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11次课] 代码复用攻击及防御介绍

授课教师：陈李维

授课时间：2024.5.6

内容概要

- 代码复用攻击
  - ret2libc
  - ROP
  - JOP
- 对代码复用攻击的防御
  - ASLR
  - CFI
- 总结

内容概要

- 代码复用攻击
  - ret2libc
  - ROP
  - JOP
- 对代码复用攻击的防御
  - ASLR
  - CFI
- 总结

研究背景

- 不可执行位保护（NX或DEP）的提出，从理论上彻底解决了代码注入攻击，也就终结了对代码注入攻击及其防御技术的学术研究。
  - 注意：是从理论上终结，但是实际上代码注入攻击仍然是一种常见的攻击方式
- 为了绕过NX保护，研究者提出了一种新的攻击方法，**代码复用攻击**（代码重用攻击），成为目前最为主流的一种攻击方法，也是学术界研究的热点问题。

代码复用攻击的原理

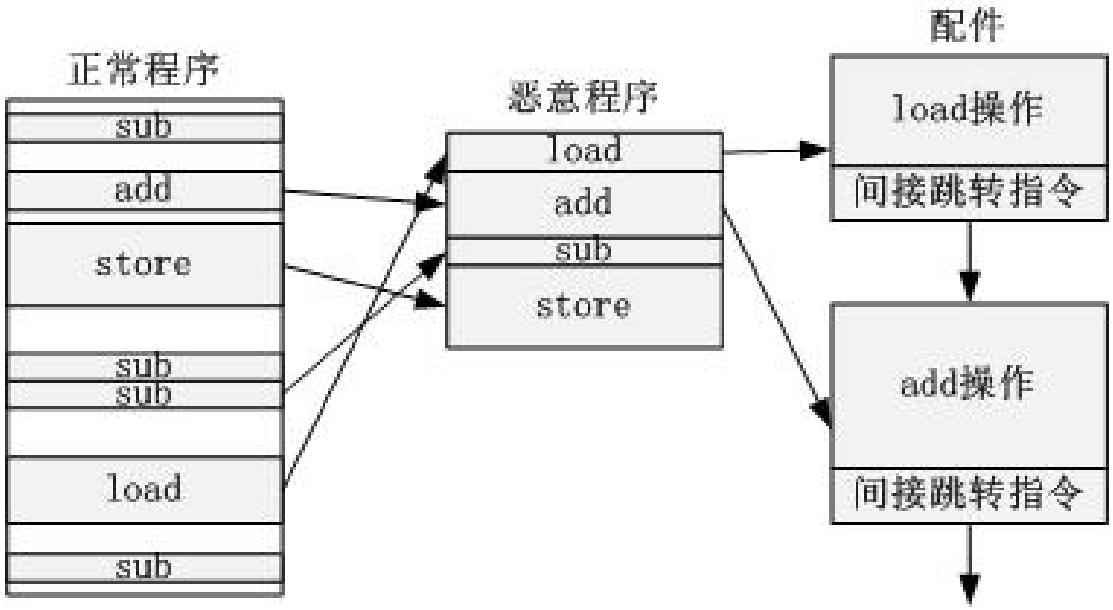
- **代码复用攻击**（Code Reuse Attack，CRA），**复用**计算机系统内部已有的代码，将已有的代码**重新组合**，形成具有一定功能的恶意代码，从而对计算机系统进行攻击。

代码复用攻击的原理

- 配件（gadget）：一段系统中**已有的**、可以被攻击者**复用的**代码
- 配件是代码复用攻击的**基本单元**，具有以下特征：
  - **具有一定的功能**
    - 比如，一个加法操作、一个内存访问操作等。
  - **以间接跳转指令为结尾。**
    - 可用的配件分布在内存中的不同位置。为了从一个配件跳转到下一个配件，需要利用间接跳转指令来劫持控制流。
  - **配件的长度通常很短**
    - 避免引入不相关的指令，破坏攻击操作。

代码复用攻击的原理

- **配件链**：将多个配件按照一定的顺序组成一段代码链，就形成了具有攻击能力的恶意代码。
- 通过配件末尾的**间接跳转指令**将不同的配件链接在一起。

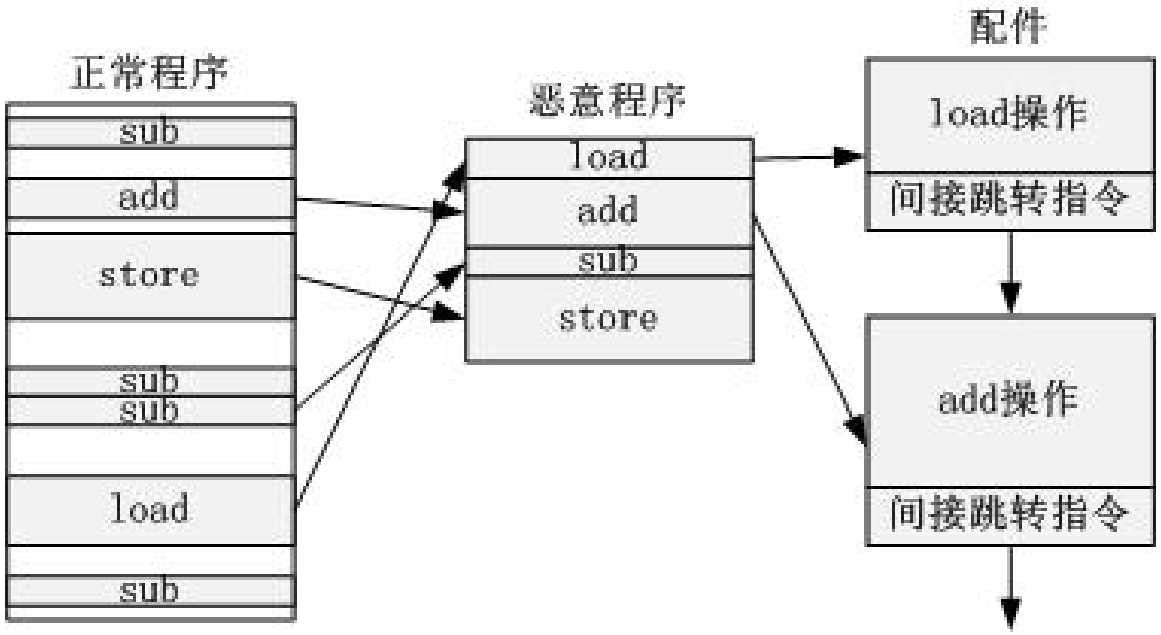


代码复用攻击的过程

- 代码注入攻击的过程：
  - 构造恶意代码
  - 注入恶意代码
  - 执行恶意代码
- 代码复用攻击的过程：
  - 构造配件链（配件链相当于恶意代码）
  - 执行配件链

代码复用攻击的过程

- **构造**配件链
  - **寻找合适的配件**：按照特定的攻击需求，从系统已有代码中寻找具有特定功能的配件。
  - **构造配件链**：将寻找到的配件按照一定的顺序串联起来，形成具有一定功能的配件链。



代码复用攻击的过程

- **执行**配件链
  - 首先，将配件链中所有配件的地址全部注入到系统内存中。
  - 然后，利用内存漏洞，**劫持控制流**，让系统按照配件的地址，依次执行配件链中的不同配件的指令，最终完成代码复用攻击。
  - 注：所有代码的地址都是**固定的**（上一讲的内容）。

代码复用攻击的分析

- 代码注入攻击：
  - 向系统中注入恶意代码。
  - 劫持一次控制流，让系统跳转到恶意代码执行。
- 代码复用攻击：
  - 向系统中注入的数据全部是真正的数据，而不是指令，其中包括配件的地址和配件指令所需要的输入。因此，能够绕过不可执行位保护。
  - 配件全部都是系统中已有的代码。
  - 需要一直劫持控制流，让控制流在不同配件之间频繁的跳转。

代码复用攻击的分析

- 代码复用攻击的**优点**：
  - 攻击的**隐蔽性更强**。
    - 完全复用系统已有代码，能够绕过不可执行位保护。
  - 代码复用攻击已经被证明是**图灵完备的攻击**。
    - 由于系统中有大量的可用代码，因此可以从中找到足够多的各种各样的配件，足够攻击者完成**任意的操作**。

代码复用攻击的分析

- 代码复用攻击的**缺点**：
  - 攻击的**复杂性较高**。
    - 相对于代码注入攻击直接让运行注入的恶意代码，代码复用攻击需要攻击者构造配件链，精心控制控制流，让系统准备的在不同配件之间跳转，加大了攻击者的攻击难度。
  - 对内存漏洞的利用更加复杂。
    - 代码注入攻击只需要劫持一次控制流，代码复用攻击需要劫持多次控制流。

代码复用攻击的分析

- 结论：
  - 相对于代码注入攻击，代码复用攻击的隐蔽性更强，对代码复用攻击的防御变得更加困难。
  - 另一方面，代码复用攻击的复杂度也提高了，攻击者进行攻击的成本也提高了。
- 安全防御研究的目的不是彻底的阻止攻击，而是加大攻击的难度，提高攻击的成本，让**攻击的成本大于攻击的收益**。
  - 其实，不光是用户怕麻烦，攻击者同样也怕麻烦。

代码复用攻击的分类

- 代码复用攻击实现的关键：
  - 如何寻找配件。
    - 以**指令**为单位，遍历整个程序代码空间，寻找所有以间接跳转指令为结尾的代码片段。将这些代码片段作为配件。
    - 以**函数**为单位，寻找所有的函数。将函数作为配件。
    - 直接将**库函数或系统调用**作为配件。
  - 配件之间如何连接。
    - 用ret连接。需要控制栈中函数返回地址。
    - 用call连接。需要控制GOT表等函数跳转地址表。
    - 用indirect-jump/call连接。需要控制通用寄存器。

代码复用攻击的分类

- 根据配件类型、配件之间连接关系、及提出的时间关系等分类，代码复用攻击可以分为两大类：
  - **经典代码复用攻击**
    - 都是在2010年以前提出的攻击方法。主要考虑绕过不可执行位保护，实现图灵完备攻击，基本不考虑针对代码复用攻击的防御。
  - **新型代码复用攻击（下一讲内容）**
    - 大多是在2010年以后提出的攻击方法。考虑到了对代码复用攻击的防御，思考如何绕过这些针对性的防御方法。

经典代码复用攻击的分类

- 经典代码复用攻击
  - ret2libc攻击，将整个库函数作为一个配件。
  - ROP攻击，以ret结尾的代码片段为配件。
  - JOP攻击，以间接跳转指令indirect-jump为结尾的代码片段为配件。
  - COP攻击，以call指令为结尾的代码片段为配件。
- 其中，ROP攻击是代码复用攻击里程碑式的工作。
  - 有时候以ROP来统称ROP/JOP/COP攻击，甚至有时候以ROP来统称整个代码复用攻击。

内容概要

- 代码复用攻击
  - ret2libc
  - ROP
  - JOP
- 对代码复用攻击的防御
  - ASLR
  - CFI
- 总结

ret2libc攻击简介

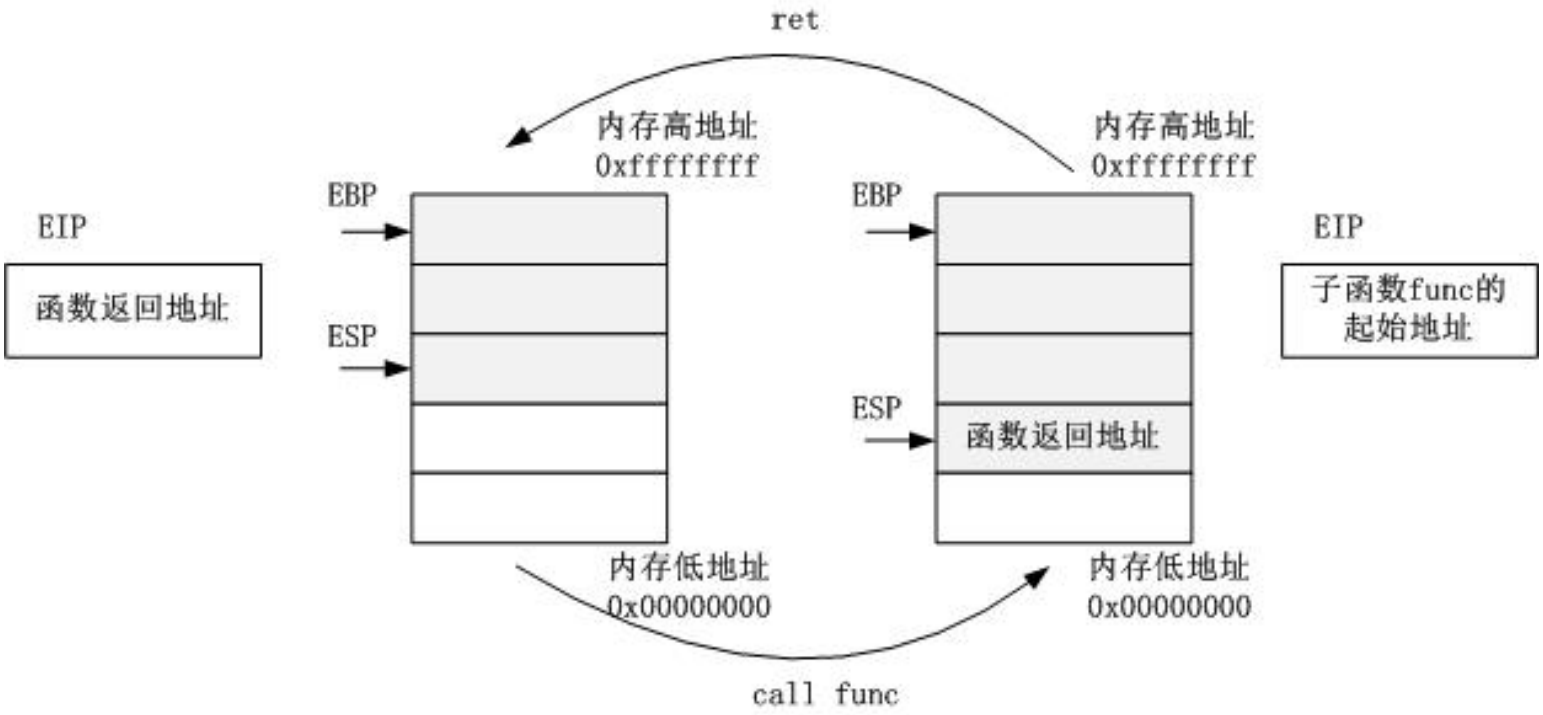
- 代码复用攻击的思想最开始出现在ret2libc攻击中。
- ret2libc攻击是在1997年首次被提出，是最原始的代码复用攻击方法，当时还没有代码复用攻击这个概念。
- 但是，由于当时代码注入攻击仍然是研究的主流，NX还没有出现，ret2libc攻击并没有得到学术界的重视。

ret2libc的原理

- ret2libc（return-into-libc），以ret作为配件之间的连接，将系统中的**库函数**作为配件，利用库函数中已有的功能，实现攻击者的预期目标。
- 更具体的，在Linux系统里，最常用的库函数就是C语言的库函数glibc，最方便攻击者控制使用的间接跳转指令就是ret指令。
- 因此，通常将这种攻击方法称之为ret2libc。

