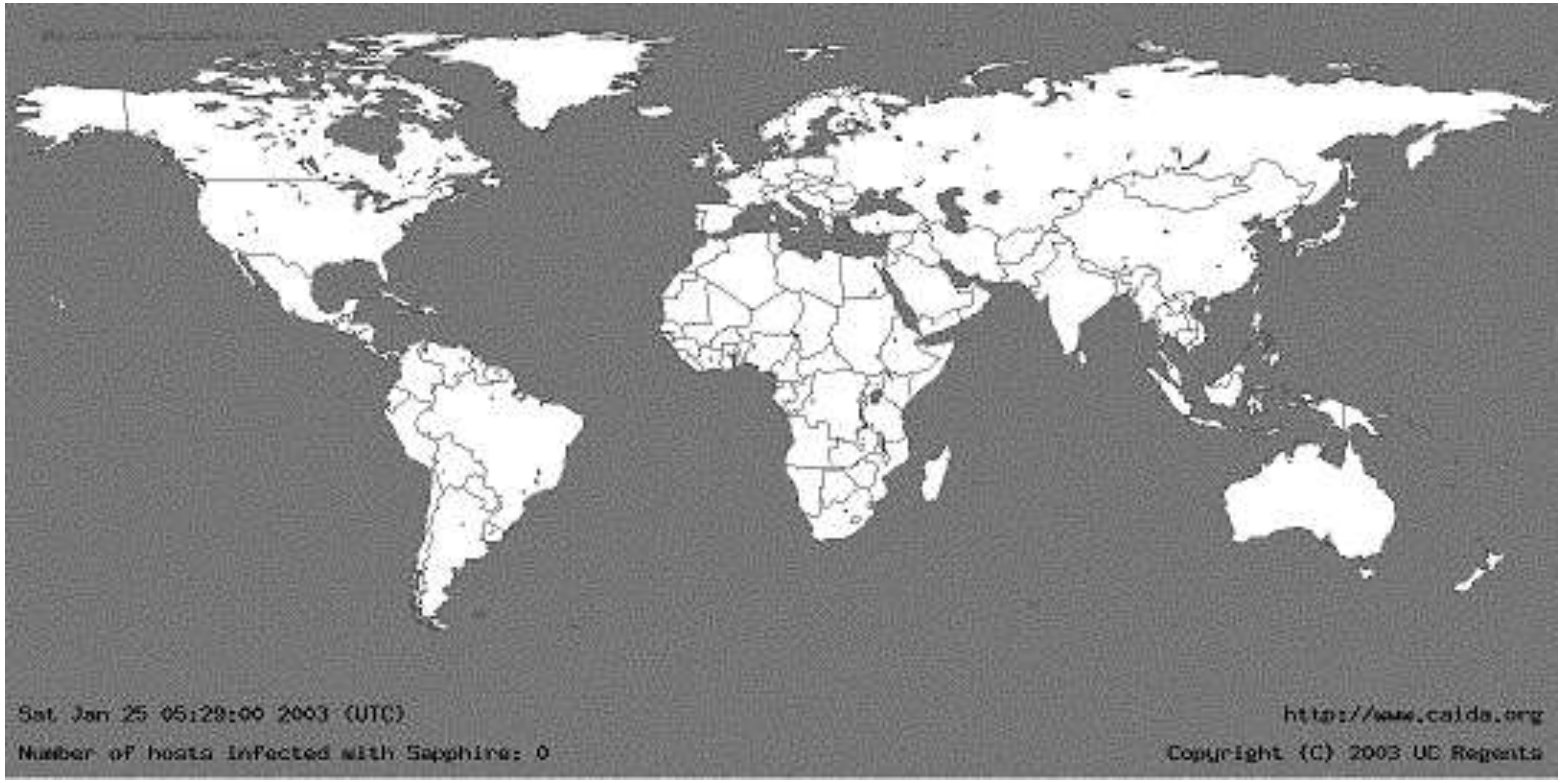
操作系统安全

李琦
清华大学网研院

Slammer蠕虫

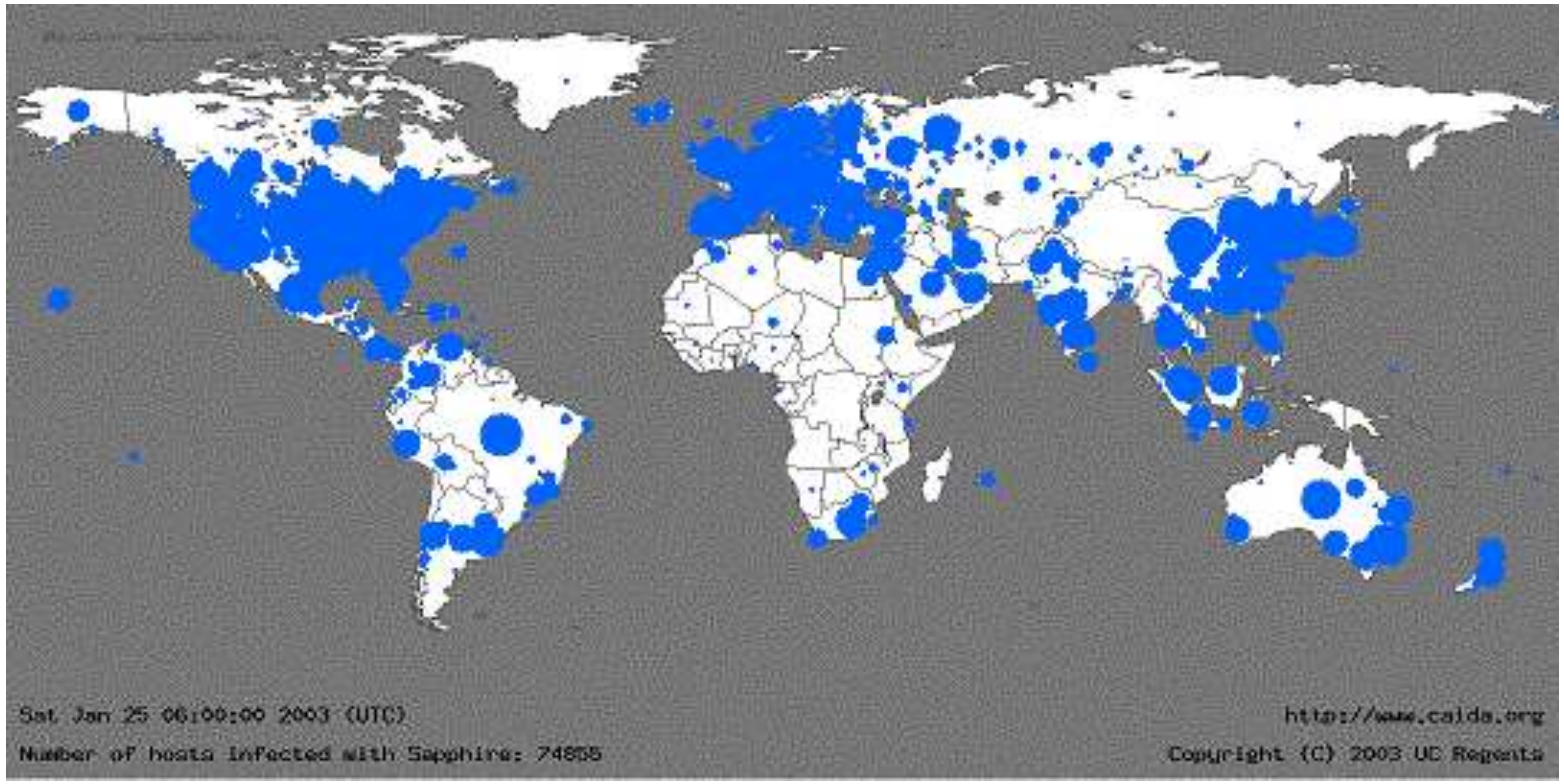
Slammer (2003年)是一款DDOS恶意程序, 采取分布式阻断服务攻击感染服务器, 它利用SQL Server弱点采取阻断服务攻击1434端口并在内存中感染SQL Server, 通过被感染的SQL Server 再大量的散播阻断服务攻击与感染, 造成SQL Server 无法正常作业或宕机, 并致使网络拥塞



05:29:00 UTC, January 25, 2003 (攻击发生前)

Slammer蠕虫

Slammer (2003年)是一款DDOS恶意程序, 采取分布式阻断服务攻击感染服务器, 它利用SQL Server弱点采取阻断服务攻击1434端口并在内存中感染SQL Server, 通过被感染的SQL Server 再大量的散播阻断服务攻击与感染, 造成SQL Server 无法正常作业或宕机, 并致使网络拥塞



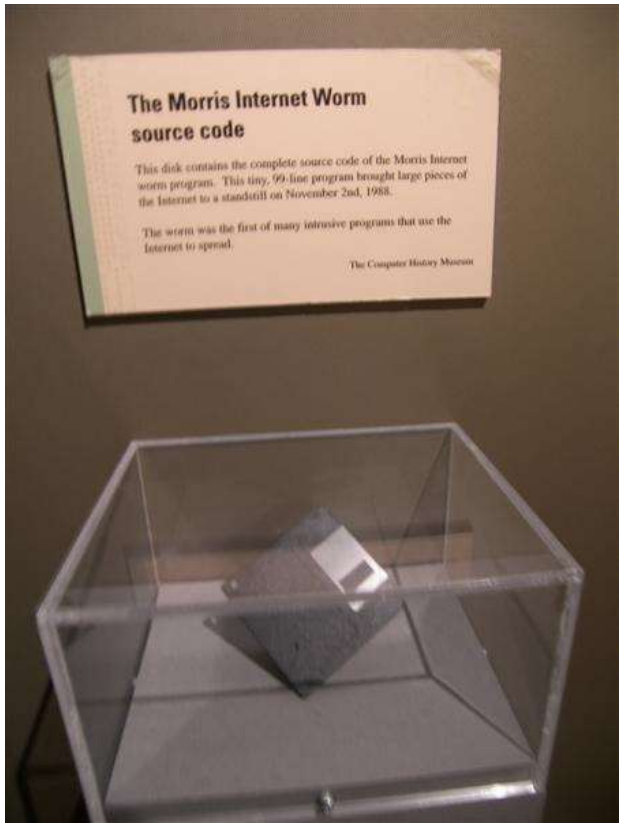
06:00:00 UTC, January 25, 2003 (攻击发生后)

操作系统安全问题的开端

最早的计算机病毒诞生于 1988年11月2日

- 由就读于康奈尔大学的研究生Robert Tappan Morris从麻省理工学院的计算机系统上散播，这便是著名的莫里斯蠕虫病毒（Morris Worm）

- 这一病毒感染了6,000余台UNIX主机（占当时主机总数的约10%）



其原理是操作系统的栈溢出漏洞（Stack Overflow）

注. 右图为保存有Morris Worm源代码的软盘，其仅有99行源代码
(图片来自Wikipedia)

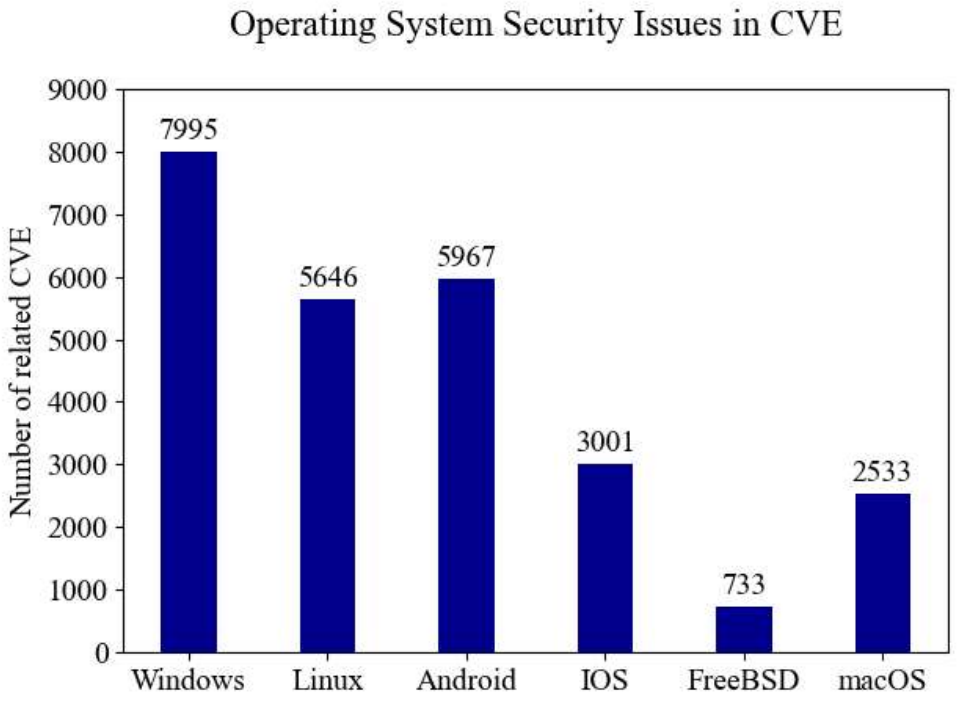
讨论

畅所欲言：

你认为开源操作系统更安全，还是闭源操作系统更安全?

操作系统安全问题的开端

- 计算机病毒诞生至今的三十余年间，有数以万记的操作系统漏洞被发现或利用
- 操作系统安全一直受到广泛关注，在CVE当中有记录操作系统相关漏洞就超过**20,000**条，占CVE当中记录的漏洞总数超过**1/6**
- **操作系统安全的相关前沿研究众多，**至今仍能在S&P、Security、NDSS、CCS等安全顶级会议上占据一席之地



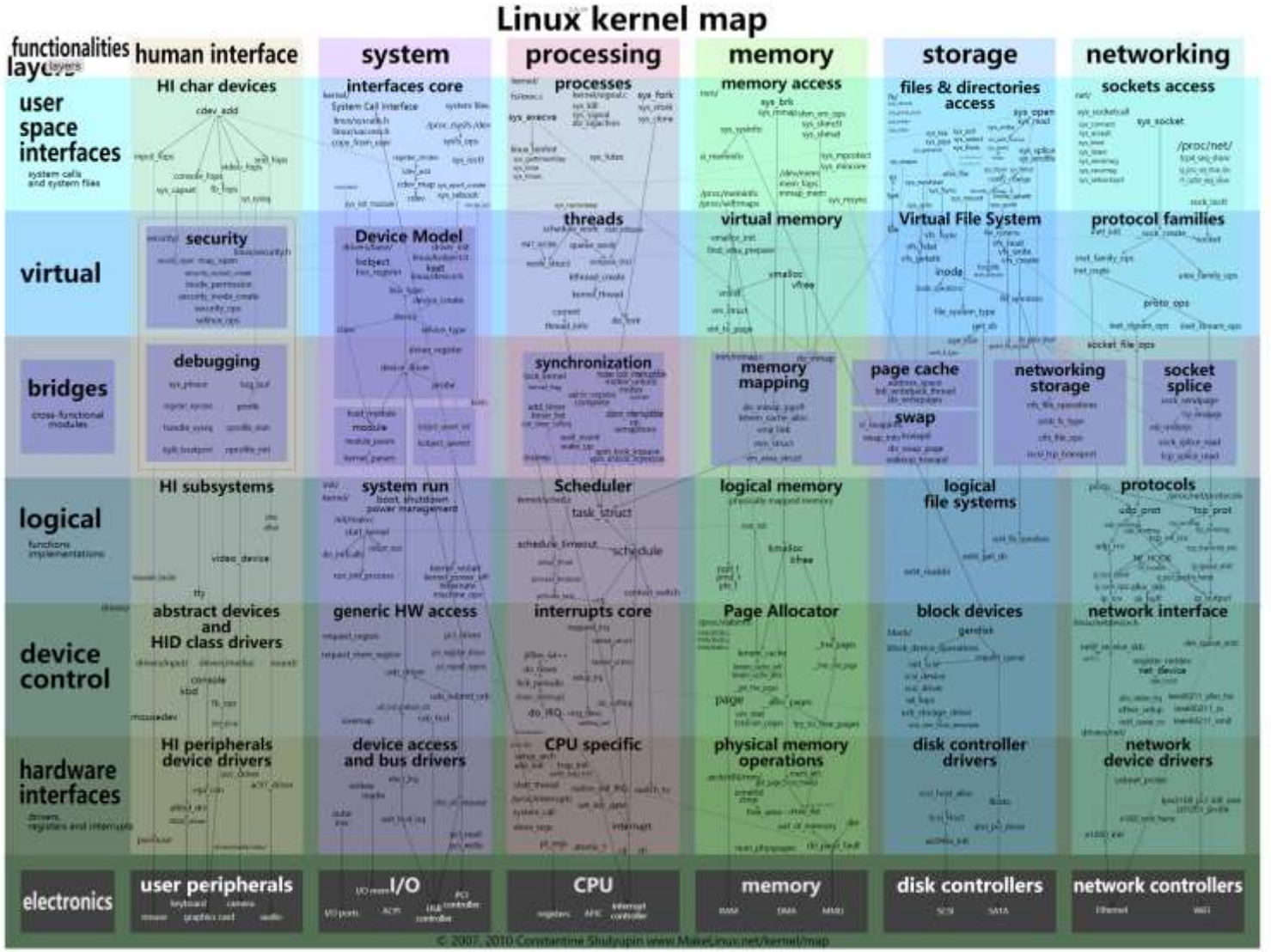
2020年的CVE中各类操作系统漏洞的数量

为何操作系统安全漏洞广泛存在

- 其一，现代操作系统是规模庞大的软件系统

现代操作系统是**系统之系统**，各个模块之间的依赖关系复杂

目前的Linux内核包含了2,780万行源代码分散在6万余个文件



交互式内核函数依赖关系图来源于https://makelinux.github.io/kernel/map/

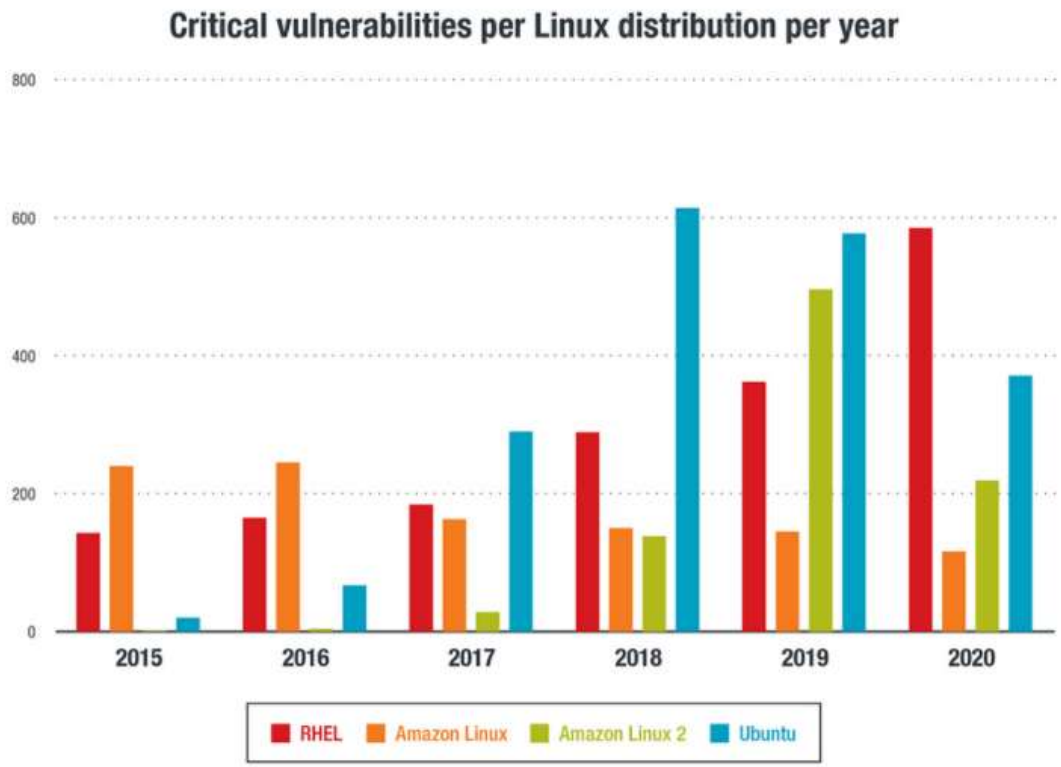
为何操作系统安全漏洞广泛存在

- 其二，现代操作系统的设计以性能为最主要目标，而非安全性

Linux已经在“小修小补”当中度过了**30**年

例如，Linux内核当中的Martian Address机制被实现以对抗来自恶意地址的数据包

而这一机制并非在操作系统的设计之初就纳入实现计划内

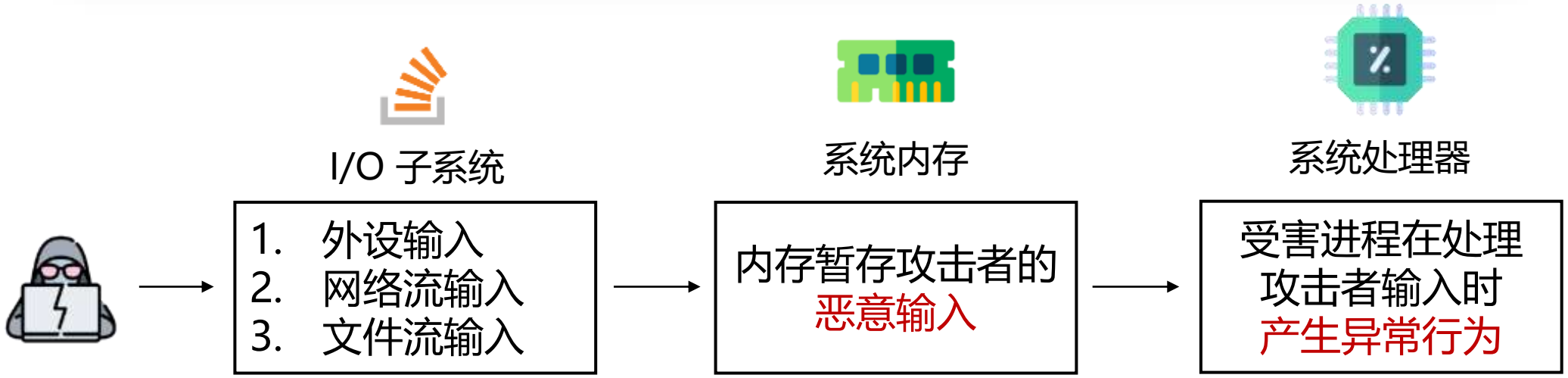


Linux不同发行版的严重漏洞数量统计

注. Martian Address, 即来自火星的地址
该机制过滤掉显然为伪造源地址的数据包，并称其来自火星的数据包

面向系统攻击的前提条件

本章中假设攻击者**位于操作系统外部**；
仅能通过**正常I/O方式**与操作系统下的受害进程进行交互



- 攻击者可以构造任意输入并接受受害进程输入校验；
- 攻击者无法**直接**读写系统下进程的内存，无法**直接**干预处理器上指令的执行

面向系统攻击的目标

攻击者通过构造恶意输入，使操作系统环境下运行的**进程产生异常行为**：

- 例如：

1. 进程直接崩溃：HTTP服务器拒绝服务
2. 恶意函数的调用：SQL Slammers 调用 UDP 发包函数实现蠕虫复制
3. 权限提升： Privilege Escalation，攻击者获得执行任意命令的权限，或者获得管理员账户的执行权限

使操作系统下受害进程产生攻击者期望的异常行为
的攻击效果被称为进程的**控制流劫持**

面向系统攻击的典型案例

- SQL slammers是针对Microsoft SQL Server 2000数据库服务器的蠕虫病毒

该病毒构造恶意携带**恶意负载**的UDP数据包，以**栈区溢**出控制SQL Server守护进程，使其调用操作系统的UDP发包接口，重复上述步骤，实现病毒复制

I/O 子系统

扫描UDP 1434端口定位SQL Server服务，发送携带恶意负载的UDP包

内存管理系统

使sqlsort.dll发生栈溢出：覆盖栈中保存的返回地址，跳转执行恶意代码，劫持进程的控制流

进程管理系统

劫持进程控制流后，借助kernel32.dll等动态链接库，恶意调用UDP发包函数，实现蠕虫复制

利用操作系统漏洞是一个分多个步进行的过程，每一步均与操作系统某一模块相关

本章的内容组织

第一节
基础攻击方案

- 内存管理基础
- 内存栈区攻击方案
- 内存堆区攻击方案

第二节
基础防御方案

- W^X (NX, DEP)
- ASLR
- Stack Canary
- SMAP, SMEP

第三节
高级控制流劫持技术

- 进程执行细节
- 面向返回地址编程
- 全局偏置表劫持
- 虚假vtable劫持

第四节
高级系统保护方案

- 控制流完整性
- 指针完整性
- 信息流控制
- I/O子系统保护

内存为对象的攻击
以内存为对象的简单攻击方案

内存粒度的防御方案
防御基础内存攻击的方案

恶意修改进程行为
基于第一章的攻击绕过第二章基础防御

系统粒度的防御方案
通过更严格的安全机制防御控制流劫持

有效防御

成功绕过

有效防御

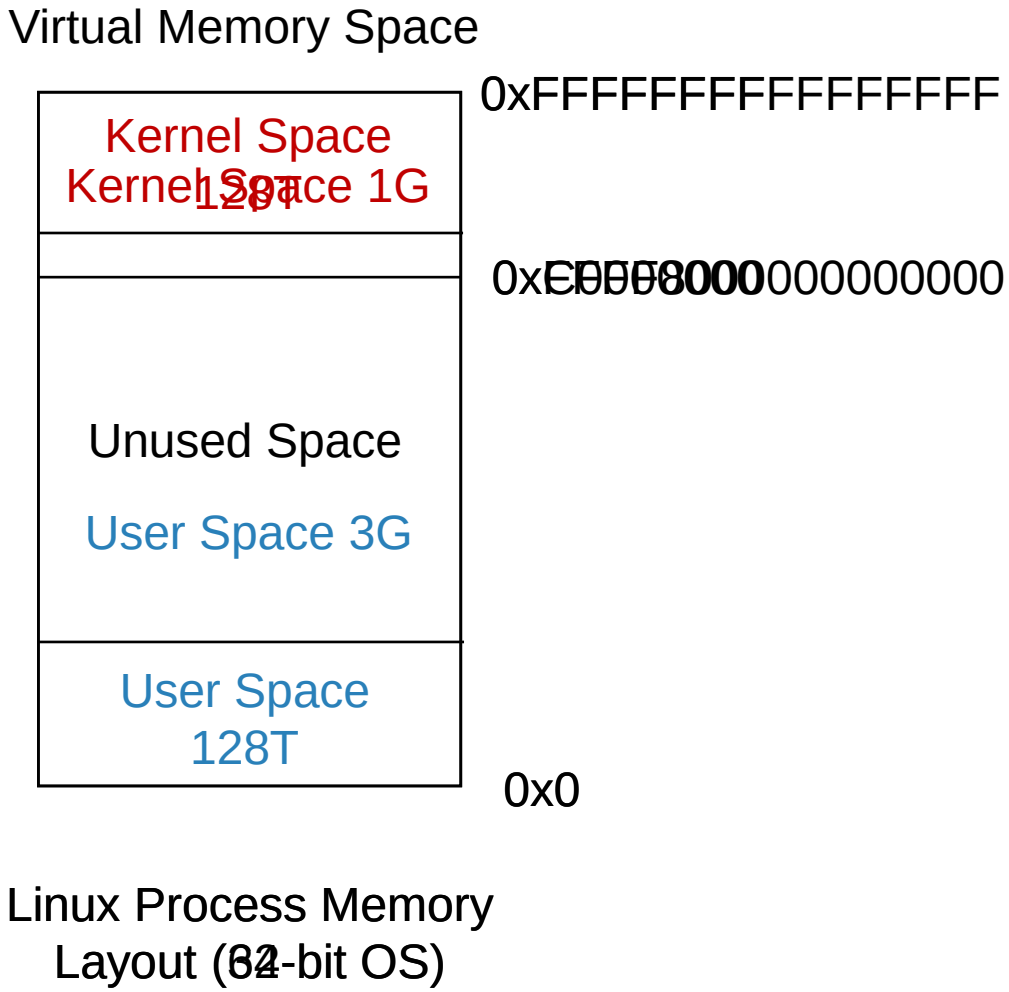
第1节 操作系统基础攻击方案

- **1.1 操作系统内存管理基础**
- 1.2 基础的栈区攻击方案
- 1.3 基础的堆区攻击方案

进程角度的内存管理

- Linux 内核为每一个进程维护一个**独立的**线性逻辑地址空间，以便于实现进程间内存的相互隔离

- 这一线性逻辑地址空间被分为**用户空间**和**内核空间**；用户态下仅可访问用户空间，系统调用提供接口以访问内核空间；内核态下亦无法访问用户空间



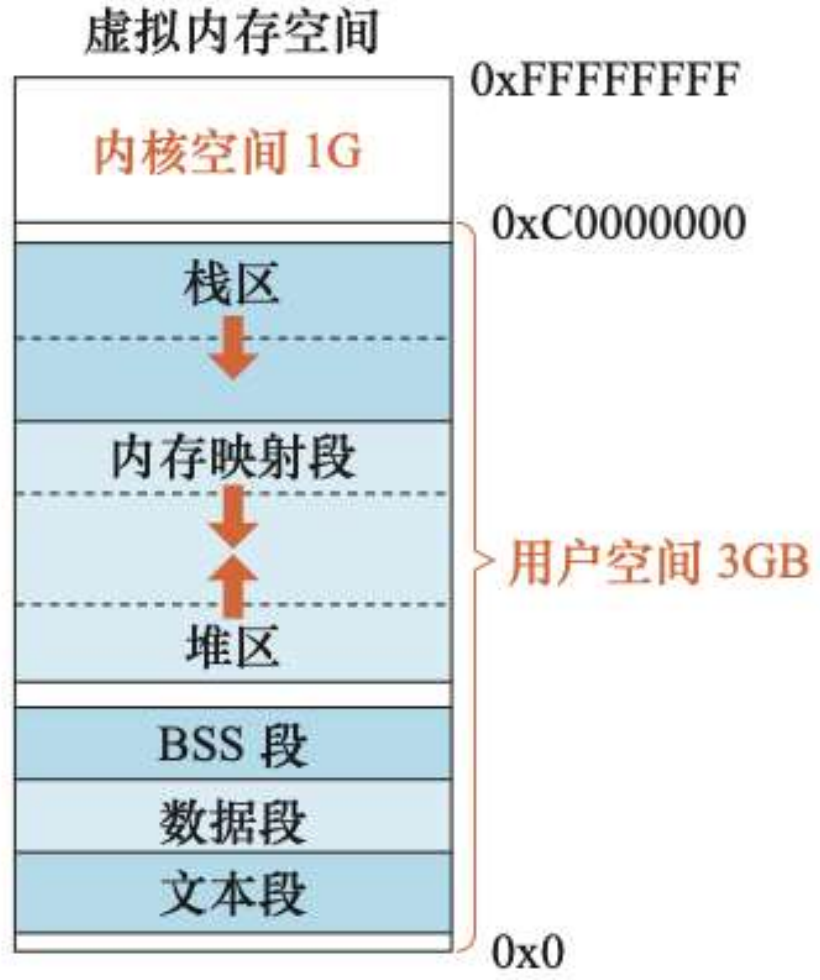
Linux Process Memory Layout (64-bit OS)

