

# Linux分析与安全设计



# 5

## 第五章 Linux漏洞分类及实例



# 第五讲 Linux内核漏洞分类及实例

- 操作系统漏洞分类及原理
- 内核漏洞危害分类
- 内核hook技术
- 内核提权方法
- 内核漏洞防御



# 操作系统漏洞

- 用户态漏洞是指用户态程序漏洞，通常包括应用程序的漏洞，以及操作系统用户态模块的漏洞。
- 内核漏洞是指运行在RING0的程序所具有的能被利用的Bug或缺陷。
- 运行在Ring0上的操作系统内核、驱动共享同一个虚拟地址空间，可以完全访问系统空间的内存。而不像用户态进程那样拥有独立私有的内存空间。



# 操作系统漏洞的分类

- 内存破坏漏洞
- 未初始化的/未验证/已损坏的指针引用
- 整数误用
- 竞态条件
- 逻辑bug
- Cache侧信道攻击



# 内存破坏漏洞

➤ 栈漏洞

➤ 堆漏洞



# ELF文件格式

```
typedef struct elf32_hdr {  
    unsigned char e_ident[EI_NIDENT];  
    Elf32_Half e_type;  
    Elf32_Half e_machine;  
    Elf32_Word e_version;  
    Elf32_Addr e_entry;           /* Entry point */  
    Elf32_Off e_phoff;  
    Elf32_Off e_shoff;  
    Elf32_Word e_flags;  
    Elf32_Half e_ehsize;  
    Elf32_Half e_phentsize;  
    Elf32_Half e_phnum;  
    Elf32_Half e_shentsize;  
    Elf32_Half e_shnum;  
    Elf32_Half e_shstrndx; } Elf32_Ehdr;
```



# ELF文件格式

- **e\_ident[EI\_MAG0]**

被称为魔数 (Magic Number), 其值一般为0x7f 45 4c 46, 实际上就是. ELF.

- **e\_ident[EI\_CLASS]**: 目标文件运行的目标机器类别, 取值可为三种值: ELFCLASSNONE (0) 非法类别; ELFCLASS32 (1) 32位目标; ELFCLASS64 (2) 64位目标。

- **e\_ident[EI\_DATA]**: 特定处理器的数据编码字节顺序。即大端还是小端方式。取值可为3种: ELFDATANONE (0) 非法数据编码; ELFDATA2LSB (1) 高位在前; ELFDATA2MSB (2) 低位在前。

- **e\_ident[OS/ABI]**: OS和应用程序二进制接口, 包括UNIX - Linux、UNIX - System V、UNIX - GNU三种类型





# ELF文件格式

- **e\_type**表示elf文件的类型，如下主要定义：

名称	取值	含义
ET_NONE	0	未知目标文件格式
ET_REL	1	可重定位文件
ET_EXEC	2	可执行文件
ET_DYN	3	共享目标文件
ET_CORE	4	Core 文件 (转储格式)



# ELF文件格式

- `e_machine`: 表示目标体系结构类型
- `e_version`: ELF版本信息, 0为非法版本, 1为当前版本
- `e_entry`: 表示程序入口虚拟地址
- `e_ehsize`: ELF Header结构大小
- `e_phoff`、`e_phentsize`、`e_phnum`: 描述Program Header Table的偏移地址、大小、数量。
- `e_shoff`、`e_shentsize`、`e_shnum`: 描述Section Header Table的偏移地址、大小、数量
- `e_shstrndx`: 字符串表在Section Header Table中的索引



# ELF文件格式

```
jwang@ubuntu:~$ readelf -h hello.o
```

```
ELF Header:
```

```
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                               ELF32
  Data:                               2's complement, little endian
  Version:                             1 (current)
  OS/ABI:                             UNIX - System V
  ABI Version:                         0
  Type:                                EXEC (Executable file)
  Machine:                             Intel 80386
  Version:                             0x1
  Entry point address:                 0x8048320
  Start of program headers:            52 (bytes into file)
  Start of section headers:           4412 (bytes into file)
  Flags:                               0x0
  Size of this header:                 52 (bytes)
  Size of program headers:             32 (bytes)
  Number of program headers:           9
  Size of section headers:            40 (bytes)
  Number of section headers:          30
  Section header string table index: 27
```



# ELF文件格式

链接视图	执行视图
ELF 头部	ELF 头部
程序头部表（可选）	程序头部表
节区 1	段 1
...	
节区 n	段 2
...	
...	...
节区头部表	节区头部表（可选）



# ELF文件格式

主要的sections:

“`.text`” 段包含程序的可执行指令。

“`.bss`” 段含有占据程序内存映像的未初始化数据。

“`.data.`” 和 “`.data1`” 段包含占据内存映像的初始化数据。

“`.rodata`” 和 “`.rodata1`” 段含程序映像中的只读数据。

“`.shstrtab`” 段含有每个section的名字，由section入口结构中的sh\_name索引。

“`.strtab`” 段含有表示符号表(symbol table)名字的字符串。



# ELF文件格式

```
jwang@ubuntu:~$ readelf --segments hello.o
```

Elf file type is EXEC (Executable file)

Entry point 0x8048320

There are 9 program headers, starting at offset 52

Program Headers:

Type	Offset	VirtAddr	PhysAddr	FileSiz	MemSiz	Flg	Align
PHDR	0x000034	0x08048034	0x08048034	0x00120	0x00120	R E	0x4
INTERP	0x000154	0x08048154	0x08048154	0x00013	0x00013	R	0x1
[Requesting program interpreter: /lib/ld-linux.so.2]							
LOAD	0x000000	0x08048000	0x08048000	0x005c4	0x005c4	R E	0x1000
LOAD	0x000f14	0x08049f14	0x08049f14	0x00100	0x00108	RW	0x1000
DYNAMIC	0x000f28	0x08049f28	0x08049f28	0x000c8	0x000c8	RW	0x4
NOTE	0x000168	0x08048168	0x08048168	0x00044	0x00044	R	0x4
GNU_EH_FRAME	0x0004cc	0x080484cc	0x080484cc	0x00034	0x00034	R	0x4
GNU_STACK	0x000000	0x00000000	0x00000000	0x00000	0x00000	RW	0x4
GNU_RELRO	0x000f14	0x08049f14	0x08049f14	0x000ec	0x000ec	R	0x1

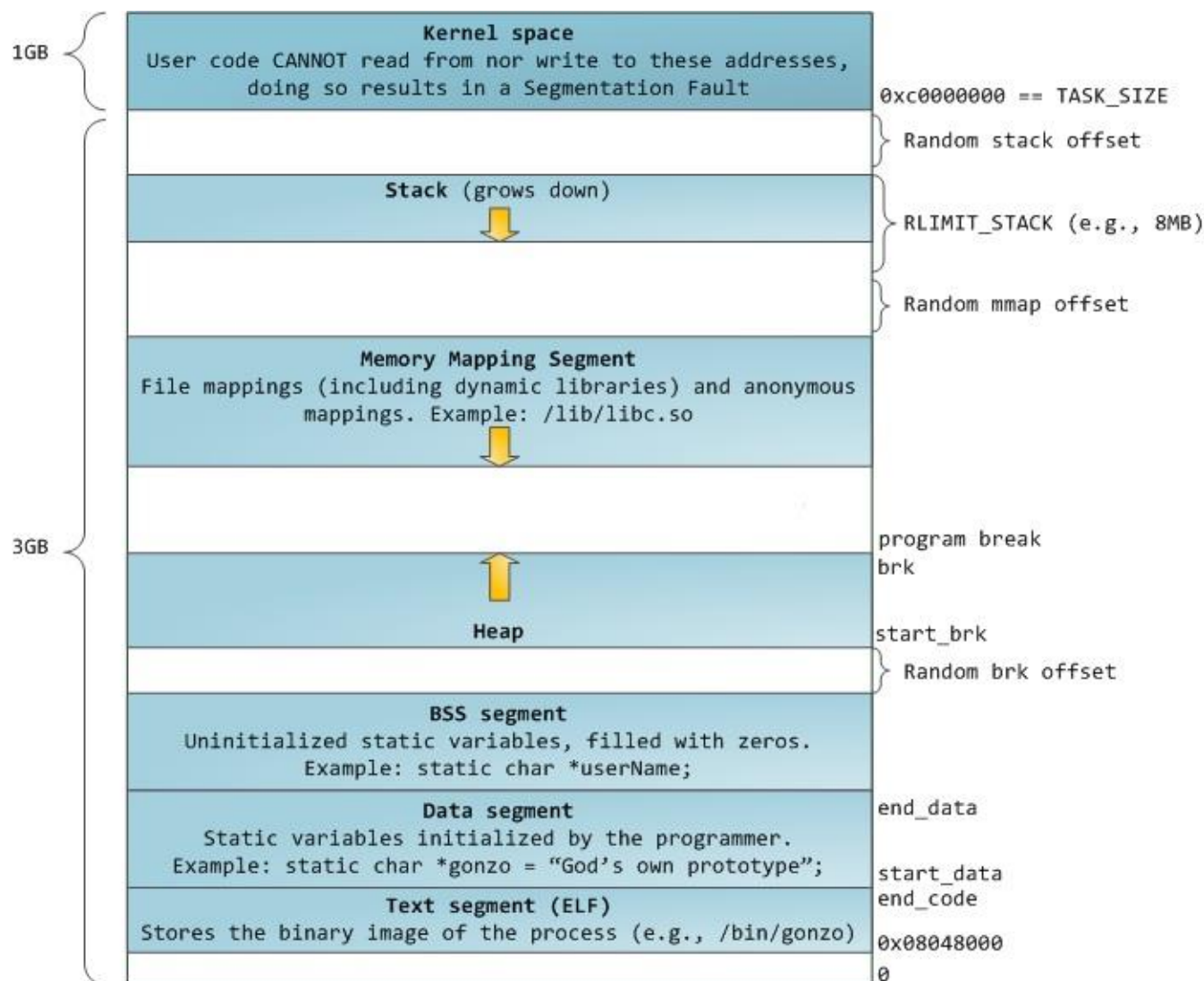
Section to Segment mapping:

Segment Sections...