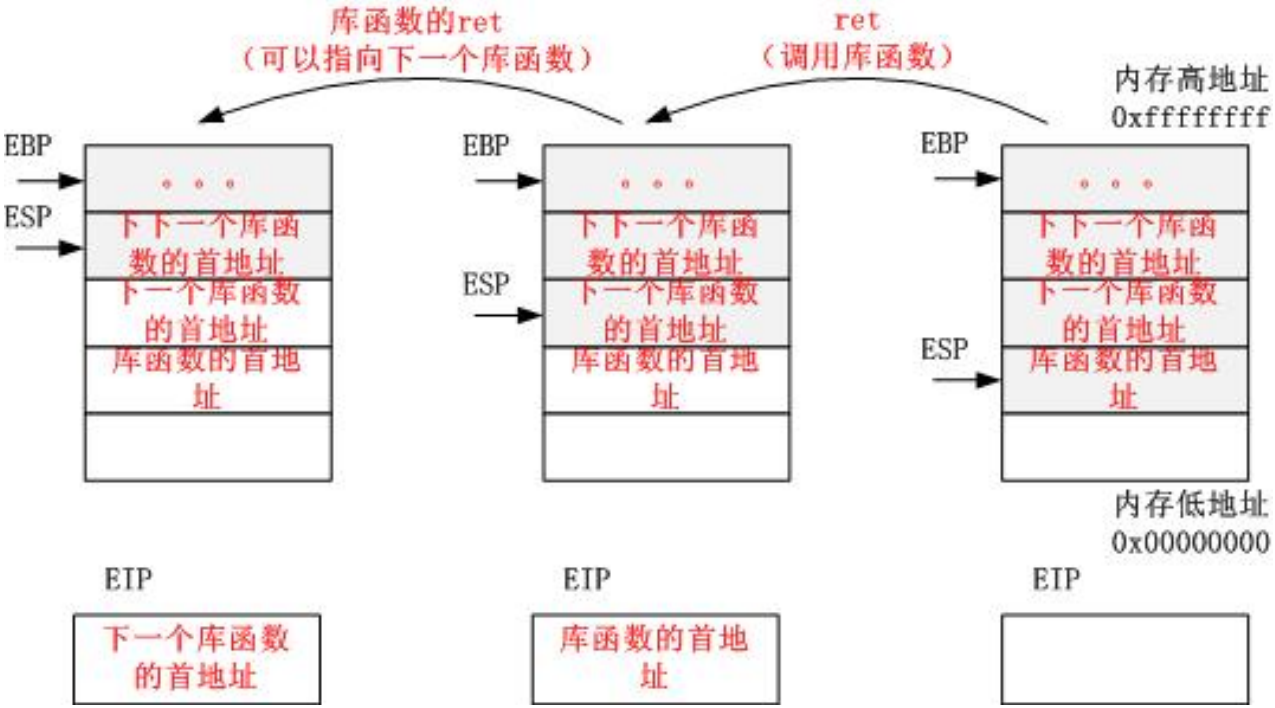
ret2libc的原理

- 正常调用函数的过程：
  - 使用call func调用子函数func。
  - 使用ret，返回主函数。



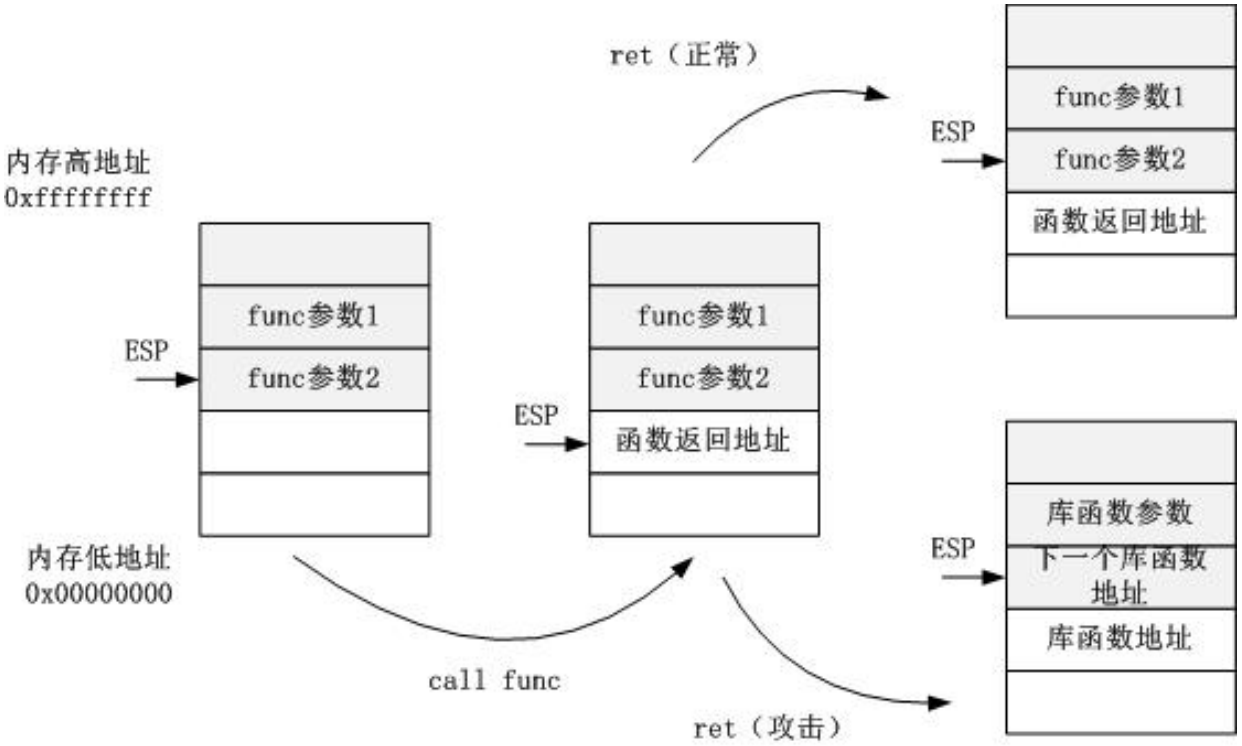
ret2libc的原理

- ret2libc攻击过程（**不考虑输入参数**）：
  - **利用内存漏洞，覆盖栈中的返回地址，利用ret跳转到库函数的首地址。**
  - **库函数使用ret试图返回，结果依然被控制，从而跳转到下一个库函数的首地址。**



ret2libc的原理

- ret2libc攻击过程（**考虑输入参数**）：在32位Linux系统中，函数参数放置在栈上。
  - 利用ret跳转到库函数，相当于用ret实现了一个call。
  - 注意：此时esp指向下一个库函数地址，当前库函数参数在esp的上面。



ret2libc攻击示例

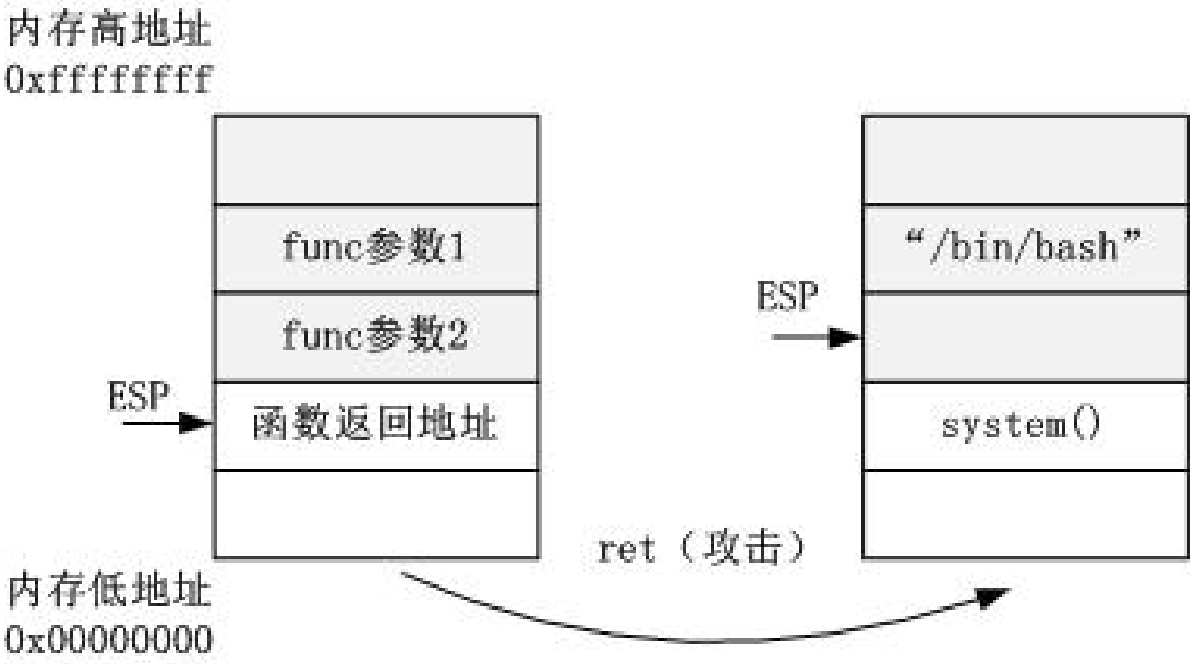
- 示例的攻击目标：利用库函数，实现打开一个shell（非root用户权限）的功能。
- 寻找配件：
  - Linux的系统函数库glibc中有个system()函数，它可以通过/bin/sh命令去执行一个用户命令或者脚本。
  - 可以利用system()函数来实现打开一个shell的功能。

ret2libc攻击示例

- 构造配件及输入：
  - 首先，通过缓冲区溢出漏洞，将栈中函数返回地址覆盖为system()函数的地址。
  - 当被调用的函数准备返回时，system()函数的地址被放入到EIP中。此时，需要准备好相应的参数，让系统执行system(“/bin/sh”)，从而开启一个shell。
  - 示例攻击只有一个配件：system(“/bin/sh”)。

ret2libc攻击示例

- ret2libc攻击：
  - 子函数结束后执行ret指令，通过返回地址上的system()地址以及准备好的参数arg执行system(“/bin/bash”)函数，从而开启一个shell。
  - esp指向system()执行结束后的返回地址。



ret2libc攻击示例

- 实际攻击需要注意的地方：
  - 1）需要关闭ASLR（一种内存地址随机化的防御机制，之后会详细讲解）。
  - 2）获取system()函数地址的方法：由于已经关闭了ASLR，每次进程地址空间中system()函数的地址都是固定的，可直接通过gdb查看。
  - 3）获取“/bin/sh”字符串地址的方法：Linux的每个进程地址空间中有一个环境变量SHELL，我们只要将这个环境变量的地址找到，就把它传给system()作为参数。由于ASLR已经关闭，故这个地址同样可以通过gdb直接找到。

ret2libc攻击的缺陷

- ret2libc攻击的缺点：
  - 需要知道库函数的地址。在实际系统中，库函数的地址是随机的，变化的。
    - 问题：如果库函数地址是不固定的，那么正常程序是怎样来调用库函数的呢?
  - 只能使用库函数已有的固定的功能，不够灵活。

GOT和PLT

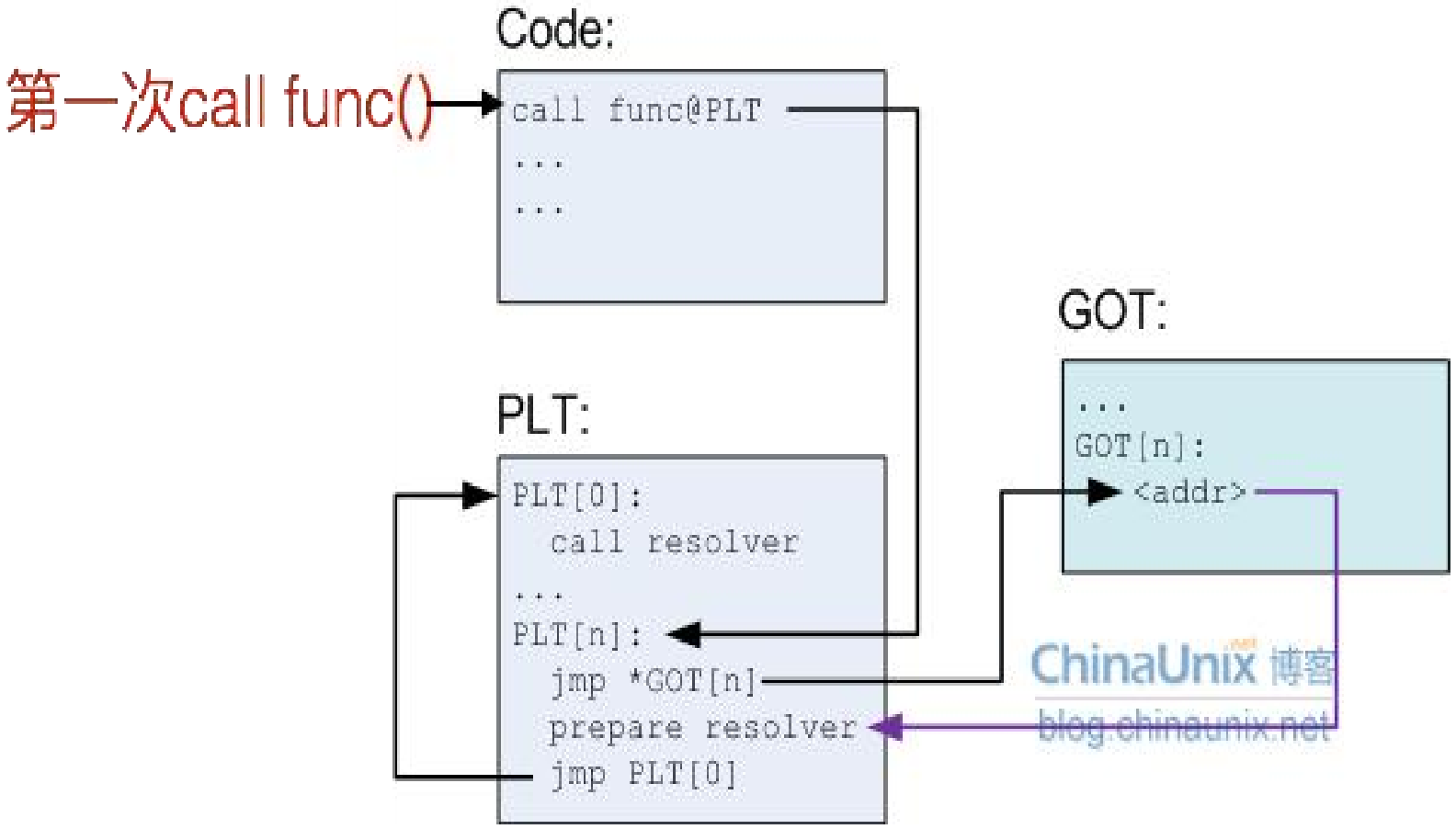
- 在实际系统中，正常程序调用动态库函数，都是通过GOT和PLT来完成的。
- **GOT**：Global Offset Table，全局偏移表。
  - 用于存放动态库函数的实际**地址**。
  - GOT中保存的是地址数据，处于在程序的**数据段**。
- **PLT**：Procedure Linkage Table，链接过程表。
  - 用于存放查找GOT表的具体的**代码**。
  - PLT中保存的是代码，处于在程序的**代码段**。

