基于二进制动态翻译的ROP攻击检测方法研究与实现

[第一章 绪论 3500+ 3](#_Toc161736794)

[1.1 研究背景 3](#_Toc1741051407)

[1.2 ROP攻击及防御发展现状 4](#_Toc1222989815)

[1.2.1 ROP攻击发展现状 4](#_Toc941784397)

[1.2.2 ROP防御发展现状 5](#_Toc1780397546)

[1.3 动态二进制插桩技术 5](#_Toc874921511)

[1.4 本文主要研究内容 7](#_Toc1809830504)

[1.4.1 ROP攻击动态特征的提取 7](#_Toc1664260855)

[1.4.2 ROP攻击检测系统的实现 7](#_Toc1562892083)

[1.5 本文组织结构 7](#_Toc675653039)

[第二章 ROP攻击原理与流程 4000+ 7](#_Toc1726503884)

[2.1 ROP攻击 7](#_Toc1060486138)

[2.2.1 原理 7](#_Toc1061686501)

[2.2.2 攻击流程 8](#_Toc1881241678)

[2.2.3 变种攻击 9](#_Toc2103326390)

[2.2 常见程序漏洞 11](#_Toc1289029302)

[2.2.1 缓冲区溢出漏洞 11](#_Toc1027431185)

[2.2.2 格式化字符串漏洞 12](#_Toc1542511227)

[2.3 辅助攻击手段 14](#_Toc1529253174)

[2.3.1 绕过随机化 14](#_Toc2039231652)

[2.3.2 篡改GOT表 14](#_Toc2118282676)

[2.4 本章小结 16](#_Toc237975304)

[第三章 ROP攻击动态特征 1500+ 17](#_Toc1973898131)

[3.1 指令特征 17](#_Toc497943094)

[3.2 内存特征 17](#_Toc2085237004)

[3.3 本章小结 18](#_Toc102414948)

[第四章 ROP攻击检测方法 4000+ 18](#_Toc1204970965)

[4.1 指令特征检测 1000+ 18](#_Toc868745339)

[4.1.2 call-ret指令数检测 500+ 18](#_Toc1195146598)

[4.1.1 连续gadget检测 500+ 18](#_Toc1319661066)

[4.2 完整性检测 3000+ 18](#_Toc1933681420)

[4.2.1 调用返回控制流完整性检测 2000+ 18](#_Toc1356883392)

[4.2.1 函数指针控制流完整性检测 1000+ 18](#_Toc913228825)

[4.3 本章小结 18](#_Toc1009187587)

[第五章 ROP攻击检测系统实现 8000+ 18](#_Toc151184141)

[5.1 假设 400+ 18](#_Toc546142723)

[5.2 总体设计 1000+ 18](#_Toc1884109099)

[5.3 系统概述(总体实现) 500+ 18](#_Toc1961014646)

[5.4 实现细则 5000+ 19](#_Toc62919930)

[5.4.1 返回地址检测 19](#_Toc1299517534)

[5.4.2影子栈 20](#_Toc489184037)

[5.4.3 阈值检测器 20](#_Toc1789423814)

[5.4.4 call-ret指令计数器 20](#_Toc212520024)

[5.4.5 GOT篡改检测器 20](#_Toc1550870538)

[5.4.6 CPR检测器 20](#_Toc1523181845)

[5.5界面实现 500+ 20](#_Toc168362766)

[5.6 实验与评估 2000+ `](#_Toc692416193)

[5.6.1实验环境 20](#_Toc403129382)

[5.6.2 Ret2libc攻击与检测 20](#_Toc1710873994)

[5.6.3 Rop攻击防御与检测 24](#_Toc74185719)

[5.7 本章小结 35](#_Toc294877386)

[第六章 总结和展望 35](#_Toc1681673022)

[6.1 总结 35](#_Toc312161023)

[6.2 展望 35](#_Toc121291870)

[参考文献 35](#_Toc32132469)

1. **绪论 3500+**
   1. 研究背景

如今，无论计算机技术发展到何种程度，计算机软件安全永远是人们最为关心的话题，相关的研究总在不断地进展和延续。随着操作系统的更新换代，软件自身的安全性不断提升，针对各种类型的攻击，大量防御策略被提出并应用，对软件进行攻击变得越发困难。但是由于操作系统代码量日益增大、复杂度逐步提高，攻击者总能找出系统漏洞，并利用漏洞进行攻击，如图1-1所示，CVE[1]漏洞数量呈现逐年提升的趋势。此外，程序员编程的不规范以及软件安全更新的不及时更是导致软件漏洞被广泛利用。软件漏洞的必然存在，就像一颗定时炸弹，给计算机系统带来了极大的安全隐患。例如勒索病毒WannaCry利用美国国家安全局泄露的危险漏洞“EternalBlue”（永恒之蓝）进行传播，从2018年初到9月中旬，总计对超过200万台终端发起过攻击，攻击次数高达1700万余次，该病毒通过互联网在全球爆发，国内大量高校及企事业单位被攻击。

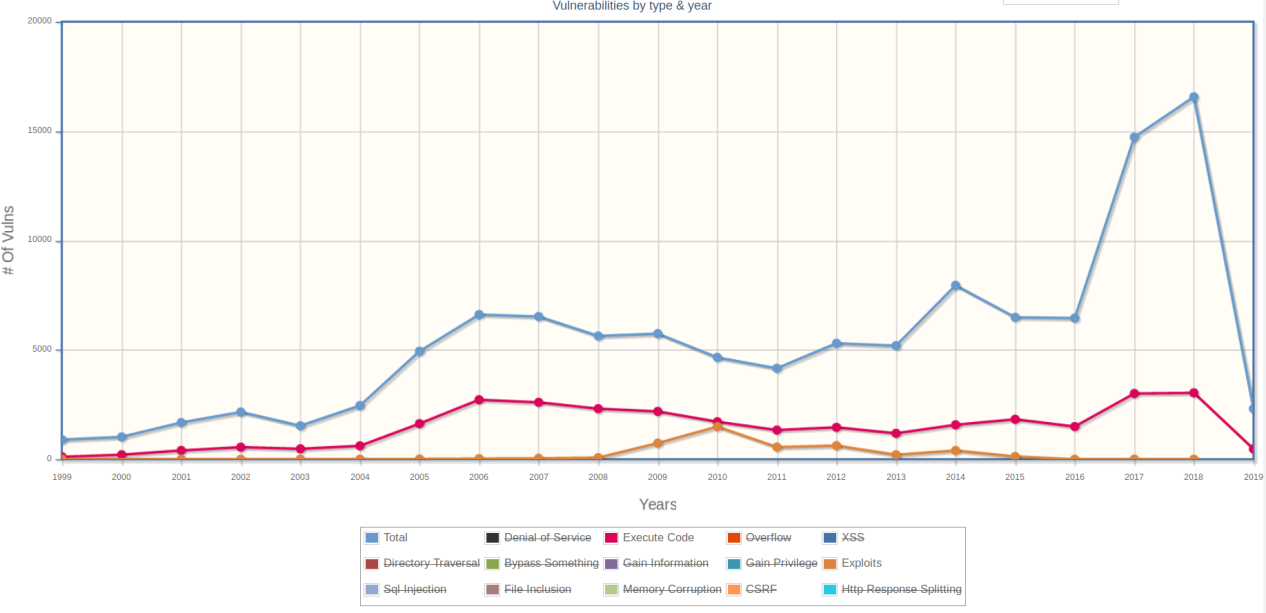


图1-1 近20年CVE漏洞数量

在众多的安全漏洞中，如图1-2，二进制安全占据了半壁江山，其中缓冲区溢出(buffer overflow)是一种常见的漏洞。由于c语言对程序缓冲区边界不进行检测，当攻击者向缓冲区写入过多数据后，缓冲区将溢出。若缓冲区在栈中发生溢出，栈中的函数返回地址将被覆盖，当程序返回时，程序控制流将被攻击者劫持。此外整型溢出、浮点型溢出、格式化字符串、UAF等常见漏洞，均可使攻击者劫持程序控制流。劫持程序控制流，然后执行攻击者构建的攻击代码，是进行攻击的基本流程。早先攻击者将恶意代码注入内存空间，并将控制流劫持至恶意代码，从而达到攻击目的。这些被注入的代码称做shellcode，他们通常是可执行的代码，通过系统调用实现打开shell、更改系统权限、执行程序等恶意行为。

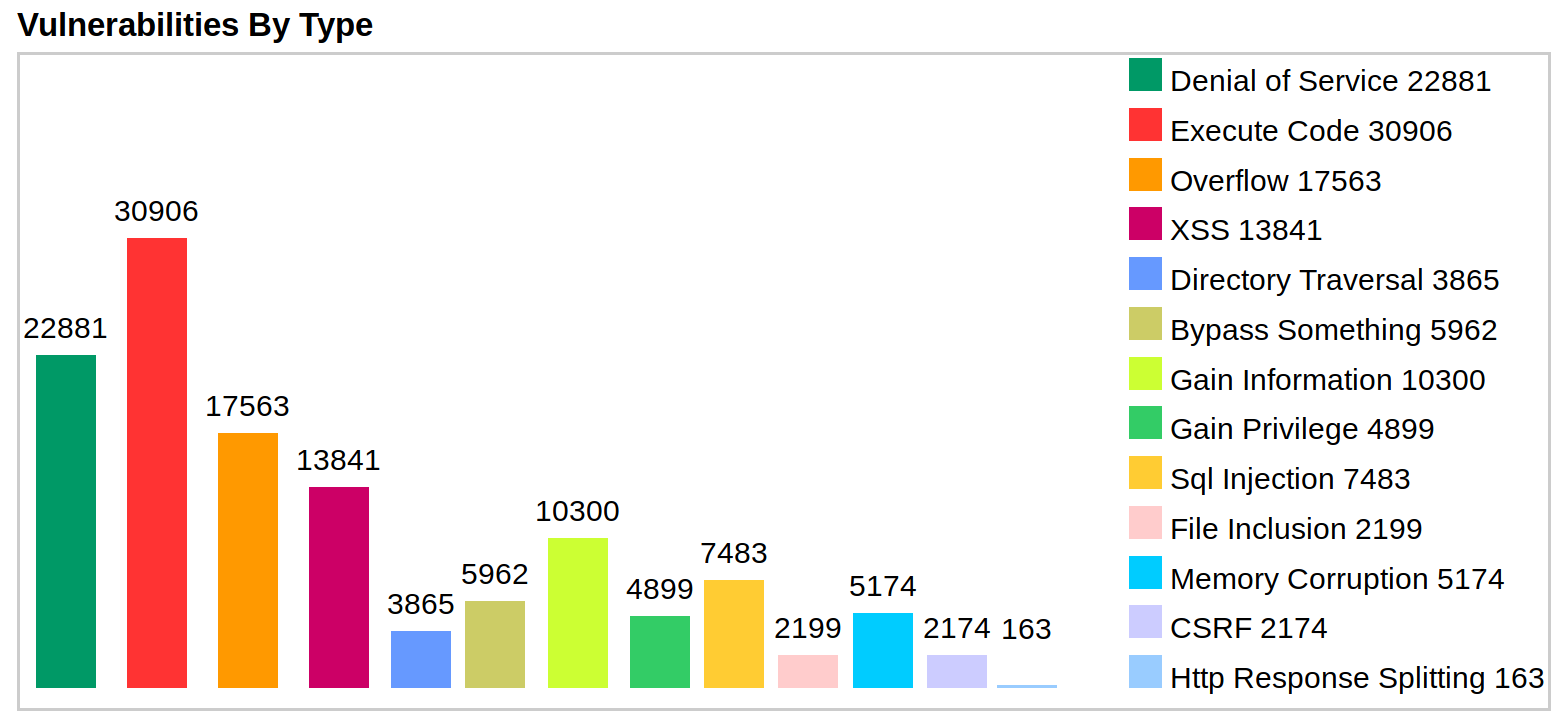


图1-2 近20年各类CVE漏洞数量统计

但是在数据执行保护（DEP）[2]广泛采用后，内存中的所有可写页面均不具有可执行权限。因此，即使攻击者将程序控制流劫持至他们注入的恶意代码，这些代码也无法执行。为了绕过DEP机制实现攻击，攻击者不再注入代码，而是通过利用漏洞进程中的现有的可执行指令来构造恶意行为，即代码复用攻击（CRA）。根据复用的代码类型不同，代码复用攻击主要可分为Return-into-libc和ROP攻击。

* 1. ROP攻击及防御发展现状

1.2.1 ROP攻击发展现状

Return-into-libc[3]是代码复用攻击的一种简单应用，攻击者利用缓冲区溢出漏洞，将栈中的返回地址覆写为某个函数的入口地址，从而使得该函数被执行。复用的函数可以是程序代码段中的函数，也可以是程序所链接的共享库中的函数。攻击者通过修改栈的布局或者寄存器中的数据，构造函数参数，从而实现完整的函数调用，进而实现攻击行为。例如：攻击者复用共享库libc中的system、execve等函数，可执行任意系统命令；复用具有输出功能的函数，如write,puts等，可以获取更多关于程序的信息，比如环境变量、所链接的共享库等；复用具有修改内存功能的函数，如read,malloc等，可以对内存进行任意写操作。

返回导向编程[4]（Return Oriented Programming，ROP）是一种常用的代码复用攻击技术，不同于Return-into-libc，攻击者在返回导向编程时，不执行整个函数，而是执行源自各函数片段中的指令序列。这些指令序列称作gadget，具有如下两个基本特点：1.具有一定的功能，如：寄存器相加、加载某值到内存等；2.以ret指令为结尾。攻击者首先搜索可用gadget，然后将各个gadget链接在一起，从而实现一次完整的攻击（详见2.1节）。ROP最初由Shacham[4]提出并应用于x86平台，随后被拓展到其他体系结构[5,6,7,8]。ROP已被证明可实现图灵完备计算[9]。此外，一些允许攻击者使用ROP自动构造任意恶意程序的工具已被开发出[10,11,12,13]。

