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

1.1 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。

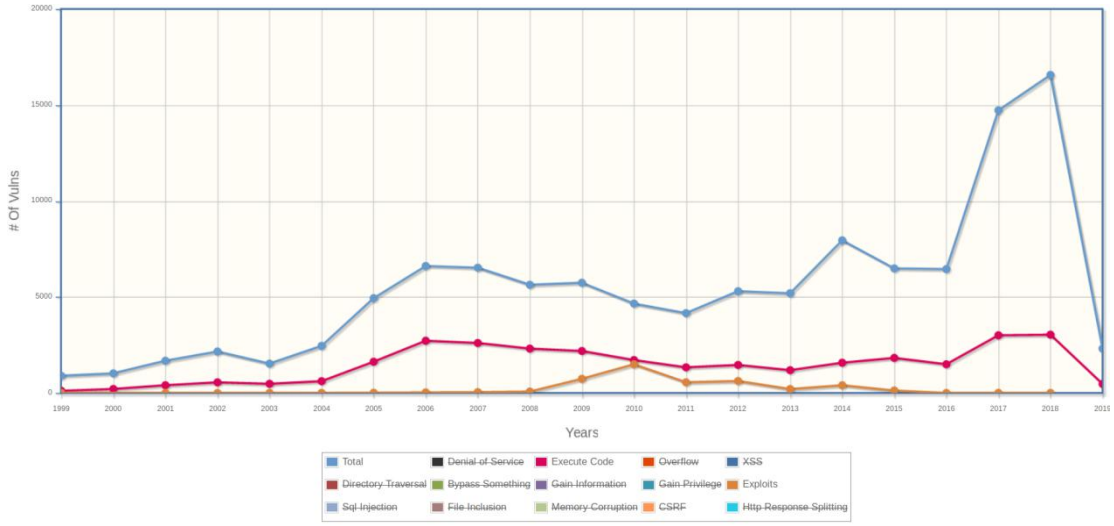


图 1-1 近 20 年 CVE 漏洞数量

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。

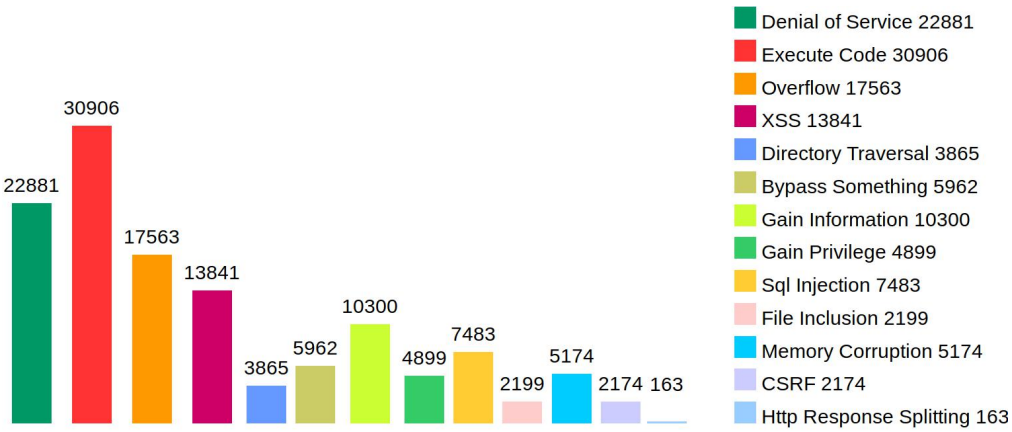


图 1-2 近 20 年各类 CVE 漏洞数量统计

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。

1.2 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[19]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。

1.3 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。

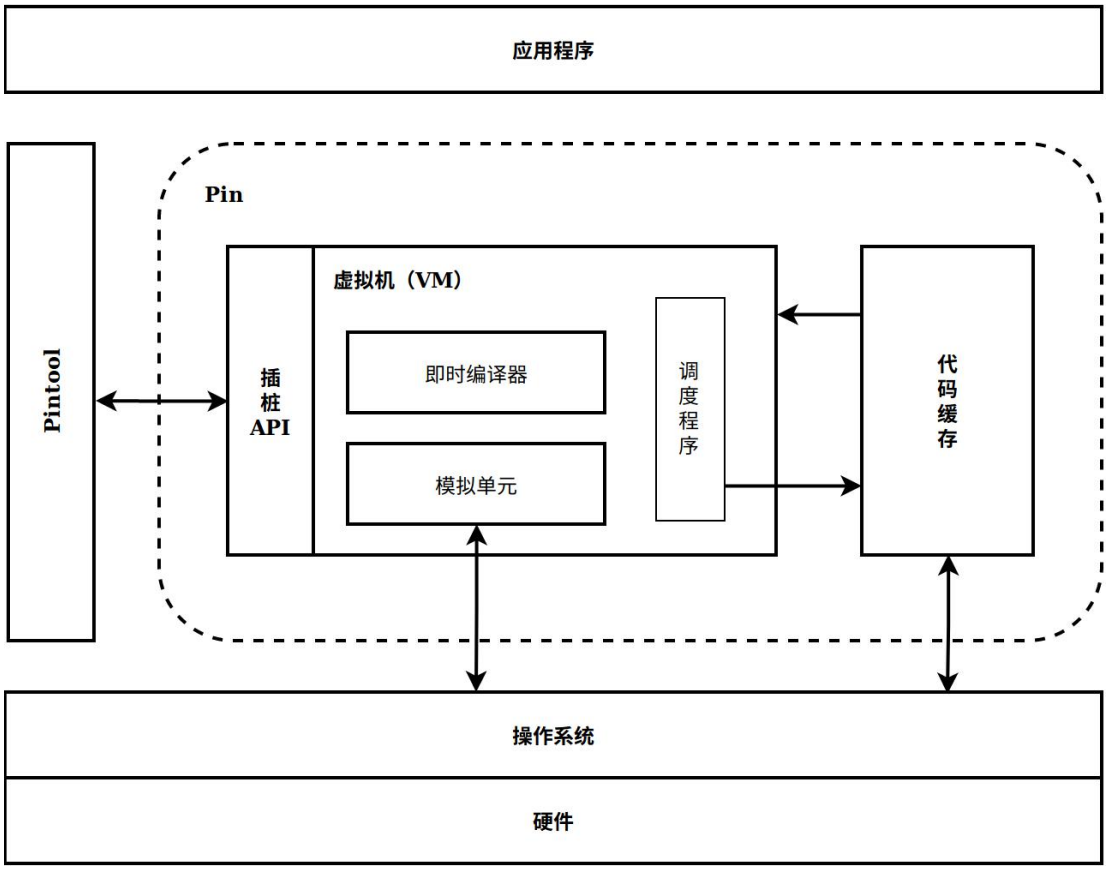


图 1-3 Pin 结构框架图

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。

1.4 本文主要研究内容

1.4.1 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）。

1.4.2 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）。

1.5 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。

第二章 ROP 攻击原理与流程

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）。

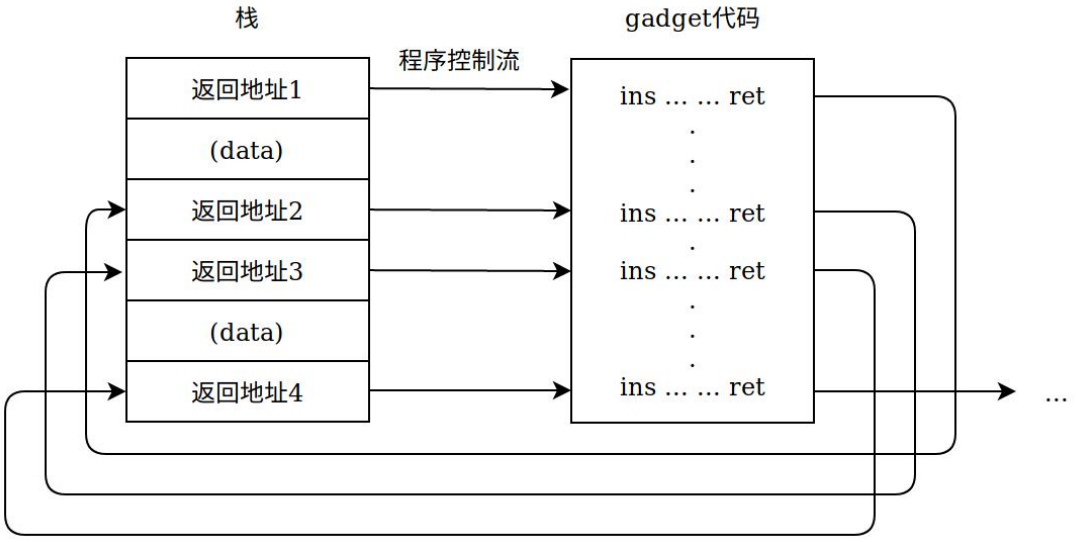


图 2-2 ROP 攻击流程

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。

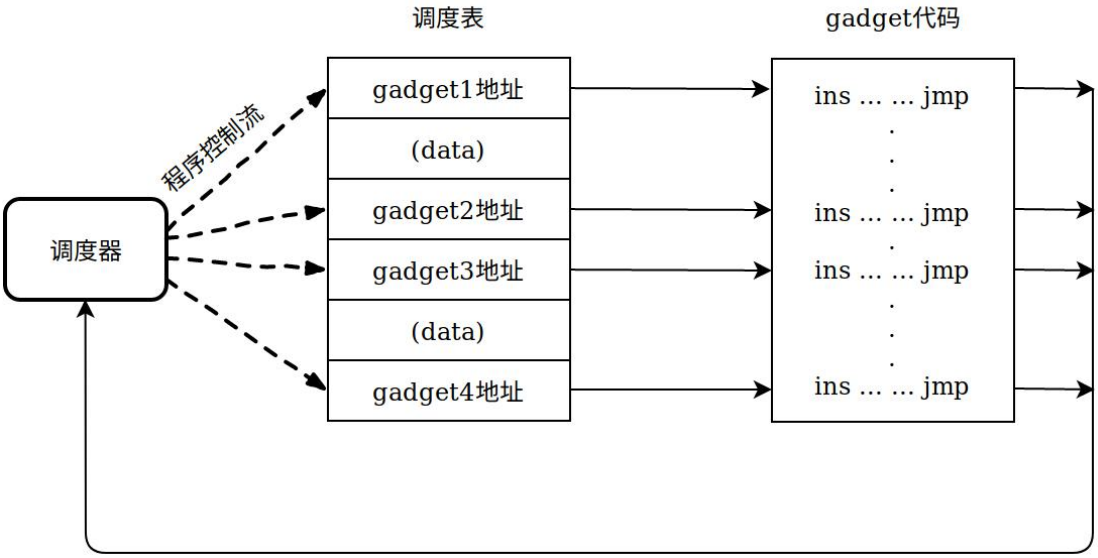
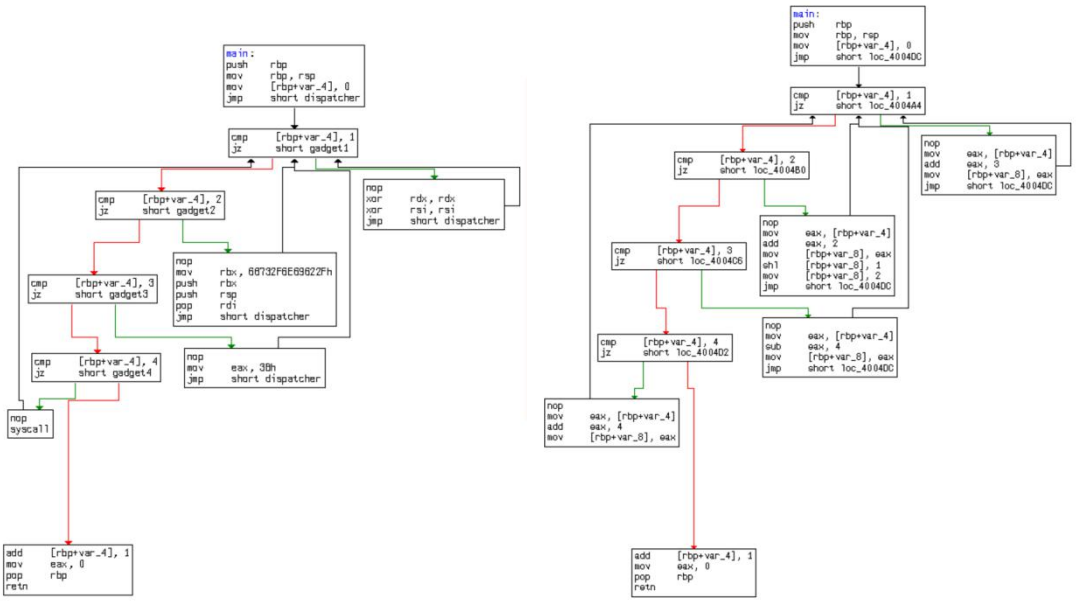


