操作系统安全

第三部分 内存安全保护

中国科学院大学
网络空间安全学院
2018.3.16

中国科学院大学
University of Chinese Academy of Sciences

中国科学院信息工程研究所
INSTITUTE OF INFORMATION ENGINEERING,CAS

第三部分 内存安全保护 纲要

- 3.1 内存攻击原理及防护技术
  - Linux系统安全威胁情况
  - 漏洞类型及攻击原理
  - Linux可执行文件组织格式
  - 内存安全机制
  - 地址空间随机化分配实现
- 3.2 Rootkit攻击原理及检测技术
  - Rootkit攻击原理
  - Rootkit检测技术

第三部分 内存安全保护 纲要

- 3.3 可信隔离技术
  - 可信隔离架构
  - 可信隔离技术
    - » Intel SGX
    - » AMD SME SEV
    - » ARM TrustZone
- 3.4 内存安全保护实验
  - 缓冲区溢出与数据执行保护（DEP）实验
  - 跨运行级提权实验
  - Rootkit检测实验

《操作系统安全》

第三部分 内存安全保护

3.1 内存攻击原理及防护技术

中国科学院大学
网络空间安全学院
2018.4

中国科学院大学
University of Chinese Academy of Sciences

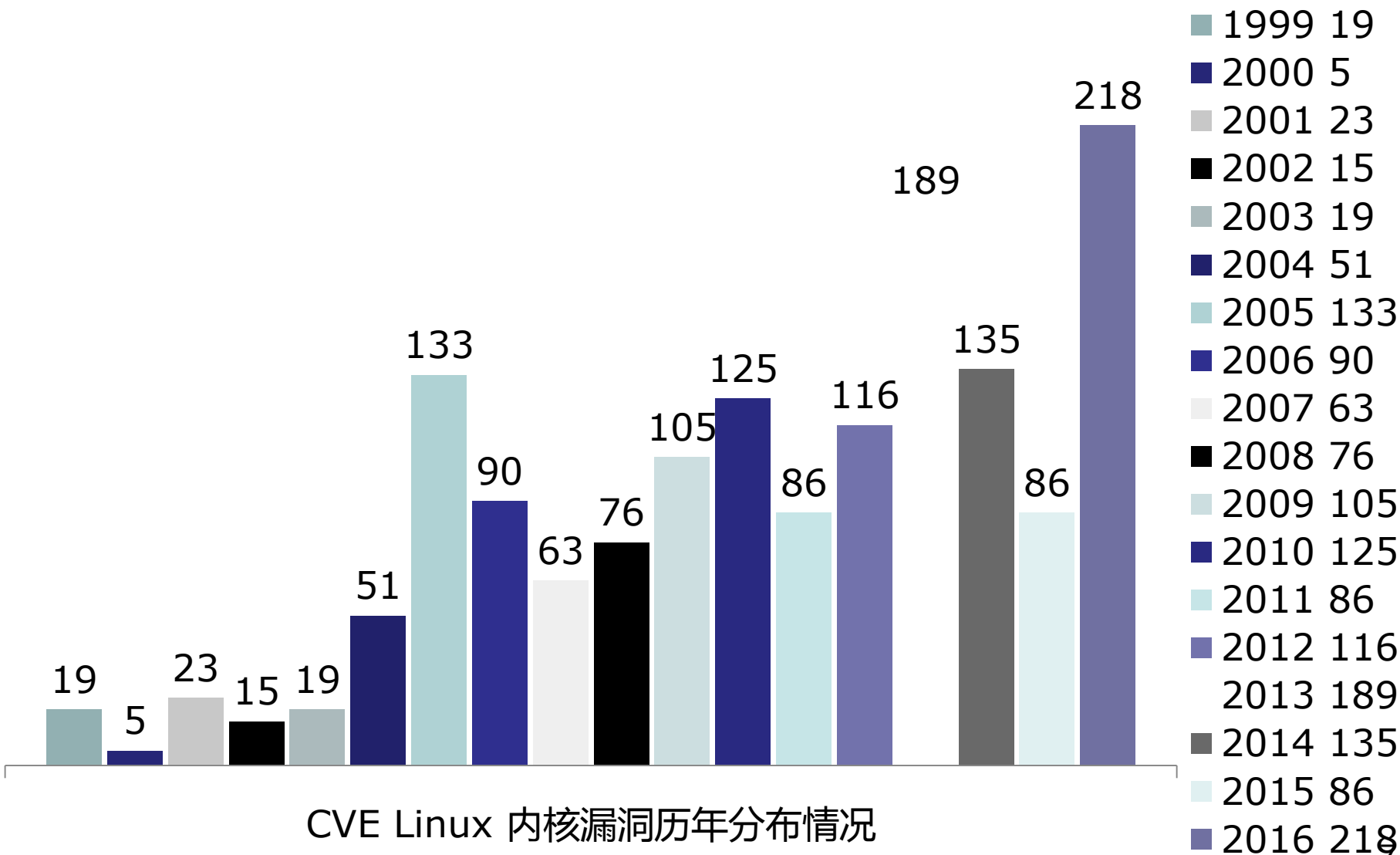
中国科学院信息工程研究所
INSTITUTE OF INFORMATION ENGINEERING,CAS

目录

重大漏洞事件

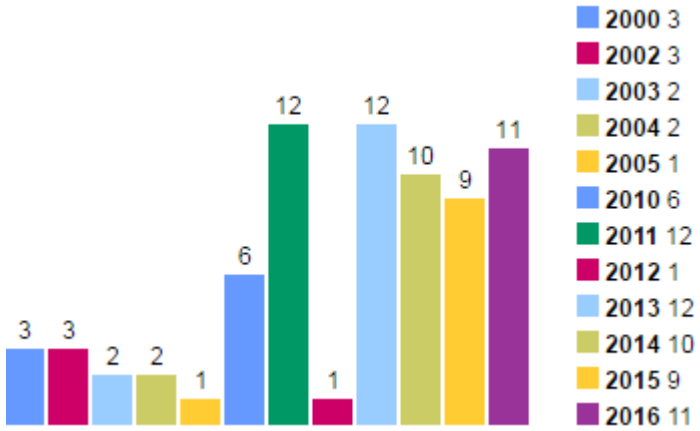
- 2001年“红色代码”蠕虫
  - 利用微软Web Server中的缓冲区溢出漏洞使30多万台计算机受到攻击
- 2003年Slammer蠕虫爆发
  - 利用微软SQL Server 2000缓冲区溢出漏洞
- 2004年“震荡波病毒”爆发
  - 利用Windows系统的活动目录服务缓冲区溢出漏洞
- 2005年“狙击波病毒”爆发
  - 利用Windows即插即用缓冲区溢出漏洞
- 2008年Conficker蠕虫病毒
  - 利用Windows处理远程RPC请求时的漏洞
- 2014年“心脏滴血漏洞”
  - OpenSSL出现“Heartbleed”安全漏洞，内存数据泄漏成千上万的服务器都处于危险之中

Linux内核漏洞历年分布情况

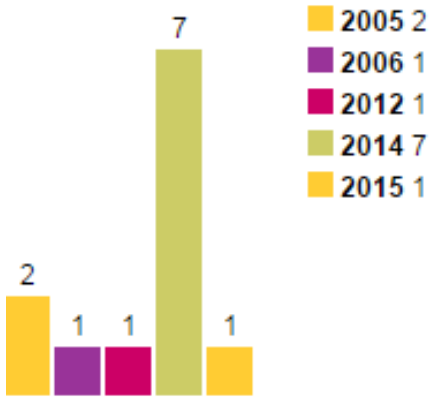


Linux核心库漏洞分布情况

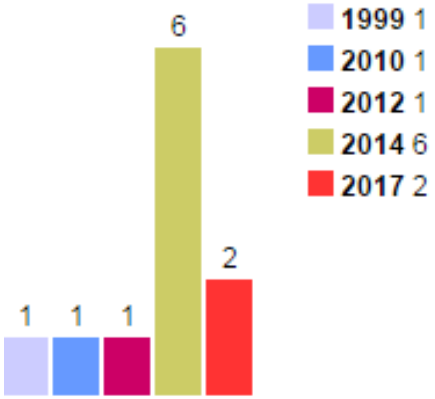
Glibc历年漏洞情况



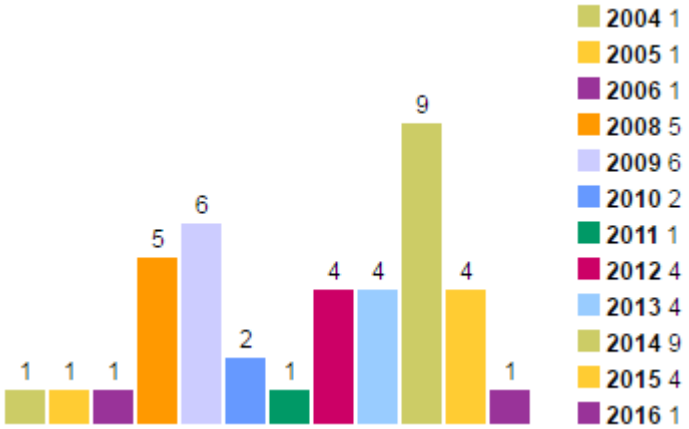
Binutils历年漏洞情况



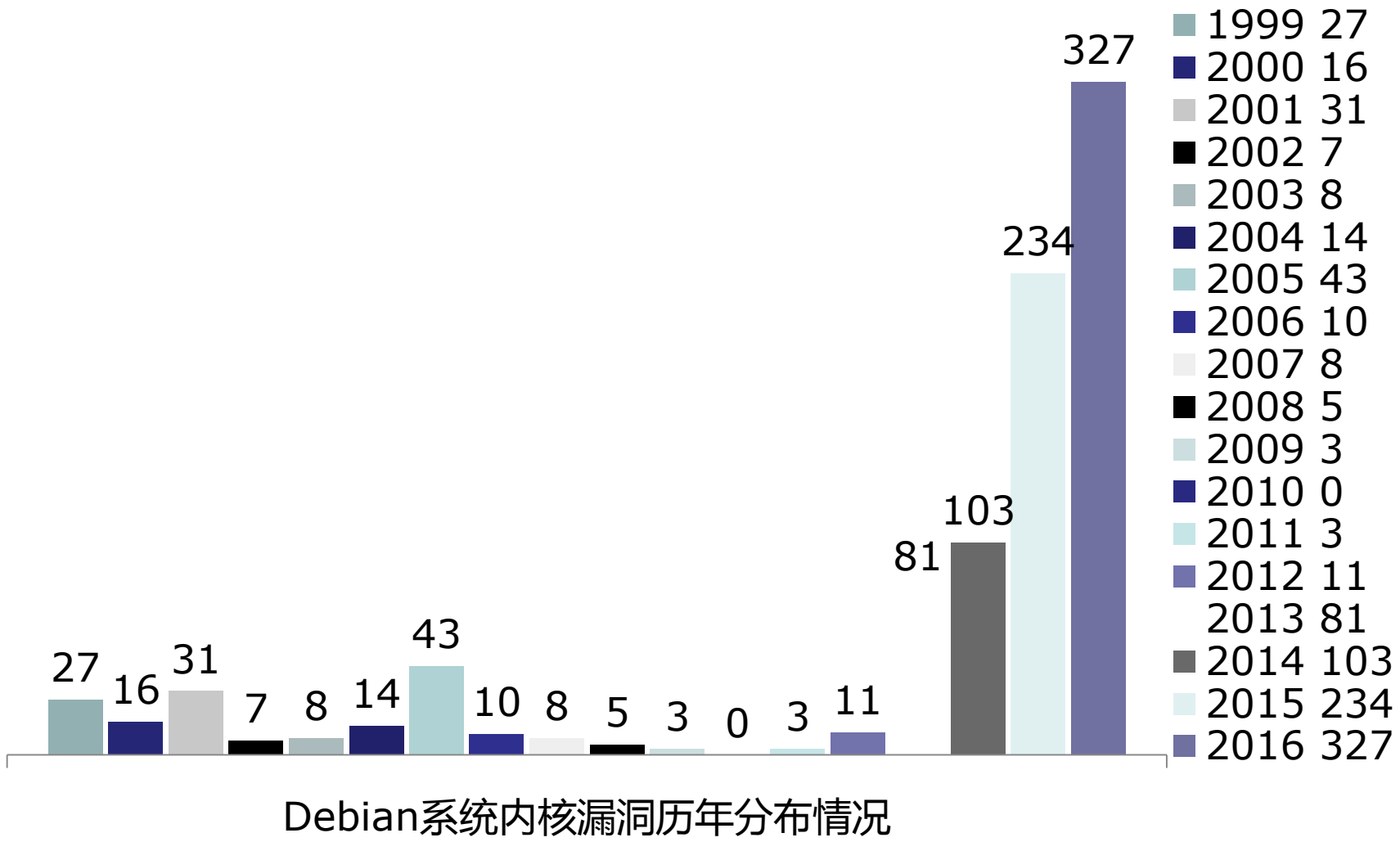
Bash历年漏洞情况



Gnutls历年漏洞情况



Linux 系统发行版漏洞历年分布情况



Debian系统内核漏洞历年分布情况

知名漏洞库

中国国家漏洞库：http://www.cnvd.org.cn
美国国家漏洞库：http://web.nvd.nist.gov
美国国家信息安全应急小组：http://secunia.com
国际权威漏洞机构Secunia：http://secunia.com
国际权威漏洞库SecurityFocus：http://www.securityfocus.com
IBM网络安全漏洞库Xforce：http://xforce.iss.net
国际权威漏洞库OSVDB：http://osvdb.org
俄罗斯知名安全实验室SecurityLab.ru：http://en.securitylab.ru
国内安全厂商绿盟科技：http://www.nsfocus.net
国内权威漏洞库：http://sebug.net
通用漏洞库CVE:cve.mitre.org

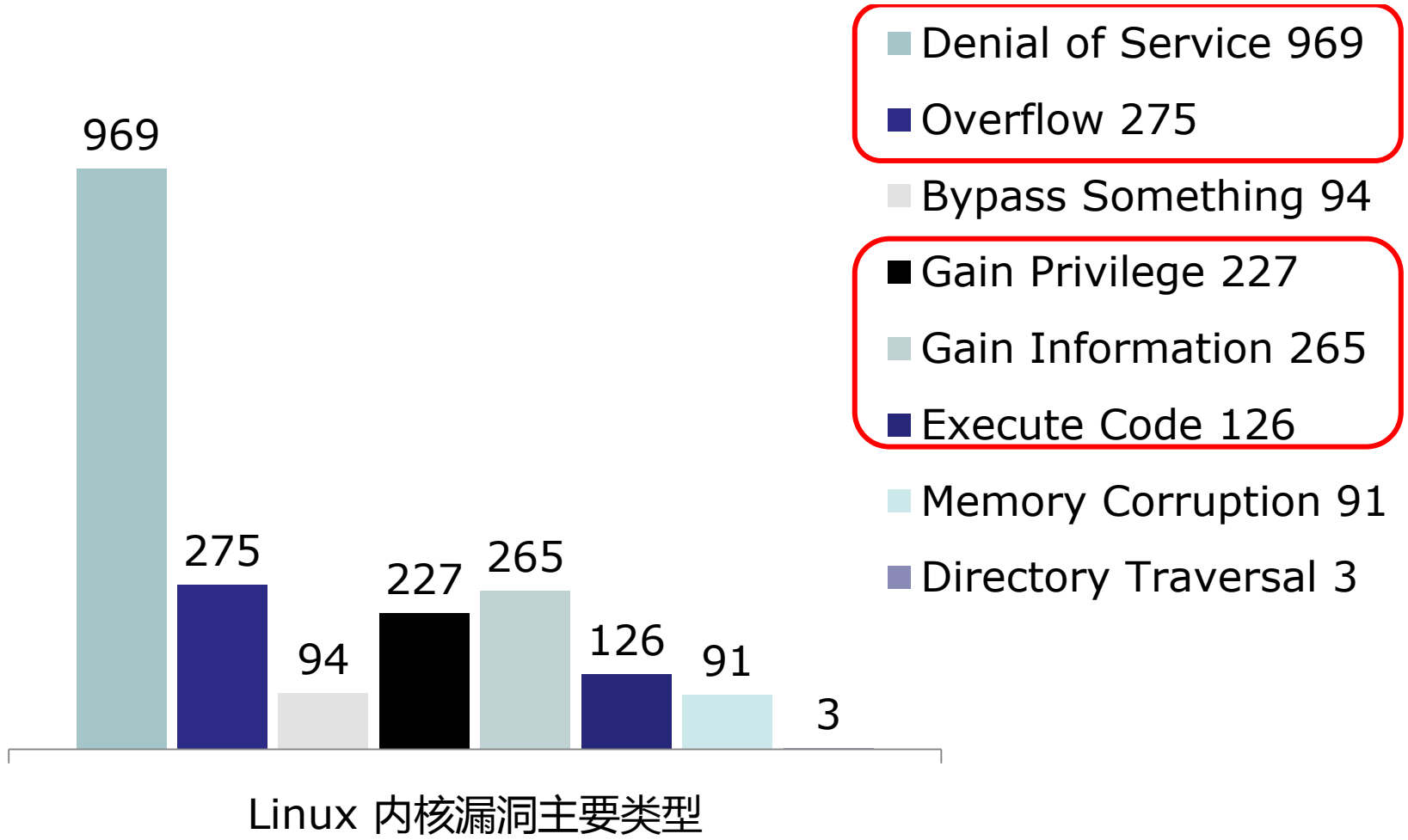
Linux内核漏洞类型

- CVE通用漏洞库
  - CVE—— Common Vulnerabilities and Exposures
  - 通用漏洞和暴露平台
  - 各漏洞库描述不统一，不同步，需要一个通用描述
  - 帮助用户在各自独立的各种漏洞数据库中和漏洞评估工具中共享数据
  - 可以快速地在任何其它CVE兼容的数据库中找到相应修补的信息，解决安全问题
  - 格式：CVE-年-编号
    - » CVE-2010-2743

CVE主要漏洞类型

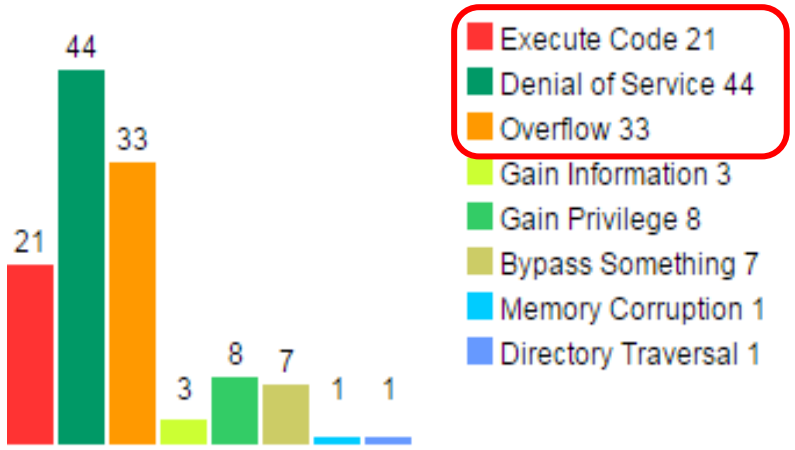
- Denial of Service
- Overflow
- Bypass Something
- Gain Privilege
- Gain Information
- Execute Code
- Memory Corruption
- Directory Traversal
- XSS
- SQL Injection
- CSRF
- Http Response Splitting

