response is needed.

The files used to analyze this alert are log files 2002.10.1 – 2002.10.18 located at http://www.incidents.org/logs/Raw. To gather information about this alert, I created my own conversion for snort's alert file data to a mysql ready format.

Table - Full listing for : RPC_portmap_request_mountd alert
Port 111, SUN Remote Procedure Call
Port 965, Believed to be ypbind

The IP addresses and ports do not change. The originator is looking for information from portmap on port 111 . The source address is not spoofed.

Consistent Targeting port 111 – Portmap								
Date	Hour	Min.	Sec.	Source IP	Source Port	Destination IP	Destination Port	protocol
11/17/04	21	40	58.7	153.33.24.3	965	170.129.113.233	111	UDP
11/17/04	21	40	59.52	153.33.24.3	965	170.129.113.233	111	UDP
11/17/04	21	41	1.13	153.33.24.3	965	170.129.113.233	111	UDP
11/17/04	21	41	4.33	153.33.24.3	965	170.129.113.233	111	UDP
11/17/04	21	41	33.71	153.33.24.3	965	170.129.113.233	111	UDP
11/17/04	21	41	34.52	153.33.24.3	965	170.129.113.233	111	UDP
11/17/04	21	41	36.13	153.33.24.3	965	170.129.113.233	111	UDP
11/17/04	21	41	39.33	153.33.24.3	965	170.129.113.233	111	UDP
11/17/04	21	42	43.37	153.33.24.3	965	170.129.113.233	111	UDP

Description of Attack

This attack flows just like the alert states ' RPC portmap request mountd'. Remote Procedure Calls (RPC) was developed by Sun Microsystems for unix machines. RPC consist of a library of routines for remote procedure calls. The routines give C programmers access to use the routines, and make procedure calls across the network on other systems. RPC also functions in a client/ server mode. The client calls a procedure, which sends a packet to the server. The server calls a routine to perform the requested service. A reply is sent back from the server, and information back to the client. A list of RPC calls follow. Highlighted is the target of this attack, mountd.

rpc.lockd [lockd] - start kernel lockd process

rpc.lockd [rpc] - start kernel lockd process
rpc.mountd [mountd] - NFS mount daemon
rpc.mountd [rpc] - NFS mount daemon
rpc.nfsd [nfsd] - NFS server process
rpc.nfsd [rpc] - NFS server process
rpc.rquotad [rquotad] - remote quota server
rpc.statd [rpc] - NSM status monitor
rpc.statd [statd] - NSM status monitor
rpc.yppasswdd [rpc] - NIS password update daemon
rpc.ypxfrd [rpc] - NIS map transfer server

RPC includes functions that communicate with portmap running on both client and server. Portmap connects RPC program numbers to TCP/IP port numbers. This function interfaces with portmap service, and can provides information in this attack:

pmap_getport() - returns the port number on which waits a service.
Ex: mountd, port 32769 / TCP

Portmap assigns TCP/IP port numbers for RPC and other programs, and is required in order to use RPC. Unique program numbers are used by portmap for identification. Specific ports may be requested for portmap to use. When the client uses RPC to make a call, the program number is used to contact portmap on the server machine. This tells the client which port to use for communication. The command pmap_dump for portmap displays its table of ports. The attacker is after port 111 to discover the mountd port 32,7769 – TCP and 32,7771 - UDP.

```
[root@localhost root]# pmap_dump
   100000    2   tcp    111  portmapper
   100000    2   udp    111  portmapper
   100024    1   udp  32768  status
   100024    1   tcp  32768  status
   100011    1   udp    812  rquotad
   100011    2   udp    812  rquotad
   100011    1   tcp    815  rquotad
   100011    2   tcp    815  rquotad
   100003    2   udp   2049  nfs
   100003    3   udp   2049  nfs
   100021    1   udp  32770  nlockmgr
   100021    3   udp  32770  nlockmgr
   100021    4   udp  32770  nlockmgr
   100005    1   udp  32771  mountd
   100005    1   tcp  32769  mountd
   100005    2   udp  32771  mountd
   100005    2   tcp  32769  mountd
   100005    3   udp  32771  mountd
   100005    3   tcp  32769  mountd
```

RPC.mountd is the Network File System (NFS) mount daemon. When a mount request is received from a NFS client, the permission is checked to see if they are available. If so, the rpc.mountd returns the handle to the directory to the client NFS.

In summary, the attacker is using portmap on port 111 to reveal the

mountd port number. If this is successful the attacker can communicate with the mount daemon. The next step is to attack the mountd at the given port. Mountd runs under root and is vulnerable to buffer overflow attacks. When an attack to mountd is successful, the attacker gains root privileges to the system. This is a layout for such an attack starting with polling portmap for information as this detect alerts.

Attack Mechanism

To show the details of this attack, the information is provided further into the packet data. Portions are highlighted to described the RPC header and data structure, as defined by RFC 1057. UDP headers are eight bytes long, and the RPC header information begins at byte 8 into the UDP payload.

```
**] RPC portmap request mountd [**]
11/17-21:40:58.696507 153.33.24.3:965 -> 170.129.113.233:111
UDP TTL:113 TOS:0x0 ID:18078 IpLen:20 DgmLen:84 Len: 64
0x0000: 00 00 0C 04 B2 33 00 03 E3 D9 26 C0 08 00 45 00  .....3....&...E.
0x0010: 00 54 46 9E 00 00 71 11 35 6C 99 21 18 03 AA 81  .TF...q.5l.!....
0x0020: 71 E9 03 C5 00 6F 00 40 D1 CA 48 C8 05 B5 00 00  q....o.@..H.....
0x0030: 00 00 00 00 00 02 00 01 86 A0 00 00 00 02 00 00  ................
0x0040: 00 03 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0x0050: 00 00 00 01 86 A5 00 00 00 03 00 00 00 11 00 00  ................
0x0060: 00 00                                            ..
```

The highlighted information is represented in the following chart.

Location	Data	Description
Bytes 8 - 11	48 C8 05 B5	RPC transaction ID
Bytes 12 – 15	00 00 00 00	RPC Call
Bytes 16 – 19	00 00 00 02	RPC Version 2
Bytes 20 – 23	00 01 86 A0	Portmap
Bytes 24 – 27	00 00 00 02	Portmap Version 2
Bytes 28 – 31	00 00 00 03	Procedure getport()
Bytes 31 – 34	00 00 00 00	Authentication data
Bytes 47 – 49	00 01 86 A5	mountd

By following the descriptions, the intent of the attack is clear. The start is from a RPC call from the attacker. The target and port is portmap 111. The procedure used is the getport() function to gather data. No authentication is used. The mountd port is the target information. Once the data is analyzed, the attacker's method is clear.

Correlations

This detect provided RPC header information to analyze the packet further.
http://cert.uni-stuttgart.de/archive/intrusions/2003/11/msg00046.html

This detect provided leads to the getport() function.
http://cert.uni-stuttgart.de/archive/intrusions/2003/01/msg00209.html

This is detect provided information about portmap program numbers and lead to other detects.
http://cert.uni-stuttgart.de/archive/intrusions/2003/07/msg00155.html

Evidence of Active Targeting

There is clear evidence of active targeting. The mysql output of the attacks reveals enough signs. The destination port numbers are the same for each attempt. How does the attacker know to use port 111 for portmap. Information from portmap is just the start. The attacker is probably prepared to attack mountd. These alerts only happen on one day, for just 3 minutes. The attacker will move on with or without port information. This targeted scan is to see if portmap is available.

Consistent Targeting port 111 – Portmap

Date	Hour	Min.	Sec.	Source IP	Source Port	Destination IP	Destination Port	protocol
11/17/04	21	40	58.7	153.33.24.3	965	170.129.113.233	111	UDP
11/17/04	21	40	59.52	153.33.24.3	965	170.129.113.233	111	UDP
11/17/04	21	41	1.13	153.33.24.3	965	170.129.113.233	111	UDP
11/17/04	21	41	4.33	153.33.24.3	965	170.129.113.233	111	UDP
11/17/04	21	41	33.71	153.33.24.3	965	170.129.113.233	111	UDP
11/17/04	21	41	34.52	153.33.24.3	965	170.129.113.233	111	UDP
11/17/04	21	41	36.13	153.33.24.3	965	170.129.113.233	111	UDP
11/17/04	21	41	39.33	153.33.24.3	965	170.129.113.233	111	UDP
11/17/04	21	42	43.37	153.33.24.3	965	170.129.113.233	111	UDP

Severity

(Target's Criticality + Lethality of Attack) - (System Defense + NetworkDefense)

Target's Criticality – This is a targeted Unix server and considered more than a work station. : 4

Lethality of Attack – If the attacker succeeds theory know, where to attack next. The portmap information itself is not lethal, but the next attack could be. : 3

System Defense – Since this is a server running portmap, the port 111 can be blocked with Iptables firewall script. : 4

Network Defense – The network has the capability to block port 111 at firewalls and routers : 4

The severity for this detection is : 1 – 0 = 1

Defense Recommendation

There are a number of defense positions to take against this type of attack. It is a good idea to disable RPC services that are not needed. Firewalls and routers can block and filter IP packets with destination port 111. The unix server can block and filter IP addresses targeting portmap. Access Control Lists (ACLs) can deny or permit IP addresses bound for portmap. It's always a good idea to upgrade and patch the systems.

