Reverse Engineering Malicious Binaries

Anthony S. Clark

6/18/2004

This paper will focus on some techniques that can prove useful in the process of reverse engineering malicious binaries. Covered in this paper will be two UNIX based examples of unidentified suspect files and one windows worm. While there are many other techniques and tools that can be used such as gdb, IDA pro, etc., the intent of this paper is to use examples to show the thought process involved in identifying malicious binaries.

CASE 1 – HACKER PENETRATION OF A UNIX SYSTEM

Through various means an organization becomes aware that a hacker has penetrated their systems. Sometimes this is in the form of intrusion detection alerts. Other times it may be through various outside sources. Whatever the notification method an investigation is launched. In this particular investigation during the forensics phase several unknown binaries were discovered and the reverse engineering process was begun in order to determine what the binaries were. This is useful in modeling an unauthorized intruder's actions and tracing their movements.

First a computer is configured with a clean operating system with a known baseline of files and configurations, then the computer is placed on a segregated network in order to prevent possible infections. Next the questionable files are moved by secure means (usually a cd) to the testing system. In this case we used a RedHat Linux 7.3 system. A listing of the files is as follows:

system1|tester|1>ls -al total 346 drwxrwx--- 2 tester
tester   512 2004-06-26 19:54 . drwx--x--- 6 tester
tester   512 2004-06-26 19:35 .. -rwx--x--- 1 tester
tester 27984 2004-06-26 19:54 ...
-rw-rw---- 1 tester tester 20641 2004-06-26 13:19 n

Often the first thing to do is to take an md5sum of the files for comparison later to other files that might be discovered as the investigation progresses.

*system1|tester|2> md5sum **
1778c5355c003599bc2aa118195d258c n

Immediately noticeable is a "*...*" file in the listing. This file was not seen by the initial md5sum so the md5sum is rerun using the filename.

system1|tester|3> md5sum ...
dfdf10dfa4c576642baa00aee79c524e ...

This "..." file is of immediate interest because it's named in such a way as to hide itself from a simple *ls. It would also* be inconspicuous to in-experienced people even if a *ls -al* is run. This is a classic sign of a hacker So this binary is selected first for analysis. First a "file" command is run to see what type of file it is:

system1|tester|4> file ...
...: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), for GNU/Linux 2.2.5, dynamically linked (uses shared libs), not stripped

The results show that it is a binary file. A favorite first pass on any unknown binary is to run "strings". This is a simple check but it often gets an analyst 90% of the way to identifying the binary.

system1|tester|4> strings ...

[... some standard strings stripped …]
USAGE %s <pid to capture>
or:
not enough arguments p:f:
blah, learn to read now
running code42
waiting for auth socket
did not get the auth
socket! changed
saved uid to %d
sendmsg
user interrupt
running code24 Finito
no proc? using default code address 0x%.8x
restoring original process state saving
file descriptor numbers:
-p pid -f fuzz (4 for some apps)
++ + APPCAP application hijacker + + by IhaQueR '2002 +
ATTACHING to pid %d with fuzz %d injecting
code42 into applications address space failed to
change suid, continuing anyway... Auth socket
%s found, sending new tty fd sent %d bytes
[OK] waiting for SIGUSR2
got SIGUSR2, application's tty now redirected
***Ooooown3d**!, press CTRL-C to detach*
Detaching completely...stay on got SIGUSR2,
restored application's tty

The program looks fairly benign at the begging but right near the end is some very useful information. There is some usage information and what appears to be an author banner. Also the word *Ooooown3d* is hacker lingo for “compromised”. This information can help make a signature useful for web research. The banner string "APPCAP application hijacker" can be used in a web search to try to identify the binary.

