

1. kernel pwn

附件

- boot.sh
 - 启动
 - 观看保护机制
- bzImage
 - kernel镜像 linux kernel boot executable bzImage
 - 压缩过的linux内核文件
- rootfs.cpio
 - linux内核文件系统压缩包
 - 解压缩获得重新压缩，从而修改文件
 - init.d -> 系统如何启动
- vmlinux
 - 未压缩的内核文件
 - 使用 `extract-vmlinux bzImage > vmlinux`

调试

- qemu启动
 - 修改boot.sh 最后加 `-gdb tcp::1234`
 - `-s` 默认参数1234
- 注意，比较慢，一条指令可能使用 `几秒钟才能执行成功`
- `gdb -q -ex "set arch..." "target remote localhost:1234"`
- or

```
1 pwngdb> target remote localhost:1234
```

- 寻找gadget M量级

```
1 ROPgadget --binary ./vmlinux > gadget.txt
2 ropper --file ./vmlinux --nocolor > gadget.txt # 多线程，更快
```

- vmlinux 可以通过 bzImage 获取

操作系统部分知识

内核

操作系统（Operation System）本质上也是一种软件，可以看作是普通应用程式与硬件之间的一层中间层，其主要作用便是调度系统资源、控制IO设备、操作网络与文件系统等，并为上层应用提供便捷、抽象的应用接口

而运行在内核态的**内核（kernel）**则是一个操作系统最为核心的部分，提供着一个操作系统最为基础的功能

kernel的主要功能可以归为以下三点：

- 控制并与硬件进行交互
- 提供应用程式运行环境
- 调度系统资源

包括 I/O，权限控制，系统调用，进程管理，内存管理等多项功能都可以归结到以上三点中

与一般的应用程式不同，kernel的crash通常会引起重启

分级保护域

分级保护域（hierarchical protection domains）又被称作保护环，简称 Rings，是一种将计算机不同的资源划分至不同权限的模型

在一些硬件或者微代码级别上提供不同特权态模式的 CPU 架构上，保护环通常都是硬件强制的。

Rings是从最高特权级（通常被叫作0级）到最低特权级（通常对应最大的数字）排列的

在大多数操作系统中，Ring0 拥有最高特权，并且可以和最多的硬件直接交互（比如CPU，内存）

内层ring可以任意调用外层ring的资源

Intel Ring Model

Intel的CPU将权限分为四个等级：**Ring0**、Ring1、Ring2、**Ring3**，权限等级依次降低

大部分现代操作系统只用到了ring0 和 ring3，其中 kernel 运行在 ring0，用户态程序运行在 ring3

💡 使用 Ring Model 是为了提升系统安全性，例如某个间谍软件作为一个在 Ring 3 运行的用户程序，在不通知用户的时候打开摄像头会被阻止，因为访问硬件需要使用 being 驱动程序保留的 Ring 1 的方法

用户空间 & 内核空间

在现代操作系统中，计算机的虚拟内存空间通常被分为两块空间——供用户进程使用的用户空间（user space）与供操作系统内核使用的内核空间（kernel space）

内核态 —> 用户态

由内核态重新“着陆”回用户态只需要恢复用户空间信息即可：

- `swapgs` 指令恢复用户态GS寄存器
- `sysretq` 或者 `iretq` 恢复到用户空间

IO

💡 万物皆文件的哲学

系统调用：`ioctl`

在 *NIX 中一切都可以被视为文件，因而一切都可以以访问文件的方式进行操作，为了方便，Linux 定义了系统调用 `ioctl` 供进程与设备之间进行通信

LKMs

- 可装载内核模块

通常与LKM相关的命令有以下三个：

- LKM
 - loadable kernel mode, 动态可加载内核模块
 - insmod insert
 - rmmod remove
 - lsmod list

进程描述符

- `task_struct` 结构体

```
1  /* Process credentials: */
2
3  /* Tracer's credentials at attach: */
4  const struct cred __rcu *ptracer_cred;
5
6  /* Objective and real subjective task credentials (COW): */
7  const struct cred __rcu *real_cred;
8
9  /* Effective (overridable) subjective task credentials (COW): */
10 const struct cred __rcu *cred;
```

slab allocator

slab allocator 则是更为细粒度的内存管理器，其通过向 buddy system 请求单张或多张连续内存页后再分割成同等大小的**对象**（object）返还给上层调用者来实现更为细粒度的内存管理

编译内核->结果

Image

- 下载源码 - linux kernel 开源
- `make menuconfig`
- `make bzImage -j4` 加速编译

vmliunx

- 编译出的原始内核文件

bzImage

- 压缩内核文件
- `file` 命令查看文件信息

内核内存分配

- kmalloc，其实现是使用的slab/slub分配器，现在多见的是slub分配器。这个分配器通过一个多级的结构进行管理。

漏洞缓冲机制

文件系统

proc/kallsyms

- 操作系统的符号表，提供了 所有的函数地址信息

sys/module

- 包含模块所有的编译信息
 - 虚拟内存的信息
 - text,bss.data...