在目前使用最广的64位x86平台下，被调函数的参数主要保存在寄存器中，因此在一般情况下，攻击者会将Return-into-libc攻击与ROP攻击结合起来，即：进行Return-into-libc攻击时，通过复用一些gadget（如pop rdi）完成函数的参数配置，然后调用函数，进而达到攻击目的。

除了返回指令以外，调用指令和跳转指令也能够实现程序控制流的转移，因此将返回指令替换为调用指令的Call Oriented Programming(COP)[14]和将返回指令替换为跳转指令的Jump Oriented Programming(JOP)[15]被相继提出。因为传统ROP攻击有着明显的特征，即：使用连续的以ret为结尾的gadget，所以一些防御机制[23,24]识别该特征，对ROP攻击进行防御。上述的变种ROP攻击，不使用或不连续使用以ret指令为结尾的gadget，从而能够绕过这些检测机制。

此外，Snow[16]还提出了实时ROP，攻击者在程序运行时完成gadget的搜索与链接。Bittau提出了BROP[17]，他指出即使不清楚任何目标服务器的信息，也能够根据服务器返回的内容，搜索gadget并构造攻击。

1.2.2 ROP防御发展现状

针对现有的各种代码复用攻击，研究者提出了几类防御方案：

第一类方案是基于内存地址随机化，通过随机化布局，减少攻击者对内存布局的知晓程度。地址空间布局随机化[18]（Address space layout randomization，ASLR）是被广泛应用的一种，ASLR在程序共享库、堆栈加载到内存的过程中，为其基址随机增加一个偏移量，从而使攻击者无法准确获取Return-into-libc攻击所需的libc函数地址以及ROP攻击所需的gadget地址。ASLR由于其方法简单，系统开销小，被广泛应用于各Linux操作系统中。ASLR的变种防御相继被提出，随机化粒度也在不断优化[19,20]。但是粒度的越小，随机化方案的部署就越复杂，系统开销也就越大，因此细粒度的随机化方案没有被广泛应用。

第二类方案是基于程序二进制动态检测技术，通过二进制插桩监测程序运行行为，从而判断程序是否被攻击。例如，Davi[23]等在ROP防御工具Ropdefender中，通过构造影子栈，对调用和返回指令进行动态监测，当函数调用时，将其预期返回地址压入影子栈顶，在函数返回时，将返回地址与栈顶地址作对比，从而阻止非预期的控制流跳转。动态检测技术虽然能获取更多的程序运行时信息，但是也带来了额外的系统开销，使得程序运行放缓。DROP[24]和ROP-Hunt[21]，为了减少额外的系统开销，基于统计学方法，通过设置阈值识别gadget，这些方案虽然性能好，但是不够灵活，防御效果差，容易被攻击者猜到阈值后绕过，且存在误报的可能。

第三类方案是检测程序控制流的完整性，通过监控程序控制流，判断控制流是否按照预期的语义执行，从而防止非预期的代码被复用。Martin[22]等通过构造控制流图(CFG)，确保了语义完整性，但其CFG的生成难以保证准确性。文章[27]中提出了一种基于硬件的完整性保护解决方案。在该方法中，堆栈被分数据栈和专门用于调用和返回的控制栈。CPU采用访问控制机制，不允许用任意数据覆盖控制栈。这有效地防止了ROP攻击，但是，这种方法并不能轻易地移植到常见的，如Intel、AMD架构的复杂指令处理器中。

还有一些防御方案如CFLocking [25]和G-Free [26]，旨在防御所有类型的ROP攻击，但它们需要用户提供程序源代码，对于一般程序用户而言，程序源代码是难以取得的，因此这些防御方案的应用范围受到了限制。

根据上述的ROP防御思想，本文将使用二进制动态插桩框架Pin，提出一种综合方案，应用于ROP攻击的动态防御与检测。

* 1. 动态二进制插桩技术

动态二进制插桩技术（DBI）是一种在程序运行阶段，对程序行为进行动态监测的技术。它能够在不影响程序正常执行的情况下，监测程序在运行过程中的寄存器、内存、指令序列等状态信息。动态二进制插桩技术的优点是可以监测到任何信息，且在程序执行阶段完成，无需改变程序代码和编译过程，缺点是给系统带来了很大的额外开销，将导致被监测的程序运行放缓。

Pin是Intel公司开发的一个动态二进制插桩框架，适用于x86和x64架构，并支持在Linux, OSX, Windows等多个平台下运行。具有易用、高效、可移植性强、鲁棒性强等特点[29]。Pin有两种工作模式：探测(Probe)模式和即时(Just-In-Time, JIT)模式。在即时模式下工作时，Pin能够在处理器执行每条指令前将其拦截。

如图1-3，Pin框架由三部分组成：虚拟机（VM）、代码缓存和插桩API。其中，虚拟机包含即时编译器、模拟单元和调度程序。当程序开始运行时，各条指令先经即时编译器编译和检测，再交由调度程序激活并执行。经过编译的指令存储在代码缓存中，以便在多次调用代码段时降低性能开销。模拟单元用于解释那些无法被直接执行的指令。Pin提供了许多预置的名为Pintool的插桩检测工具（由C/C++语言编写），插桩检测工具通过调用插桩API，与Pin框架进行交互。此外，用户可以根据需要，自定义插桩检测工具。

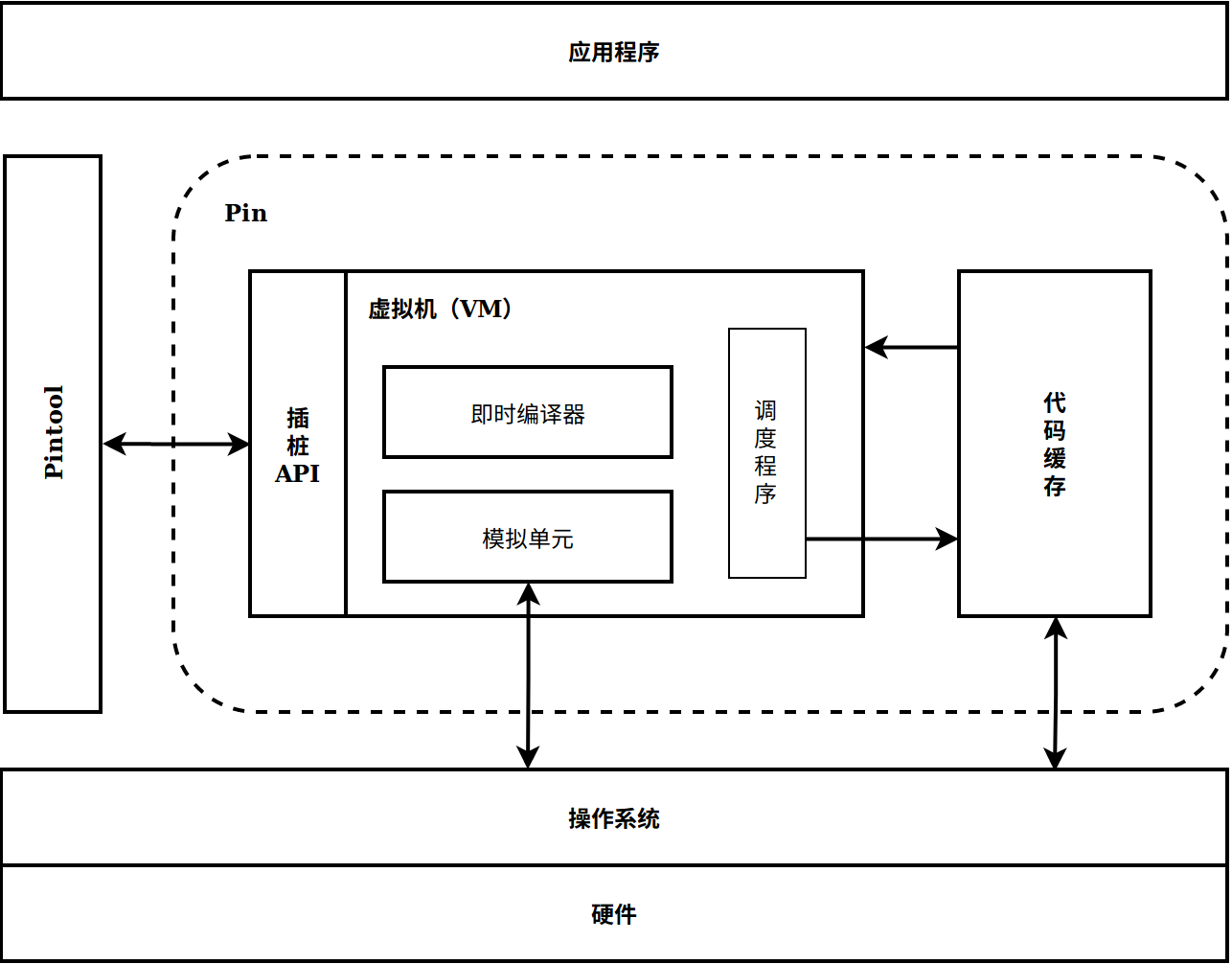


图1-3 Pin结构框架图

Pin提供了指令级别、例程级别、映像级别等多种级别的插桩函数。指令级别的插桩可以获取指令类型、指令操作数、相关寄存器等信息；例程级别的插桩可以获取函数参数、返回地址等信息；映像级别的插桩可以在映像被加载时，获取映像名称、映像地址、库函数等信息。除了获取信息之外，通过插桩API中各种级别的*InsertCall()*函数，还能在指令、基本块、例程执行前或执行后将用户自定义的分析代码插入到原程序代码序列中。使用Pin对二进制程序进行插桩的基本步骤如下：

1. 调用*PIN\_Init()* 初始化Pin框架。
2. 调用*INS\_AddInstrumentFunction()*、*RNT\_AddInstrumentFunction()*或*IMG\_AddInstrumentFunction()*声明指令级别、例程级别或映像级别的插桩函数。在插桩函数中调用一个或多个回调函数*InsertCall()*用于对二进制程序进行检测与分析。
3. 调用*PIN\_AddFiniFunction()* 声明程序退出函数。
4. 调用*PIN\_StartProgram()*启动程序。
   1. 本文主要研究内容
      1. ROP攻击动态特征的提取

由于不同程序存在的漏洞类型与数量不同，攻击者攻击的手段多种多样，由于很难预测攻击者使用的恶意代码，单纯地提取ROP恶意代码的静态特征，如：gadget的大小，gadget链的长度，很难涵盖所有类型的ROP攻击。本文发现，程序在受到ROP攻击时的运行时动态异常，往往具有相同或相似的特征。因此，本文使用动态二进制插桩技术，分析众多存在漏洞的程序实例，跟踪这些程序在受到ROP攻击时的运行时状态，对ROP攻击的动态特征进行提取，共提取了两类动态特征：指令特征和内存特征（详见第三章介绍）。

* + 1. ROP攻击检测系统的实现

本文针对ROP攻击动态特征，提出了多维度的ROP攻击检测方案（详见第四章）。本文通过动态二进制插桩框架Pin提供的各种实用的插桩API，编写插桩检测工具，对第四章中讨论的检测方案进行代码实现，完成了能够识别ROP攻击、JOP攻击和return-into-libc攻击的代码复用攻击检测系统。此外，本文通过Django框架，实现了基于B/S模式的测试界面，用户在测试界面，可以选择开启不同类型的检测方案，系统将根据用户的选择开启相应防护方案，并运行漏洞程序供攻击者攻击，攻击结束后，用户可以在界面中查看攻击检测报告（详见第五章）。

* 1. 本文组织结构

第一章 绪论。介绍本文研究背景、ROP攻击与防御的发展和现状以及动态二进制插桩技术，最后阐述了本文的主要研究内容。

第二章 ROP攻击原理与流程。详细介绍ROP攻击原理、攻击流程以及ROP攻击变种，并对与ROP攻击相关的常见程序漏洞以及辅助攻击手段进行介绍。

第三章 ROP攻击动态特征。分析、讨论、总结ROP攻击的动态特征。

第四章 ROP攻击检测方法。针对第三章中讨论的ROP攻击动态特征，讨论ROP攻击的检测方案。

第五章 ROP攻击检测系统实现。介绍ROP攻击检测系统的总体框架，分析其各部分功能，介绍关键功能的代码实现。最后，利用实验验证检测方案的有效性。

第六章 总结和展望。总结本文工作，分析不足，讨论本文工作值得改进的地方。展望ROP攻击动态检测的未来发展。

1. **ROP攻击原理与流程 4000+**

2.1 ROP攻击

2.1.1 原理

在现代操作系统中，栈被用作函数调用返回的场所。当函数被调用时，操作系统将在栈中分配一块新的内存空间（称作栈帧），供被调用的函数使用。栈帧中存储上一个栈帧的栈基址、函数返回地址、局部变量、函数参数等信息。当函数调用发生时，程序控制流会发生转移，即从原函数转移至被调函数。函数调用返回的流程如下：调用指令执行后，操作系统将被调函数的返回地址（调用指令的下一条指令地址）压入栈顶，然后程序控制流将转移到调用指令的目标地址，即被调函数的入口地址。当函数执行结束后，其末尾的返回指令，将栈顶的返回地址赋值给指令指针寄存器ip（指令指针寄存器存储CPU将要执行的指令的地址），程序控制流于是回到原函数。由于函数调用返回的信息存储在栈中，函数调用的过程也伴随着栈帧的切换。以x86框架为例，如图2-1，栈帧的切换流程如下：调用指令执行后，程序控制流转移至被调函数，被调函数首先将旧的栈基址压入栈中①，然后设置新的栈基址②，并移动栈指针，开辟新的栈空间③，返回指令执行前，将栈指针指向栈基址④，并恢复保存的栈基址⑤。其中①-③为栈帧建立过程，④-⑤为栈帧的销毁过程。通常栈帧的建立-销毁操作在函数调用-返回期间进行。

① push ebp

② mov ebp, esp

③ sub esp, 0x8

...