After entering the following url: *http://www.google.com/search?hl=en&ie=UTF-8&q=APPCAP+application+hijacker* in a web browser we see that the second link says: **appcap** home

... the following command line: root@host:/root/**appcap**> ./**appcap** 32142 +++++ **APPCAP application hijacker** + + by IhaQueR **...**
appcap.ihaquer.com/ - 6k - Cached - Similar pages

Clicking on a link we get a full description of what the program does:

APPCAP HOME

- *What is appcap?*

  Appcap is a tricky application for x86 Linux which allows an user with enough power (usually the superuser) on a machine to attach and redirect standard input and output of any application to his actual tty.

- *Why use appcap?*

Appcap can help you to trace users, for example if you are running a multiuser machine and are suspicious that some of them may be doing nasty things using your machine. It is especially very useful for tracing and monitoring ssh and telnet sessions. Of course you need at least superuser privileges to do that! Note that the session originator will be suspended as long as you are attached to that session.

Along with the banner we saw in the strings output:

```
++++++++++++++++++++++++++++++++++++++++++++
+      APPCAP application hijacker       +
+      by lhaQueR '2002                  +
++++++++++++++++++++++++++++++++++++++++++++
```

At this point we can be fairly confident we know what the binary is and what it does. The tool is used for hijacking user sessions and would allow for jumping into ssh sessions to remote sites as well. Correlating this data with theories and network session data of the initial vector of penetration we can get a clearer picture of what happened and begin to build a model of the intruders actions.

It would still be useful to do a full analysis looking at system calls and network packets but we will leave that for the next binary that we evaluate.

SECOND BINARY

The next file to look at is the "n". The “file” command came up with "*ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), for GNU/Linux 2.2.5, dynamically linked (uses shared libs), not stripped*" so we are dealing with another binary. After performing another strings on it we are disappointed to see that there isn’t much to work with. The only lines that stand out as possible signatures are:

Usage: %s start end <max threads> MEEP
MEEP! IP!

This file thankfully does not appear to be stripped. The strip command removes information which might be useful to us such as, the debug section, line number information, comments, file headers, symbol table, etc. This would make debugging much more difficult by reducing the amount of information we would have to work with. Sometimes an obfuscator such as UPX is used which make it even more difficult to analyze the software by changing the function names to UPX_XXXX etc.

There are no conclusive hits after running various Google searches on those strings. This binary proves to be more difficult to identify. At this point we have to run the program and see if its output gives any clues.

system1|tester|5>./n
Usage: ./n start end <max threads>

This output indicates that the binary might be some kind of multithreaded scanner. I decided to run it again using “*strace”* to see what calls it made and save the output by using “*script*”.

```
system1|tester|6>script Script
started, file is typescript
system1|tester|6>strace ./n
execve("./n", ["./n"], [/* 43 vars */]) = 0
uname({sys="Linux", node="tester.tested.com", ...}) = 0
brk(0)                                  = 0x804a000
old_mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb75ea000
```

[… some standard strace sections stripped …]
open("/lib/tls/libc.so.6", O_RDONLY) = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0PX\1\000"..., 512) = 512 fstat64(3,
{st_mode=S_IFREG|0755, st_size=1567768, ...}) = 0
old_mmap(NULL, 1275852, PROT_READ|PROT_EXEC, MAP_PRIVATE, 3, 0) = 0xb74a0000
old_mmap(0xb75d2000, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED, 3, 0x131000) =
0xb75d2000
old_mmap(0xb75d5000, 10188, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -
1, 0) = 0xb75d5000
close(3) = 0
old_mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb749f000
set_thread_area({entry_number:-1 -> 6, base_addr:0xb749f080, limit:1048575, seg_32bit:1, contents:0,
read_exec_only:0, limit_in_pages:1, seg_not_present:0, useable:1}) = 0 munmap(0xb75d8000, 70325) = 0
fstat64(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 6), ...}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb75e9000
write(1, "Usage: ./n start end <max th"..., 39Usage: ./n start end <max threads>
) = 39
munmap(0xb75e9000, 4096) = 0
exit_group(1) = ?
system1|tester|6> exit exit
Script done, file is typescript