图 2-3 JOP 攻击流程

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。



(a) 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。

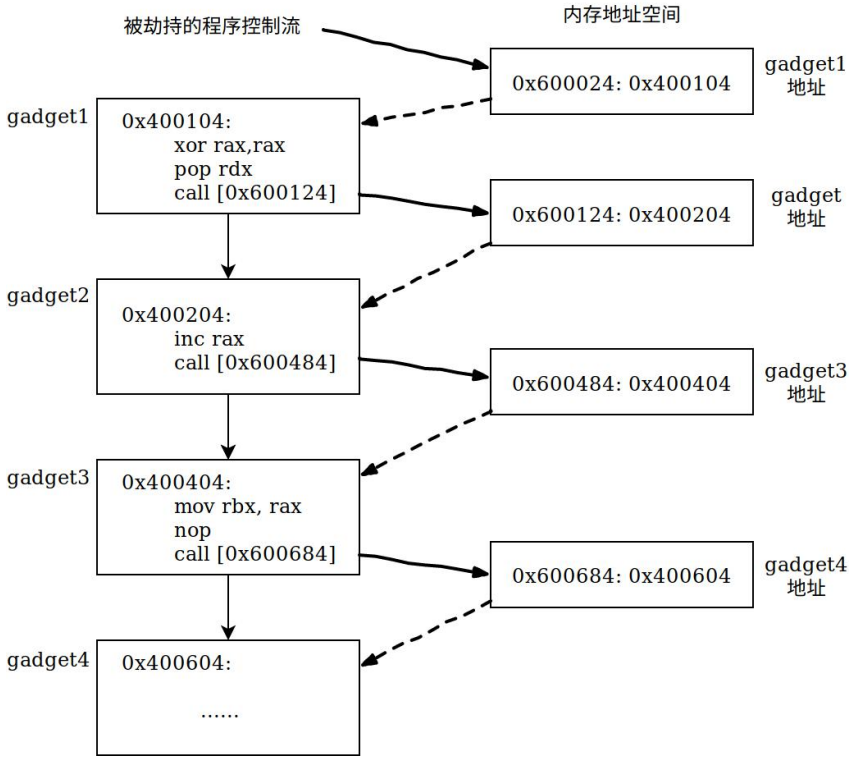


图 2-5 COP 攻击流程

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。

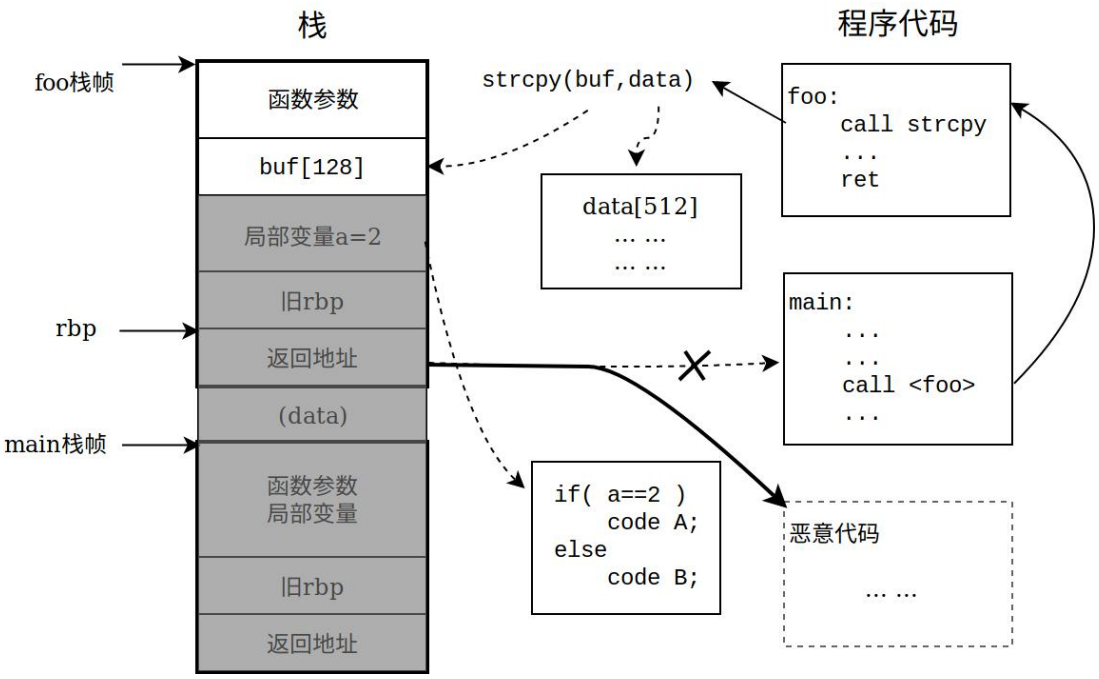


图 2-6 缓冲区溢出漏洞利用流程

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。

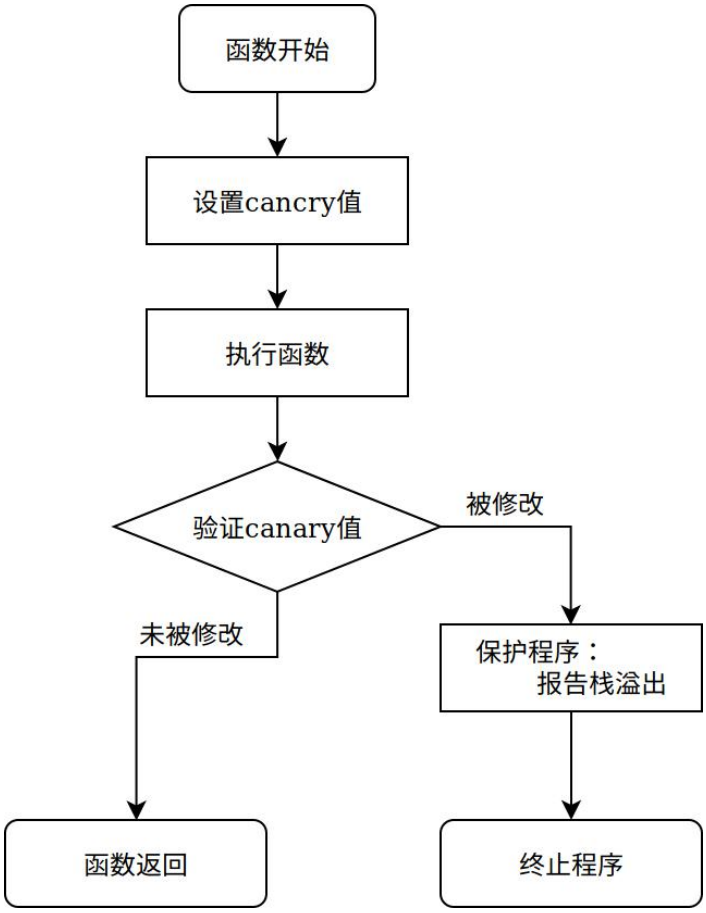


图 2-7 设置 canary 值的栈溢出防御机制

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。

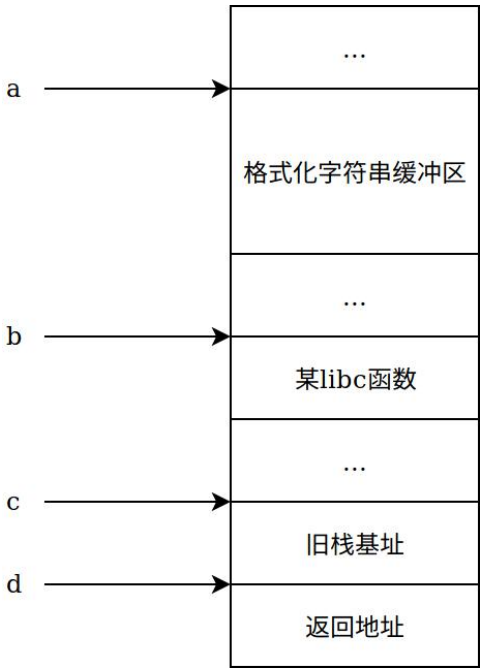


图 2-8 格式化字符串漏洞的利用

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（符号查找、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。

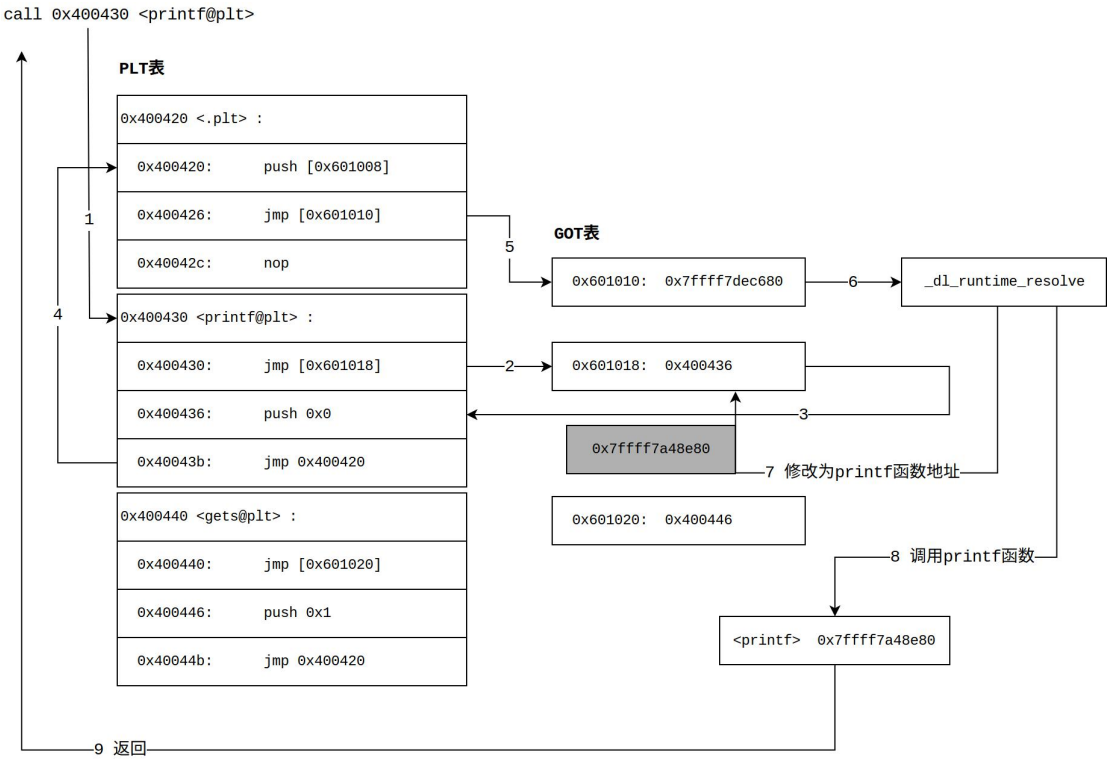


图 2-9(a) 延迟绑定流程之动态解析

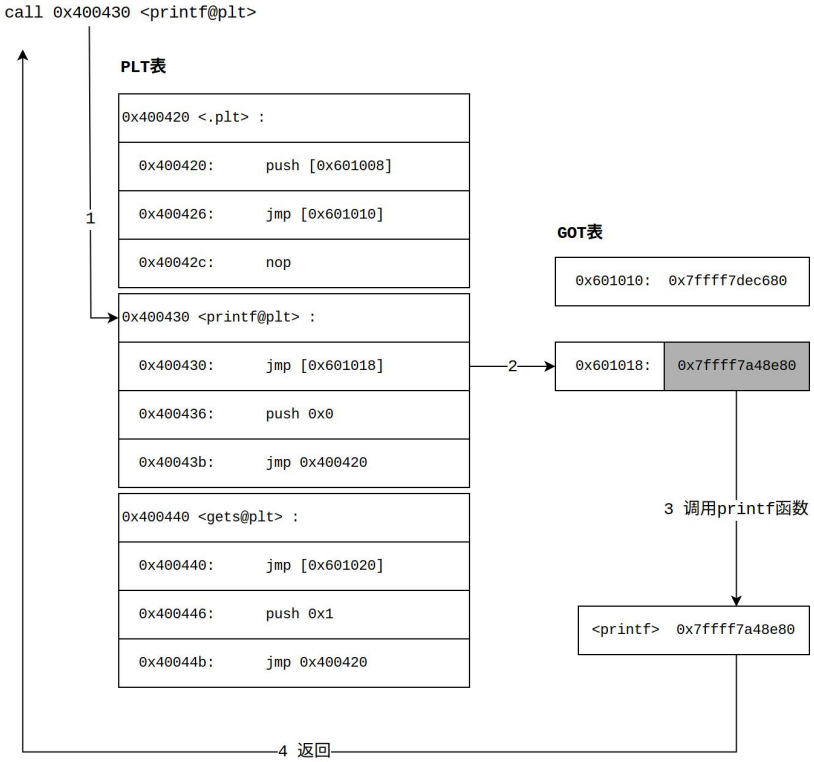


图 2-9(b) 延迟绑定流程之再次调用

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。

第三章 ROP 攻击动态特征

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、函数指针被篡改。

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。函数指针是一种特殊的内存指针，用于存储某函数的地址。间接调用或者间接跳转指令使用到了函数指针，即程序控制流转移的位置由函数指针指向的值决定。攻击者可以利用程序漏洞，如格式化字符串漏洞，篡改内存中函数指针中的值，在间接调用或间接跳转指令执行后，劫持程序控制流。由于栈指针也是内存指针的一种，返回地址被篡改和函数指针被篡改可以合并为内存指针被篡改，如图 3-1：栈指针在返回指令执行时，指向栈中保存的返回地址；函数指针在间接调用或间接跳转指令执行时，指向调用或跳转的目标地址。

