Anatomy of a Remote Kernel Exploit

(Dartmouth Edition)

Dan Rosenberg
Who am I?

- Security consultant and vulnerability researcher at Virtual Security Research in Boston
  - App/net pentesting, code review, etc.
  - Published some bugs
  - Rooted a few Android phones
  - Focus on Linux kernel
  - Research on kernel exploitation and mitigation
Agenda

▪ Motivation

▪ Challenges of remote exploitation

▪ Prior work

▪ Case study: ROSE remote stack overflow
  ▫ Exploitation
  ▫ Backdoor

▪ Future work
Motivation

Why am I giving this talk?
Why Remote Kernel Exploits?

▪ Instant root
  ▫ No need to escalate privileges

▪ Remote userland exploitation is hard!
  ▫ Full ASLR + NX/DEP
  ▫ Sandboxing
  ▫ Reduced privileges
Goals of This Talk

▪ Explore operating system internals from perspective of an attacker

▪ Discuss kernel data structures and subsystems

▪ Exploit development methodology

▪ Individual bugs vs. exploit techniques

▪ Discuss next steps for kernel hardening
Challenges of Remote Kernel Exploitation

Wait, so you mean this is kind of hard?
Warning: Fragile

- Consequence of failed remote userland exploit:
  - Crash application/service, wait until restarted
  - Crash child process, try again immediately

- Consequence of failed remote kernel exploit:
  - Kernel panic, game over
Lack of Environment Control

▪ Typical local kernel exploit:
  ▫ Can trigger allocation of heap structures
  ▫ Can trigger calling of function pointers
  ▫ High amount of information leakage available to local users

▪ Remote kernel exploit:
  ▫ ?
Primer: Process vs. Interrupt Context

▪ Systems calls occur in “process context”:
  ▫ Kernel is executing code, but is associated with userland process
  ▫ Has credentials, network/filesystem namespace, etc.

▪ On Linux, asynchronous events (e.g. network data) occur in “interrupt context”:
  ▫ Network driver generates hardware interrupt
  ▫ Kernel dispatches data to appropriate softirq handler
  ▫ No userland process associated with execution
  ▫ On Linux, associated with softirqd kernel thread
Escape From Interrupt Context

- End goal: userland code execution (remote shell)
  - How do we get there?
  - No process backing execution

- Need to transition
  - Interrupt context to process context to userland
Prior Work

What's been done before?
A Few Statistics

▪ 18 known exploits for 16 vulnerabilities
  ▫ 19 authors
  ▫ 9 with full public source code
  ▫ 3 with partial or PoC source

▪ Wide range of platforms
  ▫ Solaris and OS X still need some remote love
By Vulnerability Class

- Stack Overflow: 12
- Heap Overflow: 3
- Array Indexing: 1

Legend:
- Stack Overflow
- Heap Overflow
- Array Indexing
Highlights

▪ Barnaby Jack: Step into the Ring 0 (August 2005)
  ▫ First publication on remote kernel exploitation
  ▫ Transition to userland and kernel backdoor

▪ Sinan Eren: GREENAPPLE (May 2006)
  ▫ First remote kernel exploit in Immunity CANVAS
Highlights (cont.)

▪ hdm, skape, Johnny Cache (November 2006)
  ▫ Broadcom, Dlink, and Netgear wifi drivers
  ▫ First remote kernel exploits in Metasploit

▪ Alfredo Ortega, Gerardo Richarte: OpenBSD IPv6 mbuf overflow (April 2007)
  ▫ First public remote kernel heap overflow
  ▫ Bypasses userland NX
Primer: NX (Non-Executable Pages)

- Pages have permissions: read, write, execute
  - Initially, on Intel chips, page table entries only supported read and write flags
  - Read implied executable

- Before long, realized this was a bad idea
  - Malicious data can be executed as code!

- NX is implemented using 63\textsuperscript{rd} bit of page table entry:
  - Natively supported on 64-bit platforms
  - Supported on PAE CPUs (need hardware + software)
  - Emulated in userland by kernel
Highlights (cont.)