注. 右图为Linux当中内核区与用户区在进程虚拟地址空间下的分布

进程角度的内存管理

用户区内存空间包含了6个重要区域：

1. 文本段：进程的可执行二进制源代码
2. 数据段：初始化了的静态变量和全局变量
3. BSS段：未初始化的静态变量和全局变量
4. 堆区　：由程序申请释放
5. 内存映射段：映射共享内存和动态链接库
6. 栈区　：包含了函数调用信息和局部变量



Linux Process Memory Layout (32-bit OS)

注. 右图为Linux当中用户区的划分方式

内存的权限管理

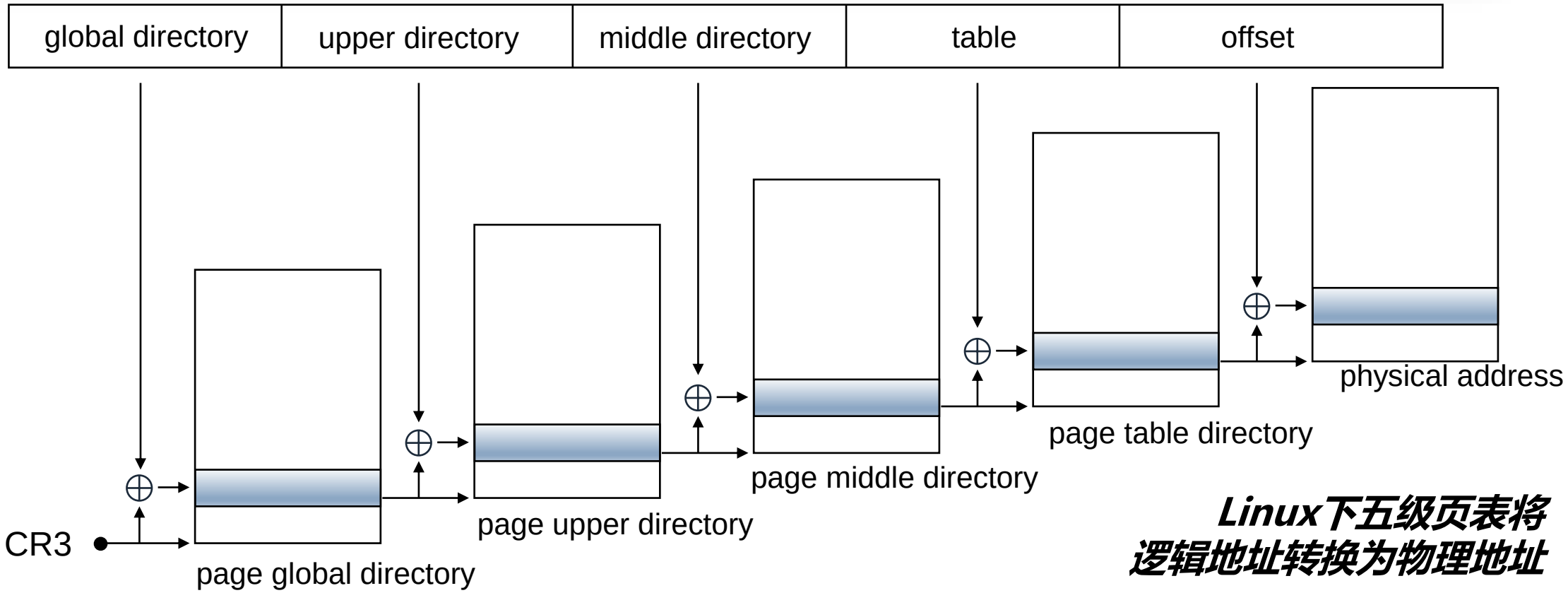
- 用户区内存区域之间的比较：

Question:
为什么文本段只读？

区域名称	存储内容	权限	增长方向	分配时间
文本段	二进制可执行机器码	只读	固定	进程初始化
数据段	初始化了的静态、全局变量	读写	固定	进程初始化
BSS段	未初始化的静态、全局变量	读写	固定	进程初始化
堆区	由进程执行的逻辑决定	读写	向高地址	堆管理器申请内核分配
内存映射段	动态链接库、共享内存的映射信息	内容相关	向低地址	运行时内核分配
栈区	函数调用信息与局部变量	读写	向低地址	函数调用时分配

虚拟地址到逻辑地址的转换

需要注意的是，上述分区均存在于**虚拟地址空间**当中，进程可见的地址均为虚拟地址，内存物理地址对进程不可见；虚拟地址需要经过**页式内存管理模块**才可转换为物理地址，本节提到地址**均为逻辑地址**



Linux下五级页表将逻辑地址转换为物理地址

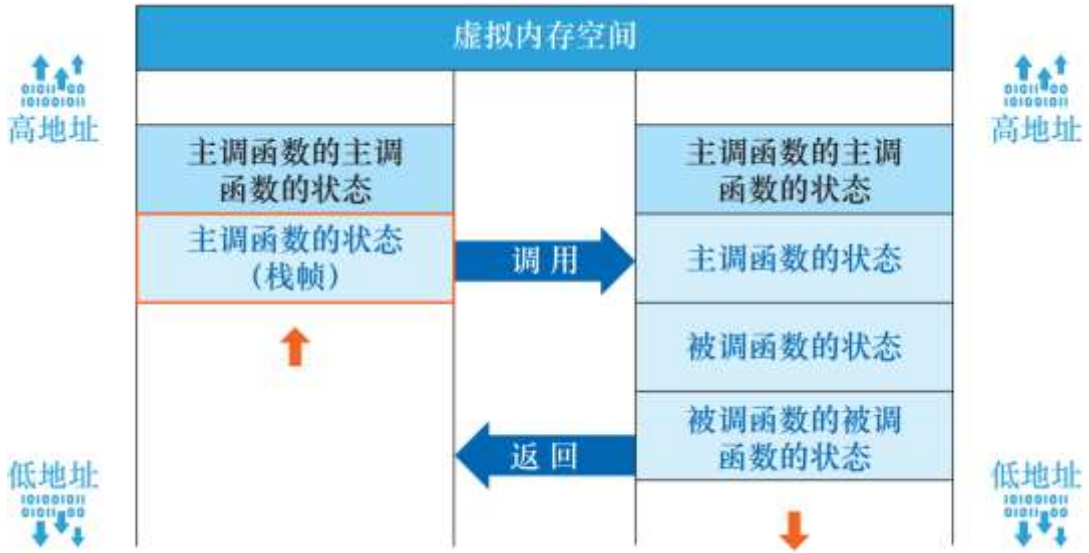
第1节 操作系统基础攻击方案

- 1.1 操作系统内存管理基础
- **1.2 基础的栈区攻击方案**
- 1.3 基础的堆区攻击方案

栈区内存的作用

- 进程的执行过程可以看作一系列函数调用的过程，**栈区内存的根本作用：**

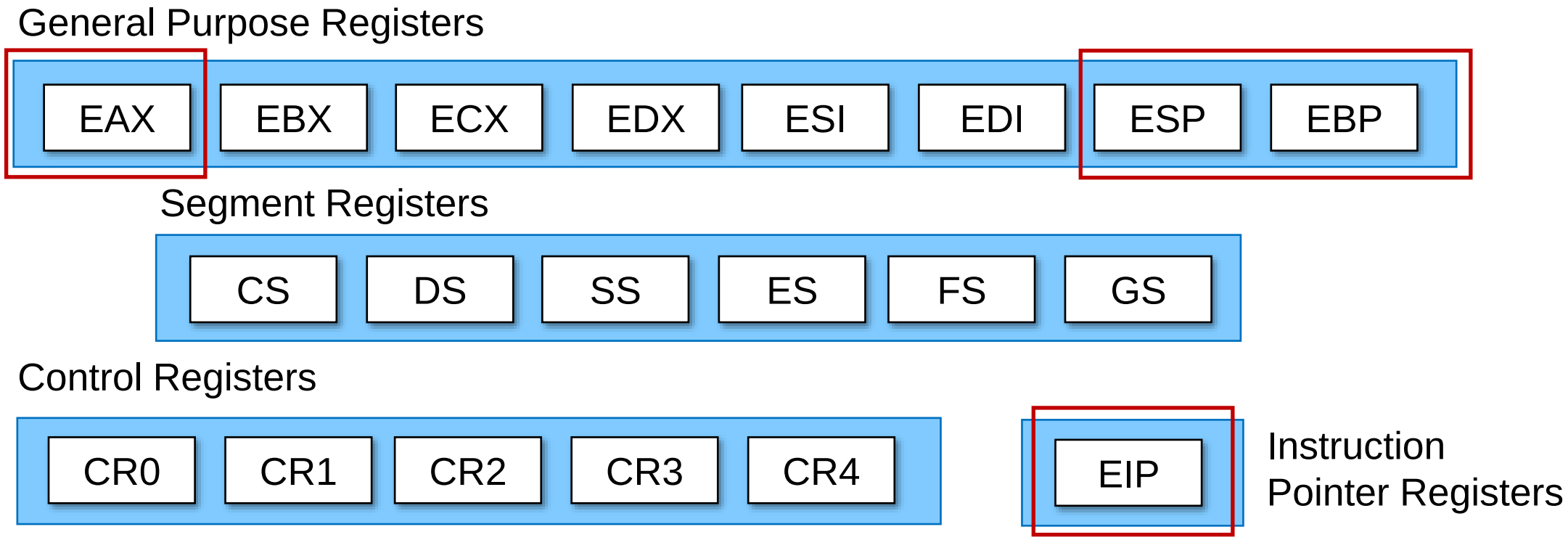
保存主调函数（Caller）的状态信息
以在调用结束后恢复主调函数状态
并创建被调函数(Callee)的状态信息



- 保存主调函数状态的连续内存区域被称作**栈帧**（Stack Frame）；当调用时栈帧进栈，当返回时栈帧出栈；栈帧是调用栈的最小逻辑单元

与函数调用密切相关的寄存器

- 为表示方便，本小节以x86_32处理器下GCC编译程序的调用过程为例，介绍保存和恢复状态的过程
- x86_32下有8个通用寄存器（位宽32），6个段寄存器（位宽16），5个控制寄存器（位宽32），1个指令寄存器（位宽32），和浮点寄存器调试寄存器等



与函数调用密切相关的寄存器

- 我们关注与函数调用相关的四个寄存器：
- 3个通用寄存器：ESP（Stack Pointer）记录栈顶的内存地址
  EBP（Base Pointer）记录当前函数栈帧基地址
  EAX（Accumulator X）用于返回值的暂存

- 1个控制寄存器：EIP （Instruction Pointer）记录下一条指令的内存地址

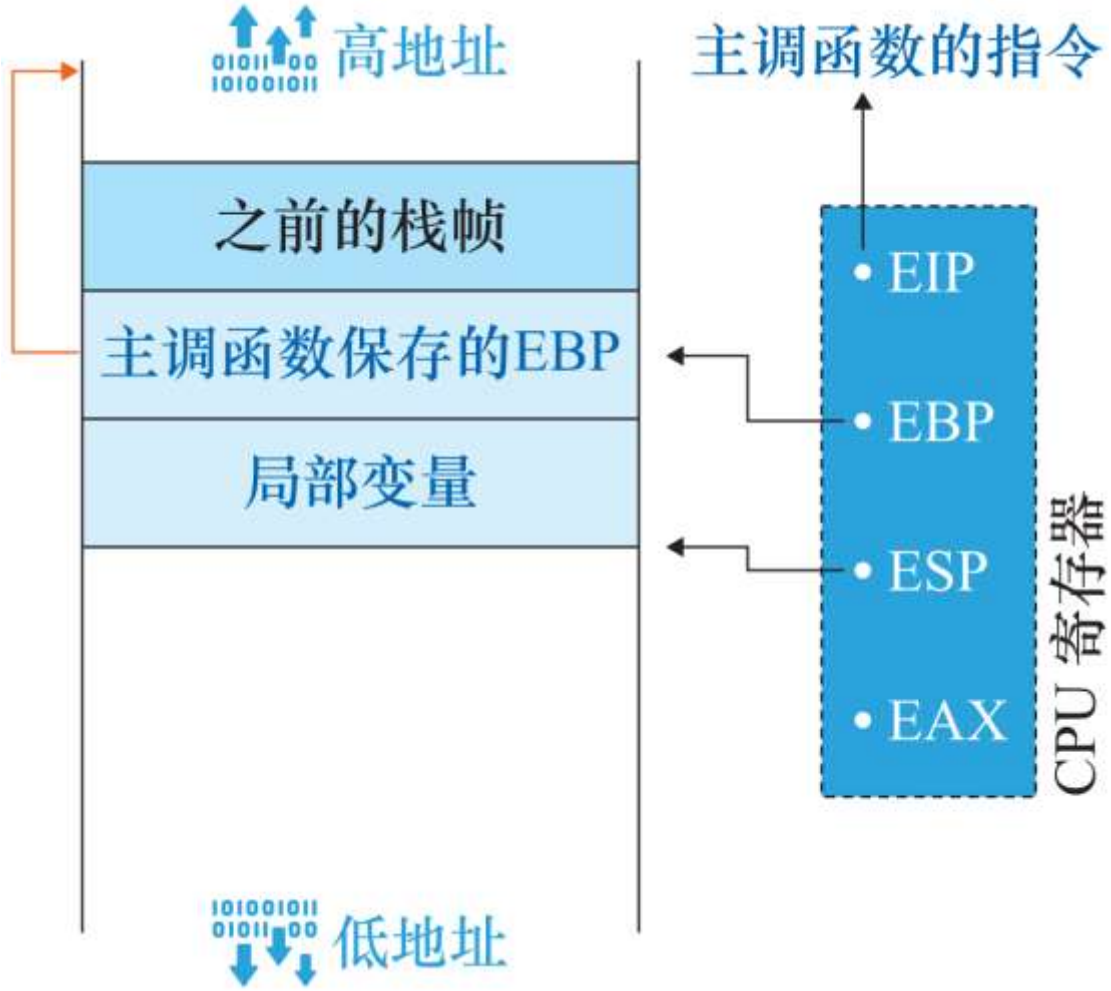
注. E表示Extend，标记32位寄存器以区别于8086当中的16位寄存器

正常的函数调用历程

常规情况下EIP指向下一条要执行指令的地址，EBP指向当前执行函数的栈帧基地址，ESP始终指向栈顶

- **函数调用前：被调函数参数压栈**

第一步为将被调函数（Callee）的参数按照逆序压入栈中；并且ESP调整位置



注. 在x64下部分参数被直接保存到寄存器当中

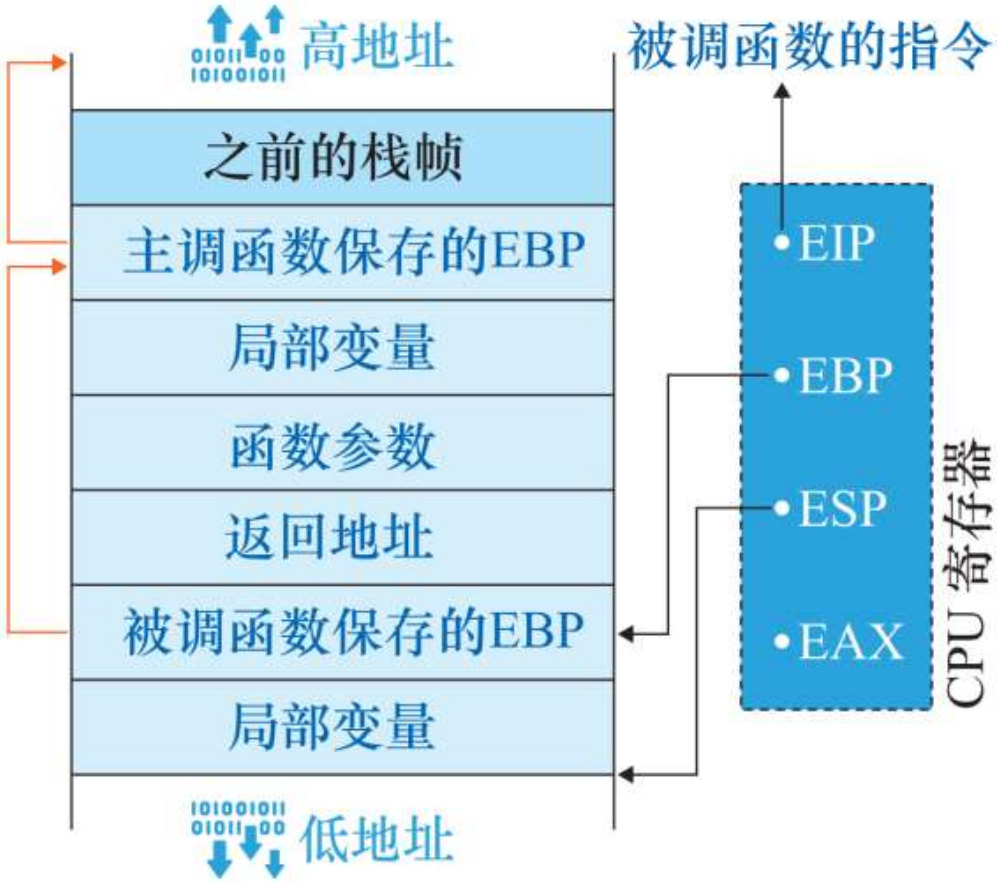
正常的函数调用历程

- **函数调用前：EIP寄存器值压栈**

EIP寄存器的数值为主调函数（Caller）下一条要执行指令的地址，指向用户地址空间当中的**代码段**；因而将EIP压入栈中作为**返回地址**

- **函数调用前：EIP寄存器更新**

而后EIP更新为调用函数指令地址，并且ESP再次调整位置指向栈顶

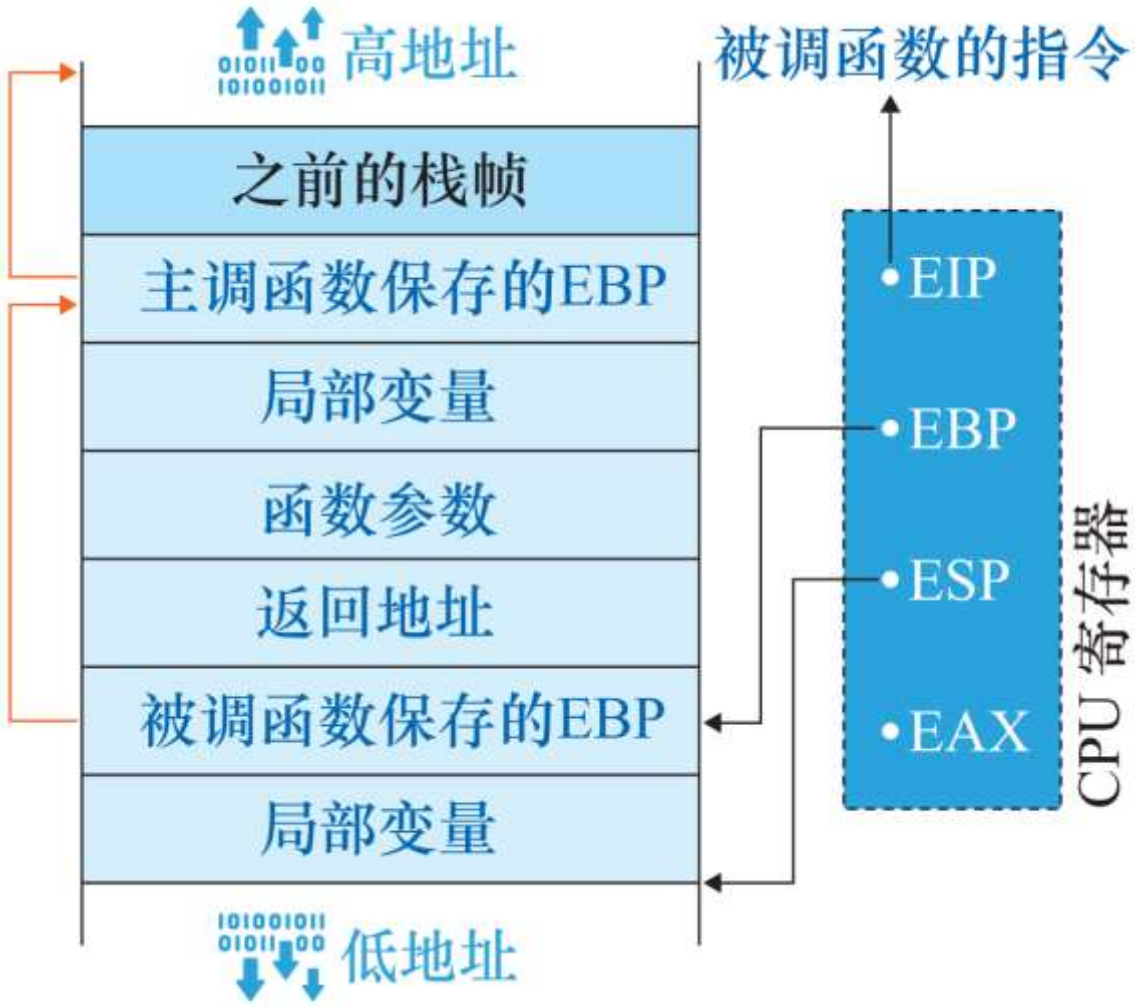


正常的函数调用历程

- **函数调用前：EBP寄存器值压栈**

EBP保存主调函数的栈帧基地址，将当前的EBP寄存器的值压入栈内；
并将 EBP寄存器的值更新为当前栈顶的地址，也就是使用当前的ESP给EBP赋值

这样主调函数（caller）的栈帧基地址信息得以保存；同时，EBP 被更新为被调用函数（callee）的栈帧基地址；指向栈顶位置的ESP也再次调整



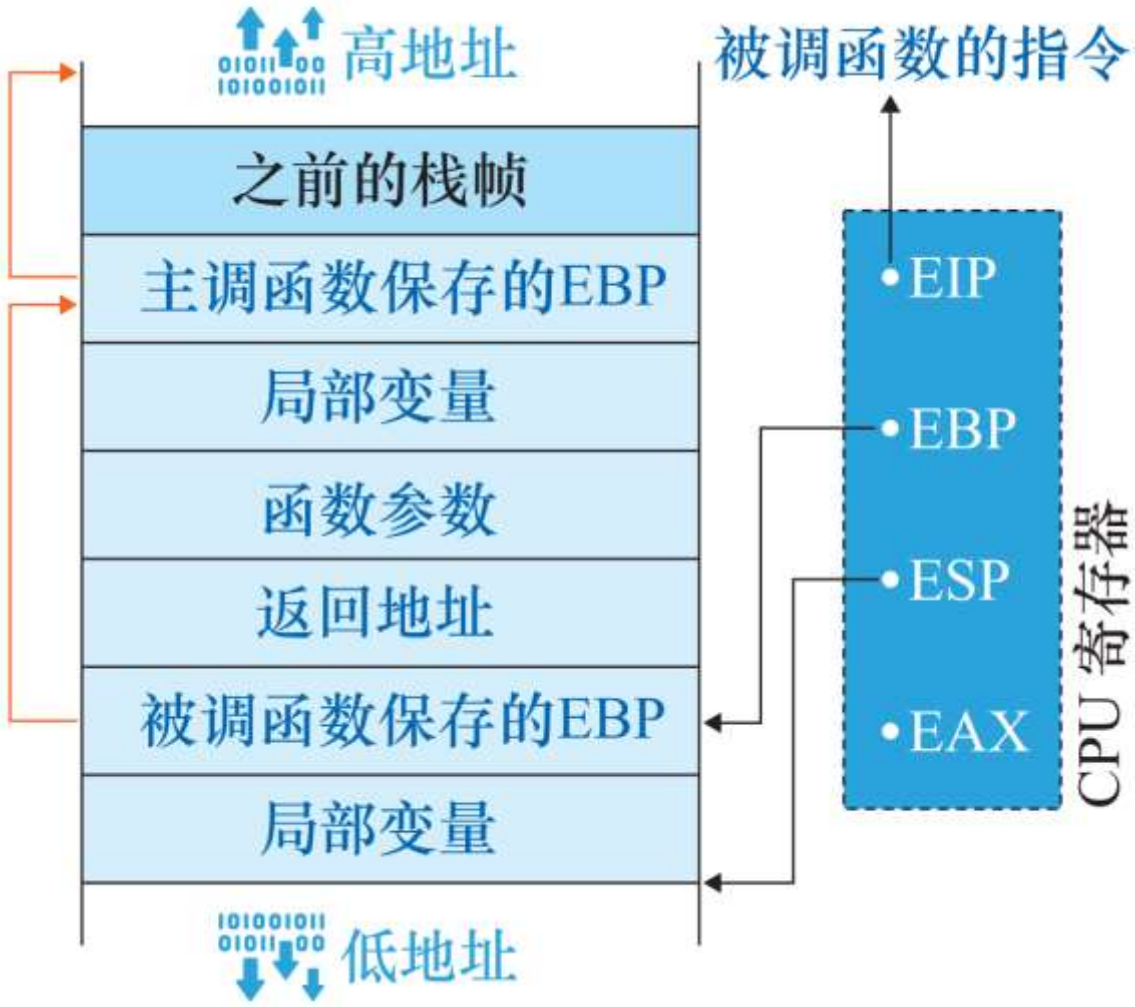
注. 红色标注Caller的栈帧，蓝色标注Callee的栈帧

正常的函数调用历程

- **函数调用前：局部变量值压栈**

调用前的最后一步是将被调函数（Callee）的局部变量压入栈中

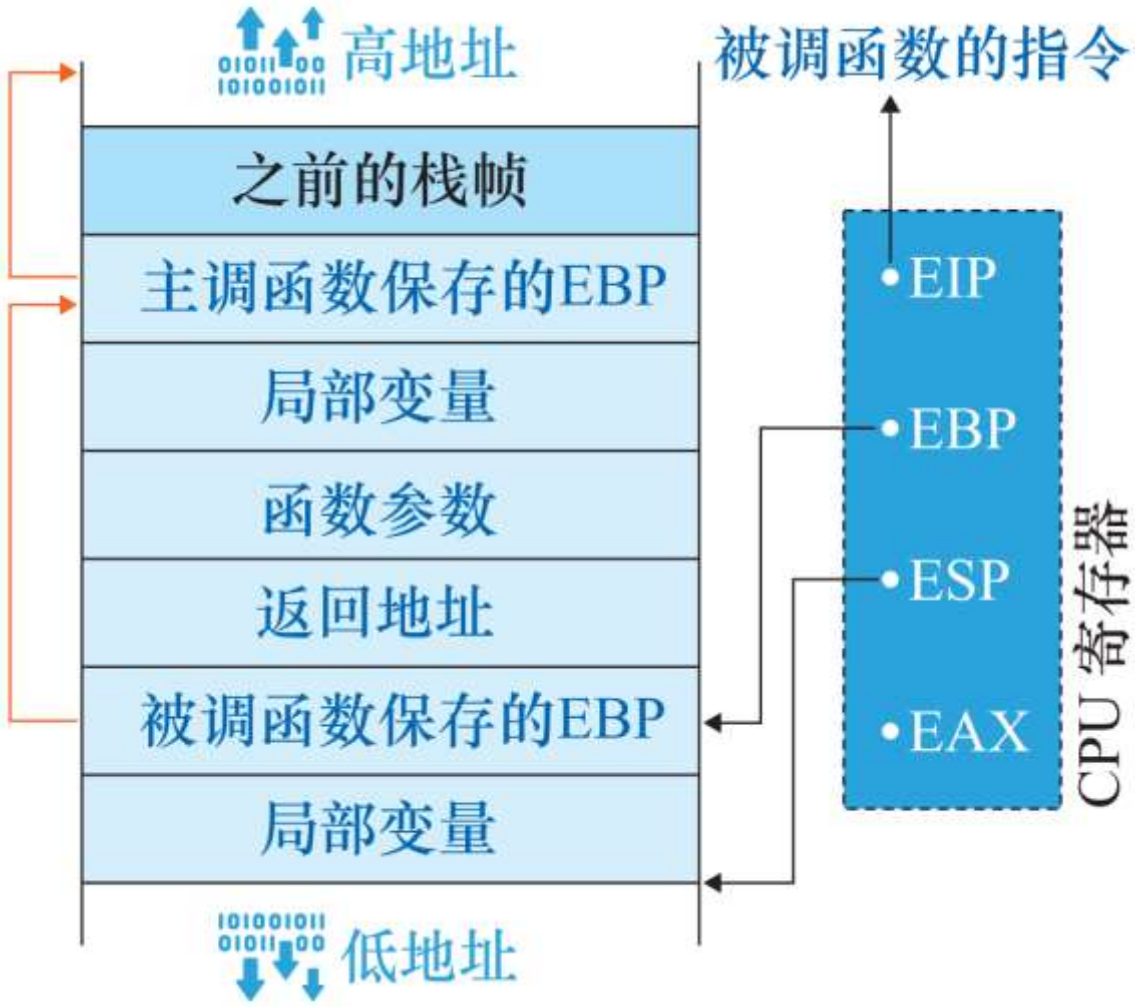
此时，EBP加偏移量（高地址方向）可获得函数的传递参数，EBP减偏移量（低地址方向）可获得函数的局部变量，EIP也已经指向被调函数的指令，被调函数开始正常执行