④ mov esp, ebp

⑤ pop ebp

ret

图2-1 x86框架下的函数栈操作

由于c语言对程序缓冲区边界不进行检测，当攻击者向缓冲区写入过多数据后，缓冲区将溢出。当缓冲区发生溢出后，栈中的返回地址被覆盖，函数返回时指令指针寄存器ip的值将被攻击者篡改，程序控制流由此被劫持。

ROP攻击是将控制流劫持至gadget中的一种代码复用攻击。如图2-2，攻击者将收集到的gadget的地址以及一些必要数据，经过精心编排后写入栈中，覆盖返回地址及其后的区域。通过对栈空间的精心布局，实现一个gadget执行完毕后，通过其末尾的返回指令，使程序控制流跳转至下一个gadget的目的。由此gadget被依次执行，直到达到攻击者目的。

传统的ROP攻击通过gadget末尾的返回指令实现控制流的转移。广义的讲，末尾指令能够实现控制流转移的指令片段，均可以称作gadget。除返回指令以外，调用指令、跳转指令也可以实现控制流转移，由此衍生出了JOP攻击与COP攻击（详见2.2.3）。

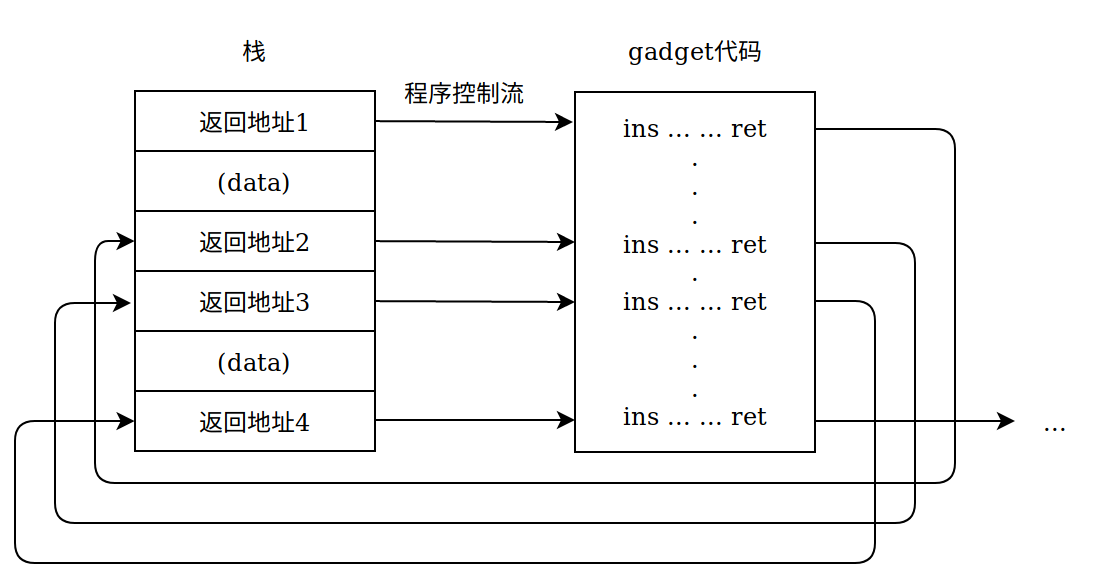


图2-2 ROP攻击流程

2.1.2 攻击流程

首先，进行攻击准备工作，准备工作包括对漏洞程序进行动态分析和静态分析，其中最重要的一步是搜集可用gadget。一般情况下，攻击者利用gadget搜索工具（如：ROPgadget、ropper）在不运行程序的情况下，对漏洞程序进行静态扫描，在程序代码段或者程序所链接的共享库的代码段中搜索可用的gadget。准备工作做完后，攻击者运行程序，触发程序中存在的漏洞，将搜集的gadget地址经过精心编排后写入栈中，并将程序栈中的返回地址覆盖为gadget的地址。如图2-2，攻击者将程序的原返回地址覆盖为返回地址1，并将一些数据以及返回地址2，3，4写入栈中，返回地址1，2，3，4分别指向三个不同的gadget。当程序返回时，程序控制流首先被劫持至第一个gadget中，当第一个gadget完成一定操作后，返回地址2位于栈顶，gadget1末尾的ret指令执行后，程序控制流将转移至下一个gadget。由此，攻击者可以将搜集到的gadget链接起来，进而实现一次完整的攻击。

2.1.3 变种攻击

跳转导向编程[15]（Jump-Oriented Programming, JOP）是ROP攻击的一种变种，它使用寄存器间接跳转指令代替了返回指令。如图2-3，JOP使用调度表(dispatch table)来保存攻击者需要的gadget的地址和一些必要数据，使用调度程序(dispatcher)作虚拟程序计数器，操控程序控制流，将程序控制流在**调度表**中转移。在gadget的末尾，攻击者利用间接跳指令使程序控制流跳回调度程序。随后，调度程序将指针指向下一个gadget。一个简单的调度程序如下：add rdx,8; jmp [rdx]。攻击者进行攻击时，只需要通过利用程序漏洞，将程序控制流劫持至调度程序入口，让调度程序接管程序控制流，便可启动一次JOP攻击。

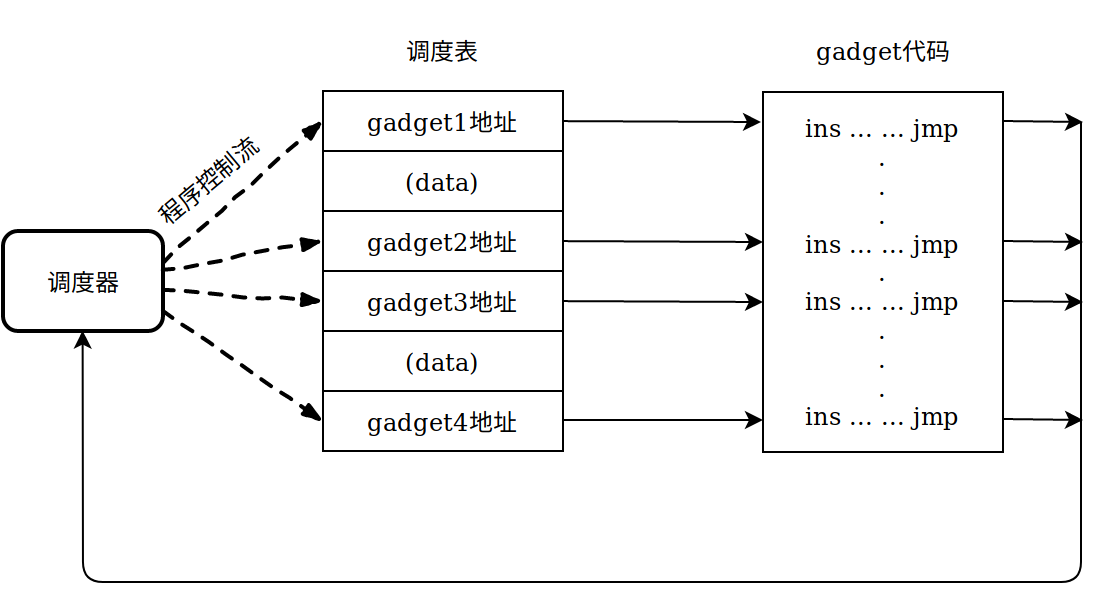
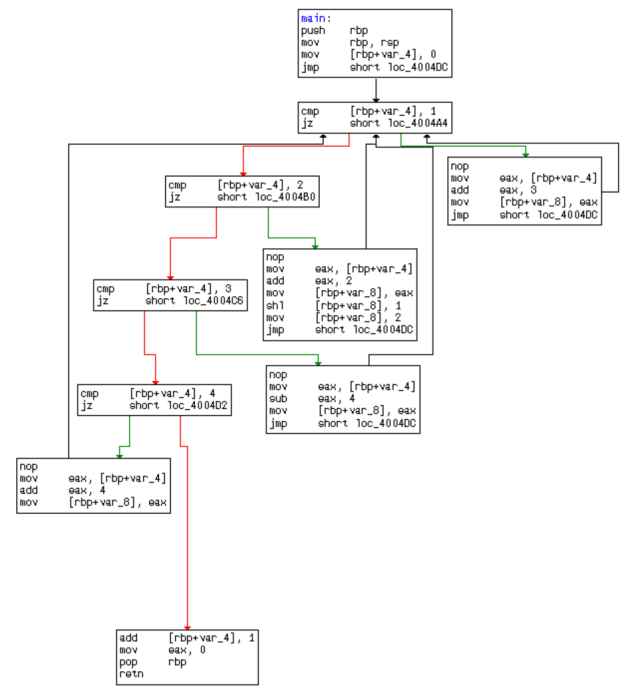
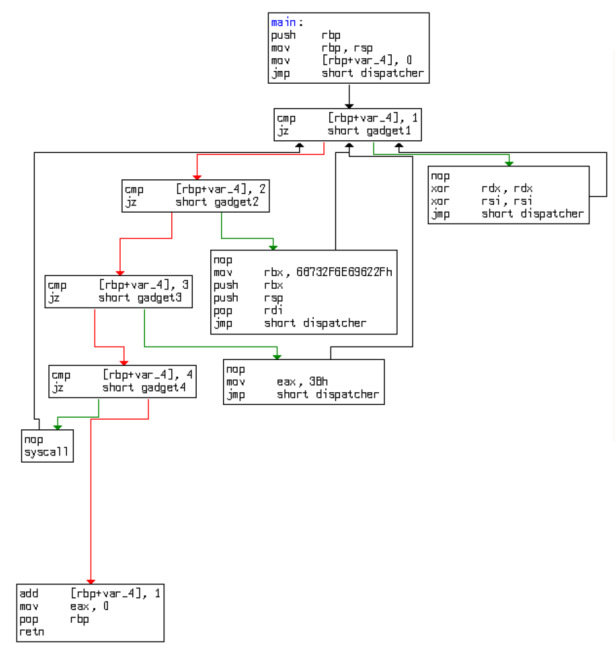


图2-3 JOP攻击流程

JOP相比与ROP，存在以下两点优势：一、ROP攻击使用返回指令完成控制流的转移，需要利用程序的堆栈完成返回操作，而JOP攻击使用跳转指令完成控制流的转移，可以脱离程序的堆栈完成攻击，即：JOP攻击不以赖于堆栈。二、返回指令正常情况下与调用指令成对出现，用于函数的调用与返回，跳转指令一般用于条件分支语句，广泛的存在于程序代码中，JOP攻击选用以跳转指令为结尾的gadget组成gadget链，如图2-4，其攻击行为的指令特点不明显，类似于正常程序的分支跳转语句，具有很好的隐蔽性。



1. JOP调度代码执行流程 (b) 分支跳转代码执行流程

图2-4 JOP调度代码与正常分支分支跳转语句对比

调用导向编程[14]（Call Oriented Programming, COP）由Nicholas Carlini和David Wagner于2014年提出。攻击者用以**间接调用指令**为结尾的gadget代替以返回指令为结尾的gadget。COP攻击与JOP攻击看似区别不大，但有一个重要的区别：间接调用通常是内存间接调用，即：程序控制流的转移位置由内存中的值决定，而不是由寄存器的值直接决定。因此，COP攻击不需要调度程序，如图2-5，攻击者只需要按顺序将内存间接位置指向下一个gadget即可将gadget链接在一起。初始化一次COP攻击，要比ROP与JOP困难得多，攻击者除了需要劫持程序控制流、覆写特定的间接调用位置之外，还必须控制堆栈。这些条件，难以通过一次漏洞利用完成，因此COP攻击通常用作ROP攻击的辅助攻击手段。

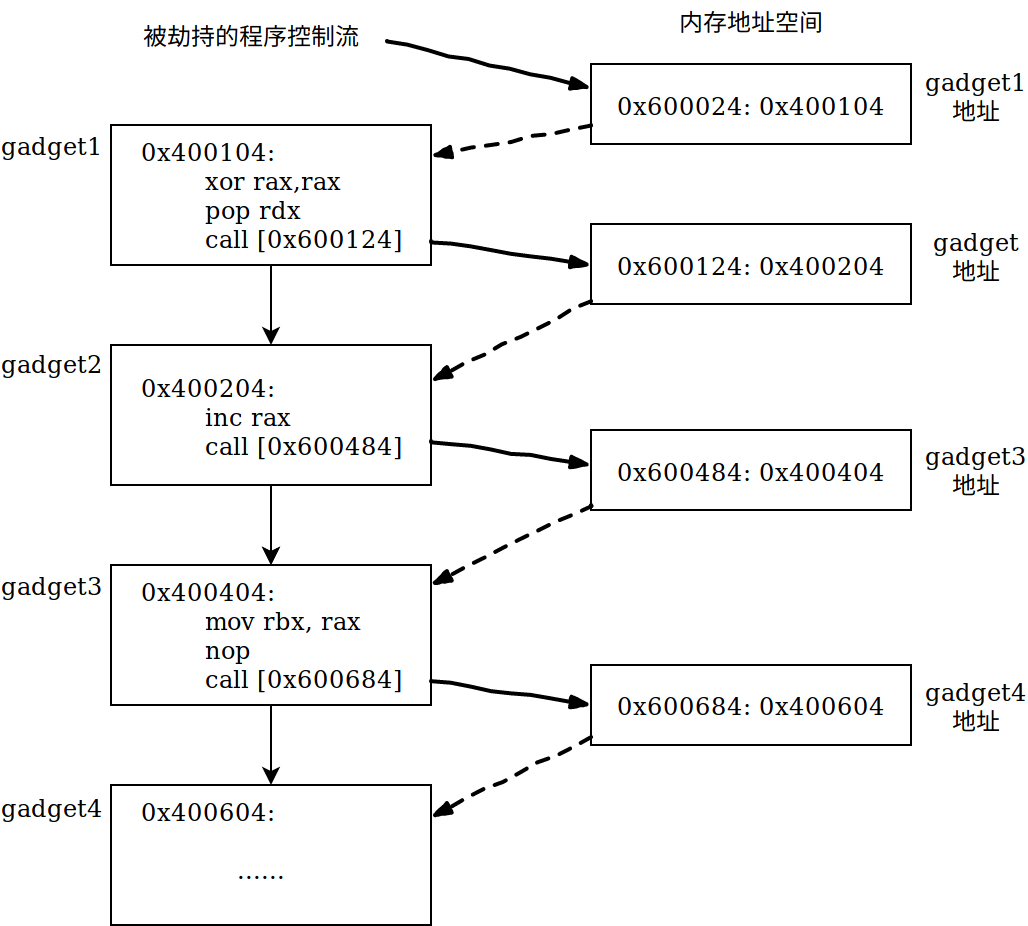


图2-5 COP攻击流程

2.2 常见程序漏洞

2.2.1 缓冲区溢出漏洞

在c语言中，缓冲区用于变量的存储，是内存中临时分配的一块空间。缓冲区溢出的原因是因为一些操作缓冲区的函数，如read,gets,strcpy,memcpy等，没有对缓冲区的边界进行保护，允许任意长度的数据数据被拷贝到缓冲区中，所以当拷贝的数据长度大于缓冲区长度时，缓冲区就会溢出。缓冲区溢出将导致内存中与缓冲区相邻的其他内存数据被覆盖。