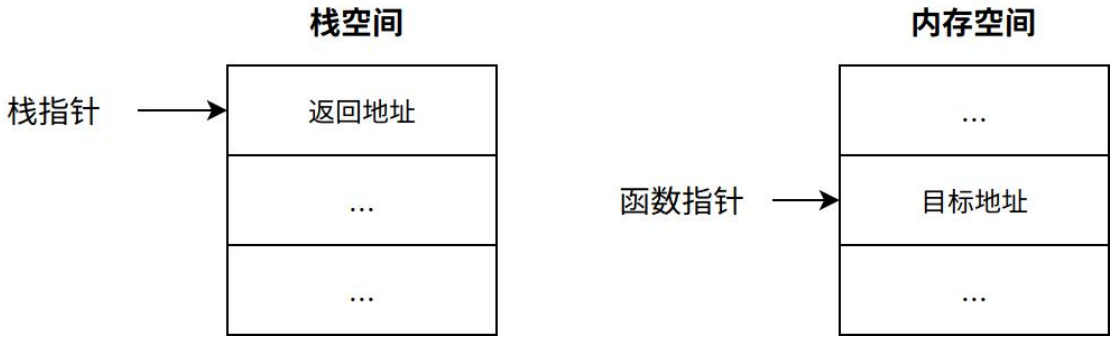


图 3-1 栈指针与函数指针的应用

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指针被篡改，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，可以将 ROP 攻击程序内存的动态特征分为：1.栈中返回地址被篡改。2.内存中函数指针的值被篡改。

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改或是内存中的函数指针的值被篡改。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。

第四章 ROP 攻击检测方法

4.1 指令特征检测

4.1.1 调用/返回指令数检测

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 展示了调用/返回指令数检测方案的流程。

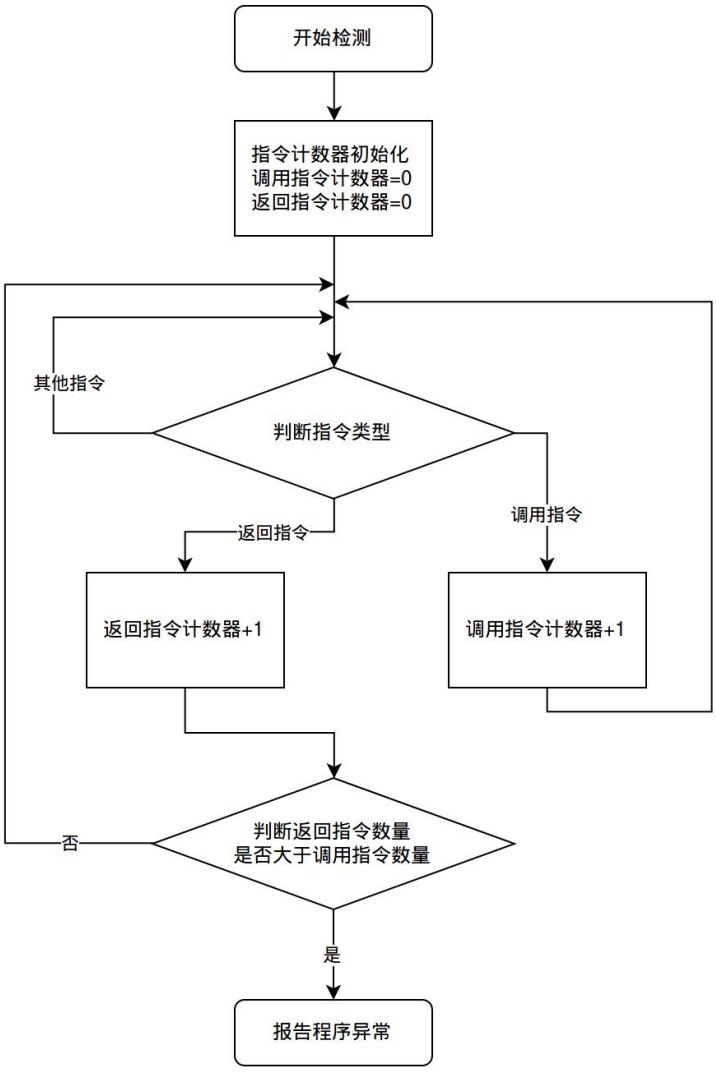


图 4-1 调用/返回指令数检测流程

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

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 内存完整性检测

4.2.1 返回地址完整性检测

攻击者如果想要利用返回指令完成程序控制流的劫持，必然会篡改栈中保存的函数返回地址。要想检测栈中的返回地址是否被修改，首先需要知道返回地址的正确值是多少，然后才能通过对比得知返回地址是否被修改。因此，返回地址的检测可分为两步：

1. 在返回地址被攻击者修改前，事先记录返回地址的正确值。
2. 在返回指令执行时，检查当前返回地址与事先记录的返回地址是否一致。

其中，如何对返回地址的正确值进行事先记录是返回地址检测的关键，因为如果第一步记录的返回地址的值已经被攻击者篡改，则第二步的对比检查操作将毫无意义。因此，为了保证记录的返回地址的正确性，需要在返回地址第一次出现在栈中时就进行记录。返回地址是在函数被调用时压入栈顶的，返回地址的值为调用指令的下一条指令地址。因此记录返回地址的工作应该在调用指令被执行后立即进行。检查返回地址与事先记录的返回地址是否一致的工作应该在返回指令执行前进行，如果一致，则说明程序正常执行，如果出现不一致，则说明程序异常，可能正在遭受 ROP 攻击。返回地址的检测流程如图 4-2 所示。

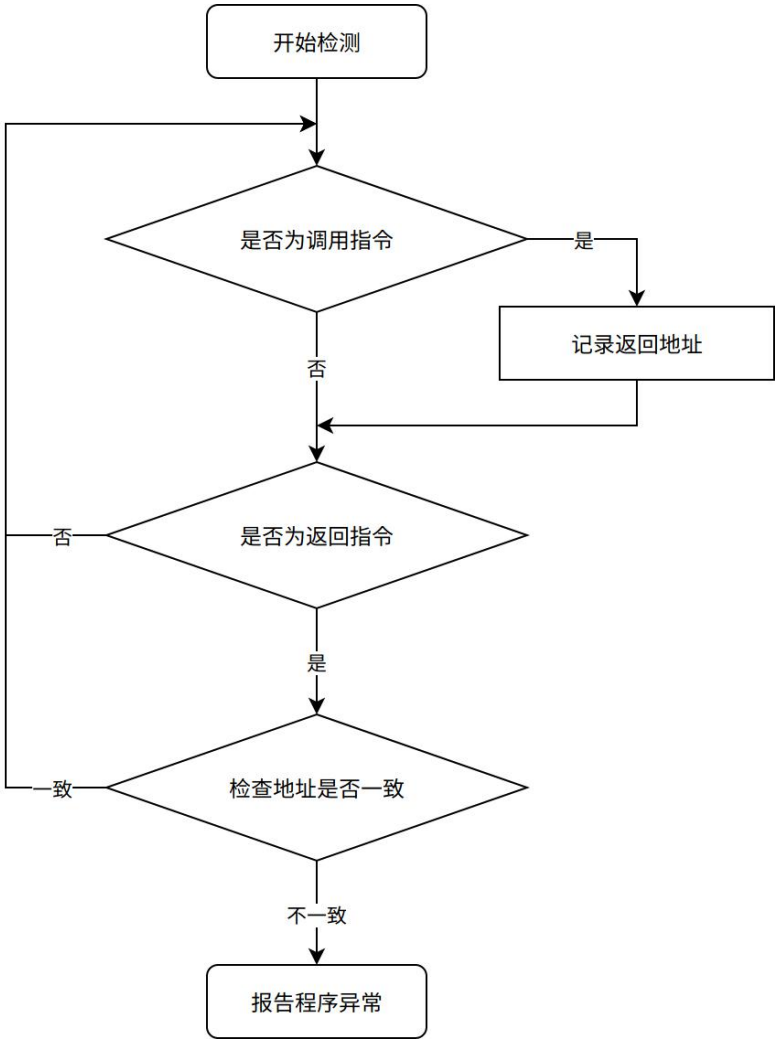


图 4-2 返回地址检测流程

此外，对程序的返回地址进行检查，不光可以检测出程序异常，还能根据被篡改的返回地址信息，识别不同类型的代码复用攻击。若返回地址被篡改为某 libc 函数，则程序可能正在遭受 return-into-libc 攻击；若返回地址被篡改为某个以返回指令为结尾的短指令序列的首地址，则程序可能正在遭受 ROP 攻击；若返回地址被篡改为某个以跳转指令为结尾的短指令序列的首地址，则程序可能正在遭受 JOP 攻击。

上述代码复用攻击以及攻击类型被检测出的前提是攻击者通过篡改栈中保存的返回地址发动攻击。返回地址存储于程序堆栈内部，因此返回地址检测的范围仅限堆栈内部，对于可以不依赖程序堆栈进行攻击的 JOP 攻击而言，返回地址检测无法对其检测。

4.2.1 函数指针完整性检测

程序控制流劫持的方式不仅限于篡改程序返回地址，攻击者通过篡改位于内存中的函数指针，也能够实现程序控制流的劫持。不同于其他的内存指针，函数指针专门用于间接调用或间接跳转。一般情况下，程序内存中存在函数指针表中，这些表的位置通常固定在程序数据段，具有可读可写的特点。常见的函数指针表有：用于程序动态链接重定位的 GOT 表和 C++语言中用于实现面向对象编程中多态特征的虚函数表。GOT 表和虚函数表的实质是指针数组，GOT 表中存储的是指向动态库函数的函数指针，虚函数表中存储的是指向虚函数的函数指针。函数指针的本质是指针变量，因此可以被攻击者篡改。由于程序中可能存在多种函数指针，因此需要对函数指针完整性的检测方法进行分类讨论：

（1）对于值在程序运行过程中不再修改的函数指针，如虚函数表中的函数指针，可以通过在程序未启动之前，或在函数指针指未被使用前，对函数指针中地址进行记录，在函数指针指向的函数被间接调用时，检查记录值与间接调用的目标地址是否一致。

（2）对于值在程序运行过程中只会修改一次的函数指针，如 GOT 表中的函数指针（进行延迟绑定的动态解析时修改一次，详见 2.3.2 节），由于在间接调用或间接跳转时，函数指针的值可能被修改，也可能未被修改，且无法确定函数指针的首次修改是正常修改还是被攻击者篡改。因此需要记录函数指针未被修改前的原有地址，以及函数指针被合法修改后的合法地址，在间接调用或间接跳转时，对函数指针的值进行检查，判断其是否为原有地址或合法地址。

（3）对于值在程序运行过程中会修改多次的函数指针，一般是程序员自定义的函数指针，由于难以预测程序员使用函数指针的次数与方式，为了不影响程序员对函数指针的正常使用，不能对这种函数指针进行检测。

函数指针的使用比较广泛，本节中提出的检测方法，只能对程序中存在的常见函数指针进行检测，对程序员自定义的函数指针则无法检测。C 语言的函数指针存在极大的安全隐患，因此程序员在编写程序的过程中，应该避免使用函数指针。

4.3 本章小结

本章根据第三章中总结的 ROP 攻击动态特征，提出了四种相应的 ROP 攻击检测方法。其中调用/返回指令数检测和连续 gadget 检测对应 ROP 攻击的指令特征，返回地址完整性检测和函数指针完整性检测对应 ROP 攻击的内存特征，完整性检测即检查返回地址或是函数指针是否被恶意修改。

调用/返回指令数检测方法的关键在于对程序执行的调用指令和返回指令进行计数，本章通过设置调用指令计数器与返回指令计数器的方法对指令进行计数，当返回指令计数器的值大于调用指令计数器的值时，报告程序异常。连续 gadget 检测方法的关键在于识别 gadget，本章提出设置阈值的方法，将长度小于阈值 T0 的短指令序列视作候选 gadget，当候选 gadget 连续执行的次数大于阈值 T1 时，报告程序异常。调用/返回指令数检测方法会受到程序溢出点的影响，连续 gadget 检测也存在一定程度的漏报和误报，且无法检测以跳转指令为结尾的 gadget。