Linux 内核漏洞主要类型

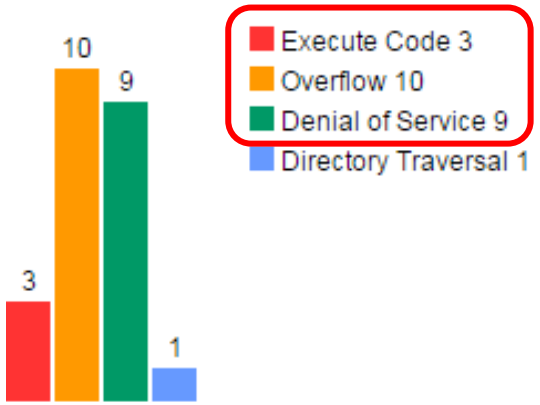


Linux核心库主要漏洞类型

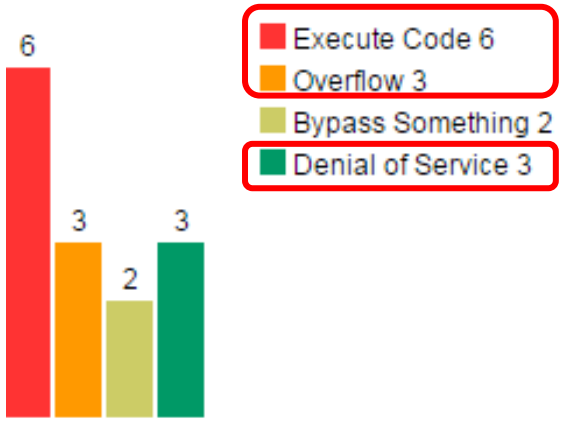
Glibc主要漏洞类型



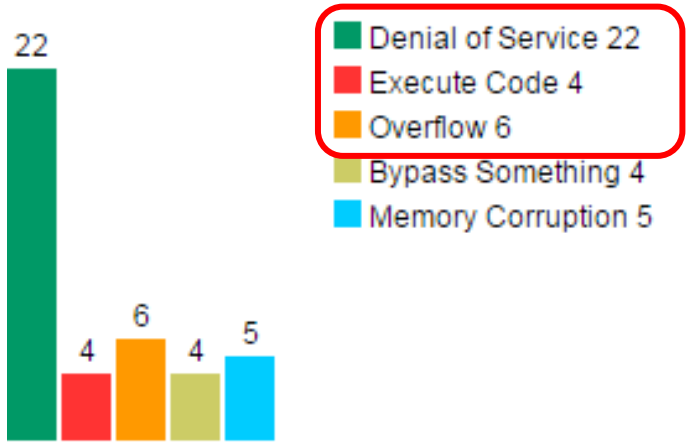
Binutils主要漏洞类型



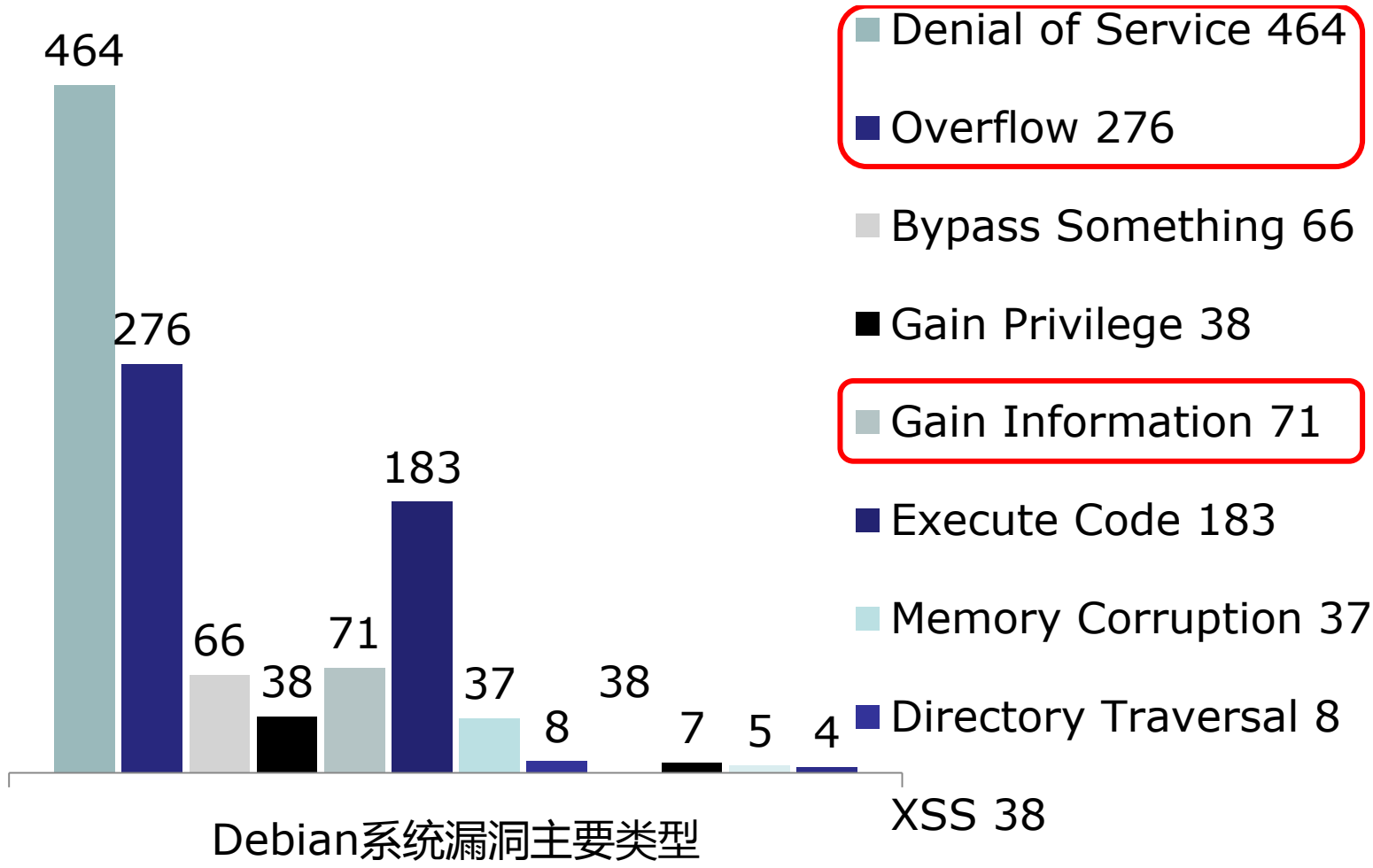
Bash主要漏洞类型



Gnutls主要漏洞类型

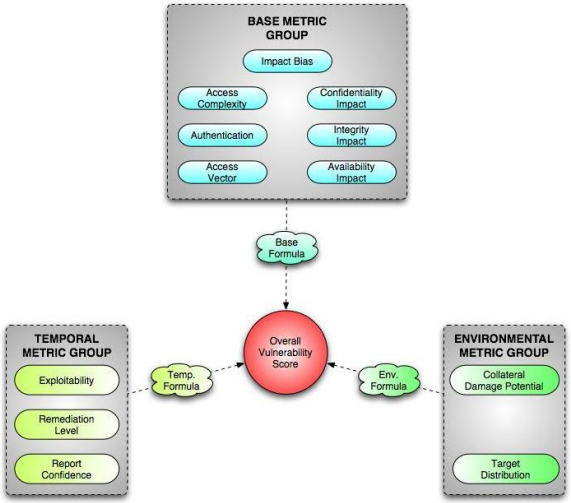


Linux 系统发行版漏洞主要类型



CVE漏洞评分系统

- 漏洞评分系统——CVSS
  - Common Vulnerability Scoring System
  - 通用漏洞评分系统：https://www.first.org/cvss
  - 评分要素：基本、环境、生命周期



Linux内核漏洞类型

- 漏洞评分系统——CVSS

基本评价（Base Metrics）			
metric	**要素**	**可选值**	**评分标准**
AccessVector	攻击途径	远程/本地	0.7/1.0
AccessComplexity	攻击复杂度	高/中/低	0.6/0.8/1.0
Authentication	认证	需要/不需要	0.6/1.0
confidentiality	机密性	不受影响/部份地/完全地	0/0.7/1.0
integrity	完整性	不受影响/部份地/完全地	0/0.7/1.0
availability	可用性	不受影响/部份地/完全地	0/0.7/1.0
bias	权值倾向	平均/机密性/完整性/可用性	各0.333/权值倾向要素0.5另两个0.25
生命周期评价（Temporal Metrics）			
metric	**要素**	**可选值**	**评分标准**
Exploitability	利用代码	未提供/验证方法/功能性代码/完整代码（无需代码）	0.85/0.90/0.95/1.00
Remediation Level	修正措施	官方补丁/临时补丁/临时解决方案/无	0.87/0.90/0.95/1.00
Report Confidence	确认程度	传言/未经确认/已确认	0.90/0.95/1.00
环境评价（Environmental Metrics）			
metric	**要素**	**可选值**	**评分标准**
Collateral Damage Potential	影响	无/低/中/高	0/0.1/0.3/0.5
Target Distribution	目标分布	无/低/中/高（0/1-15%/16-49%/50-100%）	0/0.25/0.75/1.00

Linux内核漏洞类型

- CVSS 版本

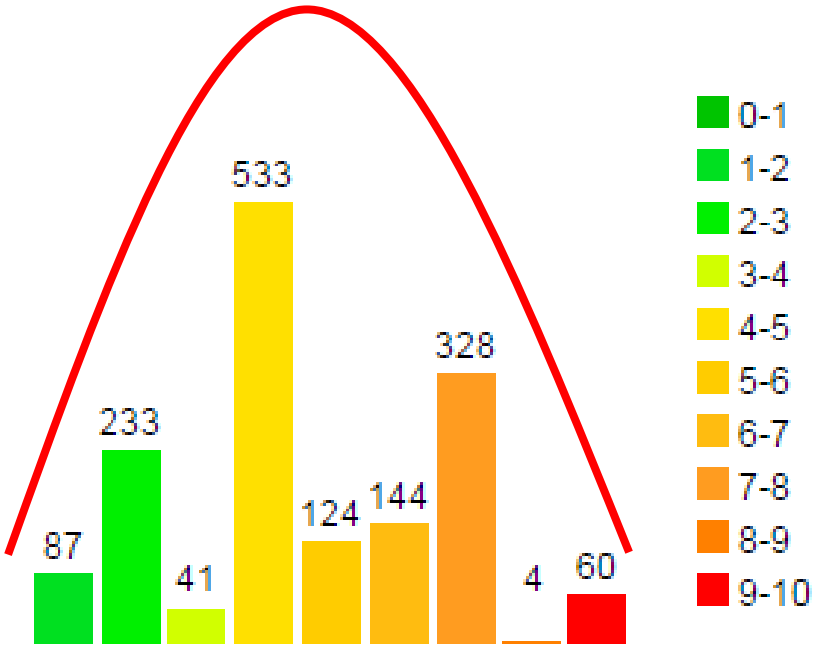
CVSS v2.0 Ratings

Severity	Base Score Range
Low	0.0-3.9
Medium	4.0-6.9
High	7.0-10.0

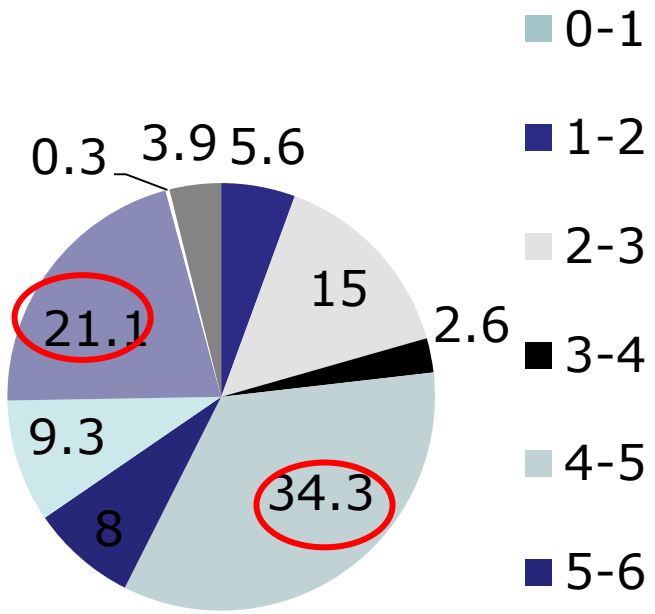
CVSS v3.0 Ratings

Severity	Base Score Range
None	0.0
Low	0.1-3.9
Medium	4.0-6.9
High	7.0-8.9
Critical	9.0-10.0

Linux 内核漏洞CVSS分值分布情况



各分值内的漏洞数目分布情况

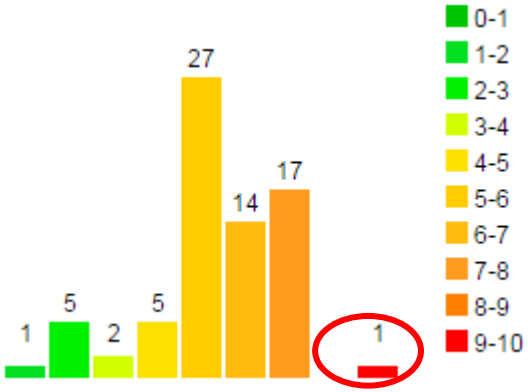


各分值内的漏洞分布百分比（%）

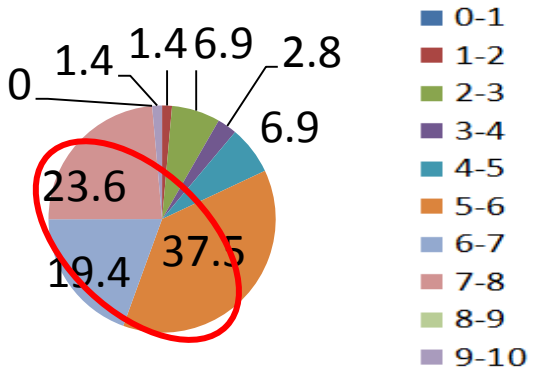
Total 1554, Weighted Average CVSS Score: **5.6**

GNU 底层工具库漏洞CVSS分值分布情况

- Glibc漏洞CVSS分值分布情况

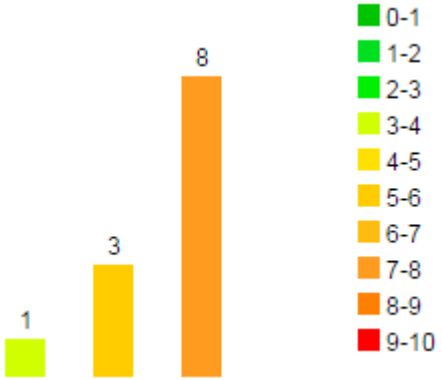


- Glibc漏洞CVSS分值分布情况

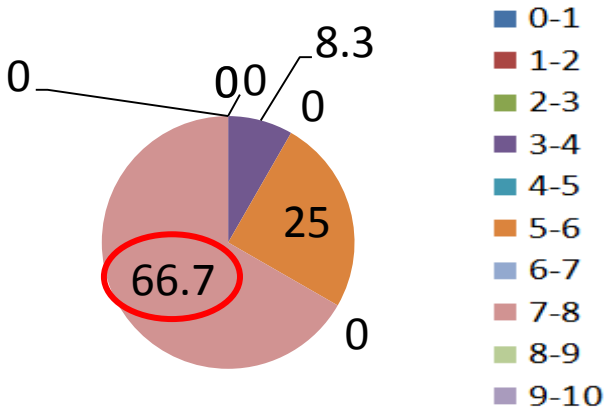


Total 72, Weighted Average CVSS Score: **6.3**

- Binutils漏洞CVSS分值分布情况



- Binutils漏洞CVSS分值分布情况

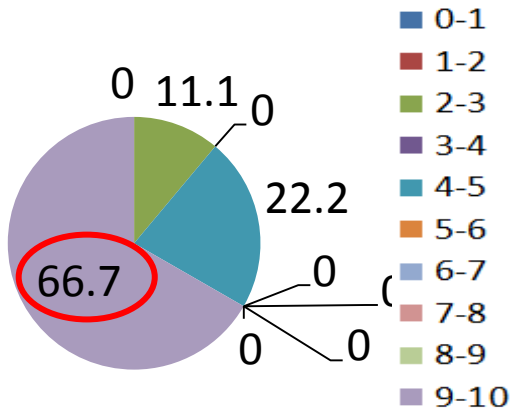


Total 12, Weighted Average CVSS Score: **7.2**

GNU 底层工具库漏洞CVSS分值分布情况

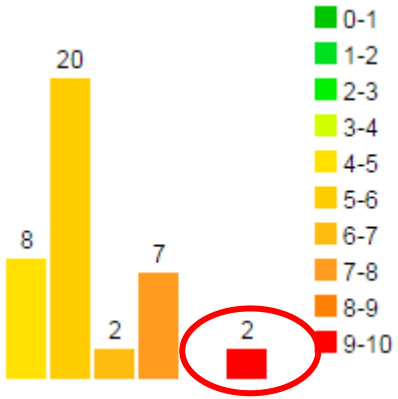
- Bash漏洞CVSS分值分布情况

- Bash漏洞CVSS分值分布情况

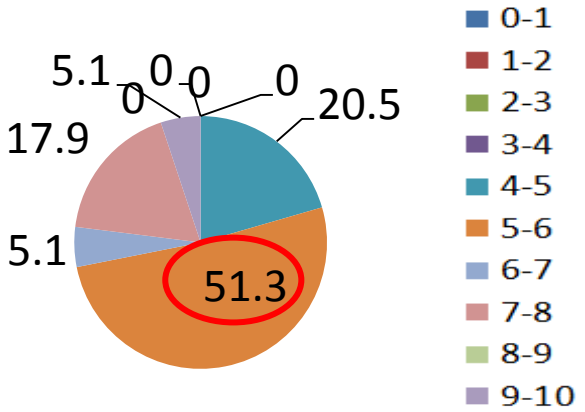


Total 9, Weighted Average CVSS Score: **8.1**

- Gnutls漏洞CVSS分值分布情况

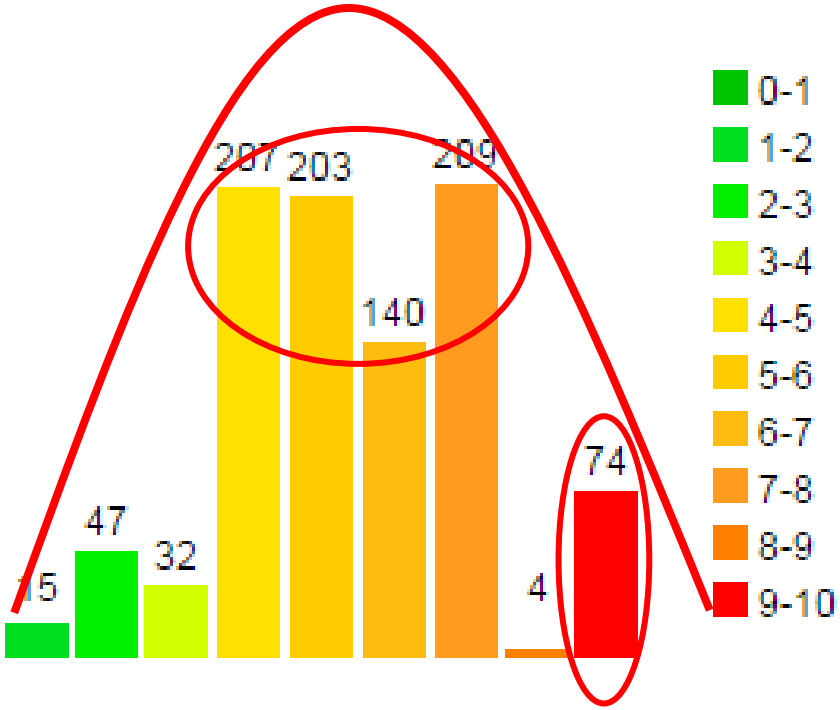


- Gnutls漏洞CVSS分值分布情况

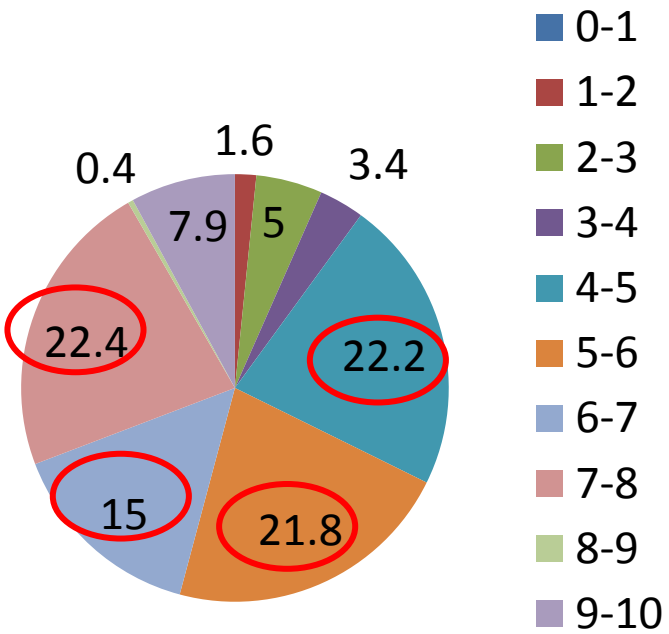


Total 39, Weighted Average CVSS Score: **6.4**

Linux 系统发行版漏洞CVSS分值分布情况



Debian系统各分值区间的漏洞数目分布情况



Debian系统各分值区间的漏洞分布百分比（%）

Total 931, Weighted Average CVSS Score: **6.4**

Linux 系统漏洞总结

- 漏洞数总体呈现扩大趋势
- 漏洞主要集中在：
  - Denial of Service
  - Overflow
  - Memory Corruption
  - Execute Code
  - Gain Privilege
  - Gain Information
- 漏洞主要集中在中、高危险漏洞，风险大
- 核心库漏洞要么不出现，要么就高危！

OpenSSL
HeartBleed

Linux内核漏洞类型

- 课后思考&动手
  - 找一个漏洞库检索
  - 找几个漏洞，查看漏洞描述，查看漏洞补丁

目录

缓冲区溢出攻击理论体系发展历史

- 1988年最早利用缓冲区溢出漏洞攻击
  - 1988年Morris蠕虫利用fingerd程序缓冲区溢出漏洞
- 1989年，Spafford提交了一份分析报告
  - 描述了VAX机上BSD版Unix的fingerd的缓冲区溢出程序技术细节，引起了部分安全人士对这个研究领域的重视
- 1996年，Aleph One 发表“Smashing the stack for fun and profit”文章
  - 首次详细介绍了Unix/Linux下栈溢出攻击的原理、方法和步骤，揭示了缓冲区溢出攻击中的技术细节
- 1999年w00w00安全小组的Conover发表基于堆缓冲区溢出的专著
  - 堆/BSS缓冲区溢出漏洞开始被关注

内存脆弱性来源

- 存储空间溢出漏洞
- 指针类漏洞
- Ret2addr类逻辑漏洞

缓冲区

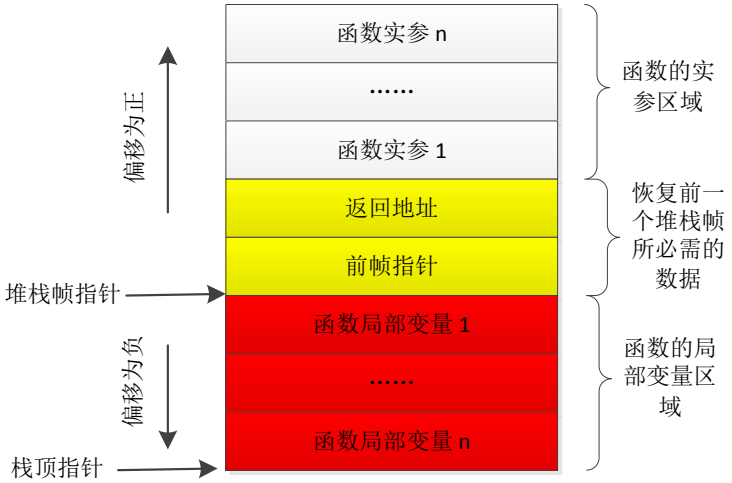
- 一段存储区域
- 有边界
- 动态存储和执行区域
- Memory，Cache
- 栈缓冲区，堆缓冲区，BSS缓冲区

缓冲区溢出

- 缓冲区溢出
  - 数据长度超出分配区域，覆盖其它数据的分配区域
  - 执行非授权指令，获取信息，取得系统特权
  - 进而进行各种非法操作。
  - 导致程序运行失败、系统宕机、重新启动等后果
  - 一种非常普遍、非常危险的漏洞
  - 在各种操作系统、应用软件中广泛存在

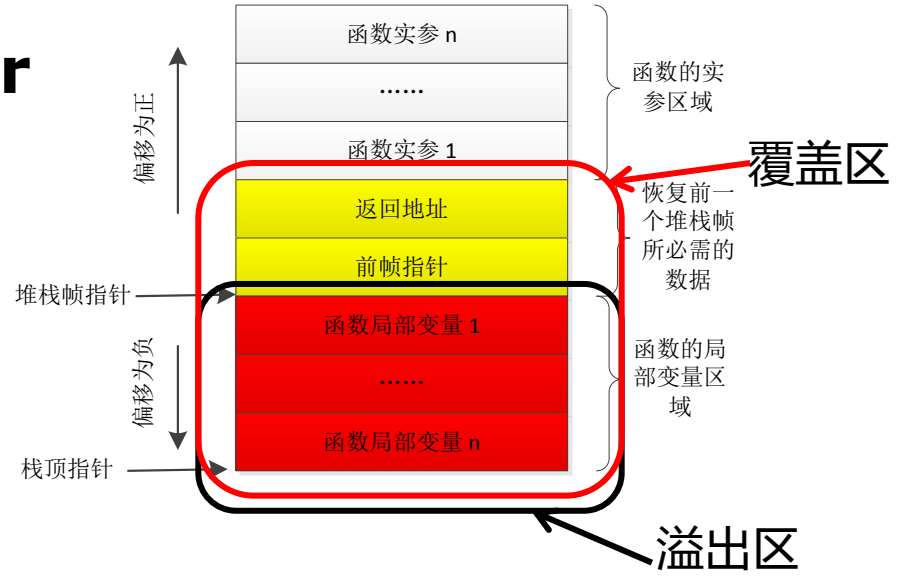
栈缓冲区溢出

- C函数栈帧结构



栈缓冲区溢出

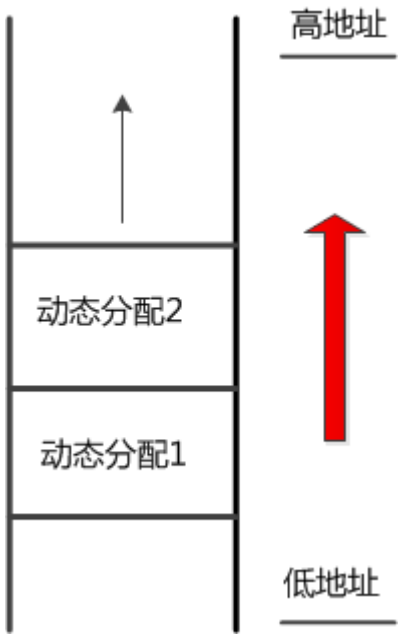
```
#include <stdio.h>
int main(int  argc,char
**argv)
{
   char buf[16];
   strcpy( buf,
argv[1]);
   return 0;
}
```