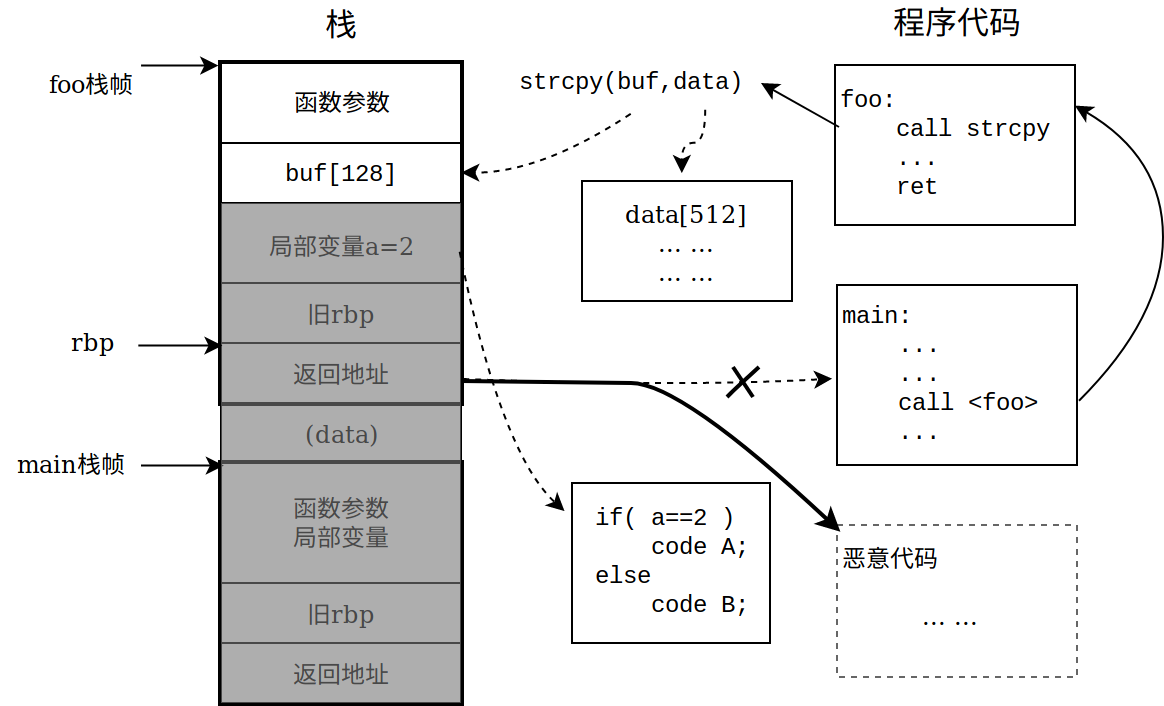


图2-6 缓冲区溢出漏洞利用流程

如图2-6所示，foo函数中调用了strcpy函数，将长度为512字节的数据拷贝到长度只有128字节的缓冲区中，由于缓冲区位于栈中，于是发生了栈溢出，缓冲区后的数据被覆盖。如图2-6，攻击者利用缓冲区溢出漏洞发起攻击，将会导致：

1. 通过覆盖局部变量，改变程序的执行逻辑；
2. 通过覆盖栈中保存的旧栈基址寄存器的值，在栈帧销毁时（详见2.1.1），控制堆栈至攻击者指定的位置；
3. 通过覆盖函数返回地址，在函数返回时，实现程序控制流的劫持。
4. 通过覆盖其他函数栈帧中的变量、栈基址、返回地址等数据，完成各式各样的攻击。

根据缓冲区溢出发生的位置不同，缓冲区溢出可以分为栈溢出和堆溢出。一般情况下，栈中保存函数局部变量、参数、返回地址等临时变量，而静态变量，调用malloc函数分配的变量等非临时变量保存在堆中。由于函数的返回地址位于栈中，对栈溢出的利用更为常见。

针对于修改返回地址的栈溢出，Cowan[28]等人提出了一种防御方式：在返回地址前设置canary值，在函数返回前先验证canary的值是否被修改，如果没被修改，程序将正常返回，如果缓冲区溢出到了函数返回地址，canary的值被修改，保护程序将被调用，报告栈溢出，终止进程。图2-7展示了这种防御机制的工作流程。由于部署方便，这种防御方式被广泛应用，加大了缓冲区溢出攻击的难度。不过攻击者可以通过泄露cancry的值，或者攻击保护程序，从而绕过这种防御机制。

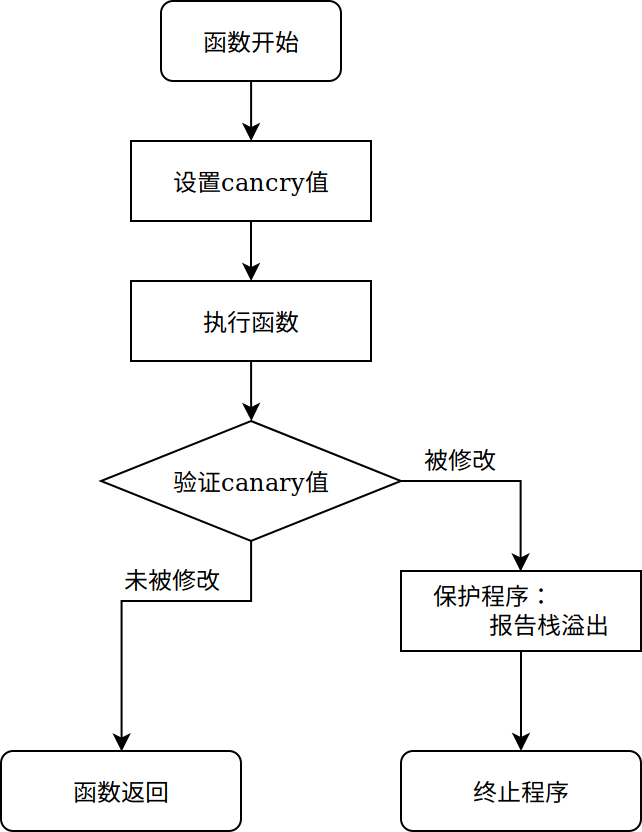


图2-7 设置canary值的栈溢出防御机制

2.2.2 格式化字符串漏洞

在C语言的标准输入输出函数库中定义了fprintf,printf,sprintf等格式化输出函数。这些函数根据控制字符串表示的格式，把输出转换成一系列格式发送到输出流中。以printf为例，他的输出流为标准输出，其函数原型为int printf ( const char \* format, ... )，其参数由两部分组成：第一部分是格式化字符串，包含一般字符和格式控制字符，其中格式控制字符是以%为开头的字符串，%后可接各种控制符，详见表2-1。第二部分为输出表列，表列中的参数个数不定，由第一部分格式化字符串中的控制符的个数决定。参数作为临时变量，存储在栈中，其中个数字符串为第一个参数，输出表列为第2至第n各参数。

表2-1 格式化字符串常见控制符号

|  |  |  |
| --- | --- | --- |
| 控制符 | 对应数据类型 | 含义 |
| d | int | 输出有符号10进制整数 |
| o | unsigned int | 输出无符号8进制整数 |
| u | unsigned int | 输出无符号10进制整数 |
| x | unsigned int | 输出无符号16进制整数 |
| f/lf | float/double | 输出单精度浮点数/双精度浮点数 |
| c | char | 输出字符 |
| s | char\* | 输出字符串 |
| p | void\* | 输出指针(16进制形式) |
| n | int\* | 将在此之前输出的字符数存储到参数指针所指的位置 |

由于printf函数的参数个数不定，当格式化字符串中控制符数量多于输出表列中的参数个数时，printf函数将会以栈中的其他数据作为其输出表列中的输出项，这就是格式化字符串漏洞。举例来说，程序中存在一条语句printf(buf)，其中buf为一个字符数组，其中的数据由用户的输入决定。这条语句原本功能是输出buf中的普通字符串，但是攻击者通过向字符串中添加格式化控制符，将其伪造成格式化字符串，由于没有输出列表，格式化字符串中控制符将直接对应栈中的数据。此外，控制符号中存在一个特殊控制符号%n，不同与其他用于输出的控制符，%n用于将先前输出的字符个数写入其参数对应的内存中。于是，除了泄漏内存信息外，攻击者还可以利用控制符%n修改内存数据。

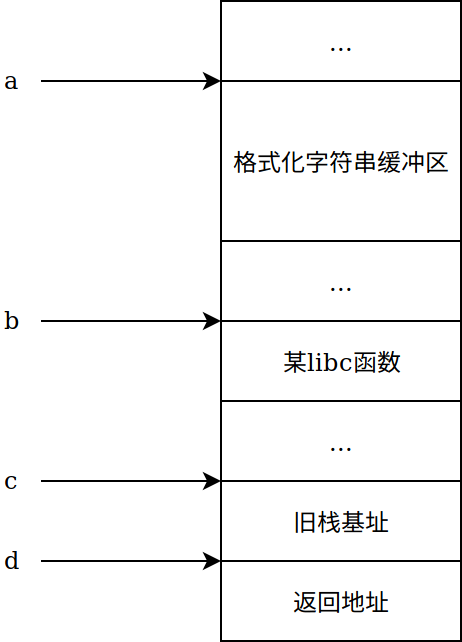


图2-8 格式化字符串漏洞的利用

如图2-8中的栈布局，攻击者可以：

1. 通过%p输出b处的libc函数地址，泄漏libc中函数的实际地址，绕过ASLR对libc的随机化保护；
2. 通过%p输出c处的旧栈基址，泄露栈空间的位置，绕过ASLR对栈空间的随机化保护；
3. 通过%n修改d处的程序返回地址，达到控制程序控制流的目的；
4. 将某一内存地址写入缓冲区中a处，通过%p或%n输出或修改a处的内容，达到泄漏任意内存数据或修改任意内存数据的目的。

除了由程序员代码编写不规范（如printf(buf)）造成的格式化字符串漏洞之外，攻击者还能够通过程序中存在其他的漏洞，手动构造格式化字符串漏洞，因此，格式化字符串漏洞很难避免且危害极大。

2.3 辅助攻击手段

2.3.1 绕过随机化

ASLR是被广泛应用的一种随机化防御机制。ASLR在程序运行前，为程序的堆栈、共享库映射等线性内存区域分配一个随机基址，实现了粗粒度的随机化布局。但是常用的共享库，如libc中的函数地址会经常出现在栈或是解析后的GOT表（详见2.3.2）中，因此，攻击者可以通过printf，puts, write等具有输出功能的函数，或者利用代码中存在的格式化字符串漏洞（详见2.2.2），将经过随机化后的libc函数地址泄漏。共享库中函数的实际地址，可由公式I计算。由于各个函数在libc库中的相对位置关系不变，因此只需要泄漏一个libc中函数的实际地址，便可以通过实际地址减去该函数在libc中的偏移量，计算libc基址，于是其他函数的实际地址便可以通过基址加上该函数在libc中偏移量的方法计算出，由此绕过ASLR防御机制。随机化大部分可以通过泄露内存的方法绕过。

实际地址=基址+偏移量 (I)

2.3.2 篡改GOT表

为了优化动态链接带来的效率问题，linux的可执行文件采用了一种叫做延迟绑定(Lazy Binding)的做法，即：当函数第一次被用到时进行绑定（符号査找、重定位等），如果没有用到则不进行绑定。所以，在程序开始执行时，所有的函数调用都没有进行绑定，只有在需要用到的时侯，才由动态链接器来负责绑定。延迟绑定大大加快程序的启动速度。为了完成延迟绑定，linux的可执行文件中引入了过程链接表(Procedure Linkage Table, PLT)，PLT表中使用了一些很精巧的指令序列来完成延迟绑定。图2-9(a)展示了延迟绑定中动态解析函数地址的流程：当printf函数首次被调用时，程序控制流进入printf的PLT表中，随后执行一条间接跳转指令，跳转指令的目标地址是printf的GOT表中保存的地址，GOT表中的初始值是printf的PLT表的第二条指令地址。于是程序控制流又回到了PLT表中，并进行后续的动态解析操作，当函数地址解析成功后，GOT表中的值将被修正为printf的实际地址。当printf函数再次被调用时，如图2-9(b)，由于其GOT表中已经是printf的实际地址，printf函数将直接被执行，而无需再进行动态的函数地址解析。