All we really see here is an *execve*() on the program itself, a bunch of standard libc system calls and then it gives back the memory it used. There is an interesting usage line which might be a good signature. The command itself gave no output. So trying again this time putting in an IP range for the start and end arguments as well as 1 for the number of threads to use:

system1|tester|8>script
Script started, file is typescript
system1|tester|7> strace ./n 127.0.0.1 127.0.0.1 1
execve("./n", ["./n", "127.0.0.1", "127.0.0.1", "1"], [/ 43 vars */]) = 0*
uname({sys="Linux", node="tester.testing.com", ...}) = 0 brk(0)
= 0x804a000
old_mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb75ea000
[… some standard strace sections stripped …]
open("/lib/tls/libc.so.6", O_RDONLY) = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0PX\1\000"..., 512) = 512 fstat64(3,
{st_mode=S_IFREG|0755, st_size=1567768, ...}) = 0
old_mmap(NULL, 1275852, PROT_READ|PROT_EXEC, MAP_PRIVATE, 3, 0) = 0xb74a0000
old_mmap(0xb75d2000, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED, 3, 0x131000) =
0xb75d2000
old_mmap(0xb75d5000, 10188, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -
1, 0) = 0xb75d5000
close(3) = 0
old_mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb749f000
set_thread_area({entry_number:-1 -> 6, base_addr:0xb749f080, limit:1048575, seg_32bit:1, contents:0,
read_exec_only:0, limit_in_pages:1, seg_not_present:0, useable:1}) = 0 munmap(0xb75d8000, 70325)
= 0 brk(0) = 0x804a000 brk(0x806b000) = 0x806b000
brk(0) = 0x806b000
***gettimeofday({1087325210, 103213}, NULL) = 0** getpid()*
= 11912
open("/etc/resolv.conf", O_RDONLY) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=125, ...}) = 0

mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb75e9000
read(3, " # Default resolv.conf file"..., 4096) = 125 *read(3, "", 4096) = 0*
close(3) = 0
munmap(0xb75e9000, 4096) = 0
clone(child_stack=0, flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID|SIGCHLD,
child_tidptr=0xb749f0c8) = 11913 exit_group(0)

This is a little more useful. We see here that it opens resolv.conf to do a DNS lookup as well as checking the time of day. *(gettimeofday({1087325210, 103213}, NULL) = 0)* and open("/etc/resolv.conf", O_RDONLY) = 3. The command still gave no output even with the proper arguments.

So its likely this does some kind of network calls. I decided to dig a little deeper with objdump. I left out most of the output because it is not valuable for this analysis and is rather lengthy.

system1|tester|9>objdump -s n

Contents of section .dynstr:
8048368 005f5f67 6d6f6e5f 73746172 745f5f00 .__gmon_start__.
8048378 6c696263 2e736f2e 36007072 696e7466 libc.so.6.printf
8048388 00736e70 72696e74 66006d65 6d637079 .snprintf.memcpy
8048398 00706d61 705f6765 74706f72 74007065 .pmap_getport.pe
80483a8 72726f72 005f5f63 78615f66 696e616c rror.__cxa_final
*80483b8 697a6500 616c6172 6d00696e 65745f61 ize.alarm.inet_**a***
***80483c8 64647200 6e746f68 6c005f5f 64657265 ddr.ntohl**.__dere*
80483d8 67697374 65725f66 72616d65 5f696e66 gister_frame_inf
80483e8 6f007761 69740073 69676e61 6c006874 o.wait.signal.ht
*80483f8 6f6e6c00 666f726b 00696e65 745f6e74 onl.fork.**inet_nt***
8048408 6f610067 6574686f 73746279 6e616d65 oa.gethostbyname
8048418 0068746f 6e730065 78697400 61746f69 .htons.exit.atoi
8048428 005f494f 5f737464 696e5f75 73656400 ._IO_stdin_used. 8048438
5f5f6c69 62635f73 74617274 5f6d6169 __libc_start_mai
8048448 6e005f5f 72656769 73746572 5f667261 n.__register_fra
8048458 6d655f69 6e666f00 636c6f73 6500474c me_info.close.GL
*8048468 4942435f 322e312e 3300474c 4942435f IBC_2.1.3.GLIBC_*
8048478 322e3000 2.0.