正常的函数调用历程

- **函数调用后：局部变量值出栈**

当返回指令被指时标志着函数调用的结束，调用后的第一步是将被调函数的局部变量弹出栈以被销毁

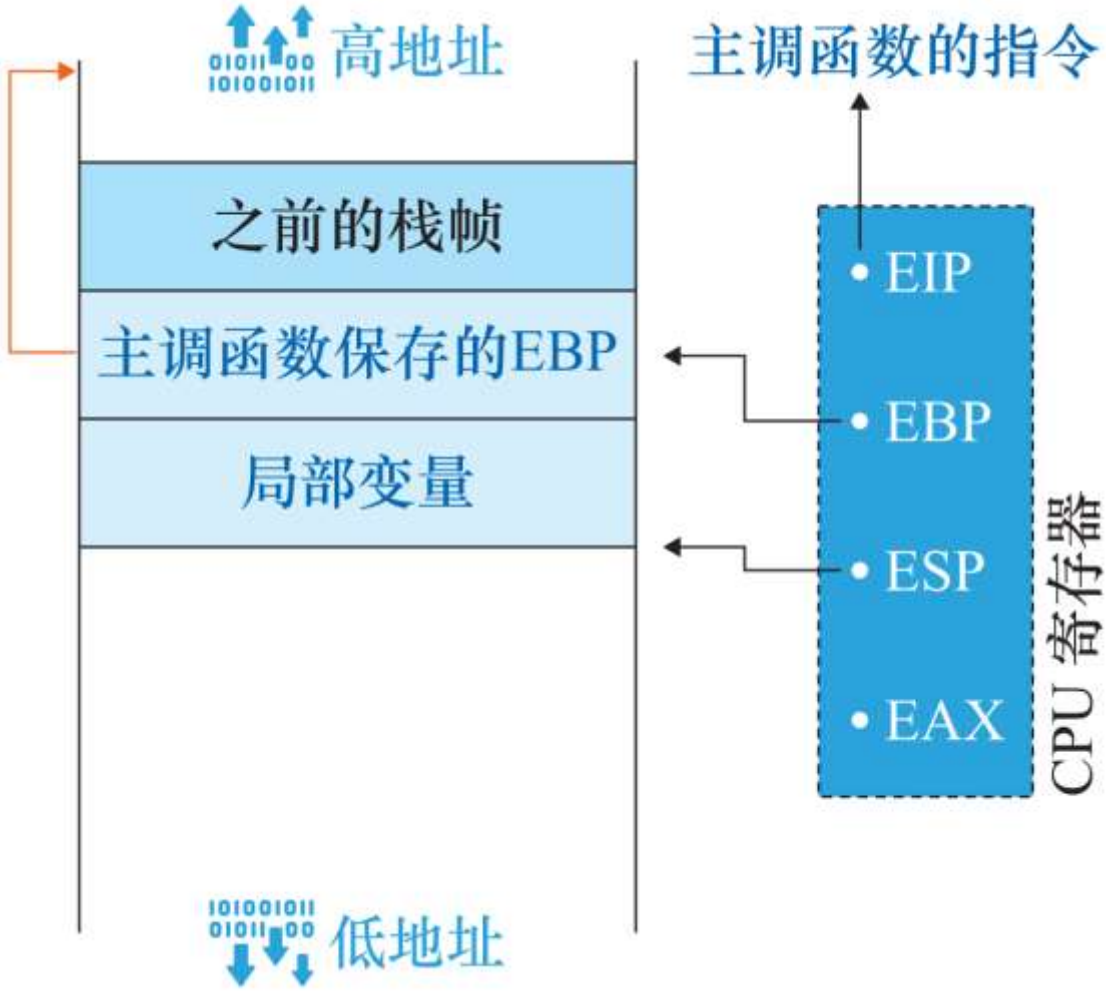


正常的函数调用历程

- **函数调用后：恢复EBP寄存器**

将栈顶的主调函数栈帧基地址赋值给EBP寄存器，之后可以重新访问主调函数的局部变量和函数参数

- **而后将主调函数的EBP弹栈销毁，并且ESP将再次调整位置**



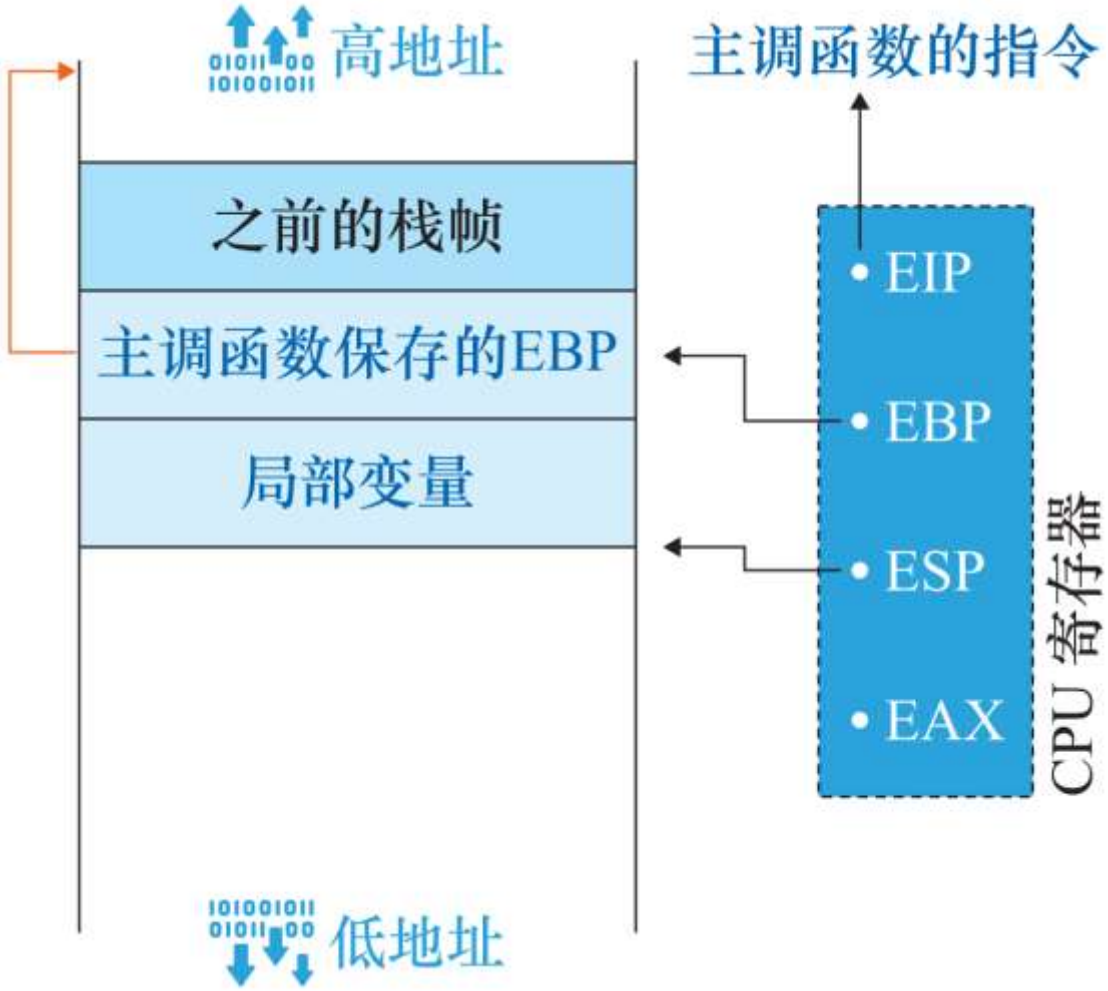
正常的函数调用历程

- **函数调用后：恢复EIP寄存器**

将栈顶的Caller的返回地址赋值给EIP寄存器，而后将从Caller调用后的下一条指令开始继续执行

- **而后，将返回地址进行弹栈**

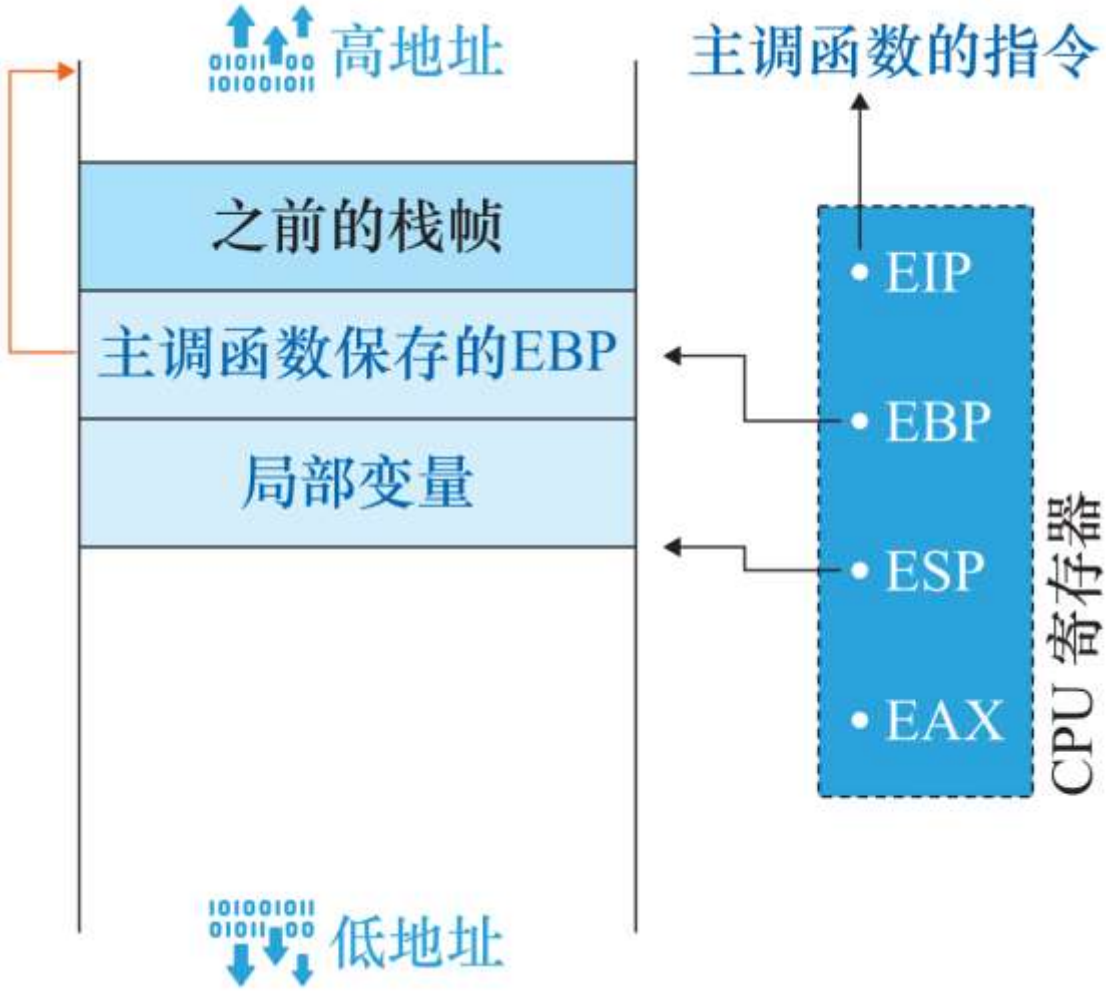
- **最后，函数调用参数也被弹栈销毁**



正常的函数调用历程

- **函数的返回值：**

函数的返回值一般通过EAX寄存器暂存，而后进行寄存器到内存的拷贝操作使用返回值



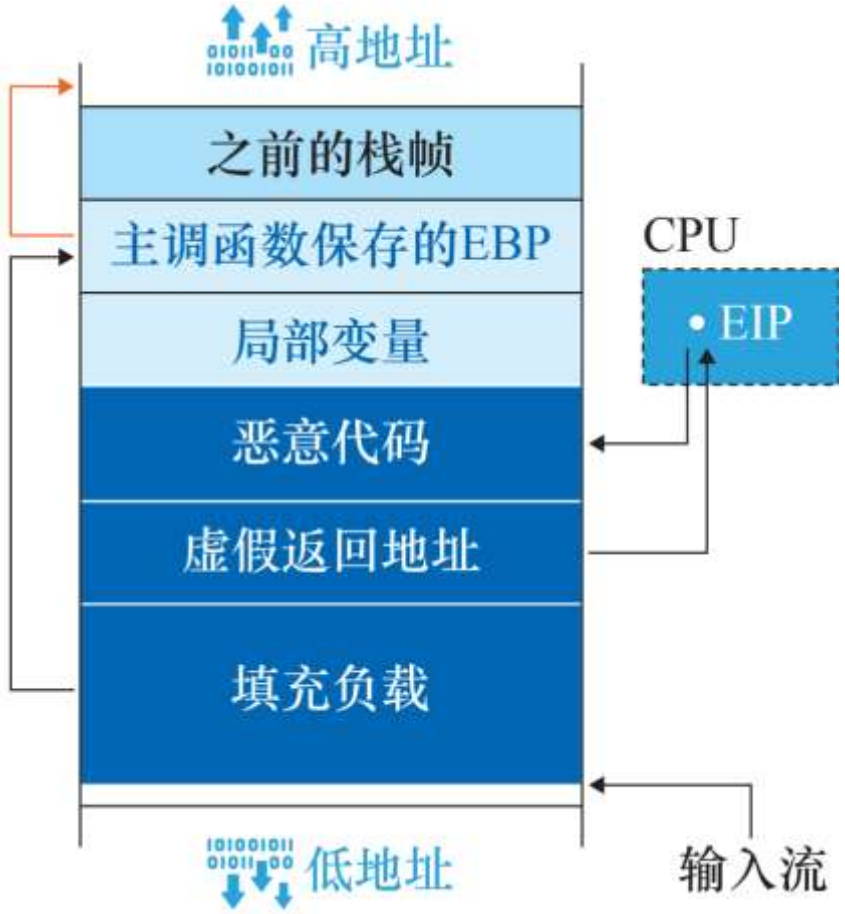
至此函数调用过程完全结束，主调函数继续执行，EBP减偏移量可获得Caller的局部变量， EBP加偏移量可获得Caller的函数参数

注. 若希望返回结构体，则EAX暂存结构体地址

栈区溢出攻击的动机

- 若攻击者希望劫持进程控制流，产生其预期的恶意行为，**则必须让EIP寄存器指向恶意指令**

- 注意到：在函数调用结束时，会将栈帧中的返回地址赋值给EIP寄存器；攻击者可以修改栈帧当中的返回地址，**使EIP指向准备好的恶意代码段**实现进程控制流劫持



栈区溢出攻击

栈区溢出攻击

是一种攻击者越界访问并修改栈帧当中的返回地址，
以控制进程的攻击方案的总称

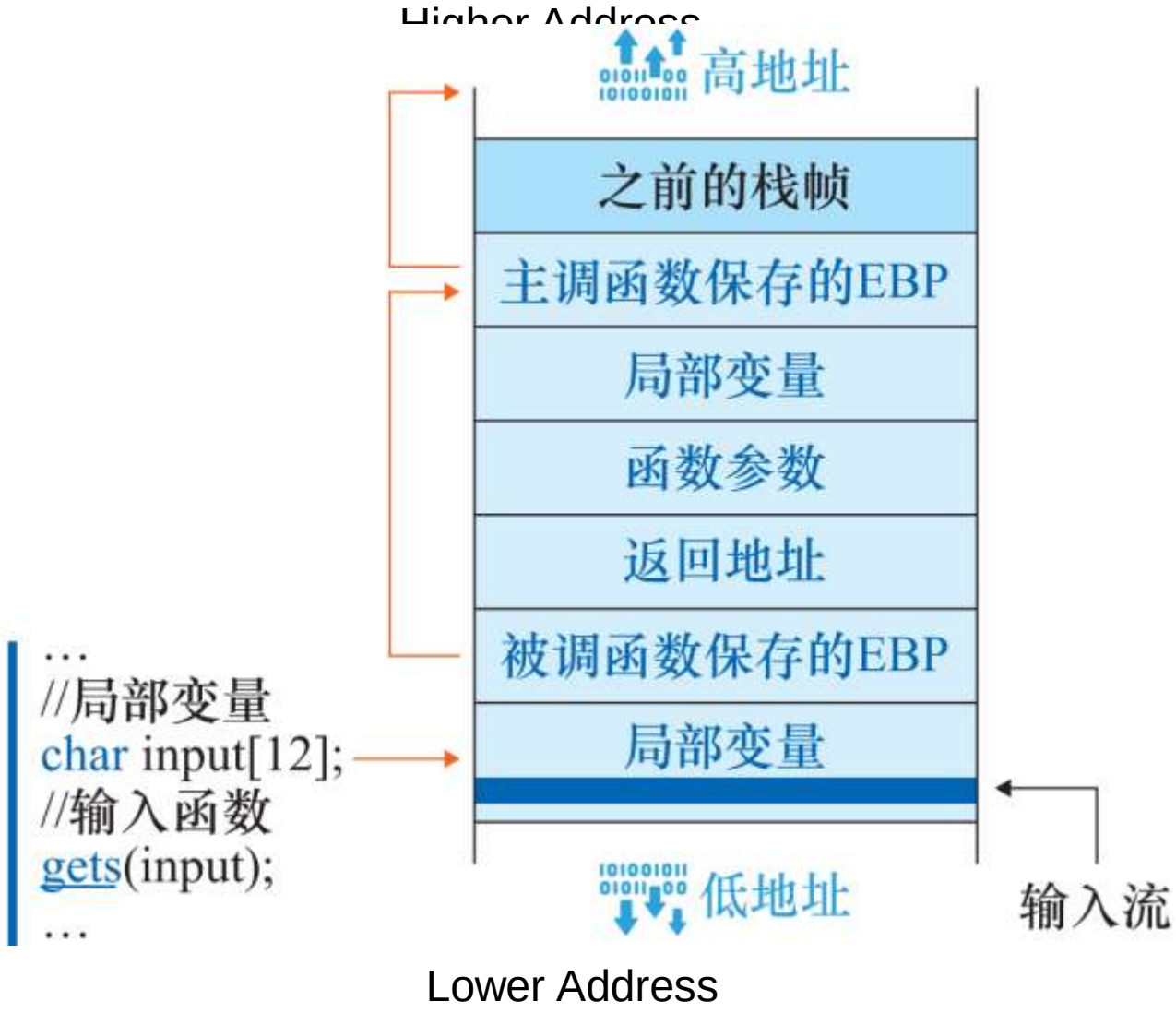
- 栈溢出攻击有多个分类和变体，但其本质均是对于**栈帧中返回地址的修改**，导致**EIP寄存器指向恶意代码**
- 下面假设在**没有任何内存防御机制**的条件下，介绍2个最简单的栈溢出攻击案例

栈区溢出攻击

- **简单的栈区溢出示例1：返回至溢出数据**

1. 攻击者发现可被利用的危险输入函数和可越界访问内存的变量

例如：著名的莫里斯蠕虫病毒就是利用了缺乏输入长度检查的gets函数



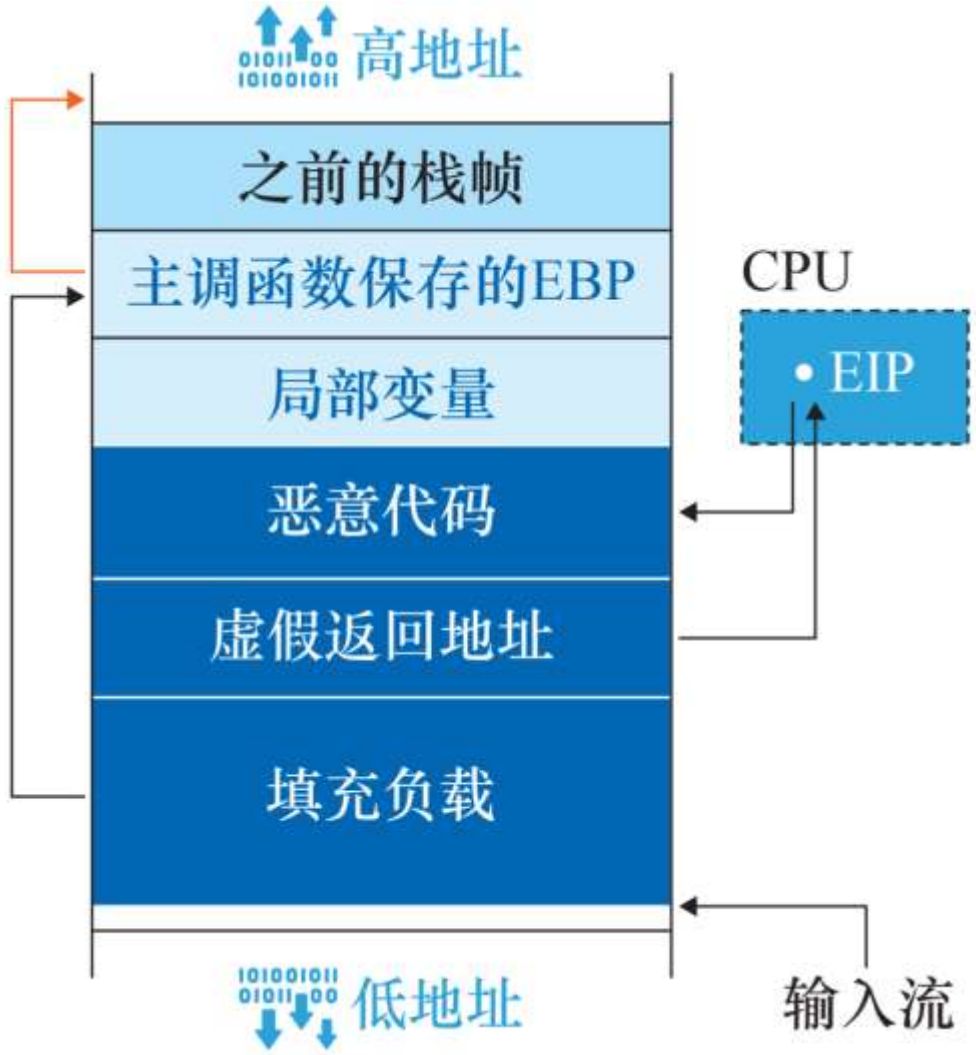
栈区溢出攻击

- **简单的栈区溢出示例1：返回至溢出数据**

2. 确定越界访问变量与返回地址的位置关系

局部变量存储于栈区，栈的增长方向向低地址，因而可以从变量地址加正向偏移访问返回地址

攻击者构造一个填充输入，以覆盖局部变量到返回地址间的内存，并覆盖掉返回地址



栈区溢出攻击

- **简单的栈区溢出示例1：返回至溢出数据**

3. 攻击者构造一段恶意代码，并设置返回地址为恶意代码开始的位置，恶意代码将在函数返回后被执行

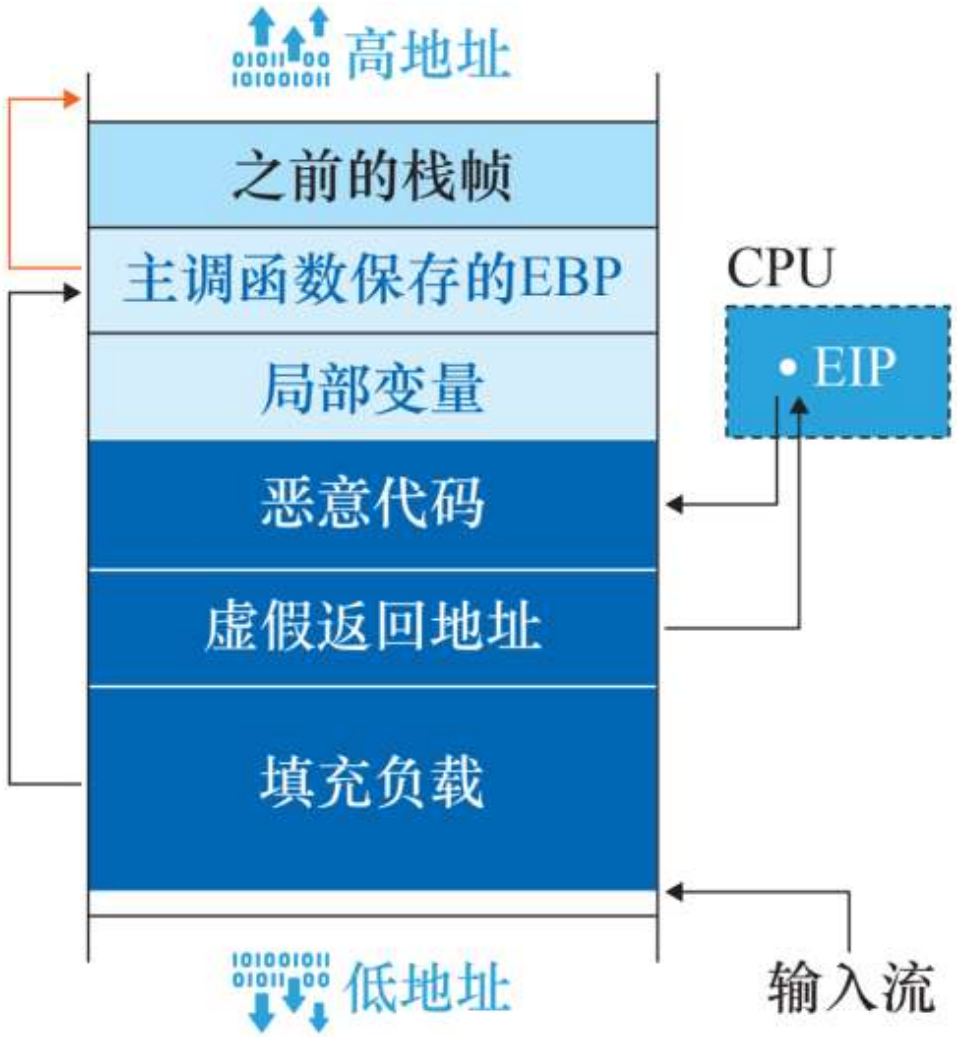
- **最终构造的恶意的输入分为三个部分：**

a. 局部变量到返回地址间的恶意填充

b. 返回地址的覆盖值

c. 恶意的代码段

此外，攻击者也可以不构造代码段，仅通过输入不存在的返回地址让程序崩溃



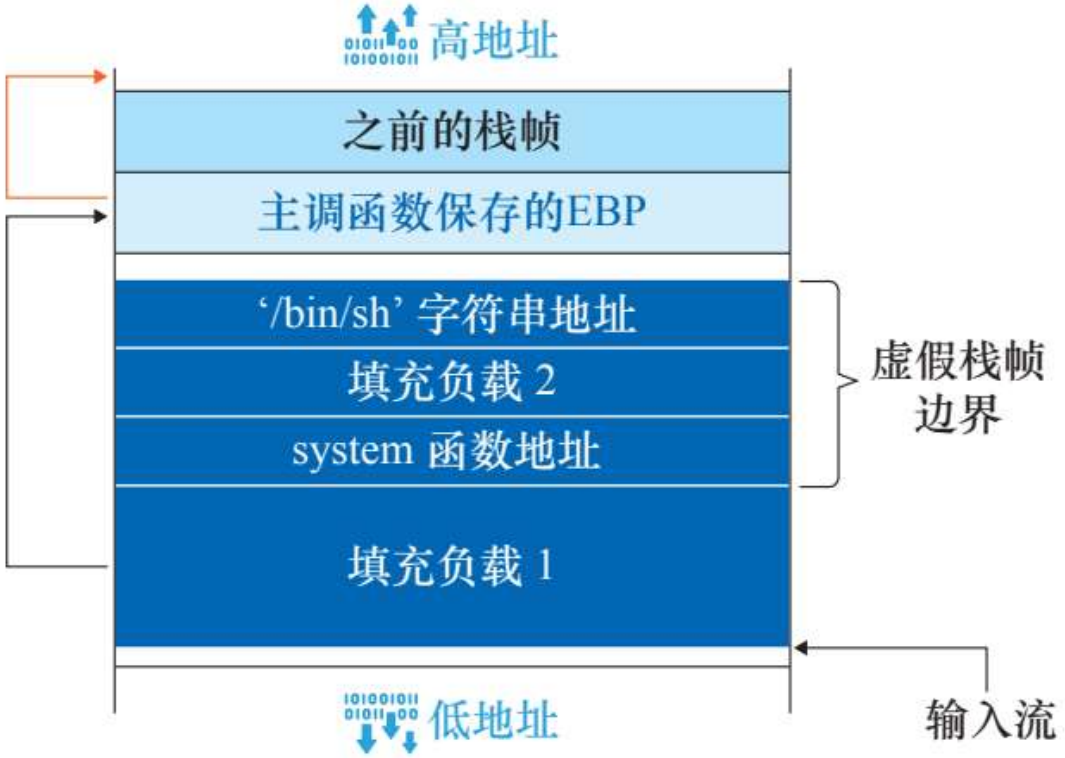
栈区溢出攻击

- **简单的栈区溢出示例2：返回至库函数**

在这个示例当中，攻击者希望进程调用某一动态链接库当中的库函数

假设攻击者希望调用libc的system函数，并传递参数为‘/bin/sh’进而获取操作系统的shell，执行任意指令

讨论：攻击者应该如何传递函数调用的参数?