GOT和PLT

- 简单说来，当程序调用某一个库函数时，先跳转到那个库函数对应的plt代码，plt负责找到库函数对应的got表项。
- 此时分两种情况：
  - 1）若got表中存有对应库函数的有效地址，则直接跳转到该有效地址并执行；
  - 2）若got表没有对应库函数的有效地址，则跳到动态链接代码resolver。由resolver负责查找对应库函数的有效地址，并完成got表的填充，然后再次调用对应的库函数。

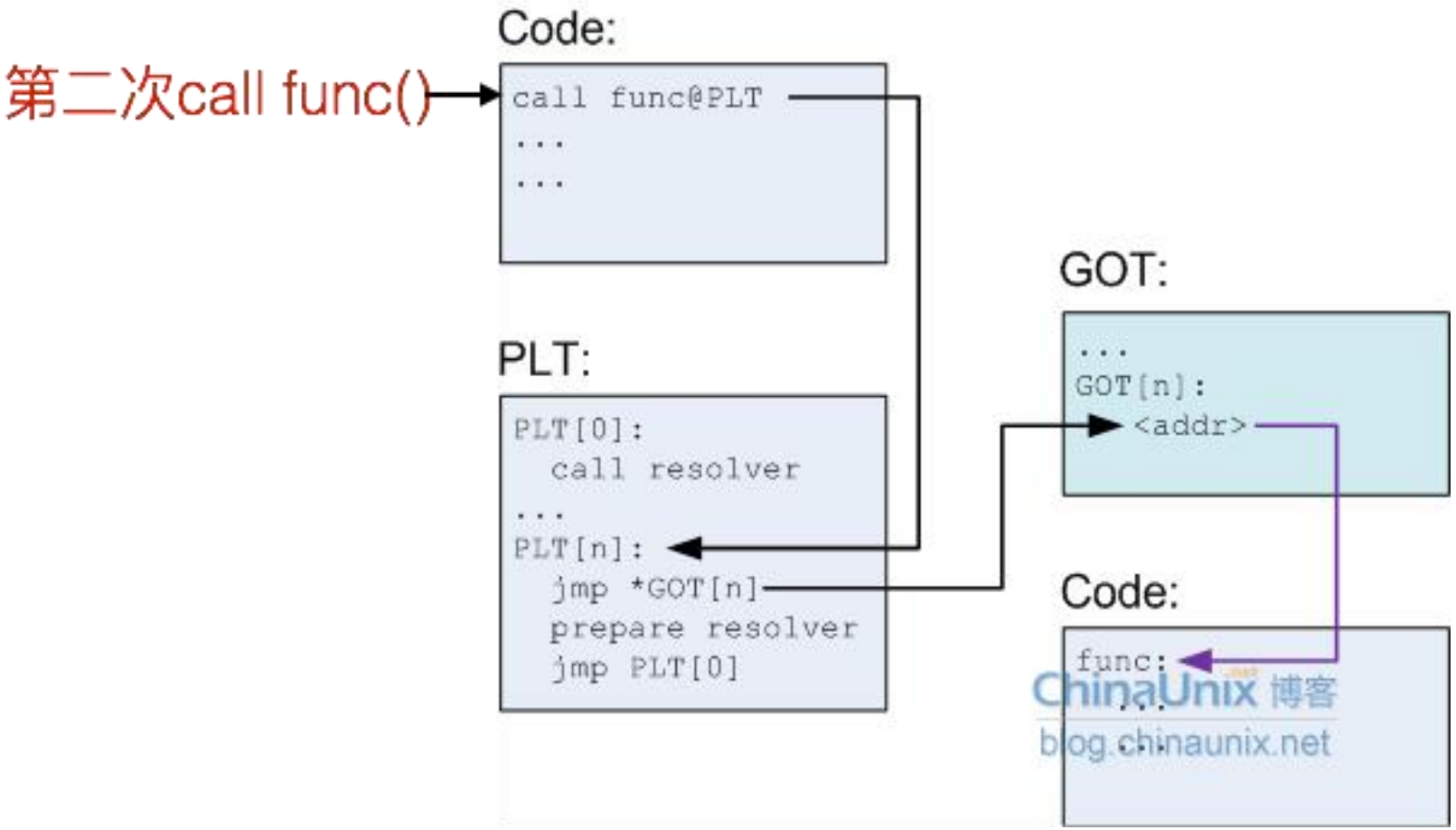
GOT和PLT

GOT和PLT简单介绍



GOT和PLT

GOT和PLT简单介绍

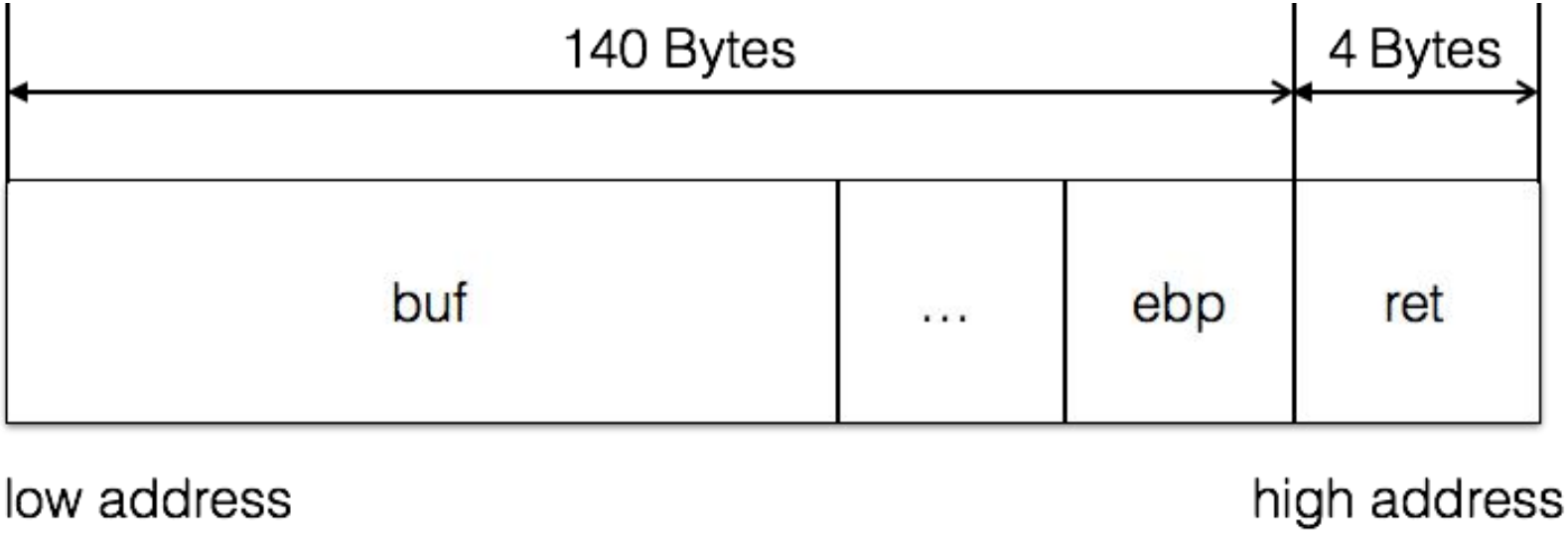


ret2plt攻击

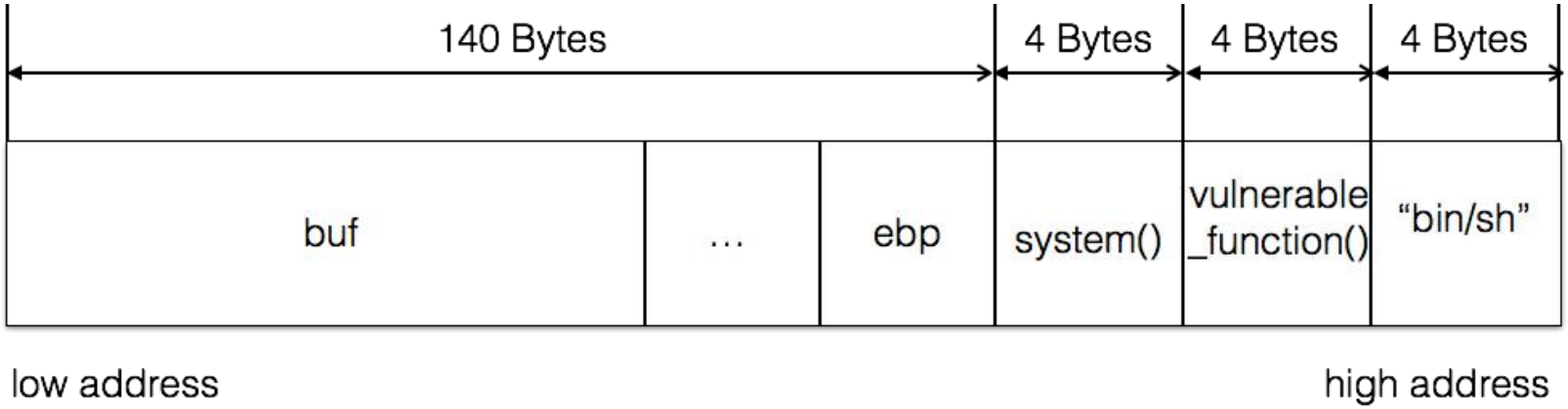
- 和正常程序一样，ret2libc攻击也可以通过GOT和PLT来调用库函数，称之为ret2plt攻击。
  - ret2libc攻击：
    - 需要获知库函数的地址，然后让程序跳转到库函数执行。
  - ret2plt攻击：
    - 不需要知道库函数的地址，只需要知道库函数对应的plt表项（func@plt）。而plt处于代码段，地址是固定的。
    - 跳转到对应的plt表项，由got和plt负责跳转到库函数执行。
    - got和plt都保存在程序的可执行文件中，可以轻易获得。

ret2plt攻击：一个攻击实例

```
void vulnerable_function() {
    char buf[128];
    read(STDIN_FILENO, buf, 256);
}
int main(int argc, char** argv) {
    vulnerable_function();
    write(STDOUT_FILENO, "Hello, World\n", 13);
}
```



ret2plt攻击：一个攻击实例



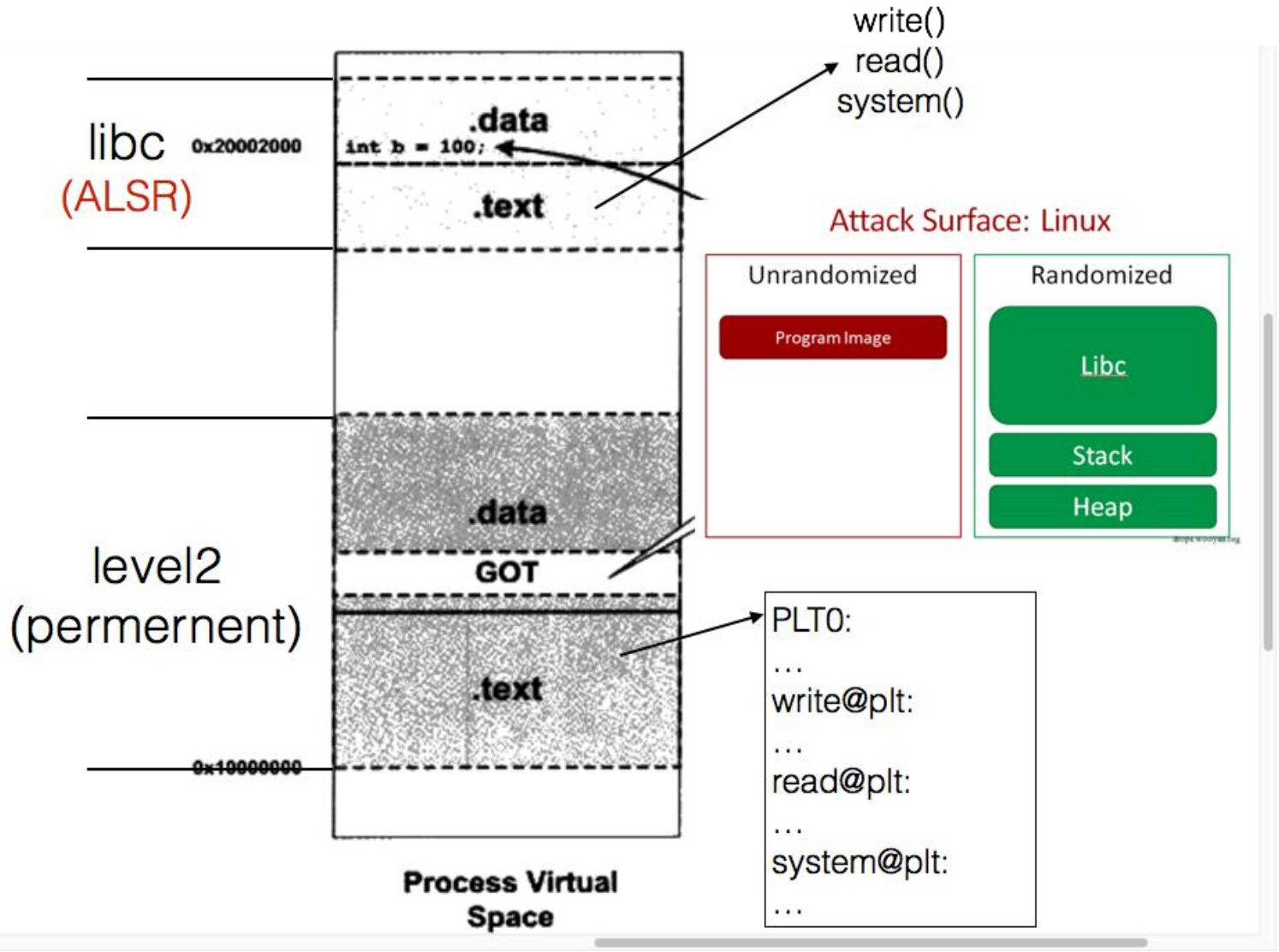
攻击目标：

1.函数返回时不回到main函数，而是执行system（“/bin/sh”）系统调用，打开一个shell。

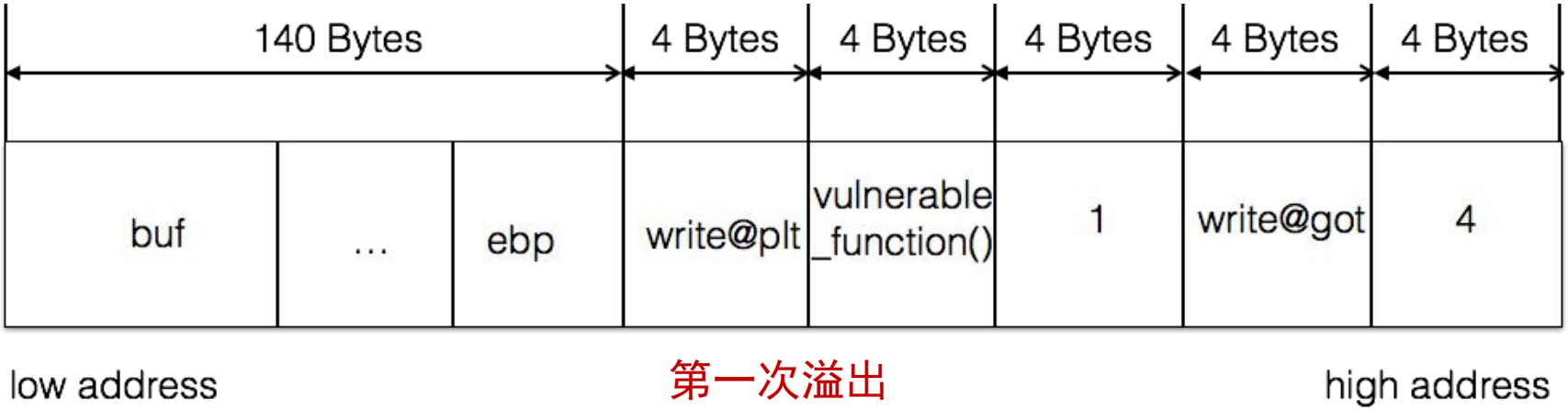
2.打开shell后，程序返回回到venerable function，以使被攻击的系统难以察觉到自己已遭到攻击。