Multiple Choice Question

The portmap daemon's purpose is to regulate?

1) numbers over 1024
2) IP Source Addresses
3) TCP/IP ports numbers
4) IP Destination Addresses

Answer : # 3) TCP/IP port numbers

Part 3 - "ANALYZE THIS"

Executive Summary

An analysis of the Intrusion Detect data downloaded are provided in this report. The findings are noted and require your immediate attention, as the condition may grow worse. The following are overall points of the condition of the MY.NET network.

Internal hosts have are infected with the Red Worm Virus. This Virus is spread throughout the network.

False positive detect are triggered when transferring files and/or documents.

Indications of the Windows Update Virus have compromised hosts.

Internal hosts are attempting to spread viruses throughout the network and to other networks on the internet.

Attackers are attempting and may be successful in gaining remote access and information internal machines.

There are two important actions to take.

- Apply defensive recommendations provided
- Verify suspicious activity on network

Log List

The logs for analysis have been downloaded from http://incidents.org/logs

alert.040116	scans.040116	oos_report_040116.txt
alert.040117	scans.040117	oos_report_040117.txt
alert.040118	scans.040118	oos_report_040118.txt
alert.040119	scans.040119	oos_report_040119.txt
alert.040120	scans.040120	oos_report_040120.txt

3.0 The MY.NET network is analyzed using the alerts, scans, and oos files. The physical network is not accessible, the results are from analysis of logs for this network. Potential HTTP, DNS, and SMTP servers are identified with analysis of logs. The scans and OOS files are analyzed to identify the most active traffic. The network traffic for MY.NET.30.4 has only 1 alert message. A further look is taken to identify some of the potential traffic for this address. Private addresses have triggered alerts when connecting to external addresses. The private addresses are identified with the related alerts.

Web Servers : Given the logs of alerts, the best way to gage how many http services are running is to use the Red Worm virus. The Red Worm initiates using port 80 and tries to communicate on port 65535. This worm is spread throughout the MY.NET network. By tracing the traffic of the worm according to the alerts file provided, an idea of possible http services can be calculated. Before using the alerts file, the scans file produces results for port 80. The following scans table shows the results for the destination address MY.NET.* with port 80 as the destination port.

Scans : MY.NET port 80

Destination IP	Destination Port	Type	Count
MY.NET.34.11	80	SYN	40
MY.NET.24.44	80	SYN	35
MY.NET.6.7	80	SYN	31
MY.NET.5.95	80	SYN	11
MY.NET.24.34	80	SYN	10
MY.NET.60.14	80	SYN	9
MY.NET.29.66	80	SYN	5
MY.NET.6.14	80	SYN	2
MY.NET.153.149	80	UDP	2
MY.NET.25.68	80	SYN	1

The MY.NET addresses from this table are cross referenced with the alert file to see if the Red Worm has infected these host.

Alerts for MY.NET Scans table

Source IP	Source Port	count	Alert
MY.NET.24.34	80	46	High port 65535 tcp - possible Red Worm - traffic
MY.NET.34.11	65535	45	High port 65535 tcp - possible Red Worm - traffic
MY.NET.25.68	65535	26	High port 65535 tcp - possible Red Worm - traffic
MY.NET.24.44	80	7	High port 65535 tcp - possible Red Worm - traffic
MY.NET.6.7	80	3	Possible trojan server activity
MY.NET.60.14	65535	3	High port 65535 tcp - possible Red Worm - traffic
MY.NET.5.95	65535	2	High port 65535 tcp - possible Red Worm - traffic
MY.NET.29.66	65535	2	High port 65535 tcp - possible Red Worm - traffic
MY.NET.6.14	65535	1	High port 65535 tcp - possible Red Worm - traffic

Given the data from both files I would say that this is just the start of http services. When listing all of the MY.NET addresses that triggered the Red Worm alert, the number is higher. The same criteria is used in the above table with the MY.NET machines as the source address. The count for MY.NET addresses triggering the Red Worm alert is 1512. This number gives a good idea of potential http services.

DNS Servers :

The DNS servers will show traffic on port 53. The scans file is referenced to see the possible connections to port 53. The listing of portscan messages separated from the alerts files, shows the port usage between tcp and udp port scans.

Port 53 from Portscans table

MY.NET IP	tcpp	udpp	count
MY.NET.1.3	0	53	150
MY.NET.150.210	53	0	26
MY.NET.34.14	53	0	12
MY.NET.53.219	5	53	12
MY.NET.1.4	0	53	12
MY.NET.84.164	0	53	5
MY.NET.97.96	0	53	1
MY.NET.97.241	3	53	1
MY.NET.97.53	53	0	1

The first IP in the table MY.NET looks like a good target for a DNS server. The IP MY.NET.1.3 stands out among the other addresses. This list is cross referenced in the alerts file to see if any DNS related alerts are triggered for the MY.NET addresses.

MY.NET table for destination port 53 with Alerts

Destination IP	Destination Port	Count	Alert
MY.NET.1.3	53	191	NMAP TCP ping!
MY.NET.1.4	53	59	NMAP TCP ping!
MY.NET.1.5	53	21	NMAP TCP ping!
MY.NET.12.2	53	8	Null scan!
MY.NET.32.4	53	4	NMAP TCP ping!
MY.NET.54.204	53	2	NMAP TCP ping!
MY.NET.111.114	53	2	NMAP TCP ping!
MY.NET.72.193	53	1	High port 65535 udp - possible Red Worm - traffic
MY.NET.83.109	53	1	Null scan!

The top three MY.NET addresses stand out as potential DNS serves. The alert 'NMAP TCP Ping!' occurs as many as 271 times for the three targeted addresses. DNS servers are often scanned and attacked. The absence of the Red Worm alert shows that the machines are most likely not running http services, and these are dedicated DNS servers.

SMTP Servers : The SMTP protocol for port 25 is checked against the scans file for any activity. The following table shows the results for port 25.

Scan table : MY.NET for port 25

Destination IP	Destination Port	Count	Type
MY.NET.12.6	25	321	SYN
MY.NET.153.149	25	9	UDP
MY.NET.69.253	25	3	UDP
MY.NET.75.13	25	2	SYN
MY.NET.60.17	25	1	SYN
MY.NET.25.66	25	1	INVALIDACK

The address MY.NET.12.6 has a SYN scan count of 321. This count stands out among the others. The process of cross referencing the above address for related alerts is in the following table.

Alerts for MY.NET Destination port 25

Destination IP	Destination Port	Count	Alert
MY.NET.12.6	25	175	TCP SMTP Source Port traffic
MY.NET.60.39	25	29	NMAP TCP ping!
MY.NET.12.2	25	8	Null scan!
MY.NET.30.3	25	3	MY.NET.30.3 activity
MY.NET.30.4	25	3	MY.NET.30.4 activity
MY.NET.153.18	25	2	Null scan!
MY.NET.60.17	25	1	NMAP TCP ping!

The address MY.NET.12.6 appears again with the count of 175. The related alert is 'TCP SMTP Source Port Activity', The alert relates to mail service, and no appearance of the Red Worm virus. This leads me to believe that MY.NET.12.6 is a dedicated mail server.

Scan files : The next table gives us information as to what types of scans are probing the network. The highest count goes to the SYN scan. The table also shows the type of scan with related count.

Most popular Scan Types on MY.NET

Destination IP	Unique Dest. IP	Destination Port	Unique Dest. Port	Type	Count
MY.NET.12.7	16587	7070	186	SYN	121994
MY.NET.1.5	321	53	2713	UDP	4220
MY.NET.97.47	8	3123	192	INVALIDACK	292
MY.NET.97.47	3	2301	3	FIN	43
MY.NET.110.86	12	20744	27	UNKNOWN	27

The SYN scan is at the top of the list with the count of 121,994. The characteristics of this type of scan is that a large number of hosts (16,587) are scanned with a lower number of ports (186) scanned. The The next highest count is the UDP scan of 4220 counts. This scan is taking the opposite approach, fewer hosts (321) are scanned with a higher number of ports (2713) scanned.

This information is helpful for the MY.NET administrators in handling scans. A defensive plan for these scans can be implemented since the information is available. The port services should be checked in order to not respond to the source, providing network information.

The next view of the the scans file is to rate the top ports scanned. The following table shows the ports with the highest scan rates.

Top Services scanned for attacks

Scan type	Destination port	Description	Count
SYN	6129	Dameware Remote Admin.	86752
SYN	20168	Worm tftp transfer	20116
SYN	7070	ARCP	8143
SYN	135	DCE Endpoint resolution	3266
UDP	0	Reserved / NULL	381
SYN	25	SMTP	337
SYN	110	Post Office Protocol	313
UDP	137	Netbios Name Server	241
UDP	3356	UPNOTIFYPS	170

The port 6129 has a count of 86,752. This port relates to the Dameware remote administration software. There are vulnerabilities in older versions of this software, which allow unauthorized logins. This software is installed by some viruses as a back door on infected systems. The port 20168 is related to the Windows Update virus. This worm uses this port to transfer itself via tftp file transfers. The 2 highest scanned ports both relate to security breaches. This is valuable information in what to guard against. Anti-virus software should be installed and maintained.

OOS Files : The OOS files are checked for the highest count of traffic to the destination address. The following table shows the results for oos table.