dev

- 文件会注册，可以使用 IO函数 操作

semprmap

- CR4寄存器,可以绕过

```
1 mov $cr4, 0x6f0
```

提权

- commit_creds()
- cred结构体，修改uid,gid
- modprobe_path结构体改变
- task_struct劫持 + prctl

常见函数

ioctl

- 系统调用 ioctl 是一个专用于设备输入输出操作的一个系统调用，其调用方式如下：
-

```
1 int ioctl(int fd, unsigned long request, ...)
```

ioctl 是设备驱动程序中对设备的 I/O 通道进行管理的函数

- 所谓对 I/O 通道进行管理，就是对设备的一些特性进行控制，例如串口的传输波特率、马达的转速等等。它的调用个数如下： `int ioctl(int fd, int cmd, ...)`;
- 其中 fd 是用户程序打开设备时使用 open 函数返回的文件标示符，cmd 是用户程序对设备的控制命令，至于后面的省略号，那是一些补充参数，一般最多一个，这个参数的有无和 cmd 的意义相关
- ioctl 函数是文件结构中的一个属性分量，就是说如果你的驱动程序提供了对 ioctl 的支持，用户就可以在用户程序中使用 ioctl 函数来控制设备的 I/O 通道。
- 意思就是说**如果一个 LKM 中提供了 ioctl 功能，并且实现了对应指令的操作，那么在用户态中，通过这个驱动程序，我们可以调用 ioctl 来直接调用模块中的操作**

others

1. printk(): 内核态的打印函数，但是输出不是终端，而是内核缓冲区，可以通过dmesg查看。
2. copy_from_user()/copy_to_user(): 内核态和用户态数据传输的函数，前者实现了用户态数据向内核态的传输，后者实现了内核态数据向用户态传输。
3. kmalloc(): 内核态的内存分配函数，使用的是slab/slub分配器。
4. commit_creds(struct cred* new): 将新的cred结构应用于当前进程；
5. struct cred* prepare_kernel_cred(struct task_struct* daemon): 创建一个新的cred结构。

通常来说如果可以在内核中执行commit_creds(prepare_kernel_cred(0))，就可以设置当前进程的uid和gid为0，实现本地提权。

kernel 实验

- 必须包含的头文件

```
1 #include <linux/module.h>
2 #include <linux/kernel.h>
3 #include <linux/init.h>
```

- 入口点，出口点

```
1 module_init()  
2 module_exit()
```

- 系统调用接口函数
- 回调函数

编译一个linux内核

- 前提

```
1 sudo apt-get install git fakeroot build-essential ncurses-dev xz-utils qemu flex
```

- 下载内核

```
1 $ wget https://cdn.kernel.org/pub/linux/kernel/v6.x/linux-6.1.23.tar.xz  
2  
3 # 解压缩  
4 tar -xvf linux-6.1.23.tar.xz
```

- 编译

```
1 make menuconfig
```

- 选项
 - Kernel hacking —> Kernel debugging
 - Kernel hacking —> Compile-time checks and compiler options —> Compile the kernel with debug info
 - Kernel hacking —> Generic Kernel Debugging Instruments —> KGDB: kernel debugger
 - kernel hacking —> Compile the kernel with frame pointers

- 编译

- 实际安装必须以root身份执行，但任何正常构建都不需要

```
1 sudo make -j16
```

- 报错1

```
1 make[2]: *** No rule to make target 'debian/canonical-certs.pem', needed by 'cer
2 # 只需要在 .config 文件中找到 CONFIG_SYSTEM_TRUSTED_KEYS, 等于号后面的值改为 ""
```

- 报错2

```
1 make[1]: *** No rule to make target 'debian/canonical-revoked-certs.pem', needed
2 # 这时需要在 .config 文件中将 CONFIG_SYSTEM_REVOCATION_KEYS 项等号后面的值改为 ""
```

- 报错3

```
1 BTF: .tmp_vmlinux.btf: pahole (pahole) is not available
2 Failed to generate BTF for vmlinux
3 Try to disable CONFIG_DEBUG_INFO_BTF
4 make: *** [Makefile:1227: vmlinux] Error 1
5
6 # 安装
7 $ sudo apt install dwarves
```