返回地址完整性检测方法的关键在于事先记录函数返回地址的正确值，这样才能在函数返回时检查返回地址是否被篡改，本章通过调用指令执行后立即记录返回地址的方法，保证事先记录的返回地址的正确性。函数指针完整性检测方法的关键在于检测函数指针表的完整

性，对于不同类型的函数指针表，本章分类讨论并提出了不同的检测方法。返回地址完整性检测针对返回指令，而函数指针完整性检测针对调用或跳转指令。攻击者只能利用返回指令、调用指令或跳转指令劫持程序控制流。因此，将返回地址完整性检测和函数指针完整性检测结合起来，能够对攻击者的程序控制流劫持行为进行检测。

程序流劫持是 ROP 攻击的前提，内存完整性检测方法能够检测攻击者对程序控制流劫持行为，具有覆盖面广，准确度高的特点，因此可以作为主要的检测方法。指令特征的检测方法都存在一定的局限性，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。

第五章 ROP 攻击检测系统实现 5000+

5.1 定义与假设

为了方便说明本文实现的 ROP 攻击检测系统，本文做出如下假设：

1. 假设操作系统开启了 DEP 保护机制，DEP 保护使可写的内存不具有可执行权限，因此攻击者无法通过注入代码进行攻击。现代操作系统默认开启 DEP 保护机制。
2. 假设操作系统开启了粗粒度的 ASLR 保护机制，粗粒度 ASLR 为程序的共享库映像与堆栈随机分配基址，因此攻击者无法直接 libc 中函数的地址。现代操作系统默认开启粗粒度 ASLR 保护机制。
3. 假设程序存在缓冲区溢出漏洞或者格式化字符串漏洞，控制流可以被劫持。
4. 假设攻击者有能力利用程序漏洞，完成内存数据泄露，劫持程序控制流，开启系统 shell 等攻击行为。
5. 假设操作系统未开启栈溢出保护机制。栈溢出的保护机制在 2.2.1 节有详细介绍，该技术与文本讨论的 ROP 攻击检测方法无关。
6. 假设程序的源代码无法被访问。

5.2 系统概述

本文提出的 ROP 攻击检测系统由二进制插桩检测框架及工具和网页服务器两部分组成。其中，二进制插桩检测工具在 PIN 框架下实现，用于对 ROP 攻击进行检测，包含信息获取模块、数据存储模块、攻击检测模块、攻击识别模块和日志报告模块。网页服务器用于提供交互界面，包含前端界面和后端服务。ROP 攻击检测系统的结构图，如图 5-1 所示：

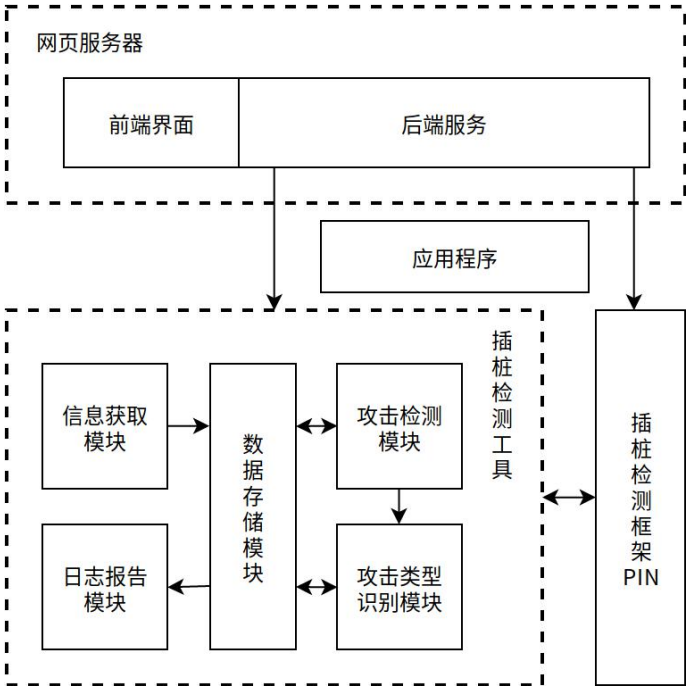


图 5-1 ROP 攻击检测系统结构图

二进制插桩检测工具是 ROP 攻击检测系统的核心，其各个模块的作用如下：信息获取模块用于获取程序运行时的各类信息。数据存储模块用于存储检测工具运行时所需的各类数据。攻击检测模块用于检测程序的运行行为是否符合 ROP 攻击的动态特征。该模块是整个 ROP 攻击检测系统的关键，将在 5.4 节中详细介绍。攻击识别模块用于识别攻击类型。日志报告模块用于记录攻击信息，便于程序员后续对软件进行维护。二进制插桩检测工具各个模块之间的关系如下：信息获取模块获取信息后将信息保存在数据存储模块中。数据存储模块为其他模块提供数据，是连接各个模块的中介。攻击识别模块在攻击检测模块检测到攻击后启动，将识别到的攻击信息保存在数据存储模块中。日志报告模块根据攻击识别模块提供的数据，生成攻击报告。

网络服务器的前端界面具有两个功能，其一是让用户选择待运行的应用程序与待开启的 ROP 攻击检测方法，其二是展示 ROP 攻击检测工具的运行效果。根据用户选择的应用程序与 ROP 攻击检测方法，网络服务器的后端服务将开启 ROP 攻击检测工具对应用程序进行监测，并将应用程序映射到网络端口，供其他机器进行远程访问。

ROP 攻击检测系统的工作流程如图 5-2 所示。

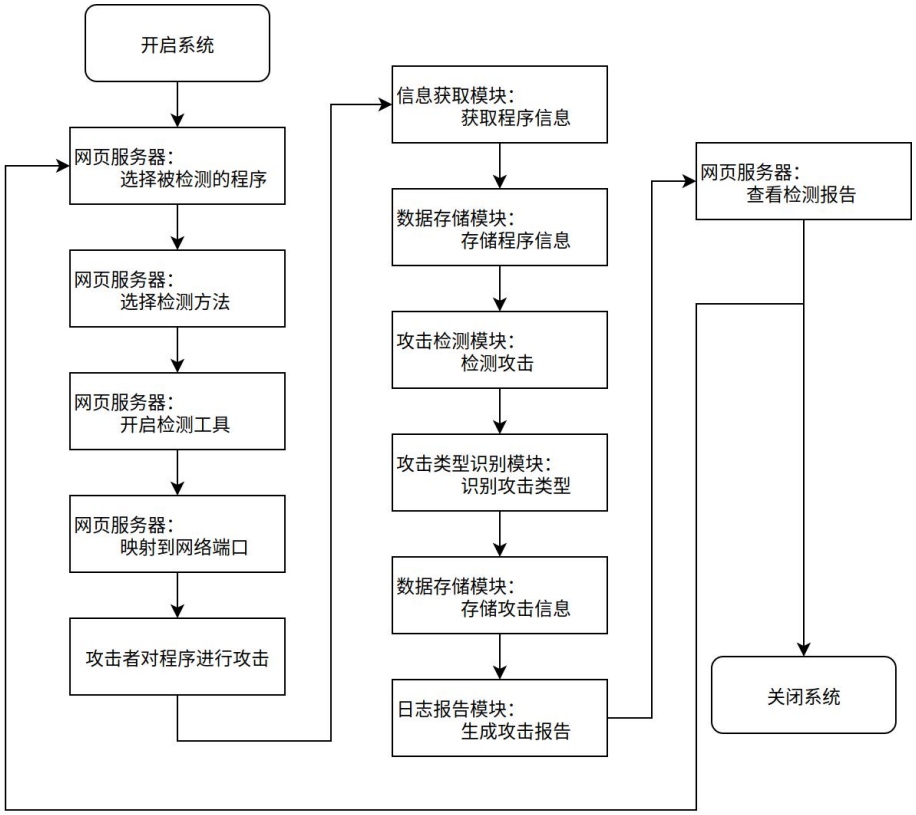


图 5-2 ROP 攻击检测系统的工作流程

5.3 设计思想

系统核心的 ROP 攻击检测工具采取了模块化的设计思想。信息获取模块、数据存储模块、攻击检测模块、攻击识别模块和日志报告模块相互独立，对检测方法进行改进或添加新的检测方法，不会对其他模块的功能造成影响。如此设计便于 ROP 攻击检测系统的升级与拓展。

在核心的检测工具之外加入网页服务器，是为了方便进行系统测试。网页服务器的设计是基于 B/S 模式的设计思想。网页服务器提供了一个完整的测试平台，在这个平台下，测试者可以针对不同的应用程序，选择不同的 ROP 攻击检测方法，在运行程序后还能够查看检测结果。如此设计能够让测试者对本文中的 ROP 攻击检测系统进行更加便捷的测试。

将应用程序映射到网络端口是基于 C/S 模型的设计，目的是模拟现实中运行在服务器各个端口上的网络服务程序，供其他客户端对程序进行访问。常见的网络服务程序有网页服务程序 Apache（默认端口 80）、用于远程连接的 OpenSSH 服务（默认端口 22）、用于邮件接受的 IMAP 服务（默认端口 143）等。现实中的 ROP 攻击通常是通过网络发起的，对网络服务程序进行 ROP 攻击检测的设计更接近于现实情况。

5.4 实现细则

本节将介绍 ROP 攻击检测系统的关键部分，即 ROP 攻击检测工具的核心模块的具体实现。攻击检测模块用于判断程序是否正在遭受 ROP 攻击，检测模块的高效实现，决定着整个 ROP 攻击检测系统的效率与准确度，因此是核心模块。本节将依据本文第三章中讨论的 ROP 攻击动态特征，第四章中提出的 ROP 检测方法，详细介绍六种在 PIN 框架下实现的 ROP 攻击检测的具体方案。

5.4.1 Return-into-libc 检测

Return-into-libc 攻击存在明显的攻击特征，因此可以单独提出一种方案，对 Return-into-libc 进行准确、高效的检测。正常程序中，返回地址为调用指令下一条地址，调用指令总是在函数中部出现，因此返回地址不可能是任何函数的起始地址。Return-into-libc 攻击因为要复用整个函数，攻击时会将返回地址篡改为 libc 中某函数的起始地址，这与“正常程序中，返回地址不可能是任何函数的起始地址。”这一基本事实矛盾。因此只需对程序的返回指令进行插桩，判断其目标地址是否为 libc 中函数的地址，即可检测 Return-into-libc 攻击。

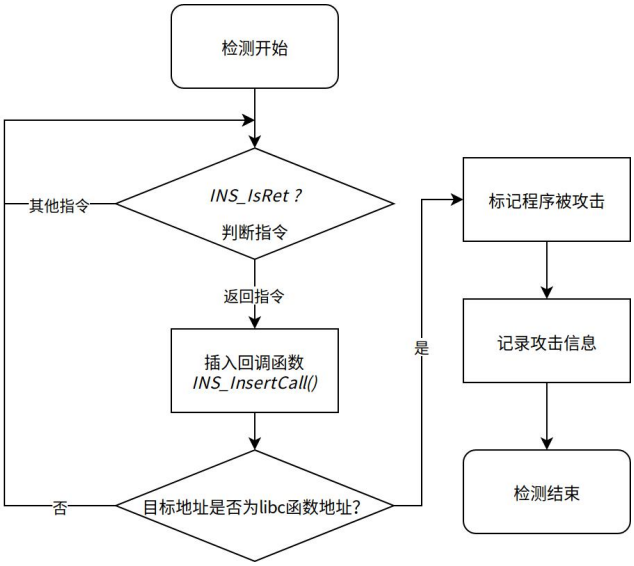


图 5-3 Return-into-libc 检测方案流程图

Return-into-libc 检测方案的流程如图 5-3 所示。在指令级别的插桩函数中，可以调用 *INS_IsRet(ins)*判断返回指令，然后调用 *INS_InsertCall()*插入回调函数进行返回地址与 libc 中各函数地址的比对工作。libc 中各函数地址的记录工作在信息获取模块中完成。信息获取模块在映像级别的插桩函数中，通过调用 *IMG_Name(img)*获取映像名，识别 libc 映像，然后通过 *IMG_RegsymHead(img)*获取 libc 的符号表，最后调用 *SYM_Name(sym)*，*SYM_Address(sym)* 提取符号表中各个函数的名称及地址。

5.4.2 阈值检测器

通过设置阈值对 gadget 进行识别是一种极高效的 ROP 攻击检测方案，阈值检测器的额外系统开销很小，但代价是检测的准确度不高，且有可能带来误判。根据本文 4.1.2 节的讨论，阈值检测器只考虑以返回指令和跳转指令为结尾的 gadget。在实现阈值检测器的过程中，通过对 libc 中可用 gadget 的长度以及大量 ROP 攻击程序中 gadget 的数量进行统计，本文最终设定限制 gadget 长度的阈值 T0 为 7，限制连续 gadget 执行次数的阈值 T1 为 4。阈值检测器利用指令计数器和 gadget 计数器实现。指令计数器在每条指令执行后加一，当执行到返回或调用指令时，如果指令计数器的值小于阈值 T0，则将 gadget 计数器的值加一，如果指令计数器的值大于阈值 T0，则说明这是正常指令序列，清零指令计数器和 gadget 计数器。当 gadget 计数器的值大于阈值 T1 时，报告攻击。