OOS Destination port count

Destination IP	Destination Port	Count
MY.NET.6.7	110	1616
MY.NET.12.6	25	1300
MY.NET.69.217	1426	493
MY.NET.24.44	80	456
MY.NET.24.34	80	261

The table shows that MY.NET.6.7 has the highest count of 1616 to port 110. Port 110 is the Post Office Protocol Version 3 (POP3). POP3 is used in conjunction with SMTP. This address may be another SMTP server. The next count is for the potential SMTP server MY.NET.12.6 with 1300 hits. Both machines have related services. The the following table, MY.NET.6.7 machine is checked for alerts .

Alerts for MY.NET IP from OOS table

Destination	Destination port	Alert
MY.NET.6.7	80	Possible trojan server activity
MY.NET.6.7	65535	High port 65535 tcp - possible Red Worm - traffic

The alerts show that MY.NET.6.7 has 2 types of alerts related to http. I would consider this machine compromised, and look further into the mail servers. The OOS files show that attention is needed in safeguarding the mail server for MY.NET network.

Insights into internal machines :

Many alerts are generated for outgoing traffic for the MY.NET network. The first alert for discarded fragments has the count of 10506. There are only 9 machines triggering this alert. This alert can be isolated and solved. The 2nd and 3rd alerts show a number of systems are compromised with the SMB Name wildcard and the Red Worm viruses. This will take a good defensive strategy to cure these alerts. The following table show the frequency of alerts.

Internal traffic alerts to External IP addresses

Source IP	Unique Src IP	Source Port	Alert	Count
MY.NET.42.9	9	0	Incomplete Packet Fragments Discarded	10506
MY.NET.153.153	146	137	SMB Name Wildcard	4496
MY.NET.25.71	1512	65535	High port 65535 tcp - possible Red Worm - traffic	2771
MY.NET.69.198	1	1961	TFTP - Internal TCP connection to external tftp server	1711
MY.NET.112.152	16	4662	Possible trojan server activity	171
MY.NET.84.216	3	6257	High port 65535 udp - possible Red Worm - traffic	25
MY.NET.111.34	4	5900	RFB - Possible WinVNC - 010708-1	7

MY.NET.30.4 Traffic :

Traffic for the alerts MY.NET.30.4 is high on the list for counts. The files are sanitized to protect private information. I expect various alerts would trigger for this address. The alerts for IP MY.NET.30.4 are copied with the text ' MY.NET.30.4 Activity'. I will analyze this machine with another approach of frequency of ports used.
The following table is a listing of the destination ports used for MY.NET.30.4.

Inbound My.NET.30.4 Alerts

Destination IP	Destination Port	Count	Alert
MY.NET.30.4	51443	34679	MY.NET.30.4 activity
MY.NET.30.4	524	4724	MY.NET.30.4 activity
MY.NET.30.4	80	2563	MY.NET.30.4 activity
MY.NET.30.4	8009	2175	MY.NET.30.4 activity
MY.NET.30.4	6129	182	MY.NET.30.4 activity
MY.NET.30.4	4000	33	MY.NET.30.4 activity
MY.NET.30.4	4899	17	MY.NET.30.4 activity
MY.NET.30.4	1257	10	MY.NET.30.4 activity

Port 51443 is the highest count which relates to iFolder Novell. Port 524 relates to NCP. Now that the most active ports are listed, the ports are compared to the alerts they generate. The following table shows the alerts related to the port in the above table.

Alerts related to MY.NET.30.4 traffic

Destination Port	Alert	Count
80	EXPLOIT x86 NOOP	1150
80	NIMDA - Attempt to execute cmd from campus host	3
80	TCP SRC and DST outside network	125
80	NMAP TCP ping!	104
80	High port 65535 tcp - possible Red Worm - traffic	94
80	Possible trojan server activity	36
80	Null scan!	8
1257	TFTP - Internal TCP connection to external tftp server	1
4000	High port 65535 tcp - possible Red Worm - traffic	3
4899	High port 65535 tcp - possible Red Worm - traffic	2
6129	NMAP TCP ping!	260
6129	EXPLOIT x86 NOOP	58
6129	High port 65535 tcp - possible Red Worm - traffic	3
6129	Null scan!	2
6129	Possible trojan server activity	1

This gives a guide as what to investigate for machine MY.NET.30.4, even though the alerts are sanitized.

Private Addresses :
The private addresses reaching external addresses is not of the highest of counts. An alert is generated when this traffic reaches an external address. Firewalls and routers can block this activity. If this is legitimate traffic via proxy or (NAT) translation, this alert could be a false positive.

Private IP address from MY.NET Network to External IP

Source IP	Destination IP	Alert	count
192.168.1.103	64.152.73.205	TCP SRC and DST outside network	45
192.168.1.100	64.152.73.206	TCP SRC and DST outside network	15
192.168.28.1	218.69.247.6	TCP SRC and DST outside network	13
169.254.204.76	220.88.26.23	TCP SRC and DST outside network	3
192.168.1.102	65.54.202.254	TCP SRC and DST outside network	2
192.168.0.32	212.58.240.141	TCP SRC and DST outside network	1

4.0 Summary of Alerts

The Following alerts are summarized by the count of unique alerts on the MY.NET network. The number of different source and destination addresses are calculated.

Prioritized Alert table for MY.NET network

Alert	Unique_src_IPs	Unique_dst_IPs	Alerts
MY.NET.30.4 activity	318	2	44438
High port 65535 tcp - possible Red Worm - traffic	1568	10486	17060
Incomplete Packet Fragments Discarded	73	1054	12028
MY.NET.30.3 activity	184	2	10916
SMB Name Wildcard	146	395	4496
TFTP - Internal TCP connection to external tftp server	3	3	3339
connect to 515 from outside	1	1	3076
EXPLOIT x86 stealth noop	12	11	2498
EXPLOIT x86 NOOP	321	96	2031
SUNRPC highport access!	23	30	1122
External RPC call	2	260	900
NMAP TCP ping!	142	162	817
[UMBC NIDS IRC Alert] IRC user /kill detected, possible tro	35	29	759
Null scan!	56	93	744
TCP SRC and DST outside network	43	77	317
Possible trojan server activity	40	42	315
High port 65535 udp - possible Red Worm - traffic	25	21	150
ICMP SRC and DST outside network	44	0	120
SMB C access	28	3	103
FTP passwd attempt	66	1	88

4.1 Alert Name : High port 65535 tcp - possible Red Worm - traffic Probability source address was spoofed

To send the buffer overflow attack code, the TCP 3-way handshake must complete. The source IP address is not spoofed. The worm starts with a TCP connection on port 80.

Description of Attack

The Red Worm targets Windows systems, and some Cisco products. The software target is IIS 4.0 or 5.0. The Red Worm attempts to connect to TCP port 80 to send a HTTP Get request to the responding host. The exploit is a buffer overflow in the indexing service. Infection can cause a denial of service (DoS) attack and possible full remote access. This is a self-propagating worm and infected systems will search random IP's for another TCP port 80 to for connect. A back door is set up using port 65535. Traffic on port 65535 is the sign that the Red Worm has infected targeted systems.

Example of attack mechanism from the attackers machine :

GET /default.ida?
>NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
N.......X.. >%u9090%u6588%ucbd3%u7801%u9090%u6588%ucbd3%u7801%
>u990%u9090%u00c%u53ff%u0078%u000%u00

High port 65,535 Alerts						
Source IP	Distinct Src IP	Source Port	Destination IP	Distinct Dst IP	Destination Port	alerts
MY.NET.109.1	1506	65535	128.171.198.49	2938	1248	2294
MY.NET.25.68	8	65535	69.6.61.10	25	1	79
193.63.229.36	5	25	MY.NET.24.20	65535	1	53
64.68.82.164	1	65535	MY.NET.34.11	80	1	39
MY.NET.34.11	1	80	64.68.82.164	65535	1	36

The mysql table out put shows that port 65535 has traffic in both directions. The host machines are infected. MY.NET has at least 1506 different source addresses with machines running http sending traffic out to IP 128.171.198.49 for a total of 2294 alerts. There is traffic from the port 65535 and port 80 for IPs MY.NET.34.11 and 64.68.82.164. This host MY.NET.34.11 is infected with traffic running on the back door port 65535.