堆缓冲区溢出

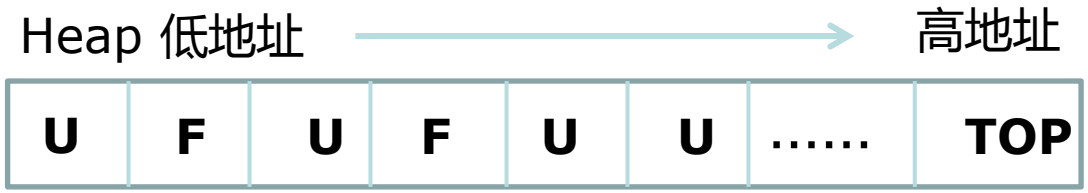
- 堆(HEAP)，是应用程序动态分配的内存区
- 堆块从低端地址向高端地址分配

```
char * malloc1 = malloc(200 *
sizeof(char));
int * malloc2 = malloc(100 *sizeof(int));
```



堆缓冲区溢出

- Linux 堆管理
  - 整个堆区被划分成若干个连续的块(chunk)

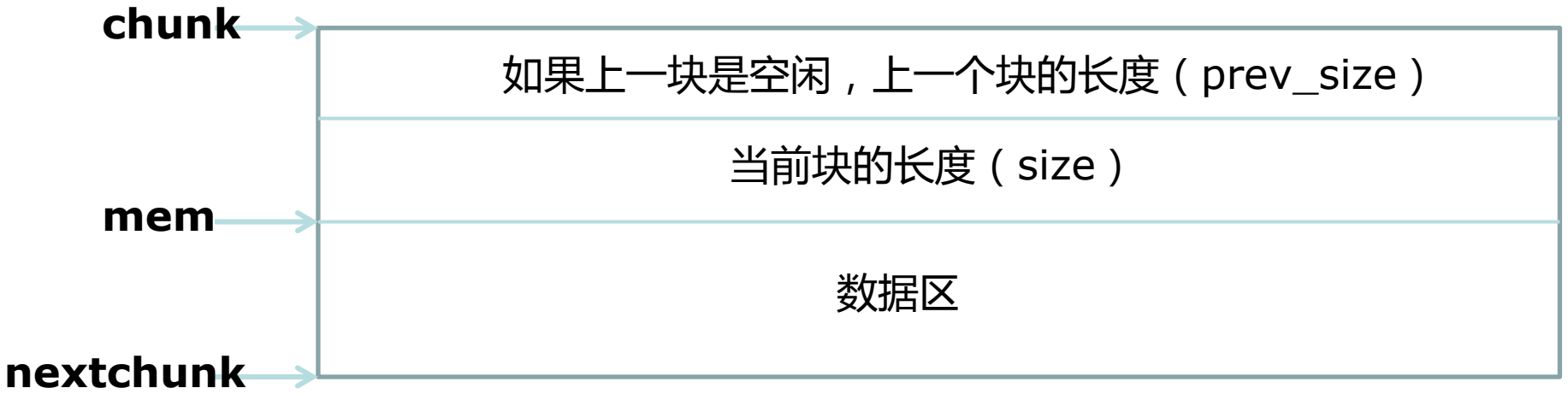


  - 标记：
    - U —正在被使用的块
    - F —空闲的块
    - TOP —位于高地址最边缘的那个块

堆缓冲区溢出

- 堆块数据（chunk）结构

```
struct malloc_chunk
{
int prev_size;                    // 如果上一块是空闲，此值为上一块
的长度
int size;                         // 当前块的长度包括管理结构本身
struct malloc_chunk* fd; //双向链表的前指针
struct malloc_chunk* bk; // 双向链表的后指针
}
```



堆缓冲区溢出

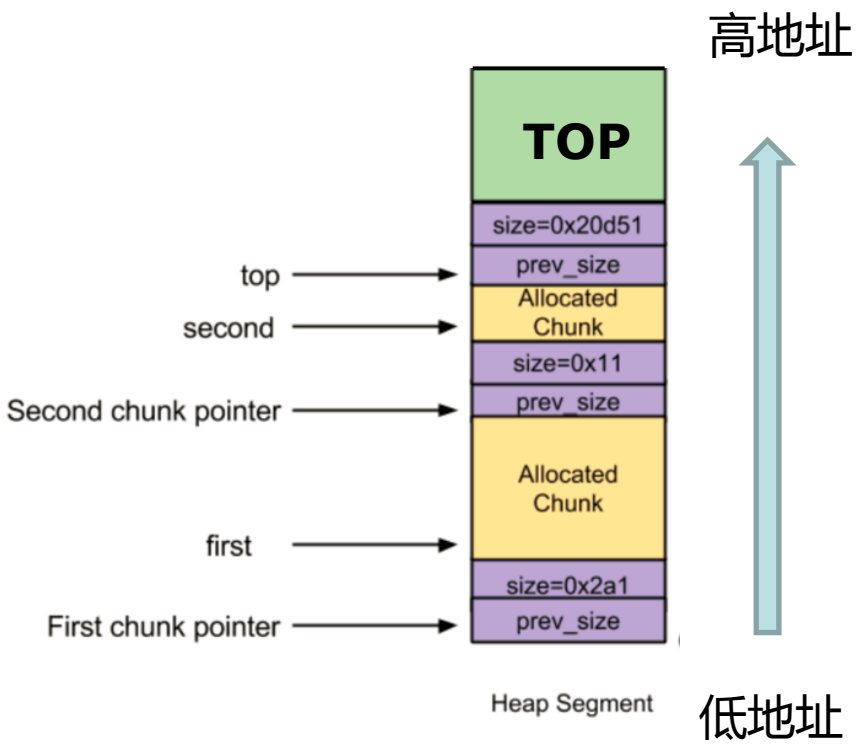
- double free 漏洞
- 利用malloc unlink机制

```
#define unlink(AV, P, BK, FD) {
        FD = P->fd;
        BK = P->bk ;
        if(FD->bk != P || BK->fd != P)
        ……
        else {  FD->bk = BK;
        BK->fd = FD; }
}
```

堆缓冲区溢出

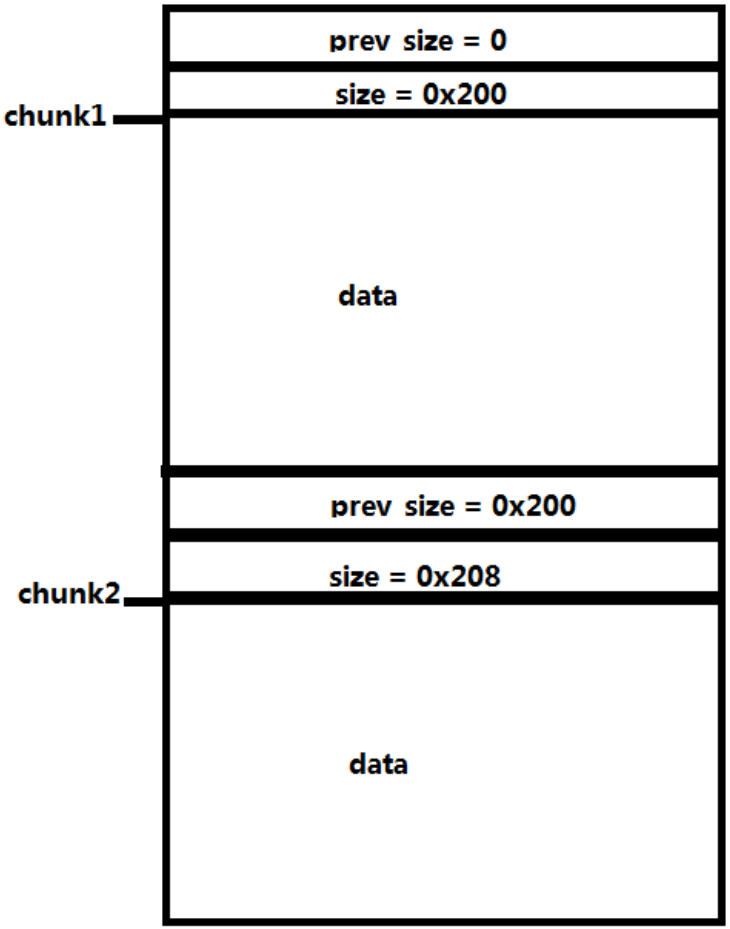
- 堆缓冲区溢出（double free malloc unlink ）

```
/*
 Heap overflow vulnerable program.
 */
#include<stdlib.h>
#include<string.h>

int main( intargc, char * argv[] )
{
        char * first, * second;

/*[1]*/ first= malloc( 666 );
/*[2]*/ second= malloc( 12 );
        if(argc!=1)
/*[3]*/       strcpy( first, argv[1] );
/*[4]*/ free(first );
/*[5]*/ free(second );
/*[6]*/ return(0 );
}
```



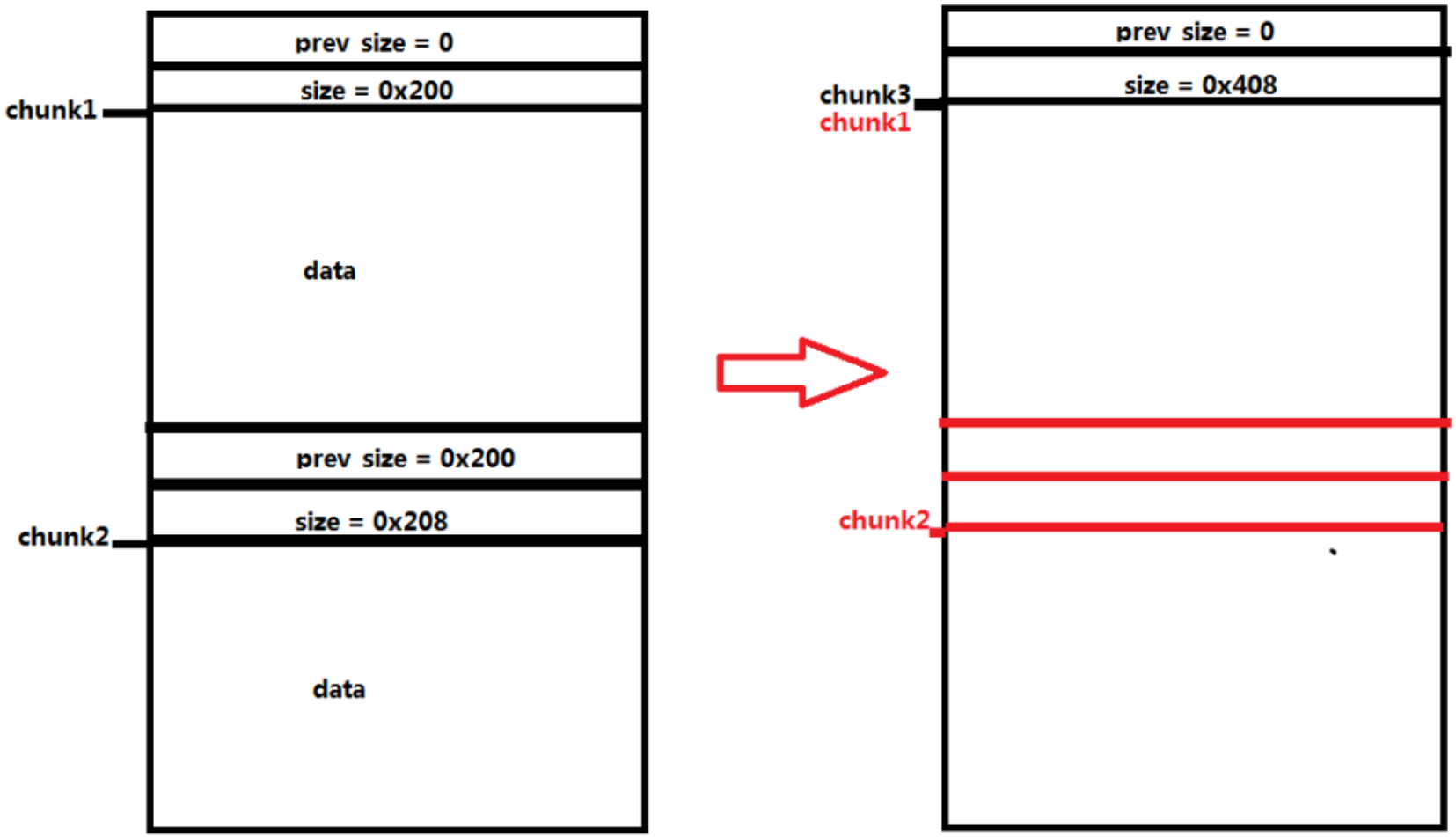
堆缓冲区溢出——double free

2个堆快 chunk1和chunk2



堆缓冲区溢出——double free

- 释放这两个堆 free(chunk1) free(chunk2)
- 再申请一个新的堆 chunk3 = malloc(1024)



堆缓冲区溢出——double free

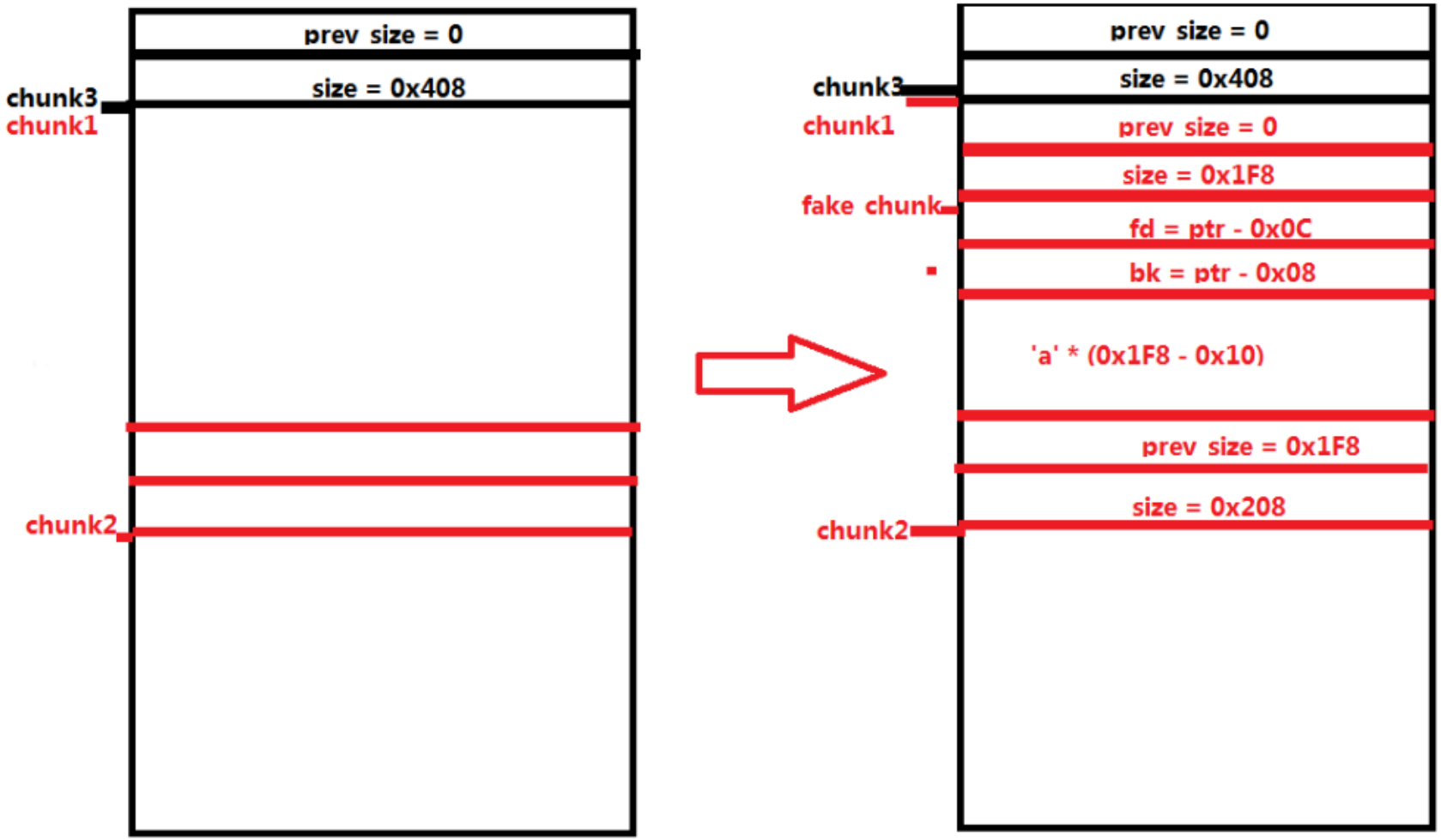
在chunk3中我们构造一个伪chunk(fake_chunk)

向chunk3中输入：

payload = （prev_size = 0） + (size = 0x1F8) + (fd = ptr - 0x0c) + (bk = ptr - 0x08) + 'a'*(0x1F8 - 0x10) + (prev_size = 0x1F8) + (size = 0x208)

堆缓冲区溢出——double free

chunk3伪造一个chunk后



堆缓冲区溢出——double free

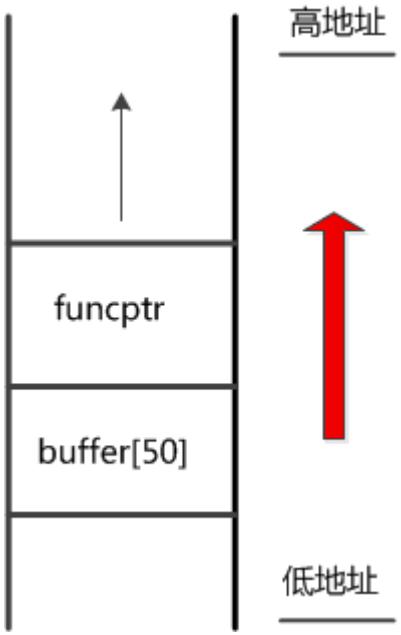
- 伪造chunk，数据结构关键参数公式
  - prev_size = even number and hence PREV_INUSE bit is unset.
  - size = -4
  - fd = free address – 12
  - bk = shellcode address

堆缓冲区溢出——double free

- 实现任意写
  - p = &p - 3
  - 实现任意写之后可以将 free@got 覆写成 system() 地址:
- shellcode
  - 覆盖free函数GOT entry
  - 改为system系统调用

数据堆缓冲区溢出

```
static char buffer [50];
static int(*funcptr)();
……
 while( *str )
{
*(buffer++) = (*str);
 *(str++);
 }
*funcptr();
```



整型溢出

- 机器数受存储空间限制，只能表示一定范围的数，如16位机器表示的数据范围-32768 ~ 32767
- 超出机器数所表达范围，产生数据上溢/下溢

整型溢出

- 整型溢出类型
  - 宽度溢出（Widthness Overflow）
    - 尝试存储一个超过变量表示范围的大数到变量中

```
void main(int argc,char* argv[])
{
        unsigned short s;
        int i;
        char buf[80];
        i = atoi(argv[1]);
        s = i;
        if(s >= 80)
                return;
        memcpy(buf,argv[2],i);
}
```

整型溢出

- 运算溢出（Arithmetic Overflow）
  - 如果存储值是一个运算操作，程序稍后使用这个运算结果进行后续操作，如申请动态存储区。

```
int func(char *userdata)
{
short datalength ;
datalength = datalength * 2;
buff = malloc( datalength );
strncpy( buff, userdata, datalength);
printf("userdata: %s\n", userdata);
printf("buff : %s\n", buff);
return 0;
}
```

整型溢出

- 符号溢出(Signedness Bug)
  - 一个无符号的变量被看作有符号，或者一个有符号的变量被看作无符号

```
int main()
{
    short int a=0;
       a=32767;
    cout<<"a before is:"<<a<<endl;
    cout<<"a after is:"<<++a<<endl;
    cout<<"a after is:"<<++a<<endl;
   return 0;
}
```

输出是？

格式化字符串溢出

- 关键字
  - “%n”
- 产生原因
  - printf()等格式化函数参数个数不固定
  - printf()等格式化函数不会检查输入参数的个数
  - 附加格式符，如%100d

格式化字符串溢出实例

- 格式化字符串溢出

```
#include
int main(void)
{
int i=1,j=2,k=3;
char buf[]="test";
printf("%s %d %d
%d\n",buf,i,j,k);
return 0;
}
```

输出 : test 1 2 3

printf("%s %d %d %d\n",buf,i,j);

输出 : test 1 2 1953719668

printf("%s %d %d %x\n",buf,i,j);

输出 : test 1 2 74736574

74→'t' 73→'s' 65→ 'e' 74→'t'

```
#include
int main(void)
{
int num;
int i=1,j=2,k=3;
printf("%d%d%d%n\n",i,
j,k,&num);
printf("%d\n",num);
return 0;
}
```

输出: 123
3

printf("%.200d%n",i,&num);

num=200

格式化主要函数

- strcpy，wcscpy，lstrcpy，_tcscpy，_mbscpy，strcat，wcscat，lstrcat，_tcscat，_mbscat，strncpy，memcpy……
- printf，scanf , gets，fprintf, sprintf, snprintf, vfprintf, vprintf，vsprintf, vsnprintf，setproctitle, syslog, err*, verr*, warn*, vwarn*……

溢出漏洞总结

- 大object向小object复制数据(字符串或整型)，容纳不下造成溢出
- 溢出会覆盖一些关键性数据（返回地址、管理数据、异常处理或文件指针等）
- 利用程序的后续流程，得到程序的控制权

内存脆弱性来源

- 存储空间溢出漏洞
- 指针类漏洞
- Ret2addr类逻辑漏洞

指针类漏洞

- 野指针引用
  - 引用一个未被初始化的指针
- 悬挂指针引用
  - 引用一个指向空间已经释放的指针
- 空指针引用漏洞
  - 引用NULL指针
- Double free
- Use after free
- 内存泄漏漏洞
  - 由指针指向的内存区域，在利用完后，没有释放内存，导致内存泄露