▪ Kostya Kortchinsky: MS08-001 (January 2008)
  ▫ Immunity CANVAS
  ▫ First publicized remote Windows kernel pool overflow

▪ sgrakkyu: sctp-houdini (April 2009)
  ▫ First remote Linux sI*b overflow
  ▫ Introduced vsyscall trick to transition from interrupt context to userland
Primer: Linux Virtual Syscalls

▪ On x86-64 machines, Linux supports “virtual syscalls”
  ▫ Three system calls that can be implemented entirely in userland: gettimeofday, getcpu, time

▪ Trapping to kernel mode is relatively expensive
  ▫ Check CPL, switch stack, store trap frame, reload %cs and %ss

▪ Faster to just stay in userland

▪ “vsyscall” page accomplishes this by mapping a page exported by the kernel into every userland process
So What Was That Trick?

- Sgrakkyu realized this is a good attack vector

- vsyscall page is a shadowed mapping: read-write version in kernel memory, read-execute in user memory

- In interrupt context, we can write into the kernel mapping of this page, overwriting a virtual syscall

- Now every userland process will execute our userland shellcode whenever they call a virtual syscall!
Observations

▪ Majority stack overflows, but none dealt with NX kernel stack
  ▫ Let's fix that

▪ No Linux interrupt context stack overflows
  ▫ sgrakkyu and twiz showed us how in Phrack 64, let's do it in real life

▪ Wireless drivers suck
  ▫ Six 802.11 remote kernel exploits
Building the Exploit

Or: How I Learned to Stop Worrying and Love the Ham
Target: 32-bit x86 PAE Kernel

- Kernel has NX support (CONFIG_DEBUG_RODATA)
  - Only enforced on PAE (32-bit) or 64-bit kernels

- Can't execute first-stage shellcode on kernel stack

- Can't introduce code into userspace without proper page permissions

- No vsyscall trick for easy transitions
Test Setup

- Attacker and victim VMs (Ubuntu 10.04)
- Debugging using KGDB over virtual serial port (host pipe)
- BPQ (AX.25 over Ethernet)
- Except for glue code, exploit written entirely in x86 assembly
Debian Security Advisory DSA-2240-1:

Dan Rosenberg reported two issues in the Linux implementation of the Amateur Radio X.25 PLP (Rose) protocol. A remote user can cause a denial of service by providing specially crafted facilities fields.
Intro to ROSE

- Rarely used amateur radio protocol
- Provides network layer on top of AX.25's link layer
- Uses 10-digit addresses and AX.25 callsigns
- Static routing only
CVE-2011-1493

- On initiating a ROSE connection, parties exchange facilities (supported features)
  - FAC_NATIONAL_DIGIS allows host to provide list of digipeaters
  - Parsing for this field reads length value from frame and copies digipeater addresses without bounds checking, causing a stack overflow
Sad Code :-(

... 

l = p[1];
...

else if (*p == FAC_NATIONAL_DIGIS) {
    fac_national_digis_received = 1;
    facilities->source_ndigis = 0;
    facilities->dest_ndigis   = 0;
    for (pt = p + 2, lg = 0 ; lg < l ; pt += AX25_ADDR_LEN, lg += AX25_ADDR_LEN) {
            memcpy(&facilities->dest_digis[facilities->dest_ndigis++], pt, AX25_ADDR_LEN);
        else
            memcpy(&facilities->source_digis[facilities->source_ndigis++], pt, AX25_ADDR_LEN);
    }
}
...
Constraint #1

▪ The seventh byte of an AX.25 address is AND'd with AX25_HBIT (0x80) if it's a destination digipeater
  ▫ Otherwise, treated as a source digipeater

▪ Every seventh byte of our payload needs to be consistently greater or less than 0x80, or we'll copy into the wrong array

▪ Requires manual tweaking
Plan of Attack

1. Get EIP
2. Unrestricted code execution
3. Install kernel backdoor
4. Restore and recover
Triggering the Bug

- Fairly trivial
- Modify ROSE facilities output functions to craft frame with overly large length field for FAC_NATIONAL_DIGIS, followed by lots of NOPs (0x90)
## Evil ROSE Frame

<table>
<thead>
<tr>
<th>ROSE header</th>
<th>Facilities Total Length = XX</th>
<th>0x00</th>
<th>FAC_NATIONAL</th>
<th>FAC_NATIONAL_DIGIS</th>
<th>len = 0xff</th>
<th>0x9090...</th>
</tr>
</thead>
</table>


Got EIP

- Recompile ROSE module, reload, and use rose_call to initiate connection to target