So there is a call to addr.ntohl and one to *inet_ntoa.gethostbyname.htons.exit.atoi* and both of these are network calls for resolving hostnames.

In section *Contents of section .rodata:* we see:

8048be0 55736167 653a2025 73207374 61727420 Usage: %s start
8048bf0 656e6420 3c6d6178 20746872 65616473 end <max threads
8048c00 3e0a004d 45455020 4d454550 21204950 >..MEEP MEEP! IP
***8048c10 210a0025 7300666f 726b0025 73202d20 !..%s.fork.%s -** 8048c20*
25640a00 00000000 00000000 00000000 %d.....

Note the "fork". This further supports the theory that this application “n” is multi-process as seen by the fork that happens here. The MEEP MEEP! IP might be useful in a web search. Later on in *Contents of section .stabstr:* we see:

0a40 2e2e2f6c 696e7578 74687265 6164732f ../linuxthreads/
0a50 73797364 6570732f 70746872 6561642f sysdeps/pthread/

0a60 62697473 2f707468 72656164 74797065 bits/pthreadtype
0a70 732e6800 2e2e2f73 79736465 70732f75 s.h.../sysdeps/u
0a80 6e69782f 73797376 2f6c696e 75782f62 nix/sysv/linux/b
0a90 6974732f 73636865 642e6800 5f5f7363 its/sched.h.__sc
...etc.

At this point I am convinced this is a multithreaded / forking application which does DNS lookups. What is the point of the program though? More objdump investigation is needed.

system1|tester|10>objdump -x n

This gives us a ton of information which we look through for anything useful. Partway down the output of objdump I start to see useful information:

*00000000 l df *ABS* 00000000 **nfs.c***
08048820 l .text 00000000 gcc2_compiled.
*080485c0 F *UND* 00000007 ntohl@@GLIBC_2.0*

and then:

08049c5c g O .data 00000004 start_howarewe
08049dc4 g O .bss 00000004 sock
08048a40 g F .text 00000058 resolve
*08048620 F *UND* 00000007 htonl@@GLIBC_2.0*
08049dc8 g O .bss 00000004 cnt
08049c54 g O .data 00000004 kiddies
*08048a98 g F .text 000000b6 **CheckRPC***
*08048630 F *UND* 00000105 inet_ntoa@@GLIBC_2.0*
08048598 g F .init 00000000 _init
08049c60 g O .data 00000004 end_howarewe

This looks rather compelling. What we see here is that this program includes **nfc.c** which is probably some sort of NFS related function and that it has a function called **howarewe()** which opens a socket, resolves a hostname and then appears to probe RPC. This information points to the possibility of this being a multithreaded NFS scanner.

At this point we have gotten all we can from objdump and we could go a couple of different ways. We could either run the program under a debugger and try to figure out more of what it does step by step or we could run it in such a way that we can sniff the network traffic and examine the packets. I opted for the network route because I believe that objdump got most of the info that we need from a debugging point of view.

In this particular case I ran "*ethereal*" on the target and then ran the "n" program against it and examined the packets.

system1|tester|11>./n 192.168.1.2 192.168.1.2 1

Strangely enough the binary still gives no output even when run against a remote active host.