空指针类漏洞

- 引发问题

  地址非法访问，信息泄漏，拒绝服务，系统崩溃……

内存脆弱性来源

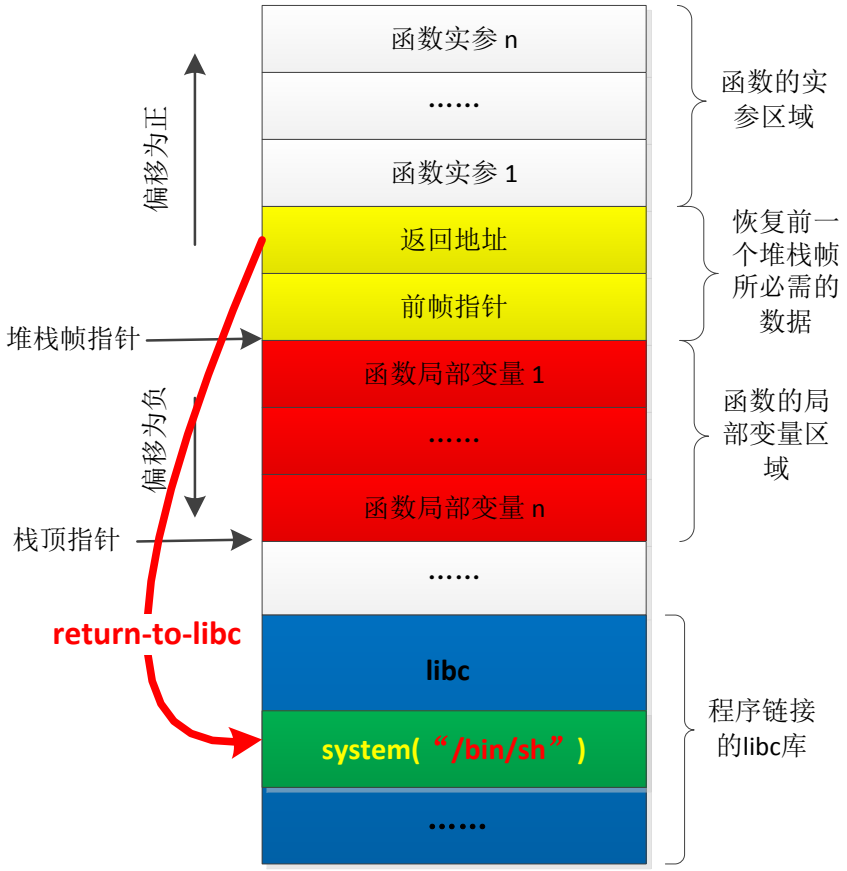
- 存储空间溢出漏洞
- 指针类漏洞
- Ret2addr类逻辑漏洞

Ret2lib

- Return to libc
- 缓冲区溢出攻击变体
  - 不需要在缓冲区中植入代码
  - 只需修改栈帧的返回地址
  - 将漏洞函数返回到内存空间已有的动态库函数中
- 利用程序链接库
  - 不破坏W⊕X

Ret2lib

- Return-to-libc攻击



Ret2plt

- PLT全称为Procedure Linkage Table
- ASCII armoring保护机制使得ret2libc不凑效
- pop; pop; ret；指令序列提取函数地址
- 不依赖libc地址空间
- 在PLT表中提取函数地址

```
0x8048340 <printf@plt>:      jmp   *0x804a000
0x8048346 <printf@plt+6>:    push  $0x0
0x804834b <printf@plt+11>:   jmp   0x8048330
0x8048350 <strcpy@plt>:      jmp   *0x804a004
0x8048356 <strcpy@plt+6>:    push  $0x8
0x804835b <strcpy@plt+11>:   jmp   0x8048330
0x8048360 <puts@plt>:        jmp   *0x804a008
```

Ret2plt

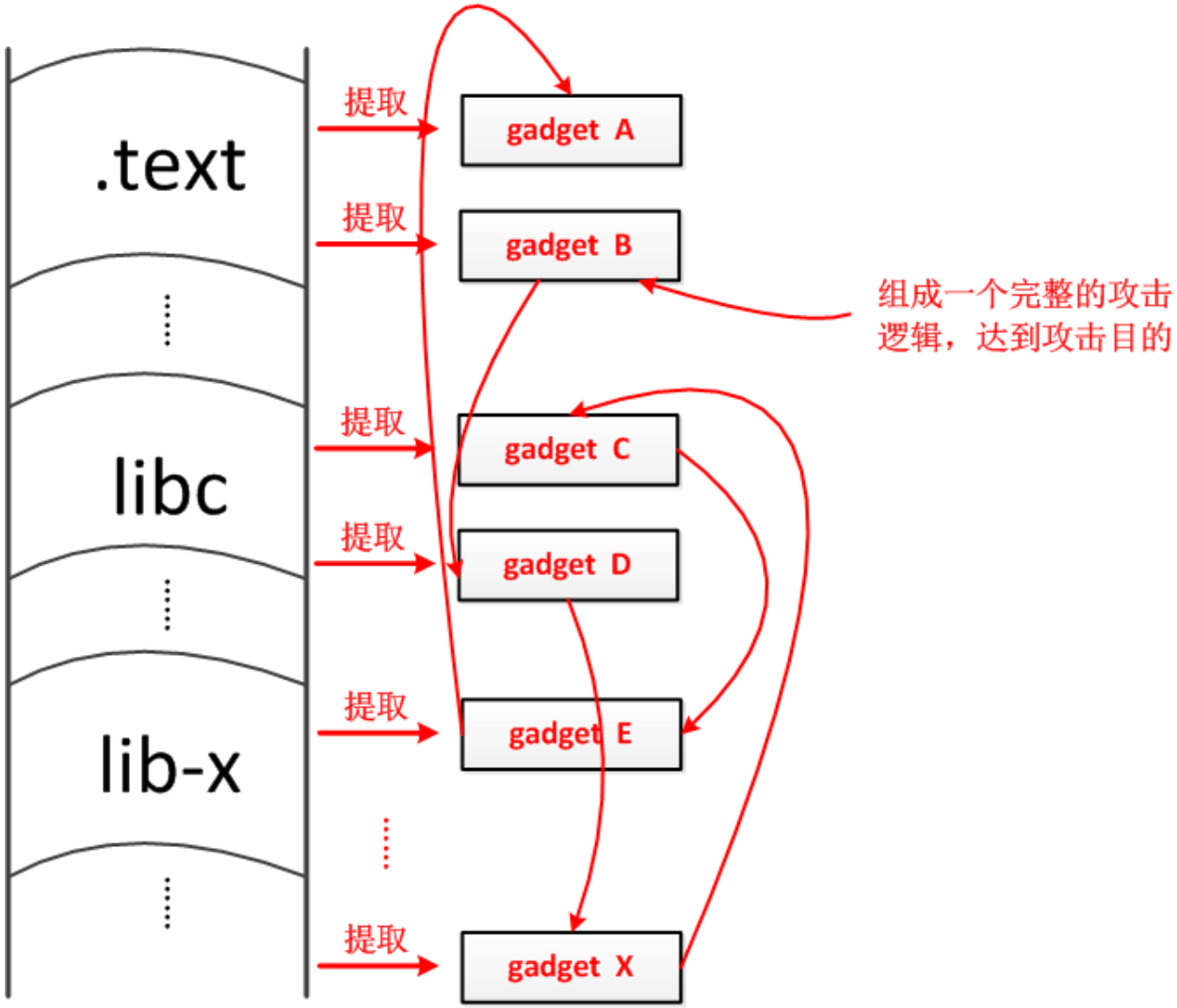
- 缓冲区溢出攻击变体
  - 不需要在缓冲区中植入代码
  - 只需修改栈帧的返回地址
  - 将漏洞函数返回到GOT和PLT地址空间中的函数
- 利用GOT表和PLT表
  - 不破坏W⊕X

ROP攻击

- ROP
  - Return-oriented programming —— 返回导向编程
  - 它借用libc代码段，其他库代码段和可执行程序自身代码段里面的多个retq前的一段指令拼凑成一段有效的逻辑（gadget），从而达到攻击的目标
  - 由一连串的gadgets组成
  - 不需要在缓冲区中植入代码
  - 只需修改栈帧的返回地址
- 利用程序链接库和可执行程序自己代码
  - 不破坏W⊕X

ROP攻击

- ROP攻击原理



return-to-address攻击总结

- 缓冲区溢出攻击变体
- 不需要在缓冲区中植入代码
- 只需修改栈帧的返回地址
- 将漏洞函数返回到确定的地址中执行
- 不破坏W⊕X

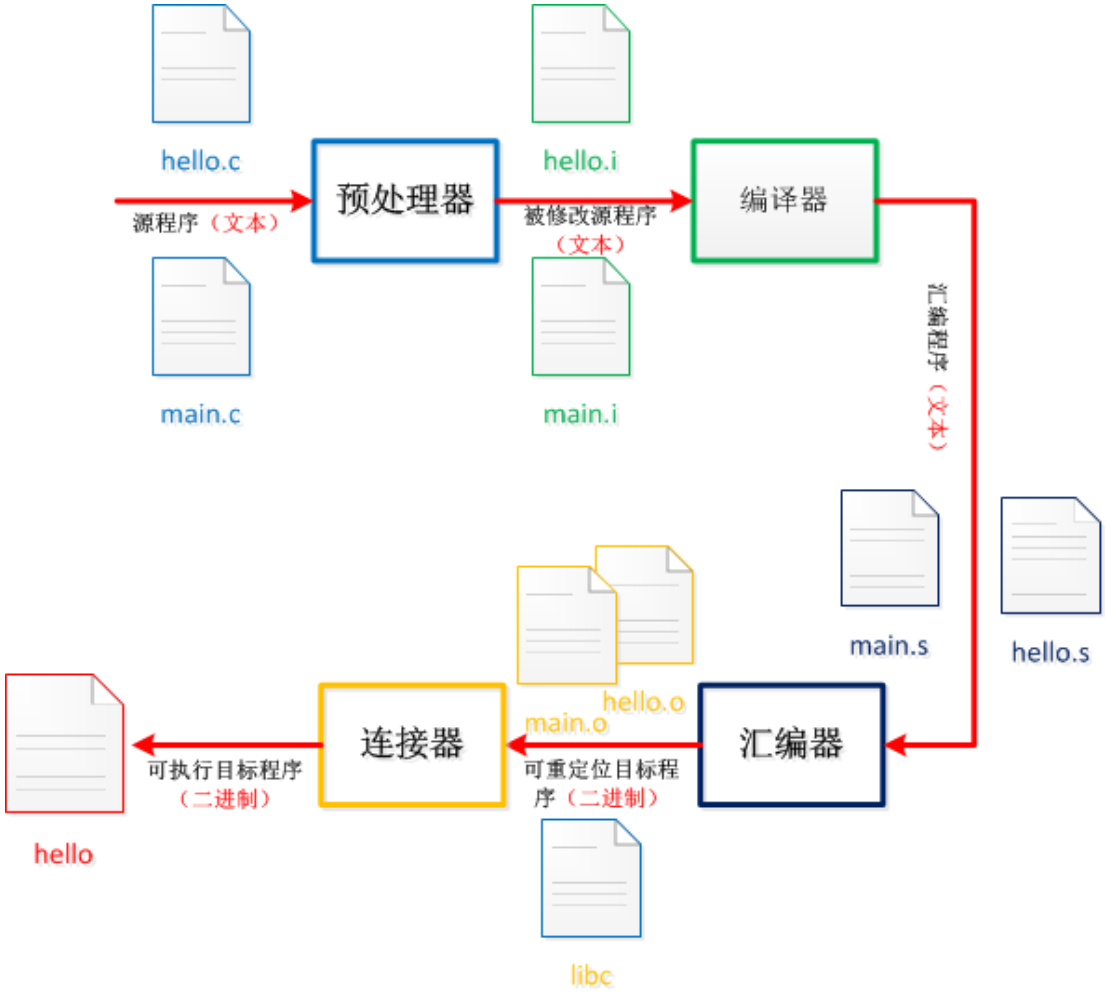
课后思考&动手

- 写一个简单的漏洞攻击程序
- 调研下漏洞扫描，利用工具

目录

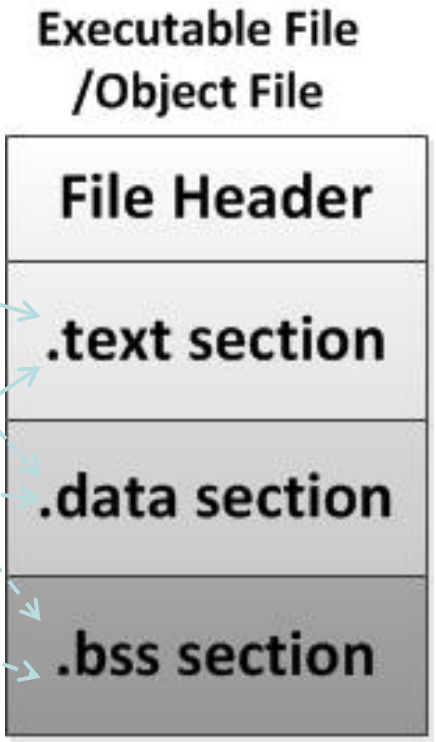
源程序到可执行文件

- 源程序→可执行文件过程



C代码变量存储分类

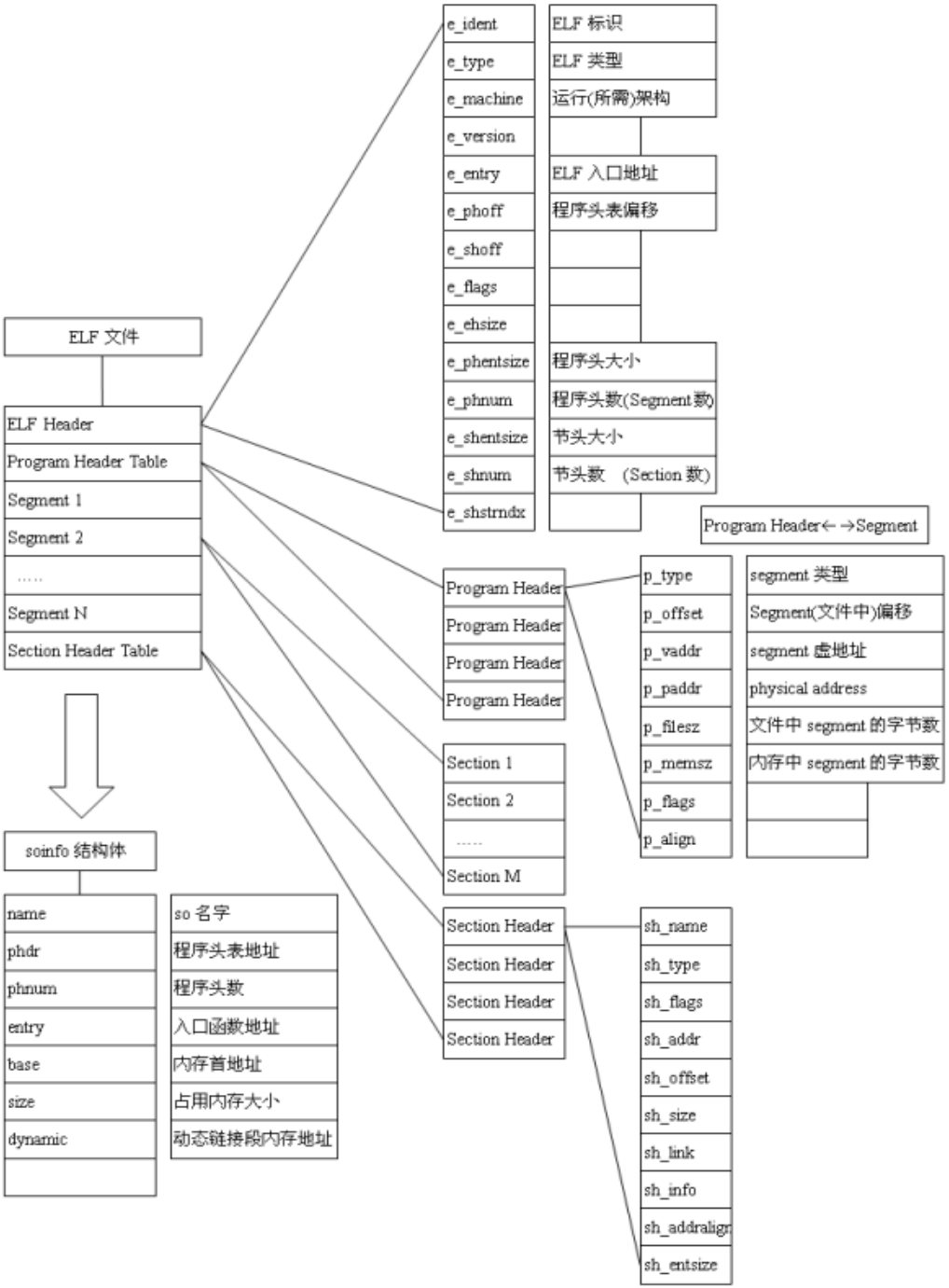
```
int global_init_var = 84;
int global_uninit_var;
void func1(int i)
{
   printf( "%d\n" ,i);
}
int main(void)
{
    static int static_var =85;
    static int static_var2;
     int a = 1;
     int b;
     func1(static_var +static_var2+a);
     return 0;
}
```



Linux可执行文件组织形式

- ELF — Executable and Linking Format三种文件格式
  - 可重定位文件（Relocatable File）
    - 包含适合于与其他目标文件链接来创建可执行文件或者共享目标文件的代码和数
  - 可执行文件（Executable File）
    - 包含适合于执行的一个程序，此文件规定了 exec() 如何创建一个程序的进程映像
  - 共享目标文件（Shared Object File）
    - 包含可在两种上下文中链接的代码和数据。首先链接编辑器可以将它和其它可重定位文件和共享目标文件一起处理，生成另外一个目标文件。其次，动态链接器（Dynamic Linker）可能将它与某个可执行文件以及其它共享目标一起组合，创建进程映像。

ELF结构


ELF 文件
ELF Header
Program Header Table
Segment 1
Segment 2
.....
Segment N
Section Header Table
e_ident
e_type
e_machine
e_version
e_entry
e_phoff
e_shoff
e_flags
e_ehsize
e_phentsize
e_phnum
e_shentsize
e_shnum
e_shstrndx
ELF 标识
ELF 类型
运行(所需)架构
ELF 入口地址
程序头表偏移
程序头大小
程序头数(Segment数)
节头大小
节头数 (Section 数)
Program Header← →Segment
Program Header
Program Header
Program Header
Program Header
p_type
p_offset
p_vaddr
p_paddr
p_filesz
p_memsz
p_flags
p_align
segment 类型
Segment(文件中)偏移
segment 虚地址
physical address
文件中 segment 的字节数
内存中 segment 的字节数
Section 1
Section 2
.....
Section M
Section Header
Section Header
Section Header
Section Header
sh_name
sh_type
sh_flags
sh_addr
sh_offset
sh_size
sh_link
sh_info
sh_addralign
sh_entsize
soinfo 结构体
name
phdr
phnum
entry
base
size
dynamic
so 名字
程序头表地址
程序头数
入口函数地址
内存首地址
占用内存大小
动态链接段内存地址

ELF目标文件格式

- 文件开始处是一个ELF 头部（ELF Header），用来描述整个文件的组织
- 节区部分包含链接视图的大量信息：指令、数据、符号表、重定位信息等等
- 程序头部表（Program Header Table），告诉系统如何创建进程映像
- 节区头部表（Section Heade Table）包含了描述文件节区的信息

<table>
<tr><th>链接视图</th><th>执行视图</th></tr>
<tr><td>ELF 头部</td><td>ELF 头部</td></tr>
<tr><td>程序头部表（可选）</td><td>程序头部表</td></tr>
<tr><td>节区 1</td><td rowspan="2">段 1</td></tr>
<tr><td>...</td></tr>
<tr><td>节区 n</td><td rowspan="2">段 2</td></tr>
<tr><td>...</td></tr>
<tr><td>...</td><td>...</td></tr>
<tr><td>节区头部表</td><td>节区头部表（可选）</td></tr>
</table>

ELF头部的数据结构

```
typedef struct
{
  unsigned char e_ident[EI_NIDENT];     /* 魔数和相关信息 */
  Elf32_Half    e_type;                 /* 目标文件类型 */
  Elf32_Half    e_machine;              /* 硬件体系 */
  Elf32_Word    e_version;              /* 目标文件版本 */
  Elf32_Addr    e_entry;                        /* 程序进入点 */
  Elf32_Off     e_phoff;                        /* 程序头部偏移量 */
  Elf32_Off     e_shoff;                        /* 节头部偏移量 */
  Elf32_Word    e_flags;                        /* 处理器特定标志 */
  Elf32_Half    e_ehsize;                       /* ELF头部长度 */
  Elf32_Half    e_phentsize;                     /* 程序头部中一个条目的长度 */
  Elf32_Half    e_phnum;                        /* 程序头部条目个数  */
  Elf32_Half    e_shentsize;                    /* 节头部中一个条目的长度 */
  Elf32_Half    e_shnum;                        /* 节头部条目个数 */
  Elf32_Half    e_shstrndx;                     /* 节头部字符表索引 */
} Elf32_Ehdr;
```

ELF头部的数据结构

EI_MAG0 到 EI_MAG3	EI_MAG0	0x7f	e_ident[EI_MAG0]
	EI_MAG1	'E'	e_ident[EI_MAG1]
	EI_MAG2	'L'	e_ident[EI_MAG2]
	EI_MAG3	'F'	e_ident[EI_MAG3]

EI_CLASS	ELFCLASSNONE	0	非法类别
	ELFCLASS32	1	32 位目标
	ELFCLASS64	2	64 位目标

EI_DATA	ELFDATA2LSB	1	高位在前
	ELFDATA2MSB	2	低位在前

e_type	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）
	ET_LOPROC	0xff00	特定处理器文件
	ET_HIPROC	0xffff	特定处理器文件

e_machine	EM_NONE	0	未指定
	EM_M32	1	AT&T WE 32100
	EM_SPARC	2	SPARC
	EM_386	3	Intel 80386
	EM_68K	4	Motorola 68000
	EM_88K	5	Motorola 88000
	EM_860	7	Intel 80860
	EM_MIPS	8	MIPS RS3000

readelf命令读取ELF头部信息

```
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:               0x80483cc
  Start of program headers:          52 (bytes into file)
  Start of section headers:          14936 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         6
  Size of section headers:           40 (bytes)
  Number of section headers:         34
  Section header string table index: 31
```

程序头部（Program Header）

```
typedef struct {
    Elf32_Word  p_type;          /* 段类型 */
    Elf32_Off   p_offset;        /* 段位置相对于文件开始处的偏移量 */
    Elf32_Addr  p_vaddr;         /* 段在内存中的地址 */
    Elf32_Addr  p_paddr;         /* 段的物理地址 */
    Elf32_Word  p_filesz;        /* 段在文件中的长度 */
    Elf32_Word  p_memsz;         /* 段在内存中的长度 */
    Elf32_Word  p_flags;         /* 段的标记 */
    Elf32_Word  p_align;         /* 段在内存中对齐标记 */
} Elf32_Phdr;
```