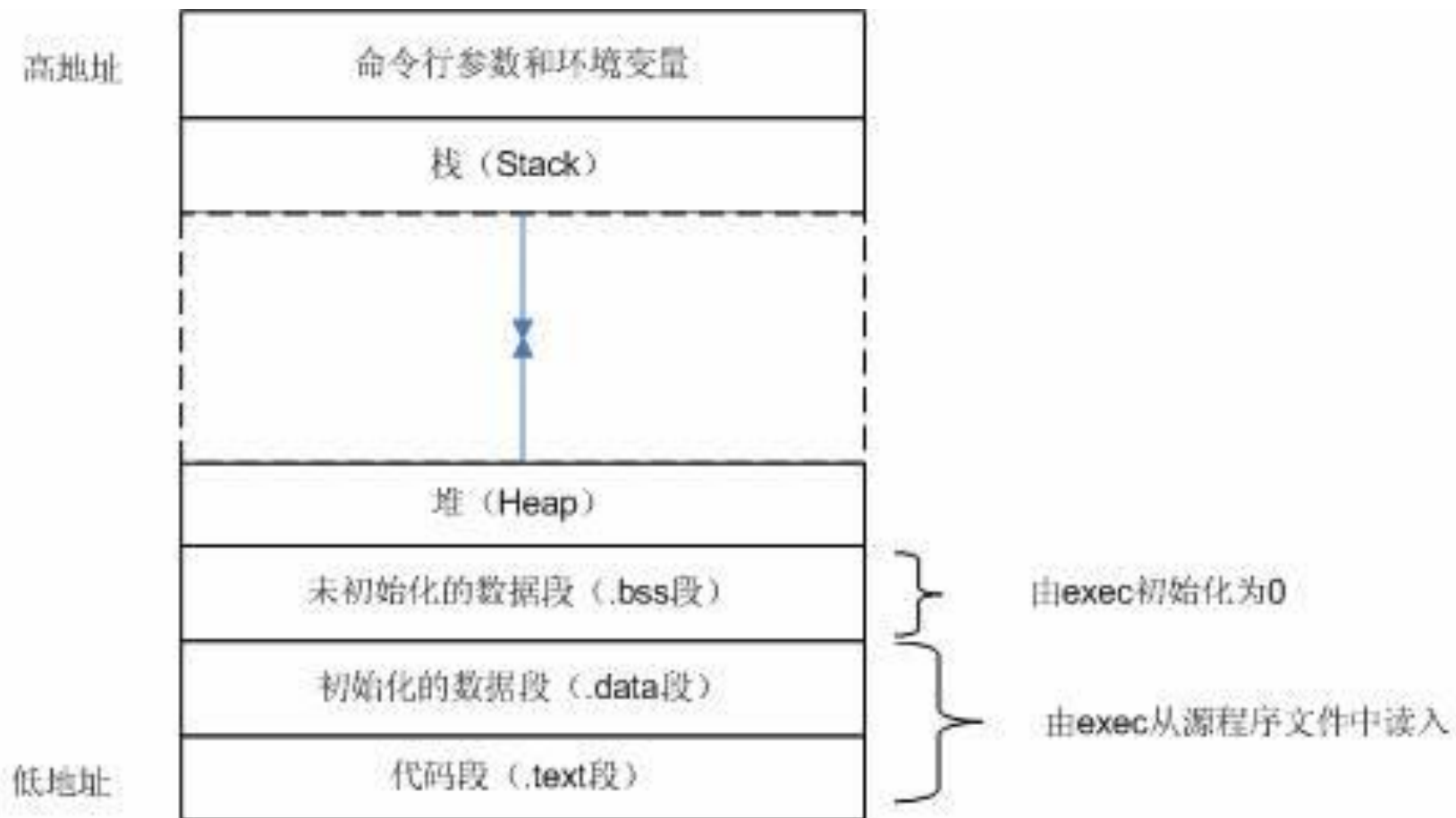




# LINUX程序内存分配原理



# LINUX程序内存分配原理





# LINUX程序内存分配原理

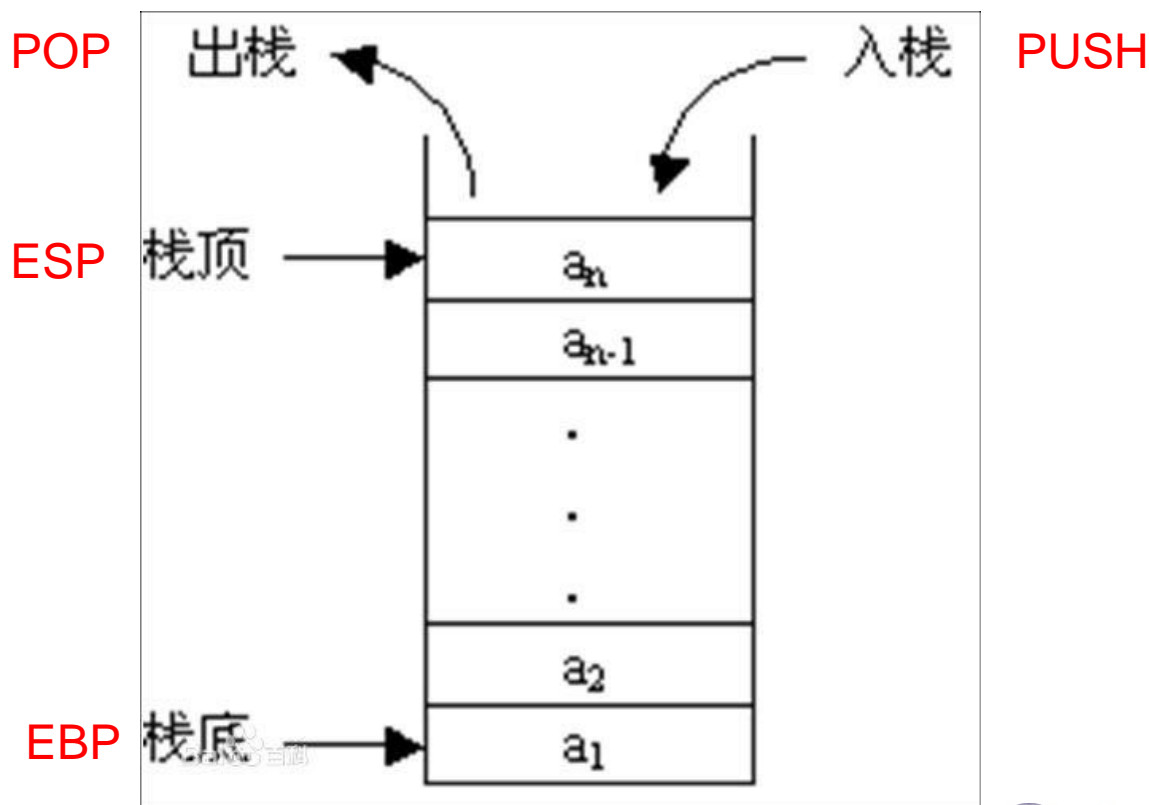
通过readelf 和cgdb查看程序的内存分配:

```
Section Headers:
```

[Nr]	Name	Type	Addr	Off	Size	ES	Flg	Lk	Inf	A
[ 0]		NULL	00000000	000000	000000	00		0	0	
[ 1]	.interp	PROGBITS	08048154	000154	000013	00	A	0	0	
[ 2]	.note.ABI-tag	NOTE	08048168	000168	000020	00	A	0	0	
[ 3]	.note.gnu.build-id	NOTE	08048188	000188	000024	00	A	0	0	
[ 4]	.gnu.hash	GNU_HASH	080481ac	0001ac	000020	04	A	5	0	
[ 5]	.dynsym	DYNSYM	080481cc	0001cc	000050	10	A	6	1	
[ 6]	.dynstr	STRTAB	0804821c	00021c	00004a	00	A	0	0	
[ 7]	.gnu.version	VERSYM	08048266	000266	00000a	02	A	5	0	
[ 8]	.gnu.version_r	VERNEED	08048270	000270	000020	00	A	6	1	
[ 9]	.rel.dyn	REL	08048290	000290	000008	08	A	5	0	
[10]	.rel.plt	REL	08048298	000298	000018	08	A	5	12	
[11]	.init	PROGBITS	080482b0	0002b0	00002e	00	AX	0	0	
[12]	.plt	PROGBITS	080482e0	0002e0	000040	04	AX	0	0	1
[13]	.text	PROGBITS	08048320	000320	00017c	00	AX	0	0	1
[14]	.fini	PROGBITS	0804849c	00049c	00001a	00	AX	0	0	
[15]	.rodata	PROGBITS	080484b8	0004b8	000014	00	A	0	0	
[16]	.eh_frame_hdr	PROGBITS	080484cc	0004cc	000034	00	A	0	0	
[17]	.eh_frame	PROGBITS	08048500	000500	0000c4	00	A	0	0	
[18]	.ctors	PROGBITS	08049f14	000f14	000008	00	WA	0	0	
[19]	.dtors	PROGBITS	08049f1c	000f1c	000008	00	WA	0	0	
[20]	.jcr	PROGBITS	08049f24	000f24	000004	00	WA	0	0	

