Module: Kernel

Privilege Escalation

Yan Shoshitaishvili
Arizona State University

Kernel Memory Corruption

Recall:

```
copy_to_user(userspace_address, kernel_address, length);
copy_from_user(kernel_address, userspace_address, length);
```

Kernel memory must be kept uncorrupted! Corruption can:

- crash the system
- **brick** the system
- escalate process privileges
- interfere with other processes

All user data should be carefully handled (haha) and only accessed with `copy_to_user` and `copy_from_user`.

Kernel Vulnerabilities Happen

Kernel code is just code!

Memory corruptions, allocator misuse, etc, all happen in the kernel!

What can you do with this?

Kernel Race Conditions

Kernel modules are not userspace programs.

- they are always prone to multi-threading
  - what happens if two devices open /dev/pwn-college simultaneously?
- they could disappear or swap resources mid-execution
  - what happens if make_root.ko is removed while /proc/pwn-college is open?

Race conditions are huge problems plaguing kernels!

The Classic: Privilege Escalation

The kernel tracks user the privileges (and other data) of every running process.

```
struct task_struct {
    struct thread_info          thread_info;

    /* -1 unrunnable, 0 runnable, >0 stopped: */
    volatile long               state;

    void                        *stack;
    atomic_t                    usage;
                // ...
    int                         prio;
    int                         static_prio;
    int                         normal_prio;
    unsigned int                rt_priority;

    struct sched_info           sched_info;

    struct list_head            tasks;

    pid_t                       pid;
    pid_t                       tgid;

    /* Process credentials: */

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

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

```
struct cred {
    atomic_t    usage;
    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 */
                // ...
};
```

How do we set these?

The credentials are supposed to be immutable (i.e., they can be cached elsewhere, and shouldn't be updated in place). Instead, they can be replaced:

```
commit_creds(struct cred *)
```

The cred struct seems a bit complex, but the kernel can make us a fresh one!

```
struct cred * prepare_kernel_cred(struct task_struct
                    *reference_task_struct)
```

Luckily, if we pass NULL to the reference struct, it'll give us a cred struct with root access and full privileges!

How do we set these?

We have to run:

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

Complications

How do we know where `commit_creds` and `prepare_kernel_cred` are in memory?

1. Older kernels (or newer kernels when kASLR is disabled) are mapped at predictable locations.
2. `/proc/kallsym` is an interface for the kernel to give root these addresses.
3. If enabled, gdb support is your friend.
4. Otherwise, it's the exact same problem as userspace ASLR!