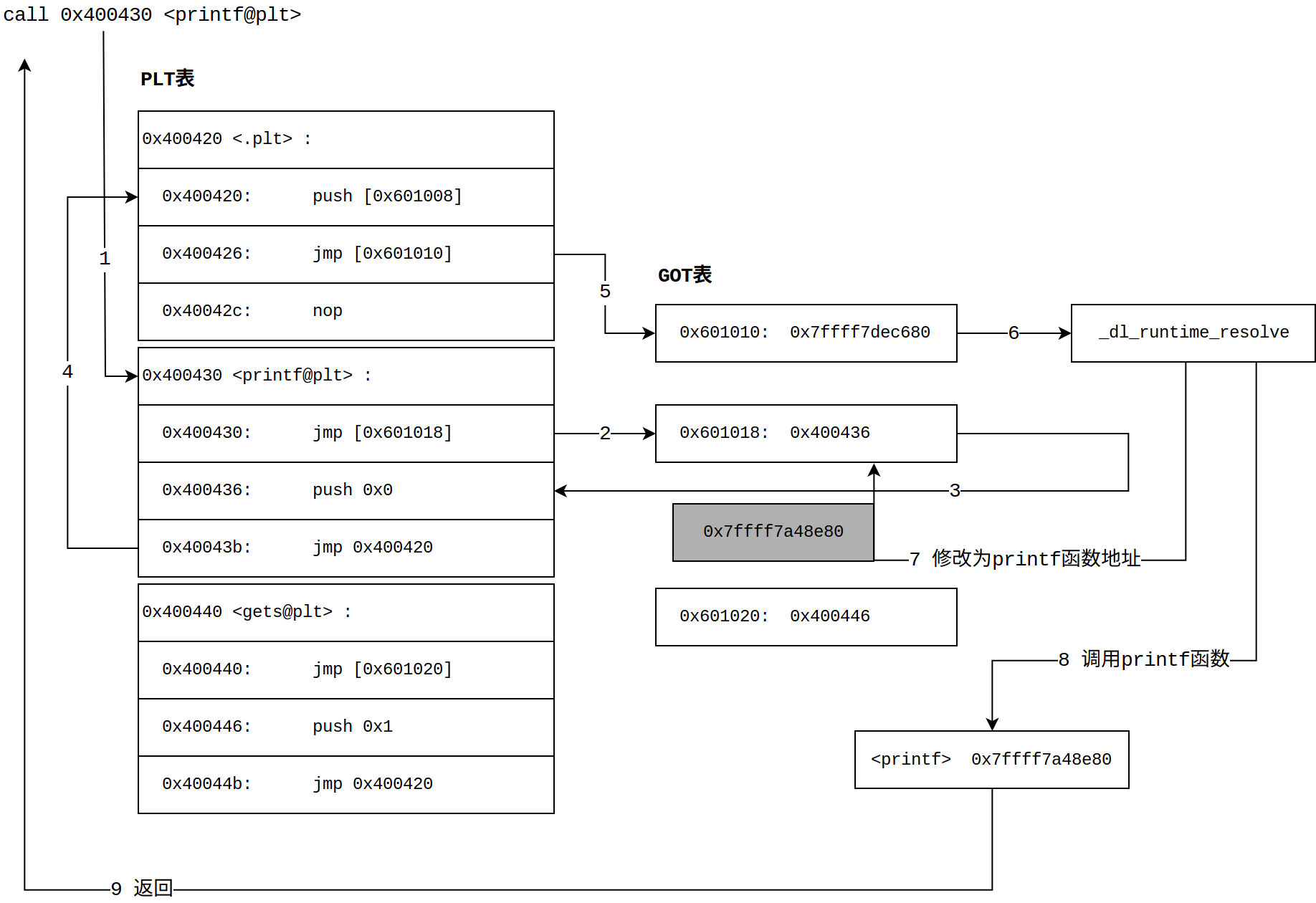


图2-9(a) 延迟绑定流程之动态解析

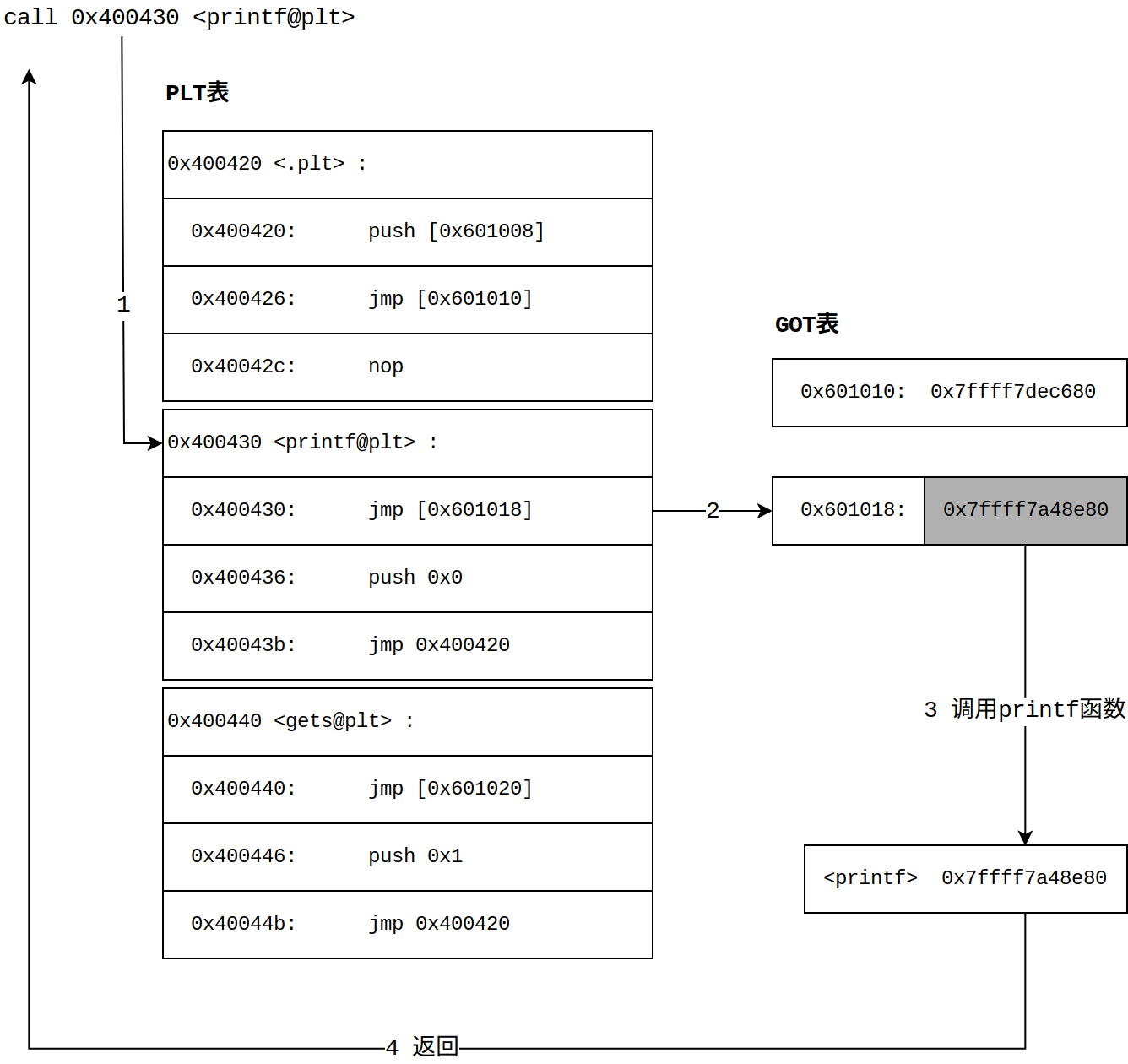


图2-9(b) 延迟绑定流程之再次调用

虽然延迟绑定大大加快程序的启动速度，但是也带来了安全隐患。从上述延迟绑定的流程中，可以看出，PLT表使用了大量的间接跳转指令，而间接跳转使用的内存指针位于GOT表中。GOT表的一个特点是位置固定，位于程序的数据段中，攻击者能够通过readelf, objdump等反汇编工具读取函数的GOT地址，因此攻击者针对GOT表进行攻击极为方便。例如，攻击者利用格式化字符串漏洞，将某函数GOT表中的值篡改，当这个函数被调用时，程序的控制流即被劫持。

GOT表之所以能够被篡改，是因为它的位置暴露，且是一张存储函数指针的表，指针作为存储内存地址的一种变量，可以被任意修改。C/C++语言编写的程序中存在大量的指针。凡是类似的函数指针表，皆可以被攻击，比如C++中虚函数表，虚函数表位于程序数据段，表中存储虚函数指针，这些指针指向虚函数的实际地址，攻击者可以利用代码中存在的漏洞，对表中虚函数指针的值进行篡改。当指针被篡改的虚函数被调用时，程序控制流将被攻击者劫持。

2.4 本章小结

本章首先介绍ROP攻击的原理与具体攻击流程，随后对ROP攻击的两种变种攻击JOP攻击和COP攻击进行介绍，最后从攻击特点、攻击难度等方面对这三种攻击进行对比，总结三种攻击方式的优势与缺点，其中JOP攻击可以脱离程序堆栈完成攻击，能够绕过各种针对堆栈进行保护的防御机制，与ROP和COP相比，具有明显的优势。

ROP攻击的启动需要利用程序存在的漏洞，即攻击者需要先劫持程序控制流，才能进而执行ROP恶意代码，完成ROP攻击，因此本章介绍了两种常见的程序漏洞：缓冲区溢出漏洞和格式化字符串漏洞。缓存区溢出漏洞较为常见，但是以赖于程序堆栈，容易被众多的针对堆栈进行保护的防御机制检测出来。格式化字符串漏洞可以导致堆栈以外的内存数据被修改，因此对堆栈的依赖性小，可以配合JOP攻击，完成一次栈无关的攻击。

攻击者在进行ROP攻击时，往往需要一些关于漏洞程序的额外信息，如被复用代码的地址，程序中易被攻击的目标，这些信息需要利用额外的辅助攻击手段取得。因此本章介绍了绕过随机化和篡改GOT表两种常见的与ROP攻击相关的辅助攻击手段。操作系统开启随机化保护后，共享库映像等内存区域被随机分配了基址，因此攻击者无法直接获取被复用代码的准确地址，但是攻击者可以通过泄露内存的方法，间接地得到被复用代码的准确地址。程序GOT表与PLT表配合，用于实现延迟绑定，GOT表的位置固定，表中存储函数指针，PLT表中存在间接跳转指令，间接跳转的目标地址即为相应GOT表中的值，因此攻击者可以篡改GOT表中函数指针的值，完成程序控制流的劫持。辅助攻击手段不能单独完成一次攻击，需要与ROP攻击配合。

1. **ROP攻击动态特征 1500+**

3.1 指令特征

ROP恶意代码由具有各种功能的gadget链接而成，gadget是以返回指令为结尾的**短指令片段**，因此，ROP恶意代码实质上是连续的以返回指令为结尾的短指令组成的指令序列。

从指令的级别考虑，ROP攻击的动态特征即为程序执行的ROP恶意代码与正常代码之间的差异。在正常程序代码中，返回指令用于函数的返回操作，而在ROP恶意代码中，返回指令用于链接gadget。

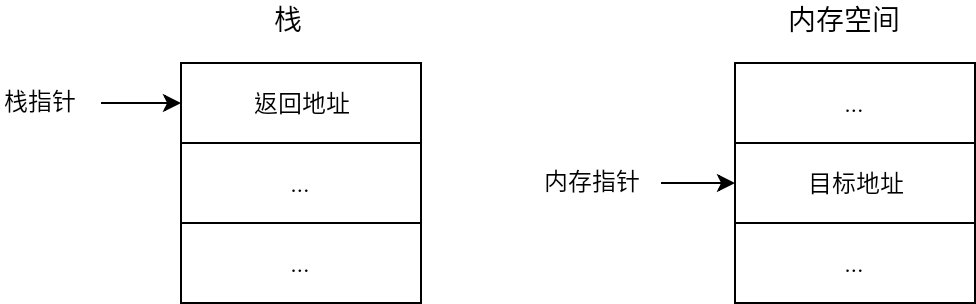
由于函数的调用与返回成对出现，因此在程序正常执行的过程中，调用指令与返回指令也总是成对出现，且被执行的调用指令数量总是大于或等于被执行的返回指令数量。而在ROP攻击代码中，由于复用了许多返回指令，破坏了调用返回指令成对出现这一特点，使得被执行的返回指令数量多于被执行的调用指令。一般情况下，这一特点可以作为ROP攻击的动态特征的指令特征。此外，正常的程序代码中，除递归调用以外，很少会出现连续的以返回指令为结尾的短指令组成的指令序列，因此gadget的指令特征，也可以作为参考。 综上所述，可以得到ROP攻击指令级别的动态特征为：1.被执行的返回指令数大于调用指令数。2.连续的以返回指令为结尾的短指令组成的指令序列被执行。

由于程序的复杂性，指令特征存在一定的局限性，本文将在第四章中进行讨论。

3.2 内存特征

ROP攻击需要利用程序存在的漏洞才能得以启动，攻击者首先需要劫持程序控制流，才能进而执行ROP恶意代码，完成ROP攻击。因此，ROP攻击必然导致程序的正常执行流程遭到破坏。能够转移程序控制流的指令有三种：返回指令、调用指令、跳转指令。因此，攻击者针对这三种指令发动攻击，可以导致程序控制流被劫持。程序控制流被劫持时的内存特征可分为两类：返回地址被篡改、内存指针被篡改。

返回指令使用到了堆栈指针，即程序控制流转移的位置由栈顶元素决定，栈顶元素正常情况下为函数调用时保存的返回地址。攻击者可以利用程序漏洞，如栈溢出，篡改栈中的返回地址，在返回指令执行后，劫持程序控制流。间接调用或者间接跳转指令使用到了内存指针，即程序控制流转移的位置由内存中的值决定。攻击者可以利用程序漏洞，如格式化字符串漏洞，篡改内存指针中的值，在间接调用或间接跳转指令执行后，劫持程序控制流。返回地址被篡改也可以算作是内存指针被篡改，因为返回地址保存在栈中，堆栈指针也是内存指针的一种。



总而言之，从程序内存的角度考虑，ROP攻击的动态特征表现为正常程序的内存数据被篡改，即程序的完整性遭到破坏。根据被攻击者利用的指令来分类，可以得到ROP攻击时程序内存的动态特征有：1.栈中返回地址被篡改。2.内存中指针的值被篡改。

3.3 本章小结

本章分析并总结ROP攻击的动态特征。所谓的ROP攻击动态特征，就是遭受ROP攻击的程序与正常运行的程序之间的差异。本章首先分析了ROP攻击的指令特征，包括被执行的返回指令数量大于调用指令数量和连续的以返回指令为结尾的短指令序列被执行。但是，由于程序的复杂性，这两种指令特征在某些特定的情况下无法作为ROP攻击的特征，因此存在有一定的局限性。于是，本章从内存角度出发，提取了ROP攻击的另一种动态特征。ROP攻击的内存特征比较明显，即内存的完整性被破坏，具体表现为栈中的函数返回地址被篡才或是内存指针的值被篡改。本章中总结的ROP攻击的动态特征，将作为第四章ROP攻击的检测的基础。