- Overflowed softirq stack (interrupt handler)

Program received signal SIGSEGV, Segmentation fault.
[Switching to Thread 1456]
0x90909090 in ?? ()
(gdb) i r
eax 0x0 0
ecx 0xde3a5f3c -566599876
edx 0x296 662
ebx 0x90909090 -1869574000
esp 0xd11e199c 0xd11e199c
ebp 0x90909090 0x90909090
esi 0x90909090 -1869574000
edi 0x90909090 -1869574000
eip 0x90909090 0x90909090
eflags 0x10286 [ PF SF IF RF ]
cs 0x60 96
ss 0x68 104
ds 0x9090007b -1869610885
es 0x9090007b -1869610885
fs 0xffff 65535
gs 0xffff 65535
How to Execute Code?

- Traditionally, return into shellcode on stack
- Problem 1: we don't know where we are
  - Trampolines are easy
- Problem 2: softirq stack is non-executable

Get EIP

Unrestricted code execution

Install kernel backdoor

Restore and recover
Primer: Registers

▪ x86-32 has several general purpose registers:
  ▫ %eax, %ebx, %ecx, %edx, %esi, %edi

▪ Some have “traditional” uses
  ▫ %eax is return code
  ▫ %ecx is a counter
  ▫ %esi/%edi are source and destination of copy

▪ Special registers: %esp (stack pointer), %ebp (frame pointer), %eip (instruction pointer)
Primer: Calling Convention

▪ How do we invoke functions?
  ▫ Traditionally, put arguments on stack (%esp), and issue a “call” instruction

▪ Different in kernel mode:
  • First argument in %eax
  • Second in %edx
  • Third in %ecx
  • Others on stack
Primer: ROP

▪ We control the return address and data at %esp

▪ Each return will direct execution to address at stack pointer and increment it

▪ Chain together function epilogues (“gadgets”) to perform arbitrary computation

▪ Relies on homogeneity of distribution (binary) kernels and lack of randomization
  ▫ Choose gadgets that are more likely to appear in constant locations across kernels
Making our Stack Executable

- Kernel has nice function to do this for us:
  - `set_memory_x()`

- Calling convention has arguments in registers

- ROP stub steps:
  - Load (%esp & ~0xfff) into %eax
  - Load 4 into %edx
  - Call `set_memory_x()`
  - Jump into stack

```c
static unsigned long rop_stub[] = {
    /*1*/   PUSH_ESP_POP_EAX,
    /*4*/   0xffffffff,
             0xffffffff,
    /*3*/   0xffffffff,
             ALIGN_EAX,
    /*2*/   0xffffffff,
             0xffffffff,
    /*1*/   RET,
    /*4*/   POP_EDX,
             0x00000004,
    /*3*/   0xffffffff,
             0xffffffff,
    /*2*/   0xffffffff,
             0xffffffff,
    /*1*/   RET,
    /*4*/   SET_MEMORY_X,
             JMP_ESP,
};
```
Overcoming Space Constraints

- We now have traditional shellcode executing on the softirq stack!
- Problem: length is limited to 0xff (255), minus what we've already used
- Not enough room for a useful payload
Needle in a Haystack

▪ Full ROSE frame is intact somewhere on the kernel heap

▪ Pointer to a memory region containing our socket data lives on the stack

▪ Walk up the stack, following kernel heap pointers

▪ Search general area for tag included in ROSE frame

▪ Mark it executable and jump to it
What Now?

- We can execute arbitrary-length payloads now!
- Goal: install kernel backdoor in ICMP handler
Primer: Linux Networking

▪ What happens when network data is received?

▪ Hardware magic happens, driver layer (linux/drivers/net) receives low-level frame

▪ Driver identifies “this is an IP packet”, sends to network layer (linux/net/ipv{4,6})

▪ Network layer checks “what protocol is this” (TCP, UDP, ICMP, etc.) and dispatches to appropriate protocol handler (linux/net/*)
Protocol Handlers

/* Array of network protocol structure */
const struct net_protocol __rcu *
inet_protos[MAX_INET_PROTOS] __read_mostly;

/* Definition of network protocol structure */
struct net_protocol {
    int   (*handler)(struct sk_buff *skb);
    void (*err_handler)(struct sk_buff *skb, u32 info);
    ...
};

