操作系统安全

为什么存在安全问题 P5-8

1. 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
2. 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

攻击前提 P9

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

攻击目标 P10

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**：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。

操作系统基础攻击方案

操作系统内存基础 P14

- 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

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

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

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（虚拟地址）。

栈区内存的作用 P19

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。

密切相关的寄存器 P20

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。

正常的函数调用流程 P22~29

栈区溢出攻击 P30

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。

1. 返回至溢出数据 P32
2. 返回至库函数 P35

总结 P37

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。

基础堆区攻击 P38

正常工作的堆管理器 P39 ~ 43

堆管理器处于用户程序与内核中间地位，主要做以下工作：

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

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。

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。堆管理器的根本区别在于堆管理算法和管理元数据。

堆溢出攻击 P44

面向堆区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。

1. P45 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
2. P46~49 构造堆块重叠：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。
3. P50 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
4. Use-After-Free 51是进程由于实现上的错误，使用已被释放的堆区内存。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。
5. Double-Free 52 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。
6. Heap Over-Read 53 直接越界读出堆区数据，造成信息泄露
7. Heap Spray 堆喷：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。

操作系统基础防御方案 P55

P56 W^X机制：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。

P57 ASLR：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。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。

P60 Stack Canary 金丝雀：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。

P61：内存隔离技术：SMAP/SMEP 和 W^X 均需要处理器硬件的支持。

防御技术的缺陷

1. 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
2. 对于ASLR已有去随机化方案，泄露内存分布信息
3. ROP等进程控制流劫持方案亦可以绕过ASLR、NX的保护机制

高级控制流劫持方案 P64

进程执行的更多细节 P65

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）。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。

共享库机制 P67

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）

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。

P71 面向返回地址编程 ROP Gadget

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。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。

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。

ROP 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。

P82 全局偏置表劫持GOT Hijacking

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。

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。

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 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。

P88 ~ 89具体步骤与 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

P90 虚假vtable劫持Fake vtable Hijacking

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。

P93 两种具体实现

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

高级操作系统基础防御方案 P94

P95~99 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

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。

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。

指针完整性保护 P101

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。

信息流控制 P103

一种操作系统访问权限控制方案（Access Control），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。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。

I/O子系统保护 P107

针对I/O子系统的攻击的本质是：发掘通讯协议中的漏洞。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”。

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。