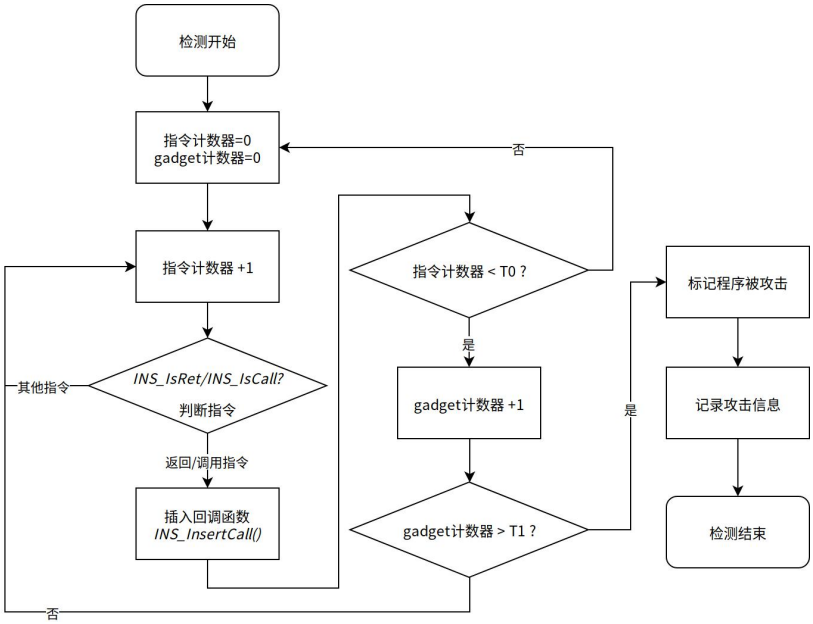


图 5-4 阈值检测方案流程图

阈值检测器的流程如图 5-4 所示。其中，指令的判断通过 *INS_IsRet(ins)*和 *INS_IsCall(ins)*实现，计数器和阈值的对比通过插入的回调函数实现。为了防止误报，本文阈值设置的比较保守，如果只开启阈值检测器，则检测的准确度不高，因此，阈值检测器可作为辅助检测方案，需要与其他检测方案配合使用。

5.4.3 调用/返回指令计数器

传统的 ROP 攻击复用了多个以返回指令为结尾的 gadget，因此会导致处理器执行的返回指令数量多于调用指令数量，通过设置调用指令计数器与返回指令计数器，在函数返回时判断返回指令数是否大于调用指令数，可以有效的检测 ROP 攻击。该方案只需要对比两个计数器的值，是一种极高效的 ROP 攻击检测方案，而且对于传统的 ROP 攻击，检测的准确度高。该方案存在一定的局限性，受到溢出位置的影响（详见 4.1.1），可能存在漏报，但是绝对不会出现误报。

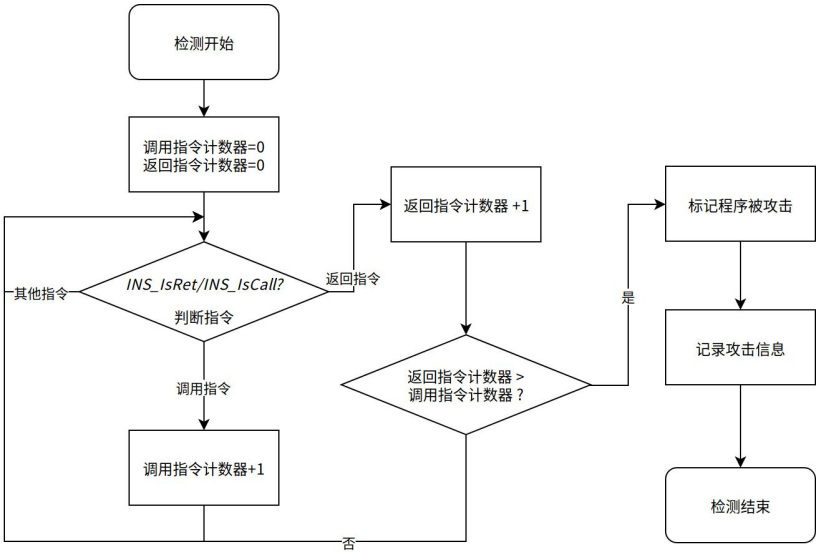


图 5-5 调用/返回指令计数器流程图

调用/返回指令计数器的流程如图 5-5 所示。调用指令和返回指令计数器的初始值为 0，在指令级别的插桩函数中，通过 *INS_IsCall(ins)*识别调用指令后调用指令计数器的值加一，通过 *INS_IsRet(ins)*识别返回指令后返回指令计数器的值加一，然后立即判断返回指令计数器的值是否大于调用指令计数器的值，若是，则报告程序被攻击。

5.4.4 影子栈

影子栈是一种基于 ROP 攻击内存动态特征的检测方案，可以弥补 5.4.2 和 5.4.3 节中基于 ROP 攻击指令特征的检测方案的局限性，但是需要更多的额外信息，效率较低。影子栈是程序堆栈的简化，影子栈只保存程序的返回地址。本文引入栈存储结构，在调用指令执行前，通过 *STK_Push()*将返回地址压入影子栈中，在返回指令执行前，通过 *STK_Pop()*将栈顶元素出栈，并与返回指令的目标地址进行对比，如果一致，则说明程序正常，如果不一致，则说明程序遭受到了攻击。上述的影子栈操作在返回指令和调用指令的回调函数中进行。影子栈检测方案的流程图如图 5-6 所示。

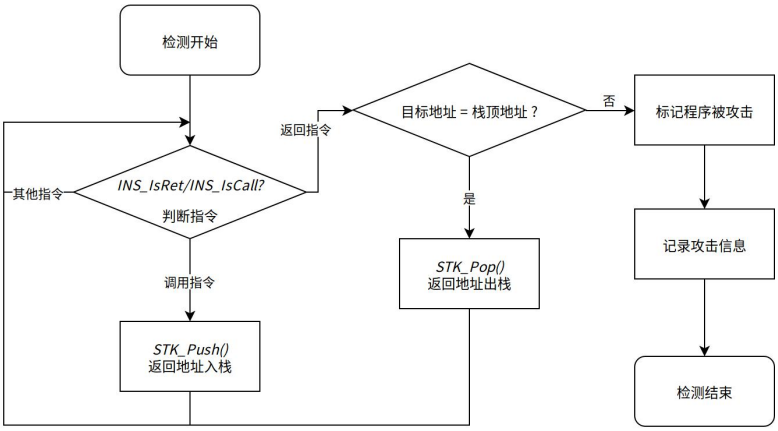


图 5-6 影子栈检测方案流程图

由于多数 libc 函数的内部存在多重嵌套调用，且存在非对称的调用返回关系[23]，对 libc 进行跟踪极其浪费系统资源，为了提高检测效率，影子栈的检测范围仅为程序代码段，即影子栈只能对所有的用户函数返回地址被篡改进行检测。如果 libc 中存在漏洞，导致 libc 函数的返回地址被篡改，影子栈则不能检测。此外，影子栈的实质是程序调用栈的备份，因此影子栈无法检测不依赖栈的 ROP 攻击。

5.4.5 CPR 检测器

先前调用返回（Call-Preceded Return, CPR）指的是返回指令的目标地址是总是调用指令的下条指令的地址。在所有可用的 gadget 中，只有 6%的 gadget 是 call-preceded gadget[22]。因此，除非攻击者使用的 gadget 均为 call-preceded gadget，通过检查返回地址是否为先前调用返回地址，可以检测绝大部分 ROP 攻击。先前调用返回地址的记录在调用指令执行前完成，通过调用 *STK_Push()*将调用指令的下条地址保存在 *cpra* 栈中。返回地址的检查则在返回指令执行前完成，通过调用 *STK_Search()*，从 *cpra* 栈顶至栈底依次查找返回指令的目标地址，若查找到，说明该返回地址是先前调用返回地址，若查找不到，则说明该返回地址指向非法的 gadget，程序遭到了 ROP 攻击。CPR 检测器的流程如图 5-7 所示。

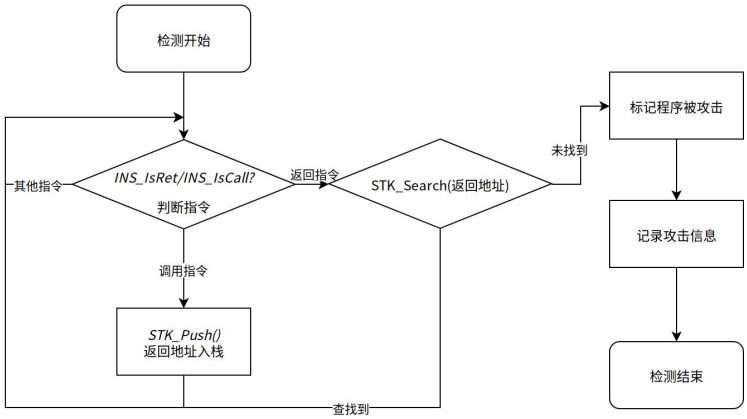


图 5-7 CPR 检测器流程图

CPR 检测器的检测范围是所有程序执行的指令，因此可以弥补影子栈不能检测 libc 的不足。cpra 栈可以视为一张顺序表，采用从栈顶至栈底依次查找的查找方式，查找效率高，且对 cpra 的查找不受 libc 函数中存在的非对称调用返回关系[23]的影响，因此无需对这种非对称调用返回关系进行额外处理。

5.4.6 GOT 表篡改检测

攻击者对程序的 GOT 表进行修改，也可以实现程序控制流的劫持，由于没有篡改返回地址，5.4.4 和 5.4.5 的基于检查返回地址的检测方案都将失效。GOT 表作为每个动态连接的程序中都存在的函数指针表，容易被攻击。GOT 表中函数指针的合法值只有两种，一种是初始时该函数的 PLT 表的第二条指令地址，位于程序代码段中，另一种是该函数经动态解析后的在 libc 中的实际地址，位于 libc 映像中。GOT 篡改检测方案，以程序符号表中的函数名为媒介，在函数被调用时，对其 PLT 表的第一条指令（间接跳转指令）的目标地址进行检查，若是合法地址，则说明程序正常，若不是则说明 GOT 表被非法篡改，程序正在遭受攻击。GOT 表篡改检测方案的流程如图 5-8 所示。

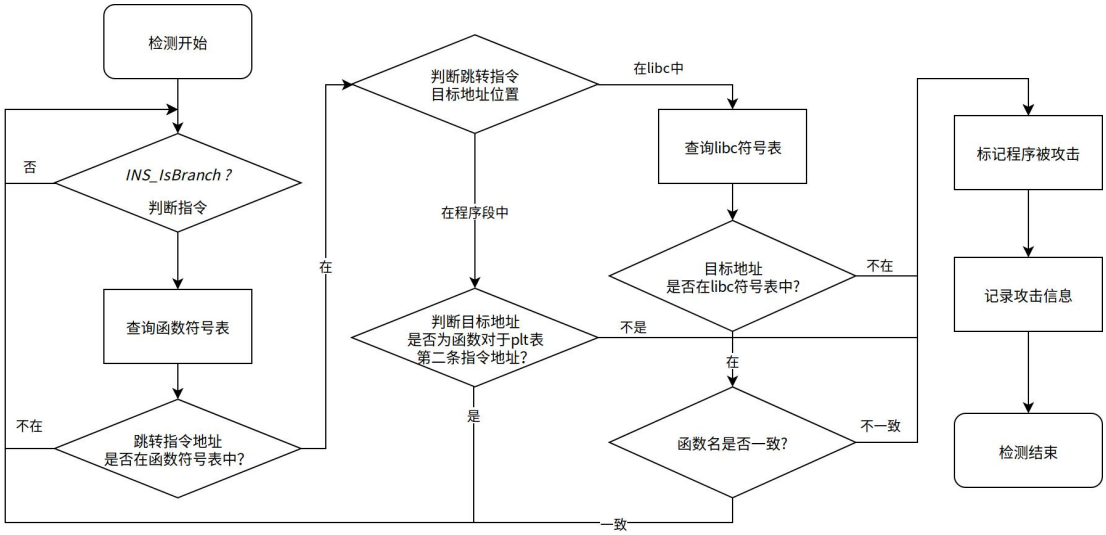


图 5-8 GOT 表篡改检测方案流程图

其中，信息获取模块通过映像级别插桩，在程序和 libc 映像加载时获取其地址范围，通过调用 *IMG_RegsymHead(img)*获取程序和 libc 的符号表。在程序符号表中，各函数的地址为其 PLT 表的首地址，PLT 表中的首条指令即为间接跳转指令。因此，通过调用 *INS_IsBranch(ins)* 对跳转指令进行识别，在回调函数中判断跳转指令的当前地址是否为在函数的符号表中，若