程序头部（Program Header）输出

```
Program Headers:
Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
PHDR           0x000034 0x08048034 0x08048034 0x000c0 0x000c0 R E 0x4
INTERP         0x0000f4 0x080480f4 0x080480f4 0x00013 0x00013 R   0x1
    [Requesting program interpreter: /lib/ld-linux.so.2]
                LOAD           0x000000 0x08048000 0x08048000 0x00684 0x00684 R E 0x1000
                LOAD           0x000684 0x08049684 0x08049684 0x00118 0x00130 RW  0x1000
                DYNAMIC        0x000690 0x08049690 0x08049690 0x000c8 0x000c8 RW  0x4
                NOTE           0x000108 0x08048108 0x08048108 0x00020 0x00020 R   0x4
                Section to Segment mapping:
 Segment Sections...
  00
  01     .interp
  02     .interp .note.ABI-tag .hash .dynsym .dynstr .gnu.version
        .gnu.version_r .rel.dyn .rel.plt .init .plt .text .fini .rodata .eh_frame
  03     .data .dynamic .ctors .dtors .jcr .got .bss
  04     .dynamic
  05     .note.ABI-tag
```

节区头部（Section Header）

```
typedef struct{
Elf32_Word sh_name;
Elf32_Word sh_type;
Elf32_Word sh_flags;
Elf32_Addr sh_addr;
Elf32_Off sh_offset;
Elf32_Word sh_size;
Elf32_Word sh_link;
Elf32_Word sh_info;
Elf32_Word sh_addralign;
Elf32_Word sh_entsize;
}Elf32_Shdr;
```

Section Headers输出

Section Headers:

```
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .interp           PROGBITS        080480f4 0000f4 000013 00   A  0   0  1
  [ 2] .note.ABI-tag     NOTE            08048108 000108 000020 00   A  0   0  4
  [ 3] .hash             HASH            08048128 000128 000040 04   A  4   0  4
  [ 4] .dynsym           DYNSYM          08048168 000168 0000b0 10   A  5   1  4
  [ 5] .dynstr           STRTAB          08048218 000218 00007b 00   A  0   0  1
  [ 6] .gnu.version      VERSYM          08048294 000294 000016 02   A  4   0  2
  [ 7] .gnu.version_r    VERNEED         080482ac 0002ac 000030 00   A  5   1  4
  [ 8] .rel.dyn          REL             080482dc 0002dc 000008 08   A  4   0  4
  [ 9] .rel.plt          REL             080482e4 0002e4 000040 08   A  4   b  4
  [10] .init             PROGBITS        08048324 000324 000017 00  AX  0   0  4
  [11] .plt              PROGBITS        0804833c 00033c 000090 04  AX  0   0  4
  [12] .text             PROGBITS        080483cc 0003cc 0001f8 00  AX  0   0  4
  [13] .fini             PROGBITS        080485c4 0005c4 00001b 00  AX  0   0  4
  [14] .rodata           PROGBITS        080485e0 0005e0 00009f 00   A  0   0 32
  [15] .eh_frame         PROGBITS        08048680 000680 000004 00   A  0   0  4
  [16] .data             PROGBITS        08049684 000684 00000c 00  WA  0   0  4
  [17] .dynamic          DYNAMIC         08049690 000690 0000c8 08  WA  5   0  4
  [18] .ctors            PROGBITS        08049758 000758 000008 00  WA  0   0  4
  [19] .dtors            PROGBITS        08049760 000760 000008 00  WA  0   0  4
  [20] .jcr              PROGBITS        08049768 000768 000004 00  WA  0   0  4
  [21] .got              PROGBITS        0804976c 00076c 000030 04  WA  0   0  4
  [22] .bss              NOBITS          0804979c 00079c 000018 00  WA  0   0  4
  [23] .comment          PROGBITS        00000000 00079c 000132 00      0   0  1
  [24] .debug_aranges    PROGBITS        00000000 0008d0 000098 00      0   0  8
  [25] .debug_pubnames   PROGBITS        00000000 000968 000040 00      0   0  1
  [26] .debug_info       PROGBITS        00000000 0009a8 001cc6 00      0   0  1
  [27] .debug_abbrev     PROGBITS        00000000 00266e 0002cc 00      0   0  1
  [28] .debug_line       PROGBITS        00000000 00293a 0003dc 00      0   0  1
  [29] .debug_frame      PROGBITS        00000000 002d18 000048 00      0   0  4
  [30] .debug_str        PROGBITS        00000000 002d60 000bcd 01  MS  0   0  1
  [31] .shstrtab         STRTAB          00000000 00392d 00012b 00      0   0  1
  [32] .symtab           SYMTAB          00000000 003fa8 000740 10     33  56  4
  [33] .strtab           STRTAB          00000000 0046e8 000467 00      0   0  1
```

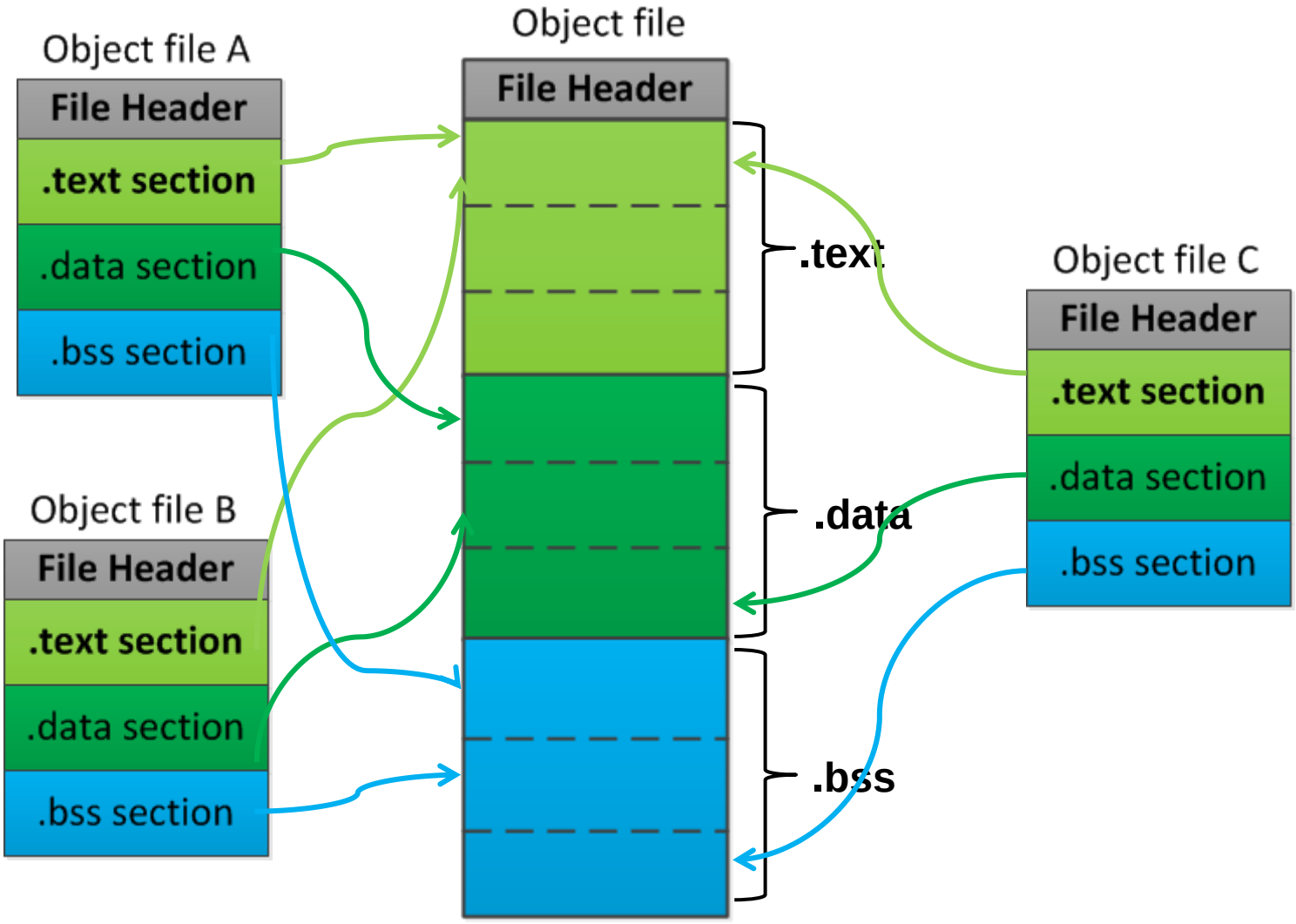
其他重要的段

- 代码段（.text）：存放程序执行代码
- 数据段（.data）：存放初始化的全局静态变量和局部静态变量。只读
- 数据段（.rodata）:存放的是只读数据，一般是程序里面的只读变量（如const修饰的变量）和字符串常量。
- BSS段（.bss）：存放未初始化的全局变量和局部静态变量。
- .debug:调试信息。一般发布程序的时候去掉这部分。
- .dynamic:动态链接信息。
- .init:程序初始化段
- .fini:程序终结代码段。

其他重要信息表

- 字符串表
- 符号表
- 重定位表
- ……

静态链接


Object file A
File Header
.text section
.data section
.bss section
Object file B
File Header
.text section
.data section
.bss section
Object file
File Header
.text
.data
.bss
Object file C
File Header
.text section
.data section
.bss section

全局偏移表——GOT, Global Offset Table

- 位置独立的代码一般不能包含绝对的虚拟地址符号表
- 全局偏移表中包含其重定位项中要求的信息
- 动态链接器要处理重定位项
- 动态链接器确定相关的符号取值，计算其绝对地址，并将相应的内存表格项目设置为正确的数值
- extern Elf32_Addr _GLOBAL_OFFSET_TABLE[];

过程链接表——PLT，Procedure Link Table

绝对过程链接表

```
.PLT0: pushl got_plus_4
jmp *got_plus_8
nop; nop
nop; nop
.PLT1: jmp *name1_in_GOT
pushl $offset@PC
.PLT2: jmp *name2_in_GOT
pushl $offset
jmp .PLT0@PC
…
```

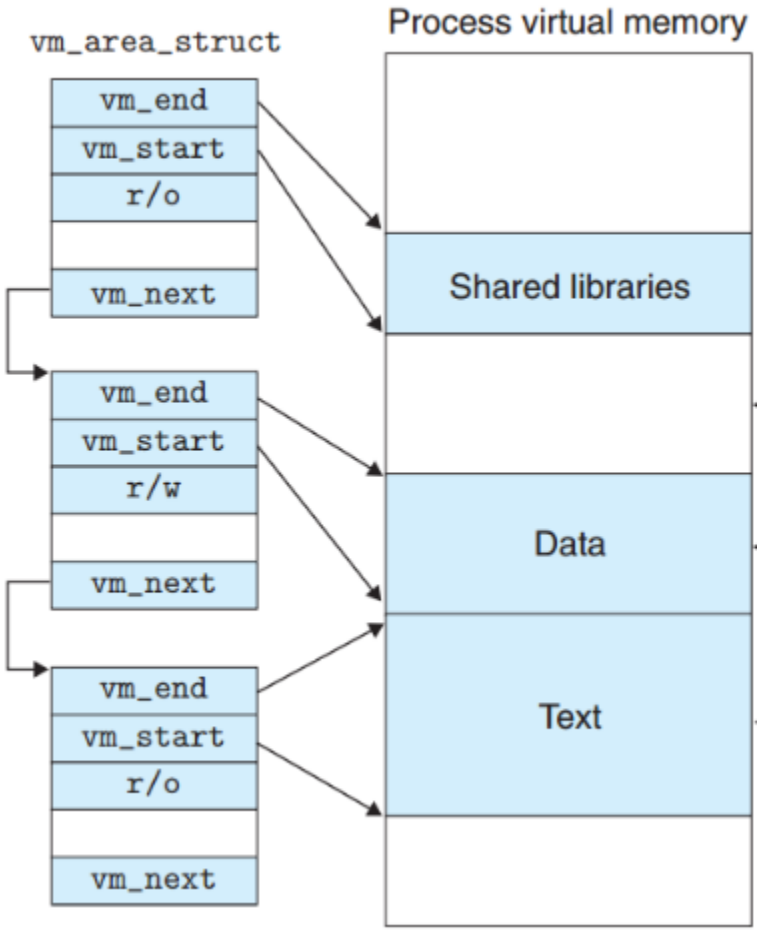
位置独立的过程链接表

```
.PLT0: pushl 4(%ebx)
jmp *8(%ebx)
nop; nop
nop; nop
.PLT1: jmp *name1@GOT(%ebx)
pushl $offset
jmp .PLT0@PC
.PLT2: jmp *name2@GOT(%ebx)
pushl $offset
jmp .PLT0@PC
…
```

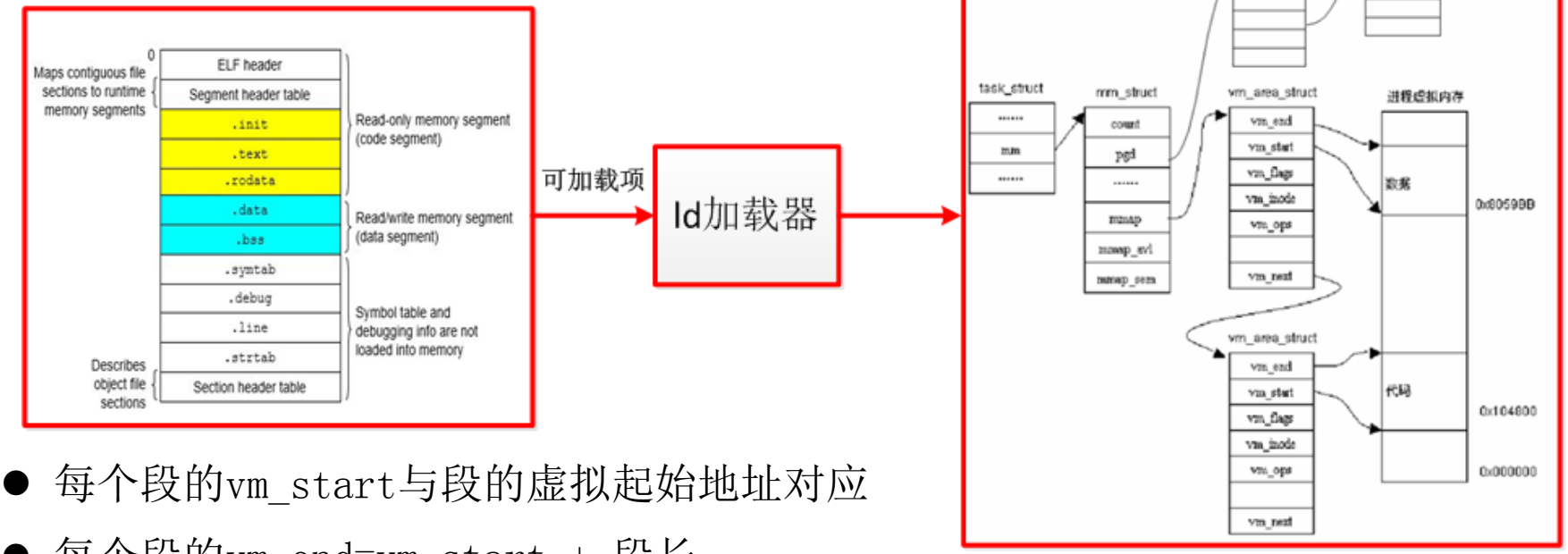
动态链接——GOT,PLT

程序调用库函数过程

可执行程序内存映射


vm_area_struct
Process virtual memory
vm_end
vm_start
r/o
vm_next
vm_end
vm_start
r/w
vm_next
vm_end
vm_start
r/o
vm_next
Shared libraries
Data
Text

可执行程序加载过程及内存组织



- 每个段的vm_start与段的虚拟起始地址对应
- 每个段的vm_end=vm_start + 段长
- 每个段的vm_flags与段的flags对应

cat程序在linux系统的地址空间映射信息

```
root@ubuntu:~# cat /proc/self/maps
08048000-08053000 r-xp 00000000 08:01 1054472    /bin/cat
08053000-08054000 r--p 0000a000 08:01 1054472    /bin/cat
08054000-08055000 rw-p 0000b000 08:01 1054472    /bin/cat
08055000-08076000 rw-p 00000000 00:00 0          [heap]
b7abc000-b7c21000 r--p 001c8000 08:01 922740     /usr/lib/locale/locale-archive
b7c21000-b7e21000 r--p 00000000 08:01 922740     /usr/lib/locale/locale-archive
b7e21000-b7e22000 rw-p 00000000 00:00 0
b7e22000-b7fc5000 r-xp 00000000 08:01 660181     /lib/i386-linux-gnu/libc-2.15.so
b7fc5000-b7fc7000 r--p 001a3000 08:01 660181     /lib/i386-linux-gnu/libc-2.15.so
b7fc7000-b7fc8000 rw-p 001a5000 08:01 660181     /lib/i386-linux-gnu/libc-2.15.so
b7fc8000-b7fcb000 rw-p 00000000 00:00 0
b7fda000-b7fdb000 r--p 005e0000 08:01 922740     /usr/lib/locale/locale-archive
b7fdb000-b7fdd000 rw-p 00000000 00:00 0
b7fdd000-b7fde000 r-xp 00000000 00:00 0          [vdso]
b7fde000-b7ffe000 r-xp 00000000 08:01 661146     /lib/i386-linux-gnu/ld-2.15.so
b7ffe000-b7fff000 r--p 0001f000 08:01 661146     /lib/i386-linux-gnu/ld-2.15.so
b7fff000-b8000000 rw-p 00020000 08:01 661146     /lib/i386-linux-gnu/ld-2.15.so
bffdf000-c0000000 rw-p 00000000 00:00 0          [stack]
root@ubuntu:~#
```

课后思考&动手

- 通过elf工具读取ELF内容，如elf头部信息，段，节信息
- 查看进程地址空间分配情况

目录

内存防护技术的四个层次

编译器

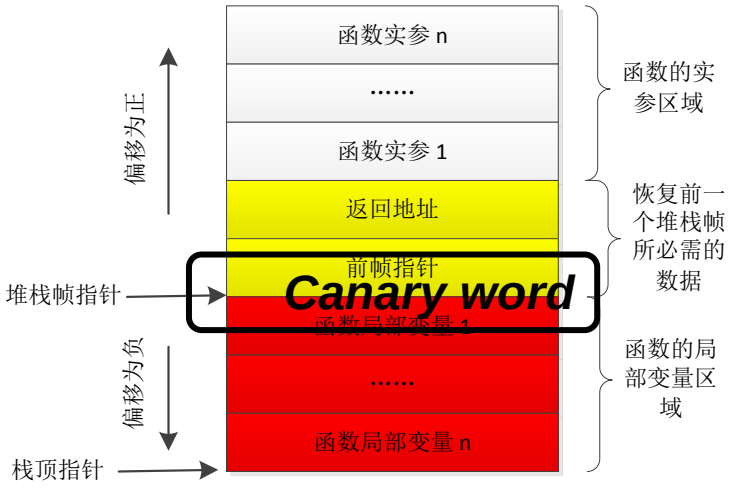
链接库

操作系统

硬件

编译器安全技术

- 溢出检测技术
  - “Canaries” 探测技术
  - 在函数栈缓冲区和控制信息（如 EBP 等）间插入 canary word
  - 函数返回时检查函数栈中的canary word是否被修改



编译器安全技术

- 溢出检测技术
  - “canary word”形式：
    - Terminator canaries
    - 0x00000000 ，C字符串遇NULL结束
    - 0x000aff0d字符串作为 canary word，NULL（0x00），CR（0x0d），LF（0x0a）和 EOF（0xff）四个字 符，0x00 使 strcpy() 结束，0x0a 会使 gets() 结束
    - 缺陷：canary word固定，容易在shellcode构造回去

编译器安全技术

- Random canaries
  - 随机产生Canary word ，程序初始化时产生，保存到特定地方
  - 优点：不同程序canary word不同
  - 缺陷：同个程序canary word 相同，最终会被猜测出
- Random XOR canaries
  - 由一个随机数和函数栈中的所有控制信息、返回地址通过异或运算得到
  - 优点：不易伪造，数栈中的 canaries 或者任何控制信息、返回地址被修改就都能被检测

编译器安全技术

- 主流编译器（GCC等）栈保护技术
  - Stack Guard
  - Stack-smashing Protection(SSP，ProPolice)
  - Canaries 探测作为它们主要的保护技术

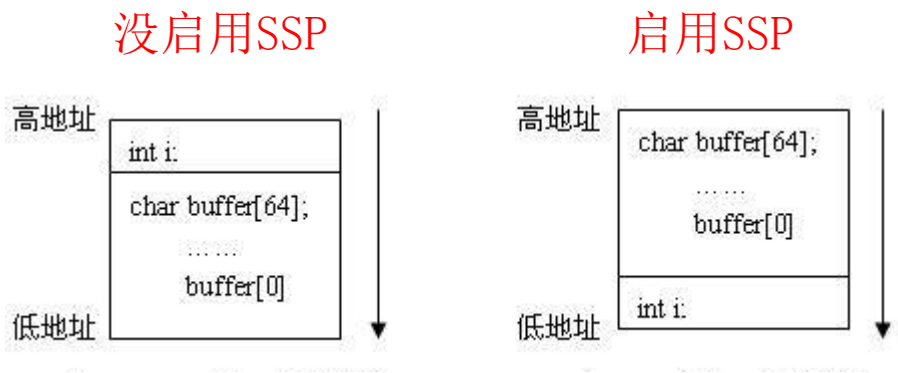
编译器安全技术

- Stack-smashing Protection（SSP）
  - 保护返回地址
  - 保护栈EBP等控制信息
  - 局部变量中的数组放在函数栈的高地址，其他变量放在低地址
  - 通过溢出一个数组来修改其他变量（如一个函数指针）变得困难

编译器安全技术

- 启用Stack-smashing Protection函数栈变化

```
int main()
{
int i;
 char buffer[64];
 i = 1;
buffer[0] = 'a';
 return 0;
}
```



编译器安全技术

- GCC （4.1） 栈保护有关的编译选项
  - -fstack-protector
  - 启用堆栈保护，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

编译器安全技术

- 溢出检查——Stackshield
  - 创建一个特别的堆栈用来储存函数返回地址的一份拷贝
  - 受保护的函数的开头和结尾分别增 加一段代码，开头处的代码用来将函数返回地址拷贝到一个特殊的表中，而结尾处的代码用来将返回地址从表中拷贝回堆栈

编译器安全技术

- 边界检查
  - 运行时对（数组、指针）边界进行检查
  - 描述了每个分配内存块的中央数据块
  - 包含了指针以及描述它们指向区域的额外数据的胖指针

链接库保护技术

- Formatguard
  - 是Glibc的补丁，遵循GPL
  - 它使用特殊的CPP（gcc预编译程序）宏取代原有的*printf()的参数统计方式，比较传递给*printf的参数的个数和格式窜的个数
  - 格式窜的个数大于实际参数的个数，判定为攻击行为，向syslogd发送消息并终止进程
  - 缺陷：程序调用Glibc以外的库，formatguard就无法保护