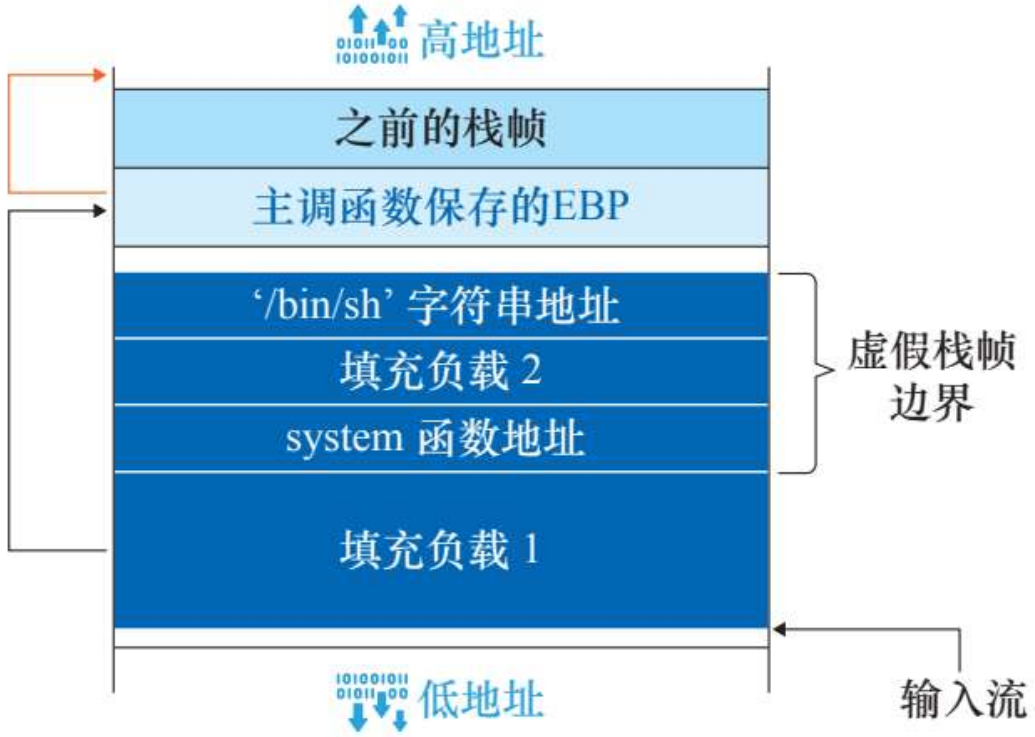


注. 假设内存映射段当中存在system函数

栈区溢出攻击

- **简单的栈区溢出示例2：返回至库函数**

攻击者在受害进程的栈区伪造
一个**栈帧的边界**



总体来说，攻击者在恶意输入当中构造了一个函数调用时的内存结构；
当进程返回时到system函数当中执行，system函数在高地址当中找到伪造的函数参数，完成恶意的函数调用过程

注. 靠上的padding payload 2的作用是填充返回地址的位置，相当于恶意调用system函数后的返回地址

栈区溢出攻击总结

- **以上简单的栈溢出攻击的局限性**

以上简单的栈溢出攻击在现实操作系统环境下几乎无法成功；为防御栈区溢出，已有诸多内存级别的保护机制，例如NX、ASLR、Stack Canary、DEP等将在第二节当中介绍

- 栈区溢出攻击是被最广泛使用的控制流劫持手段，我们将在第三节当中详细讨论**以栈溢出为基础**的复杂进程控制流劫持方案，并绕过基础操作系统防御机制

第1节 操作系统基础攻击方案

- 1.1 操作系统内存管理基础
- 1.2 基础的栈区攻击方案
- **1.3 基础的堆区攻击方案**

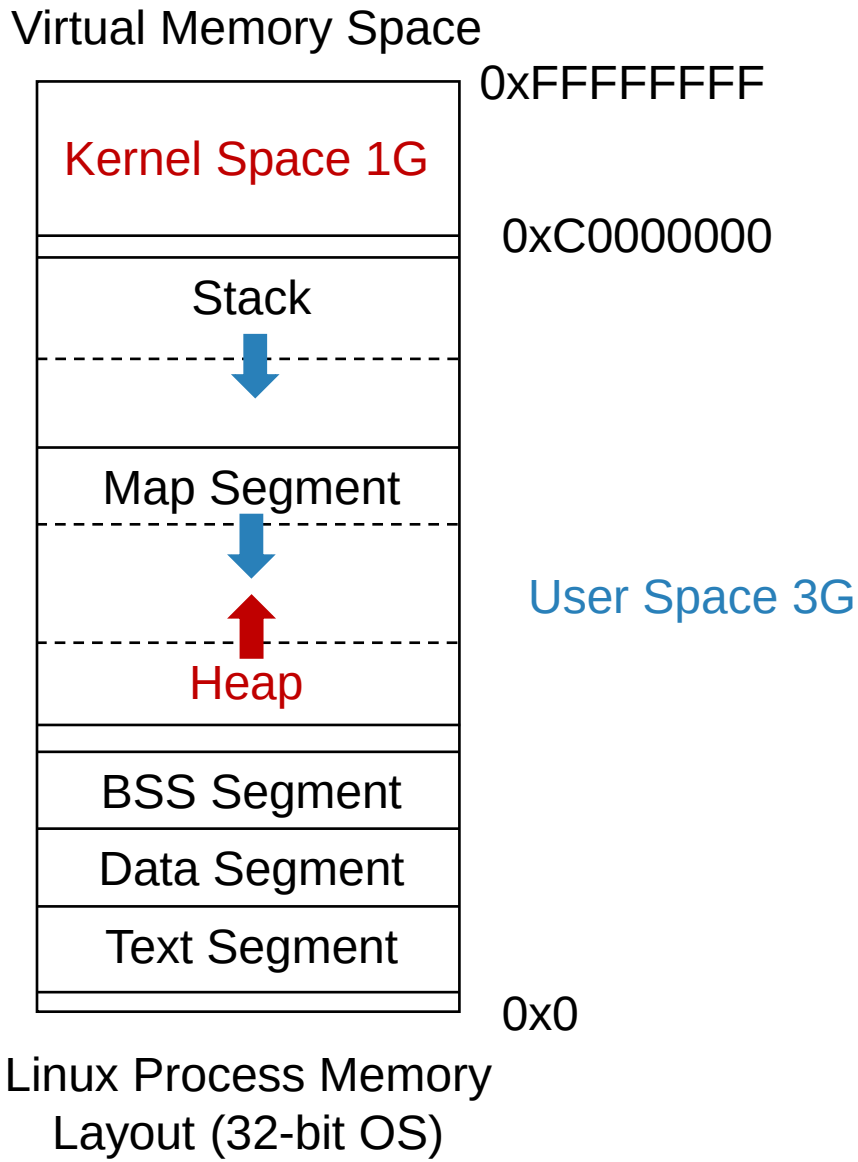
正常工作的堆管理器

在程序运行过程中，堆可以提供动态分配的内存，允许程序申请指定大小的内存

堆区是程序虚拟地址空间的一块连续的线性区域，它由低地址向高地址方向增长

我们一般称管理内存堆区的程序为**堆管理器**

称堆管理器分配的最小内存单元为**堆块（Chunk）**



Linux Process Memory Layout (32-bit OS)

正常工作的堆管理器

- **堆管理器**处于用户程序与内核中间地位，主要做以下工作：

1. **响应用户的申请内存请求。**向操作系统申请内存，然后将其返回给用户程序；堆管理器会预先向内核申请一大块连续内存，然后过**堆管理算法**管理这块内存；当出现了堆空间不足的情况，堆管理器会再次与内核行交互

2. **管理用户所释放的内存。**一般情况下，用户释放的内存并不是直接返还给操作系统的，而是由堆管理器进行管理；这些释放的内存可以来响应用户新申请的内存的请求

堆管理器的缓冲作用显著降低了动态内存管理的性能开销

正常工作的堆管理器

- 堆管理器通常不属于操作系统内核的一部分，而是属于标准C函数库的一部分，根据标准C函数库的实现而采用不同堆管理器

ptmalloc2多线程堆管理器，是glibc的堆管理器，
是最被广泛使用的堆管理器，应用于绝大多数的Linux发行版上

- 其他常见的堆管理器有：
  - dlmalloc堆管理器为glibc的早期堆管理器，所有线程共享同一堆区
  - musl堆管理器，适配于嵌入式系统

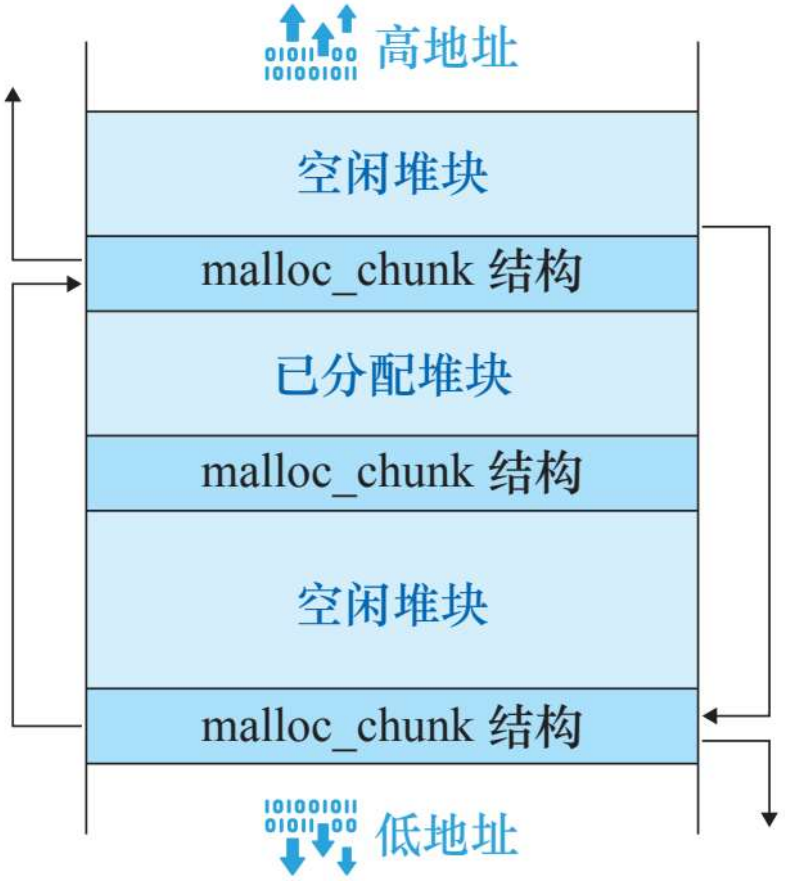
这些堆管理器的根本区别在于堆管理算法和管理元数据

正常工作的堆管理器

- ptmalloc2管理堆区内存的最小单元是堆块（Chunk）是向内核申请和归还的最小单元

- 每一个Chunk分为头部数据结构为malloc_chunk结构体（低地址），之后为分配或者未分配的数据块（高地址）

- 空闲堆块（Free Chunk）之间通过双向链表链接，并根据大小由5个Bin分类组织

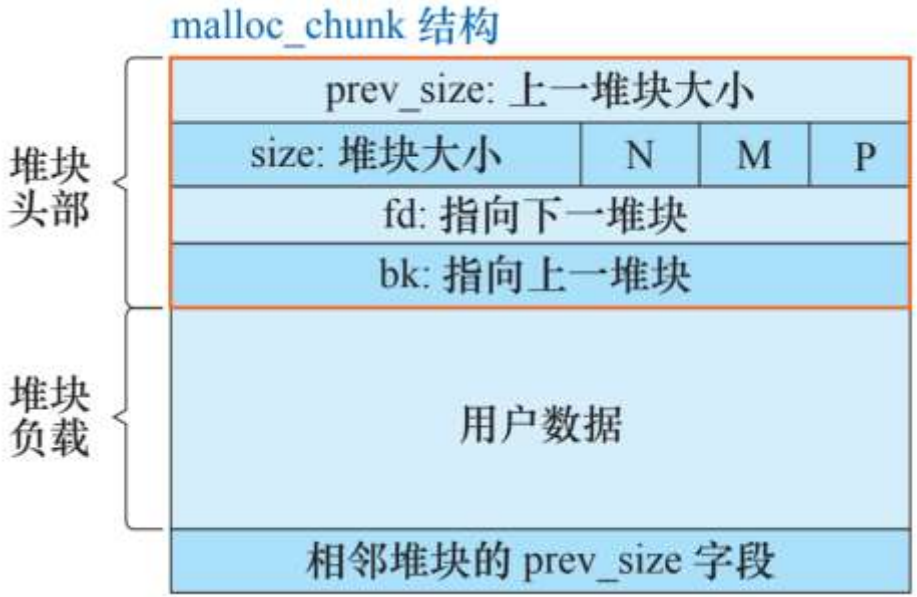


堆管理算法直接操纵malloc_chunk结构和5个Bin实现堆区的内存管理

正常工作的堆管理器

- **堆管理元数据结构 malloc_chunk：**

1. prev_size：当上一个Chunk为空闲，存储上一个Chunk大小，否则存上一个Chunk的数据
2. size：该Chunk大小
3. NON_MAIN_ARENA：是否属于子线程
4. IS_MMAPED：是否由mmap分配
5. PREV_INUSE：前一个Chunk是否被分配
6. bk，fd：链接Bin当中空闲块的前后向链表指针，只有在空闲时使用



面向堆区攻击方案的核心在于: 如何恶意操纵堆管理数据结构

堆区溢出攻击

堆溢出攻击

是一类攻击者越界访问并篡改堆管理数据结构，实现恶意内存读写的攻击

- 堆区溢出攻击是堆区最常见的攻击方式，这种攻击方式可以实现恶意数据的覆盖写入，进而实现进程控制流劫持

堆溢出的使用广泛，25%针对windows7的攻击都是堆区溢出

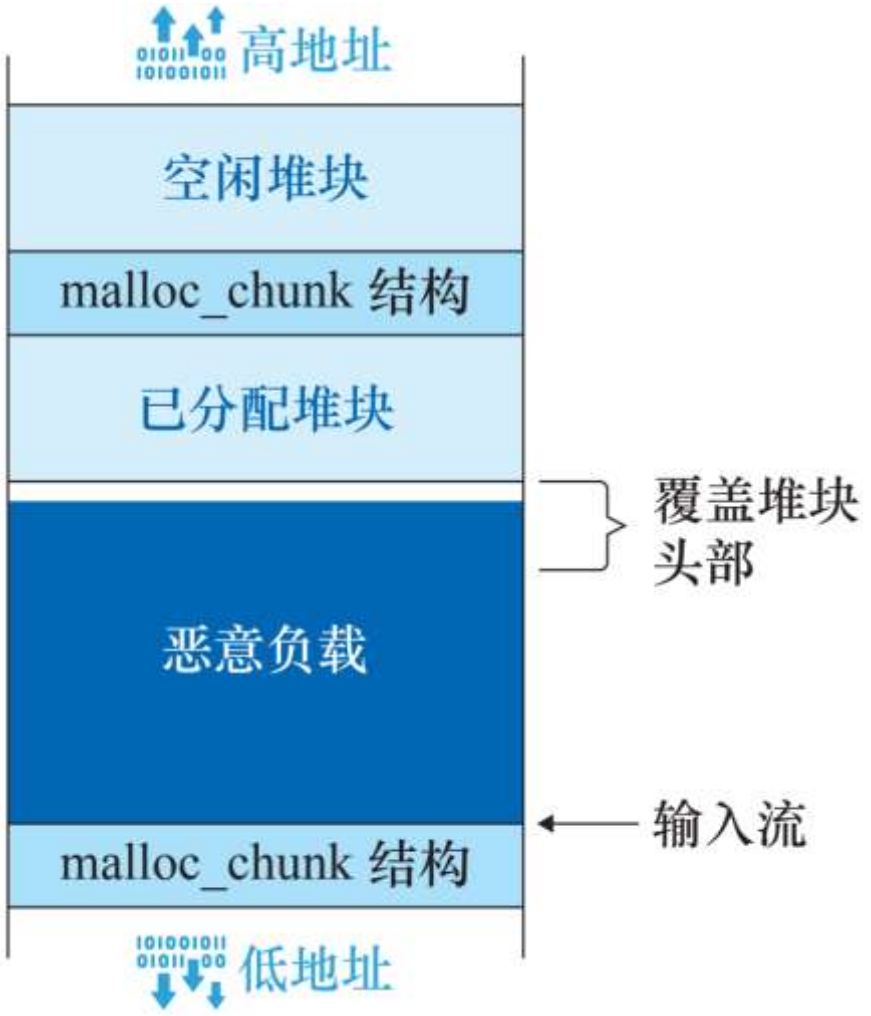
Heap Vulnerability	Occurrences
Heap Overflow	673
Use-After-Free	264
Heap Over-Read	125
Double-Free	35
Invalid-Free	33

注. 数据来自CVE-2017当中的检索结果

堆区溢出攻击

- **最简单的堆区溢出：**

直接覆盖malloc_chunk首部为无意义内容，在堆管理器处理管理元数据时将造成崩溃

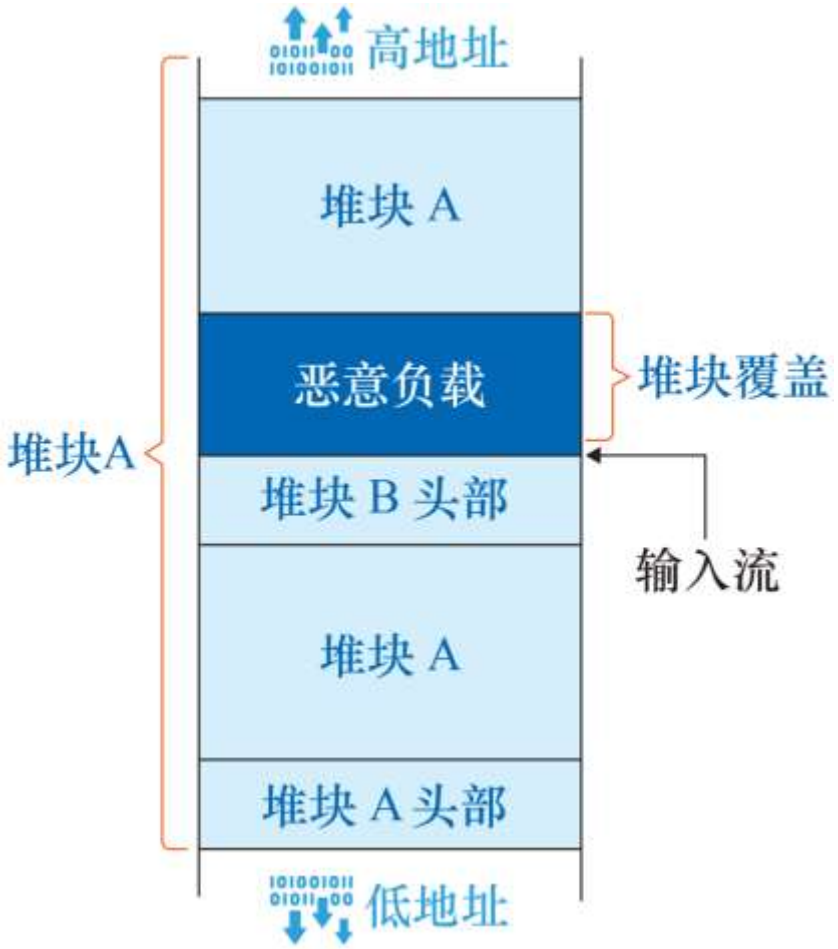


堆区溢出攻击

- **堆区溢出：构造堆块重叠（Heap Overlap）**

堆块重叠是一种**病态**堆区内存分配状态，
同一堆区逻辑地址被堆管理器**多次分配**

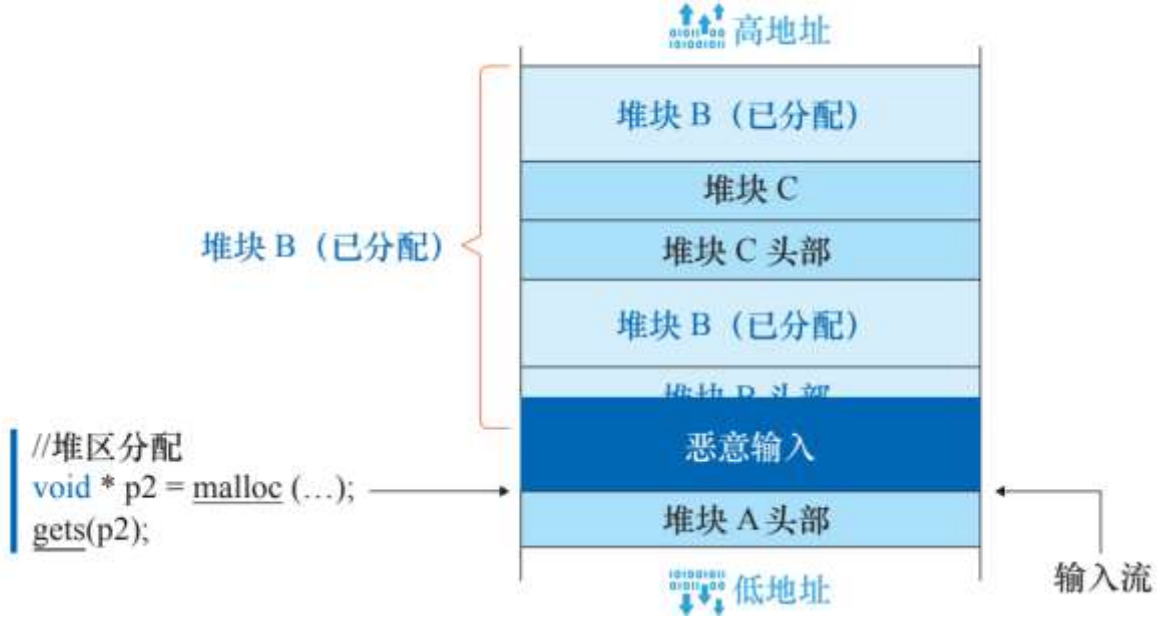
如右图所示，造成Heap Overlap之后攻击者可以通过写入一个堆块，实现对另一堆块内容的写入；同理，读出被覆盖堆块当中的数据



堆区溢出攻击

- **我们考虑一个案例：堆块A是可以发生溢出的堆块，其中B和C是被分配（allocated）状态的块，而且C是我们的攻击目标块**

- 攻击者首先通过**堆区溢出**数据去改写Chunk B的size域，把Chunk C包含到Chunk B当中，以此构造了堆块重叠

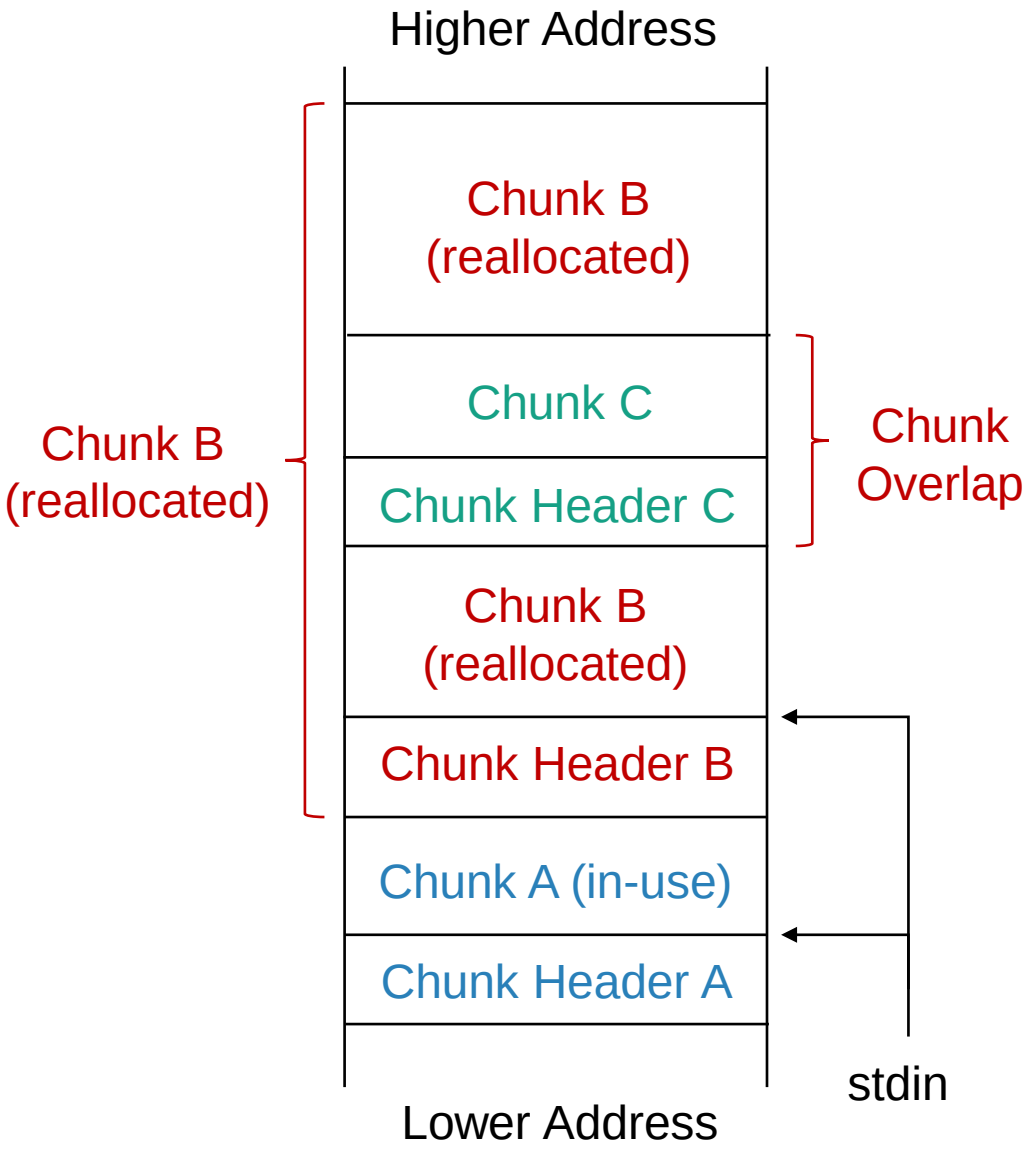


注. size字段在malloc_chunk结构的第一个字节（管理已分配堆块时），因而仅溢出1字节就可以覆盖到size域

堆区溢出攻击

- 而后攻击者操纵控制逻辑，使被修改了size字段的Chunk B将被重新分配

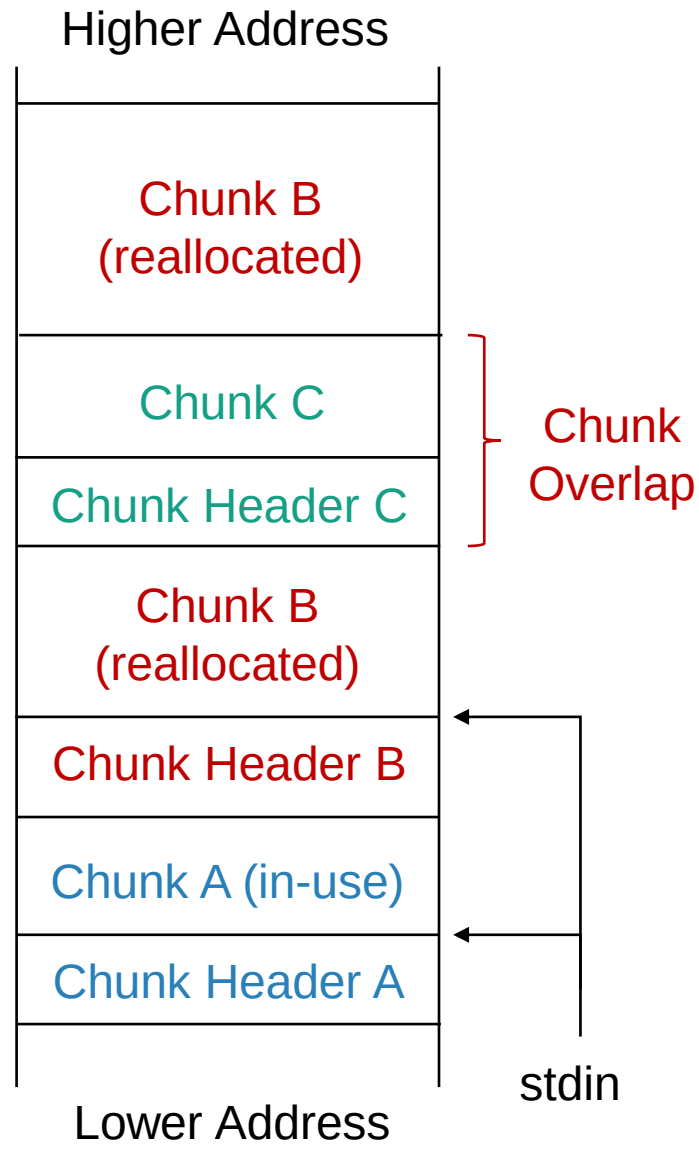
- 最终构造堆块重叠，攻击者可以通过读/写被重新分配的Chunk B来读/写块Chunk C当中的数据