/* Standard well-defined IP protocols. */
enum {
    IPPROTO_IP = 0,  /* Dummy protocol for TCP */
    IPPROTO_ICMP = 1, /* Internet Control Message Protocol */
    ...
};
Hooking ICMP

▪ Storage on softirq stack
  ▫ Already executable, safe, persistent

▪ Copy hook and address of original ICMP handler
  ▫ We'll need this later

▪ Handler is in read-only memory
  ▫ Flip write-protect bit in %cr0 register

▪ Write address of our hook into ICMP handler function pointer
Hooked In

inet_protos:
- IPPROTO_IP
- IPPROTO_ICMP
- ...

net_protocol:
- handler
- err_handler
- ...

hook:
- <hook>: push edi
- <hook+1>: push esi
- <hook+2>: push ebx
- <hook+3>: push eax
- ...

icmp_rcv:
- <icmp_rcv>: push ebp
- <icmp_rcv+1>: mov ebp, esp
- <icmp_rcv+3>: push edi
- <icmp_rcv+4>: push esi
- ...

Time to Rebuild...

- We've destroyed large portions of the softirq stack
- How can we keep the kernel running?
Cleaning Up the Locks

- ROSE protocol is holding two spinlocks
  - If we don't release these, the ROSE stack will deadlock soon

- Problem: ROSE is a module, we don't know where the locks live
Needle in a Haystack, Again

- Global modules variable: linked list of loaded kernel modules

- A plan!
  - Follow linked list until we find ROSE module
  - Read module structure, find start of .data section
  - Scan .data section for byte pattern of two consecutive spinlocks (distinctive signature)
  - Release them
Preemption Woes

- Preemption count must be consistent with what the kernel is expecting, or scheduler will...

...complain and fix it for you?!

```c
if (unlikely(prev_count != preempt_count())) {
    printk(KERN_ERR "huh, entered softirq %u %s %p"
            "with preempt_count %08x,"
            " exited with %08x?\n", vec_nr,
            softirq_to_name[vec_nr], h->action,
            prev_count, preempt_count());
    preempt_count() = prev_count;
}
```

- Let's avoid that warning...
Has Anybody Seen a Preemption Count?

- Preempt count lives at known location in thread_info struct, at base of kernel stack:

```c
struct thread_info {
    struct task_struct *task; /* main task structure */
    struct exec_domain *exec_domain; /* execution domain */
    __u32 flags;        /* low level flags */
    __u32 status;       /* thread synchronous flags */
    __u32 cpu;          /* current CPU */
    int preempt_count;  /* 0 => preemptable, <0 => BUG */

    ...  // Continued...
};
```

- Decrement it and we're done
Unwinding the Stack

- Stack is partially corrupted from overflow
- Need to restore it to recoverable state
- Walk up stack from current location until we match a signature of a known good state
- Adjust ESP to good state, and return
Refresher: What Have We Achieved?

- Trigger the overflow, gain control of EIP

- Leverage ROP to mark softirq stack executable, jump into shellcode

- Search for intact ROSE frame on kernel heap, mark executable, jump into it

- Install kernel backdoor by hooking ICMP handler

- Do some necessary cleanup and unwind stack for safe return from softirq
Kernel Backdoors for Fun and Profit

(Insert “backdoor” joke)
What About That Backdoor Part?

- Whenever an ICMP packet is received, our hook is called

- Check for magic tag in ICMP header

- Two distinct types of packets
  - “Install” packets contain userland shellcode
  - “Trigger” packets cause shellcode to execute

- May be sent independently
  - Install payload, trigger it repeatedly at later date
Backdoor Strategy

- Problem: ICMP handler also runs in softirq context
  - Want userland code execution

- Phase 1: transition to kernel-mode process context

- Phase 2: hijack userland control flow
Backdoor Phase 1

- Check for magic tag and packet type
- If “install” packet, copy userland payload into safe place (softirq stack)
Transition to Process Context

- If “trigger” packet, need to transition to process context
- Easiest way: hook system call

1. Install userland payload
2. Hook system call
3. Continue execution
Primer: System Calls

▪ Userland process invokes a system call (read, write, fork, etc.)

