Global Information Assurance Certification Paper

GIAC GCIA

Practical Assignment

Version 3.3

By Mihai Cojocea

Submitted on December 22, 2003

SANS Fire

Washington DC, July 2003

TABLE OF CONTENTS

ABSTRACT

The fist part of this paper presents two Perl "cookbook" solutions meant to help the network security analyst in the processing of Snort IDS generated scan log files like scan.log and portscan.log, This part also presents how these Perl scripts can be used to detect worms like Blaster and Welchia.

The second part analyzes three detects:

1. ICMP Large Packet alert triggered as a false positive by a MacOS Path MTU Discovery Mechanism;
2. A suite of three scans for open proxies;
3. Sadmin Worm – attack.

The third part is an analysis of five days worth of alerts, portscans, and out of spec logs generated by a Snort Intrusion Detection Sensor connected outside of the perimeter router of a University.

PART 1 - SNORT PORTSCAN ANALYSIS-PERL COOKBOOK

Traditionally, port scanning was compared to "ringing the bell to see if there is somebody home", or "jiggling the door knob to see if the door is open". Actually, port scanning identifies open channels of communications (ports) into computers[1].

Although port scanning can have legitimate uses – usually scans executed by systems owners to detect vulnerabilities of their systems – the bulk of port scans detected by Intrusion Detection Systems (IDS) are malicious in nature.

These malicious port scans can be run manually, by persons who physically sit at the keyboard of a computer and execute a port scan, or automatically, by computers infected with virusi, worms, or Trojans.

At the time this paper was written, the Distributed Intrusion Detection System Top 10 Target Ports (http://dshield.com/topports.php) still lists port 135 as the most targeted port in North America, with about 35% of all scans. At the end of October this year, the number was even higher - 56% of the port scans had port 135 as target. It is very likely that the Blaster worm executes the majority of these scans and, or the Welchia worm. Actually, the number of scans to port 135 may have been higher, but a lot of scans do not execute because they are stopped at the border routers or firewalls of a target network.

After the occurrence of the Blaster and Welchia worms, port scanning begins to resemble commercial mass mailing[2], where a huge number of requests are mailed and the rate of

[1] Fyodor.
[2] Fyodor.

response as a relative percentage is extremely low. But, because the number of mailed requests is huge, the absolute number of positive answer is quite high.

On August 11th 2003, a new worm hit computer networks: Blaster[3]. The worm exploits a buffer overflow in the Microsoft Remote Procedure Call (RPC)[4]. This vulnerability was addressed by Microsoft Security Bulletins MS03-026[5] and MS03-039[6]. About one week later, another worm was released: Welchia. This worm is a "good" worm: it scans to find computers infected with Blaster, remove the Blaster, and patch the system. Both Blaster and Welchia are random start sequential scanners and generate a large amount of traffic[7].

1.1 Detecting the Welchia computer worm with Snort Scan and Perl

The attack phase of the traffic generated by a machine infected with the Welchia worm has the following pattern[8]:

1. An ICMP echo request is sent to find computers, which are up and on-line and, therefore, could be potential targets.
2. Computers that are up and online - the targets - answer with ICMP echo replies.
3. Infected computer tries to connect to port 135 of targets.

A Snort sensor will see:

1. A sweep of ICMP echo requests meant to find machines, which are up and, therefore, are potential targets.

In a *scan.log* file generated by a Snort IDS this ICMP sweep looks like:

```
09/09-07:03:52.329907  ICMP src: 192.168.211.11 dst: 192.168.1.83 type: 8 code: 0 tgts: 7
event_id: 32
09/09-07:03:52.345681  ICMP src: 192.168.211.11 dst: 192.168.1.84 type: 8 code: 0 tgts: 8
event_id: 32
09/09-07:03:52.485964  ICMP src: 192.168.211.11 dst: 192.168.1.93 type: 8 code: 0 tgts: 9
event_id: 32
09/09-07:03:52.501444  ICMP src: 192.168.211.11 dst: 192.168.1.94 type: 8 code: 0 tgts: 10
event_id: 32
09/09-07:03:52.554114  ICMP src: 192.168.211.11 dst: 192.168.1.97 type: 8 code: 0 tgts: 11
event_id: 32
```

2. Followed by a TCP SYN to the port 135:

```
09/09-07:03:52.560161  TCP src: 192.168.211.11 dst: 192.168.1.97 sport: 2869 dport: 135 tgts:
11 ports: 21 flags: ******S* event_id: 32
```

[3] Networm.org. What happened with Blaster and Welchia?
[4] Microsoft. What You Should Know About Microsoft Security Bulletin MS03-026.
[5] Microsoft. What You Should Know About the Blaster Worm and Its Variants
[6] What You Should Know About Microsoft Security Bulletin MS03-039.
[7] Networm.org. What happened with Blaster and Welchia?
[8] Symantec. Detecting network traffic that may be due to RPC worms.

This pattern of scans can be easily analyzed with a Perl script. In the next paragraph I will present such a script. This is more like a proof of concept, rather than a solution to find computers infected with the Welchia worms. Parts of this script can be reassembled into other Perl scripts to perform similar processing of other log file formats.

```
welchia.pl
#!/usr/bin/perl -s
## usage: cat scan.log | ./welchia.pl
$max_icmp = 5;
$max_tcp = 1;
```

`$max_icmp` is the minimum number of ICMP requests originating from the same host that would pass the originating IP to the next test. You should experiment to see what would be a correct number for your environment.

`$max_tcp` is the minimum number of TCP SYN sent from the same IP that sent the ICMP request to the same target.

For the rest of this script I will present a section of the script, and then I will explain the commands:

```
while (<>) {
      chomp;
            ($time, $proto, $src, $source_host, $dst, $destination_host, $tp,
            $tp_number, $cd_prt, $dest_number, $tgts, @item) = split;
```

The three commands listed above (`while, chomp, split`) read the *scan.log* lines one by one (while, chomp) and "split" the whole scan line into separate variables (`split`) like `$time`, `$proto` etc.

To be processed by the above presented commands, the line is loaded into a default variable - `$_`. The advantage of this variable is that if it is used as an argument it could be omitted.

For example, we used `split;` instead of `split $_;`

```
if (
      ($proto eq 'ICMP') and ($tp_number == 8) or
      ($proto eq 'TCP') and ($dest_number == 135)
)
```

The Perl "if" condition follows this pattern: `if (condition) { do this }`.

If these conditions are met (if the protocol of the scan is ICMP and the packet is an ICMP request or the protocol is TCP and the target port of the scan is 135), then

```
      { # if
            $line = join (" ", ($proto, $source_host, $destination_host));
            push (@newscan, $line);
      } # if
} # while
```

Then, only the protocol, the source IP address and the destination IP address are "joined" into a new line, named `$line`. This line is added (`push`) to an array of line scans - `@newscan`.

This array can be sorted and/or printed. For example, to sort by time, you would need to join the new lines having `$time` as the first element. To sort by source host you would need to join the new lines having the source host (`$source_host`) as the first element.
Basic sorting in Perl can be done numerically, alphabetically or "mostly numerically". A good article about basic Perl sorting (by numbers or by name or mixed)[9]:

```
# numerical sorting
@newscan_sorted = sort { $a <=> $b } @newscan;

# alphabetical sorting
@newscan_sorted = sort { $a cmp $b } @newscan;

# mixed (numerical or alphabetical) sorting
@newscan_sorted = sort by_mostly_numeric @newscan;
  sub by_mostly_numeric {
      ($a <=> $b) or ($a cmp $b);
  }
```

Only one of the above three sorting methods should be used.

For more advanced scanning, you can look at Chapter 4.15. "Sorting a List by Computable Field"[10] – in *Perl Cookbook*, by Tom Christiansen & Nathan Torkington, O'Reilly.

```
@newscan = @newscan_sorted;
foreach $line (@newscan) {
   chomp ($line);
   ($proto, $source_host, $destination_host) = split (" ", $line);
```

Once scan lines are sorted each line of the new array is split again for further processing.

```
   if ($proto eq 'ICMP') { $icmp{$source_host}++}
   if ($proto eq 'TCP') { $tcp{$source_host}++}
   push (@sources, $source_host);
} # foreach
```

I created a hash of ICMP counter where the key is each source IP - `$icmp{$source_host}` – and a hash of TCP to port 135 counter where the source IP is also the key.

For each line of ICMP scan originating from the same source, the counter is increased with one. A similar procedure is applied to the TCP counter.

In Perl, the difference between an array (@name_of_array) and a hash (%name_of_hash) is that arrays are made up of members indexed only with numbers:

@color = ($color[0], $color[1]) = ("red", "green");

and hashes can be "indexed" with keys that can be words:

%color = (apple, $color{apple}, melon $color{melon}, ….)

($color{apple}, $color{melon}) = ("red", "green");

red = $color{apple}; green = $color{melon};

After being counted, all chosen source IP addresses are "pushed" into an array - `@sources`.

[9] *Cthuang.* Programming Language Using Perl -- Advanced Sorting
[10] Tom Christiansen & Nathan Torkington *Perl Cookbook*

```
@sources = uniq (@sources);
```

The line presented above calls a subroutine named `uniq`. This subroutine removes duplicate elements of an array[11].

```
sub uniq {
        ## this subroutine takes a @list and filter out duplicate items
        my (%seen) = ();
        my (@uniq) = ();
        foreach  $item (@_) {
                push (@uniq, $item) unless $seen{$item}++;
        } # foreach
return (@uniq);
} # sub

foreach $item (@sources) {
        if (($icmp{$item} >= $max_icmp) and ($tcp{$item} >= $max_tcp)) {
                print "$item\n";
        }
}
```

Each IP address of the `@source` array has a counter of ICMP and a counter of TCP scans. If there are more than `$max_icmp` and more than `$max_tcp`, the IP is printed to screen.

The next subroutine can take a numeric IP as an argument and try to resolve the IP to hostname using the function `gethostbyaddr`:

```
sub hostname {
        my $ip = "$_[0]";
        my @octet = split (/\./, $ip);
        my @host = gethostbyaddr (pack('C4', @octet), 2);
        $host = $host[0]; return $host;
}
```

In a contiguous form, the Perl script that parses Snort generated scans.log files is:

```
# more welchia.pl
#!/usr/bin/perl  -s
## usage: cat scan.log |./welchia.pl
$max_icmp = 5;
$max_tcp = 1;

while (<>) {
       chomp;
       ($time, $proto, $src, $source_host, $dst, $destination_host, $tp, $tp_number,
       $cd_prt, $dest_number, $tgts, @item) = split;
        if (
                ($proto eq 'ICMP') and ($tp_number == 8) or
                ($proto eq 'TCP') and ($dest_number == 135)
           )
        {
           $line = join (" ", ($proto, $source_host, $destination_host));
           push (@newscan, $line);
        } # if
         if ($proto eq 'ICMP') { $icmp{$source_host}++} # if ICMP increase counter with 1
         if ($proto eq 'TCP') { $tcp{$source_host}++} # if TCP increase counter with 1
         push (@sources, $source_host);
} # while
@sources = uniq (@sources);
foreach $item (@sources) {
        if (($icmp{$item} >= $max_icmp) and ($tcp{$item} >= $max_tcp)) {
```

[11] Perl Cookbook.

```
                $host = hostname($item); #does a nslookup of the IP address
                print "$item\t$host\n";
        }
}
#######################################
sub hostname {
        ## This subroutine takes an IP address as an argument and returns its hostname.
        my $ip = "$_[0]";
        my @octet = split (/\./, $ip);
        my @host = gethostbyaddr (pack('C4', @octet), 2);
        $host = $host[0]; return $host;
}
#######################################
sub uniq {
        ## this subroutine takes an array and filter out duplicate items
        my (%seen) = ();
        my (@uniq) = ();
        foreach  $item (@_) {
                push (@uniq, $item) unless $seen{$item}++;
        } # foreach
return (@uniq);
} # sub
#######################################
```

1.2 Perl analysis of Snort "portscan.log" files

The structure of the *portscan.log* file makes it very suitable for Perl analysis:

```
Sep  9 07:03:50 192.168.211.11:2845 -> 192.168.1.7:135 SYN ******S*
```

1.2.1 How to sort the portscan.log by month or by other variable

The alphabetical representation of the Month is not very easy to sort – one cannot use neither alphabetical, nor numerical sorting[12]. A hash is used to convert month names to number, which can be easily sorted.

```
%convert = ("Jan", "01", "Feb", "02", "Mar", "03", "Apr", "04",
          "May", "05", "Jun", "06", "Jul", "07", "Aug", "08",
          "Sep", "09", "Oct", "10", "Nov", "11", "Dec", "12");
```

Now, for example, `01 = $convert{Jan}, 02 = $convert{Feb}` and so on to December.

A second hash could be used to convert the "month number" back to a literal month name. Literal month names are usually used to increase the readability of reports.

```
%trevnoc = ("01", "Jan", "02", "Feb", "03", "Mar", "04", "Apr",
         "05", "May", "06", "Jun", "07", "Jul", "08", "Aug",
         "09", "Sep", "10", "Oct", "11", "Nov", "12", "Dec");

while (<>){
       chomp;
       ($mon, $day, $time, $src, $arrow, $dst, @scantype) = split;
       $mon = $convert{$mon}; # convert month name to number
       ($source_host,$source_port) = split(/:/,$src);
       ($destination_host,$destination_port) = split(/:/,$dst);
       $newline = join(" ", ("$mon.$day",$time, $source_host, $source_port,
$destination_host, $destination_port, @scantype));
       push(@newlog,$newline);
} # while
```

[12] Chris Kuethe.

At this point we can sort the portscan lines (in our case based by date and time). To sort by source port (`$source_port`) for example, we would join the new line using `$source_port` as the first element:

```
$newline = join(“.” ($source_host, .  .  .  .  ));
```

and then sort it numerically:

```
@newline_sorted = sort { $a <=> $b } @newline;
```

After sorting we need to split the lines and revert the month names to alphabetical names. We will also use the same loop to count the total number of portscan lines:

```
$i = 0; # initialize the line counter
foreach $line (@newlog) {
       ($m_day[$i], $time[$i], $source_host[$i], $source_port[$i],
       $destination_host[$i], $destination_port[$i], $flag[$i], $row_flag[$i]) = split (/ /,
       $line);
       ($month[$i], $day[$i]) = split (/\./, $m_day[$i]);
       $month[$i] = $trevnoc{$month[$i]}; # change back the name of the month
       $i++;
}
$imax = $i-1;
```

1.2.2 How to eliminate the false positives generated by FTP passive traffic

Below you will find a typical example of false positive portscans generated by a FTP passive traffic.

```
Oct  2 12:15:01 192.168.251.73:1671 -> 192.168.1.7:62542 SYN ******S*
Oct  2 12:15:02 192.168.251.73:1672 -> 192.168.1.7:62543 SYN ******S*
Oct  2 12:15:02 192.168.251.73:1673 -> 192.168.1.7:62544 SYN ******S*
Oct  2 12:15:02 192.168.251.73:1674 -> 192.168.1.7:62545 SYN ******S*
Oct  2 12:15:03 192.168.251.73:1675 -> 192.168.1.7:62546 SYN ******S*
Oct  2 12:15:03 192.168.251.73:1676 -> 192.168.1.7:62547 SYN ******S*
```

The pattern to look for to recognize the FTP passive traffic, which generates false positives, is:

a. Destination IP address is the FTP Server;
b. Source port and destination port increase with one each line;
c. TCP flag is set to SYN.

Determine the FTP false positives:

```
$ftp_ip = “192.168.100.70”; # configure here the IP address of the FTP Server
for ($i = 0; $i <= $imax; $i++)
{
   push (@fp_s_h, $source_host[$i]) if
   (("$destination_host[$i]" eq "$ftp_ip") and
   ("$source_host[$i]" eq "$source_host[$i+1]") and
   (($source_port[$i]+1) == ($source_port[$i+1])) and
   (($destination_port[$i]+1) == ($destination_port[$i+1])) and
   ("$destination_host[$i+1]" eq "$ftp_ip") and
   ("$source_host[$i+1]" eq "$source_host[$i+2]") and
   (($source_port[$i]+2) == ($source_port[$i+2])) and
   (($destination_port[$i]+2) == ($destination_port[$i+2]))
        and ("$flag[$i]" eq "SYN")
        )
} # for
@fp_uniq = uniq_list(@fp_s_h); # call uniq_list subroutine
# this is an array of IP addresses that generated FTP passive false positives
```

To mark portscan lines that are FTP false positives, one must use a “tag” - `$off_bit`. If a line is a FTP Passive false positive the tag will have a value of 1, else it will be 0.

```
for ($i = 0; $i <= $imax; $i++) {
       $off_bit[$i] = 0; #  initialize all tags to 0 - nothing is a false positive yet
}
foreach $ip (@fp_uniq) {
       for ($i = 0; $i <= $imax; $i++) {
              if ("$ip" eq "$source_host[$i]") {
                     $off_bit[$i] = 1; # tag the false positives
              } # if
       }  # for
} #  foreach
```

1.2.3 How to determine the Class Address of your netwok, local scans and scans to port 135 (possible Blaster scans)

```
## Add here the network part of your IP address.
## Based on this hosts source IP of scans will be tagged as "local" or "external"
$mynet = '172.16.0.0';
($a, $b, $c, $d) = split (/\./, $mynet);
$class = 'A' if ( ($d == 0) && ($c == 0) && ($b == 0) && ($a != 0) );
$class = 'B' if ( ($d == 0) && ($c == 0) && ($b != 0) && ($a != 0) );
$class = 'C' if ( ($d == 0) && ($c != 0) && ($b != 0) && ($a != 0) );
for ($i = 0; $i <= $imax; $i++)
{
   if ($destination_port[$i] == $blaster_port) {$off_port[$i] = 1} # tag Blaster portscans
        else {$off_port[$i] = 0}
   ($A_s_h[$i], $B_s_h[$i], $C_s_h[$i], $D_s_h[$i]) = split (/\./, $source_host[$i]);
        if ( # tag portscans that originate from your network
                ("$class" eq "A")  and  ("$A_s_h[$i]" eq "$a")
           ) { $local_bit[$i] = 1 }
        elsif ( ("$class" eq "B") && ("$A_s_h[$i]" eq "$a")
                && ("$B_s_h[$i]" eq "$b") ) {$local_bit[$i] = 1}
        elsif ( ("$class" eq "C") && ("$A_s_h[$i]" eq "$a") && ("$B_s_h[$i]" eq "$b")
                && ("$C_s_h[$i]" eq "$c") ) {$local_bit[$i] = 1}
        else {$local_bit[$i] = 0}
}
```

Now, the `$local_bit` is used to tag scans originating from your own network, and `$off_port` will tag possible port scans generated by Blaster. For example, if the $local_bit is null, then the scan was generated by a computer that dos not belong to your network. If the $local_bit is 1 (true), then the scan is originating from a host that is located on your network.