是，则说明该指令是间接跳转指令，于是对该指令的目标地址进行检查。若目标地址在程序地址范围内，则检查跳转指令的目标地址是否为该跳转指令的下一条指令的地址；若目标地址在 libc 映像地址范围内，则在 libc 的符号表中查找该地址，若未找到或找到的地址对于的函数名与该指令对于函数的函数名不一致，则说明程序遭到了攻击。

5.5 实验与测试

5.5.1 实验概述

本文实现的 ROP 攻击检测系统运行在 64 位 linux 服务器上，其发行版本为 Ubuntu 18.04，服务器默认在 8080 端口开启了网页服务，即为 ROP 攻击检测系统的测试平台。测试用的客户端为 64 位的 Ubuntu 19.04。服务器与客户端处于同一网段，客户端可以正常访问服务器。实验中使用到的程序及版本如下：python 3.6.7，gcc 8.2.0，socat 1.7.3.2，pin 3.7，django 2.2.1，pwntools 3.12.2，ROPGadget 5.4。

为了便于实验，实验中的漏洞程序均预设了泄漏 libc 函数地址的漏洞，其中程序 bof 还含有缓冲区漏洞，程序 fsb 还含有格式化字符串漏洞，可让攻击者篡改 GOT 表。

以开启漏洞程序 bof 和 Return-into-libc 检测方案为例（如图 5-10），本文进行实验时，首先在测试界面中选择待开启的漏洞程序和检测方案，然后点击 start 按钮开启服务，程序被映射到了 1101 端口上。攻击者能够访问位于 192.168.191.128 主机 1101 端口的漏洞程序。攻击结束后，点击 result 按钮可以查看检测结果。

图 5-10 开启 bof 程序和 Return-into-libc 检测方案的测试平台界面

5.5.2 Return-into-libc 攻击与检测

在 intel x86 64 位处理器架构中，函数通过 rdi，rsi，rdx 三个寄存器存储被调函数的前三个参数，因此完成 Return-into-libc 攻击需要借助一条如图 5-10 的 gadget 来完成函数参数的配置。

```
0x0040062c: pop rdi ; pop rsi ; pop rdx ; ret
```

图 5-11 Return-into-libc 攻击使用的 gadget

将参数配置完毕，通过缓冲区溢出漏洞将函数返回地址篡改为 execve 函数，函数返回时将执行 execve("/bin/sh",0,0)，开启了一个 shell。攻击结果（右侧）和检测结果（左侧）如图 5-12 所示，攻击被 Return-into-libc 检测策略检测到，并且识别出了 Return-into-libc 攻击。

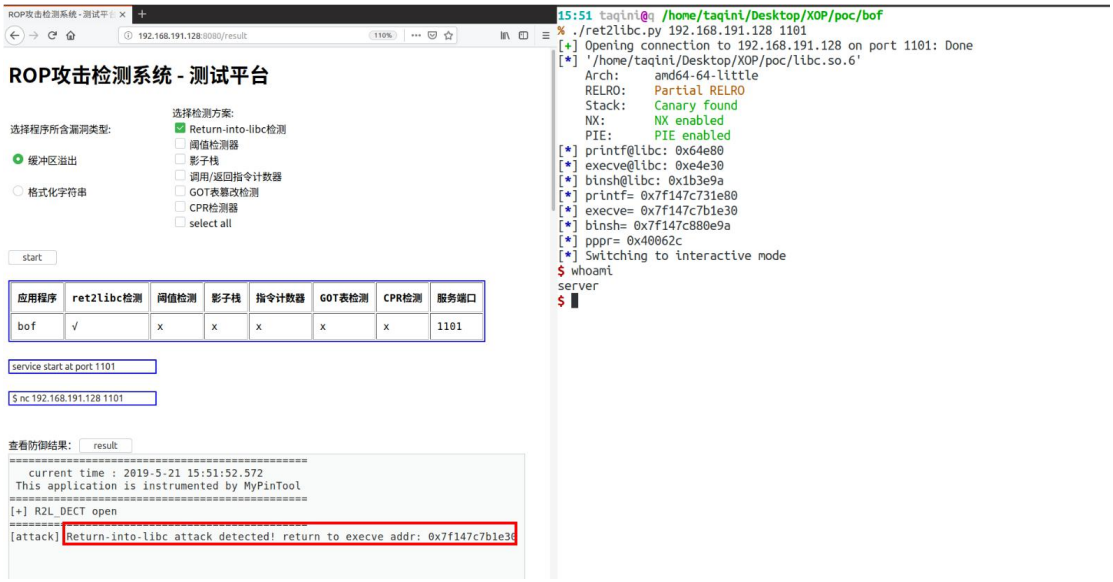


图 5-12 Return-into-libc 攻击及检测结果

为了绕过 Return-into-libc 检测方案，使用如图 5-13 的以 call 指令为结尾的 gadget 调用 libc 函数，代替以 ret 指令为结尾的 gadget，可以实现一次 Return-into-libc 的变种攻击，即 Call-into-libc 攻击。攻击结果和检测结果如图 5-14 所示，变种攻击成功开启了一个 shell，但是变种攻击并没有被 Return-into-libc 检测策略检测到。

```
0x00000000000439c8: pop rax; ret;
0x00000000001396be: mov rdx, r15; mov rsi, r14; mov rdi, r13; call rax;
```

图 5-13 Return-into-libc 变种攻击使用的 gadget

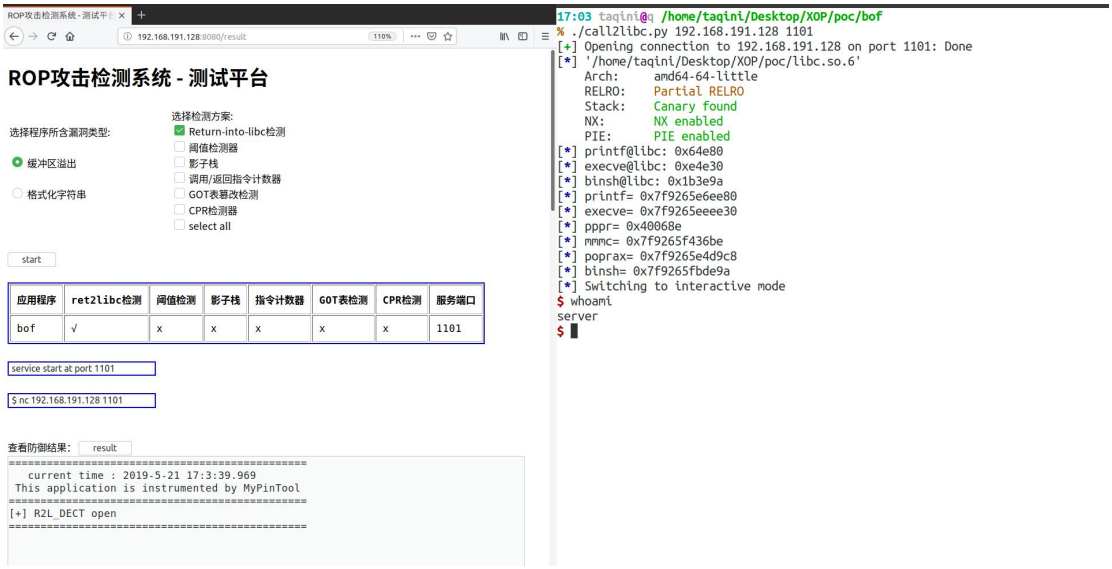


图 5-14 Return-into-libc 变种攻击及检测结果

为了检测 Return-into-libc 的变种攻击，选择开启影子栈检测策略，并再次进行 Return-into-libc 的变种攻击。攻击结果和检测结果如图 5-15 所示，变种攻击被新开启的影子栈方案检测到，并且识别出了 Call-into-libc 攻击。

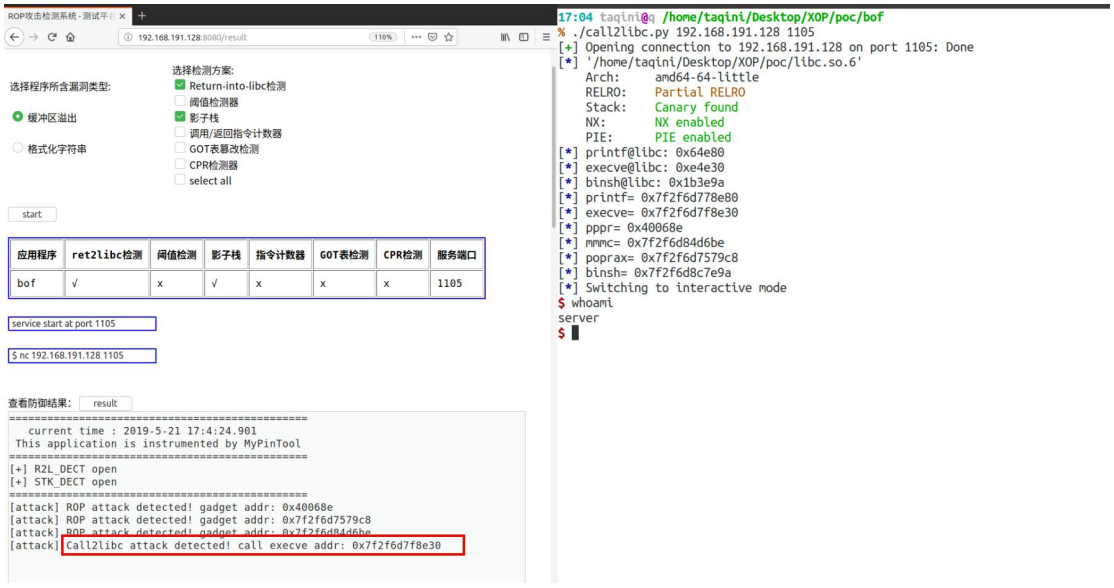


图 5-15 影子栈方案对 Return-into-libc 变种攻击的检测结果

5.5.3 ROP 攻击防御与检测

首先开启阈值检测方案，并对漏洞程序进行传统的 ROP 攻击，攻击使用的 gadget 如图 5-16 所示。攻击程序通过 gadget 完成了寄存器的布局（rax=0x3b,rdi='/bin/sh',rdx=0,rsi=0），最后调用通过系统调用 syscall，调用了 SYS_execve，执行/bin/sh，开启一个系统 shell。攻击结果和检测结果如图 5-17 所示，由于 ROP 攻击所用的 gadget 符合阈值 T0、T1，因此被阈值检测方案识别到，并识别出 ROP 攻击。

```
gadget1:
    0x0040068e      415d            pop r13
    0x00400690      415e            pop r14
    0x00400692      415f            pop r15
    0x00400694      c3              ret
gadget2:
    0x001396be      4c89fa          mov rdx, r15
    0x001396c1      4c89f6          mov rsi, r14
    0x001396c4      4c89ef          mov rdi, r13
    0x001396c7      ffd0            call rax
gadget3:
    0x001306d9      5a              pop rdx
    0x001306da      5e              pop rsi
    0x001306db      c3              ret
```

```
gadget4:
    0x000439c8      58              pop rax
    0x000439c9      c3              ret
gadget5(3):
    0x001306d9      5a              pop rdx
    0x001306da      5e              pop rsi
    0x001306db      c3              ret
gadget6:
    0x001306d7      0f05            syscall
```