堆区溢出攻击

- 总结，上述构造堆块覆盖的方案需要受害程序满足以下的条件：

① 存在错误的输入检查逻辑，使攻击者可以进行溢出（溢出1个字节即可）
② 对应的受害程序应能产生符合条件的堆区布局也就是连续分配的三个堆
③ 且攻击者可以操纵控制流实现Chunk B的释放和重新分配，并能在分配后进行读写



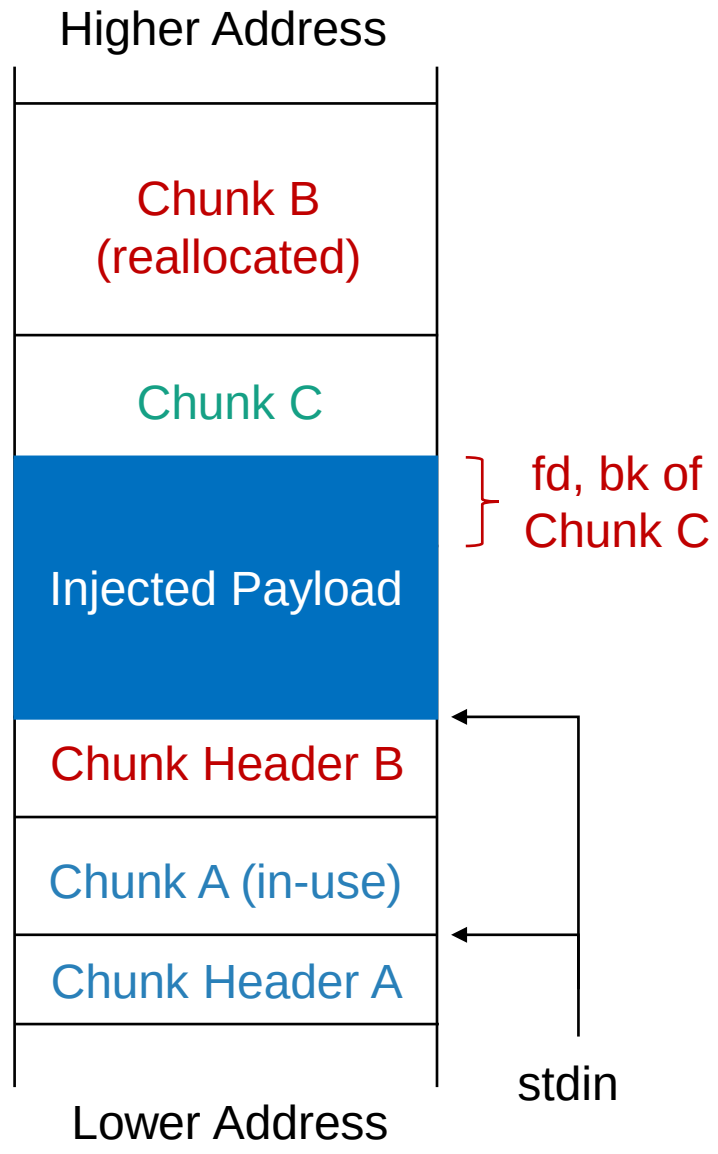
堆区溢出攻击

- **更加复杂的堆区溢出攻击：利用堆管理其他机制**

例如，基于unlink机制的堆区溢出攻击：

unlink宏从**空闲堆块构成的双向链表**中，提取**空闲堆块**，而后返回给用户**空闲堆块**

攻击者将设法用溢出数据**覆盖前/后向链表指针**（fd，bk字段），在unlink宏调用时可以将任意内存地址当作未分配的堆块使用；
最终，攻击者可**读/写任意内存区域**（write-anything-anywhere），并以此为基础进行更复杂的攻击

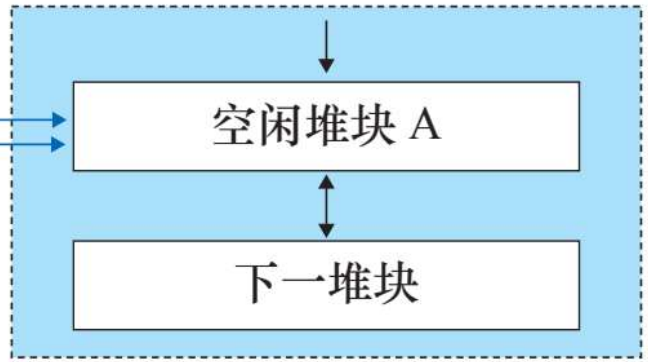


堆区的其他攻击：Use-After-Free

- **Use-After-Free** 是进程由于实现上的错误，**使用已被释放的堆区内存**，UAF是一种存在广泛的漏洞，仅2020年上半年，**CVE当中就汇报了超过90种UAF漏洞**

被free函数释放的堆块内存仍然可以被继续使用，当再次调用malloc分配内存时，会同时有两个指针指向同一堆块造成 **堆块重叠**

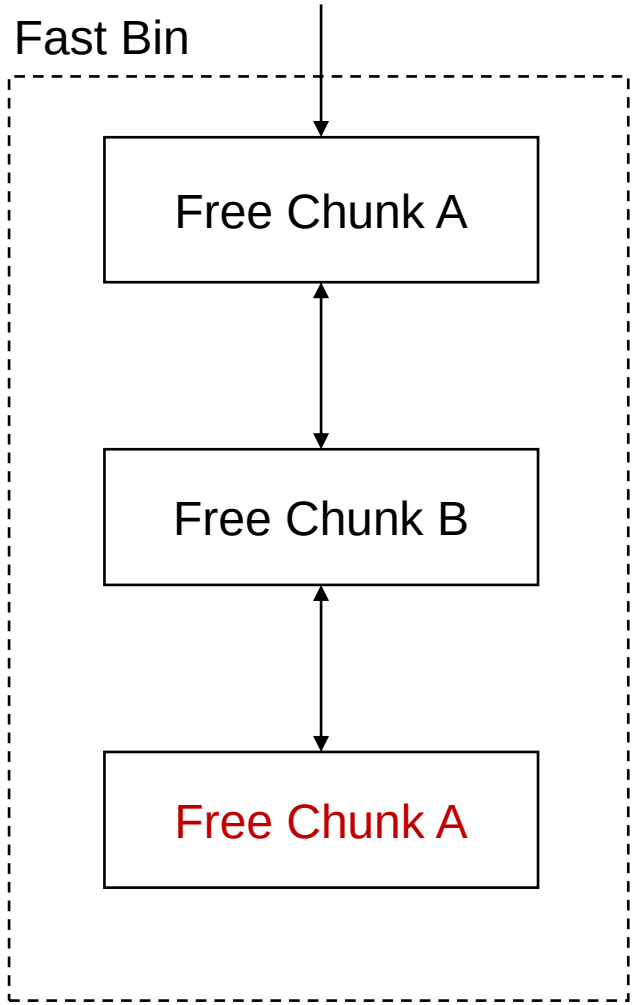
```
// 空悬指针
free(p1);
…
void * p2 = malloc(…);
//已分配堆块 A
```



注. 指向已被释放的内存的指针为空悬指针（Dangling Pointer）

堆区的其他攻击：Double-Free

- Double-Free指的是进程**多次释放统一堆块**，被多次释放的堆块将**被堆管理器分配多次**，最终产生堆块重叠；



Double-Free多发生在Fast-Bin当中，因为Fast-Bin当中的空闲块更倾向于被反复分配与释放

注. Fast-Bin 用于收集较小的空闲堆块（16-80B），方便反复申请小块内存的场景

堆区的其他攻击：Heap Over-read

- 堆溢出攻击越界写入并覆盖堆区数据，而Heap Over-Read则直接越界读出堆区数据，造成信息泄露

著名的Heartbleed Attack是最典型的Heap Over-Read Attack

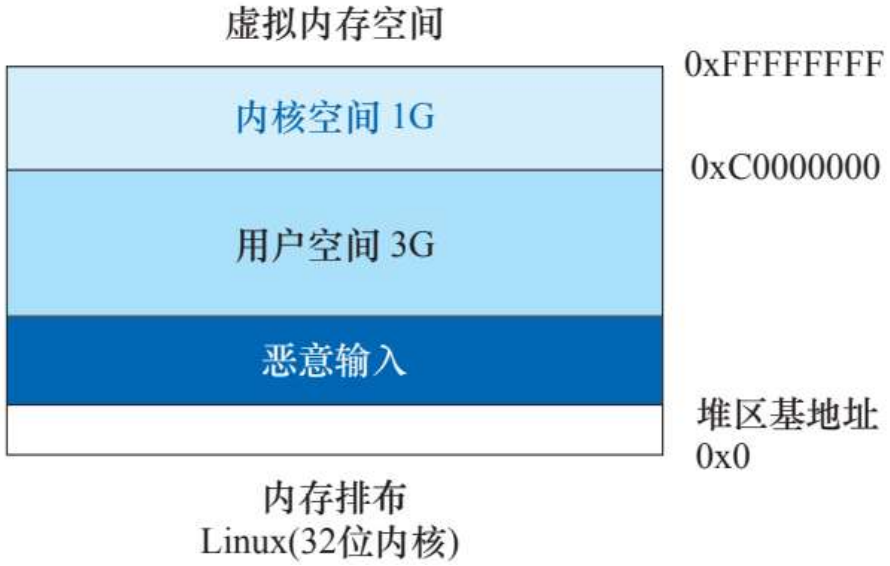
- OpenSSL的TLS实现当中，在处理心跳包时未能对长度字段做合理校验，导致攻击者可以构造恶意数据包，越界读取心跳包数据之后的堆区内存；这些内存包含了私钥等重要信息

该漏洞可倾泻的数据量高达64KB，而Fast-Bin当中的堆块最大大小仅为80B（为其大小的800-4000倍）

堆区的其他攻击：Heap Spray

- 堆喷（Heap Spray）并非是一种内存攻击的辅助技术；堆喷申请大量的堆区空间，并将其中填入大量的**滑板指令**（NOP）和攻击恶意代码；

- 堆喷使用户空间存在大量恶意代码，若EIP指向堆区时将命中**滑板指令区**，受害进程最终将“滑到”恶意代码



堆喷对抗地址的随机浮动类型的防御方案

并实现了恶意代码的注入

第2节 操作系统基础防御方案

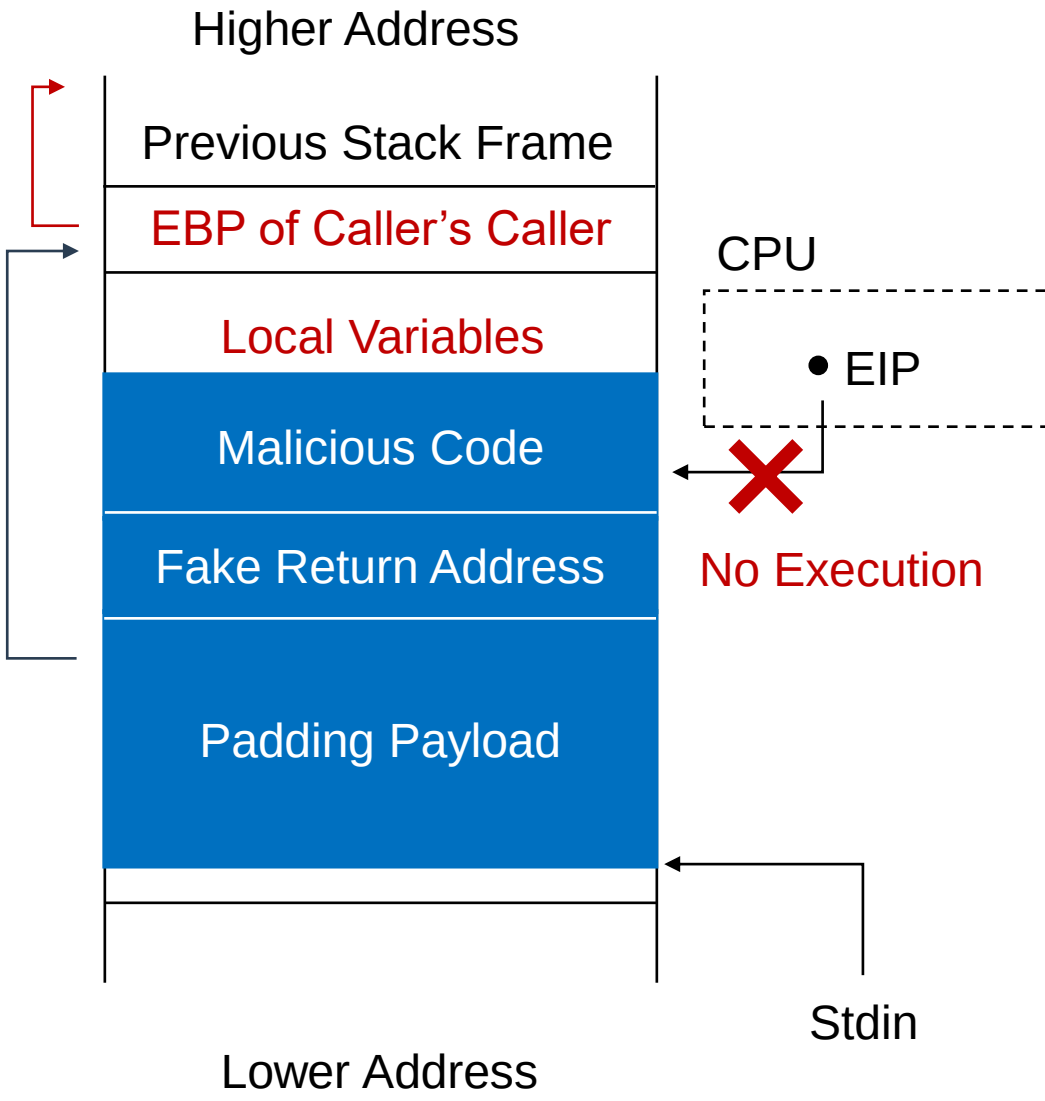
- 2.1 W^X（NX，DEP）
- 2.2 ASLR
- 2.3 Stack Canary
- 2.4 SMAP, SMEP

内存防御技术：W^X

- W^X 是**写与执行不可兼得**，即每一个内存页拥有**写权限或者执行权限**，不可兼具两者

当W^X最早在FreeBSD 3.0当中被实现，在Linux下的别名为**NX**（No eXecution），Windows下类似的机制被称为**DEP**（Data Execution Prevention）

当W^X生效时，**返回至溢出数据的栈溢出攻击**失效，因为无法执行位于栈区的注入的恶意代码；
但仍有方法可以绕过NX保护机制，将在下一节的高级控制流劫持方案中介绍

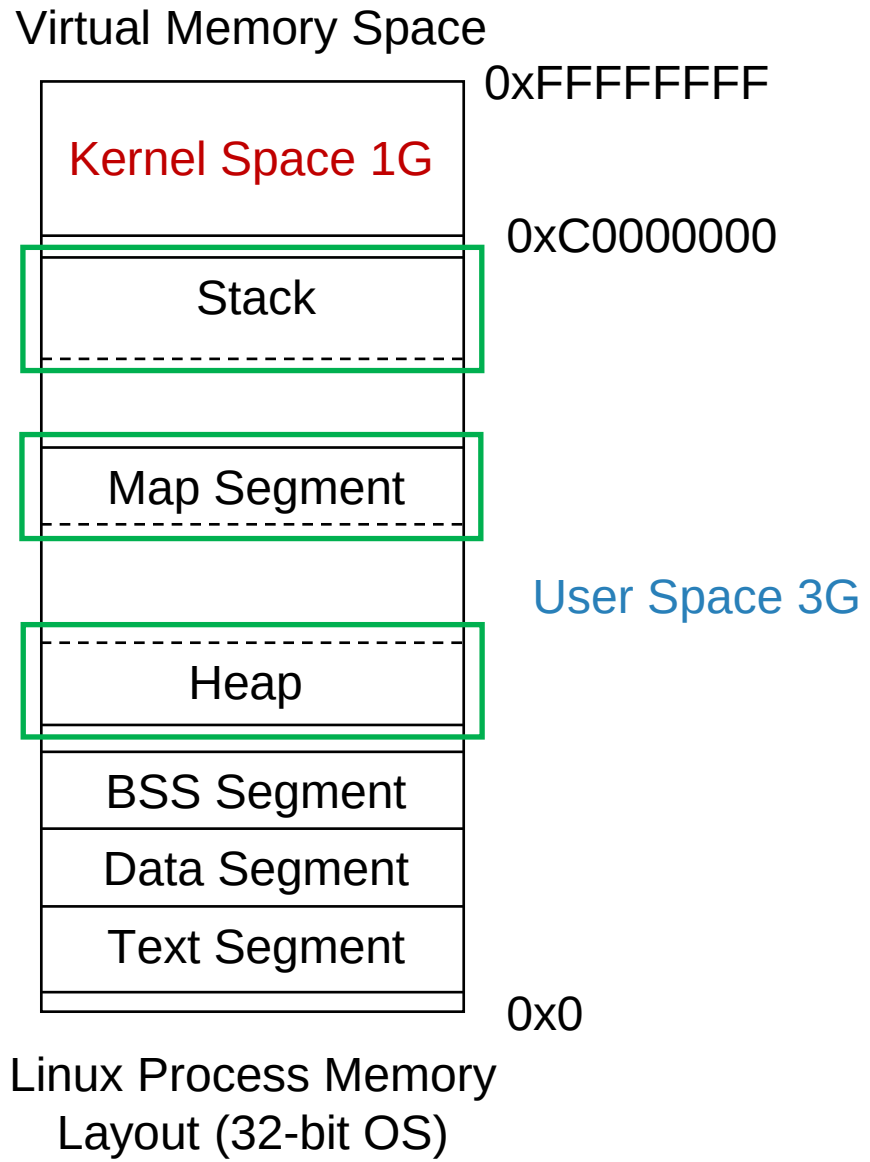


内存防御技术：ASLR

- ASLR（address space layout randomization）是一种对虚拟空间当中的**基地址进行随机初始化**的保护方案；以防止恶意代码定位进程虚拟空间当中的重要地址；目前在各主流操作系统下均有实现

ASLR随机化的对象包含了：

1. 共享库的基地址（.so文件加载的基地址）
2. 栈区的基地址
3. 堆区的基地址



注．ASLR实现需要编译器的PIE支持（Position Independent Executable）才可将代码加载到随机化的地址上

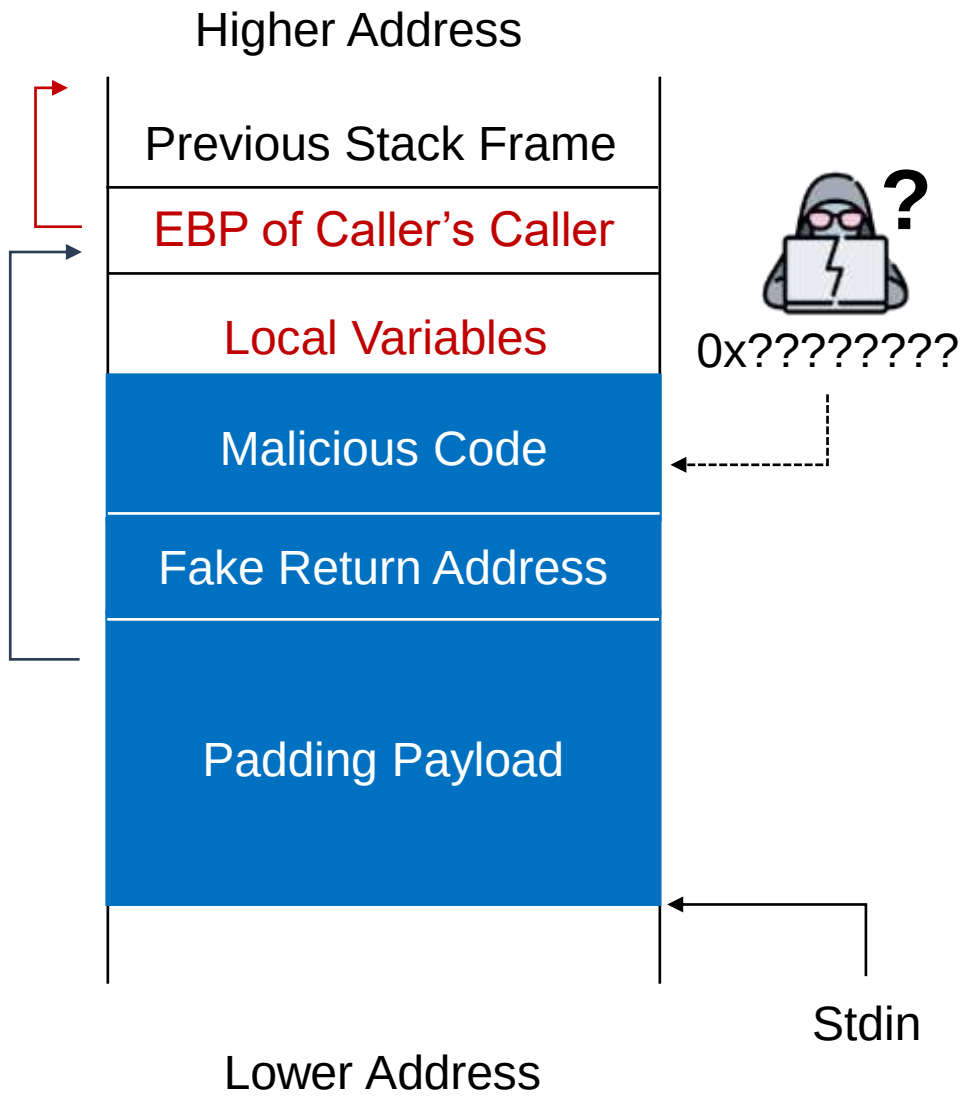
内存防御技术：ASLR

- ASLR随机化**栈区基地址**，注入到栈区的恶意代码内存位置也无法被攻击者确定

上一节当中**返回至溢出数据的栈溢出攻击**失效

因为恶意代码位于栈区，地址被ASLR随机化，

攻击者无法确定并写入恶意代码的**绝对内存地址**

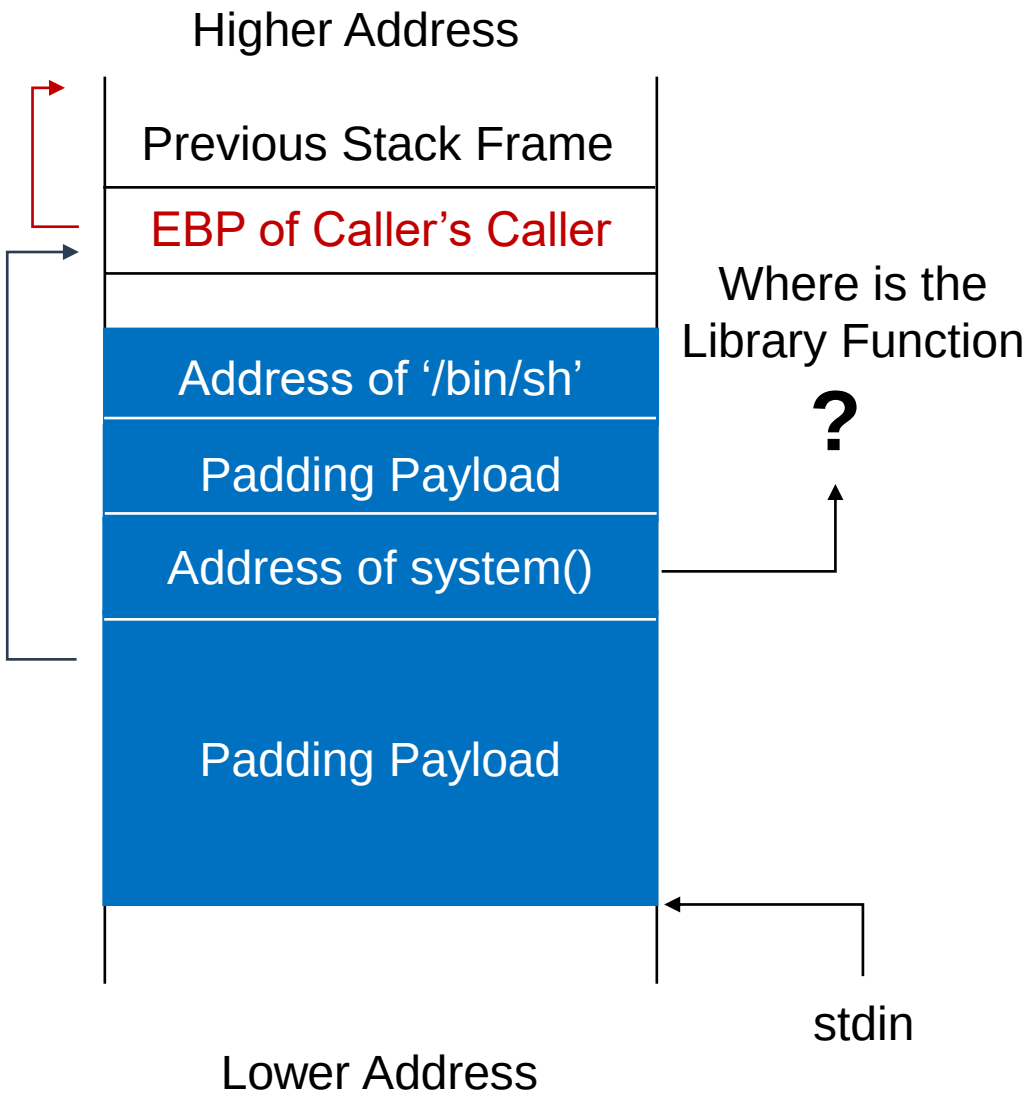


内存防御技术：ASLR

- ASLR随机化**共享库基地址**，库函数的内存位置也无法被攻击者确定

上一节当中**返回至库函数的栈溢出攻击**失效

因为ASLR将导致攻击者无法定位目标库函数

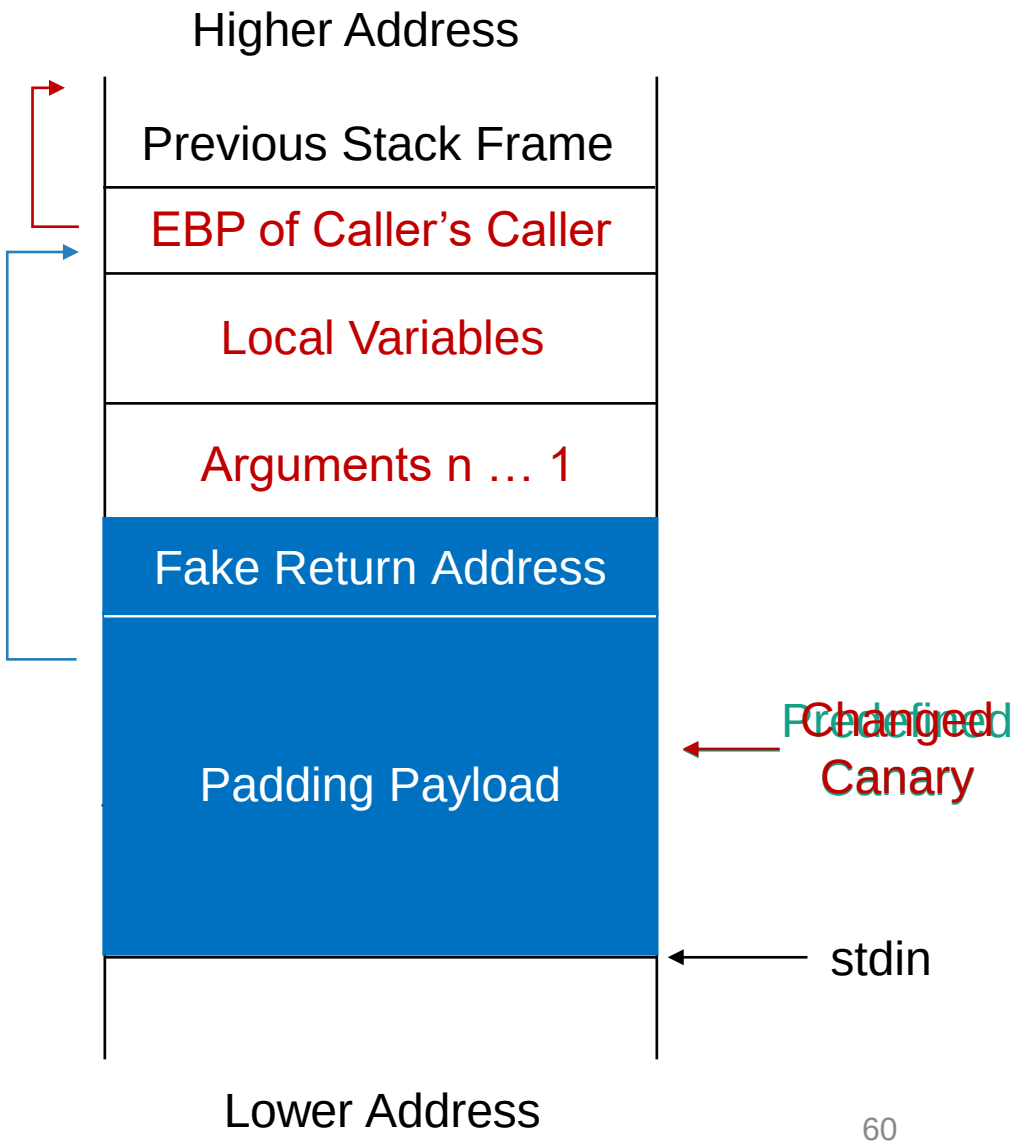


内存防御技术：Stack Canary

- Canary的本意是金丝雀，Stack Canary是一种防御栈区溢出的方案

其本质是，在保存的栈帧基地址（EBP）之后插入一段信息，当函数返回时验证这段信息是否被修改过

当发生栈区溢出时，攻击者为了修改返回地址，必须先覆盖Stack Canary，造成攻击行为暴露



注. 其得名缘由：https://en.wikipedia.org/wiki/Stack_buffer_overflow

内存防御技术：SMAP, SMEP

SMAP和SMEP是两种基础的**内存隔离技术**

- SMAP（Supervisor Mode Access Prevention，管理模式访问保护）禁止内核访问用户空间的数据
- SMEP（Supervisor Mode Execution Prevention，管理模式执行保护）禁止内核执行用户空间代码，是预防**权限提升**（Privilege Escalation）的重要机制

SMAP/SMEP 和 W^X 均需要处理器硬件的支持

内存防御技术总结

- 虽然操作系统提供了大量防御内存攻击的方案，但这些防御方案仍可以被攻击者挫败或绕过：

对于Stack Canary，作为Canary的内容可能被泄露给攻击者，或被暴力枚举破解[1]

对于ASLR已有去随机化方案，泄露内存分布信息[2,3]

在第三节将介绍ROP等进程控制流劫持方案亦可以绕过ASLR、NX的保护机制

1. Wei Wu, et al. "KEPLER: Facilitating Control-flow Hijacking Primitive Evaluation for Linux Kernel Vulnerabilities." 28th USENIX Security Symposium, 2019.
2. Daniel Gruss, et al. "Prefetch side-channel attacks: Bypassing SMAP and kernel ASLR." Proceedings of the 2016 ACM SIGSAC conference on computer and communications security.
3. Ben Gras, et al. "ASLR on the Line: Practical Cache Attacks on the MMU." NDSS. Vol. 17. 2017.