# LINUX程序内存分配原理

- 栈(stack)
  - 一种先进后出的数据结构，在程序中用于存放函数的参数值，局部变量值。



# 内存破坏漏洞

## ➤ 栈漏洞

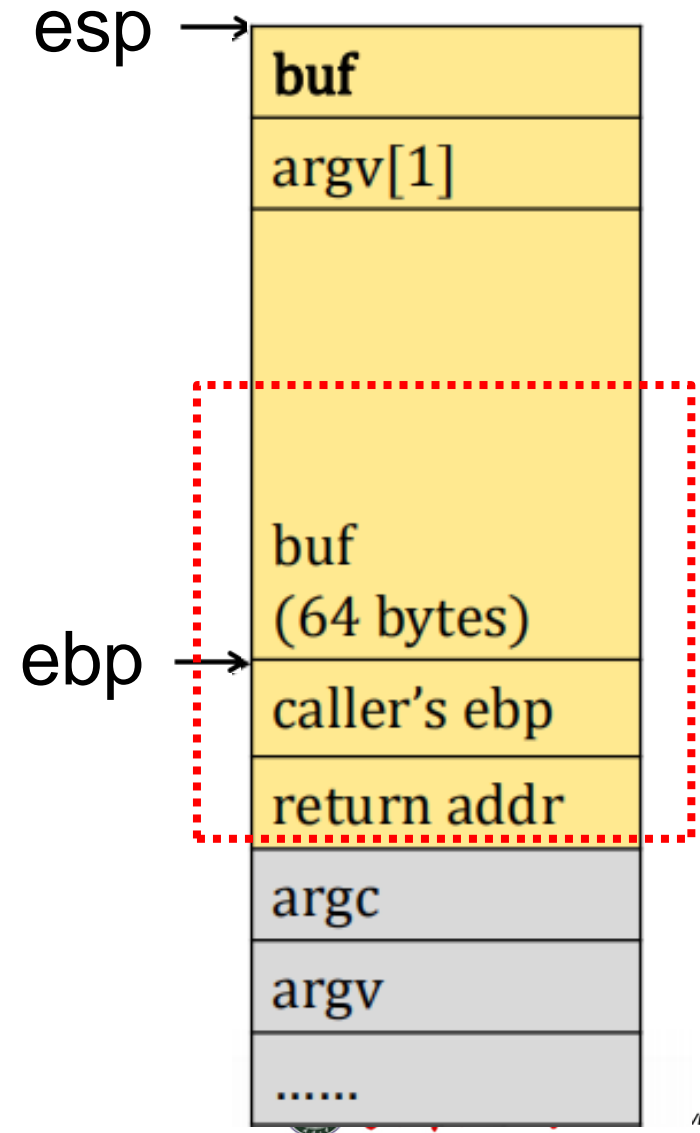
### ➤ 栈缓冲区溢出

- 使用不安全的C函数，如strcpy( )、sprintf( )、gets( )。
- 使用数组，array[]



# 栈缓冲区溢出

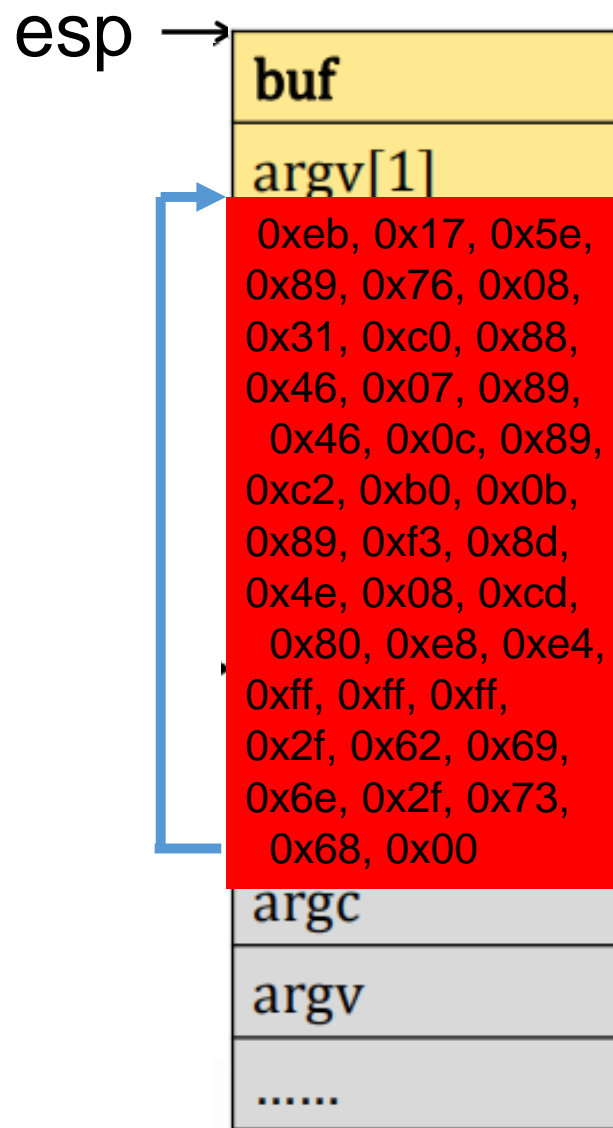
“计算机对接收的输入数据没有进行有效的检测,向缓冲区内填充数据时超过了缓冲区本身的容量,而导致数据溢出到被分配空间之外的内存空间,使得溢出的数据覆盖了其他内存空间的数据。”



# 栈缓冲区溢出覆盖返回地址

SHELL CODE:

```
0xeb, 0x17, 0x5e, 0x89, 0x76,  
0x08, 0x31, 0xc0, 0x88, 0x46,  
0x07, 0x89,  
0x46, 0x0c, 0x89, 0xc2, 0xb0,  
0x0b, 0x89, 0xf3, 0x8d, 0x4e,  
0x08, 0xcd,  
0x80, 0xe8, 0xe4, 0xff, 0xff,  
0xff, 0x2f, 0x62, 0x69, 0x6e,  
0x2f, 0x73,  
0x68, 0x00
```



# 缓冲区溢出防御机制-Stack Protector

- Canary/Stack Cookies
  - 阻止恶意代码淹没返回地址，跳转到shellcode
  - 方法：堆栈区增加canary(探测值)

栈中的守护天使

GCC编译选项：

- fstack-protector: 启用堆栈保护，不过只为局部变量中含有 char 数组的函数插入保护代码。
- fstack-protector-all: 启用堆栈保护，为所有函数插入保护代码。
- fno-stack-protector: 禁用堆栈保护。



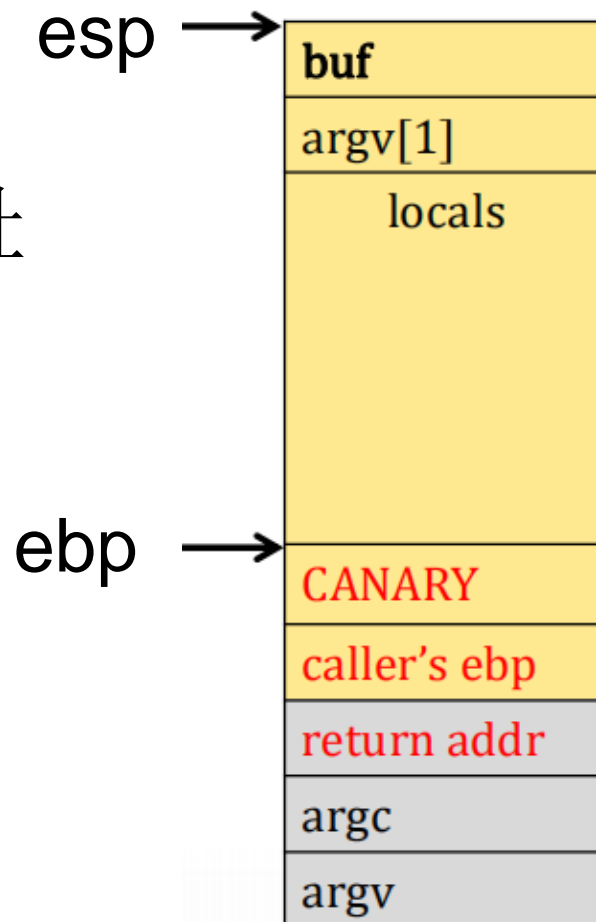
# 缓冲区溢出防御机制-Stack Protector

## ● 核心思想

- 将随机产生的canary (探测值) 插入到返回地址和局部变量之间

- 在函数返回之前检查canary

如果检测到错误的canary → 溢出



# 缓冲区溢出防御机制-Stack Protector

```
mov  %gs:0x14, %eax  
mov  %eax, -0xc(%ebp)
```

```
mov  -0xc(%ebp), %eax  
xor  %gs:0x14, %eax
```





# 绕过Stack Protector的方法

- 伪随机的猜测
- 获取真实canary的值，写入栈中，从而绕过栈保护检测
- 特殊情况下，可以劫持canary检测失效的处理函数，如stack\_check\_fail函数
- 覆盖TLS中存储的canary值



# 缓冲区溢出防御机制-NX (DEP)

- NX (No-eXecute, 不可执行)
  - 数据保护执行, 能够在内存上执行额外检查以帮助防止在系统上运行恶意代码。
- 基本原理是将数据所在内存页(默认的堆页、栈页以及内存池页)标识为不可执行, 当程序溢出成功转入shellcode时, 程序会尝试在数据页面上执行指令, 此时CPU就会抛出异常, 而不是去执行恶意指令。