High port 65,535 Alerts for IP 128.171.198.49

Source IP	Source Port	Destination IP	Destination Port	Alerts
MY.NET.111.38	65535	128.171.198.49	3486	16
MY.NET.110.172	65535	128.171.198.49	3365	12
MY.NET.111.31	65535	128.171.198.49	3479	12
MY.NET.153.148	65535	128.171.198.49	3316	12
MY.NET.153.152	65535	128.171.198.49	3320	12
MY.NET.153.163	65535	128.171.198.49	3331	12
MY.NET.110.229	65535	128.171.198.49	3422	10
MY.NET.110.235	65535	128.171.198.49	3428	10
MY.NET.110.202	65535	128.171.198.49	3395	10
MY.NET.111.30	65535	128.171.198.49	3478	10

Various MY.NET machines are sending traffic to IP 128.171.198.49. Further analysis of the destination ports shows the definite intention of the back door traffic. Each destination port is assigned and is registered for a particular program. The following is a list of the destination ports and their related programs:

Port - Description
3486 - IFSF Heartbeat Port, 3365 - Content Server , 3479 - 2Wire RPC
3316 - AICC/CMI , 3320 - Office Link 2000, 3331 - MCS Messaging
3422 - Remote USB System Port, 3428 - 2Wire CSS,3395 - Dyna License Manager (Elam),
3478 - Simple Traversal of UDP Thr

The ports listed are targeted from the back doors on the MY.NET systems. The Organization name for IP 128.171.198.49 is the University of Hawaii. I believe that this university has the Red Worm infection as well. The following is the whois information :

Results: 128.171.198.49
OrgName: OrgID: UNIVER-25Address: 2565 The MallCity: Honolulu
StateProv: HI PostalCode: 96822 Country: US
NetRange: 128.171.0.0 - 128.171.255.255 CIDR: 128.171.0.0/16
NetName: HAWAII NetHandle: NET-128-171-0-0-1
Parent: NET-128-0-0-0-0 NetType: Direct Assignment
NameServer: DNS1.HAWAII.EDU NameServer: DNS2.HAWAII.EDU
Comment: RegDate: 1988-06-06 Updated: 2000-10-25

Correlation

The signature on the CERT web page, to define the mechanics of the exploit.
http://www.cert.org/incident_notes/IN-2001-9.html

One of the attack mechanism, for Red Worm
http://cert.uni-stuttgart.de/archive/intrusions/2002/10/msg00128.html

Determining network signs of infection
http://is.rice.edu/~glratt/practical/Glenn_Larratt_GCIA.html#p65535RW

Recommendation / Defense

Defensive recommendations are to update anti-virus software , and related patches to network devices. Turn off unused service ports, and block port 65535 at the firewall and/or border router.

4.2 Alert Name : Incomplete Packet Fragments Discarded Probability source address was spoofed

Fragment attacks intending to cause Denial of Service (DoS) would most likely have spoofed addresses. Do to the numerous sent in one day, and the whois information for the source IP address, I think the address is not spoofed.

Description of Attack

The incomplete packet fragments determine that not all packet fragments arrived and the packet could not be re-assembled. Some reasons for this are transmission errors, broken stacks and fragment attacks. Possible misconfiguration of network devices and routers corrupting packets could generate this alert. Crafted packets would trigger this alert as well. The following table list the frequency for this alert.

Incomplete Packet Fragments Table

Source IP	Source port	Destination IP	Distinct Dst IP	Destination Port	Alerts
192.0.0.60	0	MY.NET.12.3	1003	0	1003
69.44.118.145	0	MY.NET.153.149	2	0	332
MY.NET.42.9	0	130.167.237.15	1	0	52
66.167.234.245	0	MY.NET.12.2	1	0	40
141.157.85.107	0	MY.NET.11.4	1	0	24

The IP 192.0.0.60 has a count of 1003 alerts distributed over 1003 MY.NET machines. This is a one-to-one ratio for packet to machine. It is unclear as what the
sender is trying to accomplish. The next table looks at the frequency of alerts.

Date/Time Incomplete Packet Fragments : 192.0.0.60

Date / Time	Source IP	Source Port	Destination IP	Destination Port	Alerts
01/20/04 12:42 pm	192.0.0.60	0	MY.NET.5.164	0	1
01/20/04 12:43 pm	192.0.0.60	0	MY.NET.6.11	0	1
01/20/04 12:43 pm	192.0.0.60	0	MY.NET.6.9	0	1
01/20/04 12:52 pm	192.0.0.60	0	MY.NET.31.109	0	1
01/20/04 12:53 pm	192.0.0.60	0	MY.NET.34.15	0	1
01/20/04 12:57 pm	192.0.0.60	0	MY.NET.43.36	0	1
01/20/04 01:01 pm	192.0.0.60	0	MY.NET.53.179	0	1
01/20/04 01:01 pm	192.0.0.60	0	MY.NET.54.48	0	1

All alerts are on the same day and are continuous. This a an attack with 1 packet targeting a MY.NET destinations. I don't believe that this is an effective DoS attack.
Whois Results : 192.0.0.60
Internet Assigned Numbers Authority RESERVED-192 (NET-192-0-0-0-1)
192.0.0.0 - 192.0.127.255
Internet Assigned Numbers Authority ROOT-NS-LAB (NET-192-0-0-0-2)
192.0.0.0 - 192.0.0.255

Correlation

Reasons for using packet fragmentation
http://is.rice.edu/~glratt/practical/Glenn_Larratt_GCIA.html

Using port 0, not effective for attacks
http://www.lurhq.com/idsanalysis.pdf

Recommendation / Defense

My recommendation for this alert is to block port 0. I would them look for the same or similar alert with another port address for further attacks. The source address may reappear with a different alert.

4.3 Alert Name : SMB Name Wildcard

Probability source address was spoofed

The source address is not spoofed. This attack is for reconnaissance which requires a response back to the source address.

Description of Attack

The Network Basic Input/Output System (Netbios) functions as a client server relationship. This service is used in Windows Systems on port 137. The Netbios name service resolves IP addresses into Netbios names. Three services provided are are datagram service, session service, and name service. The Server Message Block (SMB) is scanned on port 137. An Example of the attack mechanism follows. This is an captured packet by the Snort intrusion detection system.

```
[**] SMB Name Wildcard [**]
05/10-18:08:05.359797 Src.net.com:137 ->  Dst.net.com:137
UDP TTL:119 TOS:0x0 ID:45361 Len: 58
00 D4 00 00 00 01 00 00 00 00 00 00 20 43 4B 41 ..... CKA
41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAA
41 41 41 41 41 41 41 41 41 41 41 41 41 41 00 00 21 AAAAAAAAAAAAAA..!
00 01 ..
```

The MY.NET network has a high degree of SMB traffic. The traffic in this case is outgoing from the MY.NET network. The following table shows the alert counts from unique MY.NET hosts.

SMB Name Wildcard Alerts

Source IP	Source Port	Distinct Src port	Destination Port	Distinct Dst IP	Destination Port	Alerts
MY.NET.150.	1069	6	207.89.248.	157	137	820
MY.NET.153.	1049	2	209.209.46.	157	137	804
MY.NET.150.	1080	6	207.89.248.	154	137	804
MY.NET.75.1	137	1	65.110.13.6	136	137	560
MY.NET.190.	137	1	218.189.234	30	137	168
MY.NET.109.	137	1	211.202.69.	65	137	155
MY.NET.190.	137	1	218.189.234	19	137	58
MY.NET.189.	137	1	61.137.93.5	2	137	24
MY.NET.99.3	137	1	169.254.45.	3	137	23
MY.NET.112.	137	1	169.254.45.	1	137	22

The traffic is outgoing which show that an infection has already taken place. The probing for port 137 service is guided to external IP addresses. Privileged ports to probe for port 137. The following table points out source ports used by MY.NET Windows hosts.

SMB Name wildcard Alerts : MY.NET hosts : Source port number

Source IP	Source Port	Destination IP	Distinct Dst IP	Destination Port	Alerts
MY.NET.150.44	1063	67.70.94.240	46	137	183
MY.NET.150.44	137	218.156.198.13	64	137	168
MY.NET.150.44	1053	212.210.199.10	34	137	149
MY.NET.150.44	1054	141.156.69.66	42	137	149
MY.NET.150.44	1056	216.205.95.134	40	137	144
MY.NET.150.44	1069	207.89.248.3	8	137	27
MY.NET.153.21	1049	209.209.46.50	154	137	640
MY.NET.153.21	137	218.156.198.13	61	137	164
MY.NET.150.198	1102	213.82.24.100	46	137	189
MY.NET.150.198	1072	68.112.250.127	41	137	171
MY.NET.150.198	1092	213.224.225.47	35	137	160
MY.NET.150.198	137	194.185.90.248	58	137	158
MY.NET.150.198	1073	12.135.75.152	33	137	89
MY.NET.150.198	1080	207.89.248.3	10	137	37

The traffic from source port 137 to destination port 137 is considered normal. The use of privileged ports not related to Netbios is unclear. This does mean that administrator privileges have been acquired to use of these ports. This traffic also shows that there is a high number of Windows hosts in the network and infected.

Whois : 67.70.94.240 Results:
Bell Canada BELLNEXXIA-11 (NET-67-68-0-0-1)
67.68.0.0 - 67.71.255.255

Bell sympatico BELL190428-CA (NET-67-70-92-0-1)
67.70.92.0 - 67.70.95.255

Correlation

Good summary, background, increase in use
http://www.sans.org/resources/idfaq/port_137.php

Attack mechanism via ISAKMP port 500
http://cert.uni-stuttgart.de/archive/intrusions/2002/10/msg00074.html

SMB port 137 usage signals
http://is.rice.edu/~glratt/practical/Glenn_Larratt_GCIA.html

Good background for SMB
http://www.whitehats.ca/main/members/Herc_Man/Files/Al_Williams_GCIAPractical.pdf

Recommendation / Defense

The first port to block is port 137, to stop this traffic. The protocol suite is port 135 to port 139. These ports should also be blocked as well. Any trusted hosts
allowed for Netbios should be configured by the firewall.

4.4 Alert Name : TFTP - Internal TCP connection to external tftp server

Probability source address was spoofed

The source IP address is not spoofed. The connection requires a 3-way hand shake. There is continuous traffic between MY.NET.24.15 and 69.10.132.121.

Description of Attack

The port 69 for TFTP seems normal when used as the source and destination ports. There are 1961 different ports for MY.NET.69.198 which triggered alerts, when used as source and destination ports. Further analysis of tables show the port numbers range from 1000 to 4999. The IP 69.10.132.121 is only using port 69.