本节相关的前沿研究工作

- 对于堆区管理安全：
  GUARDER: A Tunable Secure Allocator[1] 提出一种安全的堆区分配方案，**设计安全的堆管理算法**一直是长时间来难以解决的问题
- 对于栈区管理安全：
  Stack Bounds Protection with Low Fat Pointers[2] **新型栈区内存边界保护**方案，出发点与Stack Canary类似，均为**保护栈帧边界**防止**返回地址篡改**
- 内存保护方案的安全性依赖于微处理器架构安全：
  ASLR on the Line: Practical Cache Attacks on the MMU[3] 是一种基于**微处理器架构侧信道**方案的去除ASLR随机地址浮动攻击

1. Sam Silvestro, et al. "Guarder: A tunable secure allocator." 27th USENIX Security Symposium, 2018.
2. Ben Gras, et al. "ASLR on the Line: Practical Cache Attacks on the MMU." NDSS. Vol. 17. 2017.
3. Gregory J., Duck, et al. "Stack Bounds Protection with Low Fat Pointers." NDSS. 2017.

第3节 高级控制流劫持方案

进程的内核态和用户态

- Linux下进程可处于**内核态**或**用户态**，内核态下拥有更高的指令执行权限（在Intel x86_32下对应ring0）用户态下只拥有低权限（对应ring3）

微处理器在指令执行时，对权限进行严格检查，
管理用户直接访问硬件资源的权限，提升系统的安全性

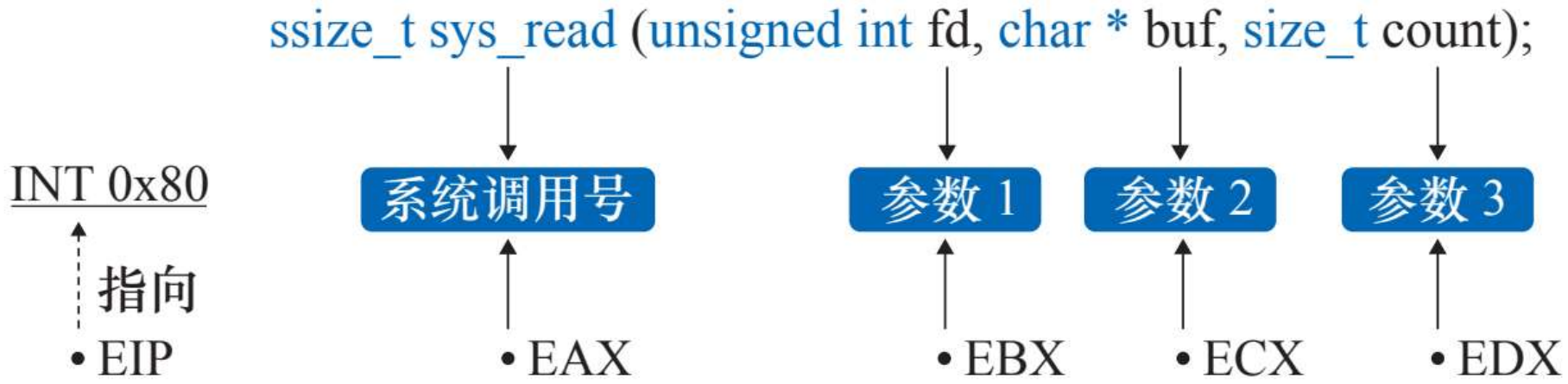
- Linux下内核态与用户态的切换主要由三种方式触发：（1）系统调用（2）I/O设备中断（3）异常执行；其中系统调用是进程主动转入内核态的方法

因而也称系统调用是**内核空间**与**用户空间**的桥梁

进程触发系统调用

- Linux在x86_32架构下触发系统调用的方法：（下图以触发sys_read为例）

1. 将EXA设置为对应的系统调用号
2. 将EXB、EXC等寄存器设置为系统调用参数
3. EIP指向并执行中断触发指令，触发0x80中断

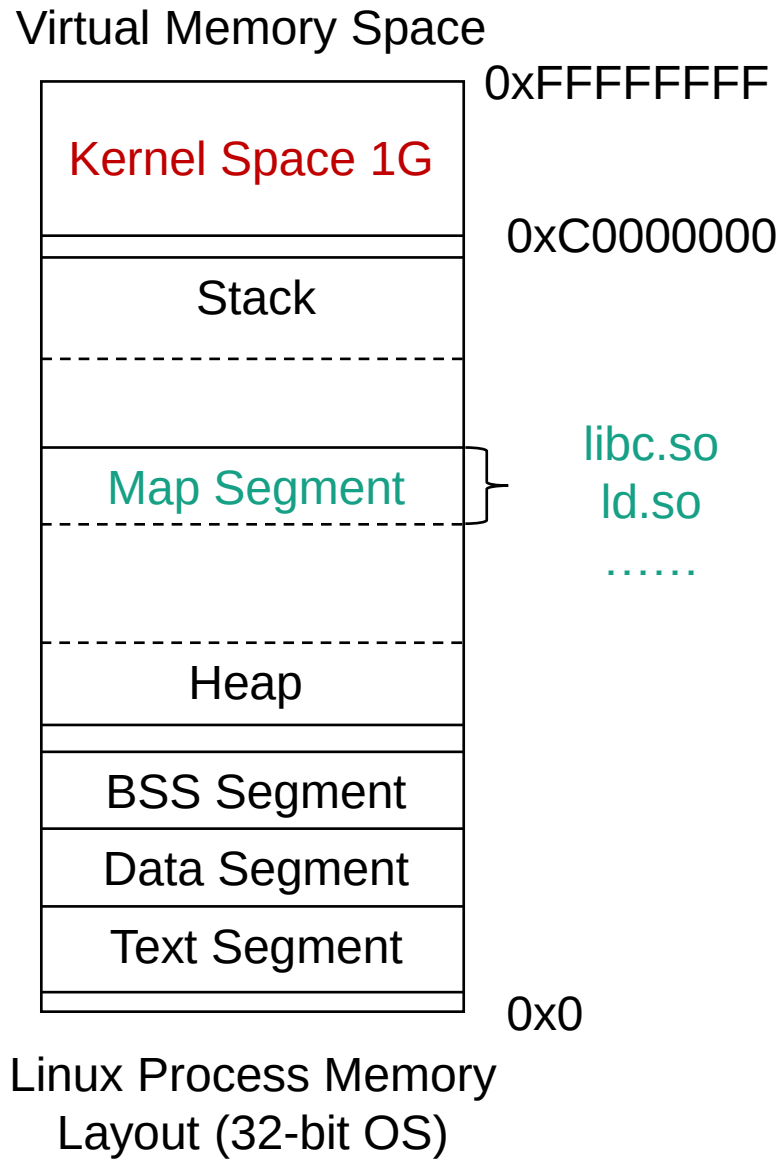


进程与共享库机制

- **共享系统库**是对系统调用的封装，例如C语言的标准系统库 glibc（libc.so.6）

- 共享库机制的实现方法是编译器的**动态链接**机制，动态链接文件在Linux下以.so结尾，在Windows下以.dll结尾

- 在进程的执行过程中，操作系统**按需求**将共享库以**虚拟内存映射的方式**映射到用户的虚拟内存空间，位于内存映射段（Memory Map Segment）

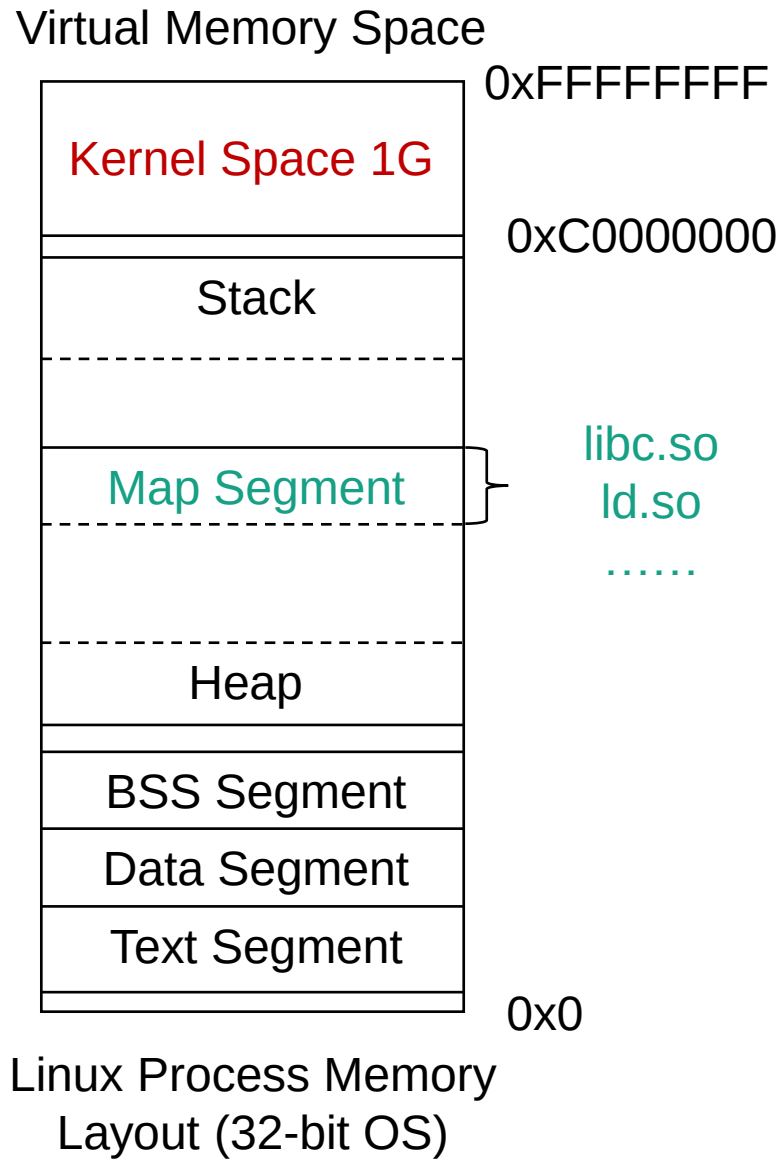


进程与共享库机制

- 与编译器的动态链接机制对应的是编译器静态链接机制，静态链接库在编译时将目标代码直接插入程序

静态链接库在Linux下以.a结尾，例如标准C++静态库，libstdc++.a

静态链接库**无法实现代码共享**，因为静态链接不属于共享库机制的一部分

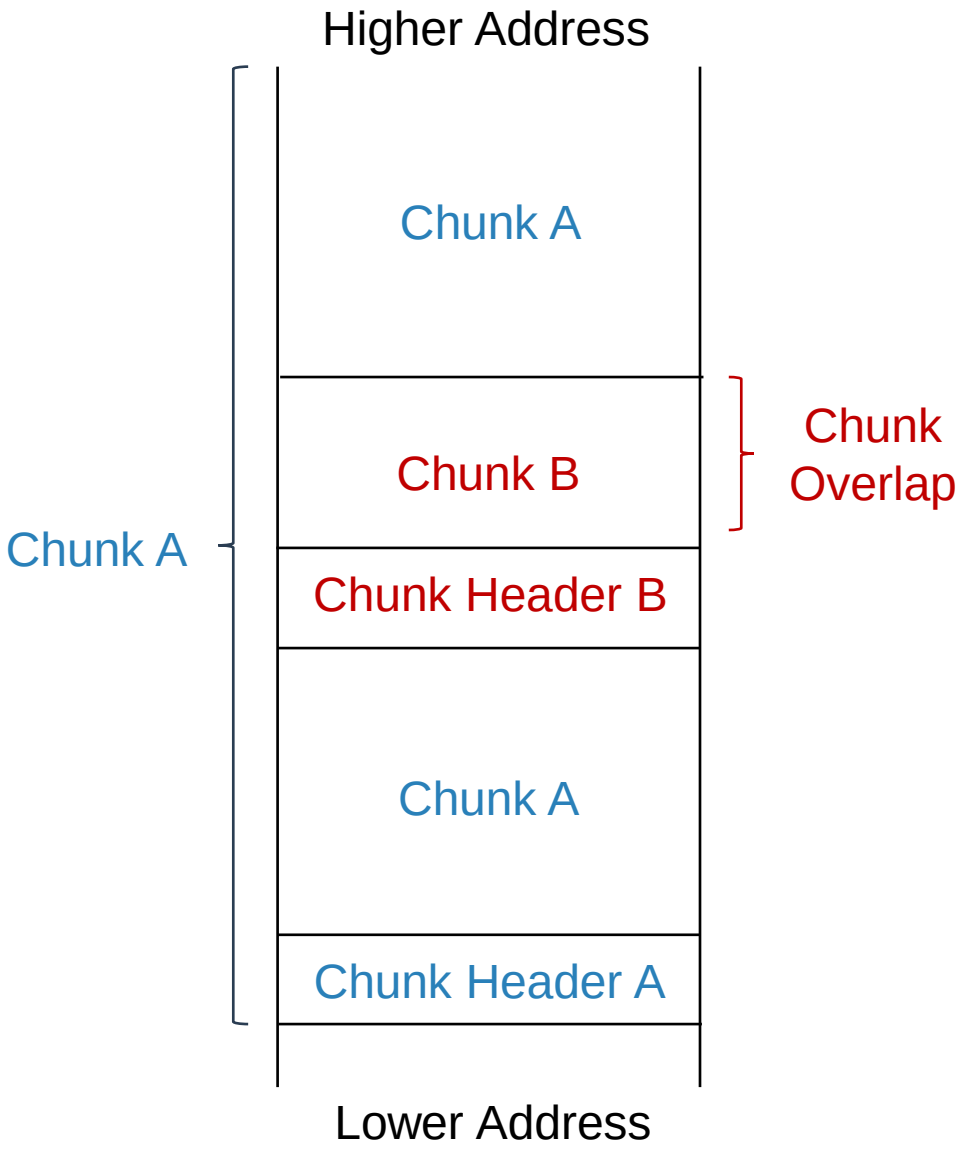


回顾：内存角度的攻击

- 第一节曾介绍，堆区溢出攻击通过越界写修改相邻堆块元数据，产生**堆块重叠**，进而攻击者恶意读写其他堆块的数据

- 结合unlink等其他堆管理器机制，攻击者可以恶意读写任意内存位置

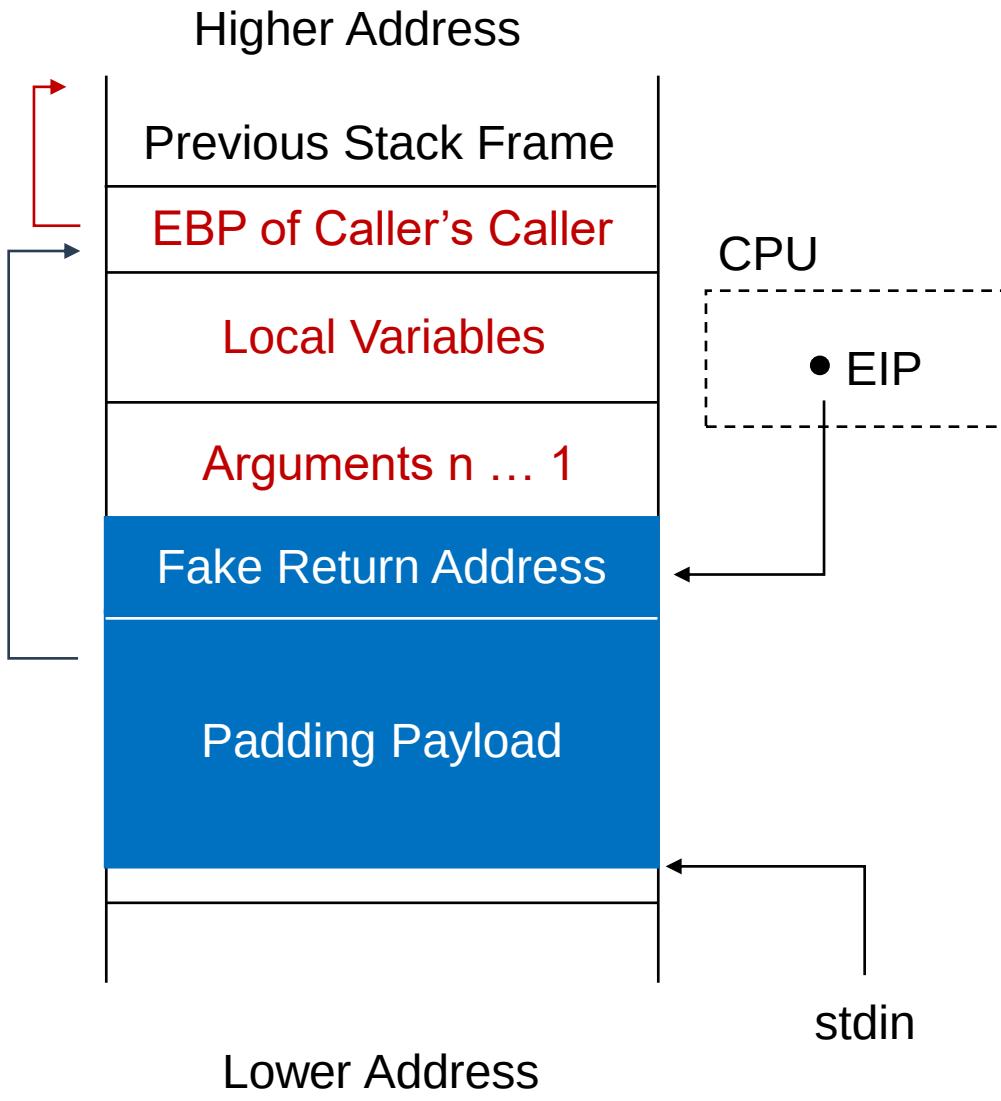
本章将以堆溢出攻击为基础
构造更复杂的进程控制流劫持方案



回顾：内存角度的攻击

- 在第一节曾介绍，简单的栈区溢出攻击修改返回地址，实现进程控制流劫持

- 攻击者找寻可越界访问内存的变量；构造恶意负载（Paylaod）；输入程序以覆盖返回地址；并提供要被执行的恶意代码



然而，简单的栈区溢出攻击在ASLR，NX，Stack Canary等保护措施下已难以成功

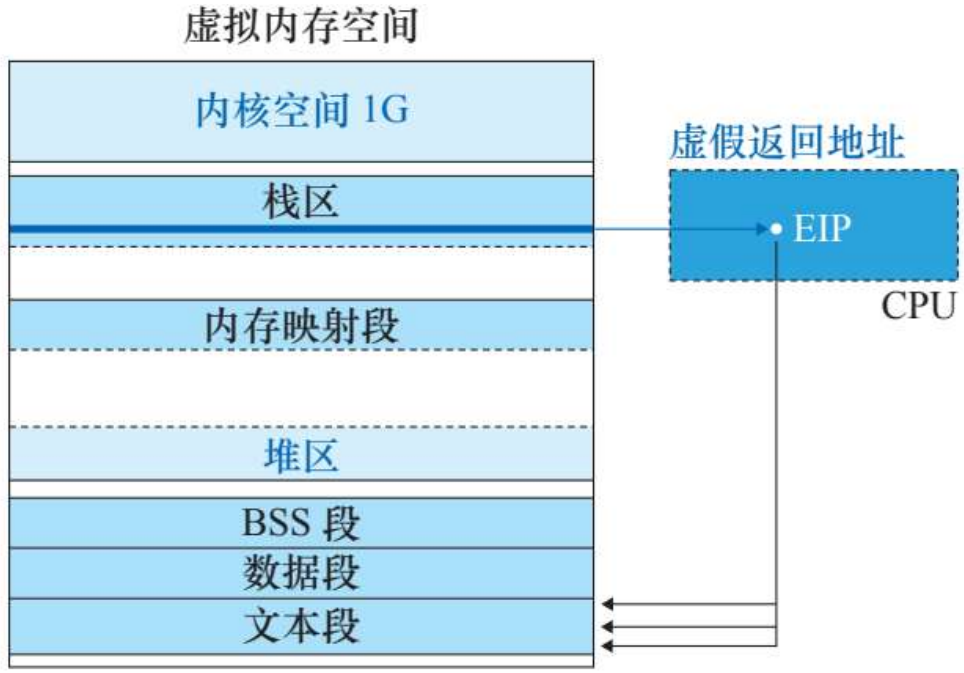
第3节 高级控制流劫持方案

- 3.1 进程执行的更多细节
- **3.2 面向返回地址编程**
- 3.3 全局偏置表劫持
- 3.4 虚假vtable劫持

面向返回地址编程

- 面向返回地址编程（Return-Oriented Programming，ROP）基于**栈区溢出攻击**，将返回地址设置为**代码段中的合法指令**，组合现存指令修改寄存器，劫持进程控制流

ROP利用进程内存空间当中现存的指令，**“编写”**恶意程序，劫持进程的控制流

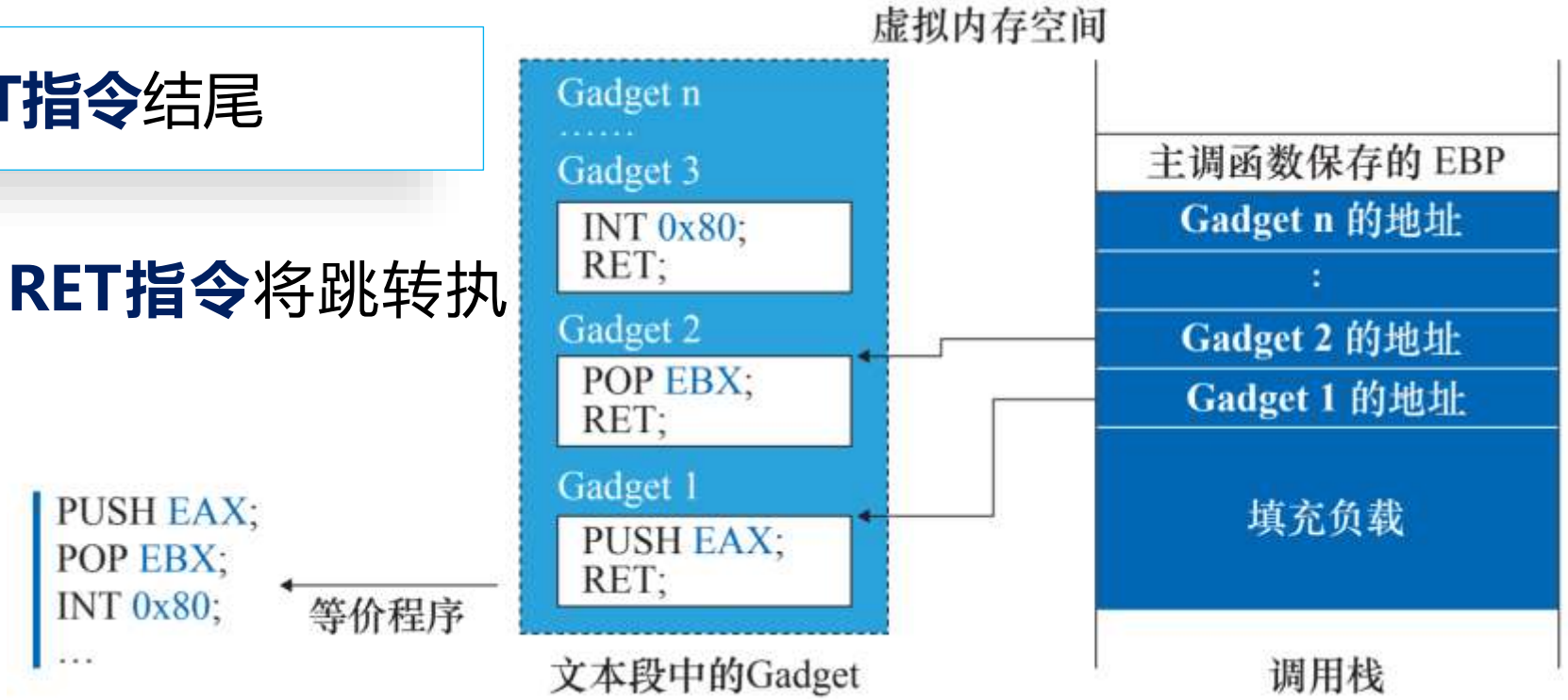


面向返回地址编程

- 面向返回地址编程构造恶意程序所需的指令片段被称为 **Gadget**

Gadget均以**RET指令**结尾

当一个Gadget执行后，**RET指令**将跳转执行下一个Gadget

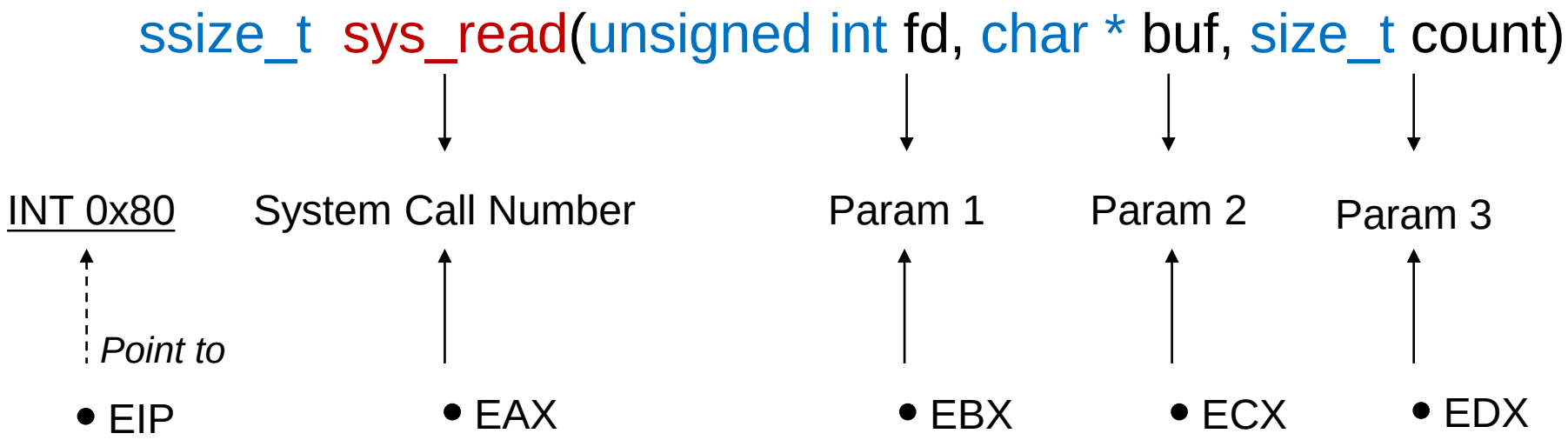


注. Gadget在代码段，地址固定，可以通过逆向工程工具直接获得

面向返回地址编程

- **示例: 基于ROP的实现的任意系统调用:**

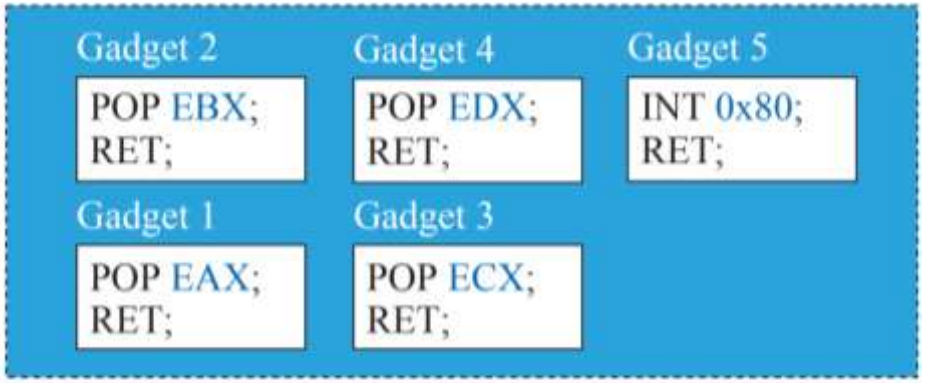
核心思想为组合排列Gadget，构造寄存器为系统调用时的状态如下:
（以sys_read系统调用为例：实现恶意I/O操作）