链接库保护技术

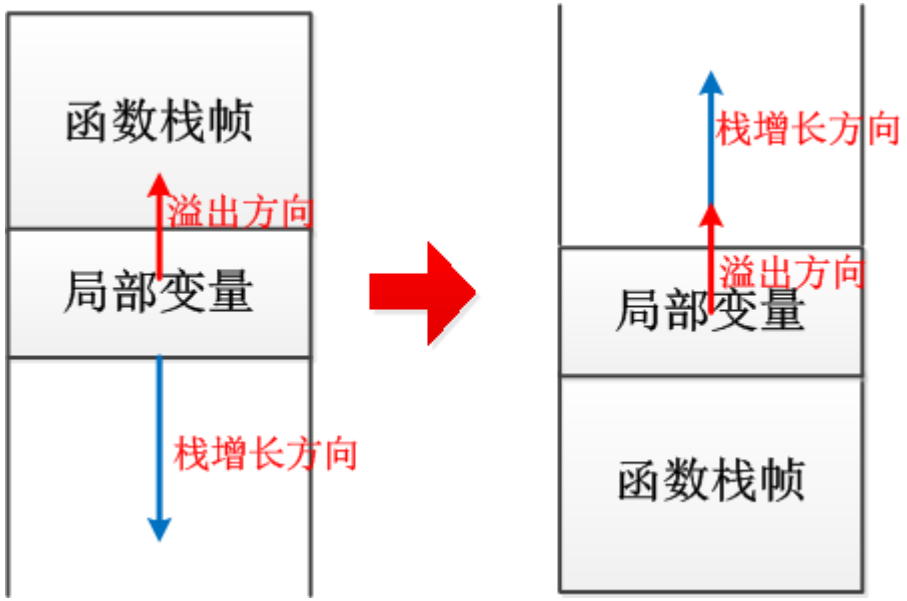
- Libsafe
  - 是一个动态链接库
  - 在标准的C库之前被加载
  - 主要加固gets()，strcpy()，strcat()，sprintf()……等容易发生安全问题的C函数
  - 针对stack smashing和 format string类型的攻击

链接库保护技术

- 安全库（函数）
  - 静态分配的缓冲区方法
  - 当缓冲区用完时，拒绝为缓冲区增加任何空间
  - 标准 C 库方法：
  - 标准 C strncpy/strncat 和 OpenBSD 的 strlcpy/strlcat
  - 动态分配的缓冲区方法
  - 当缓冲区用完时，动态地将缓冲区大小调整到更大的尺寸，直至用完所有内存
  - SafeStr
  - C++ std::string

操作系统安全技术

- 溢出保护
  - 改变栈的增长方向
  - Top to Down　　Down to Top
  - 溢出不破坏栈帧结构（返回地址）

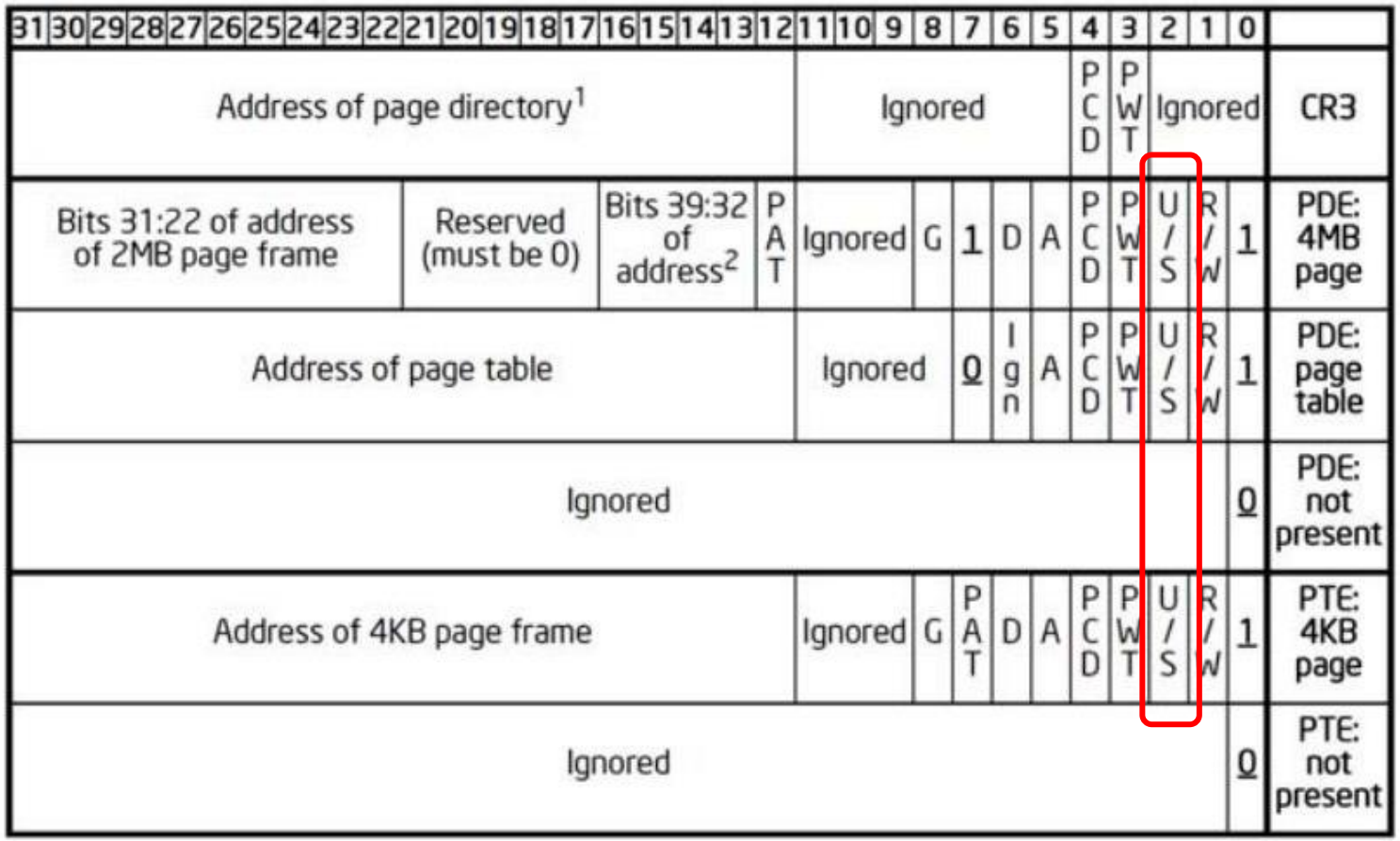


操作系统安全技术

- 不可执行保护（无直接硬件支持）
  - 基于页式管理实现
  - TLB划分成ITLB,DTLB
  - 重载页表U/S位

操作系统安全技术

- 不可执行保护（无直接硬件支持）



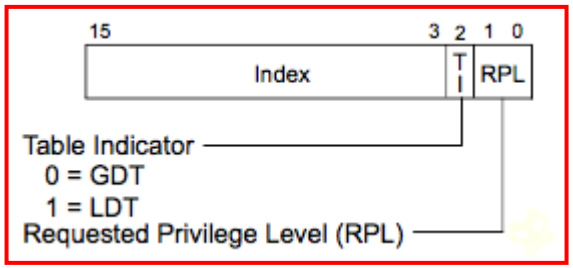
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Address of page directory[1]																				Ignored							PCD	PWT	Ignored			CR3
Bits 31:22 of address of 2MB page frame										Reserved (must be 0)				Bits 39:32 of address[2]					PAT	Ignored			G	1	D	A	PCD	PWT	U/S	R/W	1	PDE: 4MB page
Address of page table																				Ignored				0	Ign	A	PCD	PWT	U/S	R/W	1	PDE: page table
Ignored																															0	PDE: not present
Address of 4KB page frame																				Ignored			G	PAT	D	A	PCD	PWT	U/S	R/W	1	PTE: 4KB page
Ignored																															0	PTE: not present

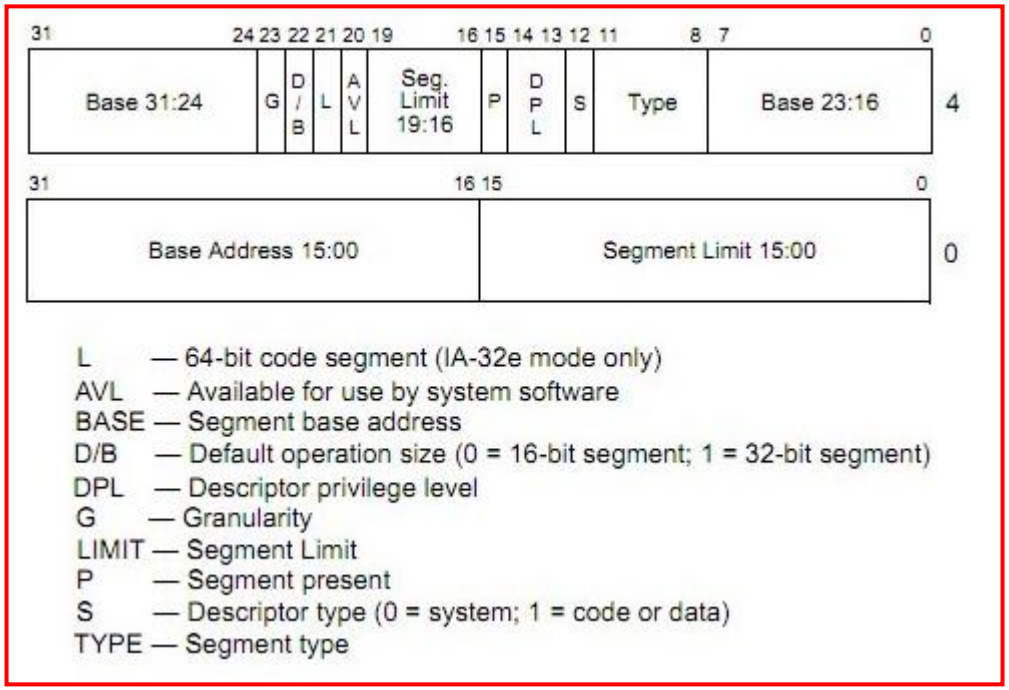
操作系统安全技术

- 不可执行保护（无直接硬件支持）
  - 访问规则
    - » Supervisor mode (levels 0-2): user or supervisor pages allowed (u/s==*)
    - » User mode (level 3): user only (u/s==1)
  - 重载U/S位　executable/non-executable status
    - » U/S=1　executable page
    - » U/S=0　non-executable page
  - 受保护页的PDE和PTE的U/S=S(0)
    - » 当访问到不可执行页时，产生页访问异常
    - » 查看异常原因
    - » 终止程序执行

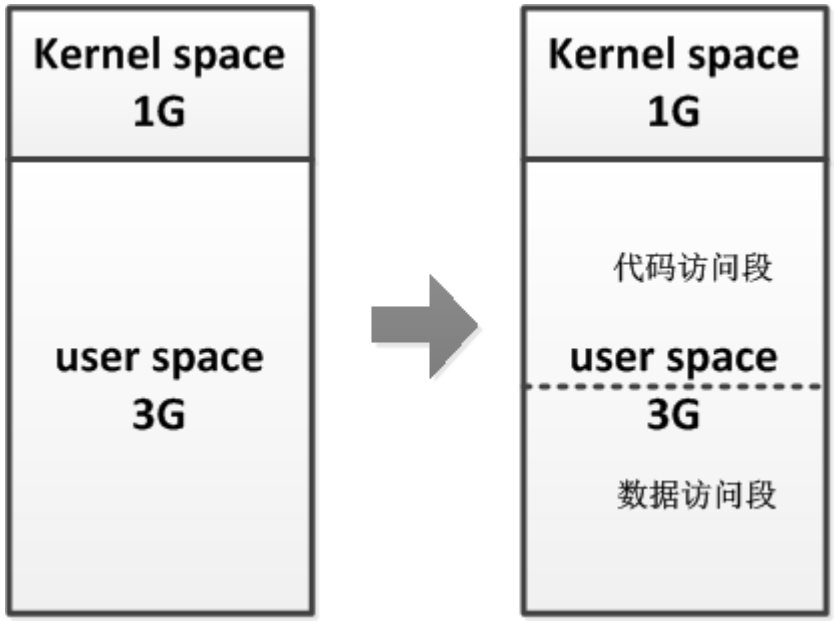
操作系统安全技术

- 不可执行保护（无直接硬件支持）





- 不可执行保护（无直接硬件支持）
  - 基于段式管理实现
  - 把用户空间3G划分出两段
  - 其中一段为数据访问段，另一段为代码访问段

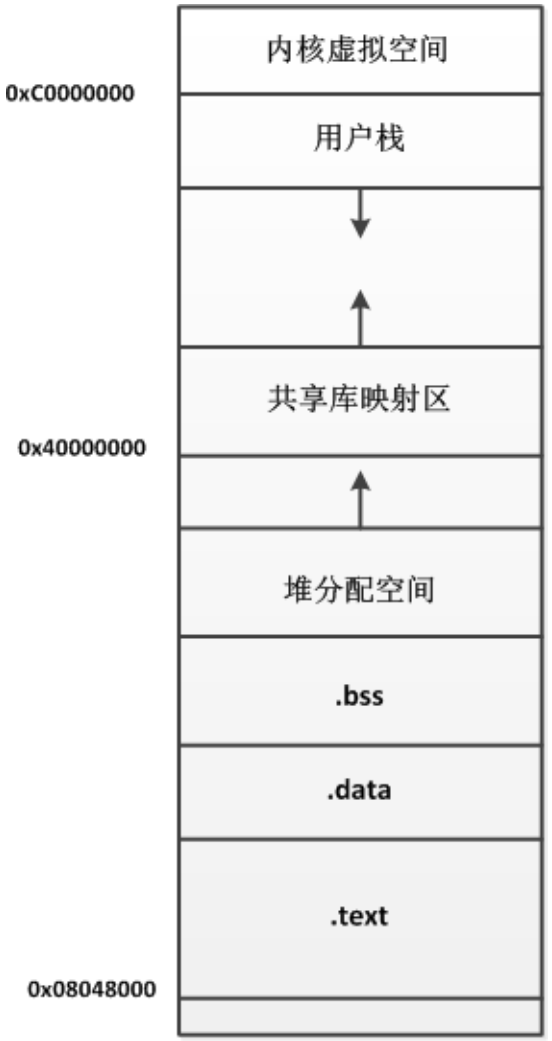


操作系统安全技术

- 不可执行保护（无直接硬件支持）
  - 主要代表
    - Solar designer’ s nonexec kernel patch
    - Solaris/SPARC nonexec-stack protection
    - kNoX
    - RSX
    - Exec shield
    - PaX

操作系统安全技术

- 空间固定分配弊端
  - 进程地址空间布局一致
  - 相同程序在同一平台的计算机中地址空间布局完全一致
  - 地址容易猜测，实施攻击难度低（ret2libc，ROP）



操作系统安全技术

- 地址空间固定分配地址空间分布

```
root@ubuntu:~# cat /proc/self/maps
08048000-08053000 r-xp 00000000 08:01 1054472    /bin/cat
08053000-08054000 r--p 0000a000 08:01 1054472    /bin/cat
08054000-08055000 rw-p 0000b000 08:01 1054472    /bin/cat
08055000-08076000 rw-p 00000000 00:00 0          [heap]
b7abc000-b7c21000 r--p 001c8000 08:01 922740     /usr/lib/locale/locale-archive
b7c21000-b7e21000 r--p 00000000 08:01 922740     /usr/lib/locale/locale-archive
b7e21000-b7e22000 rw-p 00000000 00:00 0
b7e22000-b7fc5000 r-xp 00000000 08:01 660181     /lib/i386-linux-gnu/libc-2.15.so
b7fc5000-b7fc7000 r--p 001a3000 08:01 660181     /lib/i386-linux-gnu/libc-2.15.so
b7fc7000-b7fc8000 rw-p 001a5000 08:01 660181     /lib/i386-linux-gnu/libc-2.15.so
b7fc8000-b7fcb000 rw-p 00000000 00:00 0
b7fda000-b7fdb000 r--p 005e0000 08:01 922740     /usr/lib/locale/locale-archive
b7fdb000-b7fdd000 rw-p 00000000 00:00 0
b7fdd000-b7fde000 r-xp 00000000 00:00 0          [vdso]
b7fde000-b7ffe000 r-xp 00000000 08:01 661146     /lib/i386-linux-gnu/ld-2.15.so
b7ffe000-b7fff000 r--p 0001f000 08:01 661146     /lib/i386-linux-gnu/ld-2.15.so
b7fff000-b8000000 rw-p 00020000 08:01 661146     /lib/i386-linux-gnu/ld-2.15.so
bffdf000-c0000000 rw-p 00000000 00:00 0          [stack]
root@ubuntu:~#
```

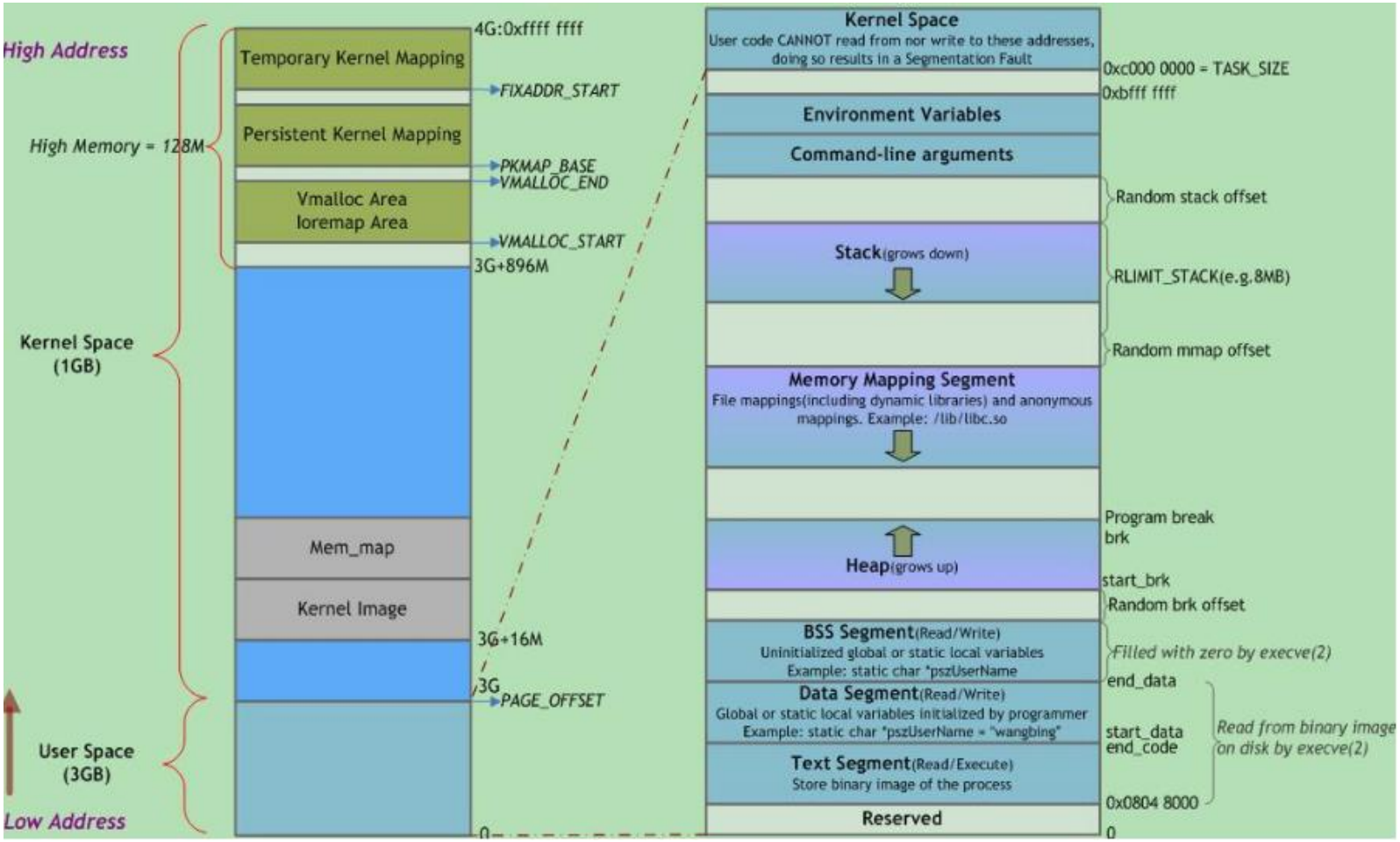
第一次运行

```
root@ubuntu:~# cat /proc/self/maps
08048000-08053000 r-xp 00000000 08:01 1054472    /bin/cat
08053000-08054000 r--p 0000a000 08:01 1054472    /bin/cat
08054000-08055000 rw-p 0000b000 08:01 1054472    /bin/cat
08055000-08076000 rw-p 00000000 00:00 0          [heap]
b7abc000-b7c21000 r--p 001c8000 08:01 922740     /usr/lib/locale/locale-archive
b7c21000-b7e21000 r--p 00000000 08:01 922740     /usr/lib/locale/locale-archive
b7e21000-b7e22000 rw-p 00000000 00:00 0
b7e22000-b7fc5000 r-xp 00000000 08:01 660181     /lib/i386-linux-gnu/libc-2.15.so
b7fc5000-b7fc7000 r--p 001a3000 08:01 660181     /lib/i386-linux-gnu/libc-2.15.so
b7fc7000-b7fc8000 rw-p 001a5000 08:01 660181     /lib/i386-linux-gnu/libc-2.15.so
b7fc8000-b7fcb000 rw-p 00000000 00:00 0
b7fda000-b7fdb000 r--p 005e0000 08:01 922740     /usr/lib/locale/locale-archive
b7fdb000-b7fdd000 rw-p 00000000 00:00 0
b7fdd000-b7fde000 r-xp 00000000 00:00 0          [vdso]
b7fde000-b7ffe000 r-xp 00000000 08:01 661146     /lib/i386-linux-gnu/ld-2.15.so
b7ffe000-b7fff000 r--p 0001f000 08:01 661146     /lib/i386-linux-gnu/ld-2.15.so
b7fff000-b8000000 rw-p 00020000 08:01 661146     /lib/i386-linux-gnu/ld-2.15.so
bffdf000-c0000000 rw-p 00000000 00:00 0          [stack]
root@ubuntu:~#
```

第二次运行

操作系统安全技术

- 地址空间随机化分配（ASLR）



操作系统安全技术

- 地址空间随机化分配
  - ASLR——address space layout randomization
  - 改变传统地址空间固定分配方式
  - 每个地址区域起始地址=固定基值+/-随机偏移值
  - 随机化关键因素
  - 随机化策略
  - 随机值的质量