TFTP - Internal TCP connection to external tftp server

Source IP	Source Port	Distinct Src port	Destination IP	Destination Port	Distinct Dst port	Alerts
MY.NET.69.198	1961	924	69.10.132.121	69	1	1711
69.10.132.121	69	1	MY.NET.69.198	1964	910	1627

The connections to ports other than 69 are unclear. It is possible that

exploit was transferred to the host and is sending traffic out on various port numbers. The destination for the port numbers is still the static IP 69.10.132.121 port 69. Further investigation of real time packets, and the host MY.NET.69.198 is needed. The dates for the TFTP traffic falls only on 1 day, 01/17 . This seems to be traffic from the attacker. The external IP 69.10.132.121 was cross referenced for other alerts. If this attacker gained access to this host, there could be signs of other attacks. No hits other than the TFTP alerts described. The Scans table was cross referenced as well. There were no hits for this IP. This MY.NET.69.198 should be checked for compromises.

Whois - Results: IP - 69.10.132.121
RackForce Hosting Inc. RACKFORCE-1 (NET-69-10-128-0-1)
69.10.128.0 - 69.10.159.255
Memset Ltd. MEMSET-MAINNET (NET-69-10-132-0-1)
69.10.132.0 - 69.10.132.255

Correlation

TFTP – external UDP connection to internal tftp server
http://www.giac.org/practical/GCIA/Doug_Kite_GCIA.pdf

Reference to Trojan communications via TFTP.
http://www.whitehats.ca/main/members/Herc_Man/Files/Al_Williams_GCIAPractical.pdf

Cross reference of TFTP connections and RPC buffer overflows
http://www.users.globalnet.co.uk/~mlewis/Downloads/David_M_Lewis_GCIA_pdf.pdf

Recommendation / Defense

The first issue is to block port 69 for TFTP traffic. For file transfers, another method using authentication should be used instead. Ex: Secure Shell or FTP. The TFTP connection is not secure.

4.5 Alert Name : connect to 515 from outside

Probability source address was spoofed

A TCP connection to port 515 is required for this connection. The source address would not be spoofed, in order to receive print information.

Description of Attack

The port 515 is commonly used for the LPD daemon for UNIX systems. According to RFC1179 the printer listening port 515 is for tcp connections. TCP ports 721 and 731 are the required source ports for the host machine. It is possible to change the port numbers and still execute the printing service. The following table shows one IP connecting with various ports to a MY.NET host port 515.

connect to 515 from outside

Source IP	Source Port	Destination IP	Destination Port	Alert
68.32.127.158	52133	MY.NET.24.15	515	1243
68.32.127.158	52185	MY.NET.24.15	515	907
68.32.127.158	53121	MY.NET.24.15	515	253
68.32.127.158	54001	MY.NET.24.15	515	220
68.32.127.158	54028	MY.NET.24.15	515	145
68.32.127.158	53948	MY.NET.24.15	515	75

IP 68.32.127.158 connects with a number of ports 52000 and higher. This is not the required ports 721 and 731. The dates of the alerts are as follows. 01/17 has 2166 alerts, 01/19 has 226 alerts, and 01/20 has 694 alerts. The ports numbers were checked against any other alerts. There were no hits for these ports. I believe this to be normal traffic given the various source ports. The port numbers above 52000 should be checked.

Results : 68.32.127.158
Comcast Cable Communications, Inc. JUMPSTART-1 (NET-68-32-0-0-1)
68.32.0.0 - 68.63.255.255
Comcast Cable Communications, Inc. BALTIMORE-A-2 (NET-68-32-112-0-1)
68.32.112.0 - 68.32.127.255

Correlation

Considered the 515 traffic normal
http://is.rice.edu/~glratt/practical/Glenn_Larratt_GCIA.html

Short Description of port 515 usage
http://members.cruzio.com/~jeffl/sco/lp/printservers.htm

Explanation of ports 515, 721, and 731 relationship
http://www.lprng.com/LPRng-Reference-Multipart/rfc1179ref.htm

BACKDOOR Q access connected to port 515
http://www.users.globalnet.co.uk/~mlewis/Downloads/David_M_Lewis_GCIA_pdf.pdf

Recommendation / Defense

Determine the relevance of the higher port numbers. Check the server MY.NET.24.15 to see if it is compromised.

4.6 Alert Name : EXPLOIT x86 stealth noop

Probability source address was spoofed

I believe the source address is not spoofed because of the high count of alerts to one IP address. A spoofed source address would make sense if this

is a Denial of Service (DoS) attack. I think this is traffic for files or documents, which create false positives under Snort rules for this type of alert. In addition the whois for the external IP is the National Aeronautics and Space Administration.

Description of Attack

This attack is presented as a buffer overflow exploit. The attacker is sending machine language code for the host computer to execute as commands. The idea is to overwrite the return pointer for the stack, to then execute your command. Snort has the following rule i the shellcode.rules file.

:alert ip $EXTERNAL_NET any -> $HOME_NET $SHELLCODE_PORTS (msg:"SHELLCODE x86 stealth NOOP"; content:"|eb 02 eb 02 eb 02|"; reference:arachnids,291; classtype:shellcode-detect; sid:651; rev:5;)

The series of 'eb 02' are used to move the return pointer to different addresses in the stack. This attack comes from a series of NOP instructions directed at the Intel x86 architecture. A NOP instruction means no operation is performed when the instruction is read. The attacker is trying to take advantage of coding practices in order to execute arbitrary code. The NOP instruction allows the attacker to load an address space with numerous NOPs followed by the code to execute. When the NOPs are executed, this is referred to as sledding into the attackers shellcode.

The shellcode alerts in Snort can cause many false positives: The x86 NOP can found in daily traffic. All it requires is a data that contains the same content, and the rule will trigger. JPG, GIF, PNG, PICT or Document (MS Word etc.) will trigger this rule. The following table shows the count of alerts with 1 source IP to 1 MY.NET IP.

EXPLOIT x86 stealth noop Alerts

Date / Time	Source IP	Source Port	Destination IP	Destination Port	Alerts
01/20/04 08:42 am	129.165.254.6	42861	MY.NET.162.56	33301	102
01/20/04 08:57 am	129.165.254.6	56545	MY.NET.162.56	33329	73
01/20/04 09:13 am	129.165.254.6	39096	MY.NET.162.56	33476	59
01/20/04 09:01 am	129.165.254.6	60291	MY.NET.162.56	33337	54
01/20/04 09:16 am	129.165.254.6	41820	MY.NET.162.56	33480	53
01/20/04 08:48 am	129.165.254.6	48343	MY.NET.162.56	33313	48
01/20/04 08:55 am	129.165.254.6	55029	MY.NET.162.56	33325	48
01/20/04 08:38 am	129.165.254.6	39760	MY.NET.162.56	33293	47
01/20/04 03:39 pm	129.165.254.6	56119	MY.NET.162.56	35209	47
01/20/04 09:17 am	129.165.254.6	43301	MY.NET.162.56	33482	46

The source IP 129.165.254.6 triggers alerts with MY.NET.162.56, both on various ports. Note that the date is 01/20/04. Do to the high false positives for this alert, I believe this traffic consist of files and/or documents. During my analysis I cross referenced the IP MY.NET.162.56 against any other alerts. I received 4 hits. The following table shows the results of other alerts.

High port 65535 tcp - possible Red Worm - traffic MY.NET.162.56				
Source IP	Source Port	Destination IP	Destination Port	Alert
128.171.198.49	1848	MY.NET.162.56	65535	High port 65535 tcp - possible Red Worm
MY.NET.162.56	65535	128.171.198.49	1848	High port 65535 tcp - possible Red Worm
128.171.198.49	1848	MY.NET.162.56	65535	High port 65535 tcp - possible Red Worm
MY.NET.162.56	65535	128.171.198.49	1848	High port 65535 tcp - possible Red Worm

The usage of port 65535 signals that MY.NET.162.56 is already compromised. The date for the above Red Worm traffic is 01/19/04, one day before the EXPLOIT x86 stealth noop alerts. For more information on the Red Worm alert see section 4.1. Listed below is the whois information for IPs 29.165.254.6 and 128.171.198.49

Results : 129.165.254.6
OrgName: National Aeronautics and Space Administration OrgID: NASA
Address: AD33/Office of the Chief Information Officer City: MSFC
StateProv: AL PostalCode: 35812 Country: US
NetRange: 129.165.0.0 - 129.165.255.255 CIDR: 129.165.0.0/16
NetName: NASA-GSFCSSE NetHandle: NET-129-165-0-0-1
Parent: NET-129-0-0-0-0 NetType: Direct Allocation
NameServer: NS.GSFC.NASA.GOV NameServer: NS2.GSFC.NASA.GOV
Comment: RegDate: 1988-01-04 Updated: 2002-09-05 OrgTechEmail: dns.support@nasa.gov

Results: 128.171.198.49
OrgName: OrgID: UNIVER-25 Address: 2565 The Mall City: Honolulu
StateProv: HI PostalCode: 96822 Country: US NetRange: 128.171.0.0 - 128.171.255.255
CIDR: 128.171.0.0/16 NetName: HAWAII
NetHandle: NET-128-171-0-0-1 Parent: NET-128-0-0-0-0 NetType: Direct Assignment
NameServer: DNS1.HAWAII.EDU NameServer: DNS2.HAWAII.EDU
Comment: RegDate: 1988-06-06 Updated: 2000-10-25

Correlation

Snort shellcode rules as being a False Alarms
http://thing.fwsystems.com/build/sburns/Kyle_Haugsness_GCIA.doc

Buffer overflow and x86 nop sled / false alarm
http://www.giac.org/practical/chris_kuethe_gcia.html

Snort high false positive alert / Exploit code
http://www.giac.org/practical/Wade_Dauphinee_GCIA.doc

Recommendation / Defense

Verify the NOP is part of an attack or part of an stream of data. This machine MY.NET.162.56 is comprised. It needs to be hardened. There may be

other problems generated from this machine as well. Apply all anti-virus software and upgrades to this system.