1. **ROP攻击检测方法 4000+**

4.1 指令特征检测 1000+

4.1.1 CALL-RET指令数检测 500+

正常程序的调用指令数量大于或等于返回指令数量，而遭受ROP攻击的程序返回指令数量大于调用指令数量。根据这一特征，可以分别设置两个计数器，用于记录程序执行的调用指令数量和返回指令数量。调用指令计数器和返回指令计数器的初始值均为0，在程序开始运行时，激活指令计数器，当一条调用指令被执行时，调用指令计数器数值加1，当一条返回指令被执行时，返回指令计数器数值加1。返回指令执行后，对比两指令计数器中的数值，若调用指令计数器中的值大于或等于返回指令计数器中的值，则表明程序正常，继续执行计数操作，若调用指令计数器中的值小于指令计数器中的值，则说明程序异常，有可能正在遭受ROP攻击。图4-1展示了CALL-RET指令数检测方案的流程。

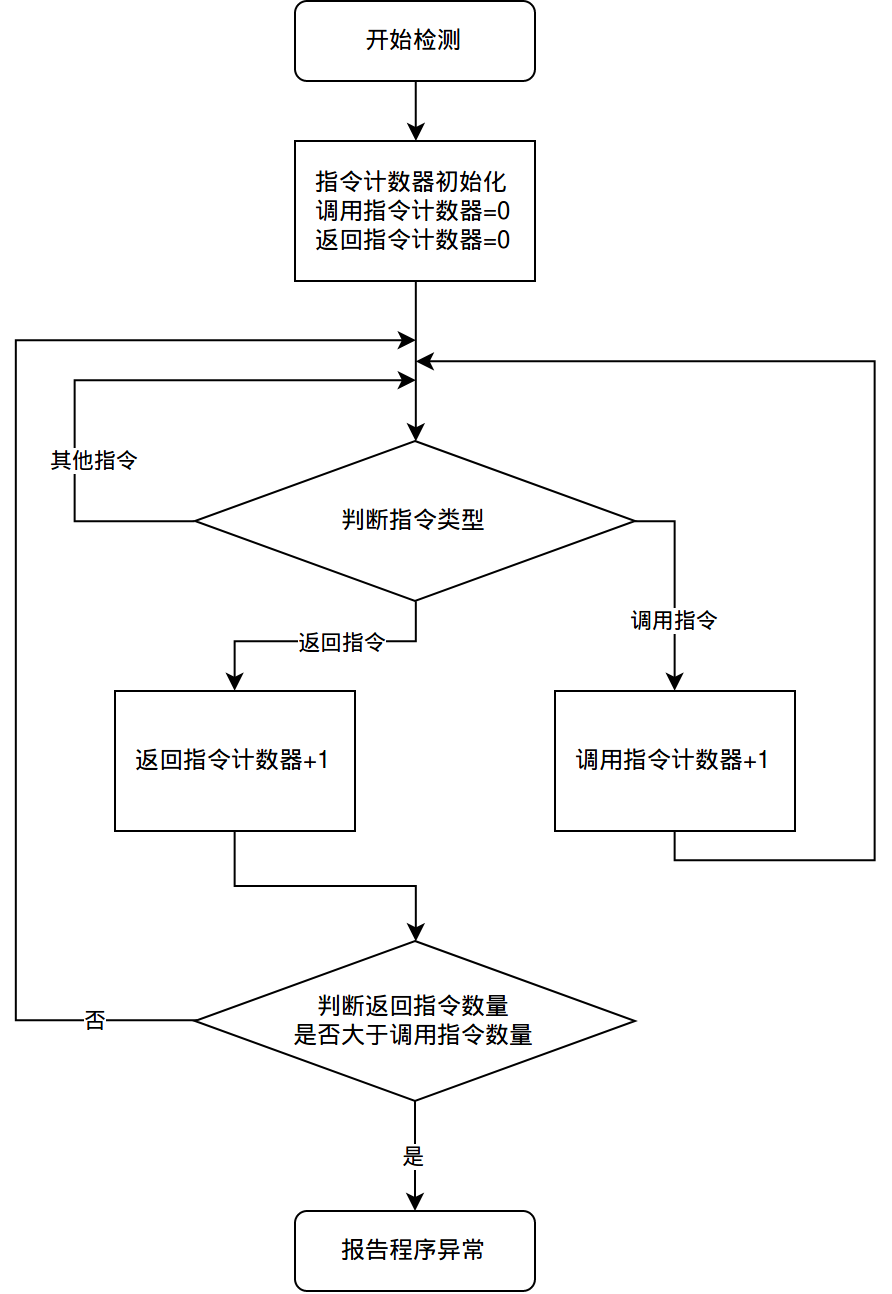


图4-1 CALL-RET指令数检测流程

因为c语言中存在函数的嵌套调用，所以调用指令数量与返回指令数量的差值不固定，差值的最大值取决于被调用函数的层次。若函数A中调用了函数B，函数B中又调用了函数C，函数C中又调用了函数D，则在函数D返回后，调用指令比返回指令的数量多3，直到函数A返回后，调用指令的数量才和返回指令的数量相等。在这个嵌入调用关系中，函数D处于嵌套调用的第4层。因此，调用指令计数器与返回指令计数器的最大差值为最内层函数的层次减1。

如果程序中存在缓冲区溢出漏洞，且溢出点出现在第n层嵌套函数，那么溢出发生时，调用指令计数器的值比返回指令计数器的值多n-1。攻击者使用长度小于n-1的gadget链进行ROP攻击，指令计数器完全检测不到程序异常。因此指令计数器受溢出点位置的限制。本章第4.2小节中介绍的方法可以弥补这一缺陷。

4.1.2 连续gadget检测 500+

在ROP攻击中，gadget的指令长度与连续被执行的gadget的数量，都在一定的数值范围内。文章[24]中指出gadget中的指令数不超过5条。Kayaalp等人[30]从libc标准库中提取了所有gadget并研究了其平均长度。研究结果表明，随着gadget长度的增加，副作用的数量呈线性增长，使得gadget越来越难以被利用。因此，可以基于统计学的方法，设置检测gadget中指令数的阈值T0以及检测连续gadget执行次数的阈值T1。对于所有的以返回指令为结尾的指令序列，只要其长度小于阈值T0，便将其视作候选gadget。若候选gadget连续执行的次数大于阈值T1，则认为程序受到了ROP攻击。这种检测方法的有效性依赖于阈值的选取，因此存在一定的局限性。

本文在2.2.3节中介绍了ROP攻击的变种攻击，攻击者除了使用以返回指令为结尾的传统gadget，还可以使用以调用指令或跳转指令为结尾的变种gadget。变种gadget的应用，增大了ROP攻击代码的指令多样性，使得gadget链与正常代码之间的差异性减小。对于以调用指令为结尾的短指令序列，若其长度小于阈值T0，勉强可以将其视作候选gadget。但是对于以跳转指令为结尾的短指令序列，即使其长度小于T0，也不能将其视作候选gadget，因为跳转指令广泛的存在于正常程序代码中，JOP调度代码与正常分支分支跳转语句相似度极高（详见2.1.3节），区分正常代码与JOP攻击所用的gadget极为困难。

此外，这种检测方法还存在一定机率的误报和漏报。有一些正常程序的指令序列和ROP攻击所使用的指令序列相似，如程序中存在递归调用或嵌套调用，则处理器从最内层函数的返回开始，会连续执行多个返回指令，若返回指令前的指令序列长度小于阈值T0，且嵌套的层数大于阈值T1，则会被误识别为gadget链；攻击者可以通过向gadget链中插入一条含有NOP指令或与NOP指令等价的超长gadget（长度大于阈值T0）的方法，将连续的gadget分成多段，控制连续执行的gadget次数不超过阈值T1，从而绕过检测。

4.2 完整性检测 3000+

4.2.1 调用返回控制流完整性检测 2000+

CPRA定义

根据这一运行时特征，可以对ROP攻击进行检测。

返回地址 - CPRA?

- pop of stack?

Libc\_func -> ret2libc

else last\_ins isret -> rop

Last\_ins iscall -> cop

Last\_ins isbranch -> jop

返回指令是否为CPRA

通过调用-返回转移控制流，if-else内部多用jmp转移。

4.2.1 函数指针控制流完整性检测 1000+

检测位置固定的指针表。

防止修改/二次修改。

GOT

首次调用a - if a.got = a.plt[1] - ok , else err

再次调用a - if a.got = libc[a] - ok , else err

存在程序控制流间接转移，c++虚表指针、GOT表

4.3 本章小结

因此通过识别连续gadget的ROP攻击检测方法具有局限性，无法检测以跳转指令为结尾的gadget。

指令特征的检测方法都存在一定的局限性，因此在对ROP攻击进行检测时，指令特征检测只能用作辅助检测方法，若只采用这种检测方法，必然存在漏报与误报。

1. **ROP攻击检测系统实现 8000+**

5.1 假设 400+

存在漏洞，控制流可以被劫持。

Libc不存在漏洞。(?)

操作系统，开启ASLR防护，libc基址随机。

未开启栈溢出保护。

5.2 总体设计 1000+

信息获取、异常监测（攻击检测），类型识别，（程序保护），日志报告

5.3 系统概述(总体实现) 500+

框架



图x 工作流程

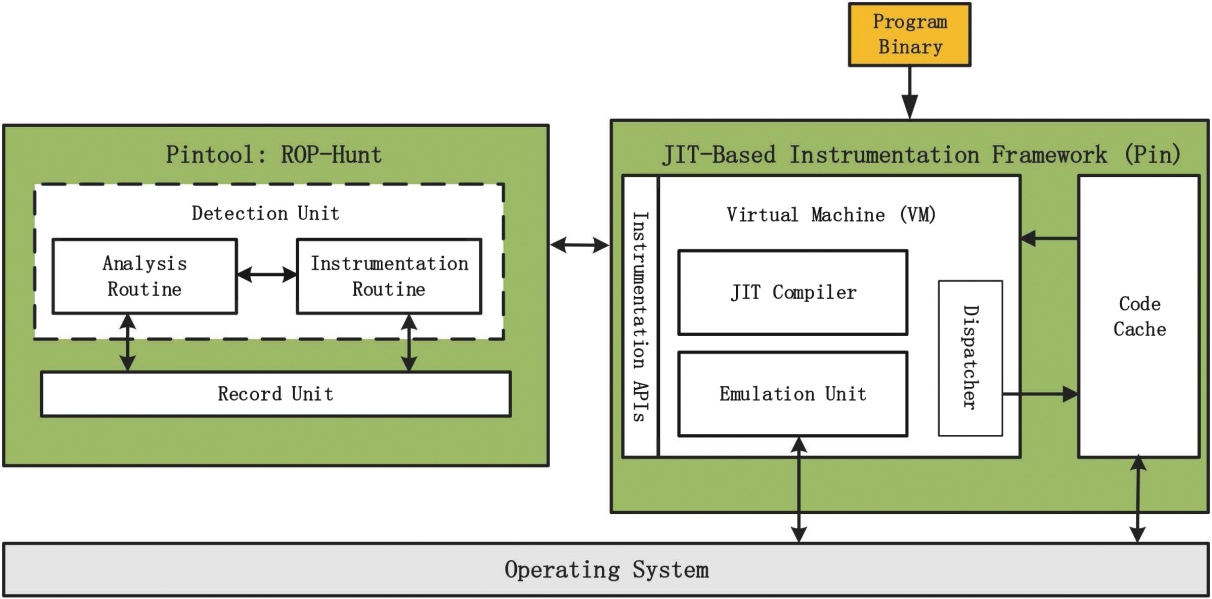


图4 在Pin框架下实现的ROP-Hunt

/\*if static else \*/if libc -> ret-to-libc else gadget(rop) if(ret) rop, if )

5.4 实现细则 5000+

5.4.1 返回地址检测

返回地址为调用指令下一条地址，不是任何函数的起始地址。Ret2libc复用整个函数，将ret的目标地址为libc中函数的起始地址。因此可以检测到ret2libc攻击。

5.4.2 影子栈

Ret2libc利用返回栈顶恶意地址进行攻击，破坏了调用返回的完整性，因此可以通过影子栈进行检测。有局限，缓冲区溢出，只能检测出通过修改返回地址，然而劫持控制流的方式多种多样，例如uaf,修改got表等，为了弥补影子栈的不足，本文对got进行了防护。见4.4.2

5.4.3 阈值检测器

这种方法，通过统计众多ROP攻击中，单个gadget的大小、gadget链的长度，以这两个参数设定阈值，用于区分gadget与正常代码。存在一定机率的误报与漏报。

5.4.4 call-ret指令计数器

影子栈防御了常见的控制流劫持方法，即：修改程序正常返回地址，未知的新型的控制流劫持方法，应提供另一个层次的防御方式。关键指令计数器，不检测程序返回地址是否正常，而是通过检测程序执行过程中call指令和ret指令执行的次数，判断程序是否被攻击。当rop攻击发生时，ret指令数会远多于call指令数。但是如果攻击者利用COP攻击，手动平衡call与ret的指令数，精心构造rop链，也可以绕过call-ret平衡检测。

与溢出点有关，调用4次，溢出在最后一个函数，返回时溢出发生，攻击者劫持控制流，call比ret多3。

可绕过。

5.4.5 GOT篡改检测器

被攻击标记，置1 -> 检测攻击类型模块

5.4.6 CPR检测器

只有6%的gadget是call-preceded gadget，限制gadget使用范围。

5.5界面实现 500+

Web-server (python django)