# 缓冲区溢出防御机制-NX (DEP)

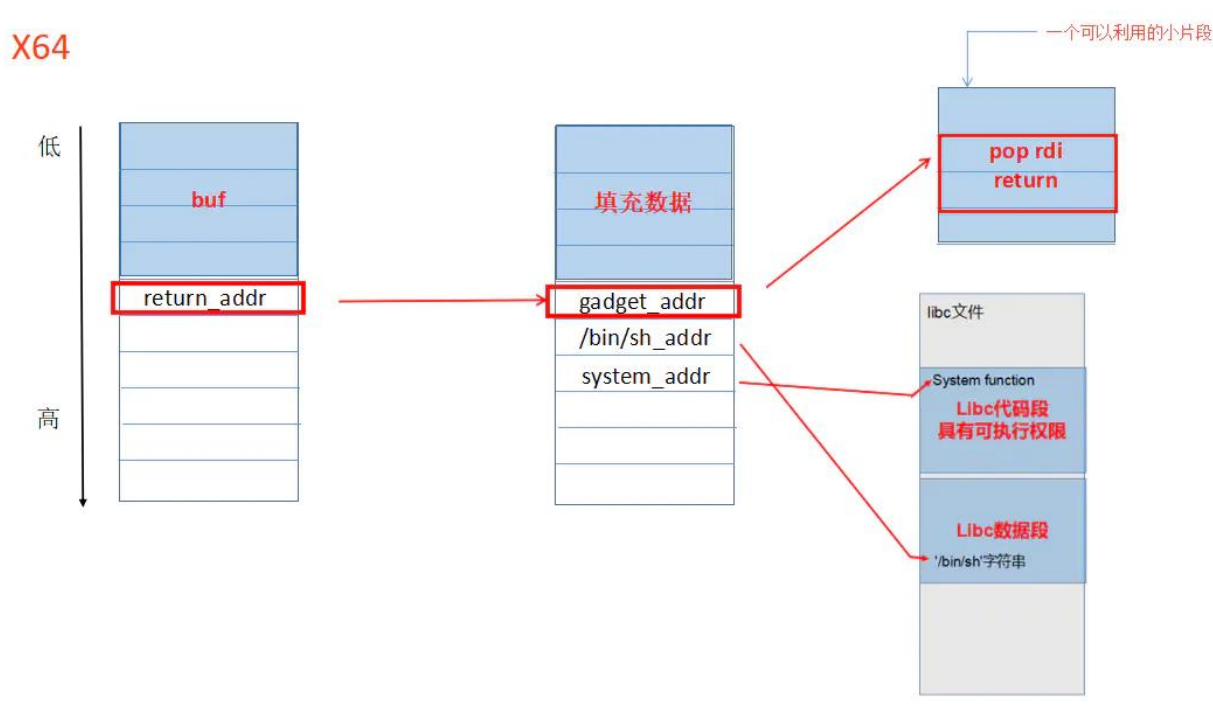
- 操作系统通过设置内存页的NX/XD属性标记，指明该内存不能执行代码。
- 在x86处理器的页表索引中，NX位置于63号的位置（以0作第一位），即64位中的最后一位，如果NX位元的数值是0（关闭），在页表内的指令集可正常执行，但如果是1（启动）的话则不能执行页表的指令集，并会把页表的一切皆当作数据。在格式上，页表需为PAE（物理地址扩展）格式，而非x86传统的格式。

# 缓冲区溢出防御机制-NX (DEP)

- 支持NX (DEP) 的系统：
  - Linux in 2004 (kernel 2.6.8) 以上
- 如果需要关闭NX选项，可以给gcc编译器添加-z execstack参数
  - gcc -o test test.c // 默认情况下，开启NX保护
  - gcc -z execstack -o test test.c // 禁用NX保护
  - gcc -z noexecstack -o test test.c // 开启NX保护

# 绕过NX (DEP) 的方法

- 关闭进程DEP
- Return-to-libc
- ROP (Return-Oriented Programming)



# 缓冲区溢出防御机制-ALSR

- ALSR(Address space layout randomization)
  - 通过对系统关键地址的随机化，防止攻击者在堆栈溢出后利用固定的地址定位到恶意代码并加以运行。
  - 在2005年被引入到Linux的内核 kernel 2.6.12 中。




# 缓冲区溢出防御机制-ALSR

- 六类地址进行随机化：
  - (1) 共享库随机化；
  - (2) 栈随机化；
  - (3) mmap() 随机化；
  - (4) VDSO (Virtual Dynamically-linked Shared Object) 随机化；
  - (5) 堆随机化；
  - (6) 内核地址空间布局随机化 (KASLR)



# 缓冲区溢出防御机制-ALSR

```
colin@colinsoft:~/ASLR$ ldd /bin/bash
linux-gate.so.1 => (0xb77cb000)
libtinfo.so.5 => /lib/i386-linux-gnu/libtinfo.so.5 (0xb778b000)
libdl.so.2 => /lib/i386-linux-gnu/libdl.so.2 (0xb7786000)
libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xb75cf000)
/lib/ld-linux.so.2 (0x80009000)
colin@colinsoft:~/ASLR$ ldd /bin/bash
linux-gate.so.1 => (0xb77e1000)
libtinfo.so.5 => /lib/i386-linux-gnu/libtinfo.so.5 (0xb77a1000)
libdl.so.2 => /lib/i386-linux-gnu/libdl.so.2 (0xb779c000)
libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xb75e5000)
/lib/ld-linux.so.2 (0x80062000)
```





# 缓冲区溢出防御机制-ALSR

- 绕过ALSR的方法：
  - 伪随机
  - 利用Heap Spray (堆喷射) 定位内存地址
  - 构造ROP链
  - GOT表劫持



# 缓冲区溢出防御机制-ALSR

- 开启和关闭ALSR:

- Linux (ubuntu)

- ```
echo 0 > /proc/sys/kernel/randomize_va_space
```

- Gdb set disable-randomization off



# 内存破坏漏洞

## ➤ 堆漏洞

### ➤ 堆

- 程序员自己分配的内存，如malloc（）函数



# 内存破坏漏洞

- 堆 (heap)

- 一般由程序员分配或释放。如调用 `malloc()` 函数

操作系统有一个记录空闲内存地址的链表，当系统收到程序的申请时，会遍历该链表，寻找第一个空间大于所申请空间的堆结点，然后将该结点从空闲结点链表中删除，并将该结点的空间分配给程序，另外，对于大多数系统，会在这块内存空间中的首地址处记录本次分配的大小，这样，代码中的 `delete` 语句才能正确的释放本内存空间。另外，由于找到的堆结点的大小不一定正好等于申请的大小，系统会自动的将多余的那部分重新放入空闲链表中。

# 内存破坏漏洞

## ➤ 堆漏洞

### ➤ 堆缓冲区溢出

- 类似栈缓冲区溢出

### ➤ 内存泄露

- 分配的内存没有释放



# 内存破坏漏洞

- 堆缓冲区溢出

- 程序向某个堆块中写入的字节数超过了堆块本身可使用的字节数，因而导致了数据溢出，并覆盖到物理相邻的高地址的下一个堆块。
- 堆溢出是一种特定的缓冲区溢出。但是其与栈溢出所不同的是，堆上并不存在返回地址等可以让攻击者直接控制执行流程的数据，因此我们一般无法直接通过堆溢出来控制 EIP 。



# 内存破坏漏洞

## ➤ 内存泄漏 (Memory Leak)

- 是指程序中已动态分配的堆内存由于某种原因程序未释放或无法释放，造成系统内存的浪费，导致程序运行速度减慢甚至系统崩溃等严重后果。
- 一般我们所说的内存泄漏指的是堆内存的泄露，堆内存是指程序从堆中分配的，大小随机的用完后必须显示释放的内存，C++/C中有free函数可以释放内存。



# 未初始化的/未验证/已损坏的指针引用

- Null指针引用
  - 空指针，不指向任何实际的对象或者函数
  - 空指针解引用是 C/C++ 程序中较为普遍的内存缺陷类型，当指针指向无效的内存地址并且对其引用时，有可能产生不可预见的错误，导致软件系统崩溃，拒绝服务。





# 未初始化的/未验证/已损坏的指针引用

- 例子 [CVE-2009-2692](#) (Linux kernel 2.6.x or 2.4.x)

```
static ssize_t sock_sendpage(struct file *file,...)
{
    struct socket *sock;
    int flags;
    sock = file->private_data;
    flags = !(file->f_flags & O_NONBLOCK) ? 0 : MSG_DONTWAIT;
    if (more)
        flags |= MSG_MORE;
    return sock->ops->sendpage(sock, page, offset, size,
        flags);
}
```



# 未初始化的/未验证/已损坏的指针引用

- Linux Kernel空指针间接引用本地拒绝服务漏洞 (CVE-2014-2678)
  - Linux kernel 3.14版本内，net/rds/iw.c 中的函数rds\_iw\_laddr\_check在实现上存在本地拒绝服务漏洞，本地用户通过盲系统调用没有RDS传输的系统上的RDS套接字，利用此漏洞可造成空指针间接引用和系统崩溃。



# 整数漏洞

## ➤ 整数

- 有符号整数和无符号整数
- 计算机语言中整数类型都有一个取值范围
- $N$ 位整数，则有 $2^N$ 取值范围



# 整数漏洞

## ➤ 整数溢出

- 两个整数进行运算时，若其结果大于最大值（**上溢**）或者小于最小值（**下溢**）就是溢出。



编号: 1758345 红动中国 (www.redocn.com) 妖精的尾巴2012



武汉大学

WUHAN UNIVERSITY

# 整数漏洞

## ➤ 整数溢出

| 编译器类型        | 数据类型             | 数据名称     | 最小值                        | 最大值                        |
|--------------|------------------|----------|----------------------------|----------------------------|
| VC++ 6.0     | unsigned short   | 无符号短整型   | 0                          | 65535                      |
|              | short            | 短整型      | -32,768                    | 32,767                     |
|              | unsigned int     | 无符号整型    | 0                          | 4,294,967,295              |
|              | int              | 整型       | -2,147,483,648             | 2,147,483,647              |
|              | unsigned long    | 无符号长整型   | 0                          | 4,294,967,295              |
|              | long             | 长整型      | -2,147,483,648             | 2,147,483,647              |
|              | unsigned __int64 | 无符号64位整数 | 0                          | 18,446,744,073,709,500,000 |
|              | __int64          | 64位整数    | -9,223,372,036,854,770,000 | 9,223,372,036,854,770,000  |
| VS2012<br>C# | ushort           | 无符号短整型   | 0                          | 65535                      |
|              | short            | 短整型      | -32,768                    | 32,767                     |
|              | uint             | 无符号整型    | 0                          | 4,294,967,295              |
|              | int              | 整型       | -2,147,483,648             | 2,147,483,647              |
|              | ulong            | 无符号长整型   | 0                          | 18,446,744,073,709,500,000 |
|              | long             | 长整型      | -9,223,372,036,854,770,000 | 9,223,372,036,854,770,000  |