图 5-16 传统 ROP 攻击使用的 gadget

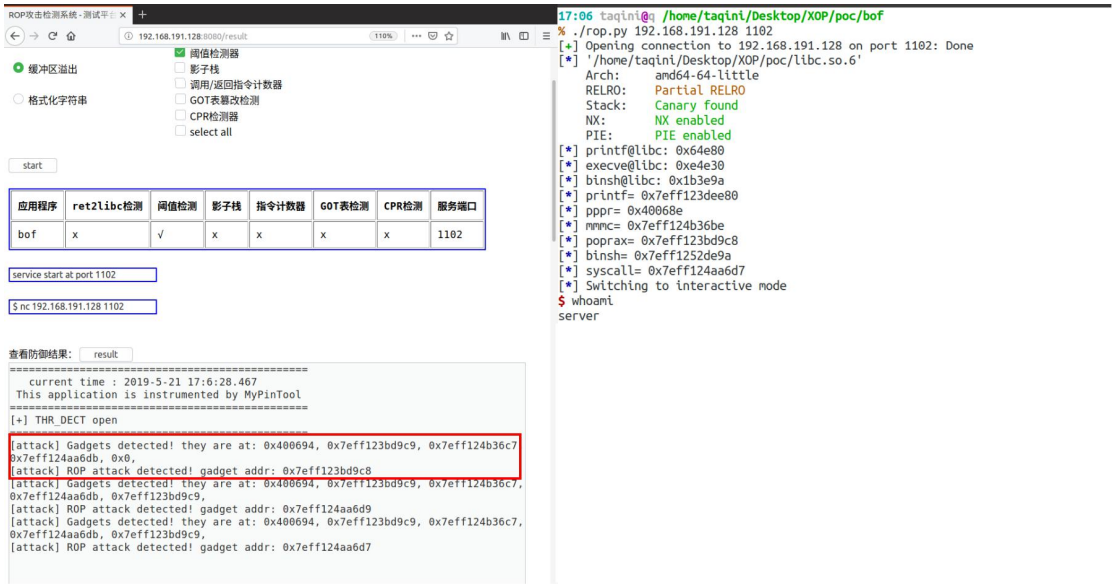


图 5-17 阈值检测方案对传统 ROP 攻击的检测结果

为了绕过阈值检测方案，可以修改攻击代码，将 gadget1 长度加长至 8，突破 T0，并将其插入到 gadget4 和 gadget5 之间，防止连续 gadget 长度超多阈值 T1。修改后的 ROP 攻击代码如图 5-18 所示。使用修改后的攻击代码再次对漏洞程序进行 ROP 攻击。攻击结果和检测结果如图 5-19 所示，由于修改后的 ROP 攻击所用的 gadget 绕过了阈值 T0、T1，因此没有被阈值检测方案识别到。

```
long gadget(G_size=8>=T0,S_length=0<=T1):
    0x00400686      4883c408        add rsp, 8
    0x0040068a      5b              pop rbx
    0x0040068b      5d              pop rbp
    0x0040068c      415c            pop r12
    0x0040068e      415d            pop r13
    0x00400690      415e            pop r14
    0x00400692      415f            pop r15
    0x00400694      c3              ret
gadget2(G_size=4<=T0,S_length=1<=T1):
    0x001396be      4c89fa          mov rdx, r15
    0x001396c1      4c89f6          mov rsi, r14
    0x001396c4      4c89ef          mov rdi, r13
    0x001396c7      ffd0            call rax
gadget3(G_size=3<=T0,S_length=2<=T1):
    0x001306d9      5a              pop rdx
    0x001306da      5e              pop rsi
    0x001306db      c3              ret
```

```
gadget4(G_size=2<=T0,S_length=3<=T1):
    0x000439c8      58              pop rax
    0x000439c9      c3              ret
long gadget(G_size=8>=T0,S_length=0<=T1):
    0x00400686      4883c408        add rsp, 8
    0x0040068a      5b              pop rbx
    0x0040068b      5d              pop rbp
    0x0040068c      415c            pop r12
    0x0040068e      415d            pop r13
    0x00400690      415e            pop r14
    0x00400692      415f            pop r15
    0x00400694      c3              ret
gadget5(G_size=3<=T0,S_length=1<=T1):
    0x001306d9      5a              pop rdx
    0x001306da      5e              pop rsi
    0x001306db      c3              ret
gadget6:
    0x001306d7      0f05            syscall
```

图 5-18 ROP 攻击使用的加长 gadget

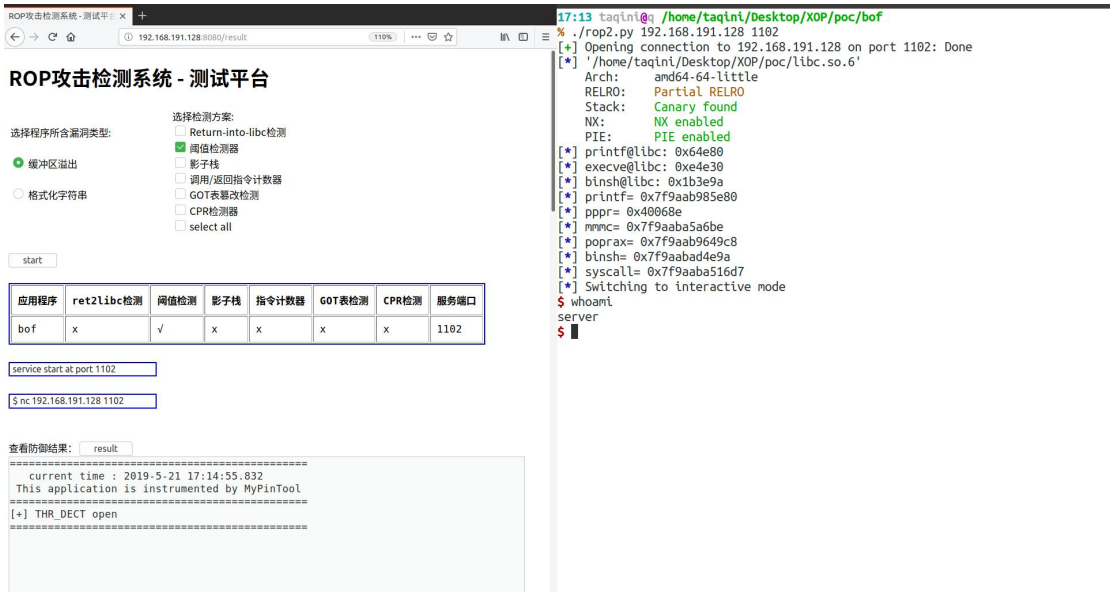


图 5-19 使用长 gadget 的 ROP 攻击及检测结果

阈值检测方案已经被绕过，因此开启影子栈检测方案，依然使用加长的 gadget 进行 ROP 攻击，攻击结果和检测结果如图 5-20 所示，ROP 攻击由于使用了位于程序代码的 gadget，被影子栈检测方案检测到，并识别出了攻击类型为 ROP 攻击。

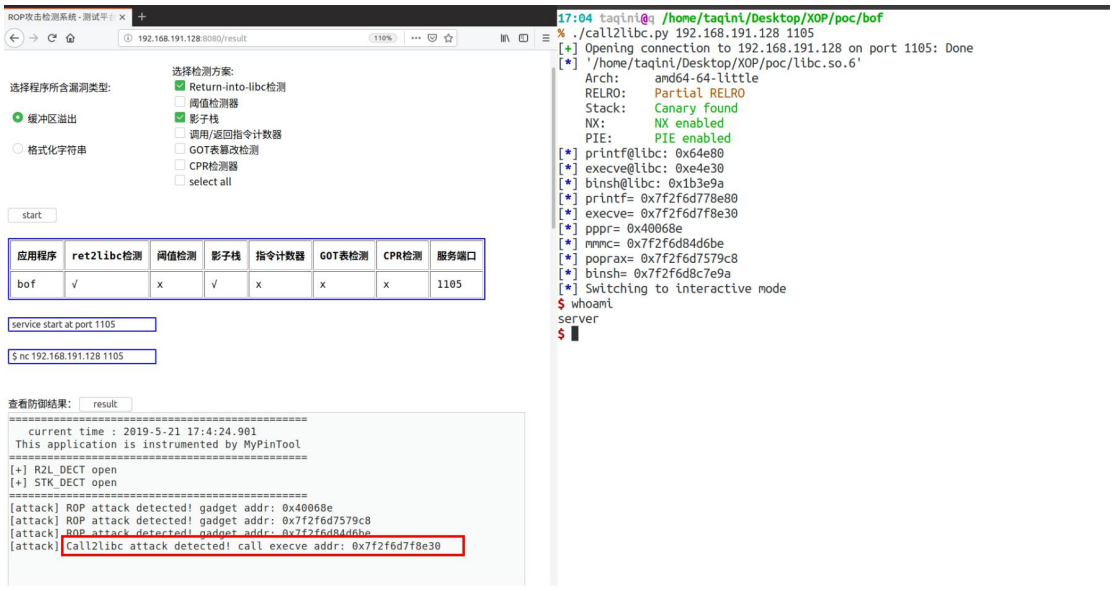


图 5-20 影子栈对长 gadget ROP 攻击的检测结果

由于影子栈为了提高检测效率，检测范围仅为程序代码段，且漏洞程序在主函数的返回时发生缓冲区溢出，溢出点不在程序代码段，所以为了绕过影子栈，只需要将所有 gadget 换成 libc 中的代码即可。使用修改后的攻击代码再次进行 ROP 攻击，攻击结果和检测结果如图 5-21 所示，影子栈并没有识别到这次 ROP 攻击。

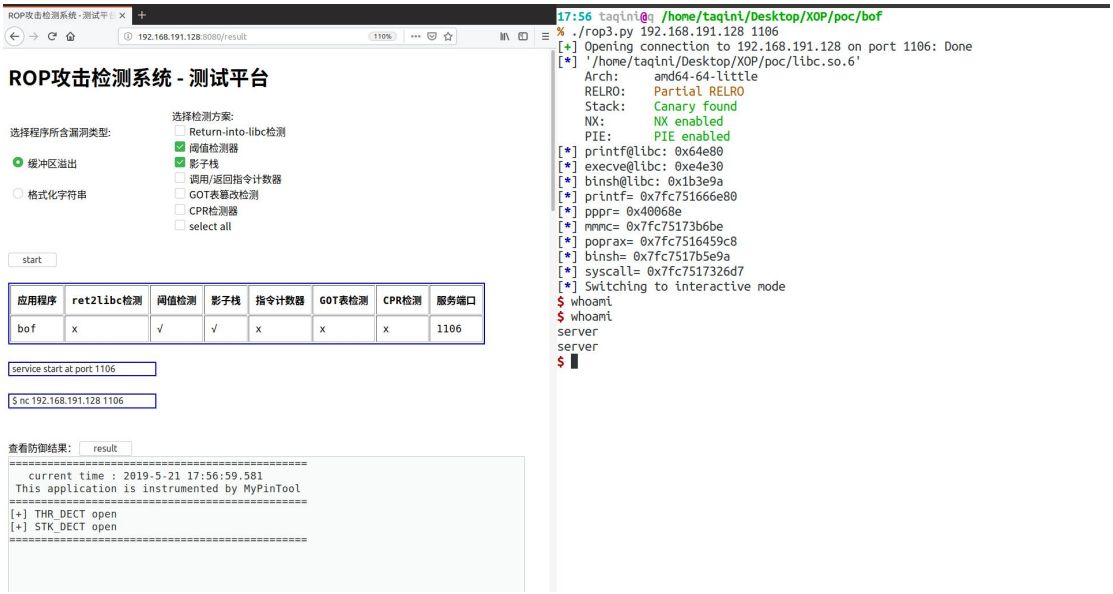


图 5-21 纯 libc gadget ROP 攻击的检测结果

影子栈检测方案已经被绕过，因此开启 CPR 检测方案，CPR 检测器和影子栈都通过检查返回地址的方式检测 ROP 攻击，CPR 的检测范围为所有地址，与影子栈相比，该方案的检测范围更广，但是效率较低。依然使用纯 libc gadget 进行 ROP 攻击，攻击结果和检测结果如图 5-22 所示，CPR 检测器检测到了全部的位于 libc 中的 gadget，并成功地识别了这次 ROP 攻击。

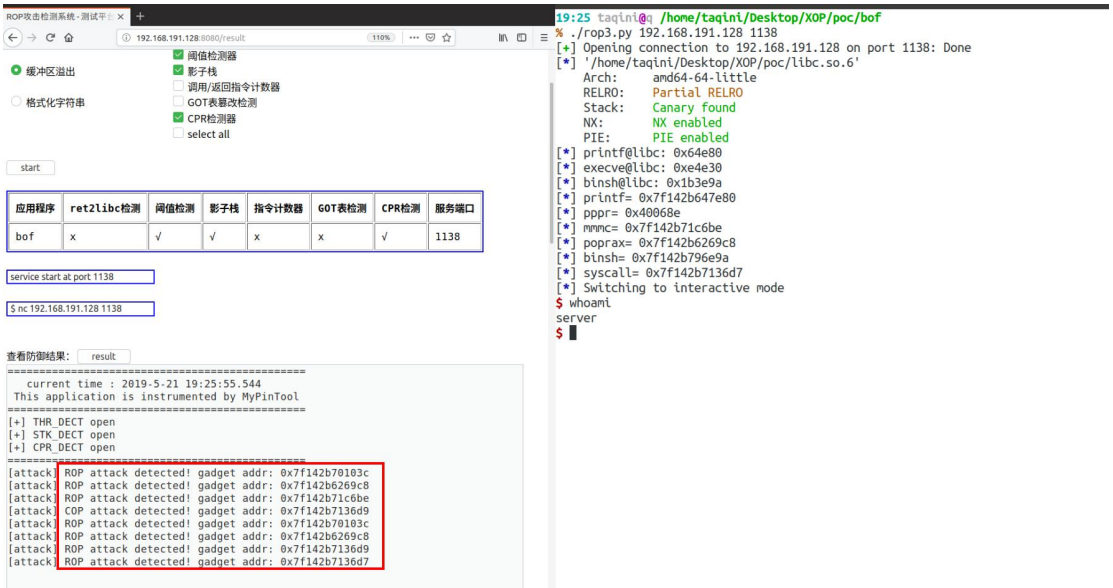


图 5-22 CPR 检测方案对纯 libc gadget ROP 攻击的检测结果

调用/返回指令计数器检测方案，从另一个角度对 ROP 攻击进行检测，也能弥补影子栈的不足，因此关闭 CPR 检测器，开启调用/返回指令计数器检测方案，依然使用纯 libc gadget 进行 ROP 攻击，攻击结果和检测结果如图 5-23 所示，即使攻击者使用了位于 libc 的 gadget，但是 gadget 以返回指令为结尾的特征没变。攻击时被执行的返回指令数量超过了调用指令数量，因此调用/返回指令计数器检测到了攻击，并成功地识别出了攻击类型为 ROP 攻击。