上图为攻击目标最理想的情况，但是由于随机化，无法获得system()的地址和/bin/sh的地址。

ret2plt攻击：一个攻击实例



ret2plt攻击：一个攻击实例



call write() →

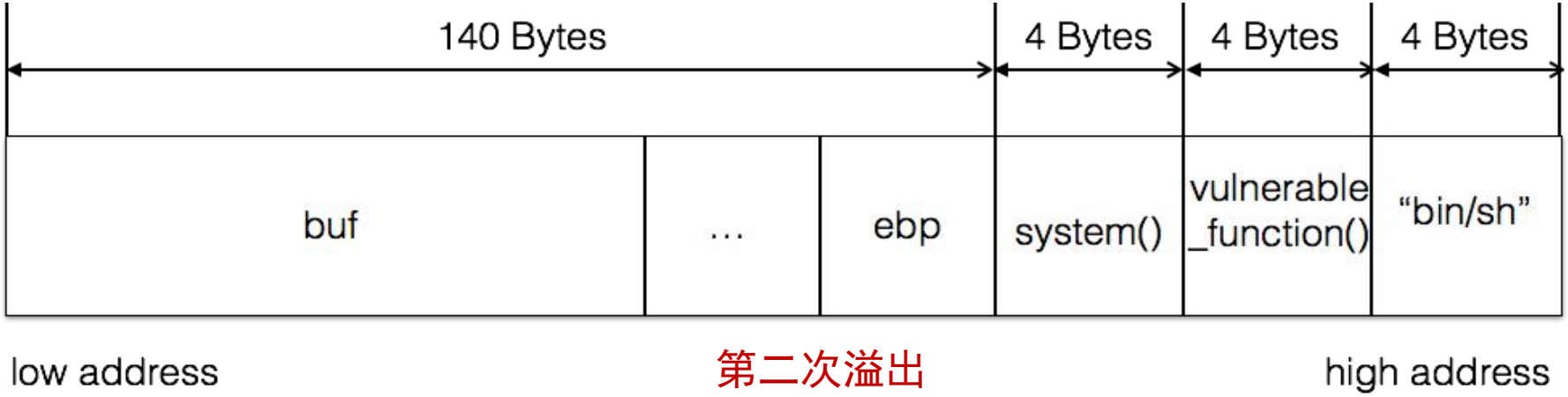
```
PLT0:
  push *(GOT + 4)    //模块ID号
  jump *(GOT + 8)    //ld处理函数_dll_runtime_resove()
…
write@plt:
  jump *(write@got)    //初始时指向“push n”指令
  push n               //write在.rel.plt的下标
  jump PLT0
```

ret2plt攻击：一个攻击实例

- write() ✓
- 通过偏移量，求出system() 和 “bin/sh”的地址

```
system_addr = write_addr - (libc.symbols['write'] -
libc.symbols['system'])
print 'system_addr= ' + hex(system_addr)
binsh_addr = write_addr - (libc.symbols['write'] -
next(libc.search('/bin/sh')))
print 'binsh_addr= ' + hex(binsh_addr)
```

- system() ✓
- “bin/sh” ✓



ret2libc攻击小结

- ret2libc攻击小结：
  - 以**库函数**为配件，以ret为配件之间的连接。
  - 能够实现**以任意参数调用任意库函数**。
  - ret2plt和ret2libc的本质是一样，只不过ret2plt不需要知道库函数的具体地址。

ret2libc攻击小结

- ret2libc和ret2plt都是通过压栈来传递函数参数，这也是i386架构（即x86）的调用约定。
- 但是，根据X86-64的调用约定，函数间传递参数不再以压栈的方式，而是以寄存器方式传递参数。
  - 函数的前6个参数依次以通用寄存器rdi，rsi，rdx，rcx，r8和r9来传递。
- 随着x86-64的普及，ret2libc和ret2plt攻击逐渐失去了作用。
  - 要么寻找额外的漏洞，控制通用寄存器来传递参数。
  - 要么开发一种新的攻击方法。

内容概要

- 代码复用攻击
  - ret2libc
  - ROP
  - JOP
- 对代码复用攻击的防御
  - ASLR
  - CFI
- 总结

ROP攻击简介

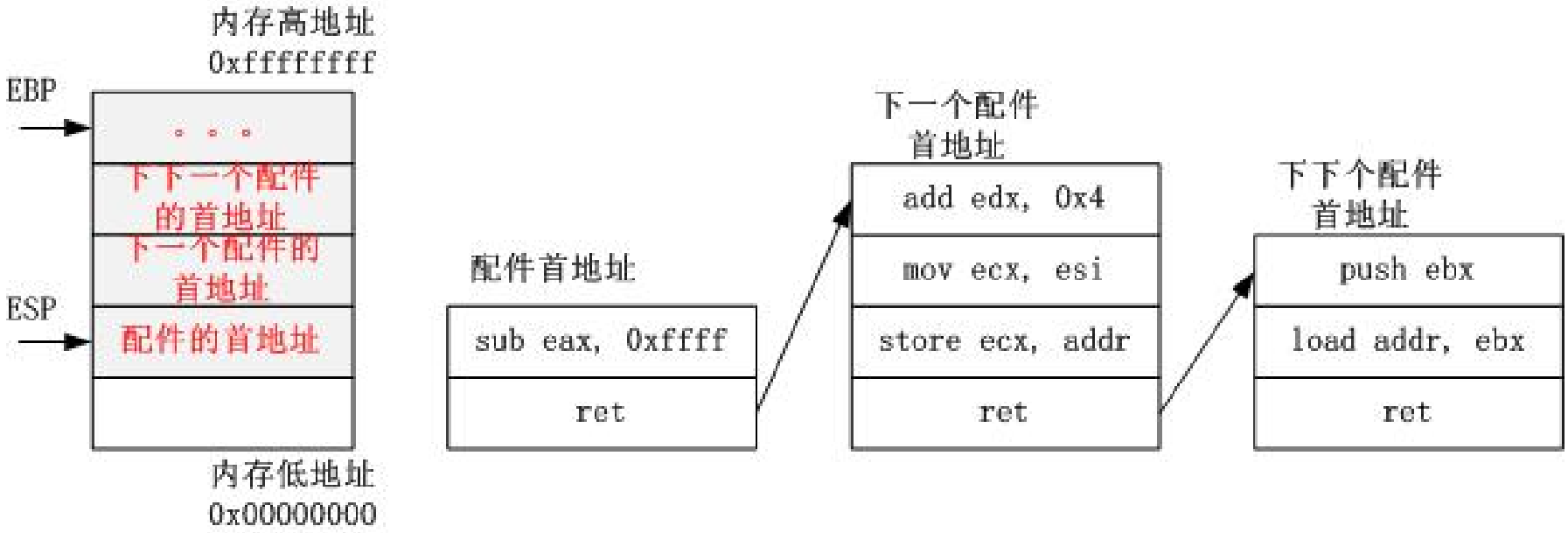
- ROP攻击是代码复用攻击的里程碑式的工作，标志着代码复用攻击的正式出现。
- 当时，NX已经提出，并逐渐开始在现实世界中大规模的推广，代码注入攻击逐渐受到了NX防御的限制。
- 因此，研究者于2007年提出了ROP攻击，标志着代码复用攻击正式代替代码注入攻击，成为了学术界研究的最主流攻击方法。

ROP攻击的原理

- ROP (Return-oriented programming)，面向返回的编程方法。简单来说，就是以ret为结尾的代码片段作为配件，并以ret作为不同配件之间的连接。
- ret2libc攻击：
  - 以库函数为配件。
  - 用ret进行连接。
- ROP攻击：
  - 以ret为结尾的代码片段为配件。
  - 用ret进行连接。

ROP攻击的原理

- 利用栈溢出漏洞，覆盖栈上函数返回地址，从而控制ret跳转地址，连接多个以ret结尾的配件，构造配件链，最终实现ROP攻击。



ret的作用

攻击者特别喜欢利用ret进行攻击。

- 为什么选择ret?
  - 因为ret指令的跳转目标是由栈上的内容决定的（函数返回地址保存在栈上）。
  - 栈溢出漏洞是最为常见的内存漏洞，栈的内容很容易被攻击者所控制。
  - 程序中ret指令的数量极多，能够轻易找到许多可用的配件。

ret的作用

攻击者特别喜欢利用ret进行攻击。

- 配件的参数如何控制?
  - ret指令前通常有一些pop指令。pop指令能够将栈上的数据弹出到对应的寄存器中。
  - 利用ret前面的这些pop指令，实现对参数的控制（此时栈上数据已经被攻击者完全控制）。

ret的作用

攻击者特别喜欢利用ret进行攻击。

- ret指令前的代码片段是否会因为功能单一，而无法实施预期的攻击目标呢?
  - 很方便利用ret实现一连串的控制流劫持。
    - 一段ret指令无法满足攻击的需求，但是可以控制ret指令跳到另一段ret指令序列。如果还达不到目标，再跳到另一段ret指令序列，直到达成攻击目的。
  - 已经有研究充分证明了ROP方法是图灵完备的。也就是说，ROP攻击可以复用系统中已有的代码实现任何逻辑功能。

ROP攻击的分析

- ROP攻击的优点：
  - 相对于ret2libc，ROP**更加灵活**。
  - ROP脱离函数，完全使用代码片段，对程序运行控制也**更加精细**。
- ROP攻击的缺点：
  - 过于依赖于ret指令
  - 过于依赖栈溢出漏洞

ROP攻击：一个攻击实例

- 攻击目标为实现system(“echo success”) 这个函数调用。
- 在这里省去“寻找ret地址在栈上的位置”以及漏洞代码分析，直奔主题，如何构造ROP指令序列来实现攻击逻辑。

ROP攻击：一个攻击实例

- 1）首先，system()参数为“echo success”字符串的地址，而字符串是栈上注入的内容，那它的地址值应该等于%rsp + offset。
- 2）system()执行时，参数由rdi传递。在“retq”或者“call *reg”指令前找到行为特征与“%rdi = %rsp + offset”逻辑等价的指令序列：

```
0x7ffff7a610a3:      lea 0x120(%rsp),%rdi
0x7ffff7a610ab:      call %rax
```

- 3）将“echo success”字符串安排在%rsp + 0x120的位置。但是，后一条指令执行“call %rax”，需要在将%rax的值改为system()函数的地址。

ROP攻击：一个攻击实例

- 4）想将system()函数地址放到%rax相当容易，只需要在“retq”指令前找到“pop %rax”指令即可。最终发现如下指令片段：

```
0x7ffff7a3b076:    pop %rax
0x7ffff7a3b077:    pop %rbx
0x7ffff7a3b078:    pop %rbp
0x7ffff7a3b0f9:    retq
```

ROP攻击：一个攻击实例

- 5）于是，构建如下的ROP配件链:
- 配件1:

```
pop %rax        //这里弹出system函数地址
pop %rbx
pop %rbp
retq            //这里从栈中跳到下一个配件
```

- 配件2:

```
lea 0x120(esp), %rdi    //需要安排好" echo success"位置，
使得此时的rsp + 0x120刚好是字符串地址
call *%rax              //调用system()，完成攻击。
```

ROP攻击：一个攻击实例

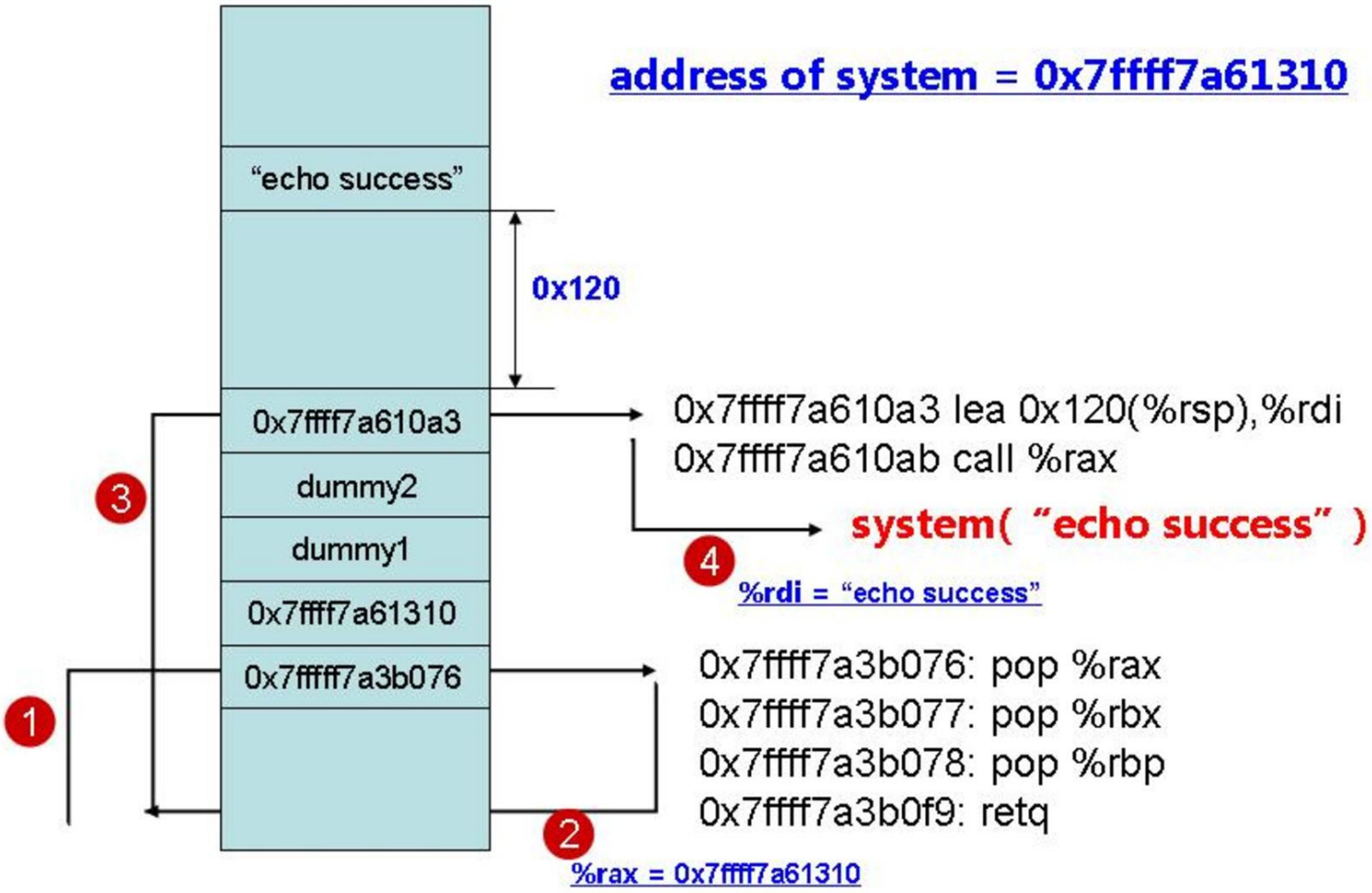
- 6）通过gdb得到：
  - system()函数的地址：0x7ffff7a61310
  - 配件1的地址：0x7ffff7a3b076
  - 配件2的地址：0x7ffff7a610a3
- 于是向栈中注入以下内容（从ret地址开始）：