4.7 Alert Name : EXPLOIT x86 NOOP
Probability source address was spoofed

I believe this source address is not spoofed as well This Alert is similar to the EXPLOIT x86 stealth noop in section 4.6. There are a high number of alerts to one IP address again. I think this is traffic for files or documents, which creates false positives under Snort rules for this alert.

Description of Attack

This attack re-enforces the spread of this alert and other attacks. This attack is presented as a buffer overflow exploit. The attacker is sending machine language code for the host computer to execute as commands. The idea is to overwrite the return pointer for the stack, to then execute your command. Snort has many rules in the shellcode.rules file. For example 2 rules are listed below. Snort compares the content against the packet data to trigger the alert.

rules:alert tcp $EXTERNAL_NET any -> $HOME_NET 22 (msg:"EXPLOIT ssh CRC32 overflow NOOP"; flow:to_server,established; content:"|90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90|"; reference:bugtraq,2347; reference:cve,CVE-2001-0144; classtype:shellcode-detect; sid:1326; rev:3;)

rules/shellcode.rules:alert ip $EXTERNAL_NET any -> $HOME_NET $SHELLCODE_PORTS (msg:"SHELLCODE SGI NOOP"; content:"|03e0 f825 03e0 f825 03e0 f825 03e0 f825|"; reference:arachnids,356; classtype:shellcode-detect; sid:638; rev:3;)

The shellcode alerts in snort can cause many false positives: The x86 NOP can found in daily traffic. All it requires is a data that contains the same content, and the rule will trigger. JPG, GIF, PNG, PICT or Document (MS Word etc.) will trigger this rule. The following table shown the relationship for this alert to the MY.NET machines.

EXPLOIT x86 NOOP					
Date / Time	Source IP	Source Port	Destination IP	Destination Port	Alerts
01/16/04 05:15 pm	202.108.32.22	20	MY.NET.84.167	1883	200
01/18/04 12:43 am	24.130.153.222	51667	MY.NET.189.62	80	44
01/18/04 12:42 am	24.130.153.222	51611	MY.NET.5.45	80	40
01/19/04 11:45 am	193.220.82.38	3034	MY.NET.5.45	80	39
01/18/04 12:26 am	24.130.153.222	49279	MY.NET.189.62	80	37
01/18/04 12:26 am	24.130.153.222	49246	MY.NET.5.45	80	36
01/18/04 12:16 am	24.130.153.222	47065	MY.NET.5.45	80	35
01/18/04 12:12 am	24.130.153.222	47029	MY.NET.189.62	80	33

The source IP 202.108.32.22 triggers alerts with MY.NET.84.167, both on various ports. Note that the date is 01/16/04 for this IP. Do to the high false positives for this alert, I believe this traffic consist of files and/or documents.

During my analysis I cross referenced the IP MY.NET.84.167 against any other alerts. I received 7 hits. The following table shows the results of other alerts.

Alert cross reference with IP MY.NET.84.167				
Source IP	Source Port	Destination IP	Destination Port	Alert
128.171.198.49	4043	MY.NET.84.167	65535	High port 65535 tcp - possible Red Worm
MY.NET.84.167	65535	128.171.198.49	4043	High port 65535 tcp - possible Red Worm
128.171.198.49	4043	MY.NET.84.167	65535	High port 65535 tcp - possible Red Worm
MY.NET.84.167	65535	128.171.198.49	4043	High port 65535 tcp - possible Red Worm
202.108.32.22	20	MY.NET.84.167	4606	EXPLOIT x86 stealth noop
202.108.32.22	20	MY.NET.84.167	4606	EXPLOIT x86 stealth noop
218.15.124.18	1113	MY.NET.84.167	2337	EXPLOIT x86 setuid 0

The usage of port 65535 signals that MY.NET.84.167 is already compromised. The date for the above Red Worm traffic is 01/19/04, one day after the EXPLOIT x86 NOOP alerts. For more information on the Red Worm alert see section 4.1. The MY.NET.84.167 machine also generates the alert EXPLOIT x86 stealth noop from section 4.6. The alert 'EXPLOIT x86 setuid 0' is an attack to get administrative authority. The date of the Exploit alerts are also on 01/19/04. The alerts are after the alert described. This is opposite the alert sequence in section 4.6. Listed below is the whois information for IPs 202.108.32.22 and 128.171.198.49

Whois - 202.108.32.22
inetnum: 202.108.0.0 – 202.108.255.255 netname: CNCGROUP-BJ descr: CNCGROUP Beijing province network descr: China Network Communications Group Corporation descr: No.156,Fu-Xing-Men-Nei Street, descr: Beijing 100031 country: CN

Results: 128.171.198.49
OrgName: OrgID: UNIVER-25 Address: 2565 The Mall City: Honolulu StateProv: HI PostalCode: 96822 Country: US NetRange: 128.171.0.0 - 128.171.255.255 CIDR: 128.171.0.0/16 NetName: HAWAII NetHandle: NET-128-171-0-0-1 Parent: NET-128-0-0-0-0 NetType: Direct Assignment NameServer: DNS1.HAWAII.EDU NameServer: DNS2.HAWAII.EDU
Comment: RegDate: 1988-06-06 Updated: 2000-10-25

Correlation

Snort shellcode rules as being a False Alarms
http://thing.fwsystems.com/build/sburns/Kyle_Haugsness_GCIA.doc

Buffer overflow and x86 nop sled / false alarm
http://www.giac.org/practical/chris_kuethe_gcia.html

Snort high false positive alert / Exploit code
http://www.giac.org/practical/Wade_Dauphinee_GCIA.doc

Recommendation / Defense

Apply all anti-virus software and upgrades to this system. Verify the data stream that sets the snort alert. This machine MY.NET.84.167 is compromised. It needs to be hardened. There may be other problems generated from this machine as well. Some adjustment for the NOOP snort rule, may decrease the alert traffic to better manage.

4.8 Alert Name : SUNRPC highport access!

Probability source address was spoofed

I do not believe the source address is spoofed. As a connection using TCP the address would not be spoofed. As a scan, the scanner needs the the reply. There are hundreds of alerts just for one source IP.

Description of Attack

SUNRPC has a number of remote services. The mountd daemon runs on port 32771 and also known to run on port 32769 as well. It is possible to attack this daemon with a buffer overflow attack. If successful, the attacker may gain control of the server because the mountd daemon runs under root authorization. The attacker may be able to communicate with the mountd daemon directly on port 32771. The following table shows the frequency of this alert.

SUNRPC highport access! Alerts

Source IP	Source Port	Destination IP	Destination Port	Alerts
128.122.20.14	22	MY.NET.97.35	32771	366
192.148.251.86	20	MY.NET.70.37	32771	165
68.167.50.126	22	MY.NET.55.77	32771	162
204.255.212.10	110	MY.NET.97.157	32771	127

The IP 128.122.20.14 has an alert count of 366 when connecting to MY.NET.97.35. It is possible to use the ephemeral port 32771 as the source port and trigger this alert. This is not the case. Scanning port 32771 could trigger this alert. To see if the alerts are products of scans, the MY.NET IPs listed above were cross referenced with the scan file. No destination ports of 32771 showed up in the output. The following table shows the results of MY.NET.97.35 as it relates to the scans file.

Scan file hits : MY.NET.97.35

Source IP	Source Port	Destination IP	Destination Port	Scan type
152.8.106.2	4630	MY.NET.97.35	6129	SYN
194.185.90.248	2389	MY.NET.97.35	6129	SYN
194.185.90.248	3679	MY.NET.97.35	6129	SYN
207.89.248.3	1665	MY.NET.97.35	20168	SYN
211.186.116.110	4312	MY.NET.97.35	17300	SYN
216.63.158.11	3807	MY.NET.97.35	6129	SYN
24.194.138.234	2682	MY.NET.97.35	6129	SYN

Since there is no support for the alerts resulting from scans, I would think that these might be connections to port 32771. In this case MY.NET is compromised and needs to be hardened. The next table is a cross reference of MY.NET.97.35 with other alerts.

Cross reference MY.NET.97.35 with other Alerts

Destination IP	Destination Port	Alert
MY.NET.97.35	65535	High port 65535 tcp - possible Red Worm

When using port 65535 in this manner, this suggest that the Red Worm has infected this machine. See section 4.1 for more information on Red Worm. This is yet another sign to check the MY.NET.97.35 machine. The whois information for the external IP addresses leads me to think that the are the source of many attacks.