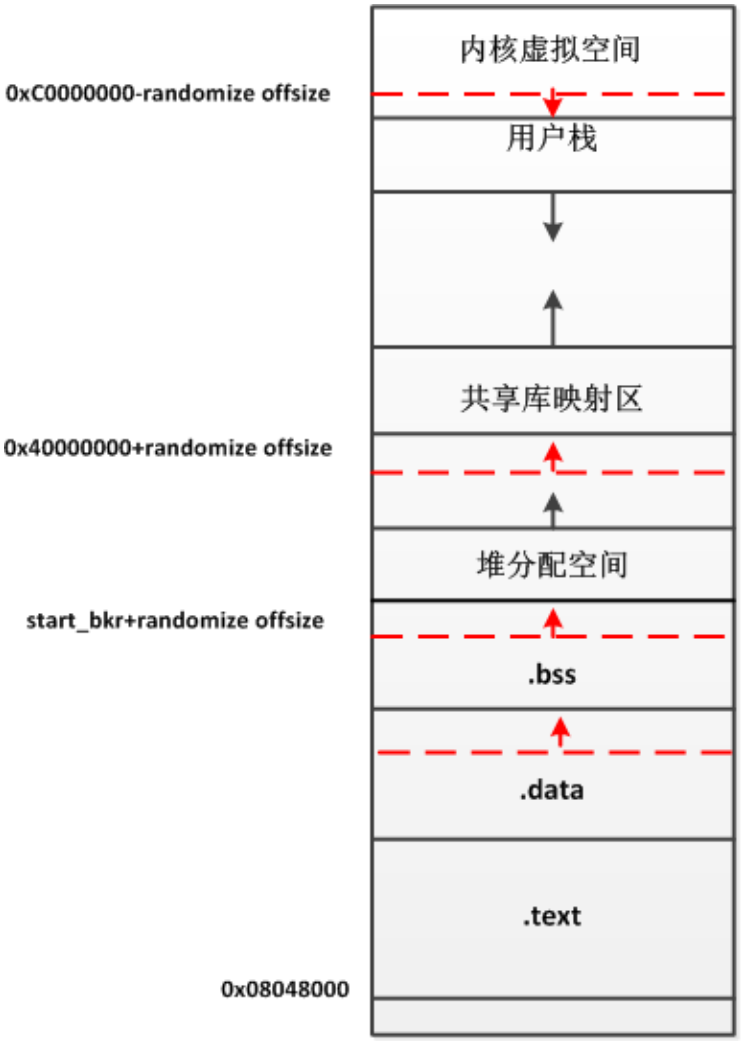
操作系统安全技术

- 地址空间随机化分配地址空间分布

```
root@ubuntu:/home/abang# cat /proc/self/maps
08048000-08053000 r-xp 00000000 08:01 1054472    /bin/cat
08053000-08054000 r--p 0000a000 08:01 1054472    /bin/cat
08054000-08055000 rw-p 0000b000 08:01 1054472    /bin/cat
0925c000-0927d000 rw-p 00000000 00:00 0          [heap]
b7241000-b73a6000 r--p 001c8000 08:01 922740     /usr/lib/locale/locale-archive
b73a6000-b75a6000 r--p 00000000 08:01 922740     /usr/lib/locale/locale-archive
b75a6000-b75a7000 rw-p 00000000 00:00 0
b75a7000-b774a000 r-xp 00000000 08:01 660181     /lib/i386-linux-gnu/libc-2.15.so
b774a000-b774c000 r--p 001a3000 08:01 660181     /lib/i386-linux-gnu/libc-2.15.so
b774c000-b774d000 rw-p 001a5000 08:01 660181     /lib/i386-linux-gnu/libc-2.15.so
b774d000-b7750000 rw-p 00000000 00:00 0
b775f000-b7760000 r--p 005e0000 08:01 922740     /usr/lib/locale/locale-archive
b7760000-b7762000 rw-p 00000000 00:00 0
b7762000-b7763000 r-xp 00000000 00:00 0          [vdso]
b7763000-b7783000 r-xp 00000000 08:01 661146     /lib/i386-linux-gnu/ld-2.15.so
b7783000-b7784000 r--p 0001f000 08:01 661146     /lib/i386-linux-gnu/ld-2.15.so
b7784000-b7785000 rw-p 00020000 08:01 661146     /lib/i386-linux-gnu/ld-2.15.so
bfac9000-bfaea000 rw-p 00000000 00:00 0          [stack]
root@ubuntu:/home/abang# echo 0 > /proc/sys/kernel/randomize_va_space
root@ubuntu:/home/abang# cat /proc/self/maps
08048000-08053000 r-xp 00000000 08:01 1054472    /bin/cat
08053000-08054000 r--p 0000a000 08:01 1054472    /bin/cat
08054000-08055000 rw-p 0000b000 08:01 1054472    /bin/cat
08055000-08076000 rw-p 00000000 00:00 0          [heap]
b7abc000-b7c21000 r--p 001c8000 08:01 922740     /usr/lib/locale/locale-archive
b7c21000-b7e21000 r--p 00000000 08:01 922740     /usr/lib/locale/locale-archive
b7e21000-b7e22000 rw-p 00000000 00:00 0
b7e22000-b7fc5000 r-xp 00000000 08:01 660181     /lib/i386-linux-gnu/libc-2.15.so
b7fc5000-b7fc7000 r--p 001a3000 08:01 660181     /lib/i386-linux-gnu/libc-2.15.so
b7fc7000-b7fc8000 rw-p 001a5000 08:01 660181     /lib/i386-linux-gnu/libc-2.15.so
b7fc8000-b7fcb000 rw-p 00000000 00:00 0
b7fda000-b7fdb000 r--p 005e0000 08:01 922740     /usr/lib/locale/locale-archive
b7fdb000-b7fdd000 rw-p 00000000 00:00 0
b7fdd000-b7fde000 r-xp 00000000 00:00 0          [vdso]
b7fde000-b7ffe000 r-xp 00000000 08:01 661146     /lib/i386-linux-gnu/ld-2.15.so
b7ffe000-b7fff000 r--p 0001f000 08:01 661146     /lib/i386-linux-gnu/ld-2.15.so
b7fff000-b8000000 rw-p 00020000 08:01 661146     /lib/i386-linux-gnu/ld-2.15.so
bffdf000-c0000000 rw-p 00000000 00:00 0          [stack]
```

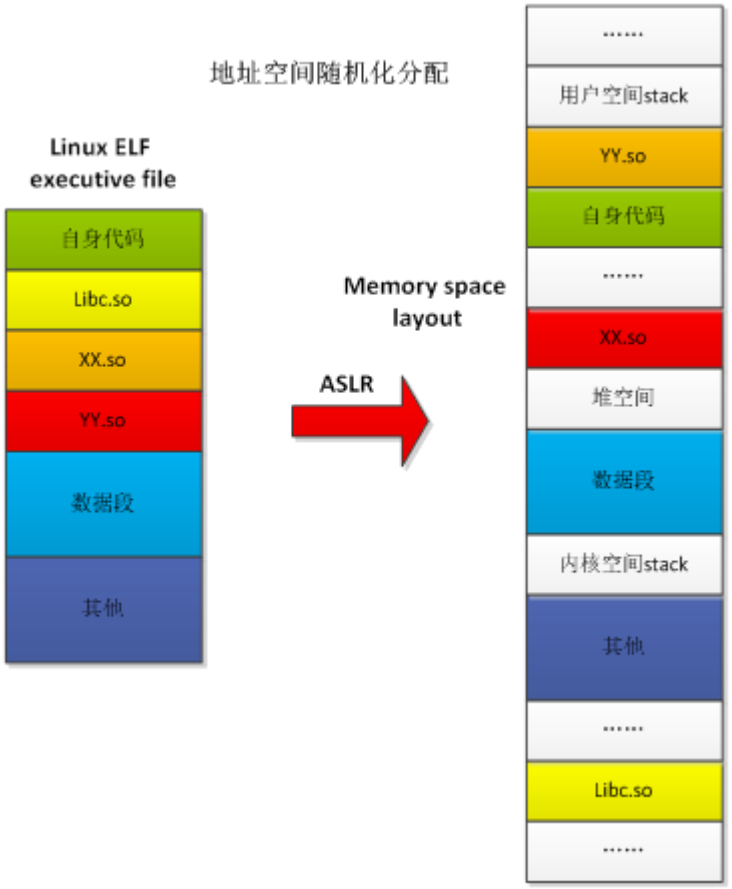
操作系统安全技术

- 地址空间随机化分配
  - 随机分配方式一：相同属性的空间区域统一随机值错位
  - 各个空间区域相对位置保持不变
  - 优点：兼容性好
  - 缺点：随机化弱，容易被暴力破解
  - 代表：linux 内核



操作系统安全技术

- 地址空间随机化分配
  - 随机分配方式二：所有的地址空间区域采用不同的随机值错位
  - 各个空间区域相对位置随机
  - 优点：随机化强，不容易被暴力破解
  - 缺点：兼容性不好
  - 代表：PaX 全随机化

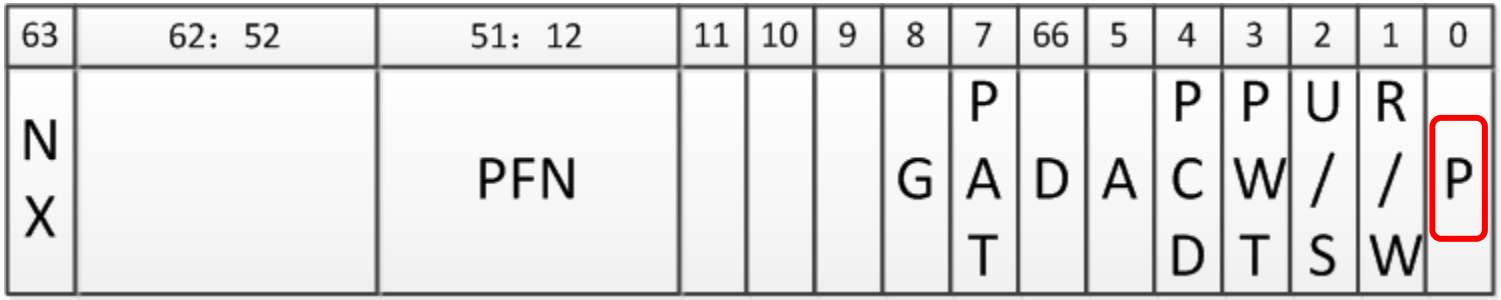


操作系统安全技术

- 地址空间随机化分配
  - 进程内核栈随机化
  - 每个进程有两个页面作为进程陷入内核态的栈，用于系统调用参数传递，上下文切换，中断、异常处理等
  - 代表：PaX

硬件安全技术

- 页面存在位保护
  - 页面存在位P=1的页面才可以访问
  - P=0,页面无效，页面访问异常，缺页处理



硬件安全技术

- 读/写位保护
  - P=1,R/W=1，页面可读、写、执行
  - P=1,R/W=0，页面只读，可执行

63	62：52	51：12	11	10	9	8	7	66	5	4	3	2	1	0
NX		PFN				G	PAT	D	A	PCD	PWT	U/S	R/W	P

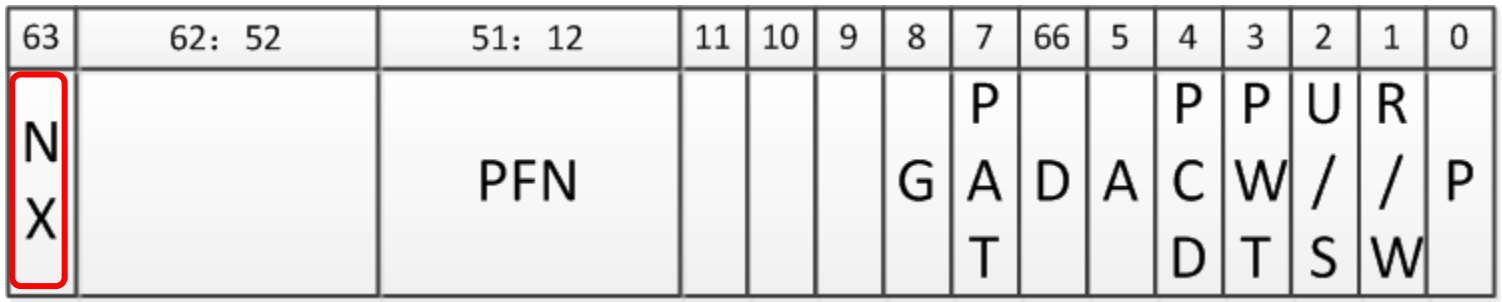
页面属于 read/write 权限，则必须要每一级页表项的 R/W 位都为 1。而属于 read-only 权限，只需要任何一级页表项的 R/W 位为 0

硬件安全技术

- WP（write protected）写保护
  - 处理器CR0.WP功能防止 supervisor-mode 改写只读（read-only）页面
  - CR0.WP = 0, supervisor-mode 可以对只读（read-only）页面进行写访问
  - CR0.WP = 1, supervisor-mode 不能对只读（read-only）页面进行写访问
  - 不论CR0.WP 为何值都不允许User-mode对只读（read-only）页面进行写访问

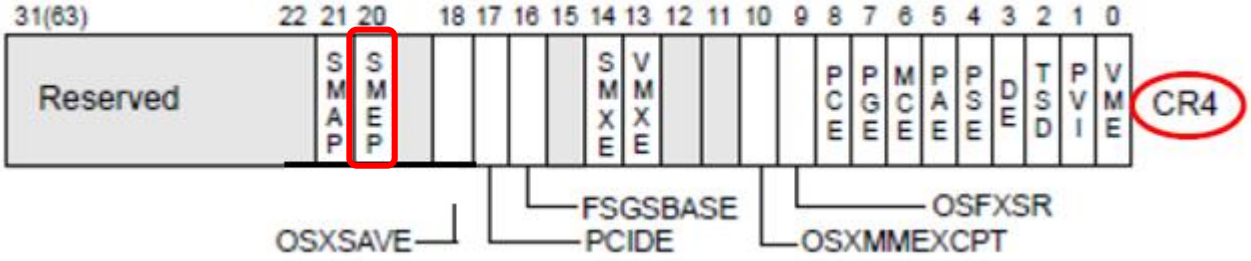
硬件安全技术

- 不可执行位
  - 增加页表执行位属性
  - NX，XD
  - 原理：取指令代码页时，如果页表的不可执行位置位时，产生页访问异常，终止进程



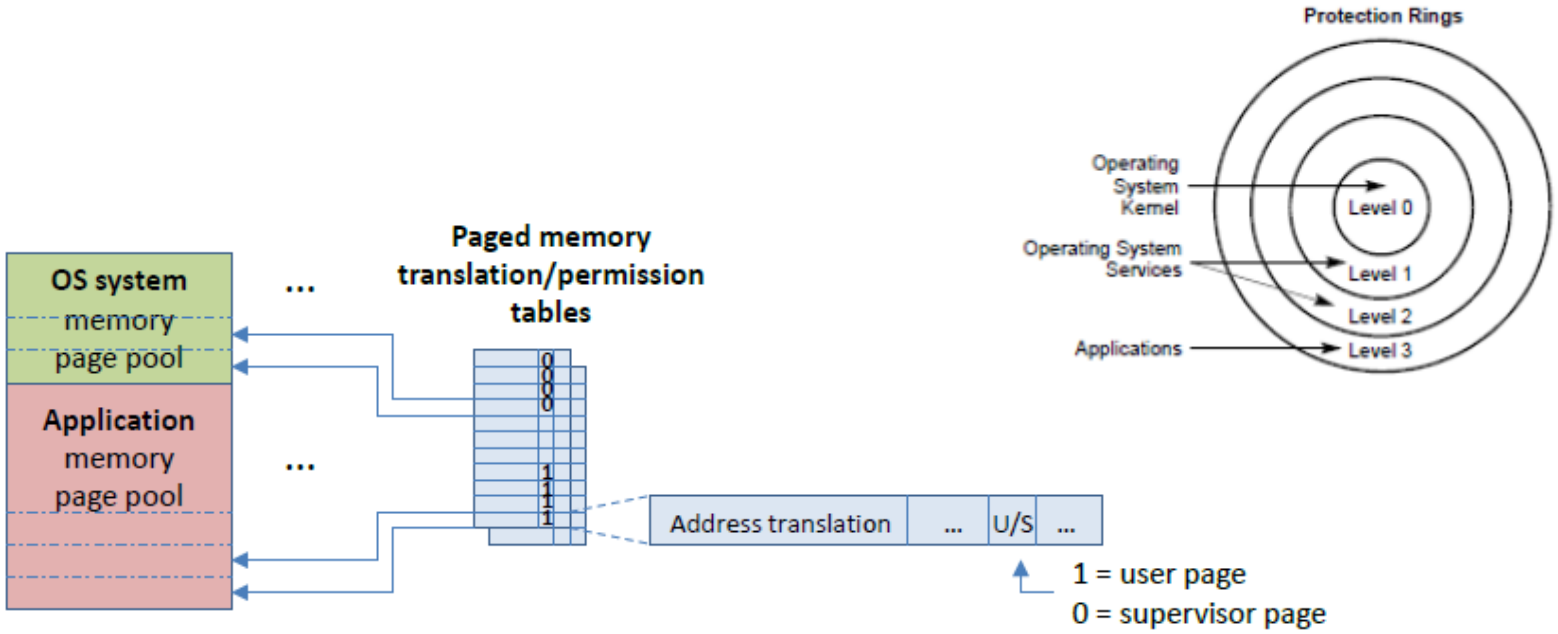
硬件安全技术

- 管理模式执行保护
  - SMEP——Supervisor Mode Execution Protection
  - 防止提权运行
  - 防止管理权限运行用户空间代码



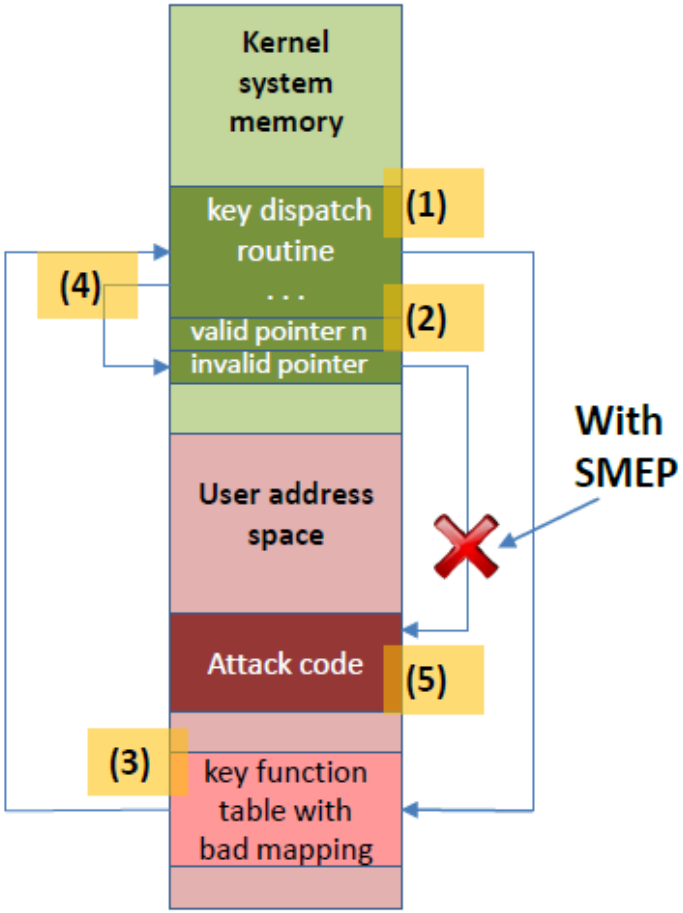
硬件安全技术

- 管理模式执行保护



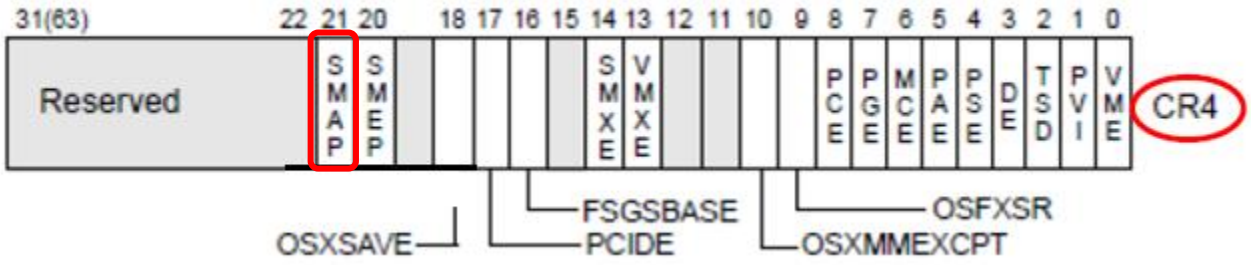
硬件安全技术

- 管理模式执行保护
  - Historical access permission rules for code execution：
    - » Supervisor mode (levels 0-2): user or supervisor pages allowed (u/s==*)
    - » User mode (level 3): user only (u/s==1)
  - When SMEP is active:
    - » Supervisor mode: supervisor only (u/s==0)
    - » User mode: user only (u/s==1)



硬件安全技术

- 管理模式访问阻止
  - SMAP——supervisor mode access prevention
  - 防止提权页面访问
    - » 阻止高特权运行模式下对用户页的访问
    - » 当开启 SMAP 机制后，处理器运行在 supervisor 权限下（CPL < 3），将不能访问（包括 read 与 write）属于 user 权限的页面里的数据（U/S = 1）