1.2.4 Filter out the lines tagged as FTP passive false positives, local and Blaster scans

```
for ($i = 0; $i <= $imax; $i++) {
  unless (  ($off_bit[$i] == 1) or ($local_bit[$i] == 1) or ($off_port[$i] == 1)  ) {
       $newline[$i] =  join       (" ", ("$month[$i]", "$day[$i]", "$time[$i]",
              "$source_host[$i]",  $source_port[$i], "$destination_host[$i]",
              $destination_port[$i], "$flag[$i]", "$row_flag[$i]"));
       push (@clean_scan, @newline[$i]);
  } #
} # for
@clean_scan = uniq_list (@clean_scan);
```

All portscan lines considered to be “clean” will be printed on screen and also saved to a file “clean_scans.txt”.

```
print "\n\n\n############## CLEAN SCAN LINES ####################\n".
"(false positives, local scans, and blaster extracted)\n\n";
open (CLEAN, ">clean_scans.txt");
```

```
$n = 0;
foreach $line (@clean_scan) {
       print "$n, $line \n";
       print CLEAN "$n $line\n";
       $n++;
} # foreach
print "\n\Total number of clean lines is $n \n";
close (CLEAN);
```

1.2.5 How to Filter for scans that are not performed by the Blaster worm and which originate from your local network

```
######### LOCAL SCANS NOT BLASTER ####################
for ($i = 0; $i <= $imax; $i++) {
  if (  ($off_bit[$i] == 0) and  ($local_bit[$i] == 1) and ($off_port[$i] != 1)  ) {
       $nbline[$i] =  join (" ",("$month[$i]", "$day[$i]", "$time[$i]", "$source_host[$i]",
       $source_port[$i], "$destination_host[$i]", $destination_port[$i], "$flag[$i]",
       "$row_flag[$i]"));
       push (@nb_scan, $nbline[$i]);
  } #
} # for
@nb_scan = uniq_list(@nb_scan);
$n = 0;
foreach $line (@nb_scan) {
       print "$n, $line \n";
       $n++;
} # foreach
```

1.2.6 How to filter for scans performed by computers from your local network infected with the Blaster worm

```
######### BLASTER LOCAL SCANS ####################
## This section filters and report the local Blaster generated scans.
print "\n\n\n############### BLASTER LOCAL SCANS ##########\n\n";
for ($i = 0; $i <= $imax; $i++) {
  if (  ($off_bit[$i] == 0) and  ($local_bit[$i] == 1) and ($off_port[$i] == 1)  ) {
       $blaster_line[$i] =  join  (" ", ("$month[$i]", "$day[$i]", "$time[$i]",
       "$source_host[$i]", $source_port[$i], "$destination_host[$i]", $destination_port[$i],
       "$flag[$i]", "$row_flag[$i]"));
       push (@blaster_scan, $blaster_line[$i]);
       push (@blasted_ips, $source_host[$i]);
  } #
} # for
@blaster_scan = uniq_list(@blaster_scan);
@blasted_ips = uniq_list(@blasted_ips);
$n = 0;
foreach $line (@blaster_scan) {
#      print "$n, $line \n"; # uncomment these two print lines if you  like to
       $n++;
} # foreach
print "\nLOCAL BLASTED IPs:\n\n";
foreach $ip (@blasted_ips) { print "$ip\n";}
```

1.2.7 Port scan lines sorting: hostscans, portscans, and centralized

Scanning is usually performed to discover:

a. If a certain service is running on a network – same destination port, same source IP address, different destination IP addresses;
b. What services a computer is running – same destination IP address, same source IP address, different destination ports.

To illustrate this concept, we will count the number of:

a. same source and same destination port scans;
b. same source and same destination host scans.

Then we will centralize all the scans originating from the same IP address, regardless if it is a port or a host scan.

```
foreach $line (@clean_scan) {
       ($m, $d, $t, $source_host, $source_port, $destination_host, $destination_port, $flag,
       $row_flag) = split (/ /, $line);
       $same_sh_dp = "$source_host $destination_port";
       $same_sh_dh = "$source_host $destination_host";
       $same_sh_dp{$same_sh_dp}++;
       $same_sh_dh{$same_sh_dh}++;
} # foreach
```

We will need to set threshold values for scanning:

a. minimum number of ports to be scanned on the same destination that will be considered a port scan - `$pmax`;
b. minimum number of hosts to be scanned for the same port that will be considered a host scan - `$hmax`.

```
$pmax = 5;
$hmax = 5;
```

All IP addresses of the hosts which performed the scans (port and host scans) will be “pushed” into an array `@offender_ip` for later processing.

The subroutine *hostname* presented in the previous Perl script (welchia.pl) will try to resolve the IP address to a hostname. If the hostname is not found, the IP address will be used instead.

The subroutine *service* tries to find, based on the /etc/services file, the service name that is running on a port. If the service name is not found, the port number is used instead.

```
#############################################################################
sub service {
        open (INSERV, "/etc/services") || die "Sorry, cannot read /etc/services. \n";
        my $srvc; my $port; my %service; my @ports; my $in_port;
        while (<INSERV>) {
                next if (/^#/); # skip comment lines
                ( my($srvc), my($port) ) = split;
                ($port, $protocol) = (split ('/', $port)); # get rid of the UDP & TCP
                if ("$protocol" eq "tcp") {
                        $service{$port} = $srvc;
                        push (@ports, $port);
                } # if
        } # while
        $in_port = $_[0];
        return "$service{$in_port}";
}
#############################################################################
```

1.2.8 “Same source host, same destination port” scan processing

```
foreach $same_sh_dp ( sort { $same_sh_dp{$b} <=> $same_sh_dp{$a} } keys %same_sh_dp ) {
       if ($same_sh_dp{$same_sh_dp} >= $pmax) { # if A
              ($source_host, $destination_port) = split (/ /, $same_sh_dp);
              push (@offender_ip, $source_host);
              $hostname = hostname("$source_host");
```

```
            unless ($hostname) { $hostname = "$source_host" }
            $serv = service($destination_port);
            if ($serv =~ /^[A-Za-z]+$/)       {
                  $serv = "$serv";
            } else {
                  $serv = "$destination_port";
            }
            print "\n\n\nPORTSCAN @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@\n";
            print "SOURCE: $hostname ($source_host) scanned $same_sh_dp{$same_sh_dp} hosts
            \nfor SERVICE: $serv ($destination_port) \n\n";
              @count = grep { /\b$source_host\b/ && /\b$destination_port\b/  } @clean_scan;
              foreach $ln (@count) {
                  print "$ln \n";
              } # foreach
      } # if A
} # foreach
```

1.2.9 "Same source host, same destination host" scan processing

```
print "\n\nPORSTSCAN SORTED by SAME HOST, SAME DESTINATION HOST [HOSTSCAN]\n";
foreach $same_sh_dh ( sort { $same_sh_dh{$b} <=> $same_sh_dh{$a} } keys %same_sh_dh ) {
      if ($same_sh_dh{$same_sh_dh} >= $hmax) { # if A
            ($source_host, $destination_host) = split (/ /, $same_sh_dh);
            push (@offender_ip, $source_host);
            $hostname = hostname("$source_host");
            unless ($hostname) { $hostname = "$source_host" }
            print "\n\n\nHOSTSCAN @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@\n";
            print "SOURCE: $hostname ($source_host) scanned the host $destination_host for
            more
            \nthan $same_sh_dh{$same_sh_dh} ports\n\n";
            @count = grep { /\b$source_host\b/ && /\b$destination_host\b/  } @clean_scan;
              foreach $ln (@count) {
                  print "$ln \n";
              } # foreach
      } # if A#
} # foreach
```

1.2.10 Centralize the list of hosts that performed port, host scanning, or both

We will use a subroutine that attempts to locate the TLD (Top Level Domain[13]) of the "scanner". It checks only for six generic TLDs (.com, .edu, .gov, .mil, .net, and .org)

```
#############################################################################
## This subroutine takes a FQDN as an argument and, if it ends in mil, net, gov, edu
## com, org returns a possible domain.
## Obsviously, you can add more domains (by coutry[14], for example).
## http://www.iana.org/cctld/cctld-whois.htm
## http://www.icann.org/tlds

sub tld {
        my @host = split (/\./, $_[0]);
        my $tld = $host[-1];
        if (("$tld" eq "com") or ("$tld" eq "net") or
            ("$tld" eq "edu") or ("$tld" eq "gov") or
            ("$tld" eq "org") or ("$tld" eq "mil")) {
               $abuse_mail = join (".", ($host[-2], $tld));
                return $abuse_mail;
        } else { return ()}
}
#############################################################################
```

The actual centralization is executed in the following lines of the script:

[13] IANA – Top Level Domains.
[14] IANA – Root Zone Whois Information.

```
print "\n\n\n############### ALL SCANS FROM EACH OFFENDER IP ####################\n\n";
@offender_ip = uniq_list(@offender_ip);
foreach $offender (@offender_ip) {
        $hostname = hostname($offender);
        print "\n\nOFFENDER IP: $offender\t\nHOSTNAME: $hostname\n";
        if ( hostname("$offender") ) {
           $domain = tld($hostname);
           # print "Hostname was found via TLD: $hostname\n";
           if ($domain) {print "DOMAIN: $domain\n"; print "Try to e-mail to abuse\@$domain. \n\n";}
        } else {
                print "Hostname was not found via TLD.\n\n";
        }
        @count = grep { /\b$offender\b/ } @clean_scan;
print "All scans originating from the same host:\n\n";
foreach $ln (@count) {
                print "$ln \n";
        }
}
```

1.2.11 Final portscan analysis script

The final portscan analysis script follows:

```
#! /usr/bin/perl -s
# USAGE: #cat portscan.log | ./parse.pl
%convert = ("Jan", "01", "Feb", "02", "Mar", "03", "Apr", "04",
          "May", "05", "Jun", "06", "Jul", "07", "Aug", "08",
          "Sep", "09", "Oct", "10", "Nov", "11", "Dec", "12");
%trevnoc = ("01", "Jan", "02", "Feb", "03", "Mar", "04", "Apr",
          "05", "May", "06", "Jun", "07", "Jul", "08", "Aug",
          "09", "Sep", "10", "Oct", "11", "Nov", "12", "Dec");
while (<>){
        chomp;
        ($mon, $day, $time, $src, $arrow, $dst, @scantype) = split;
        $mon = $convert{$mon}; # convert month name to number
        ($source_host,$source_port) = split(/:/,$src);
        ($destination_host,$destination_port) = split(/:/,$dst);
        $newline = join(" ", ("$mon.$day",$time, $source_host, $source_port, $destination_host,
$destination_port, @scantype));
        push(@newlog,$newline);
} # while
# $newline = join(' ', ($source_host, .  .  .  .  ));
# @newline_sorted = sort { $a <=> $b } @newline;
$i = 0; # initialize the line counter
foreach $line (@newlog) {
        ($m_day[$i], $time[$i], $source_host[$i], $source_port[$i],
        $destination_host[$i], $destination_port[$i], $flag[$i], $row_flag[$i]) = split / /, $line;
        ($month[$i], $day[$i]) = split (/\./, $m_day[$i]);
        $month[$i] = $trevnoc{$month[$i]}; # change back the name of the month
        $i++;
}
$imax = $i-1;
$ftp_ip = '192.168.1.7'; # configure here the IP address of the FTP Server
for ($i = 0; $i <= $imax; $i++)
{
   push (@fp_s_h, $source_host[$i]) if
   (("$destination_host[$i]" eq "$ftp_ip") and
   ("$source_host[$i]" eq "$source_host[$i+1]") and
   (($source_port[$i]+1) == ($source_port[$i+1])) and
   (($destination_port[$i]+1) == ($destination_port[$i+1])) and
   ("$destination_host[$i+1]" eq "$ftp_ip") and
   ("$source_host[$i+1]" eq "$source_host[$i+2]") and
   (($source_port[$i]+2) == ($source_port[$i+2])) and
   (($destination_port[$i]+2) == ($destination_port[$i+2]))
        and ("$flag[$i]" eq "SYN")
        )
} # for
@fp_uniq = uniq_list(@fp_s_h); # call uniq_list subroutine
# this is an array of IP addresses that generated FTP passive false positives
for ($i = 0; $i <= $imax; $i++) {
        $off_bit[$i] = 0; #  initialize all tags to 0 - nothing is a false positive yet
```

```
}
foreach $ip (@fp_uniq) {
        for ($i = 0; $i <= $imax; $i++) {
                if ("$ip" eq "$source_host[$i]") {
                        $off_bit[$i] = 1; # tag the false positives
                } # if
        }  # for
} #  foreach
$mynet = '192.168.0.0'; # configure here your IP Class
($a, $b, $c, $d) = split (/\./, $mynet);
$class = 'A' if ( ($d == 0) && ($c == 0) && ($b == 0) && ($a != 0) );
$class = 'B' if ( ($d == 0) && ($c == 0) && ($b != 0) && ($a != 0) );
$class = 'C' if ( ($d == 0) && ($c != 0) && ($b != 0) && ($a != 0) );
$blaster_port = 135; # initialize the blaster port here
for ($i = 0; $i <= $imax; $i++)
{
   if ($destination_port[$i] == $blaster_port) {$off_port[$i] = 1} # tag Blaster portscans
        else {$off_port[$i] = 0}
   ($A_s_h[$i], $B_s_h[$i], $C_s_h[$i], $D_s_h[$i]) = split (/\./, $source_host[$i]);
        if ( # tag portscans that originate from your network
                ("$class" eq "A")  and  ("$A_s_h[$i]" eq "$a")
           ) { $local_bit[$i] = 1 }
        elsif ( ("$class" eq "B") && ("$A_s_h[$i]" eq "$a")
                && ("$B_s_h[$i]" eq "$b") ) {$local_bit[$i] = 1}
        elsif ( ("$class" eq "C") && ("$A_s_h[$i]" eq "$a") && ("$B_s_h[$i]" eq "$b")
                && ("$C_s_h[$i]" eq "$c") ) {$local_bit[$i] = 1}
        else {$local_bit[$i] = 0}
}
for ($i = 0; $i <= $imax; $i++) {
  unless (  ($off_bit[$i] == 1) or ($local_bit[$i] == 1) or ($off_port[$i] == 1)  ) {
$newline[$i] =  join    (" ", ("$month[$i]", "$day[$i]", "$time[$i]", "$source_host[$i]",
$source_port[$i], "$destination_host[$i]", $destina
tion_port[$i], "$flag[$i]", "$row_flag[$i]"));
        push (@clean_scan, @newline[$i]);
  } #
} # for
@clean_scan = uniq_list (@clean_scan);
print "\n\n\n############## CLEAN SCAN LINES ####################\n".
"(FTP false positives, local scans, and Blaster scans extracted)\n\n";
open (CLEAN, ">clean_scans.txt");
$n = 0;
foreach $line (@clean_scan) {
        print "$n, $line \n";
        print CLEAN "$n $line\n";
        $n++;
} # foreach
print "\n\##########TOTAL NUMBER OF CLEAN LINES: $n \n";
close (CLEAN);

######### LOCAL SCANS NOT BLASTER ####################
for ($i = 0; $i <= $imax; $i++) {
  if (  ($off_bit[$i] == 0) and  ($local_bit[$i] == 1) and ($off_port[$i] != 1)  ) {
        $nbline[$i] =  join     (" ", ("$month[$i]", "$day[$i]", "$time[$i]", "$source_host[$i]",
$source_port[$i],
                                "$destination_host[$i]", $destination_port[$i], "$flag[$i]",
"$row_flag[$i]"));
        push (@nb_scan, $nbline[$i]);
  } #
} # for
@nb_scan = uniq_list(@nb_scan);
$n = 0;
print "\n\n######### LOCAL SCANS NOT BLASTER:\n";
foreach $line (@nb_scan) {
        print "$n, $line \n";
        $n++;
} # foreach

######### BLASTER LOCAL SCANS ####################
## This section filters and report the local Blaster generated scans.
for ($i = 0; $i <= $imax; $i++) {
  if (  ($off_bit[$i] == 0) and  ($local_bit[$i] == 1) and ($off_port[$i] == 1)  ) {
```

```
$blaster_line[$i] =  join       (" ", ("$month[$i]", "$day[$i]", "$time[$i]",
"$source_host[$i]", $source_port[$i], "$destination_host[$i]", $destination_port[$i], "$flag[$i]",
"$row_flag[$i]"));
        push (@blaster_scan, $blaster_line[$i]);
        push (@blasted_ips, $source_host[$i]);
  } #
} # for
@blaster_scan = uniq_list(@blaster_scan);
@blasted_ips = uniq_list(@blasted_ips);
$n = 0;
foreach $line (@blaster_scan) {
#       print "$n, $line \n"; # uncomment these two print lines if you  like to
        $n++;
} # foreach

print "\n\n########################LOCAL BLASTED IPs:\n";
foreach $ip (@blasted_ips) { print "$ip\n";}
foreach $line (@clean_scan) {
        ($m, $d, $t, $source_host, $source_port, $destination_host, $destination_port, $flag, $row_flag)
= split (/ /, $line);
        $same_sh_dp = "$source_host $destination_port";
        $same_sh_dh = "$source_host $destination_host";
        $same_sh_dp{$same_sh_dp}++;
        $same_sh_dh{$same_sh_dh}++;
} # foreach
$pmax = 5; # max number different ports scanned
$hmax = 5; # max number of different hosts scanned

print "\n\n\n\n#################################################################\n";
print "SCANS SORTED by SAME SOURCE HOST, SAME DESTINATION PORT [PORTSCANS]\n";

foreach $same_sh_dp ( sort { $same_sh_dp{$b} <=> $same_sh_dp{$a} } keys %same_sh_dp ) {
        if ($same_sh_dp{$same_sh_dp} >= $pmax) { # if A
                ($source_host, $destination_port) = split (/ /, $same_sh_dp);
                push (@offender_ip, $source_host);
                $hostname = hostname("$source_host");
                unless ($hostname) { $hostname = "$source_host" }
                $serv = service($destination_port);
                if ($serv =~ /^[A-Za-z]+$/)     {
                        $serv = "$serv";
                } else {
                        $serv = "$destination_port";
                }
                print "\n\nPORTSCAN @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@\n";
print "SOURCE: $hostname ($source_host) scanned $same_sh_dp{$same_sh_dp} hosts  \nfor SERVICE: $serv
($destination_port) \n\n";
                  @count = grep { /\b$source_host\b/ && /\b$destination_port\b/  } @clean_scan;
                  foreach $ln (@count) {
                        print "$ln \n";
                  } # foreach
        } # if A
} # foreach

print "\n\n\n#################################################################\n";
print "SCAN SORTED by SAME SOURCE HOST, SAME DESTINATION HOST [HOSTSCANS]\n";
foreach $same_sh_dh ( sort { $same_sh_dh{$b} <=> $same_sh_dh{$a} } keys %same_sh_dh ) {
        if ($same_sh_dh{$same_sh_dh} >= $hmax) { # if A
                ($source_host, $destination_host) = split (/ /, $same_sh_dh);
                push (@offender_ip, $source_host);
                $hostname = hostname("$source_host");
                unless ($hostname) { $hostname = "$source_host" }
                print "\n\nHOSTSCAN @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@\n";
print "SOURCE: $hostname ($source_host) scanned host $destination_host\n".
"for more than $same_sh_dh{$same_sh_dh} ports\n\n";
                @count = grep { /\b$source_host\b/ && /\b$destination_host\b/  } @clean_scan;
                  foreach $ln (@count) {
                        print "$ln \n";
                  } # foreach
        } # if A#
} # foreach
```

```
print "\n\n\n############### ALL SCANS FROM EACH OFFENDER IP ####################\n\n";
@offender_ip = uniq_list(@offender_ip);
foreach $offender (@offender_ip) {
        $hostname = hostname($offender);
        print "\n\nOFFENDER IP: $offender\t\nHOSTNAME: $hostname\n";
        if ( hostname("$offender") ) {
           $domain = tld($hostname);
           # print "Hostname was found via TLD: $hostname\n";
           if ($domain) {print "DOMAIN: $domain\n"; print "Try to e-mail to abuse\@$domain. \n\n";}
        } else {
               print "Hostname was not found via TLD.\n\n";
        }
        @count = grep { /\b$offender\b/ } @clean_scan;
print "All scans originating from the same host:\n\n";
foreach $ln (@count) {
               print "$ln \n";
        }
}
#####################################
sub hostname {
        # This subroutine takes an IP address as an argument and returns its hostname.
        my $ip = "$_[0]";
        my @octet = split (/\./, $ip);
        my @host = gethostbyaddr (pack('C4', @octet), 2);
        $host = $host[0]; return $host;
}
#####################################

sub tld {
        my @host = split (/\./, $_[0]);
        my $tld = $host[-1];
        if (("$tld" eq "com") or ("$tld" eq "net") or
            ("$tld" eq "edu") or ("$tld" eq "gov") or
            ("$tld" eq "org") or ("$tld" eq "mil")) {
               $abuse_mail = join (".", ($host[-2], $tld));
                return $abuse_mail;
        } else { return ()}
}
#############################################################################

sub service {
        open (INSERV, "/etc/services") || die "Sorry, cannot read /etc/services. \n";
        my $srvc; my $port; my %service; my @ports; my $in_port;
        while (<INSERV>) {
                next if (/^#/); # skip comment lines
                ( my($srvc), my($port) ) = split;
                ($port, $protocol) = (split ('/', $port)); # get rid of the UDP & TCP
                if ("$protocol" eq "tcp") {
                        $service{$port} = $srvc;
                        push (@ports, $port);
                } # if
        } # while
        $in_port = $_[0];
        return "$service{$in_port}";
}
#########################################################################

sub uniq_list {
        ## this subroutine takes a @list and filter out duplicate items
        my (%seen) = ();
        my (@uniq) = ();
        foreach  $item (@_) {
                push (@uniq, $item) unless $seen{$item}++;
        } # foreach
return (@uniq);
} # sub
#######################################################
```

1.3 References

Fyodor. The Art of Portscanning, 1997
http://www.insecure.org/nmap/nmap_doc.html

Microsoft. What You Should Know About Microsoft Security Bulletin MS03-026, July 16, 2003
http://www.microsoft.com/security/security_bulletins/ms03-026.asp

Microsoft. What You Should Know About the Blaster Worm and Its Variants -
http://www.microsoft.com/security/incident/blast.asp

Microsoft. What You Should Know About Microsoft Security Bulletin MS03-039. September 2003
http://www.microsoft.com/security/security_bulletins/ms03-039.asp

Symantec. Detecting network traffic that may be due to RPC worms. September 2003.
http://securityresponse.symantec.com/avcenter/venc/data/detecting.traffic.due.to.rpc.worms.html

Mitre. CAN-2003-0352, 2003
http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CAN-2003-0352

Networm.org. *What happened with Blaster and Welchia?* 2003
http://www.networm.org/faq/

Cthuang. *Programming Language Using Perl -- Advanced Sorting*, 2000.
http://larc.ee.nthu.edu.tw/~cthuang/courses/ee2320/10_asort.html

Tom Christiansen & Nathan Torkington, *Perl Cookbook*, O'Reilly, CD Edition, 1998

Randal L. Schwartz & Tom Christiansen, *Learning Perl*, O'Reilly, Second Edition 1997.