I saw some packets coming from the attacking host:

No. Time Source Destination Protocol Info 1 4.443368 192.168.1.1 192.168.1.2 NFS V3 GETATTR Call, FH:0x4e1784c4 2 4.443847 192.168.1.2 192.168.1.1 NFS V3 GETATTR Reply (Call In 128)

So this looks like an NFS session. Looking deeper into the packets I see that the request packet is about 162 bytes every time I run it and the reply packet is about 154 bytes. The don't fragment flag is set and this packet is a UDP type packet. The source port is 795 and the destination port is 2049. It looks like its expecting NFS protocol version 3.

Now we know that the program wants to do something with NFS. I decided to complement the "*ethereal*" data with some "*tcpdump"* data.

system2|tester|1>tcpdump -s 0

14:32:24.899195 attacker.testing.com.32820 > victim.testing.com.sunrpc: udp 56 (DF) 14:32:24.899283 victim.testing.com.sunrpc > attacker.testing.com.32820: udp 28 (DF)

So we see there that it is talking to SUNRPC probably the portmapper. I added a filter to "*tcpdump*" to be sure:

system2|tester|2>tcpdump udp port < = 111

4:32:24.899195 attacker.testing.com.32820 > victim.testing.com.sunrpc: udp 56 (DF) 14:32:24.899283 victim.testing.com.sunrpc > attacker.testing.com.32820: udp 28 (DF)

So it is talking UDP on source port 32820 and destination port 111.

On the victim I decided to strace the portmapper and then rerun the program and see if anything happens.

system2|tester|3>ps -ef |grep port
rpc 3185 1 0 Jun07 ? 00:00:00 portmap

system2|tester|4>strace -p 3185
Process 3185 attached - interrupt to quit poll(

It sat there for a moment and then when I ran the program from the attacking host:

system1|tester|12>./n 192.168.1.2 192.168.1.2 1

[{fd=3, events=POLLIN|POLLPRI|POLLRDNORM|POLLRDBAND, revents=POLLIN|POLLRDNORM}, {fd=4, events=POLLIN|POLLPRI|POLLRDNORM|POLLRDBAND}, {fd=-1}, {fd=-1}], 4, -1) = 1
recvmsg(3, {msg_name(16)={sa_family=AF_INET, sin_port=htons(32820), ***sin_addr=inet_addr("192.168.1.1")},***
msg_iov(1)=[{"!h\213\20\0\0\0\0\0\0\2\0\1\206\240\0\0\0\2\0\0\0\3\0"..., 8800}], msg_controllen=24, msg_control=0x8050060, , msg_flags=0}, 0) = 56
getpid() = 3185
open("/etc/hosts.allow", O_RDONLY) = 5 *fstat64(5,*
{st_mode=S_IFREG|0444, st_size=947, ...}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb75e9000
read(5, "ftpd:\t\t\t127.0.0.1, 192.168.1.,"..., 4096) = 947 close(5) = 0
munmap(0xb75e9000, 4096) = 0
sendmsg(3, {msg_name(16)={sa_family=AF_INET, sin_port=htons(32820), ***sin_addr=inet_addr("192.168.1.1")},***
msg_iov(1)=[{"!h\213\20\0\0\0\1\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"..., 28}], msg_controllen=24, msg_control=0x8050060, , msg_flags=0}, 0) = 28 poll(

It definitely talked to the portmapper which check its wrappers in hosts.allow. Now it is pretty convincing that this application is some kind of NFS scanner. The next step was to run it against a host that was actually running NFS and see what happened. I configured the victim host to run NFS and started the test again.

system1|tester|13> ./n 192.168.1.2 192.168.1.2 1
192.168.1.2 - 2049

Jackpot. The mysterious binary is a scanner for machines running NFS. It uses UDP probes to the portmapper on port 111 to determine if NFS is running and then the attacker can run "*showmount –e*" to determine what file systems are exported. The scanner is very fast and since many organizations that run protocols such as NFS and NIS have tons of RPC and port 111 traffic, it is possible and likely that these probes would be lost in the noise and not picked up by an IDS.