▪ Traditional mechanism is int 0x80 (more recently everything uses systenter/syscall)

▪ Index into Interrupt Descriptor Table, check privileges

▪ Invokes handler specified by IDT (syscall entry point)

▪ Syscall entry point parses arguments, indexes into syscall table, and calls appropriate system call handler
System Call Hijacking

▪ How to find system call table at runtime?
  ▫ sidt instruction retrieves IDT address
  ▫ Find handler for INT 0x80 (syscall)
  ▫ Scan function for byte pattern calling into syscall table

▪ Read-only syscall table
  ▫ More flipping write-protect bit in %cr0

▪ Store original syscall handler for later, write address of hook into syscall table
Carry On...

- Install userland payload
- Hook system call
- Continue execution

- Want working ICMP stack
- Call original ICMP handler
Backdoor Phase 2

- We've copied userland payload to kernel memory

- Some process comes along and calls our hooked system call...

- Need to hijack process for userland code execution
Only Root, Please

- Only interested in root processes

- How to verify?
  - `thread_info → task_struct → cred`
  - Unstable, annoying...

Check root privileges

Inject userland payload

Divert userland execution

Continue execution
System Calls from Kernel Mode?

- System calls are extremely useful abstractions
  - Friendly interface, kernel does most of the work

- Poll: is it possible to call system calls via INT 0x80 from kernel mode?
  - Tally your votes...
System Calls from Kernel Mode!

- Most system calls will work when called from kernel

- Stack switch only occurs on inter-PL interrupts
  - Based on CPL vs. DPL of GDT descriptor
  - Happens on int and iret

- When called from kernel mode, just an ordinary intra-PL interrupt
Exceptions (No Pun Intended)

▪ Doesn't work quite right with some system calls
  ▪ Some require pt_regs (per-thread register) structure
  ▪ Assumptions about state of stack at time of system call

▪ fork, execve, iopl, vm86old, sigreturn, clone, vm86, rt_sigreturn, sigaltstack, vfork
Checking for Root

- Easy: load %eax with 0x18 (getuid), INT 0x80
- Check %eax (return code) for 0
- If not zero, call original syscall handler for hooked function
- If zero, unhook syscall and continue payload
Lethal Injection

- Kernel stack contains pointer to saved userland %esp
- Copy userland payload from kernel memory to userland stack
Let it Run...

- Userland stack is non-executable (NX)
- Call mprotect syscall via INT 0x80 to mark userland stack executable
It's a Diversion!

- Need to redirect userland control flow

- Kernel stack contains pointer to saved userland %eip

- Give original saved %eip to userland shellcode for later

- Overwrite pointer with address of payload on userland stack
Keep on Running

- Want hijacked process to keep running
- Jump to original handler for hijacked system call
Userland Payloads

▪ Use your imagination!
  ▫ Connect-back root shells work just fine

▪ Payloads are prefixed with stub that keeps hijacked process running
  ▫ Fork new process
  ▫ Child runs shellcode
  ▫ Parent jumps to original saved %eip
ROSE Exploitation Demo
Future Work

No, this isn't a perfect exploit.
Hard-Coding

▪ Advantages over signatures / fingerprinting
  ▫ Reliability vs. portability

▪ On PAE kernel, ROP gadgets seem unavoidable
  ▫ Minimize number of ROP gadgets
  ▫ Minimize hard-coding of other data structures

▪ On non-PAE kernel, situation is better
  ▫ Can survive with one JMP ESP (if you know saved EIP offset)
  ▫ Partial overwrites or spraying possible
Future Work: Offense

▪ Remote fingerprinting of kernel
  ▫ Automatic generation of ROP gadgets

▪ Exploiting other packet families
  ▫ IrDA, Bluetooth, X.25?

▪ Finding that TCP/IP bug that breaks the Internet
Future Work: Defense

▪ Randomize kernel base at boot
  ▫ Prevents code reuse (e.g. ROP) remotely in absence of remote kernel memory disclosure

▪ Fuzz and audit networking protocols more rigorously

▪ Inline functions that alter page permissions directly (prevent easy ROP)

▪ Policies on preventing page permission modification after initialization
Thanks To...

- Ralf Baechle
- Nelson Elhage
- Kees Cook
- twiz, sgrakkyu
Questions?

E-mail: drosenberg@vsecurity.com
Twitter: @djrbliss

Company:
http://www.vsecurity.com

Personal:
http://www.vulnfactory.org

Exploit code:
https://github.com/djrbliss/rose-exploit