# 整数漏洞

## ➤ 整数溢出

```
#include <stdio.h> int main(void) {  
    short i = 32767;  
    unsigned short j = 65535;  
    unsigned short k = 0; // 上溢例子  
    printf("%hd %hd %hd\n", i, i+1, i+2); // 上溢例子  
    printf("%hu %hu %hu\n", j, j+1, j+2); // 下溢例子  
    printf("%hu %hu %hu\n", k, k-1, k-2); return  
    0;  
}
```



# 整数漏洞

## ➤ 整数溢出

```
F:\AmuseOneself\StudyNotes\C-Language\Examples\55>gcc toobig.c -o toobig.exe  
F:\AmuseOneself\StudyNotes\C-Language\Examples\55>toobig.exe  
32767 -32768 -32767  
65535 0 1  
0 65535 65534  
F:\AmuseOneself\StudyNotes\C-Language\Examples\55>_
```



# 整数漏洞

## ➤ 整数溢出

```
char buffer[100];  
unsigned char offset = getOffset();  
if (40 + offset < sizeof(buffer))  
{  
    buffer[offset] = 0;  
}
```





# 整数漏洞

- CVE-2019-11477: Linux内核中TCP协议栈整数溢出漏洞，影响Linux内核2.6.29及以上版本。
- 该漏洞是一个位于skb\_buff结构体上tcp\_gso\_segs成员的整数溢出漏洞。

```
struct tcp_skb_cb {  
    __u32 seq; /* Starting sequence number*/  
    __u32 end_seq; /* SEQ + FIN + SYN +datalen  
*/  
    __u32 tcp_tw_isn;  
    struct {  
        u16 tcp_gso_segs;  
        u16 tcp_gso_size;  
    };  
};
```



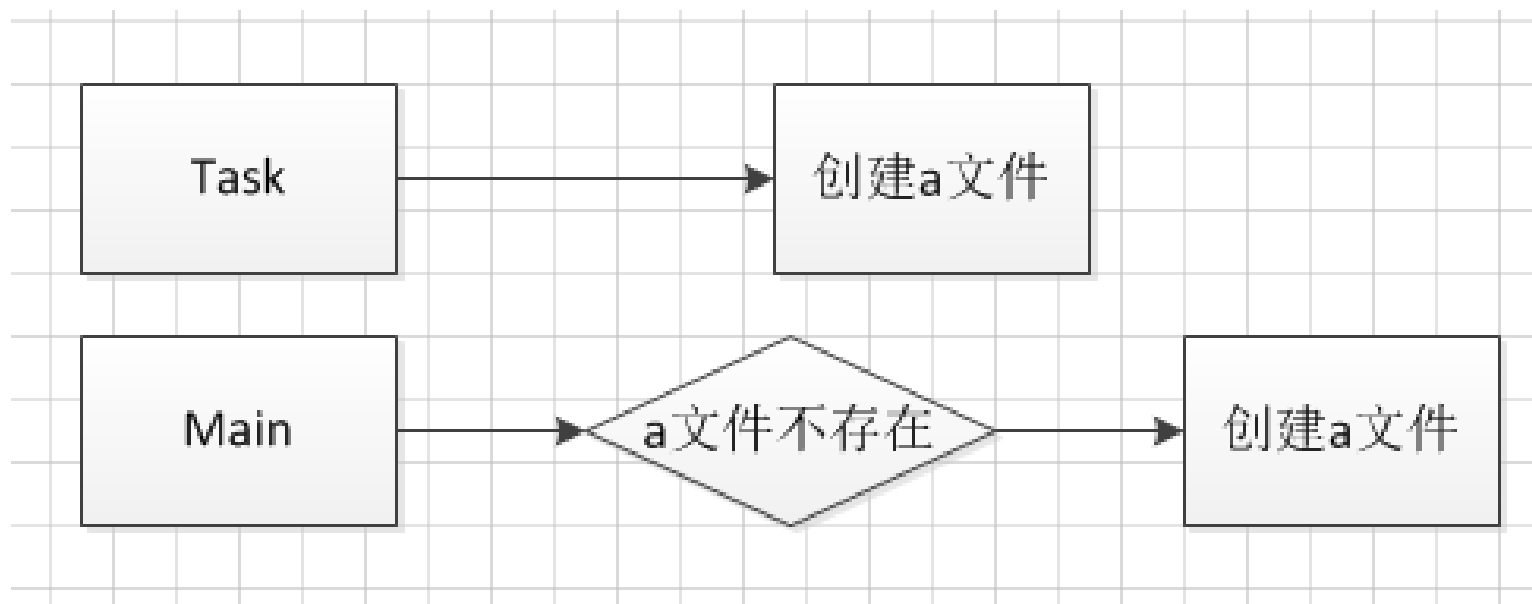
# 竞态条件

- 竞态条件 (Race Condition)
  - 当两个线程竞争同一资源时，如果对资源的访问顺序敏感，就称存在**竞态条件**
  - 计算的正确性取决于多个线程的交替执行时序时，就会发生竞态条件。



# 竞态条件

## ➤ 竞态条件 (Race Condition)



# 竞态条件

## ➤ 竞态条件 (Race Condition)

```
#include #include int main() {  
    char * fn = "/tmp/XYZ";  
    char buffer[60];  
    FILE *fp;  
    scanf("%50s", buffer );  
    if(!access(fn, W_OK)){  
        fp = fopen(fn, "a+");  
        fwrite("\n", sizeof(char), 1, fp);  
        fwrite(buffer, sizeof(char), strlen(buffer), fp);  
        fclose(fp); }  
    else printf("No permission \n"); }
```

①

②



# 竞态条件

## ➤ 脏牛漏洞

Linux内核的内存子系统在处理copy-on-write（COW）时出现竞争条件，导致私有只读存储器映射被破坏，可利用此漏洞非法获得读写权限，进而提升权限。



CVE-2016-5195



武汉大学

WUHAN UNIVERSITY

# 脏牛漏洞

- 该漏洞是Linux的一个本地提权漏洞，发现者是Phil Oester，影响 $\geq 2.6.22$ 的所有Linux内核版本，修复时间是2016年10月18号。
- 该漏洞的原因是get\_user\_page内核函数在处理Copy-on-Write(以下使用COW表示)的过程中，可能产出竞态条件造成COW过程被破坏，导致出现写数据到进程地址空间内只读内存区域的机会。
- 当我们向带有MAP\_PRIVATE标记的只读文件映射区域写数据时，会产生一个映射文件的复制(COW)，对此区域的任何修改都不会写回原来的文件，如果上述的竞态条件发生，就能成功的写回原来的文件。比如我们修改su或者passwd程序就可以达到root的目的。

# 脏牛漏洞

- POC
- Main:

```
fd = open(filename, O_RDONLY)
fstat(fd, &st)
map = mmap(NULL, st.st_size , PROT_READ,
MAP_PRIVATE, fd, 0)
start Thread1
start Thread2
```
- Thread1:

```
f = open("/proc/self/mem", O_RDWR)
while (1):
lseek(f, map, SEEK_SET)
write(f, shellcode, strlen(shellcode))
```
- Thread2:

```
while (1):
advise(map, 100, MADV_DONTNEED)
```

# 逻辑Bug

- 引用计数器溢出
- 物理设备输入验证
  - USB漏洞
- 内核生成的用户态漏洞





# 基于Cache的侧信道攻击

- 高速缓存 (CPU Cache) 是 CPU 附近用于加速 CPU 寄存器和主存信息交换的硬件设备。
- 针对 cache 的侧信道攻击也是典型的可利用的侧信道。
- cache 攻击利用了 CPU cache 访问速度和主存访问速度间的巨大时间差异进行侧信道分析。
- 早期的 cache 攻击致力于 L1 cache, 最近的针对 cache 的侧信道分析更着眼于 LLC (Last Level Cache, L3 cache), 因为该级 cache 将被所有的 CPU 核心使用。



# 基于Cache的侧信道攻击

- Cache侧信道不需要遵守权限
  - 跨CPU、跨核心、跨VM
  - 跨用户、沙盒外
- 危害：
  - 窃取机密信息，如RSA、EDCSA、AES加密密钥
  - 监视鼠标、键盘
  - 破坏内核 ASLR(地址空间配置随机加载)功能