面向返回地址编程

- **示例: 基于ROP的实现的任意系统调用:**

1. 利用**逆向分析工具**在代码段所搜5个所需的Gadget（如下图所示），并构造右图中的恶意输入



文本段中的 Gadgets

触发栈溢出的恶意输入

面向返回地址编程

- **示例: 基于ROP的实现的任意系统调用调用：**

2. **进行栈区溢出攻击**，将构造的Payload通过标准输入流（或网络流）输入受害进程，发生栈区溢出，覆盖返回地址

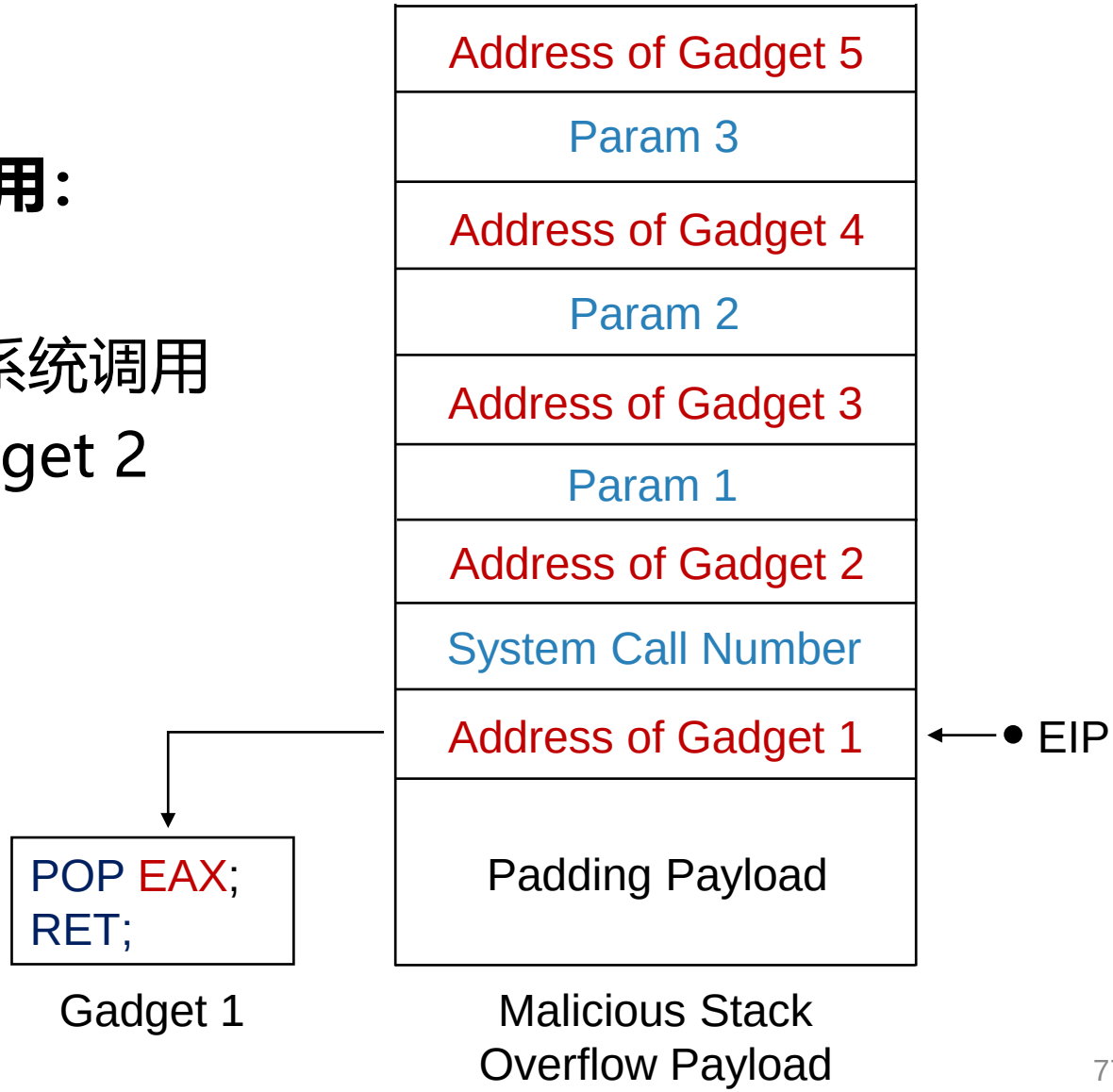
当被调函数返回时EIP将被赋值为第一个Gadget的地址

触发栈溢出的恶意输入

面向返回地址编程

- **示例: 基于ROP的实现的任意系统调用调用：**

3. EIP执行Gadget 1当中的代码，POP将系统调用号赋值给EAX，RET指令执行后EIP指向Gadget 2

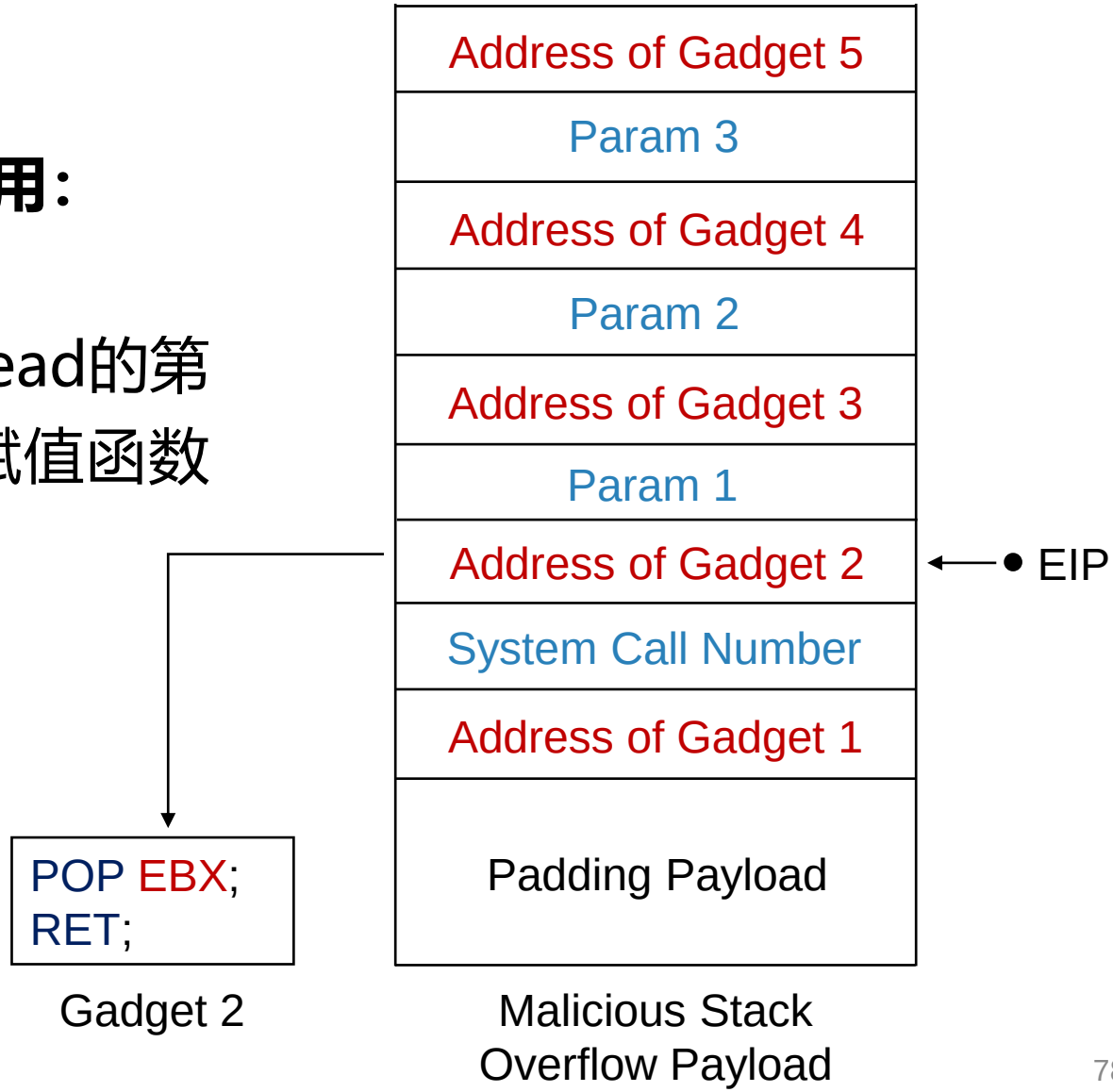


面向返回地址编程

- **示例: 基于ROP的实现的任意系统调用调用：**

4. EIP执行Gadget 2当中的代码，将sys_read的第一个参数：文件描述符赋值至EBX，同理赋值函数的其他参数给寄存器ECX，EDX

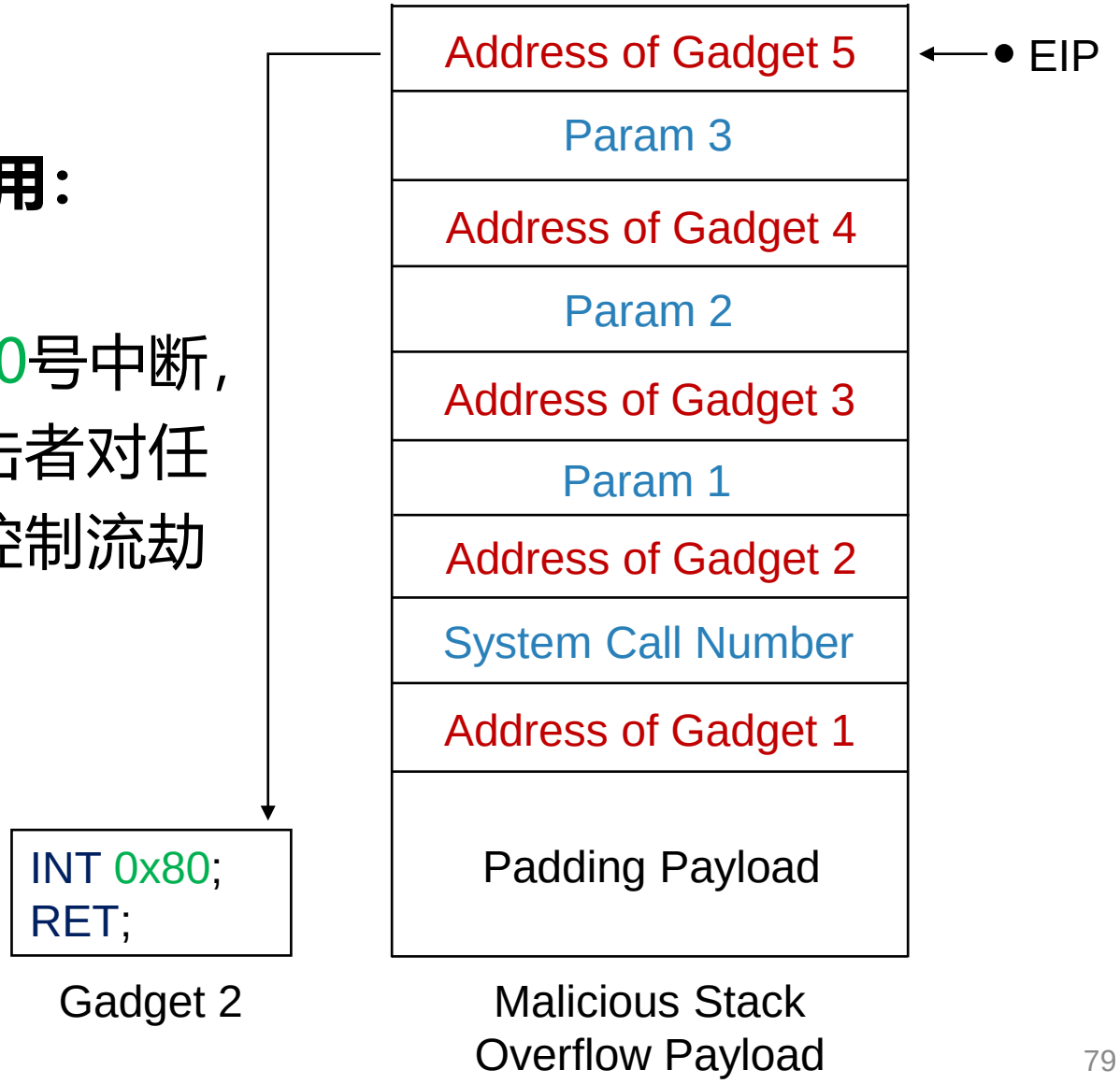
完成全部参数赋值后EIP指向Gadget 5



面向返回地址编程

- **示例: 基于ROP的实现的任意系统调用调用：**

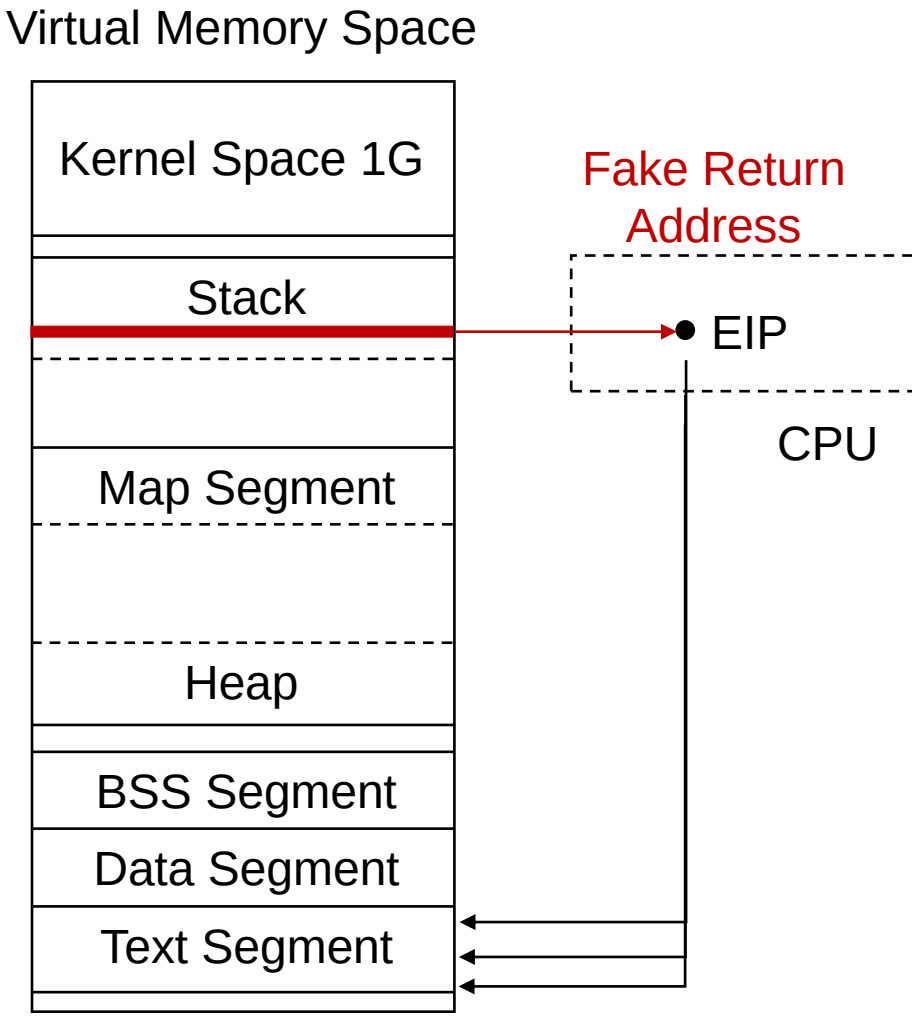
5. EIP执行 Gadget 5 当中代码，触发0x80号中断，将进入内核态进行sys_read系统调用，攻击者对任意指定的文件描述符进行读操作，完成了控制流劫持



面向返回地址编程的优势

- **面向返回地址编程的优势**
- 面向返回地址编程（ROP）**可以绕过NX** 防御机制，因为虚假的返回地址被设置在代码段（Text Segment），代码段是存放进程指令的内存区域，必有执行权限

- 面向返回地址编程（ROP）**可以绕过ASLR 防御机制**，因为目前ASLR的随机性不强，且依赖模块自身的支持



面向返回地址编程总结

对面向返回地址编程（ROP）攻击的讨论与总结：

- ROP的本质是利用程序**代码段的合法指令**，重组一个恶意程序，每一个可利用的指令片段被称作 Gadget，可以说ROP是：a chain-of-gadgets
- ROP可以绕过NX和ASLR防御机制，但对于Stack Canary则需要额外的信息泄露方案才可绕过这一防御机制
- ROP使用的Gadget以RET指令结尾；若Gadget的结尾指令为JMP，则为**面向跳转地址编程**（Jump-Oriented Programming，JOP），原理与ROP类似

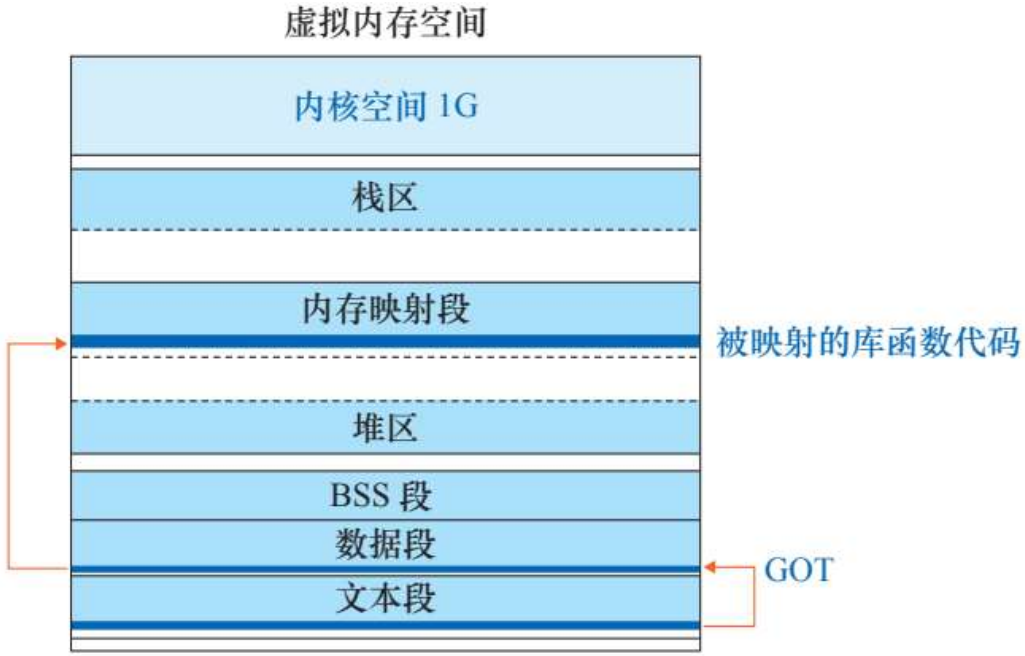
第3节 高级控制流劫持方案

- 3.1 进程执行的更多细节
- 3.2 面向返回地址编程
- **3.3 全局偏置表劫持**
- 3.4 虚假vtable劫持

全局偏移表与程序链接表

- 为了使进程可以找到内存中的动态链接库，需要维护位于**数据段**的**全局偏移表**（Global Offset Table，GOT）和位于**代码段**的**程序连接表**（Procedure Linkage Table，PLT）

- 程序使用**CALL指令**调用共享库函数；其调用地址为**PLT表地址**，而后由PLT表**跳转索引GOT表**，GOT表项指向内存映射段，也就是位于动态链接库的库函数



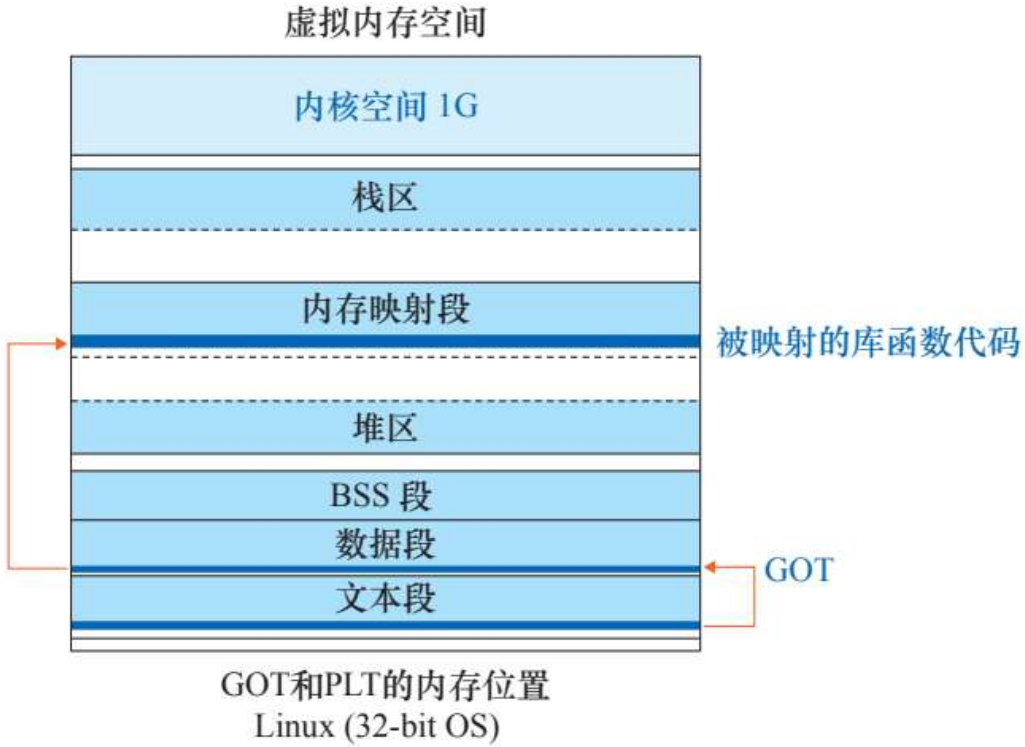
GOT和PLT的内存位置
Linux (32-bit OS)

全局偏移表与程序链接表

- PLT表在运行前确定，且在程序运行过程中不可修改（Text Segment 不可写）

GOT表根据一套“惰性的”共享库函数加载机制，GOT表项在库函数的首次调用时确定，指向正确的内存映射段位置

- **动态链接器**将完成共享库在的映射，并为GOT确定表项



GOT和PLT的内存位置
Linux (32-bit OS)

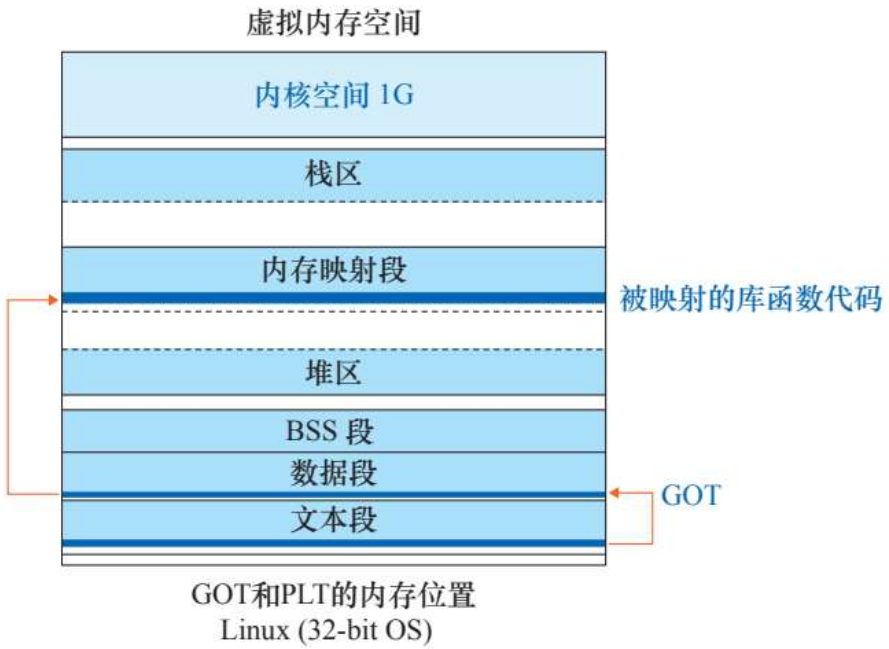
注. 动态链接器为最早被加载的动态链接库

全局偏移表与程序链接表

- PLT表不直接映射共享库代码位置的原因有二：

1. ASLR将随机浮动共享库的基地址，导致共享库的位置无法被硬编码

2. 并非动态链接库当中的所有库函数都需要被映射（降低内存开销）

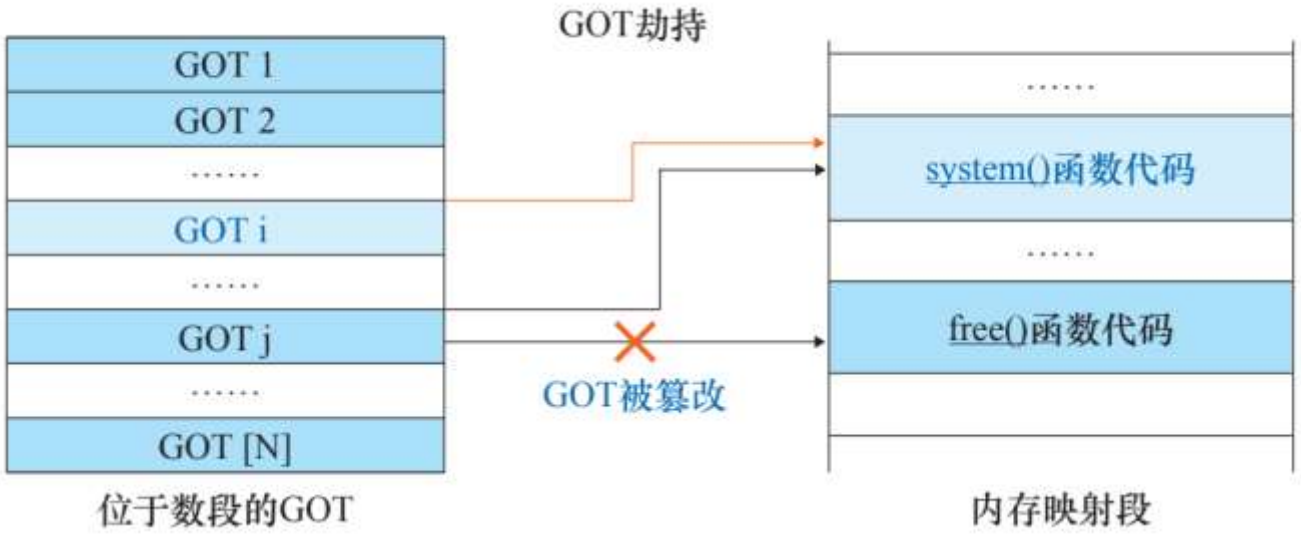


GOT和PLT的内存位置
Linux (32-bit OS)

全局偏移表劫持

- GOT Hijacking (全局偏置表劫持)攻击的本质是恶意篡改GOT表，使进程调用攻击者指定的库函数，实现控制流劫持

如图，当GOT表被修改后，当攻击者调用free函数时，实际上将调用system函数



全局偏移表劫持的优势

- **全局偏置表劫持的优势：**

- **GOT Hijacking攻击可以绕过NX保护机制。**因为装载共享库函数的页上必须有可执行权限；且GOT表位于数据段，数据段可读可写

- **GOT Hijacking攻击可以绕过ASLR保护机制。**因为ASLR无法随机化代码段的位置，攻击者仍然可以通过PLT表恶意读取GOT表项目，而后得到动态库当中函数的地址

全局偏移表劫持步骤

- GOT Hijacking可以分为大致如下的几个步骤：

1. 攻击者通过读PLT确定要被修改的受害GOT表项的地址
2. 攻击者读取这一GOT表项，得到任一库函数的内存映射段地址
3. 攻击者将得到的地址加一个偏置，得到希望被恶意调用的函数的内存映射段地址
4. 攻击者将这一地址写入定位好的受害GOT表项当中

其中，恶意读写GOT表项既可以通过基于栈溢出的ROP来实现，也可以通过基于堆溢出的任意位置读写来实现

全局偏移表劫持总结

- **GOT Hijacking的总结：**
- GOT Hijacking本质上是一种修改GOT表项来实现的控制流劫持；这种攻击方案要依赖于栈区溢出等基础攻击方式才可以实现
- 这种攻击方案可以绕过NX与ASLR防御机制
- 目前已有RELRO（read only relocation）机制，可将GOT表项映射到只读区域上，一定程度上预防了对GOT表的攻击