- 报错4

```
1 /bin/sh: 1: zstd: not found
2 make[2]: *** [arch/x86/boot/compressed/Makefile:143: arch/x86/boot/compressed/vm
3 make[2]: *** Deleting file 'arch/x86/boot/compressed/vmlinux.bin.zst'
4 make[2]: *** Waiting for unfinished jobs....
```

```
5 make[1]: *** [arch/x86/boot/Makefile:115: arch/x86/boot/compressed/vmlinux] Erro
6 make: *** [arch/x86/Makefile:257: bzImage] Error 2
7
8
9 # 安装
10 sudo apt install zstd
```


- 6.x 版的报错和5.x的差不多

- rust 环境？

- 完成

```
1 $ file arch/x86/boot/bzImage
2 arch/x86/boot/bzImage: Linux kernel x86 boot executable bzImage, version 5.15.10
3
4 $ file arch/x86/boot/bzImage
5 arch/x86/boot/bzImage: Linux kernel x86 boot executable bzImage, version 6.1.23
```

busybox

 BusyBox 是一个集成了三百多个最常用Linux命令和工具的软件，包含了例如ls、cat和echo等一些简单的工具，我们将用 busybox 为我们的内核提供一个基本的用户环境

- 官网下载

```
1 $ wget https://busybox.net/downloads/busybox-1.35.0.tar.bz2
2 tar -jxvf busybox-1.35.0.tar.bz2
```

- 配置

```
1 make menuconfig
```

勾选 勾选 Settings → Build static binary file (no shared lib)

- 安装

```
1 make install
2 # 编译完成后会生成一个_install目录，接下来我们将会用它来构建我们的磁盘镜像
```

busybox init

- 目录结构

```
1 $ cd _install
2 $ mkdir -pv {bin,sbin,etc,proc,sys,home,lib64,lib/x86_64-linux-gnu,usr/{bin,sbin}
3 $ touch etc/inittab
4 $ mkdir etc/init.d
5 $ touch etc/init.d/rcS
6 $ chmod +x ./etc/init.d/rcS
```

- 初始化 etc/inittab

```
1 ::sysinit:/etc/init.d/rcS
2 ::askfirst:/bin/ash
3 ::ctrlaltdel:/sbin/reboot
4 ::shutdown:/sbin/swapoff -a
5 ::shutdown:/bin/umount -a -r
6 ::restart:/sbin/init
```

- etc/init.d/rcS

```
1 #!/bin/sh
2 mount -t proc none /proc
3 mount -t sysfs none /sys
4 mount -t devtmpfs devtmpfs /dev
5 mount -t tmpfs tmpfs /tmp
6 mkdir /dev/pts
7 mount -t devpts devpts /dev/pts
8
9 echo -e "\nBoot took $(cut -d' ' -f1 /proc/uptime) seconds\n"
10 setsid cttyhack setuidgid 1000 sh
11
12 poweroff -d 0 -f
```

- 可以使用根目录下的 `init` 脚本代替

```
1 #!/bin/sh
2
3 mount -t proc none /proc
4 mount -t sysfs none /sys
5 mount -t devtmpfs devtmpfs /dev
6
7 exec 0</dev/console
8 exec 1>/dev/console
9 exec 2>/dev/console
10
11 echo -e "\nBoot took $(cut -d' ' -f1 /proc/uptime) seconds\n"
12 setsid cttyhack setuidgid 1000 sh
13
14 umount /proc
15 umount /sys
16 poweroff -d 0 -f
```

- 配置用户组

```
1 $ echo "root:x:0:0:root:/root:/bin/sh" > etc/passwd
2 $ echo "ctf:x:1000:1000:ctf:/home/ctf:/bin/sh" >> etc/passwd
3 $ echo "root:x:0:" > etc/group
4 $ echo "ctf:x:1000:" >> etc/group
5 $ echo "none /dev/pts devpts gid=5,mode=620 0 0" > etc/fstab
```

打包

- cpio

```
1 find . | cpio -o --format=newc > ../../rootfs.cpio
2
3 # 解包
4 cpio -idv < ./rootfs.cpio # 当前目录
```

- ext4

```
1 dd if=/dev/zero of=rootfs.img bs=1M count=32
2
3 mkfs.ext4 rootfs.img
4
5 # 挂载镜像
6 $ mkdir tmp
7 $ sudo mount rootfs.img ./tmp/
8 $ sudo cp -rpf _install/* ./tmp/
9 $ sudo umount ./tmp
```

qemu

安装

- 下载

```
1 apt 下载
```

- 启动

```
1 #!/bin/sh
2 qemu-system-x86_64 \
3     -m 1024M \
4     -kernel ./bzImage \
5     -initrd ./rootfs.cpio \
6     -monitor /dev/null \
7     -append "root=/dev/ram rdinit=/sbin/init console=ttyS0 oops=panic panic=1 lo
8     -cpu kvm64,+smep \
9     -smp cores=2,threads=1 \
10    -nographic \
11    -s
```