Randal L. Schwartz with Tom Phoenix, *Perl Objects, References & Modules*, O'Reilly, 2003

IANA – Top Level Domains (TLD)
http://www.icann.org/tlds/

IANA – Root Zone Whois Information
http://www.iana.org/cctld/cctld-whois.htm

Chris Kuethe, GCIA Practival ver 3.0,
http://www.giac.org/practical/chris_kuethe_gcia.html - 3.0

Andrew Daviel, *Scan Reporter*
http://andrew.triumf.ca/pub/security/reporter/

PART 2 – PRACTICAL DETECTS

2. 1 Detect 2: ICMP Large Packet

I decided to choose these two alerts (see alerts below) mainly because I could not fingerprint the operating systems that generated them either using a passive fingerprinting chart[1], or by the calculated TTL value[2]. According to these two documents, only a Solaris 2.7 or a CISCO 12.0 IOS could have generated an initial TTL of 255. The two hosts, which sent the large ICMP packets, were computers used by individuals who had no real need to be running Solaris. Besides this fact, the two computers were not on-line continuously – and any serious UNIX user would not shutdown one's machine a couple of times per day.

The alerts are:

```
10/27-19:43:00.119856  [**] [1:499:3] ICMP Large ICMP Packet [**] [Classification:
Potentially Bad Traffic] [Priority: 2] {ICMP} 192.168.105.138 -> 192.168.1.103

10/27-08:30:54.123591  [**] [1:499:3] ICMP Large ICMP Packet [**] [Classification:
Potentially Bad Traffic] [Priority: 2] {ICMP} 192.168.140.83 -> 192.168.1.93
```

The packet data as extracted from unified logs using Barnyard:

```
[**] [1:499:3] ICMP Large ICMP Packet [**]
[Classification: Potentially Bad Traffic] [Priority: 2]
[Xref => http://www.whitehats.com/info/IDS246]
Event ID: 21187     Event Reference: 21187
10/28/03-00:43:00.119856 192.168.105.138 -> 192.168.1.103
ICMP TTL:253 TOS:0x0 ID:7973 IpLen:20 DgmLen:1500 DF
Type:8  Code:0  ID:39612   Seq:57072  ECHO
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
 [snip]
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................

[**] [1:499:3] ICMP Large ICMP Packet [**]
[Classification: Potentially Bad Traffic] [Priority: 2]
[Xref => http://www.whitehats.com/info/IDS246]
Event ID: 2114     Event Reference: 2114
10/27/03-13:30:54.123591 192.168.140.83 -> 192.168.1.93
ICMP TTL:253 TOS:0x0 ID:43265 IpLen:20 DgmLen:1500 DF
Type:8  Code:0  ID:39612   Seq:57072  ECHO
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
 [snip]
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

[1] Lance Spitzner.
[2] SWITCH, Default TTL Values in TCP/IP.

2.1.1 Source of Trace

An IDS sensor running Snort 2.0 under Solaris 8.0 triggered the alert. The sensor was placed inside of a campus network of an undisclosed university. This subnet contains public hosts, only. Similar to the majority of universities in the United States (U.S.), the whole campus is protected by limited packet filtering at the border routers. This particular subnet is physically segregated from the rest of the campus network by another router, which performs additional packet filtering. Due to this segregation, the packet filtering done at the routers, and the fact that the subnet consists exclusively of computers that need to be accessed by public, I will name this subnet the *DMZ subnet*.

2.1.2 Detect was generated by:

Snort IDS 2.0 with a set of rules modified to meet the requirements of this particular network and eliminate false positives specific to this network and the systems it connects.

The rule that triggered the alerts is:

```
alert icmp $EXTERNAL_NET any -> $HOME_NET any (msg:"ICMP Large ICMP Packet"; dsize:
>800; reference:arachnids,246; classtype:bad-unknown; sid:499; rev:3;)
```

This rule triggers if the following conditions are met:

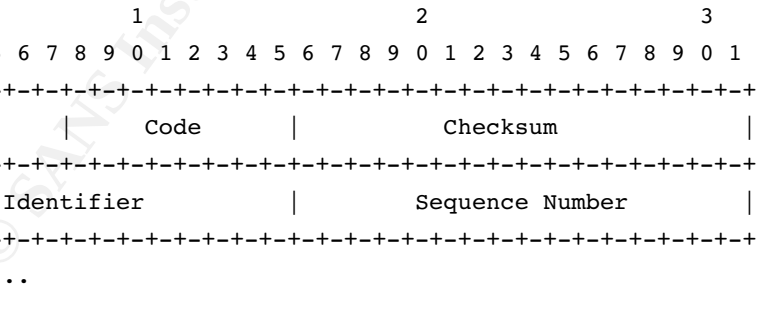
Rule Header:

a. The source is the EXTERNAL_NET as defined in file snort.conf;
b. The destination is HOME_NET as defined in file snort.conf;
c. The protocol is ICMP.

Rule Options:

a. The size of the packet should be bigger than 800 bytes.

Although this rule does not specifically mention a type and code of ICMP, I will focus on the packets that triggered this alarm, *ICMP echo request*. A graphical representation of an ICMP echo request[3] follows below:



RFC792 does not specifically give information about the size of the ICMP echo request packet. RFC792 specifies the following: “The data received in the echo message must be returned in the echo reply message.” Therefore, I was very interested to find out how a normal echo request / echo reply IP traffic would look. Here is what I found:

[3] RFC 792 - Internet Control Message Protocol.

Windows 2000:

```
# tcpdump -nn -v -X -s 1514 host 192.168.18.25 and icmp
tcpdump: listening on hme0

08:57:24.708692 192.168.18.25 > 192.168.1.107: icmp: echo request (ttl 125, id 12866, len 60)
0x0000   4500 003c 3242 0000 7d01 dcb8 8da1 1219        E..<2B..}.......
0x0010   8da1 016b 0800 9c57 0200 af04 6162 6364        ...k...W....abcd
0x0020   6566 6768 696a 6b6c 6d6e 6f70 7172 7374        efghijklmnopqrst
0x0030   7576 7761 6263 6465 6667 6869                  uvwabcdefghi

08:57:24.708816 192.168.1.107 > 192.168.18.25: icmp: echo reply (DF) (ttl 255, id 31423, len
60)
0x0000   4500 003c 7abf 4000 ff01 d23a 8da1 016b        E..<z.@....:...k
0x0010   8da1 1219 0000 a457 0200 af04 6162 6364        .......W....abcd
0x0020   6566 6768 696a 6b6c 6d6e 6f70 7172 7374        efghijklmnopqrst
0x0030   7576 7761 6263 6465 6667 6869                  uvwabcdefghi
```

As you can see, the data is a string of characters: a, b, c … i, the packet size is 60 bytes in length (20 bytes IP header + 8 bytes of ICMP header + 32 bytes of data). As per RFC792 specifications, the echo reply packet contains the date sent by the ICMP echo request packet.

RedHat Linux 9.0, kernel v 2.19:

```
# tcpdump -nn -v -X -s 1514 host 192.168.18.24 and icmp
tcpdump: listening on hme0

09:04:49.311816 192.168.18.24 > 192.168.1.107: icmp: echo request (DF) (ttl 61, id 0, len 84)
0x0000   4500 0054 0000 4000 3d01 0ee4 8da1 1218        E..T..@.=.......
0x0010   8da1 016b 0800 140d 324c 0001 db85 9e3f        ...k....2L.....?
0x0020   41dd 0b00 0809 0a0b 0c0d 0e0f 1011 1213        A...............
0x0030   1415 1617 1819 1a1b 1c1d 1e1f 2021 2223        .............!"#
0x0040   2425 2627 2829 2a2b 2c2d 2e2f 3031 3233        $%&'()*+,-./0123
0x0050   3435 3637                                     4567
09:04:49.311864 192.168.1.107 > 192.168.18.24: icmp: echo reply (DF) (ttl 255, id 41633, len
84)
0x0000   4500 0054 a2a1 4000 ff01 aa41 8da1 016b        E..T..@....A...k
0x0010   8da1 1218 0000 1c0d 324c 0001 db85 9e3f        ........2L.....?
0x0020   41dd 0b00 0809 0a0b 0c0d 0e0f 1011 1213        A...............
0x0030   1415 1617 1819 1a1b 1c1d 1e1f 2021 2223        .............!"#
0x0040   2425 2627 2829 2a2b 2c2d 2e2f 3031 3233        $%&'()*+,-./0123
0x0050   3435 3637                                     4567
```

This time the packet size is 84 bytes. Also, the ICMP echo reply returns the data sent by the ICMP echo Request packet.

Solaris 2.7:

```
# tcpdump -nn -v -X -s 1514 host 192.168.18.46 and icmp
tcpdump: listening on hme0

09:07:58.274636 192.168.18.46 > 192.168.1.107: icmp: echo request (DF) (ttl 252, id 786, len
84)
0x0000   4500 0054 0312 4000 fc01 4cbb 8da1 122e        E..T..@...L.....
0x0010   8da1 016b 0800 7321 27e8 0000 3f9e 783e        ...k..s!'...?.x>
0x0020   0003 ba13 0809 0a0b 0c0d 0e0f 1011 1213        ................
0x0030   1415 1617 1819 1a1b 1c1d 1e1f 2021 2223        .............!"#
0x0040   2425 2627 2829 2a2b 2c2d 2e2f 3031 3233        $%&'()*+,-./0123
0x0050   3435 3637                                     4567
```

```
09:07:58.274676 192.168.1.107 > 192.168.18.46: icmp: echo reply (DF) (ttl 255, id 12506, len
84)
0x0000   4500 0054 30da 4000 ff01 1bf3 8da1 016b        E..T0.@........k
0x0010   8da1 122e 0000 7b21 27e8 0000 3f9e 783e        ......{!'...?.x>
0x0020   0003 ba13 0809 0a0b 0c0d 0e0f 1011 1213        ................
0x0030   1415 1617 1819 1a1b 1c1d 1e1f 2021 2223        ..............!"#
0x0040   2425 2627 2829 2a2b 2c2d 2e2f 3031 3233        $%&'()*+,-./0123
0x0050   3435 3637                                     4567
```

The packet size is 84 bytes.

As we can see from the tcpdump captures, the size of a Windows 2000 ICMP echo request is 60 bytes, and the size of LINUX and SOLARIS ICMP echo request packets is 86 bytes. Definitely, an 800 bytes ICMP echo request packet is too large in size.

In order to obtain more context information, I customized this rule adding a "tag": `tag: host, 300, seconds, src`. This tag captures all the TCP/IP packets sent or received by the host that sent the large ICMP packet for 300 seconds after the alert triggers. I also set a tcpdump filter to capture all the traffic that was coming from two hosts, of which I knew the IP addresses that were sending large ICMP packets (192.168.105.138, and 192.168.140.83). The IP addresses mentioned previously were hosts on an internal network that I am allowed to scan (this was another reason for picking these specific hosts).

2.1.3 Probability the address was spoofed

Zero. A study of the tcpdump traffic captures to and from these two hosts shows following the TCP/IP flow:

```
# tcpdump -nn -r 192.168.105.138.dmp | more
19:42:59.691229 192.168.105.138 > 192.168.1.103: icmp: echo request (DF)
19:42:59.691245 192.168.105.138.49235 > 192.168.1.103.80: S 1395463477:1395463477(0) win
32768 <mss 1460,wscale 0,nop>
19:42:59.691686 192.168.1.103 > 192.168.105.138: icmp: echo reply (DF)
19:42:59.691738 192.168.1.103.80 > 192.168.105.138.49235: S 3523853718:3523853718(0) ack
1395463478 win 33580 <nop,wscale 0,mss 1460> (DF)
19:42:59.706535 192.168.105.138.49235 > 192.168.1.103.80: . ack 1 win 32768 (DF)
19:43:00.043275 192.168.105.138.49235 > 192.168.1.103.80: P 1:281(280) ack 1 win 32768 (DF)
19:43:00.043515 192.168.1.103.80 > 192.168.105.138.49235: . ack 281 win 33580 (DF)
19:43:00.044283 192.168.1.103.80 > 192.168.105.138.49235: P 1:529(528) ack 281 win 33580 (DF)
19:43:00.077815 192.168.105.138.49235 > 192.168.1.103.80: R 1395463758:1395463772(14) win
32768 (DF)
```

The traffic flow order is:

1. ICMP echo request
2. TCP SYN
3. ICMP echo reply
4. TCP SYN/ACK
5. TCP ACK
6. ACK PSH
7. RST

We have an ICMP echo request and its answer – the ICMP echo reply, then a TCP handshake and then the connection is closed using a RST.

A further study of the traffic reveals that after resetting the connection to port 80, the source host opens a connection to port 443.

```
19:43:00.077815 192.168.105.138.49235 > 192.168.1.103.80: R 1395463758:1395463772(14) win
32768 (DF)
19:43:00.087735 192.168.105.138.49236 > 192.168.1.103.443: S 1395724037:1395724037(0) win
32768 <mss 1460,wscale 0,nop> (DF)
19:43:00.087919 192.168.1.103.443 > 192.168.105.138.49236: S 2395942577:2395942577(0) ack
1395724038 win 33580 <nop,wscale 0,mss 1460> (DF)
19:43:00.090158 192.168.105.138.49236 > 192.168.1.103.443: P 1:55(54) ack 1 win 32768 (DF)
19:43:00.090291 192.168.1.103.443 > 192.168.105.138.49236: . ack 55 win 33526 (DF)
```

What follows next? The destination is an e-mail server offering e-mail access over HTTP. The user does not type in https:// as destination, she/he types http:// instead. The connection to HTTP is automatically reset and a connection to HTTPS is open – also automatically.

The fact that there is no sign that the packets were crafted, reinforced my conclusion that the traffic was not spoofed.

2.1.4 Description of the Attack

A *malicious*, abnormally large, ICMP packet can be used for:

Denial of service;

a. by system crash/freeze/reboot;

b. by high bandwidth utilization meant to saturate the link;

Covert communication channel.

A *legitimate*, abnormally large, ICMP packet can be used for Path MTU Discovery or it can be generated by error. Detailed descriptions of each of these possible Large ICMP packets will be presented in the Attack Mechanism section, below.

2.1.5 Attack Mechanism

Denial of Services by system crash aka "Ping of Death"[4]*.* TCP/IP specifications permit a TCP/IP packet of maximum size equal to 2^16-1 = 65535 bytes. A TCP/IP packet bigger than this size will crash the destination system (if the system is not patched for this vulnerability). Crashing is not the only possible behavior of a system when it receives a "ping of death packet" – the system can also freeze, or reboot. The reason ICMP was used, and not TCP or UPD, is probably for the simple fact that at the time of the "ping of death" discover – 1996 – a lot of networks were allowing ICMP pings and replies, ingress and outgress. At this time most operating systems were vulnerable to this attack.

In our case, the large ICMP echo request packet was not fragmented (DF flag set) and the size of the packet was 1500 bytes. Therefore, in our case, the Ping of Death attack is excluded.

Since this vulnerability was discovered in 1996, all operating systems have been patched against it.

Denial of Service by bandwidth utilization. The destination is flooded with a lot of ICMP large packets. If the packets are echo request, the destination computers will answer with an echo reply of the same size (according to RFC792), and will act as a traffic multiplicator.

[4] ISS. Ping of Death.

In our case, the total number of large ICMP packets per day is fairly low as a percentage of the total traffic. The absolute number of Large ICMP packets generated by these two specific sources per day is low also. Obviously, there is no Denial of Service for a 1 GB Internet connection.

Covert Communication Channel. The data in these packets is a string of "0" – there is no information transported. Also, as a lot of people have correctly noticed, nowadays, when steganography is largely available, covert channels are becoming obsolete. In our case no covert communication channel was used.

The data packet analysis of subsequent traffic originating from these two sources reveals what I was actually looking for: What OS has generated the large ICMP packets? The answer is Macintosh OS. Unfortunately, I was not able to determine what MacOS version. Based of the version of the browsers that were used, I believe thee two OS were installed recently and they most likely are MacOX.