高地址

0x7ffff7a3b076	address of system(): 0x7ffff7a61310	dummy1 (8 bytes)	dummy2 (8 bytes)	0x7ffff7a610a3	dummy3 (120 bytes)	“echo success”

ROP攻击：一个攻击实例

- 指令的执行过程和栈注入内存布局：



ROP攻击小结

- ROP攻击小结：
  - 以ret为结尾的代码片段为配件，以ret为配件之间的连接。
  - 以覆盖栈上数据来控制ret的跳转（配件的连接）。
  - 是图灵完备的攻击方法。

内容概要

- 代码复用攻击
  - ret2libc
  - ROP
  - JOP
- 对代码复用攻击的防御
  - ASLR
  - CFI
- 总结

JOP攻击简介

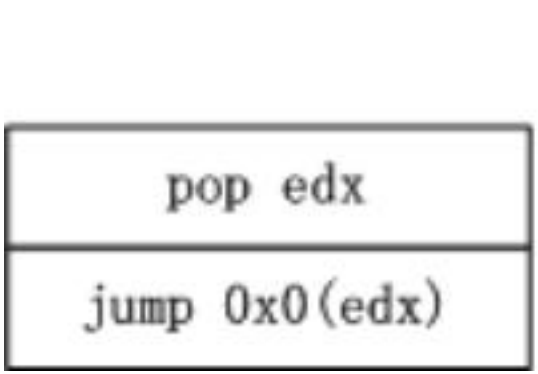
- 自从ROP提出以后，研究者对代码复用攻击进行了深入的研究和分析。
- 间接跳转指令一共有ret，call和jump三种。
- 因此，2010年提出了JOP，使用jump作为配件的连接。
  - 同样，以call作为配件链接的COP也是存在的。

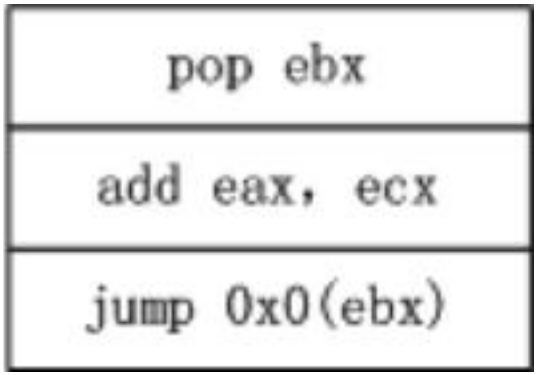
JOP攻击原理

- JOP（Jump-oriented programming），面向跳转的编程方法。简单来说，就是以jump为结尾的代码片段作为配件，并以jump作为不同配件之间的连接。
- ROP攻击：
  - 以ret为结尾的代码片段为配件。
  - 用ret进行连接。
- JOP攻击：
  - 以jump为结尾的代码片段为配件。
  - 用jump进行连接。

JOP攻击原理

- JOP的配件以jump为结尾。想要控制jump的跳转目标，必须要控制**通用寄存器**。
- 常见的方法是利用pop指令和load指令。
  - 通过**栈溢出漏洞控制栈上的数据**，然后利用**pop指令**，将栈上的数据弹出到通用寄存器中。
  - 通过**修改内存数据**，利用**load指令**，将内存数据存入通用寄存器中。





JOP攻击原理

- JOP的配件分为两大类：
  - **功能配件**(functional gadget)
    - 完成某种**特定功能**的代码片段，且以间接跳转指令结尾。相当于ROP中的普通配件。
    - 当功能配件执行完成后，需要跳回调度配件。
  - **调度配件**(dispatcher gadget)
    - 充当程序EIP的作用，实现**控制流的转移**。
    - 负责组织功能配件的执行。

调度配件

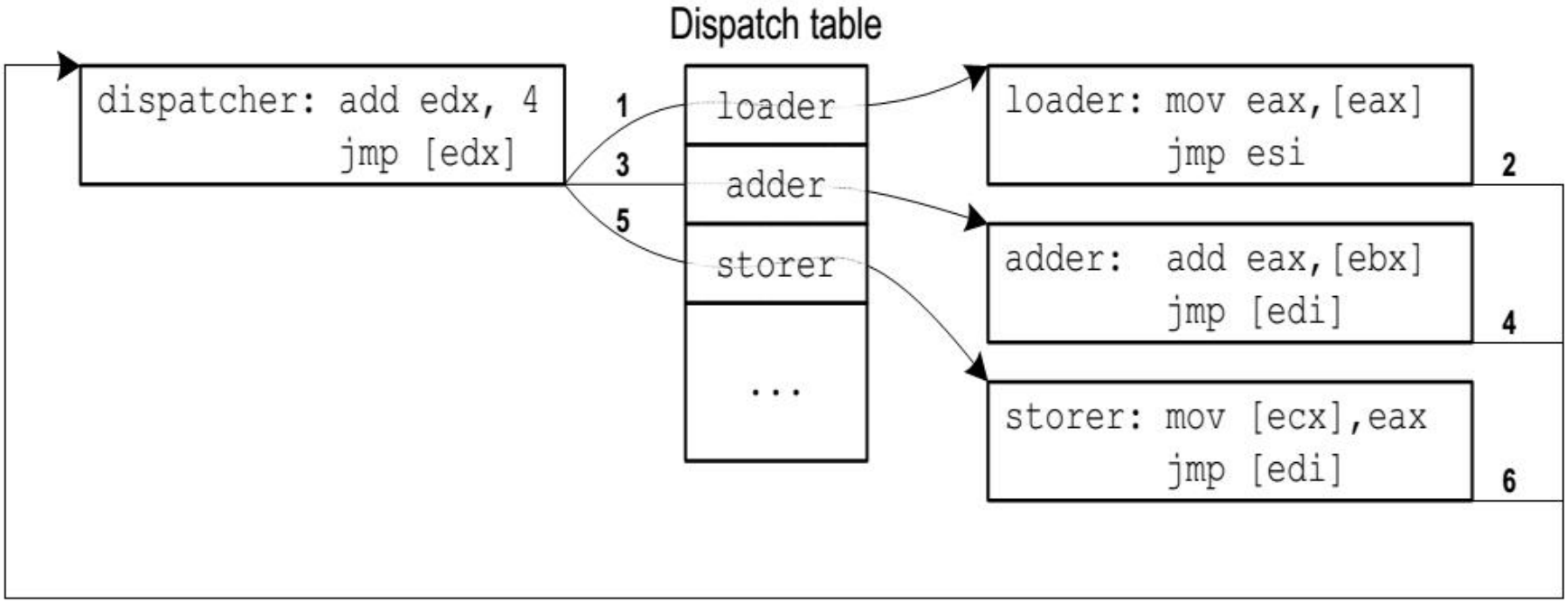
- 为了完成调度功能，调度配件应该具有以下特征：
  - 调度配件能够控制ip。
  - 调度配件能够跳转到可控的内存地址。

$$pc \leftarrow f(pc);$$
$$goto \ *pc;$$

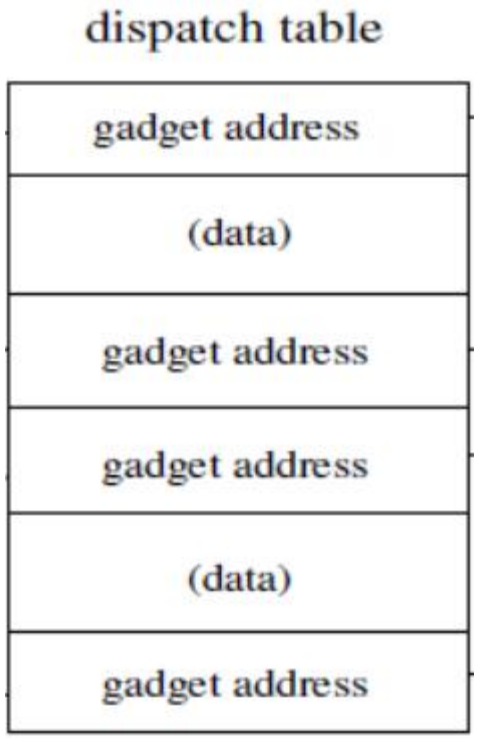
- pc（program counter）是程序计数器，也就是ip，指令寄存器。
- 简单来说，调度配件就是用来**计算并跳转**到下一个功能配件对应的地址。

调度配件

- 如果一个调度配件的 f(pc) = pc + 4，那么就可以利用这个调度配件来管理JOP配件的具体运行和连接。

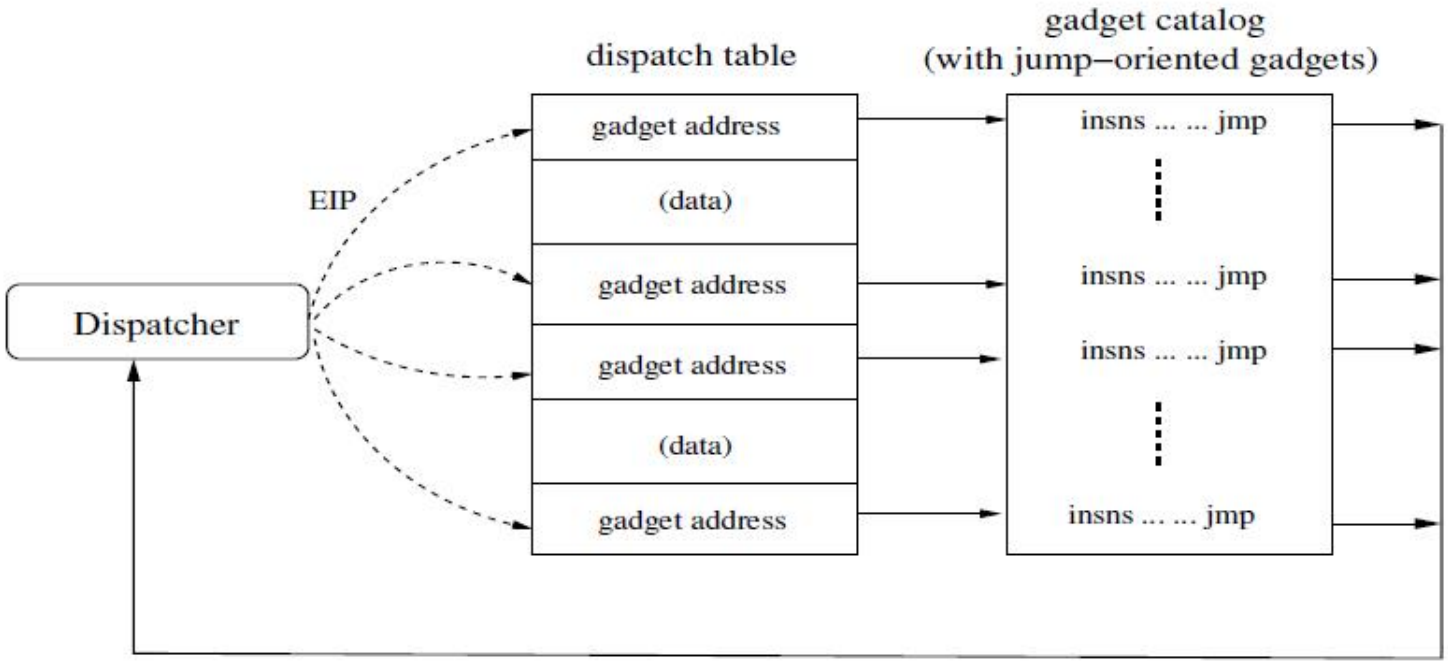


- 配件调度表（dispatch table），用于存放**配件的地址和参数**。
  - 调度表可以存放在内存的**任意可写区域**，不一定在栈上。



JOP攻击过程

- JOP攻击的具体过程：
  - 1）攻击者向进程内存中注入调度表，跳转到调度配件。
  - 2）由调度配件从调度表中获取功能配件的地址，跳转到功能配件执行。
  - 3）功能配件从调度表中获取所需数据，执行完操作后，返回调度配件。



JOP攻击过程

- JOP可以循环，但是无法自动启动。
- 将JOP攻击过程分为两个部分：
  - JOP的**初始化**：利用其它漏洞，注入调度表，配置必要寄存器，跳转到调度配件。
  - JOP的**循环过程**：调度配件->功能配件->调度配件->功能配件。。。

JOP攻击分析

- 优点：
  - JOP用jump替代了ret，不需要使用ret，攻击更加隐蔽。
  - JOP的配件调度表可以存放在内存其他位置，如堆区。
- 缺点：
  - JOP更加复杂，构造JOP链难度更大。
  - JOP无法自动启动，需要一个初始化的过程。

JOP攻击示例

- 环境：Ubuntu 16.04, 64位
- 工具：ROPGadget，用于寻找可用的配件
- 目标：利用JOP攻击执行系统调用execv(“/bin/sh”)。
  - 系统调用号为0x3b，将系统调用号保存到寄存器rax中。
  - 参数“/bin/sh”保存在rdi寄存器中。
  - rdx和rsi中置0

JOP攻击示例

- 调度配件（包含JOP的初始化过程）
  - 其中rax,rcx指向调度器配件，rbx指向调度表
  - 前3个pop是对这三个寄存器进行初始化
  - add和jmp指令是不断从调度表中取地址，循环调用功能配件

dispatcher gadget:

```
L1:     pop %rcx
        pop %rax
        pop %rbx
L2 :    add $0x08,%rbx
        jmpq *(%rbx)
```

JOP攻击示例

- 功能配件
  - 共有7个功能配件
  - 每一个功能配件实现对寄存器的赋值，并通过jmp或call指令返回调度器配件

functional gadget:

```
JMP1:   pop  %rdi; jmp  %rax
JMP2:   pop  %rsi; pop  %r15; jmp %rax
JMP3:   pop  %r12; jmp %rax
JMP4:   mov %r12, %rdx; call %rax
JMP5:   pop %rax; jmp %rcx
JMP6:   pop %rax; jmp %rcx
JMP7:   Syscall
```