5.6 实验与评估 2000+

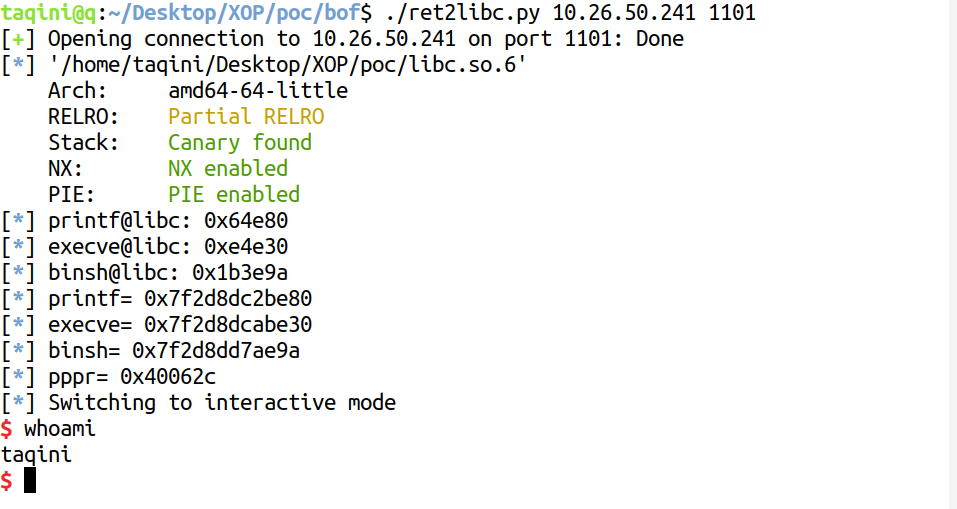
5.6.1实验环境

5.6.2 Ret2libc攻击与检测

利用gadget：



将参数配置完毕，ret至libc中的execve函数，执行execve(“/bin/sh”,0,0);开启shell。

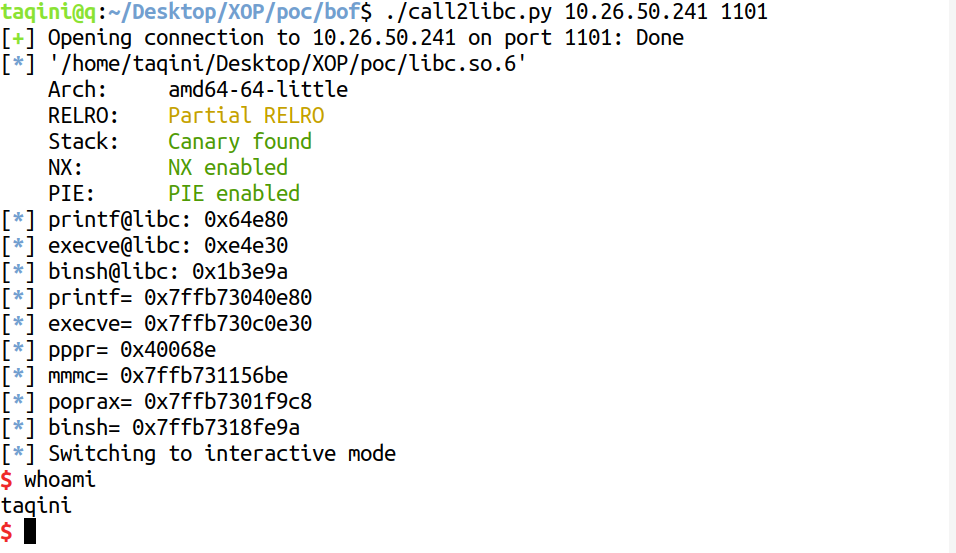




被返回地址检测策略检测到，ret2libc攻击。

使用call指令调用libc函数，代替ret到libc函数，可绕过返回地址检测。





攻击成功，



但是Pintool没有检测到这次攻击。

开启影子栈检测策略，



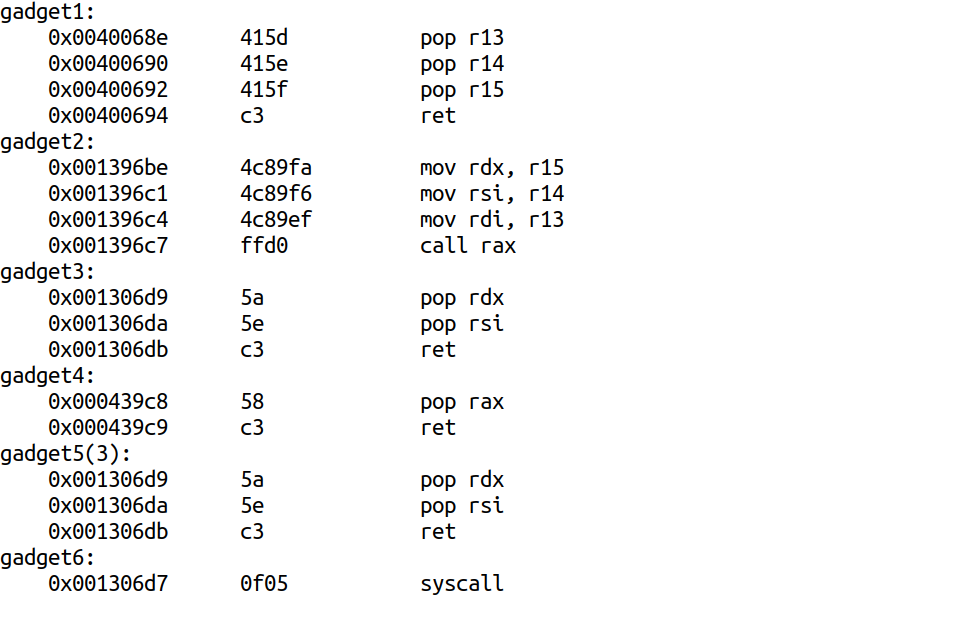
成功检测到了攻击调用了execve函数。

5.6.3 Rop攻击防御与检测

Jop

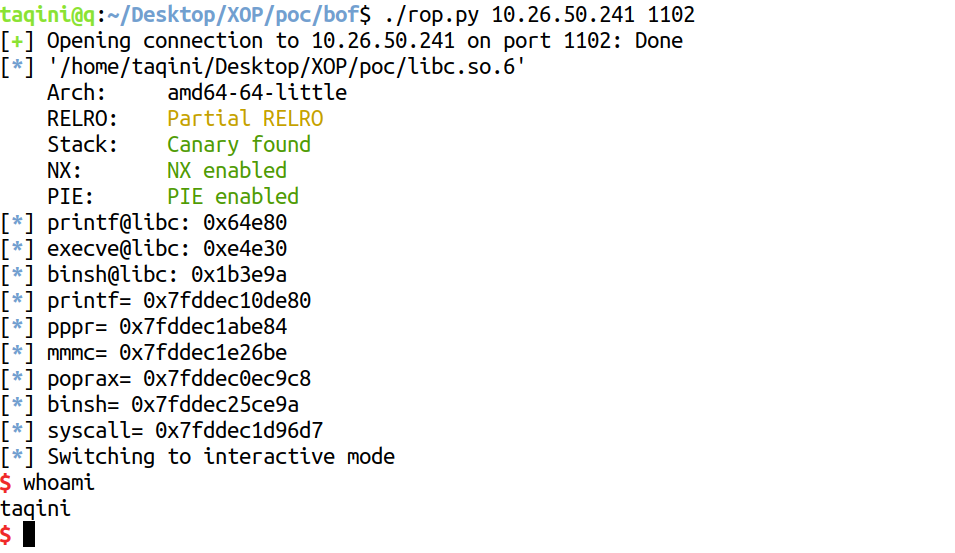
首先开启阈值检测，对漏洞程序进行传统的ROP攻击



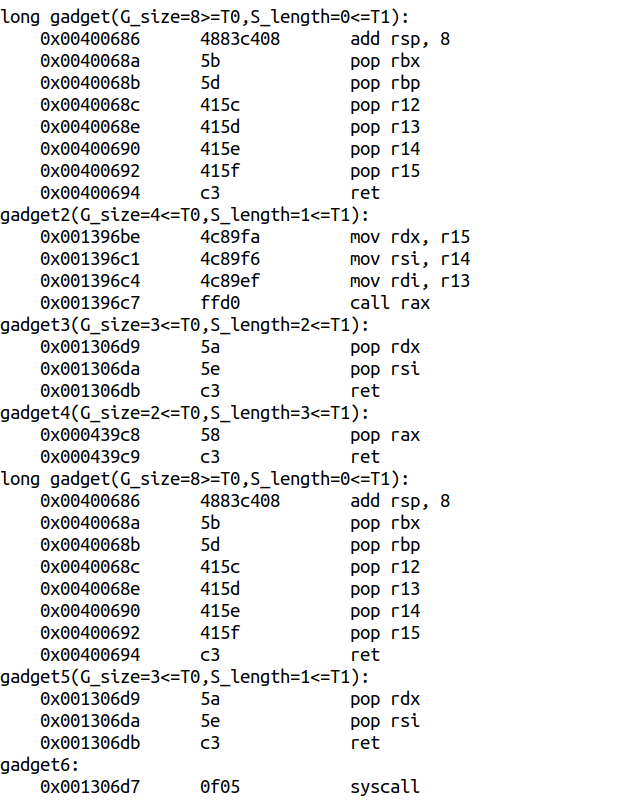


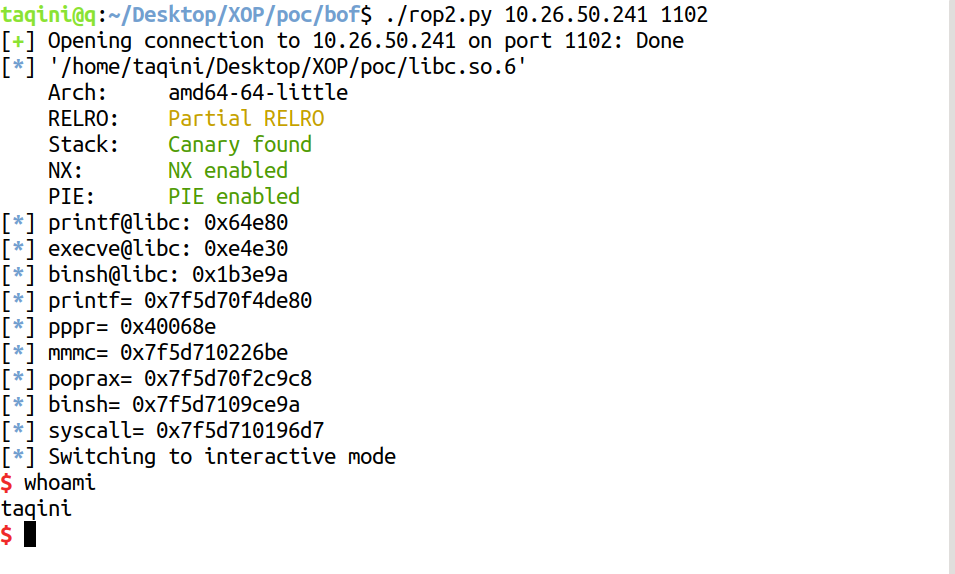
rax - 0x3b,rdx - 0,rdi - '/bin/sh',rsi - 0,syscall。

攻击成功。由于ROP攻击所用的gadget符合阈值T0、T1，被识别到。



修改攻击代码，加长gadget1长度至8，突破T0，将其插入到gadget4和gadget5之间，防止连续gadget长度超多阈值T1，绕过了阈值检测。





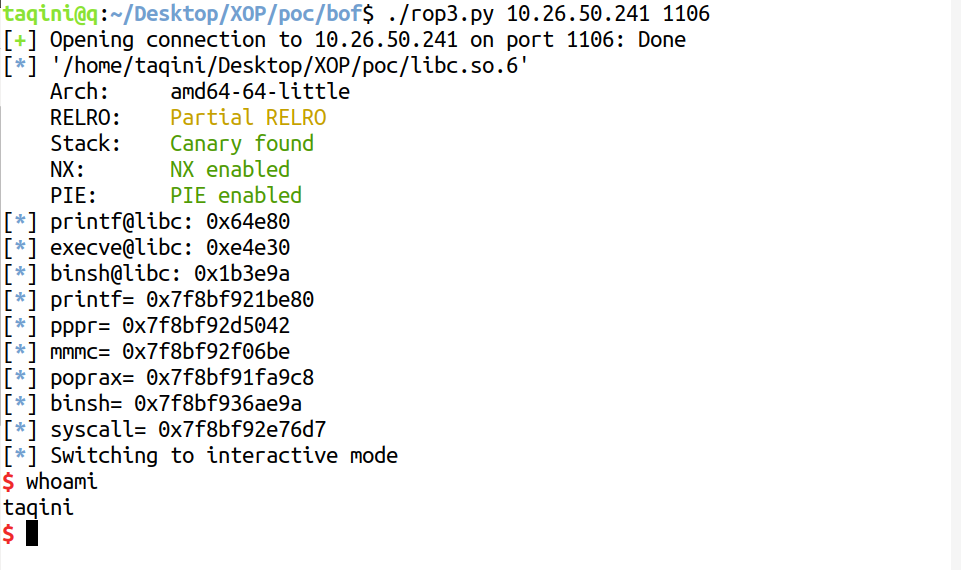


攻击成功，Pintools没有检测到攻击。

开启影子栈，依然使用rop2攻击，被检测到影子栈检测到，并报告了攻击类型。



由于影子栈只检测范围为程序代码段，为了绕过影子栈，将所有gadget换成libc中的代码进行攻击。





成功绕过了影子栈防御。

开启call-ret指令平衡检测方案，再次使用rop3进行攻击。



攻击被CR平衡检测到，并报告了攻击类型。

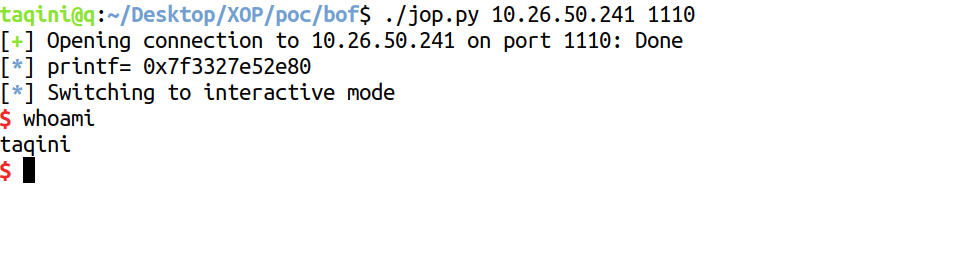
Call-ret平衡以赖于ret指令数量这一特征，攻击者使用以call为结尾的gadget，手动的平衡ret与call的数量，可以绕过指令平衡检测方案，除了混合ROP与COP攻击以外，攻击者可以选择使用纯JOP攻击，避免对ret指令的使用，从而绕过指令平衡。