```
16:07:35.832145 192.168.140.83.1766 > 192.168.1.93.80: P [tcp sum ok] 1:292(291) ack 1 win
32768 (DF) (ttl 253, id 6404, len 331)
[snip]
0x0050   5573 6572 2d41 6765 6e74 3a20 4d6f 7a69        User-Agent:.Mozi
0x0060   6c6c 612f 342e 3733 2028 4d61 6369 6e74        lla/4.73.(Macint
0x0070   6f73 683b 2049 3b20 5050 4329 0d0a 486f        osh;.I;.PPC)..Ho
[snip]
19:43:00.043275 192.168.105.138.49235 > 192.168.1.103.80: P [tcp sum ok] 1:281(280) ack 1 win
32768 (DF) (ttl 253, id 49960, len 320)
[snip]
0x00c0   7365 722d 4167 656e 743a 204d 6f7a 696c        ser-Agent:.Mozil
0x00d0   6c61 2f34 2e30 2028 636f 6d70 6174 6962        la/4.0.(compatib
0x00e0   6c65 3b20 4d53 4945 2035 2e30 3b20 4d61        le;.MSIE.5.0;.Ma
0x00f0   635f 506f 7765 7250 4329 0d0a 5541 2d4f        c_PowerPC)..UA-O
0x0100   533a 204d 6163 4f53 0d0a 5541 2d43 5055        S:.MacOS..UA-CPU
0x0110   3a20 5050 430d 0a45 7874 656e 7369 6f6e        :.PPC..Extension
 [snip]
```

Fortunately, the two to alerts are clear false positives generated by legitimate traffic.

I believe the Mac OS unsuccessfully tried to use a Path MTU Discovery Mechanism via ICMP large packets with null data (1500 bytes – the standard Ethernet packet size, not including the 14 bytes from the Ethernet headers/trailers). I don't know why the subsequent TCP connection is started before the ICMP echo reply arrives back to source. There is no time-out issue here, the network has very low latency, and the source is two hops away from destination. The Path MTU discovery via ICMP results are simply not waited for, and both systems advertise a MSS of 1460 octets (which correspond to a maximum IP packet of 1500 octets). According to RFC 1191 - Path MTU Discovery[5] - Some TCP implementations send an MSS option only if the destination host is on a non-connected network. However, in general, the TCP layer may not have the appropriate information to make this decision, so it is preferable to leave to the IP layer the task of determining a suitable MTU for the Internet path."

According to the same RFC 1191, if the Path MTU Discovery is not executed, the packets will use a default maximum size of 576 octets. In our case the maximum size is 1500 octets, which convinces me that the Path MTU Mechanism was executed not via ICMP Large pings, but via MSS TCP options.

[5] RFC 1191 - Path MTU discovery.

2.1.6 Correlations

1. An e-mail message on the Snort user list presents the hypothesis of large ICMP packets generated by normal MacOX TCP/IP traffic. In this e-mail message (http://archives.neohapsis.com/archives/snort/2002-11/0155.html) the author asks about a probable generation of large ICMP packets by Mac OX 10.
   "I am running snort -1.9.0 and it has oversight over a network of both MAC and Windows machines. I am receiving a very large number of detects on the icmp large packets rule more from inside my net than out. Does anyone know if the large ICMP packets are a trait of the MAC os 10."
   And an answer to this question (http://archives.neohapsis.com/archives/snort/2002-11/0161.html) is: "The G4s with OS X over here seem to be also causing these large-packet alerts. I don't know why, but I can tell you it is normal behavior."
2. An e-mail message at
   http://cert.uni-stuttgart.de/archive/incidents/2000/09/msg00076.html
   mentions a possible use of pings for MTU path Discovery:
   "Actually, a number of operating systems do the 'ping for Path MTU discovery' trick (most notably AIX 4.3 - it was supported in 4.3.0, but 4.3.3 made the DEFAULT) for both TCP and UDP connections. Path MTU discover is not intended to measure the "speed" of the connection, it is to discover the largest packet that can be sent without fragmenting. See RFC 1191 for the gory details."

2.1.7 Evidence of active targeting

My initial though was the "attack" (which later turned out to be a false positive) was targeting one of the enterprise mail servers. No other traffic was being generated by the same source IP, nor was any traffic directed to other hosts.

2.1.8 Severity

Criticality - The destination is an enterprise mail server, and it deserves a 5.

Criticality = 5.

Lethality - In the worse case scenario – an ICMP Denial of Service (and not a ICMP Large Packet false positive) the mail server will we be rendered unusable. I will give it a 5.

Lethality = 5

System countermeasures - This system is administered by a very security conscious system administrator. The sytems appears to be regularly patched and carefully monitored and it's "TCP wrapped". Logs are saved off line, on a separate system. Also, the system was base-lined and a very tight change management process is enforced. It is running ipfilter software. I will give it a 5.

System Countermeasures = 5

Network Countermeasures - And IDS is watching the traffic on this network. Only ACL filtering is done on the network traffic. I will give it a 3.

Network Countermeasures = 3

Severity = (Criticality + Lethality) – (System Countermeasures + Network Countermeasures) = (5+5) – (5+3) = 2

2.1.9 Defensive Recommendations

Besides the already installed defensive measures I would recommend:

1. Install a statefull firewall, and carefully set the ICMP traffic through the firewall. Special attention should be given to this because statefull firewalls (at least Checkpoint and CISCO PIX firewalls) do not treat ICMP as statefull traffic. This means that two rules have to be used for ICMP traffic – because each direction of the traffic requests a rule. For example, a rule will permit outgoing ping, another rule will permit incoming echo reply. From my experience a lot of network engineers tend to trade security for connectivity and high throughput, and permit all ICMP suite through the firewall. For example, it's common to turn on the "Accept ICMP" implied rule in the Checkpoint CheckPoint FW Global Settings (in CP FW 4.1: Policy -> Properties Setup -> Security Policy -> Implied Rules -> Accept ICMP).
2. Limit ICMP traffic to a percentage of total network traffic to prevent a Denial of Service by large ICMP packets.
3. Ensure that IMCP echo request is not allowed into the inside local networks.
4. Install Host IDS on the most sensitive systems.
5. Raise the awareness level of network engineers by making them aware of the security implications of allowing all ICMP traffic.

2.1.10 Multiple Choice Question:

An ICMP echo reply should return:

A. All the data sent in the ICMP echo request that generated it;

B. The Internet Header and the first 64 bites of data of the ICMP echo request that generated it

C. First 68 bytes of the ICMP echo request that generated it

D. Only the header of the ICMP echo request that generated it

ANSWER: A. RFC 792 states "The data received in the echo message must be returned in the echo reply message."

2.1.11 References

Lance Spitzner. "Passive fingerprinting chart, 2003
http://project.honeynet.org/papers/finger/traces.txt

SWITCH, Default TTL Values in TCP/IP
http://secfr.nerim.net/docs/fingerprint/en/ttl_default.html)

RFC 792 - Internet Control Message Protocol
http://www.faqs.org/rfcs/rfc792.html

RFC 1191 - Path MTU discovery
http://www.faqs.org/rfcs/rfc1191.html

Denial of Service by Large ICMP Packets
http://www.whitehats.com/info/IDS246

CERT® Advisory CA-1996-26 Denial-of-Service Attack via ping
http://www.cert.org/advisories/CA-1996-26.html

Oversized ICMP ping packets
http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-1999-0128

ISS. Ping of Death
http://www.iss.net/security_center/advice/Intrusions/2000012/default.htm

Ping of Death
http://www.insecure.org/sploits/ping-o-death.html

2.2 Detect 2: Scans for Open Proxy Servers

For this detect I chose three consecutive alerts:

```
1. [**] [1:615:3] SCAN SOCKS Proxy attempt [**]
[Classification: Attempted Information Leak] [Priority: 2]
05/16-07:16:45.634488 208.177.5.232:57168 -> 78.37.146.51:1080
TCP TTL:22 TOS:0x0 ID:7499 IpLen:20 DgmLen:40
******S* Seq: 0x9B18BA84  Ack: 0x0  Win: 0x800  TcpLen: 20
[Xref => http://help.undernet.org/proxyscan/]

2. [**] [1:620:2] SCAN Proxy (8080) attempt [**]
[Classification: Attempted Information Leak] [Priority: 2]
05/16-07:16:45.634488 208.177.5.232:57168 -> 78.37.146.51:8080
TCP TTL:22 TOS:0x0 ID:45318 IpLen:20 DgmLen:40
******S* Seq: 0x9B18BA84  Ack: 0x0  Win: 0x800  TcpLen: 20

3.[**] [1:618:2] SCAN Squid Proxy attempt [**]
[Classification: Attempted Information Leak] [Priority: 2]
05/16-07:16:45.634488 208.177.5.232:57168 -> 78.37.146.51:3128
TCP TTL:22 TOS:0x0 ID:38386 IpLen:20 DgmLen:40
******S* Seq: 0x9B18BA84  Ack: 0x0  Win: 0x800  TcpLen: 20
```

I chose them because:

1. They come from the same IP address - 208.177.5.232.
2. They are directed to the same IP address - 78.37.146.51.
3. Their arrival date and time are the same - 05/16-07:16:45.634488.
4. They have:
   a) the same sequence number (0x9B18BA84),
   b) the same ack number (0),
   c) the same source port (57168)
   d) the same TTL (22),
   e) the same window size (0x800 = 2048).
   f) The same IHL (IP Header Length) -20, TCP Length – 20, and Datagram Length -40.

Also, they belong to the same Snort classification: “Attempted Information Leak”.

These alerts were obtained reading back into Snort the "2002.4.16" binary file from http://www.incidents.org/logs/Raw. The command that I used to read back the file is:

```
snort -c /etc/snort/snort.conf -b -l ./nids_log -r 2002.4.16
```

The arguments of this command are:

-c /etc/snort/snort.conf - instructs Snort to look for the configuration file in /etc/snort (by default, Snort checks for the snort.conf file in /etc);

-b - instructs Snort to save the log file in a single binary file

-l ./nids_log –instructs Snort to log to this directory;

-r reads the 2002.4.16 file back in Snort.

The snort.conf file includes the most current Snort standard set of rules, minus porn, virus, icmp-info and a couple of other rules that I considered not to be relevant to our research.

Because IpLen: 20 + TcpLen: 20 = DgmLen: 40, one can infer that these packets do not have any payload attached.

To find unique MAC addresses I ran the following UNIX commands:

```
tcpdump -enn -r 2002.4.16 | awk '{print $2 "\n" $3}' | sort -u
```

where:

-e switch displays the MAC addresses;

-nn turn off the host name resolution and the service name resolver.

The unique MAC addresses that I found are: 0:0:c:4:b2:33, and 0:3:e3:d9:26:c0.

A quick search in the file "http://standards.ieee.org/regauth/oui/oui.txt" reveals that both MACs belong to CISCO, which means that the IDS sensor was placed between two CISCO routers.

Checking the tcpdumps one can clearly see that we are dealing with a class B owner and its network block of address is 78.37.x.x.

Also, the bad checksum displayed by tcpdump is due to the original IPs being modified to sanitize the traffic.

This command

```
tcpdump -nn -r 2002.4.16 | awk '{print $2 "\n"$4}' | awk -F \. '{print $5}' | awk -
F : '{print $1}' | more
```

displays all the ports used, both as source or destination, and

```
tcpdump -nn -r 2002.4.16 | awk '{print $2 "\n"$4}' | awk -F
 \. '{print $5}' | awk -F : '{print $1}' | sort -u | awk '{if ($1<=1023) print $1}'
```

displays all the "root" ports: 20, 21, 25, 515, 53, 80.

Some quick remarks:

1. Abnormal traffic is directed to port 515, which is normally used by the line printer daemon (lpd). In our case this port used by the *BackdoorQ* attack:

```
21:37:37.444488 255.255.255.255.31337 > 78.37.234.22.515: R 0:3(3) ack 0 win 0
```

More details about this attack can be read in the following GCIA practical:

http://www.giac.org/practical/GCIA/Al_Maslowski-Yerges_GCIA.pdf

2. All traffic to port 53 is made up of version DNS BIND queries looking for a vulnerable version of BIND.

```
02:57:15.024488 217.131.173.179.2290 > 78.37.129.135.53:  4660 [b2&3=0x80] TXT CHAOS?
version.bind. (30)
```

More about BIND vulnerabilities can be read at:

http://www.isc.org/products/BIND/bind-security.html

http://www.kb.cert.org/vuls/id/325431

http://www.cert.org/advisories/CA-2001-02.html

3. TCP scanning using reflective ports was also detected. For example:

```
21:16:38.944488 202.96.52.99.80 > 78.37.98.99.80: . ack 0 win 1400
```

In the “ephemeral port range” the traffic is made up of a lot of file sharing and P2P - Gnutella (port 6347, 6348, 6349), MSN Messenger (port 1863).

Sorting the traffic to the target I have discovered that all traffic generated by this host is made up exclusively of scans to ports 1080, 3128 and 8080, and every target’s scan has the same Seq number and is originating from the same port. See listing below:

```
tcpdump -r 2002.4.16 -nn host 78.37.146.51 | sort +5 -n
07:16:31.314488 208.177.5.232.57166 > 78.37.146.51.1080: S 406176085:406176085(0) win 2048
07:16:31.314488 208.177.5.232.57166 > 78.37.146.51.3128: S 406176085:406176085(0) win 2048
07:16:31.314488 208.177.5.232.57166 > 78.37.146.51.8080: S 406176085:406176085(0) win 2048
[snip]
```

Initially, I though that I was dealing with the obsolete and well known Ring0 Trojan (http://www.f-secure.com/v-descs/ringzero.shtml). But the Ring0 Trojan changes the Seq number on every scan, and this scan does not do this – our scan uses the same Seq number in all scans. Also, the Ring0 trojan changes the source port on every scan (this information about he source port and the Seq number of the packets sent by a Ring0 Trojan is extracted from the traffic presented in the following two links:

http://cert.uni-stuttgart.de/archive/intrusions/2003/06/msg00251.html and

http://archives.neohapsis.com/archives/sf/ids/2001-q1/0326.html)

To illustrate this the following paragraph shows a tcpdump capture of a nmap SYN scan obtained by this nmap scan: `nmap –P0 –p 3128,1080,8080:`

```
09:16:52.610130 192.168.1.107.34705 > 192.168.18.24.3128: S 1646658820:1646658820(0) win 2048
(DF)
09:16:52.610295 192.168.1.107.34705 > 192.168.18.24.1080: S 1646658820:1646658820(0) win 1024
(DF)
09:16:52.610398 192.168.1.107.34705 > 192.168.18.24.8080: S 1646658820:1646658820(0) win 2048
(DF)
```

	Nmap	Our Traffic	Ring0
Source Port	same	same	different
Seq #	same	same	different

To clarify this issue, I tried to determine if the source host is running Windows – The Ring0 Trojan is a Windows Trojan. Passive fingerprinting of this source reinforced my conclusion that I am not dealing with a Ring0 Trojan, but with possible separate Nmap Scans.

Here are the results of the passive fingerprinting.

```
$p0f -l -s 2002.4.16  | grep 208.177.5.232
p0f - passive os fingerprinting utility, version 2.0.2
208.177.5.232:57165 - NMAP syn scan (2) *
208.177.5.232:57166 - NMAP syn scan (2) *
208.177.5.232:57166 - NMAP syn scan (2) *
```

2.2.2 Detect was generated by

Snort 2.0 IDS running on Red Hat Linux 9, linux-2.4.20-20.9 kernel.

The rules that triggered the alarms are:

```
alert tcp $EXTERNAL_NET any -> $HOME_NET 1080 (msg:"SCAN SOCKS Proxy attempt"; flags:S,12;
reference:url,help.undernet.org/proxyscan/; classtype:attempted-recon; sid:615; rev:4;)
alert tcp $EXTERNAL_NET any -> $HOME_NET 8080 (msg:"SCAN Proxy \(8080\) attempt"; flags:S,12;
classtype:attempted-recon; sid:620; rev:3;)
```

As previously stated, all three alerts have only two characteristics that are not in common: the destination port and the IP ID number. Because IpLen: 20 + TcpLen: 20 = DgmLen: 40, these packets do not have any payload attached.

These packets are stimuli directed to IP 78.37.146.51, ports 1080,8080 and 3128 in order to elicit a response from the host – granted the host is listening on these ports.

According to the Snort User Manual (http://www.snort.org/docs/SnortUsersManual-2.0.1.pdf), the rules that trigger these alerts are similar, the only difference is the destination port. They trigger if:

The header conditions are met:

a. TCP Packets are coming from outside of the network (any not 78.37.x.x);

b. are directed towards the inside network (78.37.x.x);
c. have any source port number;
d. destination port should be 1080, 8080, or 3128.

and

The following option is met:

e. TCP Flags = SYN, regardless of the values of the reserved bits.

2.2.3 Probability the source address was spoofed

Very low. The attacker is trying to elicit a response from the target. If the packets were spoofed, the "attacker" would not get a response.

An abnormal low TTL of 22 indicates that the packets could have been crafted. If the attacker used nmap to run the scan, the "-g" switch could change the source port.

2.2.4 Description of attacks

This is a reconnaissance activity. The attacks are possible nmap TCP SYN scans run to find out if the targets are listening on ports 1080, 8080, or 3128.

These ports are normally used by: 1080 – SOCKS PROXY; 8080 – Web Cache, Wingate alternate, and HTTP alternate; 3128 – Squid, Reverse Tunnel Backdoor.

Proxy servers were originally created with the purposes of saving bandwidth by:

1. Web caching;
2. Internet connection sharing.

A problem occurs if the proxy server permits anonymous connections. In this case, it can be used to hide the identity of the user.

Some proxies are open proxies - as opposed to authenticated proxies - because of incorrect configurations, some are installed and set as open proxies, on purpose, by Trojans.

It is a common technique for modern IRC Servers to check if the IP of the IRC client hosts an open proxy.

Why somebody would scan for open proxies when there are a lot of public lists with open proxies? Usually a known open proxy is shared by a lot of malicious users, and hackers who want to have their own, exclusive, reliable, open proxy, with high speed Internet access.

Chaining open proxies decreases the chance of finding the identity they are hiding.

2.2.5 Mechanism of the attack

The "attacker" host performs a SYN (half open) scan of the target's ports 1080, 8080, and 3128. If this host is listening on these ports it will answer with a SYN/ACK. The scanner does not dot send the ACK to complete the TCP "handshake", and the connection times out after a period of time. SYN scans are less likely to be detected than "Connect Scans" (where the TCP handshake is completed), which are usually logged. For example, the default nmap TCP scan is SYN Scan if the user has root access rights; or TCP Connect, if the user does not have root privileges.

2.2.6 Correlations

There is no other connection to this target besides the scans for ports 8080, 1080, and 3128.

The "attacker" also scanned the following host - 78.37.19.143. Also, there is no other traffic flow to or from this host besides the scans for open proxies.

Unfortunately, I do not have access to other logs to correlate this reconnaissance with other security events.

The following article describes in detail the open proxy problem in the universities: *The Open Proxy Problem -Internet2 Members Meeting* - Arlington VA, Wednesday, April 9th, 2003 (http://www.internet2.edu/presentations/spring03/20030409-OpenProxy-StSauver.pdf)

Exposing the Underground: Adventures of an Open Proxy Server, by Joe Stewart (http://www.lurhq.com/open_proxy.pdf) is an article written in a plain and accessible language. It also presents the results of an "open proxy honeypot".

2.2.7 Active Targeting

The only traffic flow to or from this host is generated by the open proxy scans. There is no other indication that this specific target was chosen on purpose.

The scanner's IP resolves to 208.177.5.232.ptr.us.xo.net. XO is an Internet Service Provider. This excludes the possibility that the scanner is a "Road Runner Open Proxy Scanner" or an IRC Server scanning back to check if its client is not an open proxy.

2.2.8 Severity

Criticality. Since the only traffic flow that is directed to or is originating from this host is this series of scans, I assume that the host is not a server. Under this assumption, and the fact that I don't have other information, I would give this host an average level of criticality.

Criticality = 2

Lethality. I do not have enough information about this particular host and I assume the worst-case scenario - the target proxies any port.

Lethality = 4

System Countermeasures. Under the assumption that the target is a desktop computer owned by a student, and from my experience with academic networks, I will give it the lowest grade - 1.

System Countermeasures = 1

Networks Countermeasures. I assume that similar to other universities this one uses a NIDS (Network Intrusion Detection) to monitor the traffic and it also does limited ACL (Access Control List) filtering on the border routers. If this computer is connected to a student residential network I assume that a limited ACL filtering is done at the routers that serve the student dormitories.

Network Countermeasure = 3

Severity = (2 + 4) – (1 + 3) = 2

2.2.9 Defensive Recommendations

1. Install a statefull border firewall behind the perimeter router.

2. Periodically scan network for open proxies.
3. Install a gateway antivirus solution to protect against viruses that install Trojans and open proxies.
4. Carefully monitor egress traffic for signs of open proxies.
5. Create an ACL that permits only legitimate mail servers to connect outside to port 25 (to eliminate the SMTP connections generated by virusi or trojan software which use their own SMTP engine to send malicious e-mails). If this is not possible, monitor the amount of traffic originating from computers that are not e-mail servers and is directed to external e-mail servers. If this traffic is high – it is very likely an open proxy is relaying illegitimate, possible malicious, e-mails.

2.2.10 Multiple choice test question

To manually test if an open proxy can tunnel e-mail spam, first telnet to the open proxy port, then use the following command:

A. HEAD mailserver.domain.com:25 HTTP/1.0
B. GET mailserver.domain.com:25 HTTP/1.0
C. CONNECT mailserver.domain.com:25 HTTP/1.0
D. POST mailserver.domain.com:25 HTTP/1.0

Answer C. If the open proxy returns "200 CONNECTED" this means the server will permit SMTP tunneling to the mail server mailserver.domain.com.

2.2.11 References

The Open Proxy Problem -Internet2 Members Meeting -Arlington VA, Wednesday, April 9th, 2003
http://www.internet2.edu/presentations/spring03/20030409-OpenProxy-StSauver.pdf

Joe Stewart . *Exposing the Underground: Adventures of an Open Proxy Server*
http://www.lurhq.com/open_proxy.pdf

Nmap documentation
http://www.insecure.org/nmap/nmap_documentation.html

Snort User's Manual
http://www.snort.org/docs/SnortUsersManual-2.0.1.pdf

Richard Stevens, *TCP/IP Illustrated*, Volume I. Addison-Wesley Professional Computing Series.

2.3 Detect 3: SADMIN worm access:

The Snort Alerts triggered by the attack are:

```
09/29-06:01:00.824620  [**] [1:1375:5] WEB-MISC sadmind worm access [**]
[Classification: Attempted Information Leak] [Priority: 2] {TCP}
202.98.102.5:51186 -> 192.168.10000.83:80
```

```
09/29-06:01:02.405159  [**] [1:1375:5] WEB-MISC sadmind worm access [**]
[Classification: Attempted Information Leak] [Priority: 2] {TCP}
202.98.102.5:51391 -> 192.168.100.93:80
09/29-06:01:06.146785  [**] [1:1375:5] WEB-MISC sadmind worm access [**]
[Classification: Attempted Information Leak] [Priority: 2] {TCP}
202.98.102.5:51602 -> 192.168.100.118:80
09/29-06:01:13.421960  [**] [1:1375:5] WEB-MISC sadmind worm access [**]
[Classification: Attempted Information Leak] [Priority: 2] {TCP}
202.98.102.5:52217 -> 192.168.100.200:80
```

The log (payload) of the packet that triggered the first alert is:

```
[**] [1:1375:5] WEB-MISC sadmind worm access [**]
[Classification: Attempted Information Leak] [Priority: 2]
[Xref => http://www.cert.org/advisories/CA-2001-11.html]
Event ID: 2908     Event Reference: 2908
09/29/03-10:01:00.824620 202.98.102.5:51186 -> 192.168.100.83:80
TCP TTL:239 TOS:0x0 ID:23049 IpLen:20 DgmLen:58 DF
***AP*** Seq: 0x609B23A0  Ack: 0xB850559F  Win: 0x2238  TcpLen: 20
47 45 54 20 78 20 48 54 54 50 2F 31 2E 30 0D 0A   GET x HTTP/1.0..
0D 0A                                             ..
```

The payloads of the other three packets are similar to the payload shown above.

2.3.1 Source of Trace

The alert was triggered by an IDS sensor running Snort 2.0 under Solaris 8.0. The sensor is placed inside of a campus network of a University. This subnet connects only hosts that are "public". Similar to the majority of Universities in the USA, the whole campus is protected by limited packet filtering at the border routers. This particular subnet is physically segregated from the rest of the campus network by another router that does additional packet filtering. Because of this segregation and of the packet filtering done at the routers, and the fact that the subnet consists exclusively of computers that need to be accessed by public, I will name this subnet the *DMZ subnet*.

2.3.2 Detect was generated by

Snort IDS 2.0 with a set of rules modified to meet the requirements of this particular network and eliminate false positives specific to this network and the systems it connects.

The rule that triggered the alert is:

```
alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS $HTTP_PORTS (msg:"WEB-MISC sadmind worm access";
flow:to_server,established; content:"GET x HTTP/1.0"; offset:0; depth:15;
classtype:attempted-recon; reference:url,www.cert.org/advisories/CA-2001-11.html; sid:1375;
rev:5;)
```

The rule triggers if all following conditions are met:

Rule header:

Connection incoming from External Servers, any port, towards HTTP_SERVERS group of IPs and HTTP_PORTS - ports listening on HTTP, as defined in the "var HTTP_SERVERS" and "var HTTP_PORTS" in the file snort.conf

Rule Options:

a) The TCP connection is established (ACK flag set) and the direction of the packets is towards the server (*flow: to_server, established*);

b) The string "GET x HTTP/1.0" (content:"GET x HTTP/1.0") is found in the first fifteen bytes of the payload (offset:0; depth:15).

Using GET to request a non-existent file, in our case this filename is “x”, the worm elicit, from the web server, an “error 400 -- bad request”.

Here we have this HTTP command executed on an Apache Server:

```
GET x HTTP /1.0
HTTP/1.1 400 Bad Request
Date: Tue, 30 Sep 2003 12:43:20 GMT
Server: Apache/1.3.26 (Unix) mod_fastcgi/2.2.12 mod_ssl/2.8.10
OpenSSL/0.9.6g
```

And the same command executed on an IIS/4.0 web server:

```
GET x HTTP /1.0
HTTP/1.1 400 Bad Request
Server: Microsoft-IIS/4.0
```

An e-mail message posted on a list (http://www.dshield.org/pipermail/list/2001-December/002015.php) offers a possible explanation of why the name of this non-existent file is “x”: “It seems these are script kiddies following the steps of an old article by a Spanish hacker. The Spanish hacker used "GET x" as an example “on how to get info about the web.”

2.3.3 Probability the source was spoofed

Extremely Low. The packet that triggered the alert has the TCP flags set to AP - therefore it belongs to an established connection. Also, the attacker needs to elicit a response from the destination host. There is not sign that the TCP packets are crafted.

2.3.4 Description of the attack

Passive fingerprinting of the source tells us that the attacker computer may be running Solaris 2.x (initial TTL was 255, and the window 8760).

It appears that the attacker is a Solaris OS compromised by the SADMIND/IIS worm. This is a self-propagating worm that exploits a Sun Solaris sadmind buffer overflow to compromise the target. If successful, the attacker can execute arbitrary code with root privileges (sadmind runs as root). Solaris systems that are compromised can be used by this worm to scan and compromise other Solaris systems and deface IIS web servers.

2.3.5 Mechanism of the attack

First, the attacker, a compromised Solaris OS, scans a randomly selected class B network to identify the hosts that are listening on port 80. In our case the attacker scanned (IDS portscan logs from the same sensor):

```
Sep 29 04:17:38 202.98.102.5:37936 -> 192.168.100.82:80 SYN ******S*
Sep 29 04:17:38 202.98.102.5:37938 -> 192.168.100.84:80 SYN ******S*
Sep 29 04:17:38 202.98.102.5:37951 -> 192.168.100.97:80 SYN ******S*
Sep 29 04:17:38 202.98.102.5:37937 -> 192.168.100.83:80 SYN ******S*
Sep 29 04:17:38 202.98.102.5:37947 -> 192.168.100.93:80 SYN ******S*
Sep 29 04:17:38 202.98.102.5:37948 -> 192.168.100.94:80 SYN ******S*
Sep 29 04:17:41 202.98.102.5:38209 -> 192.168.100.101:80 SYN ******S*
Sep 29 04:17:41 202.98.102.5:38210 -> 192.168.100.102:80 SYN ******S*
Sep 29 04:17:41 202.98.102.5:38211 -> 192.168.100.103:80 SYN ******S*
Sep 29 04:17:41 202.98.102.5:38213 -> 192.168.100.105:80 SYN ******S*
Sep 29 04:17:41 202.98.102.5:38215 -> 192.168.100.107:80 SYN ******S*
Sep 29 04:17:41 202.98.102.5:38219 -> 192.168.100.111:80 SYN ******S*
Sep 29 04:17:41 202.98.102.5:38214 -> 192.168.100.106:80 SYN ******S*
Sep 29 04:17:41 202.98.102.5:38220 -> 192.168.100.112:80 SYN ******S*
Sep 29 04:17:41 202.98.102.5:38221 -> 192.168.100.113:80 SYN ******S*
Sep 29 04:17:41 202.98.102.5:38223 -> 192.168.100.115:80 SYN ******S*
Sep 29 04:17:41 202.98.102.5:38224 -> 192.168.100.116:80 SYN ******S*
Sep 29 04:17:41 202.98.102.5:38227 -> 192.168.100.119:80 SYN ******S*
Sep 29 04:17:41 202.98.102.5:38222 -> 192.168.100.114:80 SYN ******S*
Sep 29 04:17:41 202.98.102.5:38230 -> 192.168.100.122:80 SYN ******S*
Sep 29 04:17:41 202.98.102.5:38226 -> 192.168.100.118:80 SYN ******S*
Sep 29 04:17:45 202.98.102.5:38516 -> 192.168.100.200:80 SYN ******S*
```

Then, the worm queries port 111 (sunrpc - portmapper) to obtain the port on which sadmin service would listen and then tries a buffer overflow of the sadmin service.

In our case port 111 is blocked at the border router using ACL filtering. This is why the attack is unsuccessful, stops at this stage, and we do not see any other alerts. The Snort rules, which could have triggered the alerts if the attack had had been successufll, and it had reached the next stage, are:

```
alert tcp $EXTERNAL_NET any -> $HOME_NET 111 (msg:"RPC portmap sadmind request TCP";
flow:to_server,established; content:"|00 00 00 00|"; offset:8; depth:4; content:"|00 01 86
A0|"; offset:16; depth:4; content:"|00 00 00 03|"; distance:4; within:4;
byte_jump:4,4,relative,align; byte_jump:4,4,relative,align; content:"|00 01 87 88|";
within:4; reference:arachnids,20; classtype:rpc-portmap-decode; sid:1272; rev:8;)

alert udp $EXTERNAL_NET any -> $HOME_NET 111 (msg:"RPC portmap sadmind request UDP";
content:"|00 00 00 00|"; offset:4; depth:4; content:"|00 01 86 A0|"; offset:12; depth:4;
content:"|00 00 00 03|"; distance:4; within:4; byte_jump:4,4,relative,align;
byte_jump:4,4,relative,align; content:"|00 01 87 88|"; within:4; reference:arachnids,20;
classtype:rpc-portmap-decode; sid:585; rev:5;)

alert udp $EXTERNAL_NET any -> $HOME_NET any (msg:"RPC sadmind UDP NETMGT_PROC_SERVICE
CLIENT_DOMAIN overflow attempt"; content:"|00 00 00 00|"; offset:4; depth:4; content:"|00 01
87 88|"; offset:12; depth:4; content:"|00 00 00 01|"; distance:4; within:4;
byte_jump:4,4,relative,align; byte_jump:4,4,relative,align; byte_jump:4,124,relative,align;
byte_jump:4,20,relative,align; byte_test:4,>,512,4,relative; reference:cve,CVE-1999-0977;
reference:bugtraq,866; classtype:attempted-admin; sid:1911; rev:6;)

alert tcp $EXTERNAL_NET any -> $HOME_NET any (msg:"RPC sadmind TCP NETMGT_PROC_SERVICE
CLIENT_DOMAIN overflow attempt"; flow:to_server,established; content:"|00 00 00 00|";
offset:8; depth:4; content:"|00 01 87 88|"; offset:16; depth:4; content:"|00 00 00 01|";
distance:4; within:4; byte_jump:4,4,relative,align; byte_jump:4,4,relative,align;
byte_jump:4,124,relative,align; byte_jump:4,20,relative,align; byte_test:4,>,512,4,relative;
```

```
reference:cve,CVE-1999-0977; reference:bugtraq,866; classtype:attempted-admin; sid:1912;
rev:5;)
alert udp $EXTERNAL_NET any -> $HOME_NET any (msg:"RPC sadmind UDPPING"; content:"|00 00 00
00|"; offset:4; depth:4; content:"|00 01 87 88|"; offset:12; depth:4; content:"|00 00 00
00|"; distance:4; within:4; reference:bugtraq,866; classtype:attempted-admin; sid:1957;
rev:3;)
alert tcp $EXTERNAL_NET any -> $HOME_NET any (msg:"RPC sadmind TCP PING";
flow:to_server,established; content:"|00 00 00 00|"; offset:8;depth:4; content:"|00 01 87
88|"; offset:16; depth:4; content:"|00 00 00 00|"; distance:4; within:4;
reference:bugtraq,866; classtype:attempted-admin; sid:1958; rev:3;)
```

If had been successful, the sadmin buffer overflow whould have added "++"to the rhosts file of the user root(echo "++">~.rhosts). The $/user_home/.rhosts file provides authentication for all "r" services - rlogin, rsh, rcp. This file specifies remote users that are considered "trusted". Trusted users are allowed to access the local system without a password. A "rhost" file that contains ++ will "trust" any user from any host. This user can use any "r" services as root, without being authenticated. And, of course, a compromised host that can be used to attack other hosts.

In order to conceal its presence, the worm starts its own identd process that will run with its own configuration file [/usr/sbin/inetd -s /tmp/.f and not /etc/ident.conf]. This way, the original /etc/inetd.conf will not be modified, avoiding a possible detection by a security checksum type HIDS. The second inetd configuration file - /tmp/.f will not be shown by a "short" ls command.

Then, the worm tries to:

a) Multiply itself by attacking other hosts;

b) Scan for MS/IIS web servers vulnerable to MS00-078 Web Server Folder Traversal Vulnerability. These vulnerable web servers are then defaced.

2.3.6 Correlations

Log files of the traffic collected on the border router show a TCP campus wide scan originating from this source. To save space I have included only three lines of this log. Each packet is listed as having 310 octets.

```
202.98.102.5     192.168.165.212  6     61058    80       310        6
202.98.102.5     192.168.165.212  6     61109    80       310        6
202.98.102.5     192.168.165.212  6     61110    80       310        6
```

A "whois" query on this IP returned the following data:

e-mail: ipadmin@my-public.sc.cninfo.net

This is a Chinese company - Data Communication Bureau Of Sichuan Province Chengdu, PR China. An e-mail message was sent to the host owner. I have not yet received any reply to this complaint. The source was listed on the incidents.org site

http://isc.incidents.org/source_report.html?order=&subnet=202.098.102

and Dshields: http://www.dshield.org/ipinfo.php?ip=202.098.102.005

Also, an e-mail regarding an attack originating from this machine was found on the web (http://lists.jammed.com/incidents/2001/07/0218.html): "The Second IP address is an

employee of China Telecom who thinks he's a bit of a hacker. The hacker tried to attack my firewall on Jun 11th and 14th unsuccessfully:

```
Packet log: input DENY eth0 PROTO=6 202.99.64.113:33408 x.x.x.x:111
"trying to get my RPC info or overflow bug"
```

2.3.7 Evidence of active targeting

The entire campus network was port scanned by the host infected with the sadmin worm. Then, this worm attacked web servers running Solaris. The attack was clearly aimed at infecting web servers running Solaris in order to multiply the worm.

2.3.8 Severity

Criticality. Two of the targets were enterprise web servers. If it had been successful, the attack would have compromised these servers and would have used them as a platform to attack others.

Criticality: 5

Lethality. If successful, this attacker gives root access to the system.

Lethality: 5

System Countermeasure. The system is regularly patched and carefully monitored and it's TCP wrapped. Logs are saved off line, on another system. Also, the system was base-lined and a very tight change management is enforced.

System Countermeasure: 4

Network Countermeasures. And IDS is watching the traffic on this network. Only ACL filtering is done on the network traffic.

Network Countermeasures: 3

Severity = (Criticality + Lethality) – (Sys Counterm + Net Counterm) = (5+5) - (4+3) = 3

2.3.9 Defensive Recommendations

1. Install a statefull inspection firewall to protect all public servers.
2. Install a host IDS on the most sensitive hosts.
3. If a statefull firewall cannot be installed, install a server firewall on the most sensitive servers.
4. Periodically scan the network for sign of "sadmind" compromise hosts listening on port 600.
5. Write a shell script that would periodically check for any change in the list of listening ports of the hosts in the DMZ.
6. Monitor egress traffic for signs of traffic originating from possible compromised hosts.
7. Employ scripts to check web logs for signs of attacks, misuse etc.

2.1.10 Multiple Choice Test Question

The following Snort log shows the IP header and the payload of the

sadmind attack:

```
[**] [1:1375:5] WEB-MISC sadmind worm access [**]
[Classification: Attempted Information Leak] [Priority: 2]
[Xref => http://www.cert.org/advisories/CA-2001-11.html]
Event ID: 2908     Event Reference: 2908
09/29/03-10:01:00.824620 202.98.102.5:51186 -> 192.168.100.83:80
TCP TTL:239 TOS:0x0 ID:23049 IpLen:20 DgmLen:58 DF
***AP*** Seq: 0x609B23A0  Ack: 0xB850559F  Win: 0x2238  TcpLen: 20
47 45 54 20 78 20 48 54 54 50 2F 31 2E 30 0D 0A  GET x HTTP/1.0..
```

What operating system is the attacker host running?

A. Solaris OS

B. Windows 2000

C. AIX

D. DOS 5.2

Answer A. Besides the fact that "samind" is indicative of a Solaris OS, the initial TTL of the packet was 255 and the windows size is 0x2238 (8760) prompt to Solaris 2.x. Only Solaris 2.x and CISCO v12.x can send packets with TTL=255.

2.3.11 Questions and Answers

Joe Bowling, regarding this paraghraph:

"If successful, the attacker can execute arbitrary code with root privileges (sadmind runs as root). Solaris systems that are compromised can be used by this worm to scan and compromise other Solaris systems and deface IIS web servers."

Question:

>>>>>>>>Can you explain how the IIS servers are defaced? Is this part of the worm activity or are you making a general statement that compromised boxes can be used to attack other hosts?

Answer:

An answer is in the *Mechanism of the attack* section, in the: "b) Scan for MS/IIS web servers vulnerable to MS00-078 Web Server Folder Traversal Vulnerability. These vulnerable web servers are defaced."

Also, this paragraph:

“Solaris systems that are compromised can be used by this worm to scan and compromise other Solaris systems and deface IIS web servers.”

will be changed to:

“Solaris systems that are infected with this worm will by used by the worm to compromise other Solaris systems and deface Microsoft IIS servers vulnerable to the MS00-078 Web Server Folder Traversal Vulnerability.”

This paragraph:

"If successful the attack adds "++"to the rhosts file of the user root (echo "++">~.rhosts) in order to copy itself to the new machine using rcp. It runs a second inetd to open a root shell that listens on tcp port 600. This means that the target would be owned."

will be changed to:

“If had been successful, the sadmin buffer overflow would have added "++"to the rhosts file of the user root(echo "++">~.rhosts). The $/user_home/.rhosts file provides authentication for all "r" services - rlogin, rsh, rcp. This file specifies remote users that are considered "trusted". Trusted users are allowed to access the local system without a password. A "rhost" file that contains ++ will "trust" any user from any host. This user can use any "r" services as root, without being authenticated. And, of course, a compromised host can be used to attack other hosts.

In order to conceal its presence, the worm starts a second identd that will run with its own configuration file [/usr/sbin/inetd -s /tmp/.f]. This way, the original /etc/inetd.conf will not be modified avoiding a possible detection by a security checksum type HIDS. The second inetd configuration file - /tmp/.f will not be shown by a "short" ls command.”

Don Murdoch, multiple questions.

Question 1:

> The rule that triggered the alert is:
> alert tcp $EXTERNAL_NET any -> $HTTP_SERVERS $HTTP_PORTS
> (msg:"WEB-MISC sadmind worm access";
> flow:to_server,established; content:"GET x HTTP/1.0";
> offset:0; depth:15; classtype:attempted-recon;
> reference:url,www.cert.org/advisories/CA-2001-11.html;
> sid:1375; rev:5;)

Is this a default rule or a custom rule (state).

Answer:

This is not a custom (local) rule. Its SID is 1375. According to the Snort manual, SIDs in the range 100-1000000 are reserved for default rules.

Question 2:

Extremely Low. The packet that triggered the alert has the
> TCP flags set to AP - therefore it belongs to an established
> connection. Also, the attacker needs to elicit a response
> from the destination host.

Wait - this is an HTTP transaction to a web server, yes?
How can a packet be spoofed when ... A tcp handshake

needs to have happened by now in order to send the HTTP GET?

In other words, you need to re-evaluate the phrase "Extremely Low".

Answer:

In used “Extremely Low” and not “zero” because I intended to leave room for a possible TCP Session Hijacking, where the hijacker host spoofs an end of an established TCP connection. I also checked what other GIAC certified persons wrote in their GIAC papers about spoofing and established TCP connections and their answers ranged from zero to low. I decided to take the way that is in the middle.

Question 3:

How does the exploit get the ability to actually do this - what gives it capability to get onto?

Answer:

This paragraph

“If successful the attack adds "++"to the rhosts file of the user root(echo "++">~.rhosts) in order to copy itself to the new machine using rcp. It runs a second inetd to open a root shell that listens on tcp port 600. This means that the target would be owned.”

will be changed to:

If had been successful, the sadmin buffer overflow would have added "++"to the rhosts file of the user root(echo "++">~.rhosts). The $/user_home/.rhosts file provides authentication for all "r" services - rlogin, rsh, rcp. This file specifies remote users that are considered "trusted". Trusted users are allowed to access the local system without a password. A "rhost" file that contains ++ will "trust" any user from any host. This user can use any "r" services as root, without being authenticated. And, of course, a compromised host can be used to attack other hosts.

In order to conceal its presence, the worm starts its own identd process that will run with its own configuration file [/usr/sbin/inetd -s /tmp/.f and not /etc/ident.conf]. This way, the original /etc/inetd.conf will not be modified avoiding a possible detection by a security checksum type HIDS. The second inetd configuration file - /tmp/.f will not be shown by a "short" ls command.

Question 4:

As far as I know, there can only be one inetd on a system at a time? Personally, I don't know - but this sounds a bit off. Does it modify the inetd.conf file?

Answer:

In this message, http://www.netsys.com/bsdi-users/1998-03/0339.html, the sender of the e-mail complains about a second inetd process running on his server:

"Absolutely nothing looked wrong, except there were TWO copies of inetd running. I can't see how this could happen, and I don't see why it would stop things working, surely the second one would not be able to bind to the ports?"

It does not modify the *inetd.conf*, the second inetd process uses its own configuration file. This is the main reason why a second inetd process would be started – to avoid detection via modification of the inetd.conf.

2.3.12 References

Lance Spitzner, 2000
http://project.honeynet.org/papers/finger/traces.txt

Sophos, Solaris/Sadmind.worm
http://www.sophos.com/virusinfo/analyses/unixsadmind.html

CERT® Advisory CA-2001-11 sadmind/IIS Worm, 2001
http://www.cert.org/advisories/CA-2001-11.html

Microsoft Security Bulletin (MS00-078), Patch Available for 'Web Server Folder Traversal' Vulnerability
http://www.microsoft.com/technet/treeview/default.asp?url=/technet/security/bulletin/MS00-078.asp

PART 3 – Analyze this

3.1 Executive Summary

The analysis was performed on 1,289,101 alerts (covering a period of time starting on November 5th at 00:16 through November 10th at 00:07); 11,816,781 lines of scans (Nov 5th 00:00 through Nov 9th 23:52); 9,194 alerts of OOS (Out OF Spec – Oct 27 at 00:06 through Oct 29 at 14:30). Regarding the dates/filenames, all OOS files posted on GIAC are actually the same file - *oos_report_031027*.

By matching destination ports and nslookup names of UMBC IPs I concluded that the network where the alerts, scans and oos files were triggered and created has the 130.85.0.0 class B block assigned.

The analyst had no access to the payload of the packets that triggered the alerts. Access to this data would have helped a lot to screen out the false positives.

The alert analysis reveals that the false positives outnumber by far the malicious attacks making a correct analysis very difficult.

Top volume attacks and false positives are explained. Top scanners and scanned machines, are also presented.

These hosts that might be compromised and need immediate attention: 130.85.190.97, 130.85.111.34

These hosts should be immediately checked for backdoors and trojans:

130.85.15.198, 130.85.80.16, 130.85.81.18, 130.85.66.53, 130.85.42.9, 130.85.97.153

The following hosts may be infected with the Blaster/Welchia worms and need immediate attention: 130.85.70.129, 130.85.80.243, 130.85.84.194, 130.85.111.72, 130.85.112.179, and 130.85.153.174.

The users of the following hosts should be checked for activities that are not permitted by the Acceptable User Policy: 130.85.21.37, 130.85.21.116, 130.85.21.116, 130.85.21.68, 130.85.21.69, 130.85.21.79, and 130.85.21.92.

3.1.1 General Recommendations:

1. As expected, gaming, P2P and Instant Messengers traffic make the bulk of the analyzed traffic, typical for a University Campus. This type of traffic creates the majority of the false positive alerts and scans. It would be far more beneficial to stop monitoring this kind of traffic and instead use the freed up resources, time, and money to create an awareness program about basic computer security, DMCA (Digital Millennium Copyright Act) and the impact of games, P2P and Instant Messaging on computer security.
2. Redefine the IDS detection ruleset to eliminate a lot of false positives and noise to provide a clearer image to the analyst in charge of this IDS.
3. Filter out at the border routers – both ingress and egress – ports 135, 137, 138, 139, 445. Also, blocking these ports at the dormitory routers will help containing computer virusi and worms within the residential/dormitory networks. Filter out – both ingress and outgress – ports 69 (TFTP), 111 (SUN RPC) and try to create a black list of ports and IPs, which should be blocked. For example, all computers that are infected with Klez connect to 25.0.0.0.
4. To stop the spam, permit egress connections to port 25 SMTP originating only from approved SMTP mail servers. A lot of virusi and worms have their own built-in SMTP engines, which connect directly to destination mail servers. One may wish to monitor these connections.
5. Place IDS sensors on the inside networks to analyze internal traffic. This way one will be able to catch the plethora of virusi and worms. In academic networks the most lethal dangers are having the internal network as both origin and destination point.
6. Create an effective security awareness program to explain the need for:
   - Choosing strong passwords;
   - Installing and regularly updating Anti-virus software;
   - Regularly installing operating system patches.

3.2 List of Files

Alert Files	Scan Files	Out of Specification Files
alert.031105	scans.031105	oos_report_031105
alert.031106	scans.031106	oos_report_031106
alert.031107	scans.031107	oos_report_031107
alert.031108	scans.031108	oos_report_031108

alert.031109	scans.031109	oos_report_031109

3.3 Alerts

3.3.1 Alert files

After decompressing the alert files, their size increased approximately ten to twelve times. The combined size of these alerts is huge – 155 MB - and it would take a lot of time and computer resources to process them. In order to bring the size of these files to a more manageable size, I had to extract some false positives and other information that is not relevant to our purpose. Therefore, after consulting other papers and see what other GIAC certified persons did, I decided to eliminate the *spp_portscan* alerts. After eliminating these alerts (scans are going to be analyzed with the scan log files anyway) the combined alerts file size dropped to 37.5MB.

In order to avoid *snortsnarf* processing errors, I replaced the MY.NET.x.x with 255.255.x.x using a Perl script. Before making the replacement, I made sure that 255.255.x.x was not used in the alert file. Later on, it was pretty clear that I was dealing with the UMBC network, which has allocatted a class B block of IPs: 130.85.1-255.1-255. To use actual IPs, I changed the 255.255. to 130.85.

The alert file has now 384,352 alerts. Out of these alerts, 19,736 are “MY.NET.30.4 activity”, and 16,993 were “MY.NET.30.4 activity”. The alert "ICMP SRC and DST outside network" was in 255,409 lines, which is about 66% of the total alerts! "TCP SRC and DST outside network" - 166, and "connect to 515 from inside" – 13,719. After eliminating these alerts, the size of the alert file decreased to 78,329 alerts or 79.8 MB.

65 alert lines were incorrectly formatted - starting with a “;”. 348 alert lines were incorrectly formatted – starting with “ ->”. 291 alerts were incorrectly formatted (they had more than two [**] delimiters within the same line). All these incorrectly formatted alerts were eliminated. The Perl scripts I wrote to clean the alert file can be found in the *Appendix A*.

The clean alert file has now 77,649 alerts.

The alerts are listed in a single line according to the following pattern:

```
<date and time> [**] Snort Message [**] <source IP>:<source port> → <dest IP>:<dest port>
```

This format was most likely obtained by eliminating *Classification, Priority* and *Protocol* from the original fast alert mode format which is returned by Snort when it is started with the *–A fast* option:

```
<date and time> [**] Snort Message [**] [Classification: y] [Priority: x] {PROTOCOL} <source
IP>:<source port> → <dest IP>:<dest port>
```

3.3.2 Quick analysis of alerts that were eliminated

1. *MY.NET.30.4 activity* and *MY.NET.30.5 activity* – these are alerts that have relevance only for the person who created the custom Snort detection rules which triggered them. An nslookup on these two UMBC IPs returned: lan1.umbc.edu and lan2.umbc.edu. Both these two addresses are relevant only to an insider of that network.

2. *ICMP SRC and DST outside network* and *TCP SRC and DST outside network*. This IDS sensor is placed outside of the university border router and not inside (there are no alerts having both source and destination MY.NET). Possible attacks that do not have MY.NET as source or destination are not relevant to our purposes.
3. *Connect to 515 from inside.* Two years ago port 515 lpd printing service used to be on the Top 20 SANS vulnerability list. This vulnerability was exploited by worms like Red Worm (aka Adore) or lpdw0rm. Since this is an old vulnerability, and I assume it is been already patched, this alert is a false positive generated by legitimate traffic to UNIX lpd served printers. For more details, read below:

3.3.3 Hosts, and ports lists

Using a custom Perl script (servers.pl), I found that the following servers are listening on ports lower than 1023:

HTTP Servers: 130.85.24.44:80 (userpages.umbc.edu), 130.85.60.14:80 (www.gl.umbc.edu), 130.85.24.34:80 (www.umbc.edu), 130.85.6.7:80 (umbc7.umbc.edu), 130.85.5.20:80 (centrelearn.umbc.edu), 130.85.29.3:80 (bb-app4.umbc.edu), 130.85.100.165:80 (web1.cs.UMBC.EDU), 130.85.60.38:80 (linux1.gl.umbc.edu), 130.85.24.33:80 (my.umbc.edu)

POP3 Servers: 130.85.12.4:110 (mail.umbc.edu), 130.85.60.17:110 (alumni.umbc.edu)

SMTP Servers: 130.85.12.6:25 (mxin.umbc.edu), 130.85.24.20:25 (listproc.umbc.edu),

HTTPS Servers: 130.85.24.74:443 (webmail.umbc.edu), 130.85.12.7:443 (webauth.umbc.edu)

TFTP Servers: 130.85.60.16:69 (linux2.gl.umbc.edu).

Destination ports lower than 1023: 25, 69, 80, 113, 137, 427, 443

Source ports higher than 1023 and destination higher than 1023: 1075, 1220, 1221, 1467, 1469, 2676, 2992, 3015, 3057, 3107, 3166, 3203, 3208, 3232, 3234, 3310, 3311, 3334, 3432, 3436, 3437, 3809, 3986, 4057, 4098, 4102, 4103, 4153, 4540, 5900, 6660, 6661, 6662, 6663, 6664, 6665, 6666, 6667, 6897, 27374, 60291, 61289, 61300, 62479, 63597, 65535,

3.4 Distribution of Attacks by Volume

Rank	Signature	Alerts	Sources	Dest	
1	Incomplete Packet Fragments Discarded	24856	96	361	
2	SMB Name Wildcard	18706	223	14117	
3	SYN-FIN scan!	8621	3	8568	
4	High port 65535 tcp - possible Red Worm – traffic	6264	97	130	
5	UMBC NIDS IRC Alert IRC user /kill detected, possible trojan.	3609	47	54	

6	UMBC NIDS IRC Alert XDCC client detected attempting to IRC	3134	4	2	
7	EXPLOIT x86 NOOP	2544	406	111	
8	SUNRPC highport access!	2252	21	19	
9	High port 65535 udp - possible Red Worm - traffic	2089	98	80	
10	Possible trojan server activity	1304	56	289	
11	NMAP TCP ping!	1031	166	72	
12	Null scan!	725	51	47	
13	UMBC NIDS External MiMail alert	559	52	1	
14	TCP SMTP Source Port traffic	537	3	2	
15	EXPLOIT x86 stealth noop	321	7	5	
16	connect to 515 from outside	283	2	197	
17	UMBC NIDS IRC Alert Possible sdbot floodnet detected attempting to IRC	210	7	3	
18	FTP passwd attempt	120	82	2	
19	SMB C access	72	31	3	
20	FTP DoS ftpd globbing	62	8	2	
21	Tiny Fragments - Possible Hostile Activity	49	5	5	
22	TFTP - Internal TCP connection to external tftp server	41	2	2	
23	EXPLOIT x86 setuid 0	39	33	27	
24	EXPLOIT x86 setgid 0	38	30	30	
25	RFB - Possible WinVNC - 010708-1	35	15	16	
26	Probable NMAP fingerprint attempt	28	4	4	
27	EXPLOIT NTPDX buffer overflow	19	12	9	
28	IRC evil - running XDCC	19	1	3	

29	External RPC call	15	2	1	
30	Attempted Sun RPC high port access	15	8	10	
31	TFTP - Internal UDP connection to external tftp server	12	7	9	
32	UMBC NIDS IRC Alert Possible Incoming XDCC Send Request Detected.	7	4	4	
33	DDOS mstream client to handler	6	2	2	
34	NETBIOS NT NULL session	4	3	3	
35	External FTP to HelpDesk 130.85.70.49	3	3	1	
36	UMBC NIDS IRC Alert K\:line'd user detected, possible trojan.	3	1	1	
38	External FTP to HelpDesk 130.85.53.29	3	3	1	
39	UMBC NIDS IRC Alert User joining Warez channel detected. Possible XDCC bot	2	2	2	
40	UMBC NIDS IRC Alert User joining XDCC channel detected. Possible XDCC bot	2	2	2	
41	Happy 99 Virus	2	1	1	
42	TFTP - External TCP connection to internal tftp server	2	2	2	
43	Traffic from port 53 to port 123	2	2	2	
44	TFTP - External UDP connection to internal tftp server	1	1	1	
45	PHF attempt	1	1	1	
46	DDOS shaft client to handler	1	1	1	

Legend of color codes:

Needs immediate attention	**Needs attention**	**Reconessaince, scan**	**Mainly False Positives**

3.4.2 Most Frequent Alerts

Alert 1: Incomplete Packet Fragments Discarded

Snort SID: this alert is created by the Snort spp_defrag preprocessor and not by a rule.
Total Number of alerts: 24,856; **Sources**: 96; **Destinations** = 361.

Some traffic has both destination and source ports equal to 0. Some traffic has no port or destination port and I assume that this is ICMP traffic.

This event is triggered due fragmented IP packets were detected but not all the fragments of the packets arrived. What is clearly out of spec is the fact that some packets have both source and destination ports equal to 0. Port 0 is a reserved port and it should never be used. These packets were sent in bursts that lasted on average a couple of minutes.

This kind of highly fragmented traffic can occur if:

1. A bad network card or faulty router corrupt packets or if there are connectivity problems on this network.
2. A computer can perform a reconnaissance activity trying to bypass an IDS
3. A Denial of Service is in progress.

Usually, type 2 or 3 of alerts come together with other malicious or non-malicious activities: scans, attacks, games. In our case, a lot of internal hosts did not show this behavior. Some external hosts have showed this typical behavior:

Four different signatures are present for *80.130.216.16* (hostname p5082D810.dip.t-dialin.net) as a source

- 2 instances of *NMAP TCP ping!*
- 23 instances of *Probable NMAP fingerprint attempt*
- 339 instances of *Null scan!*
- 384 instances of *Incomplete Packet Fragments Discarded*

Three different signatures are present for *68.122.75.188* as a source

- 1 instances of *Probable NMAP fingerprint attempt*
- 4 instances of *Incomplete Packet Fragments Discarded*
- 10 instances of *Null scan!*

IP / Hostname	WHOIS details	Abuse | Security | Admin Coordinator
80.130.216.16 p5082D810.dip.t-dialin.net	**Name**: Deutsche Telekom AG, Internet service provider **OrgID**: DTAG **Net**: 80.128.0.0 - 80.146.159.255 **Address**: D-90492 Nuernberg **Country**: Germany	Security Team Deutsche Telekom AG Germany phone: +49 180 5334332 fax-no: +49 180 5334252 e-mail: abuse@t-ipnet.de

Hosts that should be checked – possible breach of Acceptable Computer Usage Policy:

Some computers within the campus tried a flood attack sending "Incomplete Packet Fragments", at a speed of a couple of dozen packets/second against *202.157.188.57*, which resolves to *"terorist.pakistanarmy.info".* The list of computers that tried to flood this site follows:

Source IP	Total # of packets to any	Other attacks from this source
130.85.21.37;	4000	NO
130.85.21.116	2270	NO
130.85.21.116,	3468	NO
130.85.21.68,	3486	NO
130.85.21.69,	3517	NO
130.85.21.79,	3381	NO
130.85.21.92.	2791	NO

As we can see from this table, these hosts did no trigger any other alert beside this "Incomplete Packet Fragments". This probably means that they are not compromised.

Snort has two rules that trigger if destination port is 0: "BAD-TRAFFIC tcp port 0 traffic" (SID=524), and "BAD-TRAFFIC udp port 0 traffic" (SID=524). They did not trigger because were not hit by these packets - preprocessor dropped the packets as soon it found they are made ofincomplete fragments.

Corellations:

1. An IDS GIAC paper (published at http://www.lurhq.com/idsindepth.html) presents this kind of alert.
2. Cory Steers GCIA paper briefly describes this alert http://www.giac.org/practical/GCIA/Cory_Steers_gcia.doc
3. In this e-mail Marty Roech recommends using the newer frag2 preprocessor instead of the old spp_defrag (http://www.mcabee.org/lists/snort-users/Nov-01/msg00820.html)
4. A concise description of port 0 usage can be found at: PORT 0 - http://compnetworking.about.com/library/ports/blports_0.htm

Recommendations:

5. Filter out traffic from or to port 0.
6. Log internal hosts that try to connect to port 0. They could be compromised hosts doing reconessaince or simply have defective network cards.
7. If packets of this type are coming from a certain subnetwork, check its routers for bad configurations or defective parts.
8. Turn off the spp_defrag preporcessor and use frag2 instead to reduce the number of false positives.

Alert 2: SMB Name Wildcard

Snort SID: N/A, IDS 177 ArachNIDS

Total Alerts: 18,706, **Sources**: 223, **Destinations**: 14,117

The rule that triggered this alert is:

```
UDP $EXTERNAL any -> $INTERNAL 137 (msg: "IDS177/netbios-name-query"; content:
"CKAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA|0000|";)
```

This alert is triggered by a Netbios SMB request (Broadcast Name Service Request). A SMB Netbios request can occur as a part of a normal Windows traffic, but it can also be used for reconessaince. For example, the Windows command "*nbtstat –A IP*" will trigger this alert. Information that is made public includes: the NetBIOS name of computer, Windows NT worgroup name, login names of users that are currently logged in.

I do not agree with the persons who recommend disabling this signature if both the source and destination are on the inside network. They rush to turn off a rule that could prompt them possible infections with worms like the netbios worm. A lot of these worms (like *network.vbs*) scan for for Windows computers that are possible targets of infection.

A short presentation of this worm can be read in the article "Port 137 scan" at:
http://www.sans.org/resources/idfaq/port_137.php

A big number of alerts generated by a computer placed on the inside network that sends out this kind of traffic both to outside and inside networks could be infected with this virus or a similar other virus.

This particular alert could have been triggered by such a worm.

Source	# Alerts (sig)	# Alerts (total)	# Dsts (sig)	# Dsts (total)
130.85.80.51 (ss-80-51.pooled.umbc.edu)	13806	13806	13806	13806

This IP should be checked for worms or virusi as soon as possible.

Other Sources that triggered a lot of the SMB Wildcard alarms are presented in the table below.

Number of alerts	Orig Host	Originating IP	Destination IP
1231	-	130.85.11.61	169.254.0.0
187	dc2.ad.umbc.edu	130.85.11.7	169.254.0.0
41	ark.umbc.edu	130.85.62.2	137.69.102.199
41	dc1.ad.umbc.edu	130.85.11.6	137.69.102.199
40	dc2test.adtest.UMBC.EDU	130.85.11.3	137.69.102.199
39	collab.umbc.edu	130.85.29.23	137.69.102.199
39	quarantine.UMBC.EDU	130.85.11.4	137.69.102.199
38	fungi.ad.umbc.edu	130.85.30.86	137.69.102.199

38	fp1.ad.umbc.edu	130.85.11.2	137.69.102.199
37	ndms.umbc.edu	130.85.5.44	137.69.102.199
37	kryten.ucs.umbc.edu	130.85.70.177	137.69.102.199
37	asp1.umbc.edu	130.85.5.92	137.69.102.199
36	ndmssupport.umbc.edu	130.85.5.64	137.69.102.199

The first two lines have 169.254.0.0/16 as destination. 169.254.0.0/16 is the "link local" block. This block is allocated for communication between hosts on a single link. Hosts obtain these addresses by auto-configuration, such as when a DHCP server may not be found[1].

What is interesting is that all these source IPs seems to be servers. Some of them can be Domain Controllers (they have dc in their hostnames) or Active Directory computers (they have ad in their names).

The rest of the lines in this table have the same IP destination 137.69.102.199:

IP / Host	Whois Details	Abuse | Security | Admin Contact
137.69.102.199	**Name**: Legato Systems, Inc **Netblock**: 137.69.0.0/16 **NetID**: LEGATO-NET **Address**: 2350 EL CAMINO REAL MOUNTAIN VIEW, CA 94040 **Country**: USA	**E-mail**: nadmin@legato.com **Phone**: +1-650-210-7228

This IP is registered with Legato, a company specialized in backup and disaster recovery solutions. It is very likely that this IP was an off site location for UMBC backups.

.**Corellations:**

1. SANS IDS FAQ – Port 137 http://www.sans.org/resources/idfaq/port_137.php
2. E-mail message by Jim Forster about SNM Name Wildcard - http://archives.neohapsis.com/archives/snort/2000-01/0222.html
3. In his e-mail message Max Vision (http://archives.neohapsis.com/archives/snort/2000-01/0220.html) recommends using SMB rules that triggers when $C or $ADMIN shares are accessed from external networks. Again, this will lower the number of false positives, but it will make the IDS blind to attacks made by internal machines trying to use these shares for malicious purposes.

Recommendations:

1. Block port 137 (NetBIOS name service) at the border routers or firewalls, both incoming and outgoing. Windows NBT was created for LAN usage and not for WAN. Also, filter out all the following ports: 135 (Windows RPC), 138 (Netbios datagram service), 139 (NetBIOSFile and Print Sharing), 445 (SMB over TCP).

[1] RFC 3330.

2. If external computers need access to internal Windows SMB Services allow this only via VPN. Check Point SecuRemote Client, for example, can login to a domain controller via VPN.
3. Check computers for blank passwords and null shares.
4. Follow best practices regarding passwords, patching and antivirus.

Alert 4: High port 65535 tcp - possible Red Worm – traffic

Snort SID: N/A, this is a custom rule.

Total number of alerts: 6,264, Sources = 97, Destinations: 130.

This is a custom alert and I think that the rule triggers based on the usage of the port 65553 tcp. The Red Worm (aka Adore) is a Linux worm that opens a shell on port 65553. For more information on this worm, you can read http://www.f-secure.com/v-descs/adore.shtml and "Adore Worm" - http://www.sans.org/y2k/adore.htm.

Port 65553 is a legitimate port in the ephemeral range and traffic originating from this port is not always malicious. For example, the following alert is positively identified as a false positive.