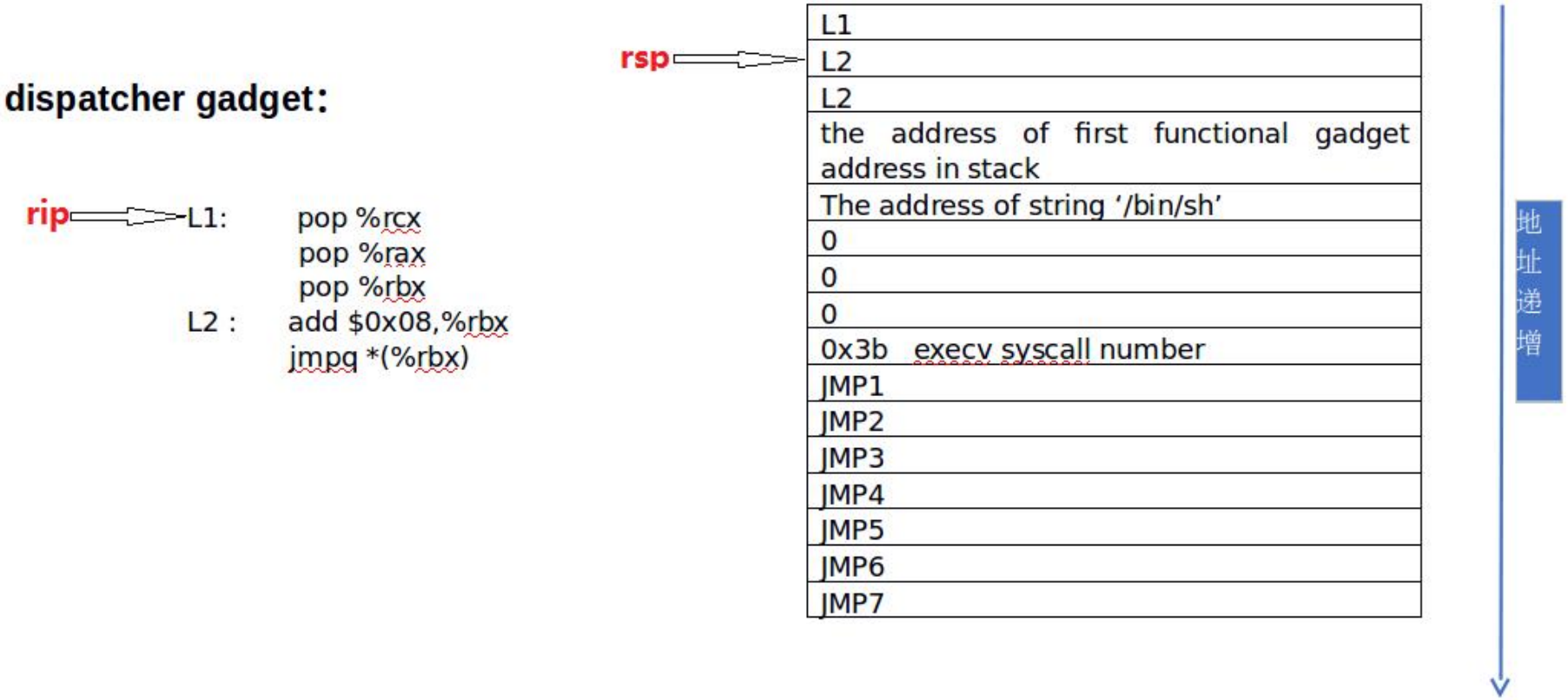
JOP攻击示例

- 调度表
  - 此调度表存放在栈中，当然也可存放于内存的其他可写区域
  - 其中L1，L2，JMP1-7均表示配件的地址

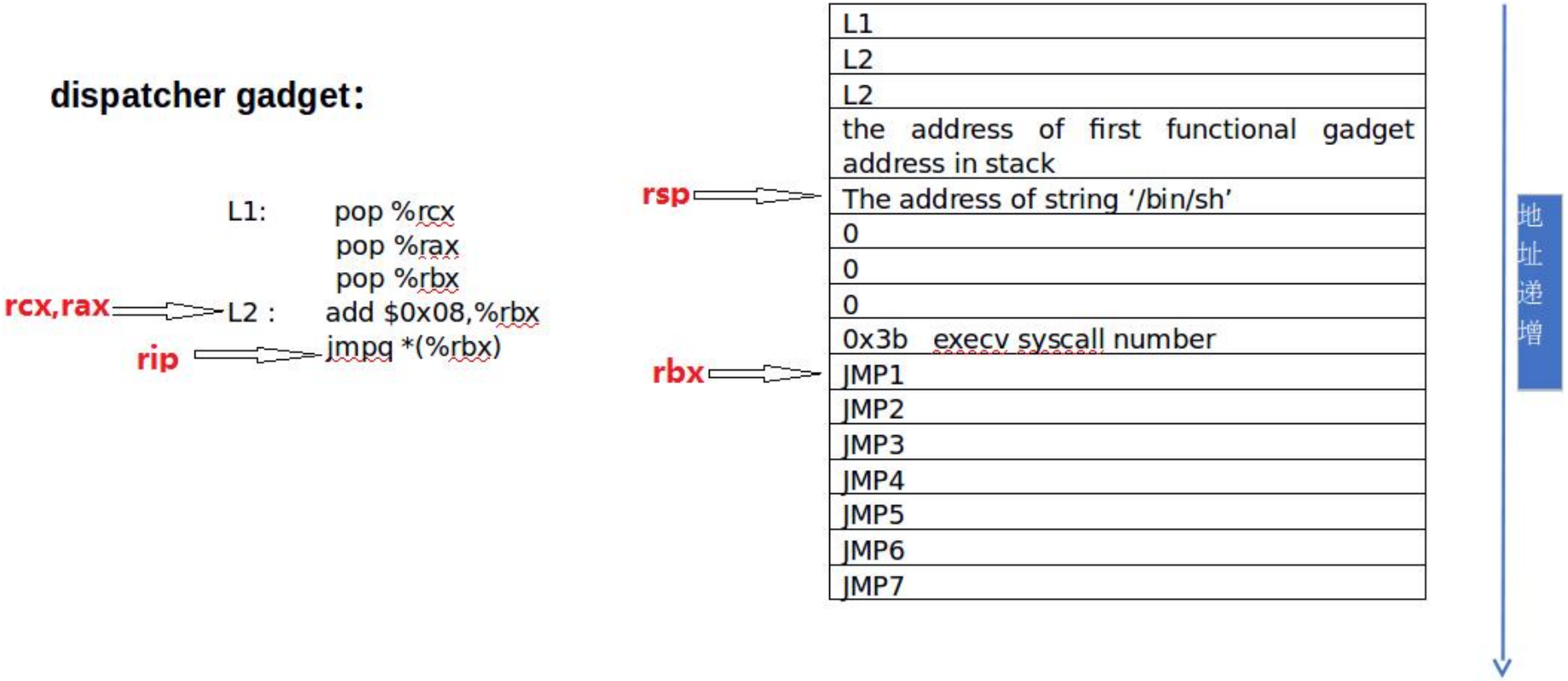
L1
L2
L2
the address of first functional gadget address in stack
The address of string '/bin/sh'
0
0
0
0x3b execv syscall number
JMP1
JMP2
JMP3
JMP4
JMP5
JMP6
JMP7

JOP攻击示例

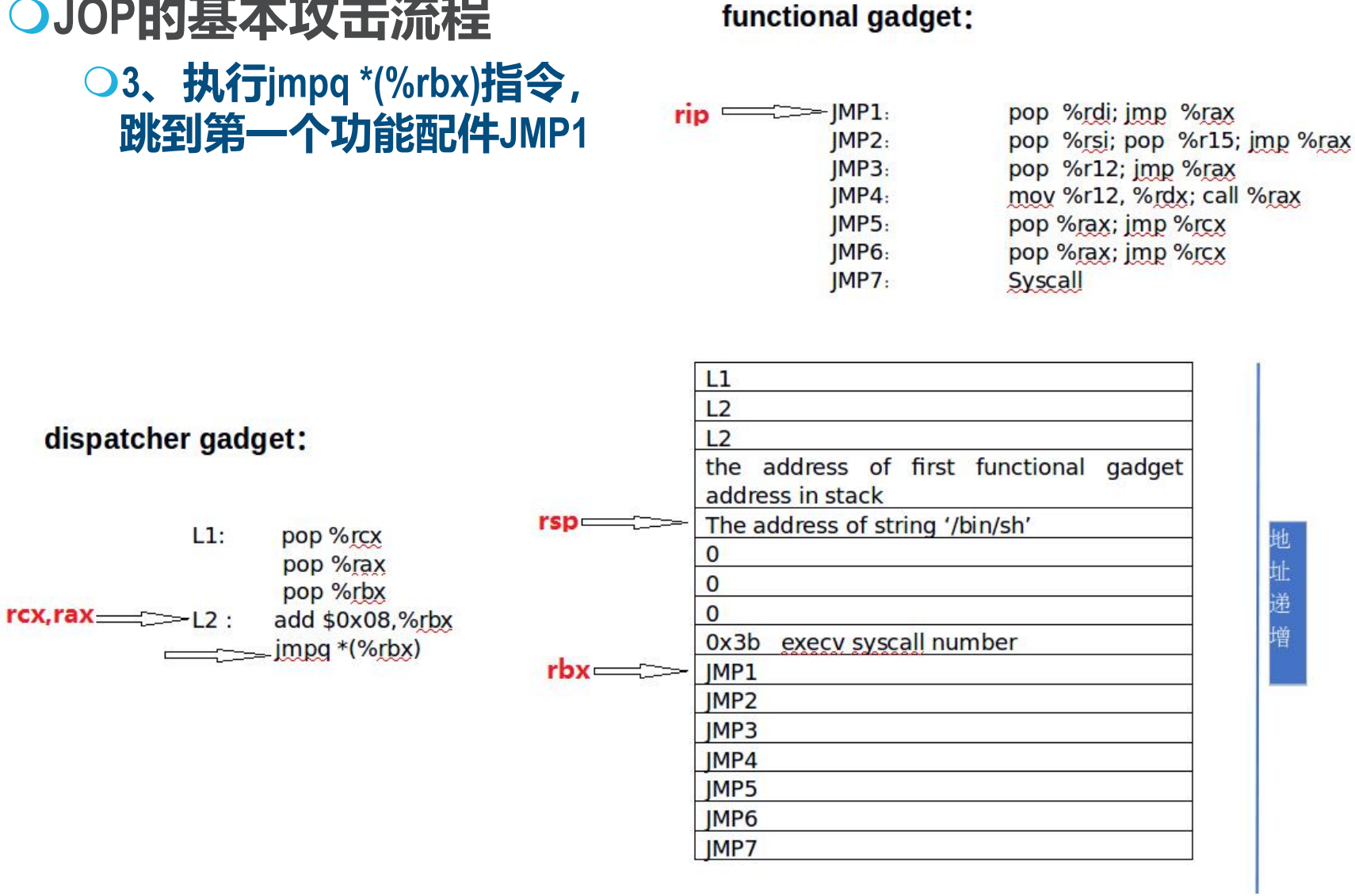
- 以上介绍了本次攻击所需的调度配件，功能配件，调度表
- JOP的基本攻击流程：
  - 1、通过缓冲区溢出漏洞使得rip指向L1，开始执行调度器配件



- JOP的基本攻击流程
  - 2、执行完3个pop指令和add指令之后，rcx,rax指向L2，rbx指向栈中JMP1



- JOP的基本攻击流程
  - 3、执行jmpq *(%rbx)指令，跳到第一个功能配件JMP1



JOP攻击示例

- JOP的基本攻击流程
  - 4、执行第一个功能配件pop %rdi，此时rdi寄存器指向字符串"/bin/sh"
  - 5、然后，执行jmp %rax，跳回调度器配件。

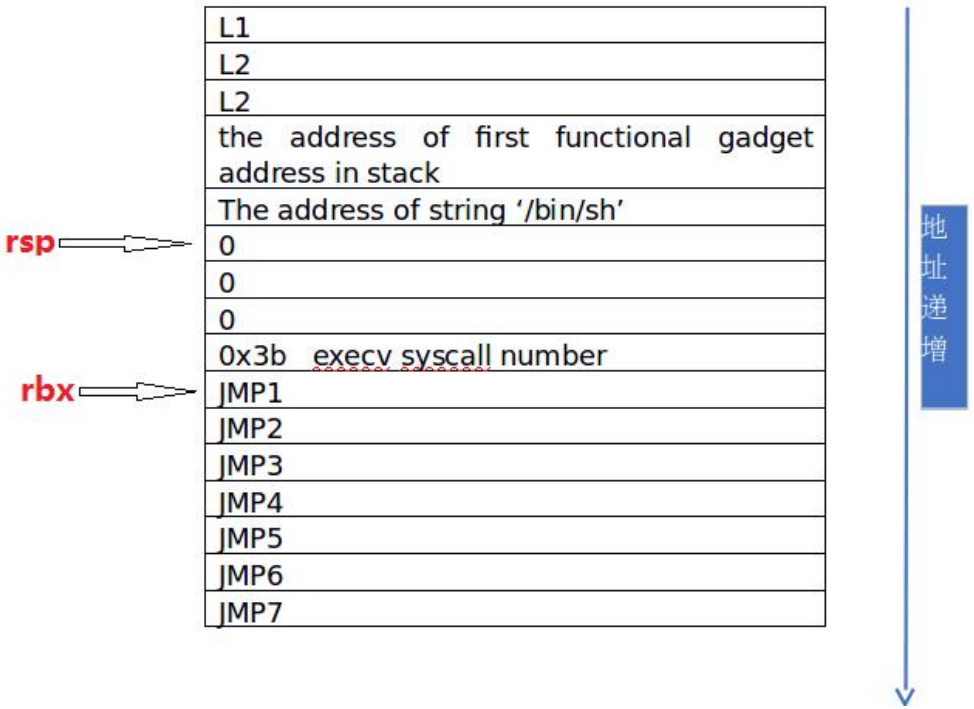
functional gadget:

```
JMP1:  pop  %rdi; jmp  %rax
JMP2:  pop  %rsi; pop  %r15; jmp %rax
JMP3:  pop  %r12; jmp %rax
JMP4:  mov %r12, %rdx; call %rax
JMP5:  pop %rax; jmp %rcx
JMP6:  pop %rax; jmp %rcx
JMP7:  Syscall
```

dispatcher gadget:

```
L1:   pop %rcx
      pop %rax
      pop %rbx
L2 :  add $0x08,%rbx
      jmpq *(%rbx)
```

rip,rcx,rax → L2

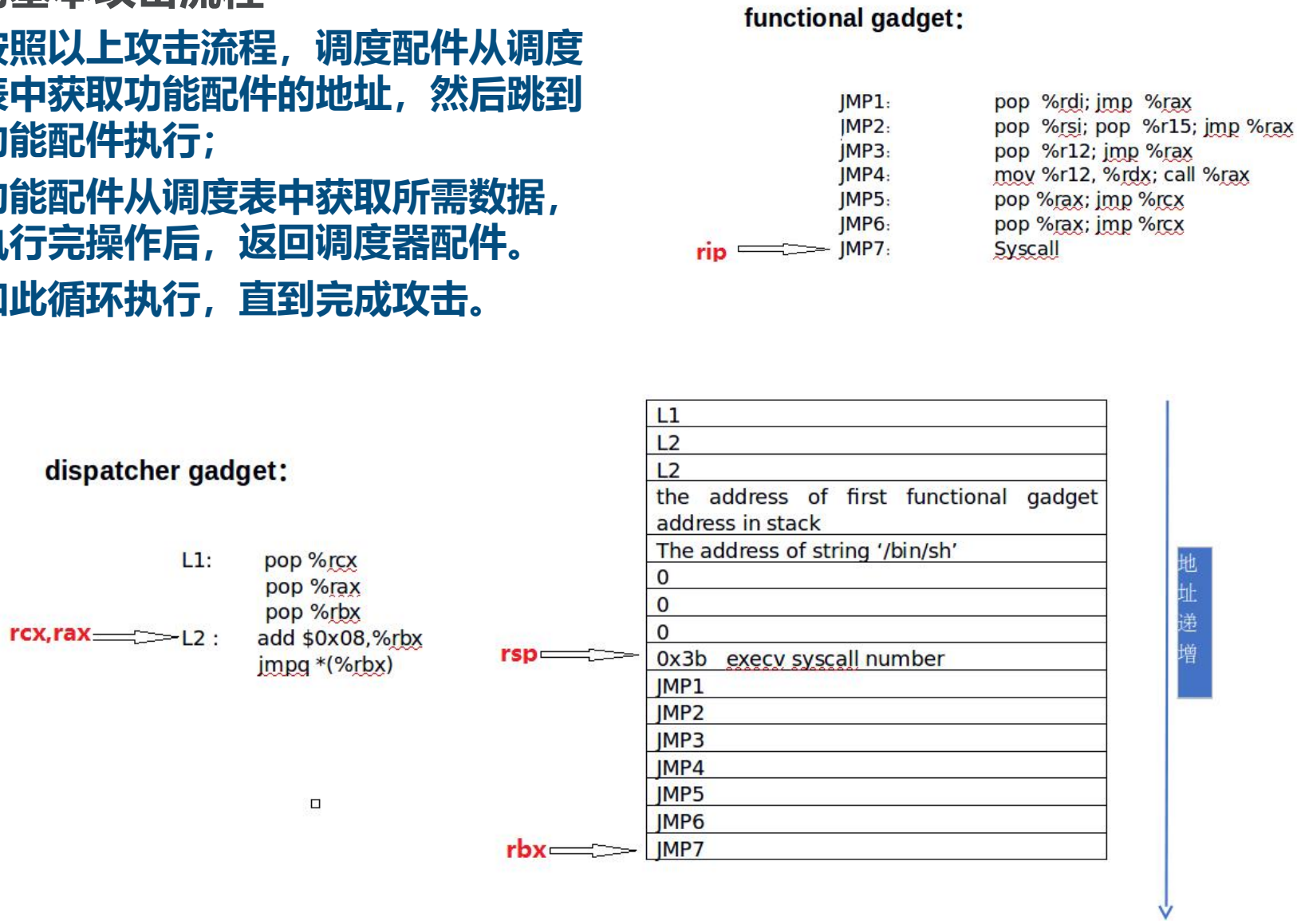


JOP攻击示例

- JOP的基本攻击流程
  - 6、执行调度器配件add $0x08,%rbx，将rbx加8指向下一个地址
  - 7、执行jmp *(%rbx)，跳到第二个功能配件执行