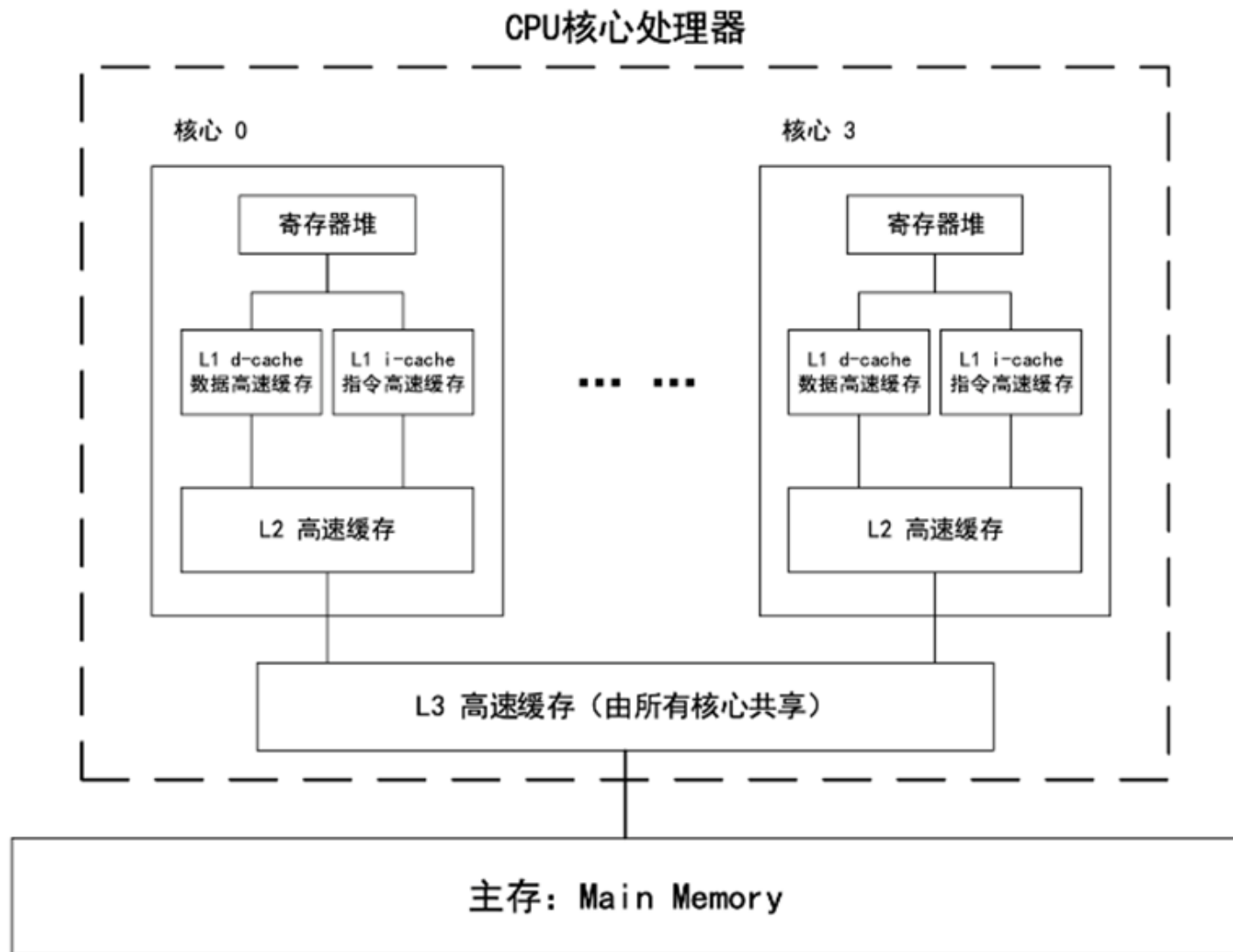


# 基于Cache的侧信道攻击

- Cache 采用了和寄存器相同的存储介质 SRAM，比主存要快得多（典型的读取时长是 4 个时钟周期）：它存储了 CPU 最近访问过的数据项和其相邻的数据项，由于局部性原理，CPU 之后的处理将会大概率用到这些数据。
- 当 CPU 执行一条取  $w$  字指令时，它将会向靠近的 cache 请求这个字，并最后由 cache 返回这个字。当请求的字已经存储在 cache 中，这种情况称为 cache 命中；反之称为 cache 未命中 (cache missing)，并执行一定的替换策略 (replacement policy)。



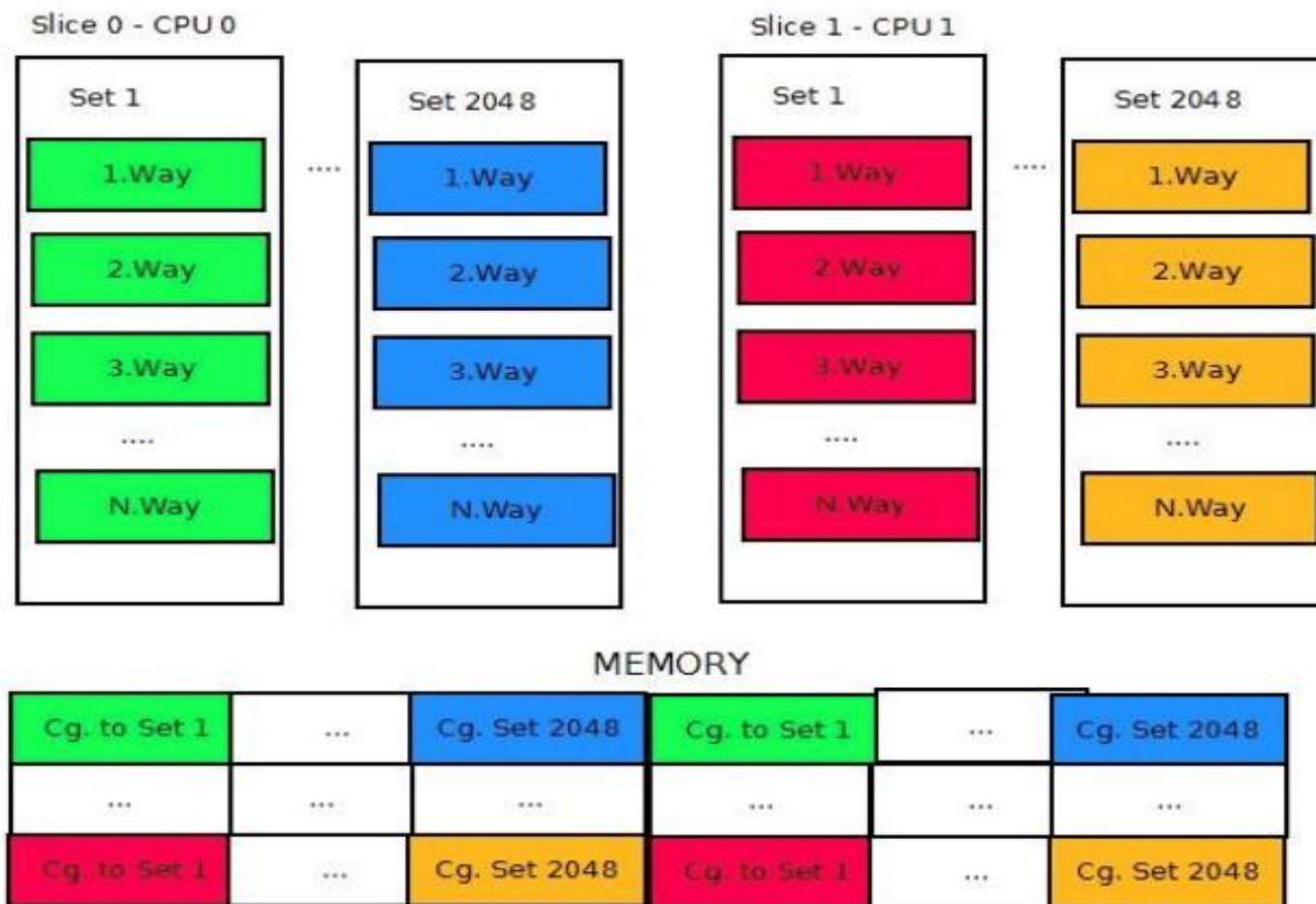
# 基于Cache的侧信道攻击



# 基于Cache的侧信道攻击



# 基于Cache的侧信道攻击



N 路组相连高速缓存



武汉大学

WUHAN UNIVERSITY

# Prime+Probe

- Prime + Probe 方法是第一种常用的 cache 侧信道攻击方法。在 Prime + Probe 攻击中，对侧信道信息的提取方法需要填充 (Prime)、触发 (Trigger)、预测 (Probe) 3 步进行：
  1. Prime: 在这一操作中，攻击者需要不断的填充 CPU cache (即将原有的 cache line 从 cache 中驱逐 (eviction)), 直到填满整个 cache 结构。对于一个 E 路 S 组相联的 cache，攻击者需要准备 n 个映射到在同一组的虚拟地址来填充这样的 cache 的一组 cache，其中 n 取决于 E 和 cache 控制器的替换策略。因此对于 Prime 操作，找到完全填充一个 cache set 的方法显然十分重要。
  2. Trigger: 等待目标读取敏感信息，更新 cache。
  3. Probe: 重新访问之前填充的地址组，记录每次访问的时间；由于攻击目标代码对敏感内存的访问会导致之前填充的某一条 cache line 对应的地址被驱逐，重新访问会出现 cache missing，需要从低级存储器中请求数据，这造成很大的可以被观测到的时间开销。



# Flush+Reload

- Flush + Reload 是一种改进了 Prime + Probe 的更有效的 cache 攻击方式。这种攻击中，攻击者和目标需要共享内存(共享动态库或者重复页面)。
- Flush + Reload 依赖于间谍进程和受害者进程之间的共享页面。通过共享页面，间谍进程可以确保从整个缓存层次结构中逐出特定的内存行。间谍进程用这个来监视对内存行的访问。





# Flush+Reload

- 一轮攻击包括三个阶段：

1. Flush: 在这一操作中，受监控的内存行从缓存层次结构中驱逐(如可以使用 `cflush` 指令)。
2. Trigger: 间谍进程等待受害者进程在第三阶段之前访问受监控的内存行。
3. Reload: 间谍进程重新加载受监控的内存线，测量加载时间。如果在等待阶段，受害者进程访问了内存行，则该内存行将在缓存中可用，重新加载操作将需要很短的时间。另一方面，如果受害者没有访问内存行，则需要从内存中取出该行，重新加载将花费相当长的时间。