```
11/09-23:22:43.438986 [**] High port 65535 tcp - possible Red Worm - traffic [**]
130.85.25.10:65535 -> 204.127.202.26:25
```

Host 130.85.25.10 (mxlout.umbc.edu – a mail server) connects to the mail server 204.127.202.26, which is a COMCAST mail gateway (Checked with Andrew Daviel's SMTP IP Adress Indentifier http://andrew.triumf.ca/cgi-bin/smtpview?204.127.202.26 - *"220 sccrmxc14.comcast.net - Maillennium ESMTP/MULTIBOX sccrmxc14 #113, 221 sccrmxc14.comcast.ne).*

It is clear that we are dealing here with a false positive.

Another example of false positive Red Worm is this:

```
11/05-09:48:24.229776 [**] High port 65535 tcp - possible Red Worm - traffic [**]
130.85.5.20:80 -> 198.26.120.13:65535
```

It is possible that we are dealing with a web server. A quick search for alerts generated by this host (130.85.5.20 - centrelearn.umbc.edu) revealed that the only other alert type triggered by 130.85.5.20 is:

```
11/05-12:34:33.098580  [**] Possible trojan server activity [**] 129.176.151.123:27374 ->
130.85.5.20:80
```

It is very clear that this type of alert is is triggered by the usage of the port 27374, which usually is used by one of these trojans: 27374 Bad Blood, SubSeven, SubSeven 2.1 Gold, Subseven 2.1.4 DefCon 8 (http://www.pcconcrete.ro/phtml/p09288.html, in Romanian Language)

And yet another example of false positive, in this case it could be a legitimate POP3 conection:

```
11/09-17:45:10.942486 [**] High port 65535 tcp - possible Red Worm -
traffic [**] 130.85.60.17:110 -> 68.55.62.110:65535
```

These two examples show again the pitfalls of rules that are too generic, and therefore prone to create a lot of false positives.

Corellations: Les Gordon presents a similar alert in his GIAC paper http://www.giac.org/practical/GCIA/Les_Gordon_GCIA.doc.

Recommendations:

1. Turn off this rule and use nmap scans to find internal hosts listening on port 65553.
2. Use best practices regarding patching procedures.
3. If you choose not to turn off the rule, then modify it to check only the incoming traffic.

Alert 7: EXPLOIT x86 NOOP

Alerts: 2,544; **Sources**: 406, **Destinations**: 111.

SID: 648, 1394

There are two *EXPLOIT x86 NOOP* rules in the standard Snort rule set: 648, 1394.

NOP stands for “NO OPERATION”. NOOPs are used to “pad” the code.

According to Snort.org (http://www.snort.org/snort-db/sid.html?sid=648): *“The NOP allows an attacker to fill an address space with a large number of NOPs followed by his or her code of choice. This allows"sledding" into the attackers shellcode.”*

Unfortunatelly, this “atack” is prone to a lot of false positives – transfers of binary files over ftp or http connections trigger a lot of false positives. And I assume that this is exactly what a lot of students do – transfer big files over http or ftp.

This is a typical FTP file transfer conection using FTP Active:

```
11/07-18:56:04.662057 [**] EXPLOIT x86 NOOP [**] 134.91.11.145:5004 ->
130.85.6.63:20
11/07-18:56:05.993348 [**] EXPLOIT x86 NOOP [**] 134.91.11.145:5004 ->
130.85.6.63:20
```

Corellations: A message posted on the “incidents” e-mail list: Nick FitzGerald - http://www.securityfocus.com/archive/75/218557/2001-10-05/2001-10-11/0: “Are your users normally allowed to transfer Windows program files around via HTTP??

If so, the above is nothing to worry about”.

Recommendations:

1. Check the paylods of the packets that triggered these alerts.

2. Customize this rule and the shell ports variable on snort.conf to eliminate some possibile false positives. For example, a shell code transferred over DNS could be a better indicative of an attack than a shell code transferred over HTTP.

Alert 8: SUNRPC highport access!

Total number of alerts: 2,089; **Sources**: 98; **Destinations**: 80

SID: This is a custom rule that triggers if a SUN RPC port is used, in our case 32771.

RCP portmapper is very well known for its wealth of vulnerabilities. The UNIX portmapper listens on port 111 and "map" a service to a port in the range 32771-34000.

Because this rule is very generic it will generate a lot of false positives any time a legitimate conection will use the port 32771. For example, I checked to see if these connections were legitimate or not. I was surprised to find that the rule was triggered by a benign surf of the google.com - http://216.239.37.99 is Google.com

Corellations: In his practical posted on http://www.zeltser.com/sans/idic-practical/ Lenny Zeltser writes that this alert is triggered by attempted connections to port 32771 (ruserd).

Recommendations:

1. Block port 111 (UNIX RPC portmapper at the border router/firewall).
2. Make sure that UNIX boxes that do not really need RPC have it turned off or if not, use TCP-wrapped rpcbind for Solaris (TCP wrapper style access control to RPC services): http://www.eits.uga.edu/wsg/sun/security/wrapped_rpcbind.html
3. Customize this rule to reduce the number of false positives.

Alerts that may show signs of compromised hosts:

Alert 6: UMBC NIDS IRC Alert XDCC client detected attempting to IRC

Total number of alerts: 3,134, **Sources**: 4, **Destinations**: 2

Snort SID: N/A, this is a customized detection rule.

XDCC is a method of file-sharing that allows users to download files (usually *warez*), on demand, from a user's computer, similar to a P2P .

These four machines should be imediately checked:

130.85.15.198, 130.85.80.16, 130.85.81.18, 130.85.66.53,

In a similar category (*UMBC NIDS IRC Alert User joining Warez channel detected. Possible XDCC bot*, and *UMBC NIDS IRC Alert User joining Warez channel detected. Possible XDCC bot*), are these two hosts 130.85.42.9, 130.85.97.153. They should be immediatelly checked.

Corellations:

1. A very good article about this IRC Client can be read at: Instructions on Cleaning IRC bot & backdoor: XDCC http://security.duke.edu/cleaning/xdcc.html
2. Also, Tonikgin describes in detail the XDCX in this article – *An .EDU Admin's Nightmare*, http://www.cs.rochester.edu/~bukys/host/tonikgin/EduHacking.html
3. The XDCC search engine can be found at: http://www.xdccsearch.com/flash/index.html

Recommendations:

1. Scan suspicious machines for listening ports – XDCC bot starts an FTP server
2. Make sure that the desktop virus scanners are installed and their signatures updated periodically, according to your policy.
3. Close ports 137 (used by nbtstat –A) and 139 (file sharing) for incoming traffic.
4. Check egress traffic for asymetric (high out, low in) volume of traffic (signals a computer that is used to store and share warez).

Hosts that may be compromised and need immediate attention:

130.85.190.97

Six different signatures are present for *130.85.190.97* as a destination

- 1 instances of *SYN-FIN scan!*
- 1 instances of *NETBIOS NT NULL session*
- 2 instances of *connect to 515 from outside*
- 5 instances of *Possible trojan server activity*
- 30 instances of *SMB C access*
- 223 instances of *EXPLOIT x86 NOOP*

A lot of tipical Microsoft specific alerts have been triggered by traffic having this host as destination. This initally made me think that this could be Domain Controller or a file/print server. After noting that the whole traffic to this host was originating from outside hosts it became clear to me that this host could be compromised.

More than that, this host was used as a platform for reconessaince or attacks directed to other hosts:

Two different signatures are present for *130.85.190.97* as a source

- 4 instances of *Possible trojan server activity*
- 25 instances of *SMB Name Wildcard*

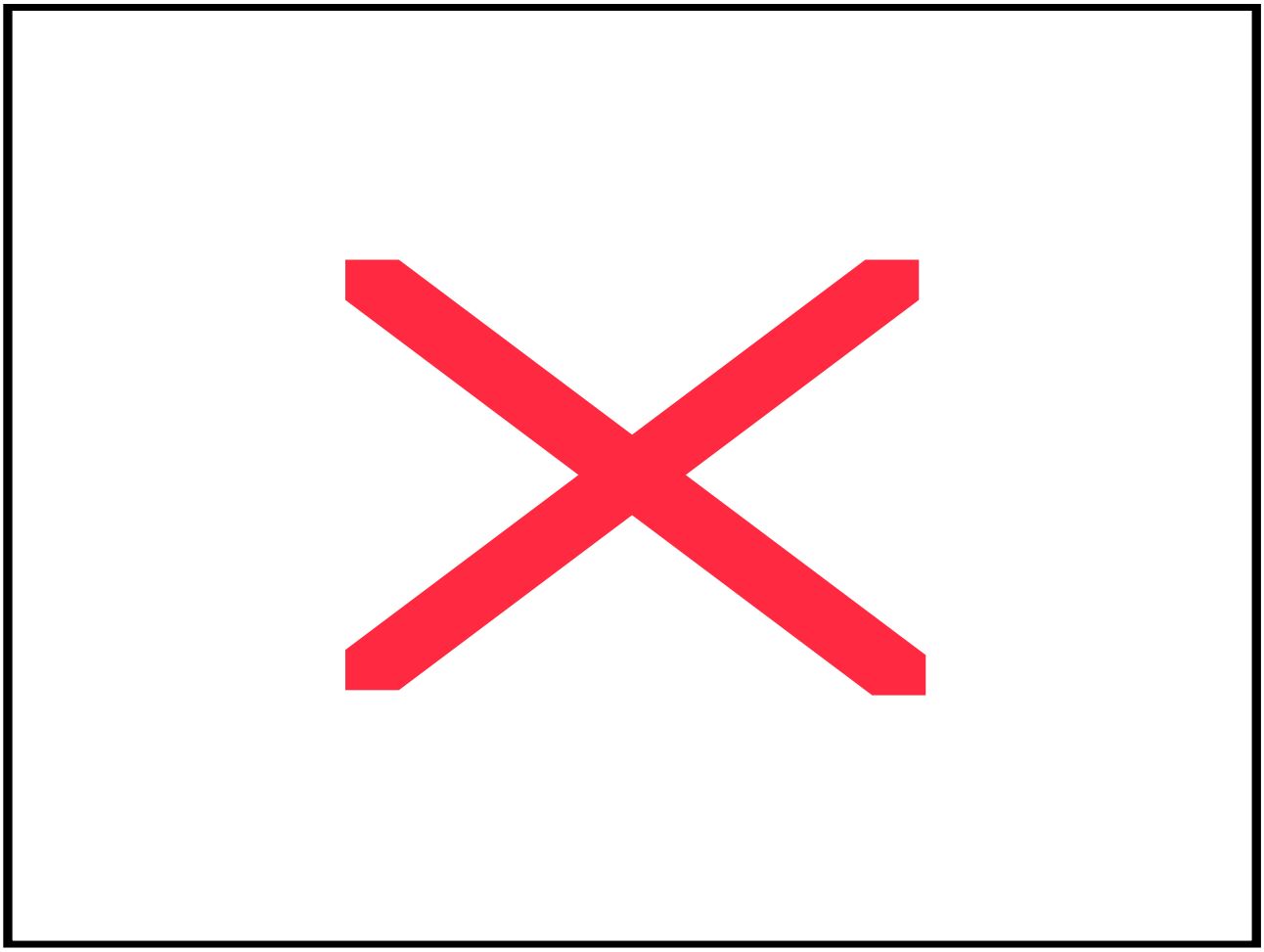
130.85.111.34

This host connects to an outside TFTP Server: 210.64.76.251, which is a host located in Taiwan - sw64-76-251.adsl.seed.net.tw.

```
$ whois -h whois.twnic.net 210.64.76.251
```

IP/Host	WHOIS Details	Abuse | Security | Admin Contact
210.64.76.251 sw64-76-251.adsl.seed.net.tw	**Name:** Seednet-KaohsiungDP-S **Netblock:** 210.64.76.0 - 210.64.76.255 **NetID**: SEEDNET-NET **Address**: Seednet-KaohsiungDP-S No. 220, Gangchi Rd., Neihu Chiu, Taipei, Taiwan 114, R.O.C.	**Technical & Admin contact**: root123@seed.net.tw tel: +886-2-0800-073330

Also, this computer receives a lot of VNC Connections from hosts on 151.196.x.x, and 141.157.66.49.



Link graph of alerts triggered by traffic to or from host 130.85.111.34

TOP 10 SOURCE IPs					
	Total Alerts	**Source IP (Hostname)**	**# Sign. triggered**	**Destinations involved**	**Note**

1	13806	130.85.80.51 (ss-80-51.pooled.umbc.edu)	1	13806 destination IPs	Wide scale *SMB Wildcard*
2	8500	64.243.84.43 (-)	1	8500 destination IPs	Wide Scale *Fin scan* port 554 to port 554.
3	4428	130.85.21.67 (c00149.umbc.edu)	1	14 destination IPs	900 *Incomplete Packets Discarded* over 4 minutes, originated from 255.255.21.116 255.255.21.37 255.255.21.68 255.255.21.67 255.255.21.69
4	4000	130.85.21.37 (c00111.umbc.edu)	1	12 destination IPs	See above # 3
5	3517	130.85.21.69 (c00151.umbc.edu)	1	15 destination IPs	See above #3
6	3486	130.85.21.68 (c00150.umbc.edu)	1	14 destination IPs	See above #3
7	3381	130.85.21.79 (c00166.umbc.edu)	1	9 destination IPs	See above #3
8	3071	64.157.246.22 (-)	2	130.85.80.16 (ss-80-16.pooled.umbc.edu), 130.85.15.198 (tfc-pplant.umbc.edu)	1 instances of *UMBC NIDS IRC Alert User joining XDCC channel detected. Possible XDCC bot* 3070 instances of *UMBC NIDS IRC Alert IRC user /kill detected, possible trojan.*
9	2791	130.85.21.92 (c00184.umbc.edu)	1	(11 destination IPs)	See above #3 – Incomplete packets Discarded
10	2525	130.85.15.198 (tfc-pplant.umbc.edu)	1	64.157.246.22 (no hostname)	2525 instances of *UMBC NIDS IRC Alert XDCC client detected attempting to IRC*

```
$ whois -h whois.arin.net 64.157.246.22
```

IP / Host	Whois Details	Abuse | Security | Admin Contact
64.157.246.22	**Name**: Level 3 Communications, Inc. **Netblock**: 64.152.0.0 - 64.159.255.255 **NetID**: LC-ORG-ARIN **Address**: 1025 Eldorado Blvd., Broomfield, CO 80021 **Country**: US	**E-mail**: abuse@level3.com **Phone**: +1-877-453-8353

TOP 10 DESTINATION IPs

	Total Alerts	Destination IP (Hostname)	# Sign triggered	Originating sources	Notes
1	6867	195.219.153.7 -	1	255.255.21.37 255.255.21.67 255.255.21.68 255.255.21.69 255.255.21.79 255.255.21.92 255.255.21.116	6867 instances of *Incomplete Packet Fragments Discarded* originating from these seven IPs
2	3443	24.227.67.205 rrcs-se-24-227-67-205.biz.rr.com	1	(6 source IPs) Same originating IPs as above #1	3443 instances of *Incomplete Packet Fragments Discarded*
3	2980	64.157.246.22 -	1	130.85.80.16 (ss-80-16.pooled.umbc.edu) 130.85.15.198 (tfc-pplant.umbc.edu)	2980 instances of *UMBC NIDS IRC Alert XDCC client detected attempting to IRC*
4	2607	130.85.15.198 tfc-pplant.umbc.edu	3	64.157.246.22, 64.243.84.43	1 instances of *UMBC NIDS IRC Alert User joining XDCC channel detected. Possible XDCC bot* 1 instances of *SYN-FIN scan!* 2605 instances of *UMBC NIDS IRC Alert IRC user /kill detected, possible trojan*
5	2500	202.157.188.57 terorist.pakistanarmy.info	1	(7 source IPs) See above #1	2500 instances of Incomplete Packet Fragments Discarded
6	2273	24.188.139.201 ool-18bc8bc9.dyn.optonline.net	1	(5 source IPs) See above #1	2273 instances of Incomplete Packet Fragments Discarded
7	2196	65.147.28.178 0-1pool28-178.nas53.stockton1.ca.us.da.qwest.net.	1	(7 source IPs) See above #1	2196 instances of Incomplete Packet Fragments Discarded
8	1926	130.85.70.38 dobbs.ucs.umbc.edu	1	64.12.28.189.	1926 instances of SUNRPC highport access! Created by legitimate AOL traffic originating from port 5190
9	1694	209.152.170.161 IP-209-152-170-161.dinix.com	1	(7 source IPs) See above #1	1694 instances of Incomplete Packet Fragments Discarded.
10	1595	130.85.152.19 lib037pc50.ucslab.umbc.edu	2	62.101.126.219, 66.28.249.232	1 instance of EXPLOIT x86 setuid 0 1594 instances of High port 65535 tcp - possible Red Worm - traffic

3.5 Scanning

Alert 14: TCP SMTP Source Port traffic

Total number of alerts: 537, **Sources**: 3, **Destinations**: 2

There could be only one reason to initiate a TCP connection using the port 25 as source port – to pass a packet filtering firewall.

A close look at these alerts will reveal that actually both source and destination ports are 25. This is a recon activity.

Recommendation: Install a statefull firewall

3.6 Scan Logs

Blaster infected hosts do a lot of scans on port 135. Based on this, I determined that the following local hosts might be infected with Blaster worm:

130.85.70.129	ecs128pc01.umbc.edu
130.85.80.243	pplant-80-243.pooled.umbc.edu
130.85.84.194	engr-84-194.pooled.umbc.edu
130.85.111.72	cuereims.umbc.edu
130.85.112.179	
130.85.153.174	libstkpc28.libpub.umbc.edu
130.85.163.107	physics105pc-01.umbc.edu

Recommendation: Scan this computers for listening port 707. A computer that listens on port 707 is very likely infected with Blaster.

Usually, portscan detectors perceive the DNS traffic as a portscan. After a quick search I found that the following three hosts performed portscaning exclusively to port 53:

130.85.1.2, 130.85.1.5, 130.85.1.200. A DNS lookup on these hosts will tell us that 130.85.1.5 is a DNS server for UMBC:

```
130.85.1.200
130.85.1.2      umbc2.UMBC.EDU
130.85.1.5      UMBC5.UMBC.EDU

85.130.in-addr.arpa     nameserver = UMBC5.UMBC.EDU.
UMBC5.UMBC.EDU  internet address = 130.85.1.5
```

It is very likely that 130.85.1.2 is a DNS Server too. What is not OK with these three hosts is that the targets of “port 53 UDP portscans” are not all DNS Servers.

Recommendations: Have 130.85.1.200 checked.

The hosts on this list, due to their infection with Blaster or possible DNS false positives, trigger 85% of the scans:

130.85.84.194, 130.85.111.72, 130.85.163.107, 130.85.70.129, 130.85.112.179, 130.85.80.243, 130.85.153.174, 130.85.1.200, 130.85.1.2, 130.85.1.5

After eliminating these hosts using the custom Perl script “take_out.pl”, the size of the scans file dropped from 11,816,781 lines to 1,739,122 lines.

Scan Distribution by Number			
Type	**Flags**	**Count**	**Percentage**
SYN	******S*	963989	55.42%
UPD		743553	42.75%
SYN with OPT	12****S*	17186	0.98%
SYNFIN		8615	0.49%
FIN	*******F	3278	0.18%
INVALIDACK		649	0.03%
NULL	********	549	0.03%
UNKNOWN		396	0.02%
VECNA		200	0.01%
FULLXMAS	12UAPRSF	6	0.00%
XMAS	**UAPRSF	2	0.00%

Top 10 local scans– same local source to any destination					
Host IP	**Count**	**Percentage**	**NS Lookup**	**Type of Scans / Traffic**	**Notes**
130.85.163.76	200644	11.53%	phys008pc-01.umbc.edu	WinMX P2P	Port 6257 WinMX File Sharing P2P. Also triggered a lot of “Red Worm – High Port” alerts
130.85.1.3	93816	5.39%	umbc3.umbc.edu	DNS Server -	
130.85.69.137	91397	5.25%	lib-69-137.pooled.umbc.edu	Games	22321<-> 22321 UDP 7674 <-> 7674 UPD http://lists.insecure.org/lists/security-basics/2003/Feb/0293.html
130.85.84.143	66179	3.80%	engr-84-143.pooled.umbc.edu	e-Donkey	e-Mule P2P
130.85.100.230	59948	3.44%	imap.cs.UMBC.EDU smtp.cs.UMBC.EDU mailhost.cs.UMBC.EDU mailserver-ng.cs.UMBC.EDU pop.cs.UMBC.EDU	MAIL (SMTP, IMAP, POP3) for the CS Department.	
130.85.53.225	55197	3.17%	ecs104pc25.ucslab.umbc.edu	Gnutella	Port 6346 UPD Gnutella
130.85.70.207	39272	2.25%	ecs020pc09.umbc.edu	Games	Games – GameSpy, Alllied

					Assault http://www.autokick.com/content/adminguide.pdf
130.85.66.23	27197	1.56%	-	Games	22321<-> 22321 UDP 7674 <-> 7674 UPD
130.85.112.159	26035	1.49%	-	Games	12404 UPD – Game Server (GameSPy)
130.85.1.4	22329	1.28%	UMBC4.UMBC.EDU	DNS Server -	

Top 10 external scans – same external source to any destination

Host	Count	Percentage	Hostname	Note
80.200.65.115	29671	1.70%	115.65-200-80.adsl.skynet.be	Wide scale SYN scan for port 21 TCP
66.7.239.116	25115	1.44%	66-7-239-116.cust.telepacific.net	Wide Scale SYN scan for port 80 TCP
195.197.107.194	24021	1.38%		Wide Scale SYN scan for port 4000 TCP - ICQ
12.39.196.46	21096	1.21%		Wide Scale SYN scan for port 4000 TCP - ICQ
210.91.83.34	20751	1.19%		Wide Scale SYN scan for port 4000 TCP - ICQ
80.15.56.23	20052	1.15%	ANantes-106-1-11-23.w80-15.abo.wanadoo.fr	Wide scale SYN scan for FTP – port 21 TCP
148.244.114.28	19893	1.14%	host-148-244-114-28.block.alestra.net.mx	Wide scan SYN scan for port 554 TCP Real Time Streaming Protocol
217.227.183.224	18717	1.07%	pD9E3B7E0.dip.t-dialin.net	Wide scale SYN scan for port 80 TCP
158.121.109.201	17940	1.03%	dellsrvr.geog.umb.edu	Wide scale SYN scan for port 80 TCP
12.148.246.105	16257	0.93%		Wide scale SYN scan for port 80 TCP

IP 148.244.114.28 (host-148-244-114-28.block.alestra.net.mx) scanned a fairly new exploit – "RealNetworks media server RTSP protocol parser buffer overflow" (http://www.kb.cert.org/vuls/id/934932)

Since this is new exploit it is likely that we are dealing with a more dangerous attacker.