Whois results : Results: 128.122.20.14
OrgName: New York University OrgID: NYU Address: Academic Computing Facility Address: 251 Mercer Street City: New York StateProv: NY PostalCode: 10012 Country: US NetRange: 128.122.0.0 - 128.122.255.255 CIDR: 128.122.0.0/16 NetName: NYU-NET NetHandle: NET-128-122-0-0-1 Parent: NET-128-0-0-0-0 NetType: Direct Assignment NameServer: CMCL2.NYU.EDU NameServer: EGRESS.NYU.EDU NameServer: NYUNSB.NYU.EDU Comment: RegDate: 1986-05-02 Updated: 2001-05-21
TechEmail: NOC@nyu.edu

IP : 192.148.251.86 Results: Referring data:
OrgName: OARnet OrgID: OAR Address: 1224 Kinnear Road City: Columbus StateProv: OH PostalCode: 43212-1198 Country: US NetRange: 192.148.235.0 - 192.148.251.255 CIDR: 192.148.235.0/24, 192.148.236.0/22, 192.148.240.0/21, 192.148.248.0/22 NameServer: NS1.OAR.NET NameServer: NS2.OAR.NET RegDate: 1992-04-23 Updated: 1999-07-27
OrgTechEmail: hostmaster@oar.net

IP : 68.167.50.126 Results: Referring data:
OrgName: Covad Communications OrgID: CVAD Address: 2510 Zanker Rd City: San Jose StateProv: CA PostalCode: 95131-1127 Country: US NetRange: 68.164.0.0 - 68.167.255.255 CIDR: 68.164.0.0/14 NetName: NETBLK-COVAD-IP-4-NET

NetHandle: NET-68-164-0-0-1 Parent: NET-68-0-0-0-0 NetType: Direct Allocation NameServer: NS3.COVAD.COM NameServer: NS4.COVAD.COM Comment: RegDate: 2002-11-12 Updated: 2004-03-11 OrgNOCEmail: noc-ipservices@covad.com OrgTechEmail: sancha@covad.com

Correlation

referrence to yahoo and aol instant messenger triggers alert
http://www.lurhq.com/idsindepth.html

port 32771 – 34000 to block
http://is.rice.edu/~glratt/practical/Glenn_Larratt_GCIA.html

possible legitimate traffic due to enphemeral port 32771
http://www.users.globalnet.co.uk/~mlewis/Downloads/David_M_Lewis_GCIA_pdf.pdf

Correlating scan evidence
http://www.giac.org/practical/GCIA/Donald_Cunningham_GCIA.pdf

Recommendation / Defense

Ports for RPC services should be blocked, which include port 32771. The server MY.NET.97.35 must be checked as it seems to be compromised.

4.9 Alert Name : External RPC call

Probability source address was spoofed

The source address is not spoofed in this case. The destination port is targeted and the sender awaits a possible response from the portmap daemon running on port 111.

Description of Attack

The attacker is using portmap on port 111 to reveal the mountd port number. If this is successful the attacker can communicate with the mount daemon. The next step is to attack the mountd at the given port. Mountd runs under root and is vulnerable to buffer overflow attacks. When an attack to mountd is successful, the attacker gains root privileges to the system. This is a layout for such an attack starting with polling portmap for information as this detect alerts. The following is a table shows a frequency of the alert described.

External RPC call Alerts						
Source IP	Source port	Distinct Src port	Destination IP	Distinct Dst IP	Destination Port	Alerts
68.167.238.6	54892	261	MY.NET.5.5	260	111	748
62.111.213.88	3941	150	MY.NET.5.5	150	111	152

The table speaks for itself with a little concentration. In the first line,

address 68.167.238.6 uses 261 different port numbers to probe 260 different MY.NET addresses.
The 260 MY.NET address all are targeted with port 111. The same process applies to address 62.111.213.88. The count is 150 and is directed toward port 111. This is active targeting of MY.NET addresses for query information from the portmap daemon running on port111. The whois for the external IP addresses are sources for such an attack. The 2 external IP addresses did not show any hits in the scans file.

Results : 68.167.238.6 Referring data:
OrgName: Covad Communications OrgID: CVAD Address: 2510 Zanker Rd City: San Jose
StateProv: CA PostalCode: 95131-1127 Country: US NetRange: 68.164.0.0 - 68.167.255.255
CIDR: 68.164.0.0/14 NetName: NETBLK-COVAD-IP-4-NET NameServer: NS3.COVAD.COM
NameServer: NS4.COVAD.COM Comment: RegDate: 2002-11-12 Updated: 2004-03-11
AbuseEmail: abuse-isp@covad.com

Results : 62.111.213.88
inetnum: 62.111.128.0 – 62.111.255.255 netname: PL-CDP-20021126 descr:
PROVIDER Local Registry country: PL admin-c: AC4103-RIPE tech-c: PM452-RIPE
tech-c: GZ847-RIPE
tech-c: NL47-RIPE status: ALLOCATED PA notify: hostmaster@cdp.pl
changed: hostmaster@ripe.net 20021126 source: RIPE

route: 62.111.128.0/17
descr: Crowley Data Poland remarks: ---------- remarks: All abuse reports orginated from CDP network: abuse@cdp.pl changed: zorka5@cdp.pl 20021126 source: RIPE

Correlation

This detect provided RPC header information to analyze the packet further.
http://cert.uni-stuttgart.de/archive/intrusions/2003/11/msg00046.html

This detect provided leads to the getport() function.
http://cert.uni-stuttgart.de/archive/intrusions/2003/01/msg00209.html

This is detect provided information about portmap program numbers and lead to other detects.
http://cert.uni-stuttgart.de/archive/intrusions/2003/07/msg00155.html

Recommendation / Defense

Block port 111 and the related SUNRPC port range. Blocking the 2 external addresses may help in the short term. Use an authenticated method for remote services.

4.10 Alert Name : TCP SRC and DST outside network Probability source address was spoofed

The source addresses are not spoofed because this is a TCP 3-way connection. The source ports are known and suggest legitimate traffic.

Description of Attack

Private address are reserved and not meant for internet use. Since the alert triggered the connection was already made. It is possible that the internal address is used as a hop off point or using a proxy to mask the source's identity. The reason for masking your identity is for malicious intent. It is possible that the traffic is getting out by way of proxy or network address translation (NAT). It is possible that the addresses are not filtered or there is a mis configuration at some point. The analysis of this alert shows that multiple source addresses are used with multiple port numbers. There are multiple destination addresses when reviewing the entire alert table. The following table shows multiple source ports connecting to port 80.

Alerts for Private Addresses with listed ports

Source IP	Source port	Description	Destination IP	Destination Port	Count	Alert
192.168.1.103	1101	pt2-discover	64.152.73.207	80	12	TCP SRC and DST outside network
192.168.1.103	1117	ardus-mtrns	66.35.229.143	80	12	TCP SRC and DST outside network
192.168.1.103	1097	sunclustermgr	64.152.73.205	80	10	TCP SRC and DST outside network
192.168.1.103	1100	mctp	64.157.165.217	80	10	TCP SRC and DST outside network
192.168.1.103	1099	rmiregistry	64.152.73.238	80	10	TCP SRC and DST outside network
192.168.1.103	1098	rmiactivation	66.35.229.143	80	8	TCP SRC and DST outside network
192.168.1.103	1092	obrpd	66.35.229.172	80	6	TCP SRC and DST outside network
192.168.1.103	1110	nfsd-status	207.46.249.57	80	6	TCP SRC and DST outside network
192.168.1.103	1116	ardus-cntl	64.152.73.217	80	4	TCP SRC and DST outside network
192.168.1.103	1115	ardus-trns	66.35.229.143	80	4	TCP SRC and DST outside network

All of the source port numbers are known programs for that port. The destination for this source IP remains at port 80. I do not think this is an attack. This is traffic that should have been blocked, or legitimate traffic that should not trigger a snort alert.

Whois Results:64.152.73.207
Level 3 Communications, Inc. LC-ORG-ARIN (NET-64-152-0-0-1)
64.152.0.0 - 64.159.255.255
Gator.com LVL3-GATOR-01 (NET-64-152-73-0-1)
64.152.73.0 - 64.152.73.255

Correlation

This paper talks about possible proxy or NAT usage, and malicious intent
http://www.users.globalnet.co.uk/~mlewis/Downloads/David_M_Lewis_GCIA_pdf.pdf

This paper is on 'UDP SRC and DST outside network' with a fixed port 137
http://www.giac.org/practical/GCIA/Brian_Cahoon_GCIA.pdf

Overview of reasons why this traffic can leave the network
http://www.giac.org/practical/Rick_Yuen_GCIA.doc

Recommendation / Defense

Review mis configured firewalls and routers. The source address could be added to Snort's HOME_NET variable so the alert does not trigger. Review the traffic headed to the firewalls and/or routers.

5.0 The criteria used to select the top ten talkers is based on the Source IP addresses. The number of hits for the source IP address is bolded. The alerts related to that source IP address are listed below with the alert count and number of unique destination IP addresses. This is the Top Ten Alert generators.