# Flush+Reload

- Flush + Reload 经过研究证明为一种行之有效、准确度高且危害巨大的攻击手段；因此云服务商禁用了在云环境中唯一的共享内存手段，重复页面删除技术 (page deduplication)，以保证用户数据安全。
- 然而，Flush + Reload 是一种获取访问信息的有效方法，因此可以被其它攻击方式结合使用，而基于分支预测的侧信道攻击就在信息提取阶段使用了这一方法。



# Spectre & Meltdown

- 针对推测执行的攻击是一种新的侧信道攻击方式，利用了现代 CPU 在乱序执行时的分支预测和推测执行行为，推测执行的一些指令对存储器有瞬态影响，这些指令被称为瞬态指令。
- Spectre 和 Meltdown 攻击都利用了 CPU 中乱序执行时的推测执行机制，核心思想是尽管攻击者不能直接访问敏感内存，但可以利用推测执行将敏感数据读入 cache，再通过 cache 时间差异进行侧信道分析。

# Meltdown

```
1. raise_exception();  
2. access(probe_data[data * 4096]);
```

```
1. rcx = kernel address, rbx = probe array  
2. xor rax, rax  
3. retry:  
4. mov al, byte [rcx]  
5. shl rax, 0xc  
6. jz retry  
7. mov rbx, qword [rbx + rax]
```

Meltdown核心代码

# Meltdown

- Meltdown 是一种用于从用户态攻击内核的侧信道攻击方式。
- 在程序控制流中，由于在访问 `probe_data` 执行前引起了一个异常，对 `probe_data` 的访问不应被执行；
- 推测执行使得 CPU 在一定程度上执行了对 `probe_data` 的访问，虽然这些访问将会在 CPU 意识到程序控制流程已经改变时退役且不会改变寄存器或内存状态，但其访问已经被高速缓存 `cache` 记录。
- 通过在更高层面上利用这种 Meltdown 访问模式，恶意代码可以在很大程度上攻击操作系统。

# Meltdown

- 首先，在用户态内存生成一个256页的空内存：（因为要从核心态内存中窃取一个 byte 的值，这个值可能是0到255，所以使用256页大小的数组）

```
char[] probeArray = (char [])malloc(256 * 4096);
```

- 保证不读/写该数组，也就能避免这块数组中任何一页不进入 cache。
- 第4行（越权代码）：从内存中的 rcx 地址处（指向核心态）读一个 byte 的数据到 rax 的最后8位（al）；  
第5行：将 rax 向左移动12位（ $2^{12} = 4096$ ，即一页的字节数）；  
第6行：如果获取到的是0则继续尝试；  
第7行：从 probe array 中读取第 rax 页的头8个 byte 到寄存器 rbx 中（读到寄存器的值没有作用，但能够使得CPU将内存中第 rax 页上，cache 块大小的数据调入 cache ）。

# Meltdown

- 随后该先导程序越权被发现，操作系统终止掉该程序。
- 攻击者使用 Flush + Reload 确定访问的 cache line，从而确定存储在所选内存位置的秘密。通过对不同的内存位置重复这些步骤，攻击者可以转储内核内存，包括整个物理内存。
- Meltdown 实现了将一个不可访问的内核地址被移到寄存器中，引发异常。在引发异常之前，后续指令会无序执行，通过间接内存访问从内核地址泄漏数据。

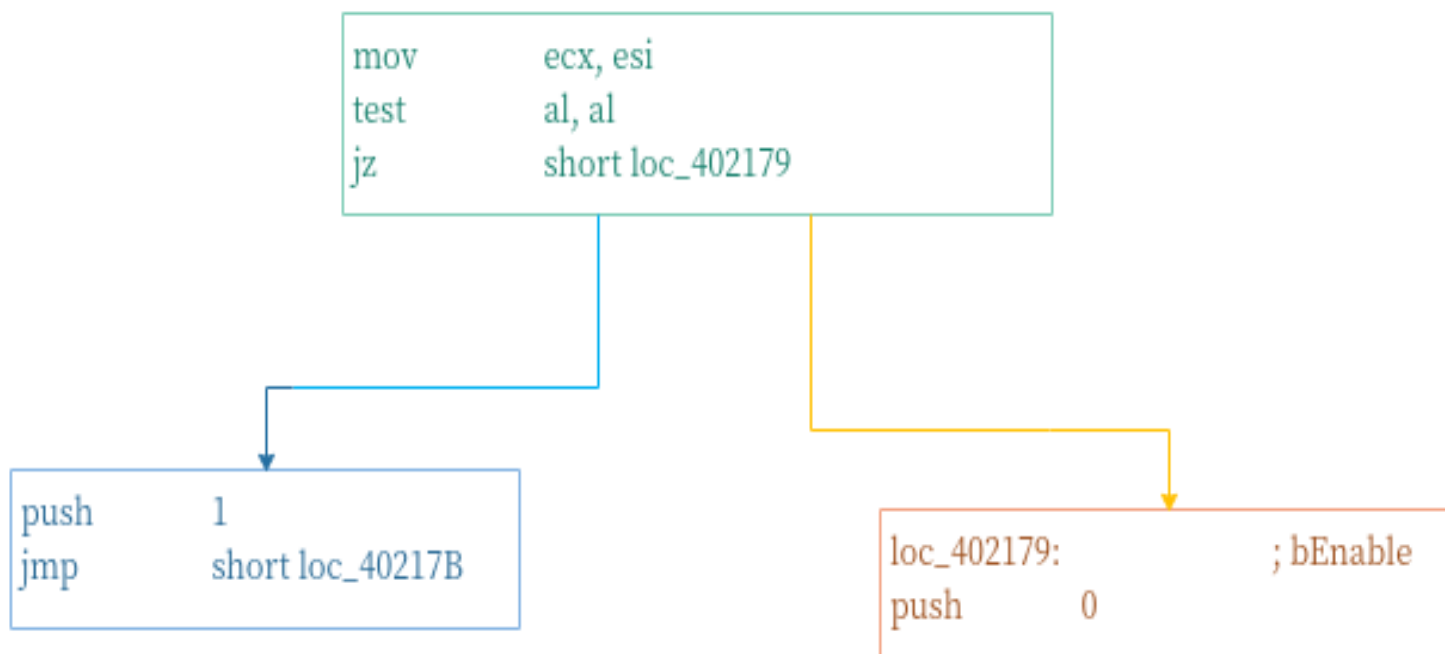
# Spectre

- Spectre 给出了通用的对推测执行的利用：通过训练分支预测机制来推测执行非法指令，其攻击对象不仅可以是本进程或共享库，充分的训练也可以攻击其他进程的内存空间。
- Spectre 攻击实现了对两种分支语句的注入：**条件分支语句和间接跳转语句**，这两种语句的特点是都存在推测执行且都可以通过一定方式被训练，这在 Spectre 攻击中是重要的利用。





# Spectre



条件跳转指令



# Spectre

- 现代 CPU 采取了一种被称为分支预测（Branch Prediction）技术，在遇到分支时，CPU 会猜测是否会选择分支，同时还会预测分支的目标地址。CPU 会开始取出位于它预测的分支会跳转到的指令，并对指令进行译码，并且会提前开始执行这些指令。
- 指令控制单元负责从寄存器中读出指令，对指令进行译码执行，并且指出分支预测是否正确。
- 如果分支点预测正确，那么被预测执行的该指令就可以执行退役（retired），所有对寄存器的更新就可以进行；如果分支预测错误，这条指令就会被清空（flushed），结果不会实际写入寄存器。但是，其执行结果却已经载入到 CPU 的 cache 中了，这就是 Spectre 得以发生的根源所在。



# Spectre

```
1. if(index1 < array_a_size) {  
2.     Index2 = array_a[index1];  
3.     if(index2 < array_b_size)  
4.         value = array_b[index2];  
5. }
```

Spectre 条件分支目标代码模式



# Spectre

Spectre 攻击对于条件分支的攻击开始于寻找一定的代码模式，这种模式充当了侧信道泄漏的来源，代码展示了这种模式：

- **arrayb** 是攻击者可以访问的数组，**arraya** 则是私有的敏感数组；
- 推测执行使得第 1 行的边界检查代码退役之前便访问了 **arrayb** 中的地址，进一步可以通过 cache 时间分析对 **arrayb** 进行检查已获取 **arraya** 的 **index1** 偏移处的数据(尽管越界)；



# 内核漏洞危害分类

- 远程拒绝服务内核漏洞
- 本地拒绝服务内核漏洞
- 远程缓冲区溢出内核漏洞
- 本地缓冲区溢出内核漏洞
- 任意地址写任意数据内核漏洞
- 任意地址写固定数据内核漏洞

# 内核漏洞危害分类

## ➤ 远程拒绝服务内核漏洞

受影响系统:

Linux kernel

描述:

---

BUGTRAQ ID: 70883

CVE(CAN) ID: CVE-2014-3673

Linux kernel的SCTP（Stream Control Transmission Protocol，流量控制传输协议）栈收到畸形ASCONF数据块后存在skb\_over\_panic内核崩溃，攻击者可利用此漏洞造成拒绝服务

# 内核漏洞危害分类

## ➤ 本地拒绝服务内核漏洞

linux kernel 2.2.12

2.2.14

2.3.99-pre2

不受影响系统: linux kernel 2.2.14 + solar designers kernel patch

描述:

-----

--

linux的unix域名套接字没有考虑

/proc/sys/net/core/wmem\_max的参数限制，本地普通用户可以通过向某个套接字传送大量数据，导致linux内核分配内存空间时出错，系统停止响应。必须重新启动系统。