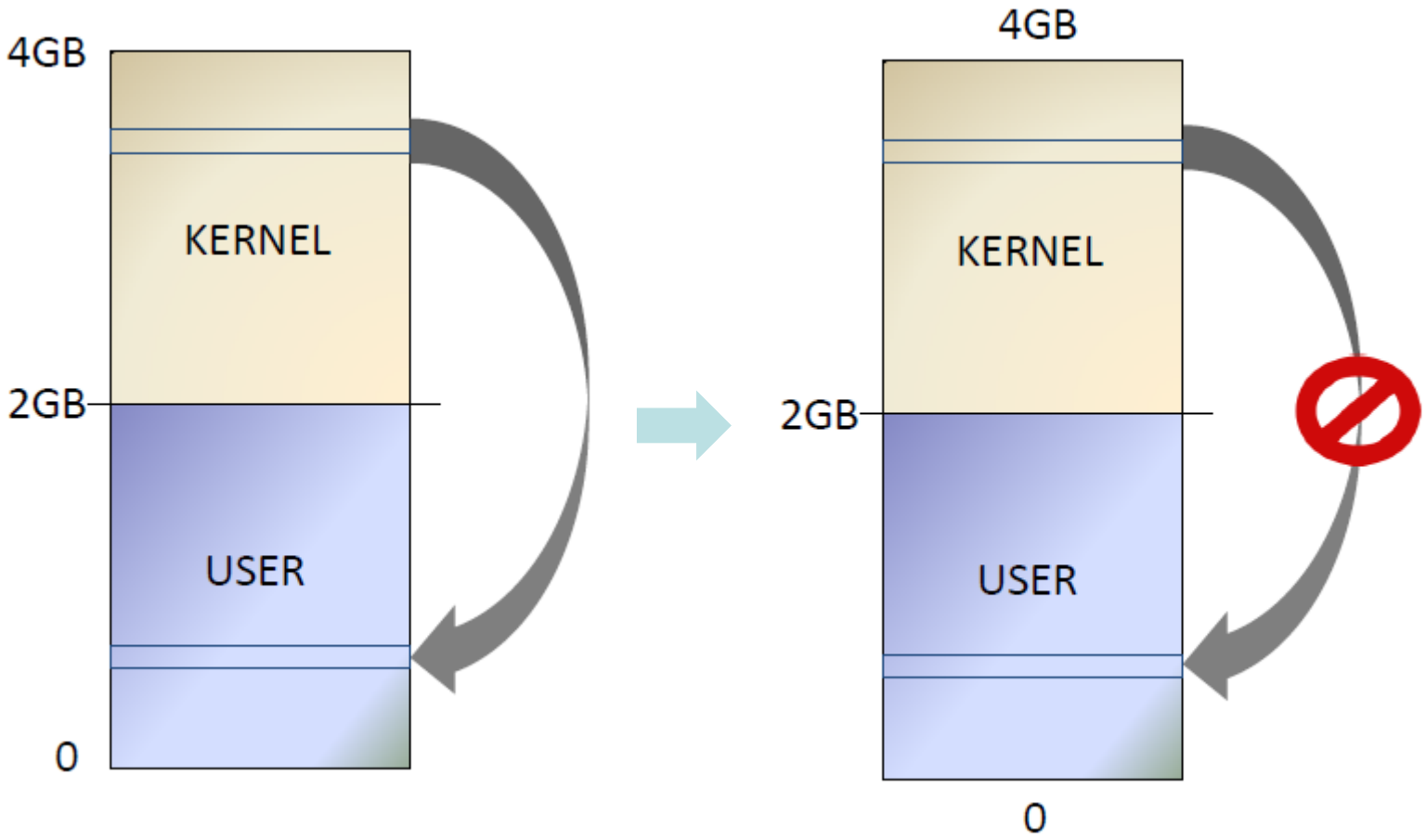
- supervisor 与 user 页面
  - 在 32-bit paging 模式下，PDE 以及 PTE 的 U/S 位（bit 2）相 "AND" 后的结果。
  - 在 PAE paging 模式下。
    - 使用 4K 页面时：PDPTE，PDE 以及 PTE 三者的 U/S 位（bit 2）相 "AND" 后的结果。
    - 使用 2M 页面时：PDPTE 以及 PDE 两者的 U/S 位（bit 2）相 "AND" 后的结果。
  - 在 IA-32e paging 模式下。
    - 使用 4K 页面时：PML4E, PDPTE，PDE 以及 PTE 四者的 U/S 位（bit 2）相 "AND" 后的结果。
    - 使用 2M 页面时：PML4E, PDPTE 以及 PDE 三者的 U/S 位（bit 2）相 "AND" 后的结果。
    - 使用 1G 页面时：PML4E 以及 PDPTE 两者的 U/S 位（bit 2）相 "AND" 后的结果。
  - 因此，页面属于 user 权限，则必须要每一级页表项的 U/S 位都为 1。而属于 supervisor 权限，只需要任何一级页表项的 U/S 位为 0 即可。

硬件安全技术

- 无SMAP:
  - 在基于 CPL 权限的页级保护措施里，有二种访问权限：supervisor 与 user 访问权限:
  - 属于 supervisor 访问权限的只有在处理器处于 CPL < 3 权限时才允许访问
  - 属于 user 访问权限时， supervisor 与 user 都可以访问 user 页面数据

硬件安全技术

- 管理模式访问阻止



硬件安全技术

- SMAP 机制下的读访问
  - 当处理器运行在 CPL < 3 权限下（supervisor），尝试读访问 user 页面时，有下面的情况：
    - » 当 CR4.SMAP = 0 时，supervisor 允许读访问 user 页面。
    - » 当 CR4.SMAP = 1 时，取决于 eflags.AC 标志位的值，将有下面的情形
      - eflags.AC = 0 （CLAC 指令）时，supervisor 不能读访问 user 页面，将产生 #PF 异常。
      - eflags.AC = 1（ STAC 指令）时，supervisor 允许读访问 user 页面
  - 将 eflags.AC 位清为 0，并且 CR4.SMAP = 1 时，才真正开启 SMAP 功能。

硬件安全技术

- SMAP 机制下的写访问
  - 当 CR0.WP = 0 并且 CR4.SMAP = 0 时，supervisor 允许写访问所有的 页面（包括 read-only 以及 read/write 页面）。
  - 当 CR0.WP = 1 并且 CR4.SMAP = 0 时，supervisor 允许写访问所有 read/write 页面，不能改写 read-only 页面。
  - 当 CR0.WP = 0 并且 CR4.SMAP = 1 时，取决于 eflags.AC 标志位有下面的情形：
    - AC = 0 时，supervisor 只能写访问 supervisor 的页面（包括 read-only 与 read/write 页面）。
    - AC = 1 时，supervisor 允许写访问所有的页面（包括 read-only 与 read/write 页面）。
  - 当 CR0.WP = 1 并且 CR4.SMAP = 1 时，取决于 eflags.AC 标志位有下面的情形：
    - AC = 0 时，supervisor 只能写访问 supervisor 的 read/write 页面。也就是：不能改写所有的 read-only 页面以及 user 的 read/write 页面。
    - AC = 1 时，supervisor 允许写访问所有的 read/write 页面，不能改写任何 read-only 页面。

硬件安全技术

- 内存保护扩展
  - MPX——Memory Protection Extensions
  - ISA扩展，增加相应的边界寄存器和处理边界寄存器的指令
  - 防止 user mode和supervisor mode缓冲区的越界访问
  - 防止缓冲区溢出

硬件安全技术

- 内存保护扩展
  - MPX——Memory Protection Extensions
    - » bndmk： 在界限寄存器中创建 LowerBound (LB) 和 UpperBound (UB)
    - » bndmov： 从内存中获取（上下）界限信息并将其放在界限寄存器中
  - bndcl： 检查下界限
  - bndcu：检查上界限

硬件安全技术

- 内存保护键
  - MPK——Memory Protection Keys
    - 提供更轻量的内存访问控制，可以指定一个内存区域的读，写，执行(用户空间可以直接设置)
    - 为频繁切换内存访问属性的应用提供方便，如加密应用
    - 可以把地址区域划分出16（PK）个区域，每个区域独立设置访问控制权限

硬件安全技术

- 控制流增强技术
  - CET ——Control-flow Enforcement Technology
  - 提高防御ROP/JOP控制流攻击的技术
  - 影子堆栈（Shadow Stack）
  - 返回地址保护来防范返回导向编程攻击
  - 间接分支跟踪（Indirect branch tracking）
  - 分支保护，以防止跳转/调用导向编程攻击

硬件安全技术

- 随机化因子产生器
  - 由硬件协处理器生成随机化因子
  - 增强地址空间随机化，加密等

课后思考&动手

- 找一个处理器平台用户手册阅读
- 找一个开源安全解决方案，阅读文档，代码

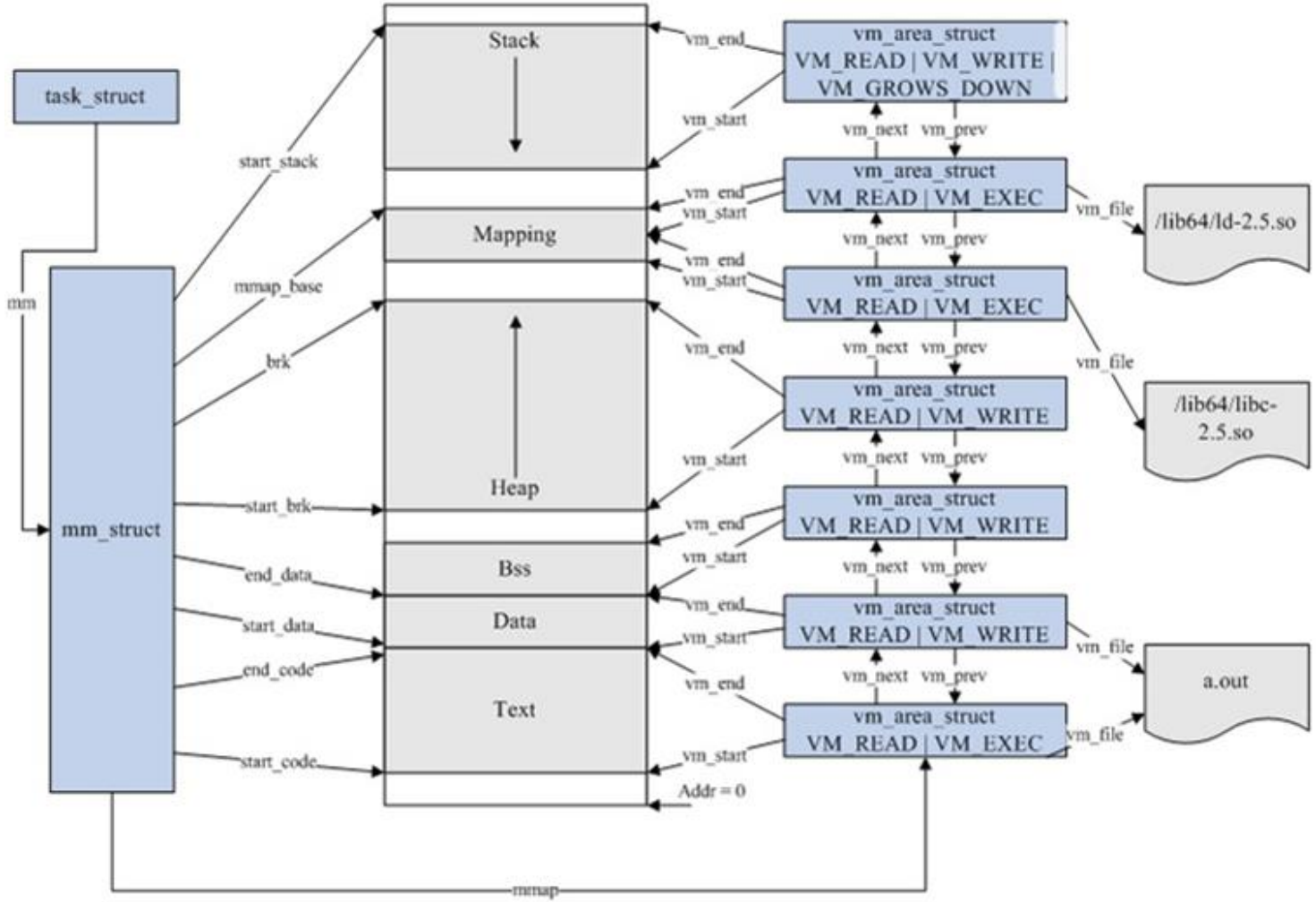
目录

Linux系统地址空间随机化分配实现

- 随机化实现
  - 知识准备
  - 编译基本原理
  - ELF组织形式
  - 动态连接
  - Linux进程数据结构
  - 内存管理
  - Linux 进程地址空间组织
  - 平台体系结构
  - 可执行文件加载过程
  - 系统调用（exec,mmap……）
  - ……

Linux系统地址空间随机化实现架构

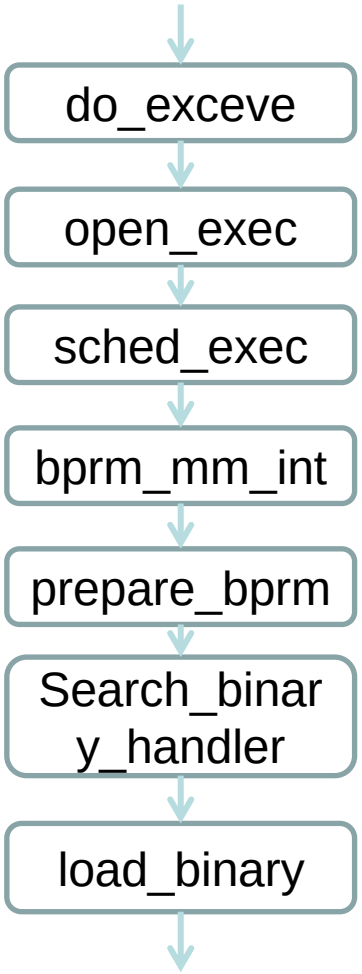
- 架构图



地址空间随机化分配运行效果

```
root@ubuntu:/home/abang# cat /proc/self/maps
08048000-08053000 r-xp 00000000 08:01 1054472    /bin/cat
08053000-08054000 r--p 0000a000 08:01 1054472    /bin/cat
08054000-08055000 rw-p 0000b000 08:01 1054472    /bin/cat
0925c000-0927d000 rw-p 00000000 00:00 0          [heap]
b7241000-b73a6000 r--p 001c8000 08:01 922740     /usr/lib/locale/locale-archive
b73a6000-b75a6000 r--p 00000000 08:01 922740     /usr/lib/locale/locale-archive
b75a6000-b75a7000 rw-p 00000000 00:00 0
b75a7000-b774a000 r-xp 00000000 08:01 660181     /lib/i386-linux-gnu/libc-2.15.so
b774a000-b774c000 r--p 001a3000 08:01 660181     /lib/i386-linux-gnu/libc-2.15.so
b774c000-b774d000 rw-p 001a5000 08:01 660181     /lib/i386-linux-gnu/libc-2.15.so
b774d000-b7750000 rw-p 00000000 00:00 0
b775f000-b7760000 r--p 005e0000 08:01 922740     /usr/lib/locale/locale-archive
b7760000-b7762000 rw-p 00000000 00:00 0
b7762000-b7763000 r-xp 00000000 00:00 0          [vdso]
b7763000-b7783000 r-xp 00000000 08:01 661146     /lib/i386-linux-gnu/ld-2.15.so
b7783000-b7784000 r--p 0001f000 08:01 661146     /lib/i386-linux-gnu/ld-2.15.so
b7784000-b7785000 rw-p 00020000 08:01 661146     /lib/i386-linux-gnu/ld-2.15.so
bfac9000-bfaea000 rw-p 00000000 00:00 0          [stack]
root@ubuntu:/home/abang# echo 0 > /proc/sys/kernel/randomize_va_space
root@ubuntu:/home/abang# cat /proc/self/maps
08048000-08053000 r-xp 00000000 08:01 1054472    /bin/cat
08053000-08054000 r--p 0000a000 08:01 1054472    /bin/cat
08054000-08055000 rw-p 0000b000 08:01 1054472    /bin/cat
08055000-08076000 rw-p 00000000 00:00 0          [heap]
b7abc000-b7c21000 r--p 001c8000 08:01 922740     /usr/lib/locale/locale-archive
b7c21000-b7e21000 r--p 00000000 08:01 922740     /usr/lib/locale/locale-archive
b7e21000-b7e22000 rw-p 00000000 00:00 0
b7e22000-b7fc5000 r-xp 00000000 08:01 660181     /lib/i386-linux-gnu/libc-2.15.so
b7fc5000-b7fc7000 r--p 001a3000 08:01 660181     /lib/i386-linux-gnu/libc-2.15.so
b7fc7000-b7fc8000 rw-p 001a5000 08:01 660181     /lib/i386-linux-gnu/libc-2.15.so
b7fc8000-b7fcb000 rw-p 00000000 00:00 0
b7fda000-b7fdb000 r--p 005e0000 08:01 922740     /usr/lib/locale/locale-archive
b7fdb000-b7fdd000 rw-p 00000000 00:00 0
b7fdd000-b7fde000 r-xp 00000000 00:00 0          [vdso]
b7fde000-b7ffe000 r-xp 00000000 08:01 661146     /lib/i386-linux-gnu/ld-2.15.so
b7ffe000-b7fff000 r--p 0001f000 08:01 661146     /lib/i386-linux-gnu/ld-2.15.so
b7fff000-b8000000 rw-p 00020000 08:01 661146     /lib/i386-linux-gnu/ld-2.15.so
bffdf000-c0000000 rw-p 00000000 00:00 0          [stack]
```

可执行程序加载过程



读取可执行文件

选取最小负载CPU以执行程序

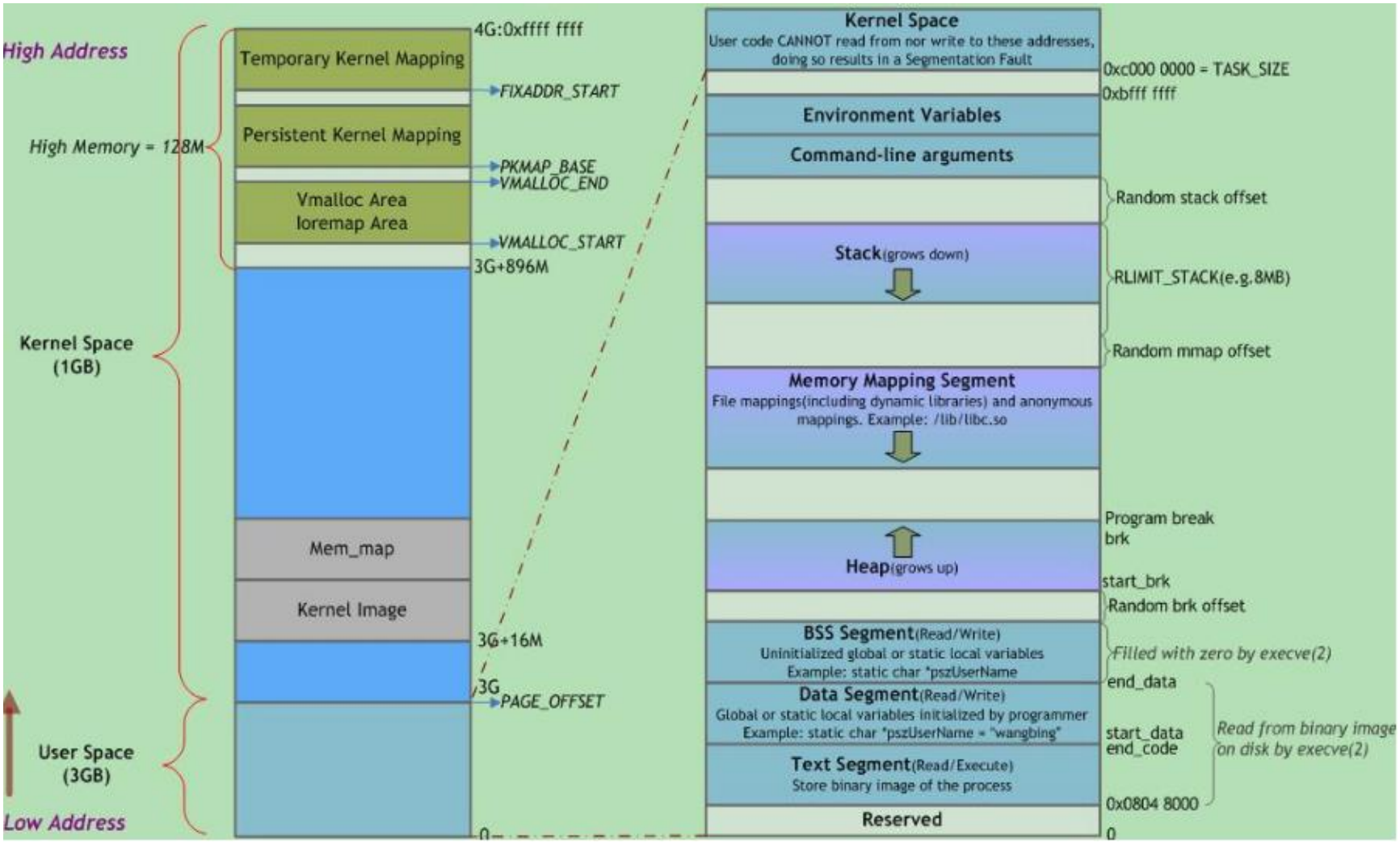
初始化内存管理

填充linux_binprm数据结构

Load_binary函数关键调用

- load_binary
  - 读取可执行文件头部，校验魔数，读取动态链接程序，读取可加载项，检查动态链接执行许可权.....
  - arch_pick_mmap_layout
    - » 决定进程线性区的布局，平台相关
  - setup_arg_pages
    - » 为进程用户态分配一个新的线性区描述符
  - do_mmap
    - » 创建一个新线性区对可执行文件正文段进行映射
  - do_brk
    - » 创建一个新的匿名线性区来映射程序的bss段
  - start_thread
    - » 处理内核态栈

进程空间随机化布局


High Address
High Memory = 128M
Kernel Space (1GB)
User Space (3GB)
Low Address
4G:0xffff ffff
Temporary Kernel Mapping
FIXADDR_START
Persistent Kernel Mapping
PKMAP_BASE
VMALLOC_END
Vmalloc Area
Ioremap Area
VMALLOC_START
3G+896M
Mem_map
Kernel Image
3G+16M
3G
PAGE_OFFSET
0
Kernel Space
User code CANNOT read from nor write to these addresses, doing so results in a Segmentation Fault
0xc000 0000 = TASK_SIZE
0xbfff ffff
Environment Variables
Command-line arguments
Random stack offset
Stack(grows down)
RLIMIT_STACK(e.g.8MB)
Random mmap offset
Memory Mapping Segment
File mappings(including dynamic libraries) and anonymous mappings. Example: /lib/libc.so
Program break
brk
Heap(grows up)
start_brk
Random brk offset
BSS Segment(Read/Write)
Uninitialized global or static local variables
Example: static char *pszUserName
Filled with zero by execve(2)
end_data
Data Segment(Read/Write)
Global or static local variables initialized by programmer
Example: static char *pszUserName = "wangbing"
start_data
end_code
Read from binary image on disk by execve(2)
Text Segment(Read/Execute)
Store binary image of the process
0x0804 8000
Reserved
0

PaX Linux内核增强补丁

- 支持不可执行
  - 页式管理
  - 段式管理
- 完全地址随机化映射
  - 每个系统调用的内核栈随机映射
  - 用户栈随机映射
  - ELF可执行映像随机映射
  - Brk()分配的heap随机映射
  - Mmap()管理的heap随机映射
  - 动态链接库随机映射

PaX Linux内核增强补丁

- 整数溢出保护
- 内核，用户空间数据拷贝保护
- 软件实现SMEP，SMAP（特定平台）
- 内核栈清零保护
- 内核页只读
  - Const结构只读
  - 系统调用表只读
  - 局部段描述符表（IDT）只读
  - 全局段描述符表（GDT）只读
- 数据页只读
- ……

Linux系统地址空间随机化分配开源实现

- 获取PaX
  - Docs: http://pax.grsecurity.net/docs/index.html
  - 补丁代码： http://pax.grsecurity.net/
- PaX缺点：不支持LKM,兼容性不好

课后思考&动手

- 找一个Linux系统，玩一下地址空间随机化分配等功能
- 给linux系统内核打PaX补丁，运行下，体验一下各个安全功能

谢谢！