Top Ten Talkers Based Alerts for Source IP

Source IP	Alert Name	Count	# Unique Dst IP
128.171.198.49	**Total :**	**14024**	
	High port 65535 tcp - possible Red Worm - traffic	14020	10406
	MY.NET.30.4 activity	2	1
	MY.NET.30.3 activity	2	1
24.35.58.199	**Total :**	**12079**	
	MY.NET.30.4 activity	12079	1
68.163.65.108	**Total :**	**7370**	
	MY.NET.30.4 activity	7370	2
68.54.254.152	**Total :**	**4784**	
	MY.NET.30.4 activity	4784	1
68.32.127.158	**Total :**	**3076**	
	connect to 515 from outside	3076	1
68.57.90.146	**Total :**	**2813**	
	MY.NET.30.3 activity	2753	1
	MY.NET.30.4 activity	60	1
129.165.254.6	**Total :**	**2481**	
	EXPLOIT x86 stealth noop	2481	1
68.55.62.79	**Total :**	**2467**	
	MY.NET.30.4 activity	2107	1
	MY.NET.30.3 activity	360	1
MY.NET.21.67	**Total :**	**2454**	
	Incomplete Packet Fragments Discarded	2451	0
	High port 65535 tcp - possible Red Worm - traffic	3	1
68.50.114.89	**Total :**	**2361**	
	MY.NET.30.4 activity	2178	1
	MY.NET.30.3 activity	183	1

Registration Source Address - 5

1) The host 128.171.198.49 is selected because it is the first external source IP in the Top Ten Talkers for source address. This host launched 14,024 alerts on the MY.NET network. The majority of alerts from this host is the 'High port 65535 tcp - possible Red Worm – traffic' attack.

OrgName: Internet Assigned Numbers Authority OrgID: IANA Address: 4676 Admiralty Way, Suite 330 City: Marina del Rey StateProv: CA PostalCode: 90292-6695 Country: US
ARIN WHOIS database, last updated 2004-02-01 19:15
Enter ? for additional hints on searching ARIN's WHOIS database.

2) The host 24.35.58.199 is selected because it is the second external source IP in the Top Ten Talkers for source address. This host launched 12,079 alerts on the MY.NET network. The majority of alerts from this host is the 'MY.NET.30.4 activity'.

OrgName: Cablespeed - Maryland OrgID: CSPE Address: 406 Headquarters Dr. City: Millersville StateProv: MD PostalCode: 21108 Country: US NetRange: 24.35.0.0 - 24.35.127.255
CIDR: 24.35.0.0/17 NetName: CSPE-2002-01 NetHandle: NET-24-35-0-0-1 Parent: NET-24-0-0-0-0 NetType: Direct Allocation NameServer: NS1.MIVLMD.CABLESPEED.COM NameServer: NS2.MIVLMD.CABLESPEED.COM Comment: RegDate: 2002-10-18 Updated: 2002-10-18
ARIN WHOIS database, last updated 2004-02-01 19:15
Enter ? for additional hints on searching ARIN's WHOIS database.

3) The host 68.163.65.108 is selected because it is the third external source IP in the Top Ten Talkers for source address. This host launched 7,370 alerts on the MY.NET network. The majority of alerts from this host is the 'MY.NET.30.4 activity'.

OrgName: Verizon Internet Services OrgID: VRIS Address: 1880 Campus Commons Dr City: Reston StateProv: VA PostalCode: 20191 Country: US NetRange: 68.160.0.0 - 68.163.255.255 CIDR: 68.160.0.0/14 NetName: VIS-68-160 NetHandle: NET-68-160-0-0-1
Parent: NET-68-0-0-0-0 NetType: Direct Allocation NameServer: NSDC.BA-DSG.NET NameServer: GTEPH.BA-DSG.NET Comment: RegDate: 2002-08-30 Updated: 2003-07-18
ARIN WHOIS database, last updated 2004-02-01 19:15
Enter ? for additional hints on searching ARIN's WHOIS database.

4) The host 68.54.254.152 is selected because it is the fourth external source IP in the Top Ten Talkers for source address. This host launched 4,784 alerts on the MY.NET network. The majority of alerts from this host is the 'MY.NET.30.4 activity'.

68.54.254.152 Results:
Comcast Cable Communications, Inc. JUMPSTART-1 (NET-68-32-0-0-1)
68.32.0.0 – 68.63.255.255
Comcast Cable Communications, Inc. DC-6 (NET-68-54-240-0-1)
68.54.240.0 - 68.54.255.255
ARIN WHOIS database, last updated 2004-03-22 19:15
Enter ? for additional hints on searching ARIN's WHOIS database.

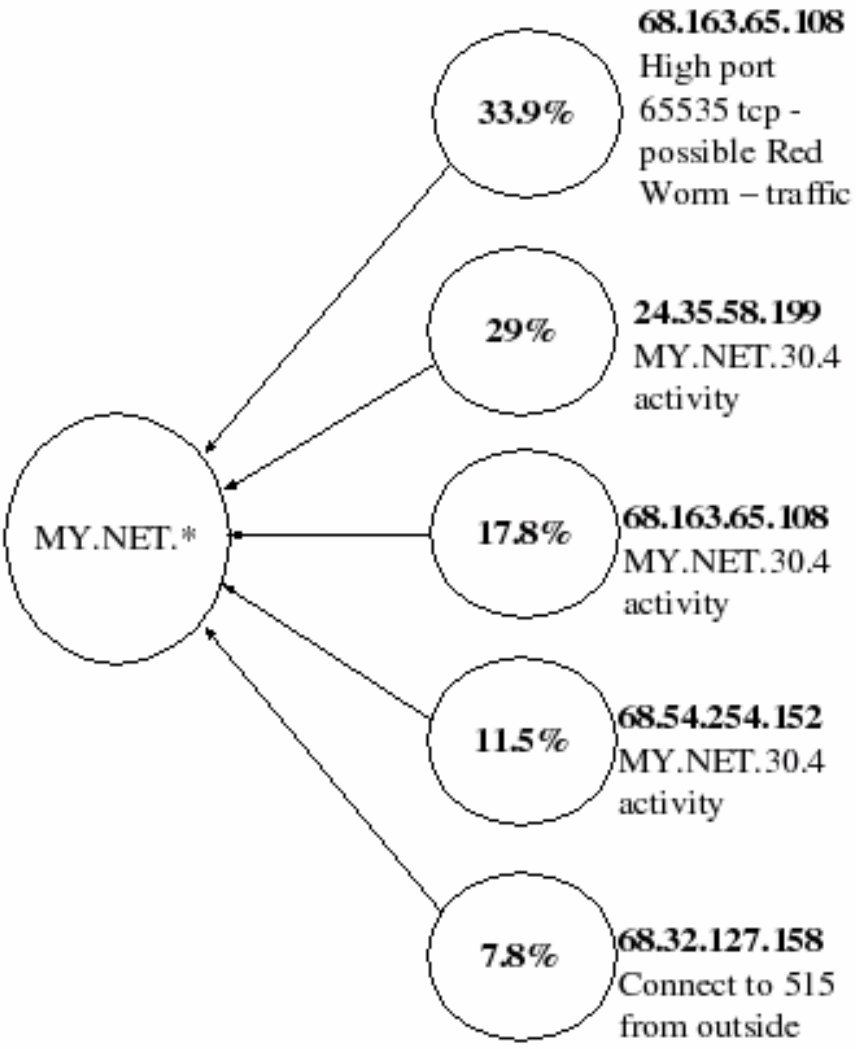
5) The host 68.32.127.158 is selected because it is the fifth external source IP in the Top Ten Talkers for source address. This host launched 3,076 alerts on the MY.NET network. The majority of alerts from this host is the 'MY.NET.30.4 activity'.

68.32.127.158 Results:

Comcast Cable Communications, Inc. JUMPSTART-1 (NET-68-32-0-0-1)
68.32.0.0 - 68.63.255.255
Comcast Cable Communications, Inc. BALTIMORE-A-2 (NET-68-32-112-0-1)
68.32.112.0 - 68.32.127.255
ARIN WHOIS database, last updated 2004-03-22 19:15
Enter ? for additional hints on searching ARIN's WHOIS database.

Link graph

The link graph shows what is considered to be legitimate traffic. The focus is on the IP addresses used for MY.NET.30.4 connections. The IP address 68.163.65.108 also generates 33.9 % of Red Worm traffic. This percentage is higher than any other address connecting to MY.NET.30.4. The highport alerts must be addressed by the MY.NET network administration to provide a defense against such traffic.



Analysis Process

> Use grep, awk, and sed programs to determine what condition the log files are in. The goal is to parse the data to an one line entry for Mysql database.
Example of sed command to prepare alert file for mysql input.

```
sed -e 's/\[\*\*\]/%/g' alert-all.txt | sed -e 's/->/%/g' | sed -e 's/:/ % /4' | sed -e 's/:/ % /3' > alert-all.mysql
```

> Prepare files to load into mysql database. This include creating tables, and brushing up on SQL commands.
Example of mysql command to read alert-all.mysql.

```
use gcia;
drop table alerts;
create table alerts (date varchar (21),alert varchar (60),src varchar (15),srcp int (6),dst varchar (15),dstp int (6) );
load data infile '/home/GCIA/Alerts/alert-all.mysql' into table alerts fields terminated by '%' ;
```

> Understanding how to use SQL commands to make sense of the data for analysis.
The power of SQL is more than enough to analysis the large amounts of data in detail.
> Research alerts to find out what they really do, and if they are false positives.
> Research and analyze the relationships between IP addresses both internal and external.
> Use Internet to research alerts.
> Start making sense of the numbers via SQL listings.
Example select statement:

```
select alert,
  count(distinct src)as unique_src_IPs,
  count(distinct dst) asunique_dst_IPs,
  count(*) as alerts from alerts
  group by alert order by alerts desc limit 20;
```

> Run 'Whois' on the external IP addresses chosen.

> Start the report phase to put the findings in words.