```
$ whois -h whois.lacnic.net 148.244.114.28
```

IP / Host	Whois Details	Abuse | Security | Admin Contact
148.244.114.28 host-148-244-114-28.block.alestra.net.mx	**Name**: Digital Comunitation **Netblock**: 148.244.114/24 **NetID**: MX-DICO-LACNIC **Address**: Av. Universidad S/N col. Bosques del Prado Aguascalientes, Aguascalientes 20127 **Country**: MX	Rodolfo Rubio **E-mail**: admin2@ALESTRA.NET.MX **Address**: Digital Comunitation Av. Universidad S/N col. Bosques del Prado Aguascalientes, Aguascalientes 20127

		MX **Phone**: 4-914-1291

Top 10 Inbound Scan Destination Ports			
Port	**Count**	**Percentage**	**Service**
80	308002	17.71%	http
21	82140	4.72%	ftp
4000	65868	3.78%	ICQ Control
554	57329	3.29%	Real Time Stream Control
3410	48433	2.78%	
1243	47006	2.70%	SubSeven TCP Control Port
20168	43025	2.47%	
135	35109	2.01%	
4899	27569	1.58%	Radmin (Remote Access Admin)
17300	20581	1.18%	Kuang2 Virus
6370	14268	0.82%	

HTTP and FTP are the most scanned services. Also, ICP is well known for a beavy of vulnerabilities.

Recommendations:

1. If possible, restrict http and ftp servers to computers that are adiministred by professional staff, not students.
2. Start an awarness program about the Dangers of ICQ and P2P software
3. Create and use scripts that scan for internal vulnerable machines (for example, computers that are listening on port 17300 are infected with the Kuanq2 trojan/virus)

Top 10 outbound destination ports:			
Port	**Count**	**Pecentage of Total**	**Note**
6257	194777	11.19%	WinMX P2P
53	116726	6.71%	DNS
22321	76022	4.37%	Games
7674	68906	3.96%	Games
25	60074	3.45%	SMTP
4673	32541	1.87%	e-Mule P2P
6346	30375	1.74%	Possible e-Mule P2P
4672	20516	1.17%	e-Mule P2P
4662	19582	1.12%	e-Mule P2P

7000	14559	0.83%	ICQ

3.6 Out Of Spec Alerts

All OOS files for the period of time November 5 – November 25 are actually the same file (http://www.incidents.org/logs/oos/oos_report_031027) . The first alert was triggered on 10/27-00:06:01, and the last on 10/29-14:30:59. As of today, November 25, 2003, incidents.org/logs is listing the same file. Under the pressure of the deadline, although the dates of these OOS alerts do no overlap the dates of the Scans and Alerts, just for didactic puposes, I decided to use this file

Out Of Spec Alert logs are triggered by anomalus TCP flag combinations.

TCP anomalous flag combinations can have

1. Malign causes - IDS evasion attacks, OS fingerprinting using anamolous packets, Exploits, Denial of Service
2. Benign causes - Corrupted TCP packets, ECN (Explicit Congestion Notification.

Using the script “process_oo.pl” I found the TOP 10 IP sources and destinations that generated OOS:

Top 10 Source IPs OOS			
IP	**Count**	**Reverse Name Lookup**	**Note**
195.111.1.93	2446	moon.ilab.sztaki.hu	All to port 80 12*S
213.54.173.255	277	p213.54.173.255.tisdip.tiscali.de	All to port 4662 12*S
158.196.149.61	264	monica.vsb.cz	All to port 25, 12*S, of 130.85.111.52, ecs111dell01.engr.umbc.edu
195.101.94.101	226	x1crawler2-1-0.x-echo.com	All to port 80 of different web servers
66.225.198.20	224	unknown.servercentral.net	
195.101.94.208	204	x1crawler1-1-0.x-echo.com	All to port 80 of different web servers
63.71.152.2	140	wall.turbinegames.com	All to port 113 of 130.85.100.230 (an e-mail server). Please read below the Recommendations paragraph.
61.135.129.99	140	-	All to 130.85.24.34 port 80 [www.umbc.edu] , 12*S.
216.95.201.13	131	smtp3.dbhits.com	130.85.12.6 [mxin.umbc.edu] 130.85.100.230[mail.cs.umbc.edu]

			130.85.75.3 [no reverse name]
195.101.94.209	126	x1crawler3-1-0.x-echo.com	All to port 80 of different web servers

A complete list of web spiders, crawlers, and robots can be found at

"*List of User-Agents/Browsers, Web Spiders, Robots*" – [http://www.netsys.com/cgi-bin/display_article.cgi?1193]

Top 10 Source IPs OOS			
IP	**Count**	**Hostname**	**Notes**
130.85.12.6	2651	mxin.umbc.edu	All port 25 12*S
130.85.24.34	2528	www.umbc.edu	All Port 80 12*S
130.85.24.44	1203	userpages.umbc.edu	All Port 80 12*S
130.85.84.143	391	engr-84-143.pooled.umbc.edu	All Port 4662 12*S eDonkey [client-client]
130.85.112.159	349	-	All Port 4662 12*S eDonkey [client-client]
130.85.100.165	338	web1.cs.umbc.edu	Majority to port 80, a couple to port 25,
130.85.111.52	279	ecs111dell01.engr.umbc.edu	All to port 25, 12*S This computer runs an SMTP server
130.85.6.7	260	umbc7.umbc.edu	All port 80, 12*S
130.85.100.230	193	mailserver-ng.cs.UMBC.EDU. pop.cs.UMBC.EDU. imap.cs.UMBC.EDU. imap.cs.UMBC.EDU. smtp.cs.UMBC.EDU.	The bulk to port 113 (ident), some to port 25 **
130.85.60.14	121	www.gl.umbc.edu	All to port 80 12*S

EDonkey FAQ (at http://www.overnet.com/documentation/donkeyfaq.html) for EDoneky ports:

"It can run over any port. The defaults it uses are:

TCP port 4661 to connect to the server.

TCP port 4662 to connect to other clients.

UDP port 4665 to send messages to servers other then the one you are connected to."

All OOS alerts are generated by packets that have the TCP flag set to 12****S*. I found that there is no other flag used by processing the OOS file using the custom made Perl script real_oos.pl.

The vast majority of OOS alerts were triggered by legitimate traffic (web surfing, e-mail, games, and P2P)

ECN and it's impact on Intrusion Detection - http://www.sans.org/y2k/ecn.htm

Random Early Detection (RED) and ECN (Explicit Congestion Notification)

RFC 2481 - A Proposal to add Explicit Congestion Notification (ECN) to IP: http://www.faqs.org/rfcs/rfc2481.html

RFC 2884 - Performance Evaluation of Explicit Congestion Notification (ECN) in IP Networks – [http://www.faqs.org/rfcs/rfc2884.html]

Recommendations:

1. Protect the e-mail servers with firewalls that REJECT all connections to port 113. For more details about why REJECTING and not simply DROPPING them by the cleanup rule, please read: Robert Graham – "Installing a firewall causes slow connections to POP and SMTP services" (http://www.robertgraham.com/pubs/firewall-seen.html#7.5). "This is a protocol that runs on many machines that identifies the user of a TCP connection. In standard usage this reveals a LOT of information about a machine that hackers can exploit. However, it used by a lot of services by loggers, especially FTP, POP, IMAP, SMTP, and IRC servers. In general, if you have any clients accessing these services through a firewall, you will see incoming connection attempts on this port. Note that if you block this port, clients will perceive slow connections to e-mail servers on the other side of the firewall. Many firewalls support sending back a RST on the TCP connection as part of the blocking procedure, which will stop these slow connections.
2. Check the computer at 130.85.75.3. This PC runs an e-mail server on port 25 and it does not have any DNS entry. It could be an "illegal" e-mail server or even worse, a spammer.
3. Write BPF filters to pass traffic that triggers the bulk of False Positives OOS.

3.7 Appendix A – Methodology, File processing

All files where concateneted in a single file using *cat*, then sorted, and cleaned. Basic UNIX commands like *wc*, *grep*, *sort* were used.

Below you can find the Perl scripts that I wrote and used to process the log files.

The concatenated alert file was anlyzed using *snortsnarf*, *snortalog* and *snortstat*.

To check if 130.85.x.x is used:

```
$grep '\<255\.255\.*\.*\.\>' alert-1
```

To replace MY.NET.x.x with 130.85.x.x I wrote the following Perl script:

```
#!/usr/bin/perl
while (<>) {
      chomp;
```

```
        s/\bMY\.NET/255\.255/g;
        print; print "\n";
}
```

To find source ports:

```
source_ports.pl
#!/usr/bin/perl
while (<>) {
        chomp;
        @line = split;
        $dst = $line[-1];
        $src = $line[-3];
        ($src_h, $src_p) = split (/:/,$src);
        if (($src_h =~ /\b255\.255/) and ($src_p <= 1023) and ($src_p != 137) and ($src_p) ) {
                $new_dst = join (":", ($src_h, $src_p));
                push (@source, $new_dst);
        }
}
```

To eliminate “0” from the beginning of the IP is used this script

```
ZERO_OUT.pl
#!/usr/bin/perl
while (<>) {
        chomp;
        s/\b0//g;
        print; print "\n";
}
```

To eliminate the square bracketes inside the [**] [… } [**], I used the following script”

```
paranthesis.pl
#!/usr/bin/perl
while (<>) {
        chomp;
        @line = split(/\[\*\*\]/);
        $line[1] =~ s/\[//;
        $line[1] =~ s/\]//;
        $new_line = join ("\[\*\*\]", @line);
        print "$new_line\n";
}
```

To eliminate alert lines with errors (more than two [**] delimiters within the same line) I used the following script:

```
error.pl
#!/usr/bin/perl
while (<>) {
        chomp;
        @line = split(/\[\*\*\]/);
$leng = @line;
  if ($leng == 3) {
        $new_line = join ("\[\*\*\]", @line);
        print "$new_line\n";
  }
}

@source = uniq(@source);
foreach $item(@source) { print "$item\n"; }
sub uniq {
        ## this subroutine takes a @list and filter out duplicate items
```

```
        my (%seen) = ();
        my (@uniq) = ();
        foreach  $item (@_) {
                push (@uniq, $item) unless $seen{$item}++;
        } # foreach
return (@uniq);
} # sub
```

To “doctor” the alert to work with snortalog.pl, inserting fake Classifications, Priorities and Protocol, I used:

to_snortalog.pl

```
#!/usr/bin/perl
while (<>) {
        chomp;
        @line = split(/\[\*\*\]/);
        $end = join (" ", (' [Classification: Attempted Information Leak] [Priority: 0] {TCP}', $line[2]));
        $new_line = join ("\[\*\*\]", ($line[0], $line[1], $end));
        print "$new_line\n";
}
```

This script eliminates the scan line generated by the hosts in the array @out:

take_out.pl

```
#!/usr/bin/perl

@out = qw (
130.85.84.194
130.85.111.72
130.85.163.107
130.85.70.129
130.85.112.179
130.85.80.243
130.85.153.174
130.85.1.200
130.85.1.2
130.85.1.5
);

@out = map(quotemeta($_), @out);
$outRE =   join('|', @out);

while (<>) {
        chomp;
        ($month, $day, $time, $src, $arrow, $dst, $proto, $flag) = split;
        ($source_host, $source_port) = split (/:/, $src);
        ($destination_host, $destination_port) = split (/:/, $dsr);
                unless ("$source_host" =~ /\b$outRE\b/) {
                $newline = join (" ", ($month, $day, $src, $arrow, $dst, $proto, $flag));
                print "$newline\n";
                }
}
```

To process the Out Of Spec:

process_oo.pl:

```
#!/usr/bin/perl
# Find the top source and destination IPs that generated OOS
while (<>) {
        chomp;
        if (/->/) { @line = split;
          ($time, $src, $arrow, $dst) = @line;
          ($source_host, $source_port) = split (/:/, $src);
```

```
        ($destination_host, $destination_port) = split (/:/, $dst);
            $source_host{$source_host}++;
            $destination_host{$destination_host}++;
        }

}

print "\n\nTop sources\n\n";
foreach $ip (sort {  $source_host{$b} <=> $source_host{$a}  }  keys %s_h) {
        print "$ip \t$source_host{$ip}\n";
}

print "\n\nTop destinations\n\n";
foreach $ip (sort {  $destination_host{$b} <=> $destination_host{$a}  }  keys %d_h) {
        print "$ip \t\t$destination_host{$ip}\n";
}
```

real_oos.pl

```
#!/usr/bin/perl
# It finds all OOS that don't have the Flag 12****S* set

while(<>) {
        $line[$i] = $_;
        $i++;
} $imax = i;
for ($i = 0; $i <= $imax; $i++) {
        unless (
                (/=\+=/) or
                (/\bDgmLen/)  or
                (/\b12\*\*\*\*S/) or
                (/\bTCP/) or
                (/undef/)
                ) {
                print "$l[$i-4]\n$l[$i-3]\n$l[$i-2]\n$l[$i-1]\n$l[$i]\n$l[$i+1]\n$l[$i+2]\n";
                }
}
```

The processing of alert files was done with:

a. Snarf (Sillicon Defense, http://www.silicondefense.com/software/snortsnarf/)
b. Snortalog (Jeremy Chartier, http://jeremy.chartier.free.fr/snortalog/);
c. Snortstat (Yen-Ming Chen, http://cvs.sourceforge.net/viewcvs.py/snort/snort/contrib/snort_stat.pl)

3.8 References for Part 3 – Analyze This

Securityfocus – "Multiple Vendor LPRng User-Supplied Format String Vulnerability", 2000
http://www.securityfocus.com/bid/1712

CERT "CERT® Advisory CA-2000-22 Input Validation Problems in LPRng", 2000
http://www.cert.org/advisories/CA-2000-22.html

Russell, Ryan "lpdw0rm analysis", 2001
http://cert.uni-stuttgart.de/archive/isn/2001/04/msg00171.html

LURHQ Threat Intelligence Group
http://www.lurhq.com/idsindepth.html

Steers, Cory GCIA Practical, May 2002
http://www.giac.org/practical/GCIA/Cory_Steers_gcia.doc

Roesch, Martin e-mail list answer “Incomplete Packet Fragments Discarded”, November 2001
http://www.mcabee.org/lists/snort-users/Nov-01/msg00820.html

Anonymous, What you need to know about – “Port 0”
http://compnetworking.about.com/library/ports/blports_0.htm

Bryce, Alexander - SANS IDS FAQ – “What you need to know about port 137”, May 2000
http://www.sans.org/resources/idfaq/port_137.php

Forster, Jim – mail list answer “SMB Name Wildcard”, Jan 2000
http://archives.neohapsis.com/archives/snort/2000-01/0222.html

Vision, Max – mail list answer “SMB Name Wildcard”, Jan 2000
http://archives.neohapsis.com/archives/snort/2000-01/0220.html

Rautiainen, Sami “F-Secure Virus Descriptions: Adore”, Aprill 2001
http://www.f-secure.com/v-descs/adore.shtml

Fearnoww, Matt (SANS) Staerns, William (Dartmouth Institute for Security Technology Studies) – SANS “Adore Worm”, April 2001
http://www.sans.org/y2k/adore.htm

Anonymous, e-mail list answer “Port 27374”
(http://www.pcconcrete.ro/phtml/p09288.html

Gordon, Les – GIAC Paper, 2002 November
http://www.giac.org/practical/GCIA/Les_Gordon_GCIA.doc

Jon Hart, SNORT rule SID 648 documentation,
http://www.snort.org/snort-db/sid.html?sid=648

FitzGerald, Nick e-mail list answer “SHELLCODE x86 NOOP”, Oct 2001
http://www.securityfocus.com/archive/75/218557/2001-10-05/2001-10-11/0

Zeltser, Lenny “Practical Assignment for SANS Security“, 2000
http://www.sans.org/dc2000/ID_assignment.htm

Anonymous, Univ of Georgia, “TCP-wrapped rpcbind for Solaris”, Jan 2002
http://www.eits.uga.edu/wsg/sun/security/wrapped_rpcbind.html

Anonymous, Duke University, “Instructions on Cleaning IRC bot & backdoor: XDCC”, 2002
http://security.duke.edu/cleaning/xdcc.html

TonikGin, XDCC – An .EDU Admin’s Nightmare, Sept 2001
http://www.cs.rochester.edu/~bukys/host/tonikgin/EduHacking.html

King, Crow – Beginners’ Administration Tutorial for Medal of Honor: Allied Assault, 2003
http://www.autokick.com/content/adminguide.pdf

Netsys.com - List of User-Agents/Browsers, Web Spiders, Robots, 2003
http://www.netsys.com/cgi-bin/display_article.cgi?1193

Miller, Toby - GIAC Special Notice - “ECN and it’s impact on Intrusion Detection”, 1999
http://www.sans.org/y2k/ecn.htm

Metacmachine, published at Overnet.com – “Edonkey 2000 FAQ”, 2003
http://www.overnet.com/documentation/donkeyfaq.html

RFC 2481 - A Proposal to add Explicit Congestion Notification (ECN) to IP, 1999:
http://www.faqs.org/rfcs/rfc2481.html
RFC 2884 - Performance Evaluation of Explicit Congestion Notification (ECN)
in IP Networks, 2000
http://www.faqs.org/rfcs/rfc2884.html
RFC 3300 - Special-Use IPv4 Addresses
http://www.faqs.org/rfcs/rfc3330.html
Graham, Robert. Firewall Forensics (What am I seeing?), Jan 2003
http://www.robertgraham.com/pubs/7.5
L'équipe d'Infinity Perl. Scanner and Exploit for RTSP (Real Time Streaming Protocol), 2003
http://rafinc.ath.cx/haxor/scanner.pl
CERT. RealNetworks media server RTSP protocol parser buffer overflow, Aug 2003
http://www.kb.cert.org/vuls/id/934932
Wall, Larry; Christiansen, Tom and Orwant, Jon. *Programming Perl*, 3rd edition
O'Reilly & Associates, July 2000.
Christiansen, Tom and Torkington, Nathan. *Perl Cookbook*,
O'Reilly & Associates, August 1998.
Sun Educational Services. *Shell Programming*
SUN Microsystems, SunService, Inc.1993