# 内核漏洞危害分类

## ➤ 远程缓冲区溢出内核漏洞

受影响系统:

Linux kernel 2.6.23.1

描述:

BUGTRAQ ID: 26438

CVE(CAN) ID: CVE-2007-5904

-----  
Linux Kernel的CIFS VFS代码存在缓冲器溢出漏洞，远程攻击者可能利用此漏洞控制系统。

transport.c文件的SendReceive()函数将消息的有效负载memcpy到通过out\_buf参数所传送的缓冲区中，该函数假设所有缓冲区的大小为（CIFSMaxBufSize + MAX\_CIFS\_HDR\_SIZE），但调用时所使用的缓冲区为较小的MAX\_CIFS\_SMALL\_BUFFER\_SIZE。如果远程攻击者向有漏洞的系统发送了特制响应的话，就可以触发缓冲区溢出。



# 内核漏洞危害分类

## ➤ 本地缓冲区溢出内核漏洞

受影响系统:

Linux kernel

描述

BUGTRAQ ID: [72643](#)

CVE(CAN) ID: [CVE-2014-9683](#)

---

Linux kernel在eCryptfs的实现上存在缓冲区溢出安全漏洞，攻击者可利用此漏洞执行任意代码。

# 内核漏洞危害分类

## ➤ 任意地址写任意数据内核漏洞

受影响系统:

Linux kernel

描述

-----  
MTK FrameBuffer内核驱动是Linux内核中为了方便用户态应用操作图形硬件相关功能的模块。其通过/dev/graphics/fb0设备接口与用户态通信。

在处理MTKFB\_CAPTURE\_FRAMEBUFFER命令过程中，代码没有对用户传入的指针进行限定，导致可以对任意虚拟地址进行改写。如果覆盖的地址中包含类似uid的数据结构即可用于提权等目的。

# 内核漏洞危害分类

## ➤ 任意地址写固定数据内核漏洞

受影响系统:

Windows kernel

描述

---

HWiNFO32驱动过滤不严，造成任意地址写固定数据漏洞。

驱动精灵中包含HWiNFO32，其名称为Mydriver32.sys  
对DeviceIoControl例程中，当IoControlCode=0x85FE2600时，不严格过滤用户传入的 lpOutBuffer参数,直接调用  
nt!IoPnpCompleteRequest后，经过一系列处理，最终在  
nt!IoPnpCompleteRequest产生漏洞，可写任意地址。

# 内核Hook技术

- Rootkit 的一种经典形式是通过 Hook 系统调用实现。
- 系统调用挂钩技术中，最简单、最流行的方案是修改[sys call table](#)

# sys\_call\_table

- [/arch/x86/syscalls/syscall\\_32.tbl](#)

```
..
0      i386  restart_syscall      sys_restart_syscall
1      i386  exit                sys_exit
2      i386  fork                sys_fork                stub32_fork
3      i386  read                sys_read
4      i386  write               sys_write
5      i386  open                sys_open                compat_sys_open
6      i386  close               sys_close
7      i386  waitpid             sys_waitpid             sys32_waitpid
8      i386  creat               sys_creat
9      i386  link                sys_link
10     i386  unlink              sys_unlink
11     i386  execve              sys_execve              stub32_execve
12     i386  chdir               sys_chdir
13     i386  time                sys_time                compat_sys_time
14     i386  mknod               sys_mknod
15     i386  chmod               sys_chmod
16     i386  lchown              sys_lchown16
17     i386  break
18     i386  oldstat             sys_stat
19     i386  lseek               sys_lseek                compat_sys_lseek
```



# 内核Hook技术

- Hook系统调用
  - 劫持system\_call->sys\_call\_table进行系统调用Hook
    - 获取sys\_call\_table(保存所有系统调用例程的入口地址)的地址
    - 找到其中需要hook的函数，替换成自己编写的hook函数

# 内核Hook技术

- Hook系统调用
  - 获取sys\_call\_table中函数入口地址的方法
    - 通过/proc/kallsyms获取系统函数例程的入口地址
    - 通过/boot/system.map.\*文件获取
    - 通过Kprobe获取

# 内核Hook技术

```
ca247360 t mem_cgroup_oom_control_read
ca2480d0 t mem_cgroup_read_u64
ca249230 t mem_cgroup_swappiness_read
ca25a4e0 T vfs_readf
ca25a9e0 t do_iter_readv_writev
ca25b1c0 t new_sync_read
ca25c450 T vfs_iocb_iter_read
ca25c550 t do_iter_read
ca25c6e0 T vfs_iter_read
ca25d7e0 T __vfs_read
ca25d830 T vfs_read
ca25d980 T kernel_read
ca25d9c0 T ksys_read
ca25da90 T __ia32_sys_read
ca25dba0 T ksys_pread64
ca25dc30 T __ia32_sys_pread64
ca25de20 T vfs_readv
ca25dea0 t do_readv
ca25dfb0 T __ia32_sys_readv
ca25dfd0 t do_preadv
ca25e0b0 T __ia32_sys_preadv
ca25e0d0 T __ia32_sys_preadv2
ca262850 t do_readlinkat
ca262970 T __ia32_sys_readlinkat
ca262990 T __ia32_sys_readlink
ca263330 T read_code
ca263a30 T kernel_read_file
```

/proc/kallsyms



武汉大学

WUHAN UNIVERSITY



# 内核Hook技术

- 由于sys\_call\_table所在的内存是有写保护的， 所以我们需要先去掉写保护， 再做修改。

```
void  
disable_write_protection(void)  
{  
    unsigned long cr0 = read_cr0();  
    clear_bit(16, &cr0);  
    write_cr0(cr0);  
}
```

[LINUX hook实验](#)



武汉大学

WUHAN UNIVERSITY

# Linux提权方法

## 1) 修改cred结构提权

```
1  struct cred {
2      atomic_t    usage;
3  #ifdef CONFIG_DEBUG_CREDENTIALS
4      atomic_t    subscribers; /* number of processes subscribed */
5      void        *put_addr;
6      unsigned    magic;
7  #define CRED_MAGIC    0x43736564
8  #define CRED_MAGIC_DEAD 0x44656144
9  #endif
10     kuid_t      uid; /* real UID of the task */
11     kgid_t      gid; /* real GID of the task */
12     kuid_t      suid; /* saved UID of the task */
13     kgid_t      sgid; /* saved GID of the task */
14     kuid_t      euid; /* effective UID of the task */
15     kgid_t      egid; /* effective GID of the task */
16     kuid_t      fsuid; /* UID for VFS ops */
17     kgid_t      fsgid; /* GID for VFS ops */
18     unsigned    securebits; /* SUID-less security management */
19     kernel_cap_t cap_inheritable; /* caps our children can inherit */
20     kernel_cap_t cap_permitted; /* caps we're permitted */
21     kernel_cap_t cap_effective; /* caps we can actually use */
22     kernel_cap_t cap_bset; /* capability bounding set */
```

# 提权方法

2) 利用系统权限检查缺陷，为/bin/bash设置S位，再使用设置了S位的/bin/bash打开shell，从而获取root权限

具体实例：

漏洞编号：CVE-2015-8660

漏洞提权原理：该漏洞是fs/overlayfs/inode.c中ovl\_setattr()函数的一个失误引起的。在一个namespace中的进程会拥有自己的CAP\_SYS\_ADMIN权限，所以可以修改自己namespace下mount的overlayfs上的文件属性，可被利用绕过文件系统检查，从而逃逸namespace，修改任意文件属性，通过设置挂载的bash的SUID位，子进程结束后在主进程中执行setreuid将uid、gid、eid置零，再起一个shell即为root权限。



# 提权方法

## 3) 利用漏洞对任意文件写数据

具体实例

漏洞编号: CVE-2016-5195

漏洞提权方法:

修改/etc/passwd下root用户的shell指向, 间接使用管理员权限运行给定的命令。

修改/etc/group文件, 将普通用户添加到sudo组

如果拥有一个普通用户, 可以利用漏洞修改/etc/group文件, 将用户添加到sudo组, 进而获取root权限。



# 提权方法

## 4) 直接修改/etc/passwd中用户的uid和gid为0

(以下修改 enjoy用户, uid=1000, gid=1000)

. /CVE-2016-5195

```
/etc/passwd "$(sed '
s/enjoy:x:1000:1000:enjoy,,,:\/home\/enjoy:\/b
in\/bash/enjoy:x:0:0:enjoy,,,:\/home\/enjoy:\/
bin\/bash /g' /etc/passwd)"
```



# 提权方法

## 5) 直接修改/etc/passwd文件中root用户的密码

先用python生成密码的sha-512加密密文

```
import crypt
```

```
crypt.crypt('abc123', '$6$123456')
```

```
>>> import crypt
>>> crypt.crypt("abc123", "$6$123456")
'$6$123456$AgHJOZCWFmiFlrHae1nc116/r9Hacnd8PwBkvEja6v.aXS8TJPZJvywkfdMzoBjN6Xbot
8W4mvktAc6IOWFCi/'
>>>
```



# 内核漏洞防御

## ➤ 输入输出检查

- 对输入和输出参数的指针和长度进行检查

## ➤ 安全验证和过滤

- 对驱动设备控制函数的调用者进行验证，一般只允许自己的进程调用

## ➤ 安全编码

- 编写安全的代码，防止逻辑错误、缓冲区溢出等

## ➤ 驱动白名单机制

# 内核漏洞防御

一些新的机制：

- CFI（控制流完整性）
- VSM（虚拟安全模式）
- SMEP（监管模式执行保护）
- KPTI（内核页表隔离）
- Grsecurity
- Selinux



完成实验四：Linux下Hook系统服务功能，  
替换成自己编写的服务函数。