JOP攻击示例

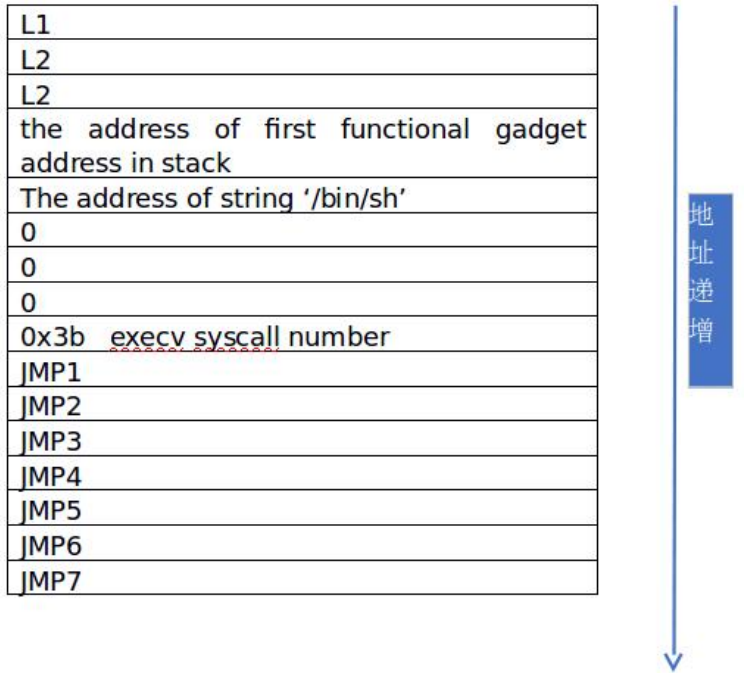
- JOP的基本攻击流程
  - 按照以上攻击流程，调度配件从调度表中获取功能配件的地址，然后跳到功能配件执行；
  - 功能配件从调度表中获取所需数据，执行完操作后，返回调度器配件。
  - 如此循环执行，直到完成攻击。



JOP攻击示例

- JOP的基本攻击流程
  - 每一个功能配件都完成一定的操作，最终使得：
    - 寄存器rdi指向字符”/bin/sh”
    - 寄存器rdx和rsi置0
    - 寄存器rax存放execv系统调用号0x3b
  - 执行完所有的功能配件之后，通过最后一个功能配件syscall指令执行系统调用execv(“/bin/sh”)，JOP攻击完成。

```
JMP1:   pop  %rdi; jmp  %rax
JMP2:   pop  %rsi; pop  %r15; jmp %rax
JMP3:   pop  %r12; jmp %rax
JMP4:   mov %r12, %rdx; call %rax
JMP5:   pop %rax; jmp %rcx
JMP6:   pop %rax; jmp %rcx
JMP7:   Syscall
```



JOP攻击小结

- JOP攻击小结：
  - 以jump为结尾的代码片段为配件，以jump为配件之间的连接。
  - 有专门的调度配件，有专门的调度表负责配件之间的连接和参数设置。
  - 也是图灵完备的攻击方法。

代码复用攻击小结

- 代码复用攻击基本类型和对应配件类型：
  - ret2libc：以库函数为配件
  - ROP：以ret结尾代码片段为配件
  - JOP：以jump结尾代码片段为配件
  - COP：以call结尾代码片段为配件
- 在实际环境中，不需要过于关注攻击的名称，可以混合使用，将不同类型的配件结合起来。通常以ROP为统称。
- 需要注意的是，不同配件利用方法有所不同，需要仔细的配置。

内容概要

- 代码复用攻击
  - ret2libc
  - ROP
  - JOP
- 对代码复用攻击的防御
  - ASLR
  - CFI
- 总结

代码复用攻击的特征

- 代码复用攻击和程序正常执行的最本质区别是代码执行的顺序，即**控制流不同**。
  - 代码复用攻击完全改变了程序执行顺序，需要在系统已有代码中不断跳转。
- 根据代码复用攻击的特征，研究者提出两种主要的防御思路：
  - **随机化方法。**
  - **异常行为检测。**

代码复用攻击的防御

- 代码复用攻击的防御方法：
  - **随机化方法：**
    - 代码复用攻击由配件链组成，配件是系统中已有的代码片段。因此，代码复用攻击需要获知系统中代码的**具体地址**。
    - 如果将系统中代码地址**随机化**，攻击者就无法找到对应配件的真实地址，从而无法进行攻击。
  - **最典型的随机化方法就是ASLR，Address Space Layout Randomization（地址空间布局随机化）**

代码复用攻击的防御

- 代码复用攻击的防御方法：
  - **异常行为检测：**
    - 代码复用攻击的代码执行顺序和正常程序顺序**完全不同**。
    - 因此，可以分析检测程序执行过程，如果发现程序执行过程和正常过程不同，就认为发生了攻击。
  - **最典型的异常行为检测方法就是控制流完整性，(Control-Flow Integrity, CFI)**

内容概要

- 代码复用攻击
  - ret2libc
  - ROP
  - JOP
- 对代码复用攻击的防御
  - ASLR
  - CFI
- 总结

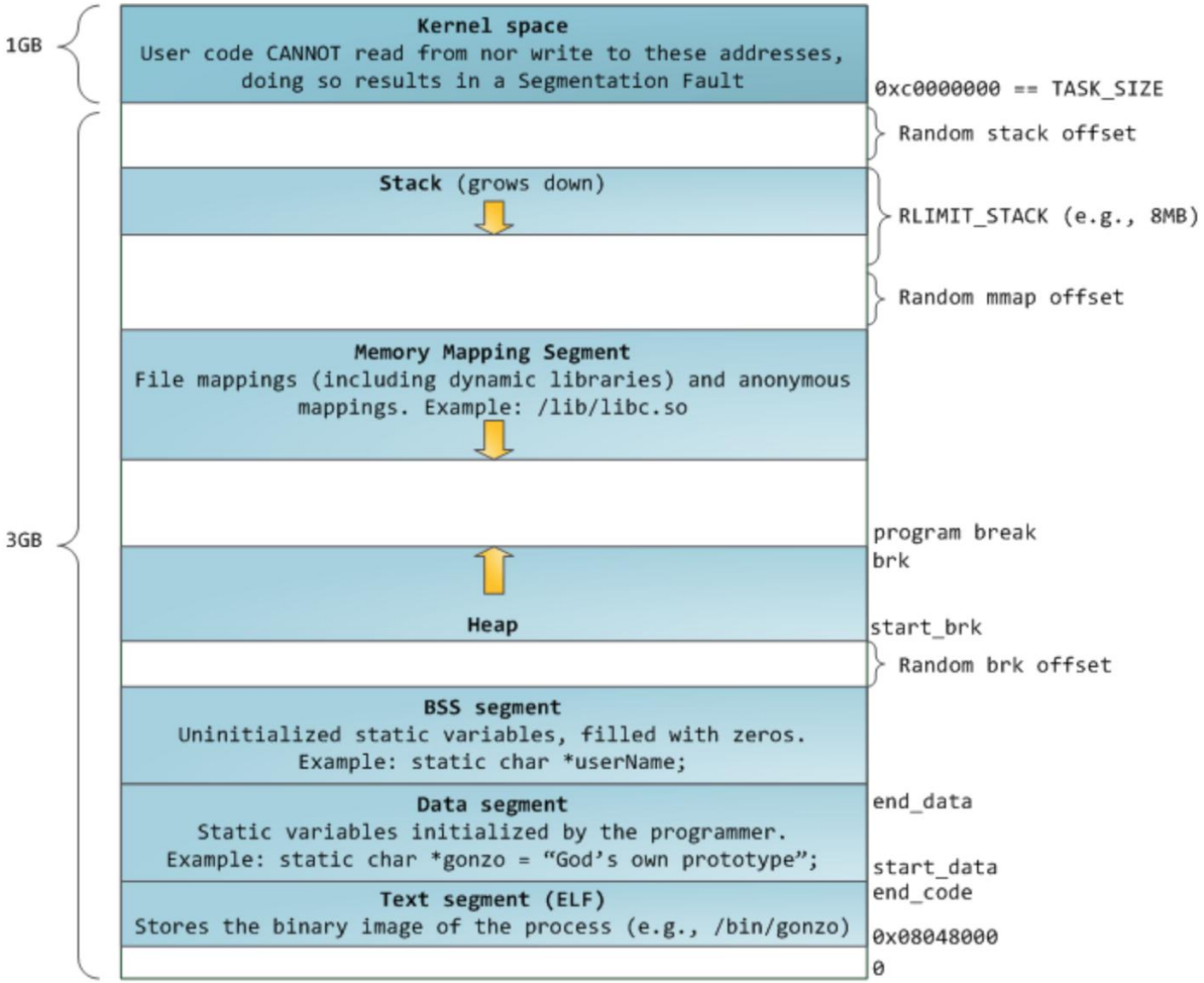
攻击的前提

- 操作系统每次加载进程和动态链接库时，进程和动态链接库的**基地址**都加载到**固定的内存地址**，即程序每次运行时虚拟地址空间布局都是一模一样的。
- 因此，攻击者能够很轻易的获知程序的整个内存布局，知道程序代码的具体地址，知道栈中数据的具体排布。

攻击的前提

典型内存布局

- 内核空间
- 栈
- 共享链接库
- 堆
- 静态数据区
- 代码区



攻击的前提

- 无论是代码注入攻击，还是代码复用攻击，都需要知道进程的内存布局。
  - 代码注入攻击，需要获知函数返回地址在栈中的具体位置，还需要知道注入代码的首地址。
  - 代码复用攻击需要知道每一个配件的具体地址。
- 如果进程每次运行，其内存布局是一样的。那么，只需要先运行一次程序，利用gdb等获得程序内存布局。
- 之后，就可以一直根据已知的内存布局进行攻击。

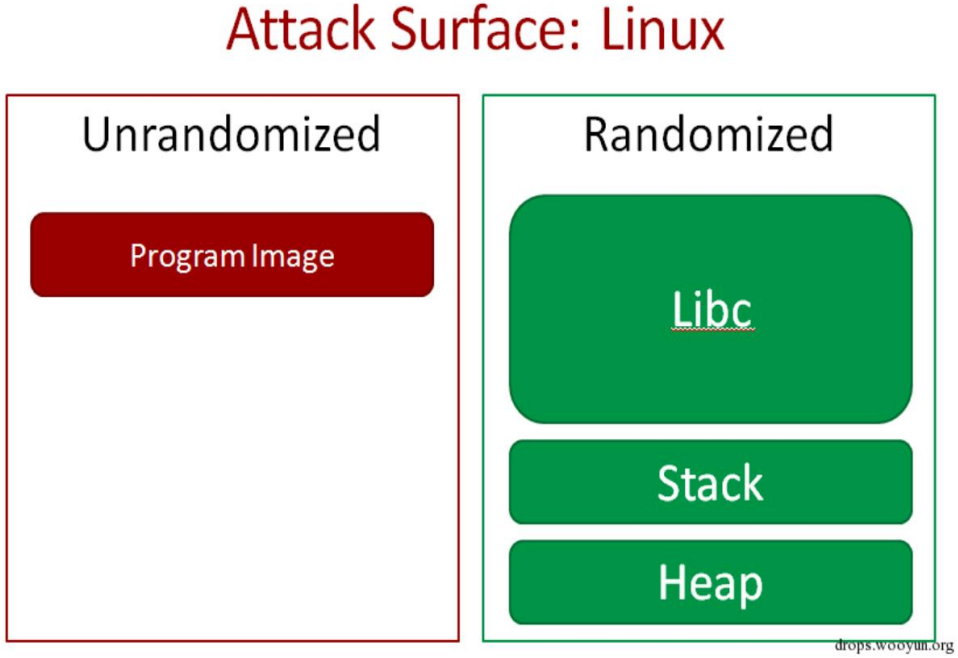
ASLR的原理

- ASLR，Address Space Layout Randomization（地址空间布局随机化），就是将进程的内存地址空间随机化的一种方法。
- ASLR通过对堆、栈、共享库映射等线性内存区域布局的随机化，防止攻击者直接定位攻击代码位置，从而增加攻击者预测目的地址的难度。

ASLR的原理

- ASLR是一种位于操作系统的安全机制，已经被主流的操作系统实现，如Linux系统。
  - 通常来说，系统自动进行地址随机化的只有进程的**堆、栈、以及库函数**。
  - 若要使**代码段**也进行地址随机化，在Linux系统中，使用GCC编译时需要加“--pie”选项。

- 由于ASLR的开启，相同程序再次被执行后，其地址空间不会和之前的一次完全一样。



代码段的随机化

- 代码段的随机化PIE (Position Independent Executables)
- 代码段随机化需要在编译的时候开启PIE选项，将程序编译成位置无关代码，并链接为ELF共享对象，使其每次加载的时候地址都不一样。
  - gcc –fpie –pie –o hello hello.c
  - -fpie选项在编译时使用
  - -pie选项在链接ld时使用

PaX：ASLR的一个实例

- PaX是Linux操作系统的一组安全补丁，其中就包含了ASLR的实现。
- PaX将用户内存地址空间分为三个部分：executable，mapped，stack，每个区域在映射的时候偏移一个随机变量。
  - Executable：16bit随机化（X86）
    - Program code，uninitialized data， initialized data
  - Mapped：16bit随机化
    - Heap，dynamic libraries，thread stacks，shared memory
  - Stack：24bit随机化
    - Main user stack