- 可以

```
Boot took 2.05 seconds
/ $ ls
bin      etc      home     lib      linuxrc  root     sys
dev      gen.sh   init     lib64    proc     sbin     usr
/ $ whoami
ctf
/ $ uname -a
Linux (none) 6.1.23 #2 SMP PREEMPT_DYNAMIC Sun Apr  9 16:01:27 CST 2023 x86_64 GNU/Linux
/ $ █
```

- 权限不行

```

/ $ ls -al
total 12
drwxrwxr-x 14 ctf ctf 0 Apr 9 10:28 .
drwxrwxr-x 14 ctf ctf 0 Apr 9 10:28 ..
-rw----- 1 ctf ctf 65 Apr 9 10:29 .ash_history
drwxrwxr-x 2 ctf ctf 0 Apr 9 10:24 bin
drwxr-xr-x 9 root root 2980 Apr 9 10:28 dev
drwxrwxr-x 3 ctf ctf 0 Apr 9 10:24 etc
-rwxrwxr-x 1 ctf ctf 48 Apr 9 10:23 gen.sh
drwxrwxr-x 2 ctf ctf 0 Apr 9 10:24 home
-rwxrwxr-x 1 ctf ctf 296 Apr 9 10:28 init
drwxrwxr-x 3 ctf ctf 0 Apr 9 10:24 lib
drwxrwxr-x 2 ctf ctf 0 Apr 9 10:24 lib64
lrwxrwxrwx 1 ctf ctf 11 Apr 9 10:24 linuxrc -> bin/busybox
dr-xr-xr-x 130 root root 0 Apr 9 10:28 proc
drwx----- 2 root root 0 Apr 9 07:13 root
drwxrwxr-x 2 ctf ctf 0 Apr 9 10:24 sbin
dr-xr-xr-x 13 root root 0 Apr 9 10:28 sys
drwxrwxrwt 2 root root 40 Apr 9 10:28 tmp
drwxrwxr-x 4 ctf ctf 0 Apr 9 10:24 usr

```

- 报错

LKMs

gdb调试

```

1 gdb vmlinux
2 set architecture <arch>
3 target remote localhost:1234
4 lsmod
5 cat /sys/module/xxx/sections/.text
6 add-symbol-file xxx.ko <text_addr> # 默认 text 段
7 b __init_begin
8 c

```

- 参考文章

补充

- 本机内核
 - `/boot/` 下
- 替换本机内核

pwn题目部署

- 和常规的CTF题目的布置方法是相类似的，最常见的办法便是使用 `ctf_xinted` + `docker` 布置，我们只需要配置用 `ctf_xinetd` 启动 `boot.sh` 即可

```
1 $ git clone https://github.com/Eadom/ctf_xinetd.git
```

PWN

- 写一个C语言 exp (去找一个模板)
 - 有时候为了减少其体积，可以使用 `musl-gcc`
 - 甚至可以使用汇编
- user -> root(ring0) -> user
 - 最后要保存并返回user态 ring3
- 打包与解包

```
1 find . | cpio -o --format=newc > ./rootfs.cpio
2 cpio -idmv < ./rootfs.cpio
```

保护

aslr

Stack Protector

- 相当于 canary
- 错误直接panic

reload

pie

MMAP_MIN_Addr

- 防御 null pointer
- 没有此保护
 - 指针为0，可以申请到0x000000处

KALLSYMS

- 符号表 `/proc/ksyscalls`
- 没有就去泄露

SMEP && SMAP

- smep
 - 如果处理器处于 ring0 模式，并试图执行有 user 数据的内存时，就会触发一个页错误。
 - 禁止 执行
- SMAP

- 禁止内核访问用户空间数据
- **SMAP**：位于Cr4的第21位，作用是让处于内核权限的CPU **无法读写** 用户代码

KASLR

- 内核地址空间布局随机化，并不默认开启，需要在内核命令行中添加指定指令。
- qemu 增加启动参数 -append "kaslr" 即可开启

FGKASLR

- 基于 KASLR 实现了 FGKASLR，**以函数粒度重新排布内核代码**

KPTI



1. 到了内核版本 `4.15`，**KPTI**（Kernel Page Table Isolation，内核页表隔离）这一巨大杀器出现了——内核与用户进程使用两套**独立的页表**
2. 4.15版本与其之后的几个版本的内核当中似乎在 `open("/dev/ptmx")` 时所分配的**第一个结构体都不是 `tty_struct`**，我们似乎不能够通过 `tty_struct` 来泄露内核基址与劫持内核执行流了，不过在内核当中**仍然有着数量相当可观的有用的结构体**供我们利用

- 内核页表隔离，内核空间和用户空间使用不同的页表集
- **使ret2usr成为过去式**
- 在启动项 `append` 中添加 `pti=on` 选项开启 KPTI