Google searches turned up nothing definitive on a scanner of this type. No information about the author was identified inside the utility so it is unknown who wrote it or where it can be found on the internet. Its possible that this was a custom coded tool by the attacker.

http://www.honeynet.org/ is a great source for finding more information about hacker binaries. I searched honeynet to see if it had a record of this scanner using both the strings identified earlier as well as "SUNRPC", "port 111", "NFS" and the closest utility that could be found is "*luckscan-a*". I downloaded and compiled this utility and compared it to "n" but they did not match up. At that point I ended the analysis of the "n" binary.

CASE 2 WINDOWS WORM

On April 13, 2004 Microsoft released security bulletin MS04-011 which outlined a serious problem with LSASS among other parts of windows which could lead to remote code execution and administrative privilege level compromise.

http://www.microsoft.com/technet/security/bulletin/MS04-011.mspx

On April 30, 2004 the W32.Sasser.Worm was first seen in the wild. This worm exploited the LSASS vulnerability.

The antivirus companies typically do a good job of analyzing worms and viruses but they don't often provide a signature which can be easily used for intrusion detection or automatic blocking systems. It can be useful to perform an analysis in order to develop these signatures and to get a clearer picture of the threat.

Sometimes acquiring live virus code can be challenging. There are several companies which offer virus code for a fee but in this case we found a system that was infected and segregated it.

Since I had an infected machine I started from the network point of view. On a segregated network I put a sniffer (ethereal) on and watched for worm traffic.

The worm sends a 62 byte packet with the following attributes:

microsoft-ds SYN

flags 0x0002 SYN

window size 64512

The Payload:

```
00 04 75 8b 84 be 00 c0  a8 81 7e ee 08 00 45 00  ..u.....  ..~...E. 00
30 26 34 40 00 80 06  8f f1 ac 1e 10 1a ac 1e  l0&4@...  ........ dc 4b
07 3c 01 bd 0f fb  2c 6d 00 00 00 00 70 02  .K.<....  ,m....p.
fc 00 fd 1a 00 00 02 04  05 b4 01 01 04 02        ........  ......
```

Much of the packet changes each time but the last chunk is always re-occurring: 05 b4 01 01 04 02

The first part is constant as well:
00 04 75 8b 84 be 00 c0 a8 81 7e ee 08 00 45 00 ..u..... ..~...E.

here is some timing between packets:

8.690632 ms
8.690976 ms
8.691723 ms
8.692742 ms

The network connections are opened very quickly. Based on the packets the worm appeared to go after a totally random IP each time it opened a connection.

At that point I had a good network signature. I decided to go deeper into the sasser worm.

I nmapped the box using nmap -p 1-65535 to find all ports and the output is as follows:

==========FULL NMAP==========

(The 65531 ports scanned but not shown below are in state: closed)
Port State Service
135/tcp open loc-srv
139/tcp open netbios-ssn
[BAD!] --> 5554/tcp open unknown

==========PORT PROBING========== nc 172.30.16.102 5554
220 OK

(nc is Netcat. Telnet could also be substituted for nc in some cases.)

I decided to ftp to the infected host from a Linux box and send it a file.
I then retrieved the same file and noticed it was different so I performed md5sums and checked the sizes. I then compared the md5sum of what I got back from the ftp server with that of the actual virus and they matched. No matter what filename you ask for from the ftp server you will get a copy of the virus with that name.

echo test > test.txt md5sum test.txt
d8e8fca2dc0f896fd7cb4cb0031ba249 test.txt ls -al test.txt
-rw-r--r-- 1 root root 5 May 3 13:58 test.txt

md5sum test.txt
1fdb0075e5ed17d256bd88cbf79d1e23 test.txt
ls -al test.txt
-rw-r--r-- 1 root root 15870 May 3 13:59 test.txt

md5sum test.txt

1a2c0e6130850f8fd9b9b5309413cd00 test.txt

ls -al test.txt
-rw-r--r-- 1 root root 15872 May 3 13:59 test.txt

md5sum virus.exe
1a2c0e6130850f8fd9b9b5309413cd00 virus.exe

*C:\Documents and Settings\rasclark\Desktop>md5sum 30208_up.exe 1a2c0e6130850f8fd9b9b5309413cd00 *30208_up.exe*

Then I started looking at the computer itself. Sysinternals (http://www.sysinternals.com/) makes a great tool called TCPVIEW which shows what ports are open on a box and what processes opened the ports.