第3节 高级控制流劫持方案

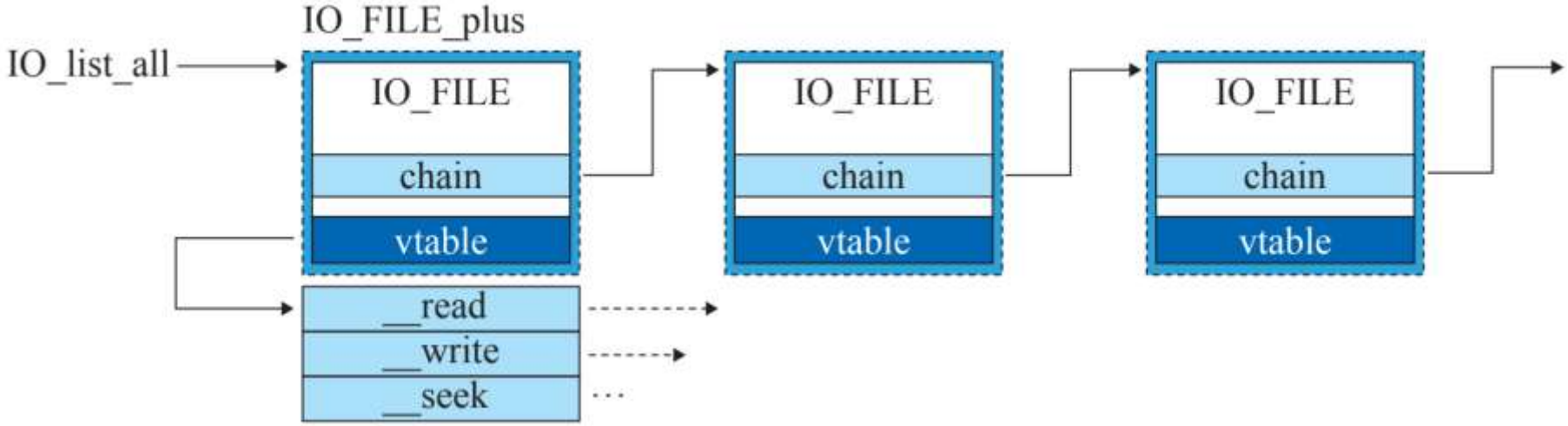
- 3.1 进程执行的更多细节
- 3.2 面向返回地址编程
- 3.3 全局偏置表劫持
- **3.4 虚假vtable劫持**

文件系统角度的安全威胁

- **Linux文件系统维护的元数据：**

在 Linux 系统的标准文件I/O库中 IO_FILE 结构用于描述文件；该结构在程序执行 fopen 等函数时会进行创建，并**分配在堆区**中

IO_FILE 结构外层包裹 IO_FILE_plus 结构，并会通过 chain 字段彼此连接形成一个链表，链表头部用全局变量 IO_list_all 表示



针对文件系统攻击的本质是：恶意操纵文件系统的管理数据结构

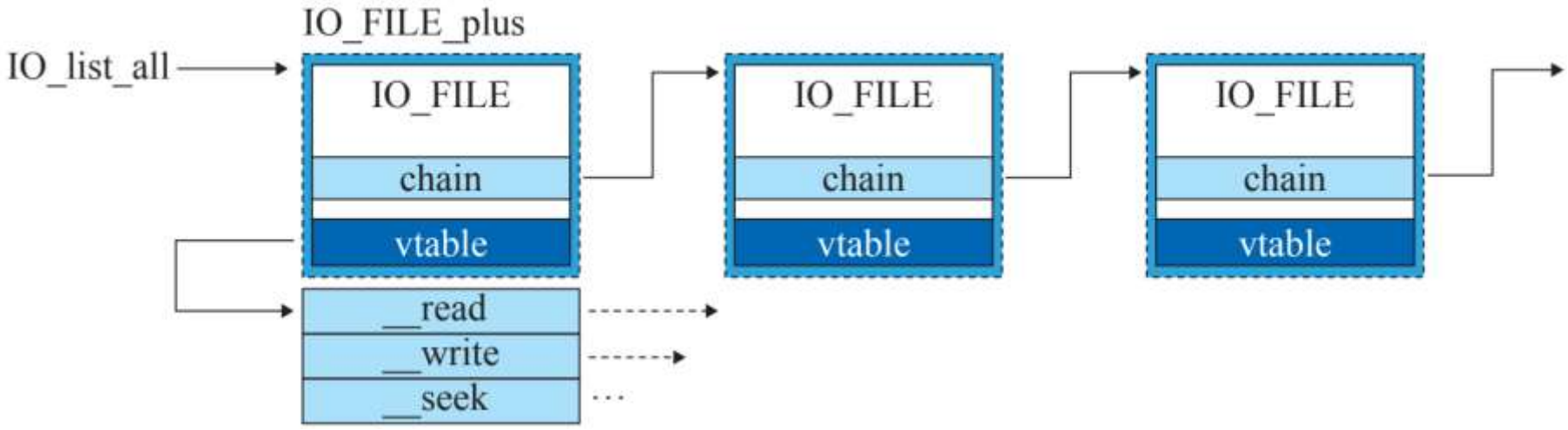
虚假vtable劫持攻击

- **虚假vtable劫持攻击 （Fake vtable Hijacking）**

Linux 中的文件 I/O 操作函数都需要经过 IO_FILE 结构进行处理，尤其是结构中的**vtable**字段，大部分函数会取出vtable指向的一系列**基础文件操作函数**进行调用

虚假Vtable劫持攻击的核心是：篡改文件管理数据结构中的vtable字段

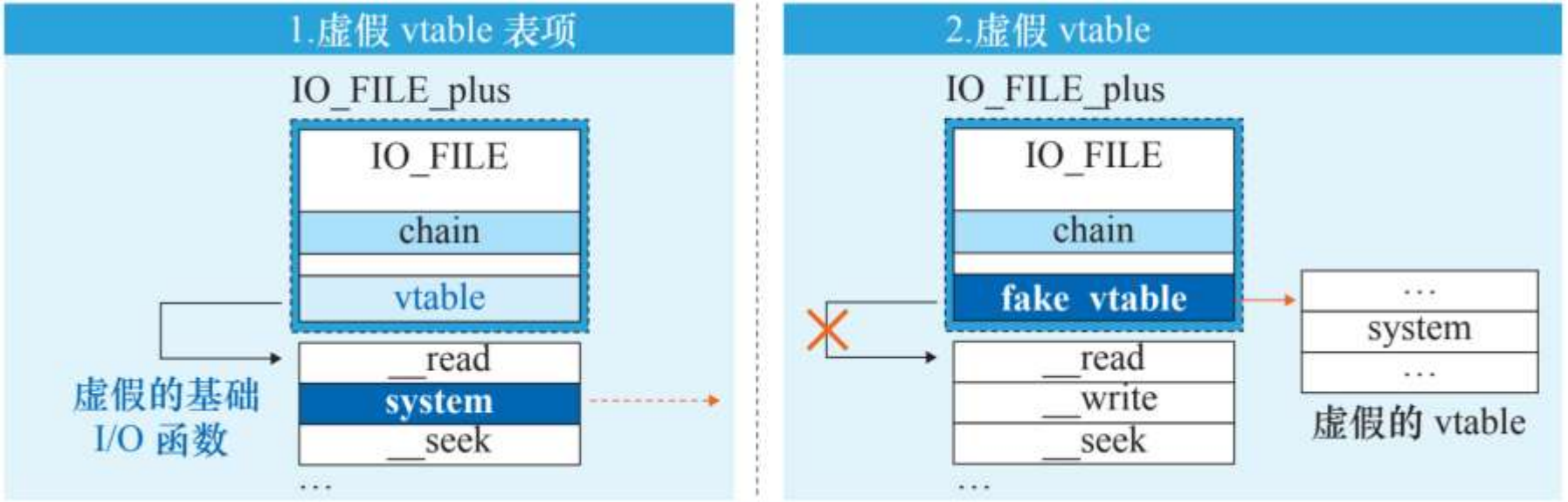
通过把vtable指向攻击者控制的内存，并在其中布置函数指针



虚假vtable劫持攻击

- **虚假vtable劫持攻击，2种具体实现：**

1. 直接改写vtable指向的函数指针，可通过构造**堆块覆盖**完成
2. 覆盖vtable字段，使其指向攻击者控制的内存，然后在其中布置函数指针



第4节 高级操作系统保护方案

- **4.1 控制流完整性保护**
- 4.2 指针完整性保护
- 4.3 信息流控制
- 4.4 I/O子系统保护

控制流完整性保护

- 在DEP ASLR Canary等基础内存保护技术陆续提出以后，用于绕过这些防御机制的攻击手段也随之而来

- 2005年ACM CCS发表了一篇名为《Control-Flow Integrity》的文章，正式提出了 **控制流完整性 CFI** 的概念

控制流完整性保护

- 控制流完整性保护（CFI）依赖于程序的控制流图

控制流图(Control Flow Graph, CFG) 是一个程序的抽象表现；是用在编译器中的一个抽象数据结构，代表了一个程序执行过程中会遍历到的所有路径；它用图的形式表示执行过程内所有基本块执行的可能顺序

Frances E. Allen于1970年提出控制流图的概念
控制流图成为了编译器优化和静态分析的重要工具

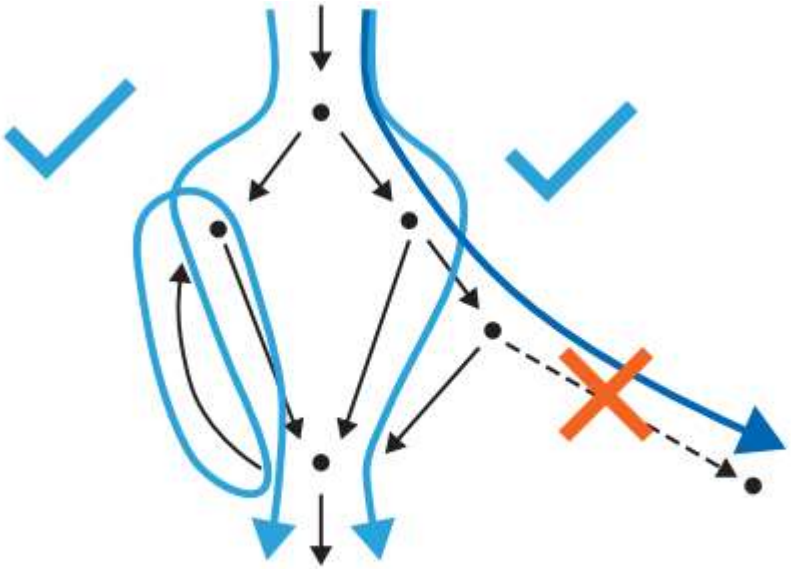
控制流完整性保护

- CFI防御机制的核心思想是限制程序运行中的控制流转移，使其始终处于原有的控制流图所限定的范围

- 主要分为两个阶段：

1. 通过二进制或者源代码程序分析得到控制流图(CFG)，获取转移指令目标地址的列表

2. 运行时检验转移指令的目标地址是否与列表中的地址相对应；控制流劫持会违背原有的控制流图，CFI 则可以检验并阻止这种行为



控制流图

控制流完整性保护

- **对于CFI的一系列改进：**

原始的CFI机制是对所有的间接转移指令进行检查，确保其只能跳转到预定的目标地址，但这样的保护方案开销过大

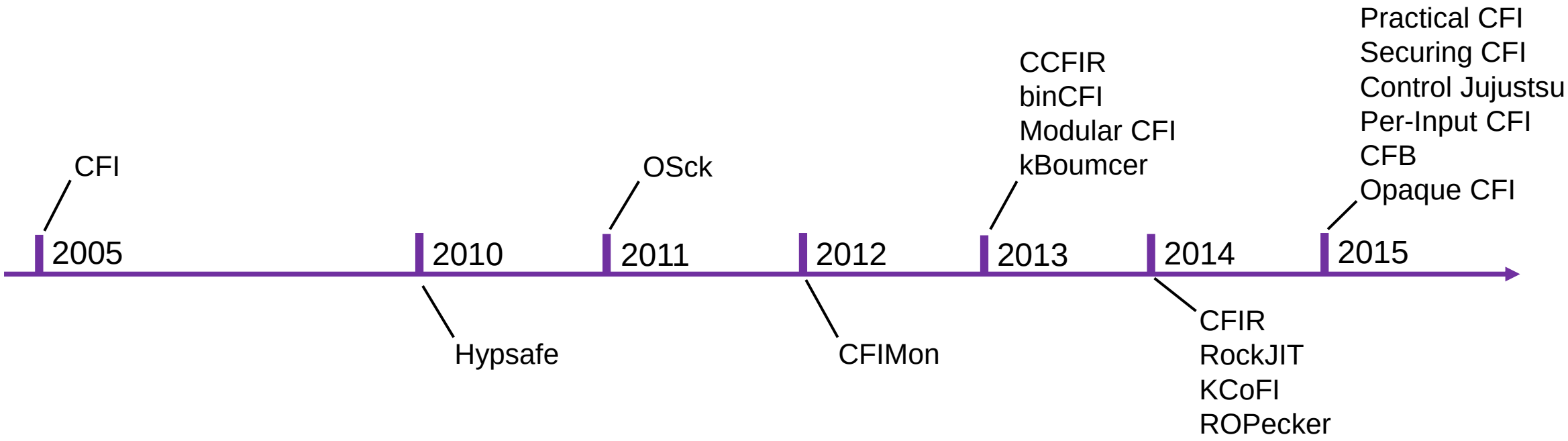
✓ Martín Abadi, et al.[1] 改进的CFI中CFG的构建过程**只考虑将可能受到攻击的间接调用、间接跳转和 RET 指令**，以约束开销

✓ Chao Zhang, et al.[2] 在2013年又提出了CFI 的**低精确度版本**CCFIR；CCFIR 将目标集合划分为三类，分类处理将低开销。间接调用的目标地址被归为一类，RET 指令的目标地址被归为两类，另类是敏感库函数（比如libc中的system函数），最后一类是普通函数

1. Martín Abadi, et al. "Control-flow integrity principles, implementations, and applications." ACM Transactions on Information and System Security (TISSEC) 13.1 (2009): 1-40.
2. Chao Zhang, et al. "Practical control flow integrity and randomization for binary executables." 2013 IEEE Symposium on Security and Privacy. IEEE, 2013.

控制流完整性保护

- 目前CFI方案的平均情况下，额外开销为常规执行的**2-5**倍，距离真实部署仍然存在比较大的距离
- 目前已经提出了大量的CFI的方案，但其中部分方案存在安全问题而失效，或者无法约束开销而彻底不可用



第4节 高级操作系统保护方案

- 4.1 控制流完整性保护
- **4.2 指针完整性保护**
- 4.3 信息流控制
- 4.4 I/O子系统保护

指针完整性保护

- **拓展性内容：指针完整性保护CPI**

Volodymyr K, Laszlo S, et al. Code-pointer integrity. OSDI 2014.

与CFI提出动机相同：CPI的提出动机仍然是因为ASLR和DEP等内存保护无法防御ROP等进程控制流劫持方案

CPI希望解决CFI的高开销问题，其核心在于控制和约束指针的指向位置

关于CPI**是否可用**的性能评价性研究请参考脚注[1]

1. Isaac Evans, et al. "Missing the point (er): On the effectiveness of code pointer integrity." 2015 IEEE Symposium on Security and Privacy. IEEE, 2015.

第4节 高级操作系统保护方案

- 4.1 控制流完整性保护
- 4.2 指针完整性保护
- **4.3 信息流控制**
- 4.4 I/O子系统保护

信息流控制

- **信息流控制**，Information Flow Control（IFC）是一种操作系统**访问权限控制方案（Access Control）**

操作系统可以利用IFC控制进程访问数据的能力[1]
分布式操作系统可以使用IFC控制节点间信息交换的能力[2]

- 即便程控制流被劫持，IFC可以保证受害进程无法具备正常执行之外的能力；例如，访问文件系统中的密钥对，调用操作系统的网络服务

1. Nickolai Zeldovich, et al. "Making in formation flow explicit in HiStar." In Proc. of the 7th OSDI, 2006.
2. Nickolai Zeldovich, et al. "Securing Distributed Systems with Information Flow Control. " In NSDI 2008.

信息流控制

- **信息流控制的三要素：**

1. 约束：调用服务或访问数据需要满足什么样的要求，需要有什么权限才可访问
   > IFC中以Label的形式体现
2. 权限：标志进程具有哪些被赋予的权限，IFC中权限可以动态获取
   > IFC中以Ownership[1] 的形式体现
3. 属性：权限和约束当中包含的单元，是访问能力的元数据形式
   > IFC中以Categories的形式体现

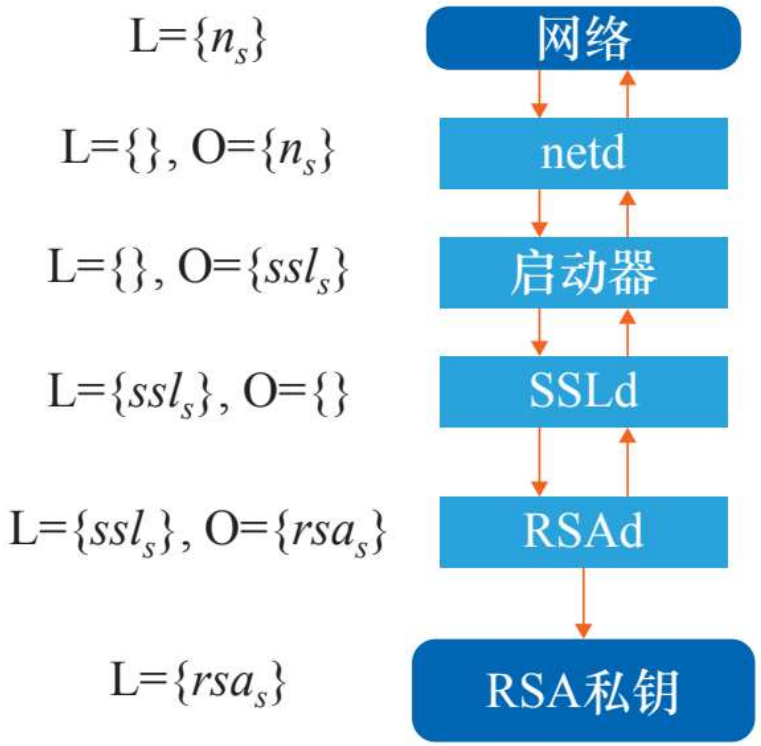
信息流可流动的条件是：
进程的**权限**满足**约束**对其中全部**属性**的要求

1. 部分文献亦称之为privilege，与Ownership完全等价

信息流控制举例

- **IFC举例：**

信息流控制对SSL/TLS链接建立的约束：



符号	含义
网络	资源：网络设备
RSA私钥	资源：RSA私钥文件
netd	进程：链接响应进程
启动器	进程：验证请求进程
SSLd	进程：SSL协议进程
RSAd	进程：RSA服务进程
L	标签
O	权限
n_s	属性：可使用网络
ssl_s	属性：可调用SSL协议接口
rsa_s	属性：可访问私钥

对信息流控制的评论

- **信息流控制**解决的根本问题是**访问控制**
- **信息流控制**缺陷是：IFC是否生效严重依赖于配置的正确性；IFC的三要素：权限、属性、约束都需要具体问题具体分析得到

适用于现代操作系统的IFC在2007年被提出[1]
因为配置的复杂性，之后的十余年当中IFC并没有被广泛普及

- IFC借助属性、标签、构建图结构的方法启发了权限管理的后续工作：例如，User Account Access Graphs[2] 应用了类似的思想于账户权限管理

1. Maxwell Krohn, et al. "Information flow control for standard OS abstractions." In Proc. of the 21st SOSP, 2007.
2. Sven Hammann, et al. "User Account Access Graphs." In Proc. of CCS, 2019.

第4节 高级操作系统保护方案

- 4.1 控制流完整性保护
- 4.2 指针完整性保护
- 4.3 信息流控制
- **4.4 I/O子系统保护**

I/O 子系统保护

I/O 子系统是操作系统的重要组成部分

- I/O 子系统是操作系统一个**重要而庞大**的模块，实现了网络协议栈与一系列复杂的人机交互功能；有文献证实，Linux 中I/O子系统占据了超过**70%**的代码量

- 在内核当中实现有USB (Universal Serial Bus)，蓝牙 (BlueTooth) 等一系列外设I/O交互协议；
- 也包含了完整的网络协议栈，例如TCP/IP协议栈：IPv4/6，ICMP，TCP，UDP等一系列协议

针对I/O子系统的攻击的本质是：发掘通讯协议中的漏洞

I/O 子系统保护

针对外设 I/O 系统的攻击方案示例

- 针对USB协议，BadUSB 攻击允许外设执行其额外的服务功能，例如闪存可以向操作系统注册键盘的输入输出功能

- 对于蓝牙，BlueBrone 攻击伪造恶意的蓝牙通讯报文，实现了针对于操作系统内核蓝牙协议的越界访问攻击；类似的有 BleedingBit 攻击方案

- 对于NFC (Near Field Communication) 也有诸多类似的攻击方案

外设I/O因为其功能复杂多样，一直是操作系统安全问题的“重灾区”

I/O 子系统保护

针对外设 I/O 系统的保护方案示例

- 随着近年来恶意I/O外设引发的安全事件数量逐步上升，产生了一系列保护外设I/O协议的方案：

› 例如USBFirwall[1]，更具固定规则检查USB协议包以抵御恶意的外设
› 类似地USBFLITER[2] 可以使用用户自定义的I/O协议过滤规则，抵御恶意I/O协议报文
› 在2019年Tian et al.[3] 借助Linux的eBPF机制实现了LBM，是一个协议无关的外设I/O协议数据包审查框架

1. Peter C. Johnson, et al. "Protecting against malicious bits on the wire: Automatically generating a USB protocol parser for a production kernel." Proceedings of the 33rd Annual Computer Security Applications Conference. 2017.
2. Dave Jing Tian, et al. "Making USB Great Again with USBFILTER." 25th USENIX Security Symposium, 2016.
3. Dave Jing Tian, et al. "LBM: a security framework for peripherals within the linux kernel." 2019 IEEE Symposium on Security and Privacy, 2019.

I/O 子系统保护

针对网络 I/O 的攻击与保护方案

- 需要时刻牢记，**内核协议栈是操作系统的组成部分之一**，因而面向网络当中端节点的攻击方案的本质均是面向**操作系统网络I/O子系统**的攻击

- 例如，TCP侧信道攻击发掘内核协议栈当中的设计或者实现上的缺陷，是对操作系统网络I/O的攻击方案，需要借助**网络入侵检测系统**进行保护

- **关于网络协议栈的攻击/保护方案，将在第八章详细展开**

本节相关的前沿研究工作

- SoK: Shining Light on Shadow Stacks[1]
  Shadow Stack 防御机制的综述文章，Shadow Stack 的核心是**构建内存栈区拷贝**，以防止各类溢出攻击的

- Efficient Protection of Path-Sensitive Control Security[2]
  较为有效的强化版CFI，原始的CFI有很多问题，性能低有时还引入新安全问题

- CFI/CPI/Shadow Stack，要求**保护方案的元数据不可篡改**：

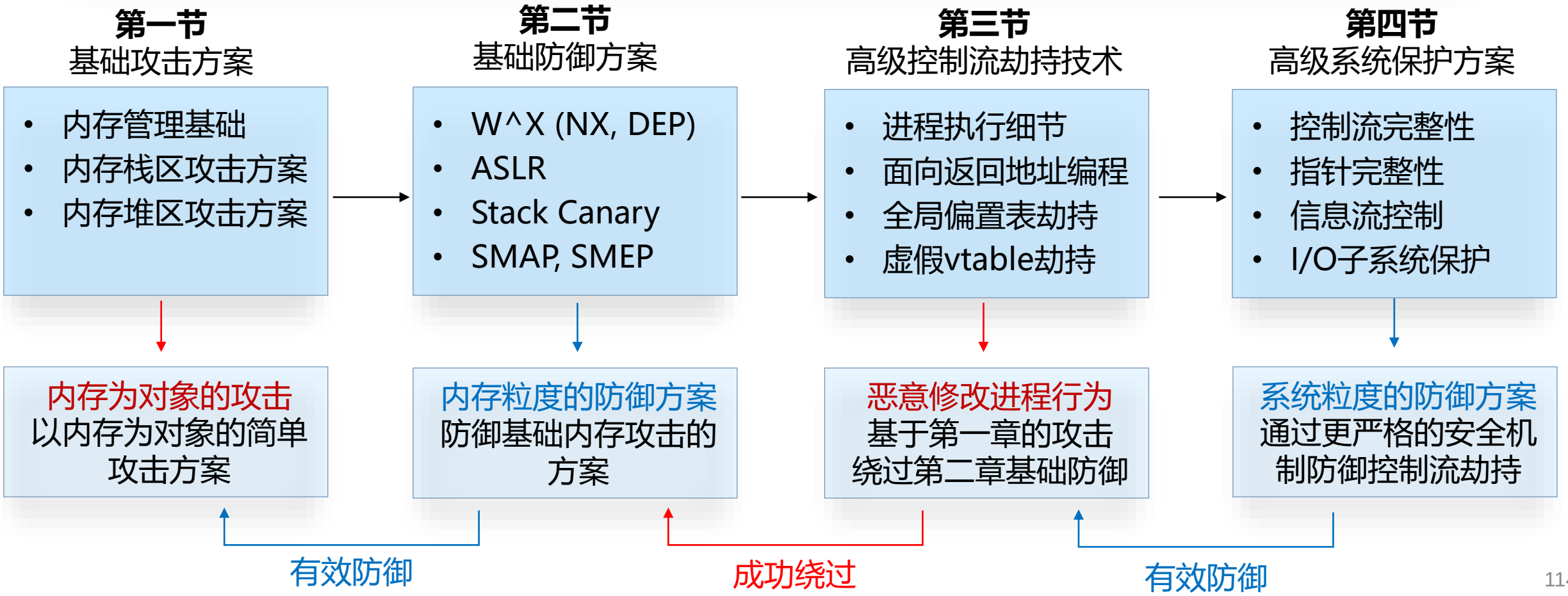
1. 基于软件的方案：SafeHidden: An Efficient and Secure Information Hiding Technique Using Re-randomization[3]
2. 基于拓展ISA的方案：IMIX: In-Process Memory Isolation Extension[4]

1. Nathan Burow, et al. "SoK: Shining light on shadow stacks." 2019 IEEE Symposium on Security and Privacy, 2019.
2. Ren Ding, et al. "Efficient protection of path-sensitive control security." 26th USENIX Security Symposium, 2017
3. Zhe Wang, et al. "Safehidden: an efficient and secure information hiding technique using re-randomization." 28th USENIX Security Symposium, 2019.
4. Tommaso Frassetto, et al. "IMIX: In-Process Memory Isolation EXtension." 27th USENIX Security Symposium, 2018.

第5节 总结和展望

总结

本章当中，我们首先回顾了最早的操作系统安全问题，Morris Worm；并分析了 SQL Slammers 侵染SQL Server的全过程，之后由浅入深地介绍了操作系统下一系列攻击与防御方案



展望

宏观来看操作系统安全研究发展趋势有二：

- 其一，对于操作系统的防御与攻击方案**与新型硬件特性**加以结合，例如：

Gras, et al.[1] 在2017年提出了一种利用硬件缓存侧信道去除ASLR随机化的方法
Frassetto, et al.[2] 通过拓展指令集，实现了内存隔离，防止了非法内存访问的发生

借助SGX，MPK等新型硬件安全功能，借助硬件虚拟化基础设施实现新的系统安全保障机制研究目前是一个**活跃的研究分支**

1. Ben Gras, et al. "ASLR on the Line: Practical Cache Attacks on the MMU." NDSS. Vol. 17. 2017.
2. Tommaso Frassetto, et al. "IMIX: In-Process Memory Isolation EXtension." 27th USENIX Security Symposium, 2018.

展望

宏观来看操作系统安全研究发展趋势有二：

- 其二，操作系统环境下的**漏洞的自动化挖掘**

Wang, et al.[1] 实现了一种对于操作系统缺乏二次检查漏洞的全自动化挖掘方法，其使用的传统数据流与控制流分析，类似的工作还有很多

Jia, et al.[2] 针对于内核驱动当中的未初始化漏洞进行了分析

Eckert, et al.[3] 利用模型验证（Model Checking）方法实现了对于堆管理器的安全性验证

1. Wenwen Wang, et al. “Check it again: Detecting lacking-recheck bugs in os kernels." in Proceedings of CCS, 2018.
2. Xiangkun Jia, et al. "Towards efficient heap overflow discovery." 26th USENIX Security Symposium, 2017.
3. Moritz Eckert, et al. "Heaphopper: Bringing bounded model checking to heap implementation security." 27th USENIX Security Symposium, 2018.

展望

宏观来看操作系统安全研究发展趋势有二：

- 其二，操作系统**漏洞的自动化挖掘**

目前系统相关漏洞自动化挖掘是极其活跃的研究分支

然而，其中具有显著成效的方法均以传统的**静态源代码分析**为主，具有更好漏洞挖掘效果的**动态分析方案**（例如模糊测试，Fuzzing）在应用到操作系统这类高度复杂软件系统时仍然存在各种各样的局限性

目前来看，精准全面的操纵系统漏洞的自动化挖掘仍然是一个“美好的理想”