5.6.4 JOP攻击防御与检测

首先开启阈值检测，对漏洞程序进行传统的JOP攻击



正常代码中也存在诸多跳转指令，因此阈值检测方案无法检测以jmp为结尾的gadget，若检测，则会带来极高的误报。



攻击成功，绕过了指令检测。

[图再说，]

gadget1:

xor rdx,rdx

xor rsi,rsi

jmp dispatcher

gadget2:

mov rbx, '/bin/sh'

push rbx

push rsp

pop rdi

jmp dispatcher

gadget3:

mov rax, 0x3b

jmp dispatcher

gadget4:

syscall

dispatcher:

inc rcx

cmp rcx, 1

je g1

cmp rcx, 2

je g2

cmp rcx, 3

je g3

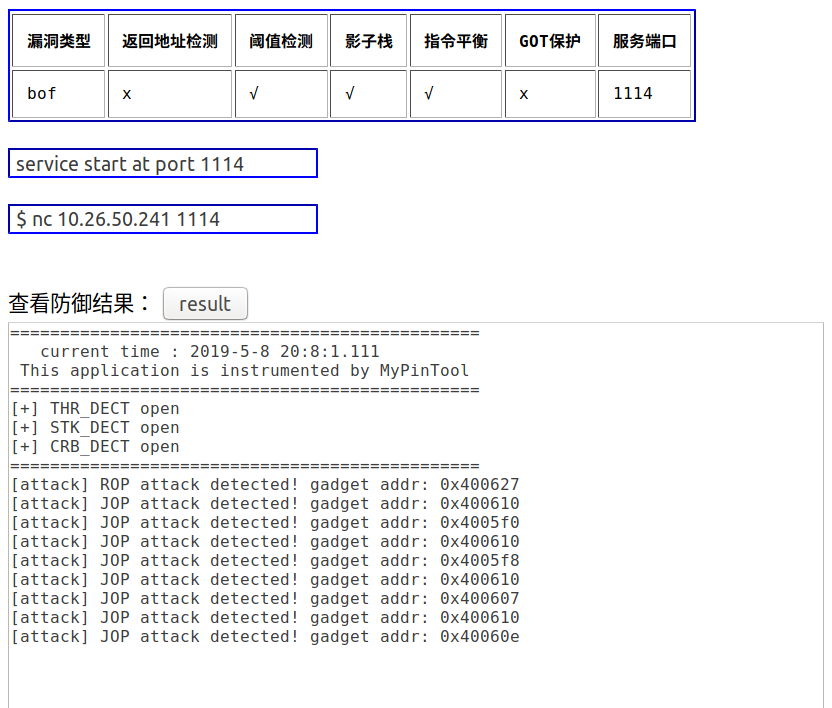
jmp g4

init:

xor rcx, rcx

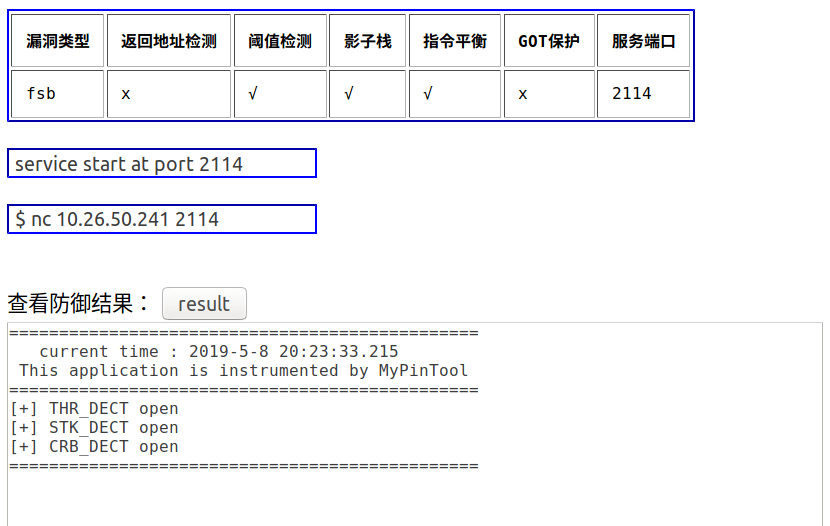
jmp dispatcher

利用缓冲区溢出，将控制流劫持至调度器，这一步可以被影子栈检测到，开启影子栈：

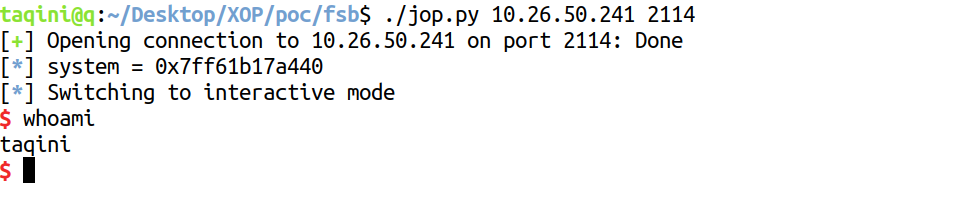


检测到了JOP攻击。

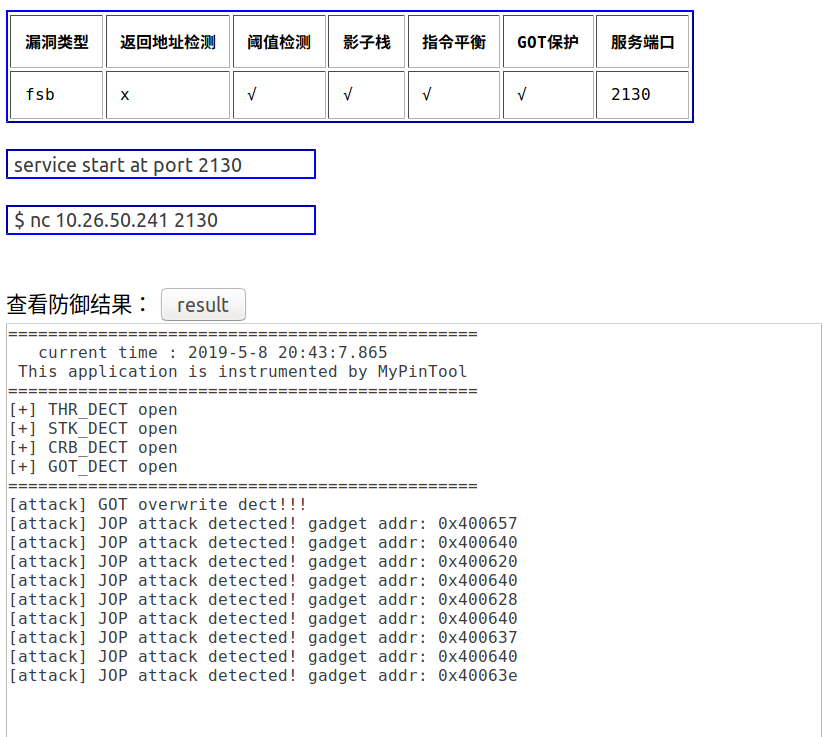
不利用栈溢出，进行攻击，让整个攻击流程与栈无关，攻击带有格式化字符串漏洞的程序。



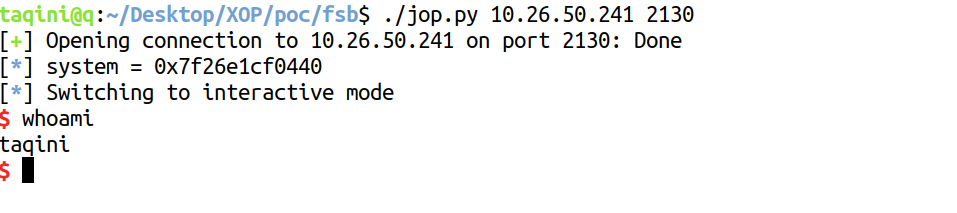
攻击程序利用格式化字符串漏洞，修改了GOT，当程序执行到被修改GOT的函数时，控制流被劫持至JOP调度器中，影子栈检测失效。



开启GOT表检测。



GOT表的非法修改被检测到。



5.6.5 结果评估（表格）

都讲一讲

5.7 本章小结

1. **总结和展望**

6.1 总结

6.2 展望

**参考文献**

1. https://www.cvedetails.com/vulnerabilities-by-types.php
2. Data execution prevention. <http://support.microsoft.com/kb/875352/EN-US>
3. Wojtczuk, R.: The advanced return-into-lib(c) exploits: PaX case study. Phrack Mag. 0x0b(0x3a), Phile# 0x04 of 0x0e (2001)
4. Shacham, H.: The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In: Proceedings of the 14th ACM Conference on Computer and Communications Security, pp. 552-561. ACM (2007)
5. Kornau, T.: Return oriented programming for the ARM architecture. Ph.D. thesis, Masters thesis, Ruhr-Universität Bochum (2010)
6. Buchanan, E., Roemer, R., Shacham, H., Savage, S.: When good instructions go bad: generalizing return-oriented programming to risc. In: Proceedings of the 15th ACM Conference on Computer and Communications Security, pp. 27-38. ACM (2008)
7. Checkoway, S., Feldman, A.J., Kantor, B., Halderman, J.A., Felten, E.W., Shacham, H.: Can DREs provide long-lasting security? The case of return-oriented programming and the AVC advantage. In: EVT/WOTE 2009 (2009)
8. Francillon, A., Castelluccia, C.: Code injection attacks on Harvard-architecture devices. In: Proceedings of the 15th ACM Conference on Computer and Communications Security, pp. 15-26. ACM (2008)
9. Tran, M., Etheridge, M., Bletsch, T., Jiang, X., Freeh, V., Ning, P.: On the expressiveness of return-into-libc attacks. In: Sommer, R., Balzarotti, D., Maier, G. (eds.) RAID 2011. LNCS, vol. 6961, pp. 121-141. Springer, Heidelberg (2011). doi:10.1007/978-3-642-23644-0 7
10. Dullien, T., Kornau, T., Weinmann, R.P.: A framework for automated architecture-independent gadget search. In: WOOT (2010)
11. Hund, R., Holz, T., Freiling, F.C.: Return-oriented rootkits: bypassing kernel code integrity protection mechanisms. In: USENIX Security Symposium, pp. 383-398 (2009)
12. Roemer, R.G.: Finding the bad in good code: automated return-oriented programming exploit discovery (2009)
13. Schwartz, E.J., Avgerinos, T., Brumley, D.: Q: Exploit hardening made easy. In: USENIX Security Symposium, pp. 25-41 (2011)
14. Carlini, N., Wagner, D.: ROP is still dangerous: breaking modern defenses. In: 23rd USENIX Security Symposium (USENIX Security 2014), pp. 385-399 (2014)
15. Bletsch, T., Jiang, X., Freeh, V.W., Liang, Z.: Jump-oriented programming: a new class of code-reuse attack. In: Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security, pp. 30-40. ACM (2011)
16. Kevin Z. Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and A.R. Sadeghi, Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization - IEEE Symposium on Security and Privacy, 574-588 (2013)
17. A Bittau, A Belay, A Mashtizadeh, D Mazières, D Boneh - IEEE Symposium on Security and Privacy, 227-242 (2014)
18. PaX Team. <http://pax.grsecurity.net/.>
19. Pappas V , Polychronakis M , Keromytis A D . Transparent ROP exploit mitigation using indirect branch tracing, in 22nd USENIX conference on Security, pages 447-463. (2013)
20. M. Backes and S. Nurnberger. Oxymoron: Making fine-grained memory randomization practical by allowing code sharing. In Proceedings of the 23rd USENIX Security Symposium. (2014)
21. Si, Lu , et al. "ROP-Hunt: Detecting Return-Oriented Programming Attacks in Applications." International Conference on Security, Privacy and Anonymity in Computation, Communication and Storage Springer, Cham. (2016)
22. Pappas, V., Polychronakis, M., Keromytis, A.D.: Transparent ROP exploit mitigation using indirect branch tracing. In: Presented as Part of the 22nd USENIX Security Symposium (USENIX Security 2013), pp. 447-462 (2013)
23. Davi, L., Sadeghi, A.R., Winandy, M.: Ropdefender: adetection tool to defend against return-oriented programming attacks. In: Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security, pp. 40-51. ACM (2011)
24. Chen, P., Xiao, H., Shen, X., Yin, X., Mao, B., Xie, L.: DROP: detecting returnoriented programming malicious code. In: Prakash, A., Sen Gupta, I. (eds.) ICISS 2009. LNCS, vol. 5905, pp. 163-177. Springer, Heidelberg (2009). doi:10.1007/978-3-642-10772-6 13
25. Bletsch, T., Jiang, X., Freeh, V.: Mitigating code-reuse attacks with control-flow locking. In: Proceedings of the 27th Annual Computer Security Applications Conference, pp. 353-362. ACM (2011)
26. Onarlioglu, K., Bilge, L., Lanzi, A., Balzarotti, D., Kirda, E.: G-free: defeating return-oriented programming through gadget-less binaries. In: Proceedings of the 26th Annual Computer Security Applications Conference, pp. 49-58. ACM (2010)
27. Aurélien Francillon, Daniele Perito, and Claude Castelluccia. Defending embedded systems against control flow attacks. In Proceedings of the 1st Workshop on Secure Execution of Untrusted Code (SecuCode'09), pages 19–26. ACM, 2009.
28. Cowan C, Beattie S, Johansen J, et al. Pointguard TM : protecting pointers from buffer overflow vulnerabilities[C]// Conference on Usenix Security Symposium. 2003.
29. Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: building customized program analysis tools with dynamic instrumentation. In: ACM Sigplan Notices, vol. 40, pp. 190-200. ACM (2005)
30. 25.Kayaalp, M., Schmitt, T., Nomani, J., Ponomarev, D., Abu-Ghazaleh, N.: SCRAP: architecture for signature-based protection from code reuse attacks. In: 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA2013), pp. 258-269. IEEE (2013)