System Security
Rootkit, countermeasures

Samuel Thibault <samuel.thibault@u-bordeaux.fr>
https://dept-info.labri.fr/~thibault/enseignements#SecuSys

- last slide of cours1

rootkits

Today, we will write a rootkit

- Installs some code into privileged area
- Contains a backdoor
- Hides itself from discovery from the admin

rootkits

Basic principle

- **`sys_call_table`** is the table of the implementations of the system calls
  - Indexed by system call number
  - Mere function pointers to implementations
- Replace entry in **`sys_call_table`** with our own implementation
  - Usually calls the *real* implementation before/after doing its stuff

Thus **redirecting** a system call

rootkits

First difficulty: addresses

- KASLR!
- Finding the address of **`sys_call_table`**
  - Not trivially exposed to modules
- Finding the address of the real implementation
  - Some times trivially exposed, most often not

« Trivially » solved thanks to /proc/kallsyms

- root-only
- But could be obtained on another system

rootkits

Then also RO-data, we will see that later

rootkits

- Visibility in **`lsmod`**
- Visibility in **`/lib/modules/.../`**
- Visibility in auto-load configuration

Hide them: redirect more system calls (read, getdents, ...)

Or load them further away

- System BIOS
- Blue pill
  - virtualization
- Device firmware

Kernel countermeasures

Protect memory: make most of the kernel R/O

- Code, obviously
- Methods structures: **`const`**
  - file_operations, proto_ops, ...
- Anything that only needs to be written at initialization
  - e.g. machine description
  - post-init read-only memory
  - **`__read_only`** qualifier

We'll however see that on x86 it's trivial to bypass this :/

Kernel countermeasures

KASLR

- Just like ASLR, but for the kernel code

Then we want to hide kernel addresses

- Don’t print pointers in dmesg
  - printk("%p") format is hashed
- Don’t show addresses
  - /proc/kallsyms hides addresses

Kernel countermeasures

Structure layout randomization

- Order structure members randomly!
- Performed by the compiler
  - Thus only "random" per build
  - Attacker "just" needs to use the same build

```
struct t {
    int x;
    int y;
}
```

```
struct t {
    int y;
    int x;
}
```

Kernel countermeasures

Just disabling module loading?

- Would prevent « plug-and-play »... :/

At least prevent from loading unsigned modules

Blacklist modules for elder protocols/hardware

- Even if signed

Other kinds of loads: BPF filters

- E.g. for **`tcpdump`**, but also used for various subsystems
- Actually a virtual machine
- Security-sensitive
- Limited action, limited amount of computation

Linux Kernel Lockdown

Prevents loading unsigned modules

Prevent various userland access to system

- I/O ports
- MSR
- ACPI
- ...

Filtering system calls : sandboxing

seccomp

- One-way switch to being able only to
  - `exit()`
  - `sigreturn()`
  - `read()`
  - `write()`
- Quite extreme :)

seccomp-bpf

- BPF filter to decide which system call is allowed
- E.g. ssh now uses it