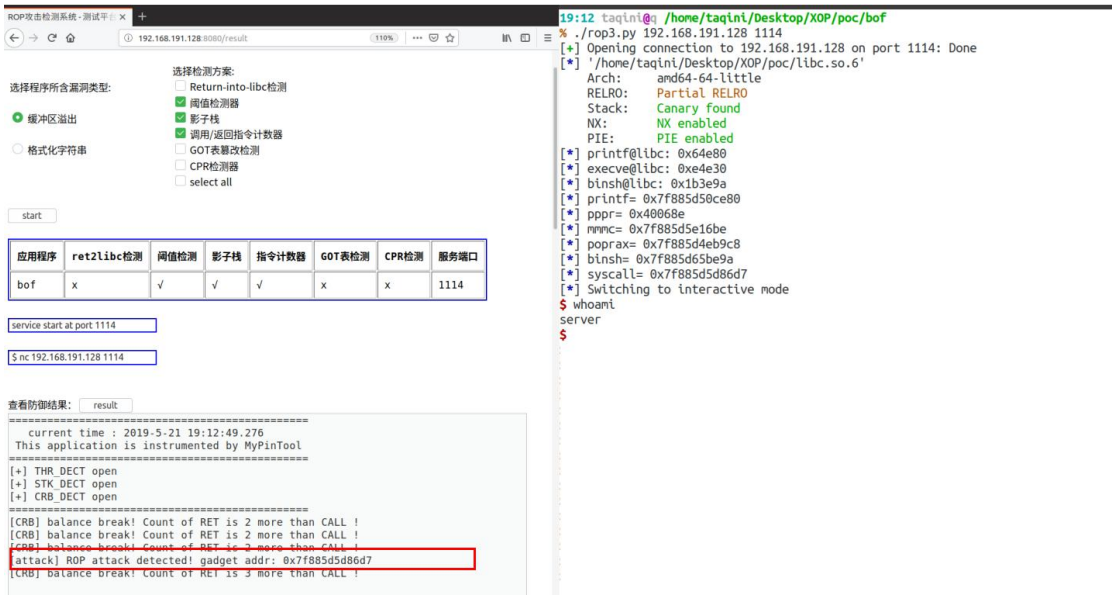


图 5-23 调用/返回指令计数器对纯 libc gadget ROP 攻击的检测结果

为了绕过调用/返回指令计数器检测方案，攻击者可以使用 JOP 攻击，或者交替使用以调用指令为结尾的 gadget 和以返回指令为结尾的 gadget，手动的平衡调用与返回指令的数量。因此，要想保证 ROP 攻击检测的准确性，需要开启多种检测方案对 ROP 攻击进行综合检测。

5.5.4 JOP 攻击防御与检测

在 JOP 攻击中，攻击者首先构造调度器，然后劫持程序控制流至调度器入口，由调度器控制程序控制流在各个 gadget 之间跳转。调度器的构造比较困难，因此在本次实验中，使用漏洞程序预置的 JOP 调度器来模拟实际的 JOP 攻击。由于基于指令特征设计的阈值检测器和调用/返回指令计数器没有检测 JOP 攻击的功能，因此只开启影子栈检测方案，对 JOP 攻击进行检测。JOP 攻击的攻击与检测结果如图 5-24 所示。

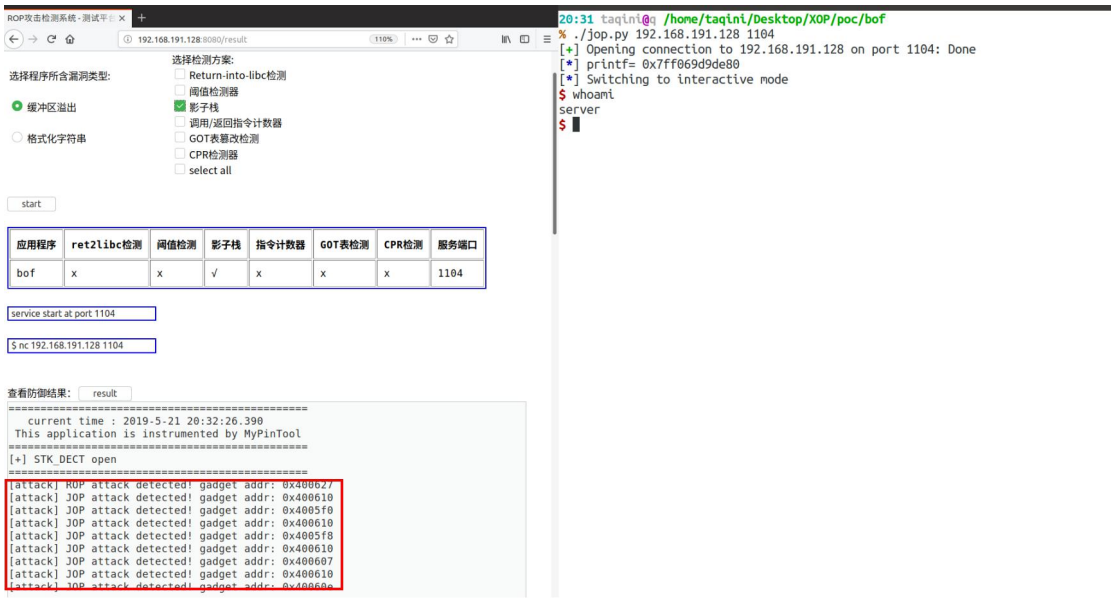


图 5-24 依赖程序堆栈的 JOP 攻击与检测结果

由于本次 JOP 攻击是依赖于程序堆栈的，通过缓冲区溢出漏洞，将函数返回地址篡改为调度器的入口地址，由于调度器位于程序代码段中，因此被影子栈检测到。其实无论调度

器位于何处，只要修改了返回地址，就能够被 CPR 检测器检测到，因此本次实验不再赘述 CPR 检测方案。

更换漏洞程序，选择 fsb 程序，对程序进行不依赖程序堆栈的 JOP 攻击。攻击及检测结果如图 5-25 所示。攻击程序利用格式化字符串漏洞，修改程序 GOT 表，实现了控制流劫持至调度器。因为整个攻击过程在，栈中的返回地址未被篡改，影子栈检测方案失效，没有检测到此次 JOP 攻击。

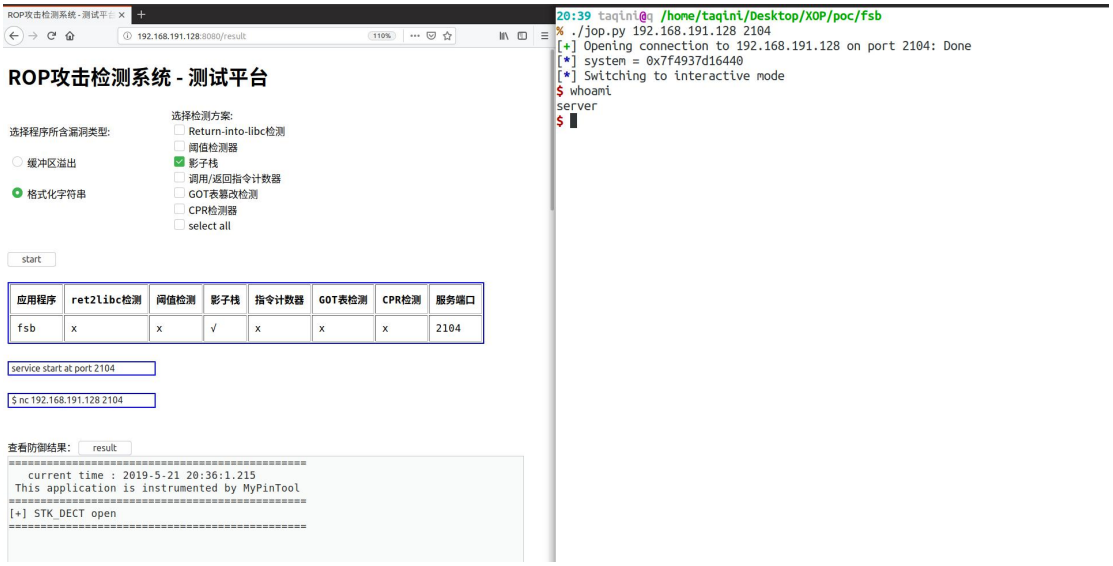


图 5-25 不依赖程序堆栈的 JOP 攻击与检测结果

开启 GOT 表篡改检测方案，再次对存在格式化字符串漏洞的程序进行 JOP 攻击，攻击和检测结果如图 5-26 所示，GOT 表篡改检测器检测到了 JOP 攻击对程序 GOT 表的非法修改，并识别出了 JOP 攻击。

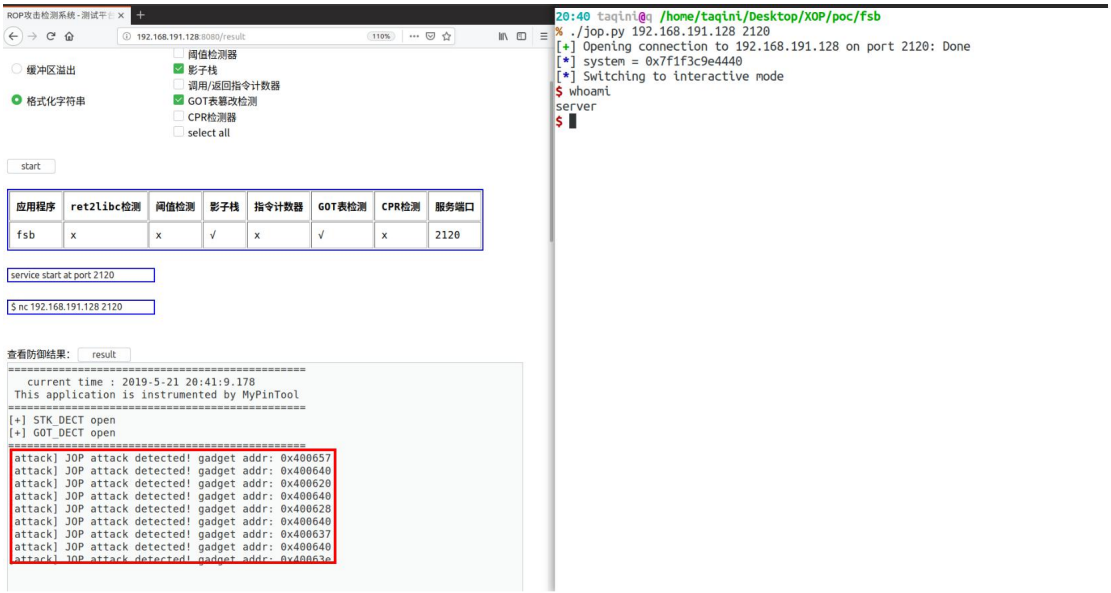


图 5-26 GOT 表篡改检测器对不依赖程序堆栈的 JOP 攻击的检测结果

5.5.5 结果评估

各检测方案对 return-into-libc，ROP 攻击，JOP 攻击的检测结果见下表。

表 5-1 检测方案对各类型攻击的检测结果

漏洞类型	攻击类型	阈值检测器	Return-into-libc 检测	影子栈	调用/返回指令计数器	CPR 检测器	GOT 表篡改检测	综合检测
缓冲区溢出漏洞	Return-into-libc	×	√	√	√	√	×	√
	Call-into-libc	×	×	√	√	√	×	√
	ROP	√	×	√	√	√	×	√
	长 Gadget ROP	×	×	√	√	√	×	√
	纯 Libc gadget ROP	×	×	×	√	√	×	√
	JOP	×	×	√	×	√	×	√
格式化字符串漏洞	JOP	×	×	×	×	×	√	√

5.6 本章小结

第六章　总结和展望

6.1　总结

为了防止误报，本文阈值设置的比较保守，如果只开启阈值检测器，则检测的准确度不高，因此，阈值检测器可作为辅助检测方案，需要与其他检测方案配合使用。从不同角度，降低漏报率。

影子栈能够检测常见的控制流劫持，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，也可以绕过调用/返回平衡检测。

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。

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] Vasilis Pappas, Michalis Polychronakis, and Angelos D Keromytis. Transparent ROP exploit mitigation using indirect branch tracing. In Proceedings of the 22nd USENIX Conference on Security, 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] 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)