TCPVIEW reports a TON of listening ports attached to the process 30208_up.exe. 30208_up.exe is the initial virus file. These ports open and close consistently which correspond with the sniffer data about the outgoing port 445 packets every fraction of a second.

30208_up.exe:464 TCP nicole:4646 nicole:0 LISTENING
30208_up.exe:464 TCP nicole:4647 nicole:0 LISTENING
30208_up.exe:464 TCP nicole:4648 nicole:0 LISTENING
30208_up.exe:464 TCP nicole:4649 nicole:0 LISTENING
30208_up.exe:464 TCP nicole:4650 nicole:0 LISTENING
30208_up.exe:464 TCP nicole:4651 nicole:0 LISTENING
30208_up.exe:464 TCP nicole:4652 nicole:0 LISTENING

The worm also puts randomly named files in c:\winnt\system32\ that is usually a 4 digit number like 7128_up.exe.

Also in the TCPVIEW output under the lsass.exe running process every so often (about every 2 minutes) a cmd.exe process is opened as well as a ftp.exe process.

Using another tool called autoruns, also from Sysinternals, I looked at what was setup to run automatically:

==========REGISTRY AUTORUNS==========

[… Some standard autorun information stripped …]
+ D:\Program Files\Java\j2re1.4.2_01\bin\jusched.exe D:\Program Files\Java\j2re1.4.2_01\bin\jusched.exe
+ "D:\Program Files\QuickTime\qttask.exe" -atboottime Apple Computer, Inc. D:\Program Files\QuickTime\qttask.exe
[BAD!] --> + C:\WINNT\avserve2.exe C:\WINNT\avserve2.exe
HKCU\Software\Microsoft\Windows\CurrentVersion\Run
+ C:\WINNT\System32\ctfmon.exe CTF Loader Microsoft Corporation C:\WINNT\system32\ctfmon.exe
HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\ShellServiceObjectDelayLoad
+ PostBootReminder -> C:\WINNT\system32\shell32.dll Windows Shell Common Dll Microsoft Corporation
C:\WINNT\system32\shell32.dll
+ CDBurn -> C:\WINNT\system32\shell32.dll Windows Shell Common Dll Microsoft Corporation
C:\WINNT\system32\shell32.dll

The avserve2.exe should not be there. A quick Google search showed this to be malicious. The good news is now we had the names of the virus binaries that we could reverse engineer. At this point we've gotten pretty far and could almost end the investigation here because everything we need to make a network based signature is available. Before doing that I decided to copy the virus binary to a Linux box for some further analysis.

file 30208_up.exe
30208_up.exe.dat: MS-DOS executable (EXE), OS/2 or MS Windows

strings didn't turn up any useful info.

Objdump -p shows that the worm makes use of kernel32.dll.

objdump -p 30208_up.exe

DLL Name: kernel32.dll vma:
Hint/Ord Member-Name Bound-To
9059 0 LoadLibraryA
9069 0 GetProcAddress
907b 0 VirtualAlloc
908b 0 VirtualFree

objdump -g 30208_up.exe.dat
30208_up.exe.dat: file format efi-app-ia32
objdump: 30208_up.exe.dat: no recognized debugging information

Other objdump commands turned up nothing. This is a pretty clean binary. At the point the investigation was closed.

CONCLUSION

The various methods and techniques discussed in this paper have proven to be successful in identifying malicious binaries. The examples show a thought process and set of steps which aid in making the analysis process successful and efficient.