PaX：ASLR的一个实例

- PaX，Stack：**用户栈**地址随机化
- 内核创建进程时会调用execve()系统调用，此时便会创建用户栈，通常情况下用户栈映射的虚拟地址是0xBFFFFFFF，PaX则为其产生一个**随机的基地址**。
- PaX同时随机化其分配的**区间长度**。

PaX：ASLR的一个实例

- PaX，Stack：**内核栈**地址随机化
- Linux为每个进程分配内核栈，当系统调用、中断和异常陷入内核时使用。
- PaX随机化每个进程的内核栈指针
  - 5bit随机化
- **每次系统调用随机化都不同**
  - 用户栈只在进程第一次被调用时随机化

PaX：ASLR的一个实例

- PaX：mmap随机化，即堆空间的随机化
- Linux为每个进程分配堆空间时会调用do_mmap()函数，其会在进程非映射区寻找第一个足够大的区间分配给该进程。
- PaX在其寻找新区间之前在其所要分配的基地址上加入了一个随机变量delta_mmap。
  - 16bit随机化

PaX：ASLR的一个实例

- PaX：**可执行区**随机化
- 将每个ELF二进制文件的映射区间随机化
- 若是二进制文件被链接时设置了其固定的加载地址，省略了其可重定位信息
  - PaX将该二进制文件映射到固定区，但将其属性设为不可执行，并随机创建了一份可执行的拷贝镜像
  - 访问固定映射区时会产生page fault
  - 若是安全的，则page handler会将其重定向到那份拷贝的镜像

ASLR的分析

- 优点：
  - 简单有效，损耗很小，防御效果不错
- 缺点：
  - ASLR只在进程加载时进行**一次随机化**，之后整个程序运行过程，内存空间排布保持不变，容易被内存信息泄露破解。
  - ASLR只随机化了整个segment的**基地址**，它们内部的**相对位置却没有变**，攻击者可以通过一个已知地址信息推导出segment中其他信息。

ASLR小结

- 本节介绍的ASLR是最经典的最基本的随机化方法，由于原理简单、防御效果好，已经被主流操作系统所采用。
- 目前，随机化防御方法仍然是目前热门的研究方向之一。
- 围绕破解或增强随机化防御方法的研究有很多，下一讲会进一步介绍。

内容概要

- 代码复用攻击
  - ret2libc
  - ROP
  - JOP
- 对代码复用攻击的防御
  - ASLR
  - CFI
- 总结

异常行为检测

- 前提：代码复用攻击和正常程序的执行过程差别很大。

- **问题：**代码复用攻击和正常程序执行有哪些非常明显的差别？

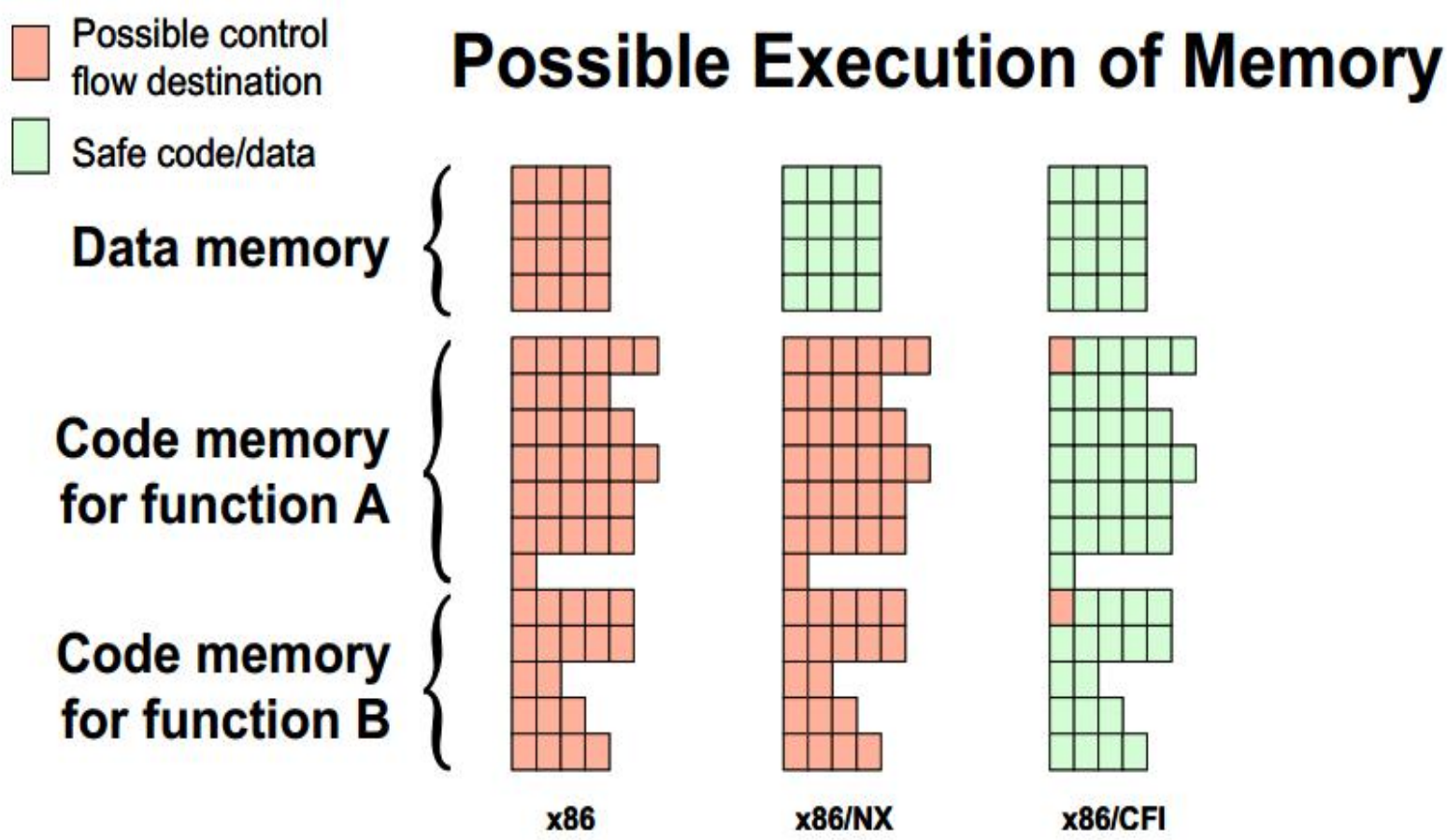
CFI的原理

- CFI（Control-Flow Integrity，控制流完整性），在2005年被首次提出。
- 首先，通过分析正常程序行为，得到正常程序的控制流图。
- 然后，分析当前程序行为，判断当前程序的控制流是否符合正常。
  - 如果不符合，则认为发生了异常，需要终止程序运行。

CFI的原理

- 静态分析：
  - 首先，通过静态程序分析预先得出程序的**控制流图**（control flow graph，CFG）。
  - CFG包含了程序中每一个间接跳转指令的所有可能的目标。
- Instrumentation（插桩）：
  - 通过重写可执行的二进制代码对程序进行插桩来实现**运行时检查**。
  - 运行时检查要确保程序的运行要始终符合静态分析出的CFG。

- CFI的实际作用：限制程序的控制流向。



CFI的实现

静态分析

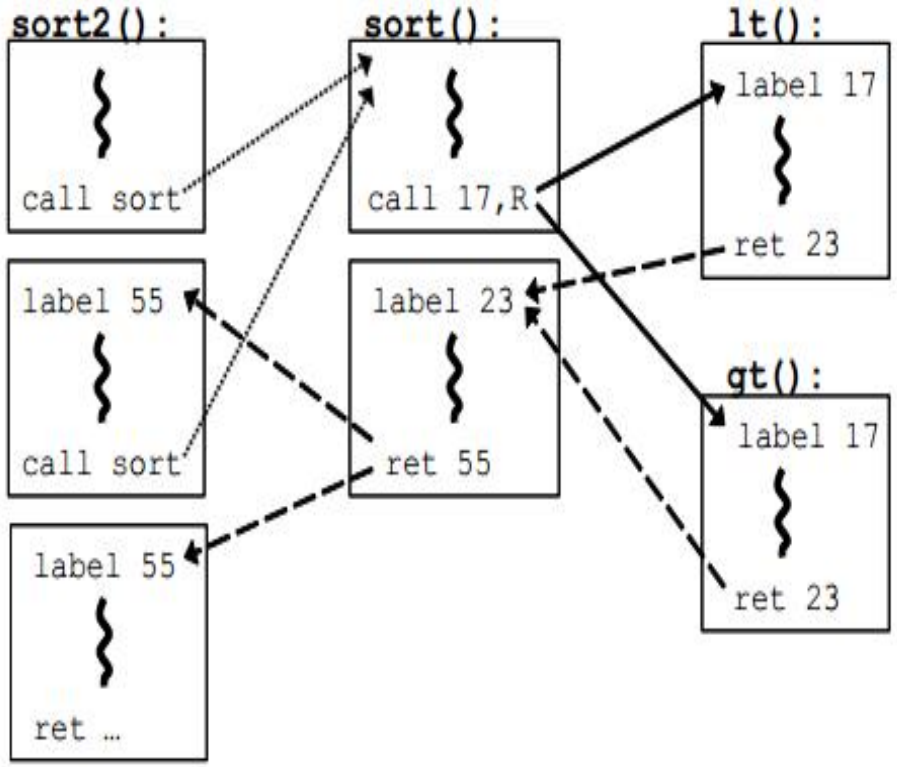
- 静态分析主要针对间接跳转，不考虑直接跳转。
- CFI为每一个间接转移指令（包括间接跳转、间接调用、和函数返回指令）的源和目标加入一个**标记**，只有标记匹配才可以跳转。
  - 标记是一个常数，嵌入在程序的二进制机器码中。
  - 标记不是秘密的，但必须是唯一的。
- 如果CFG中同一个源可以跳转到两个目标，则这两个目标是等价的。
  - **等价的目标标记是相同的。**
  - 也就是说，一个标记代表一个CFG的等价类。

CFI的实现

```
bool lt(int x, int y) {
    return x < y;
}

bool gt(int x, int y) {
    return x > y;
}

sort2(int a[], int b[], int len)
{
    sort( a, len, lt );
    sort( b, len, gt );
}
```



插桩

- 通过重写可执行的二进制代码对程序进行插桩来实现运行时检查。
- 运行时检查要确保程序的运行要始终符合静态分析出的CFG。
  - 每当有指令转移控制流时，转移的目标必须是由CFG所允许的目标。
  - 对二进制代码进行插桩，运行时检查跳转指令的标记与跳转目标的标记**是否匹配**。
- 目标：防止任意代码注入以及**非法的控制流转移**
  - 攻击者即使完全控制了进程的地址空间也可以保证程序运行的安全。

CFI的实现

原始代码

Source			Destination		
Opcode bytes	Instructions		Opcode bytes	Instructions	
FF E1	jmp ecx	; computed jump	8B 44 24 04	mov eax, [esp+4]	; dst

插桩后代码

```
B8 77 56 34 12    mov  eax, 12345677h   ; load ID-1
40                inc  eax              ; add 1 for ID
39 41 04          cmp  [ecx+4], eax     ; compare w/dst
75 13             jne  error_label      ; if != fail
FF E1             jmp  ecx              ; jump to label
```

```
3E 0F 18 05    prefetchnta              ; label
78 56 34 12      [12345678h]            ;    ID
8B 44 24 04    mov  eax, [esp+4]        ; dst
...
```

Jump to the destination only if the tag is equal to “12345678”

Abuse an x86 assembly instruction to insert “12345678” tag into the binary

CFI的分析

- CFI和ASLR是目前针对代码复用攻击最主要的两种防御方法，是学术界研究的热点。
- 但是，CFI仅仅是一个学术上的研究，还没有在实际系统中被应用。
  - CFI实现过于复杂。
    - 需要静态分析，获得控制流图CFG。
    - 需要二进制插桩，修改二进制文件。
  - CFI性能损耗很大，需要实时监控和分析每一个间接跳转是否合法。

CFI的分析

- 优点：
  - 抓住了代码复用攻击的一个本质特征，防御效果很好。
  - 可以有效防止基于非法控制流转移的攻击，包括代码注入攻击和代码复用攻击。
- 缺点：
  - 实现复杂，性能损耗过高，因此实用性不高，没有被真实系统采用
  - 难以生成一个绝对精确的CFG
  - 对于不违反程序CFG的攻击无能为力

CFI小结

- 本节介绍的CFI是异常行为分析最基本的防御方法。
- 目前，CFI也是安全领域研究的热点之一，启发了许多基于控制流的防御方法。
- 围绕破解或优化基于控制流的防御方法的研究有很多，下一讲会进一步介绍。

内容概要

- 代码复用攻击
  - ret2libc
  - ROP
  - JOP
- 对代码复用攻击的防御
  - ASLR
  - CFI
- 总结

总结

- 介绍了代码复用攻击的基本原理和过程，并且介绍了几种经典的代码复用攻击方法。
  - ret2libc，以库函数作为配件。
  - ROP，以ret为配件链接。
  - JOP，以jump为配件链接。
- 针对代码复用攻击的特征，介绍了经典的代码复用攻击防御方法
  - ASLR，随机化防御的代表性方法，已被实际系统采用。
  - CFI，异常行为检测的代表性方法。

总结

- 攻击和防御总是针锋相对，互相促进。
- 对于ASLR和CFI等防御方法，研究者提出了新的代码复用攻击方法。
  - 针对随机化防御的JIT+代码复用攻击
  - 针对粗粒度CFI的代码复用攻击
  - COOP和FOP
- 同样，研究者也提出更加强大的防御方法，和更加简单实用的防御方法。
  - 粗粒度CFI
  - CPI

经典论文推荐

- 1、经典代码复用攻击：
  - ROP：The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)，CCS 2007
  - JOP：Jump-Oriented Programming: A New Class of Code-Reuse Attack，ASIACCS 2011
- 2、对代码复用攻击的防御：
  - ASLR：On the effectiveness of address-space randomization，CCS 2004
  - CFI：Control-Flow Integrity，CCS 2005

作业

- 实验：**代码注入攻击和ROP攻击的实现**
  - 在上次实验（栈溢出漏洞）的基础上，自行构造代码注入攻击和ROP攻击
  - 攻击目标（二选一）：1）在屏幕上打印输出“attack success！”字符；2）打开一个shell
  - ROP攻击**应至少包含3个配件**，可在漏洞程序中**手动添加**ROP攻击所需配件
- 报告：
  - 将代码注入攻击和ROP攻击的构建及攻击整个过程进行描述和分析，形成**实验报告**
  - 附上最终的漏洞代码及攻击代码，并在报告中对代码进行注释和说明（以上代码应可编译可运行**可复现**）

作业

- 扩展实验（不强制）：
  - 感兴趣的同学可以尝试更多的漏洞和更多的攻击，任意进行组合
    - 堆漏洞+代码注入攻击/ROP攻击
    - 栈漏洞+JOP攻击/FOP攻击/DOP攻击
    - 尝试绕过已有的安全机制（如加入信息泄露漏洞破解ASLR或栈cookie保护，破解不可执行位保护等）
    - 。。。。。。
  - 可以将相关实验过程补充到实验报告中，根据实验效果和工作量酌情加分


Q&A