

本科生毕业设计(论文)

基于二次汇编的ROP攻击防御技术研究与实现

Research and implementation of ROP attack defense based reassemble

|  |  |
| --- | --- |
| 学 院： | 计算机学院 |
| 专 业： | 软件工程 |
| 学生姓名： | 李博 |
| 学 号： | 1120162015 |
| 指导教师： | 孙建伟 |

2020 年 月 日

原创性声明

本人郑重声明：所呈交的毕业设计（论文），是本人在指导老师的指导下独立进行研究所取得的成果。除文中已经注明引用的内容外，本文不包含任何其他个人或集体已经发表或撰写过的研究成果。对本文的研究做出重要贡献的个人和集体，均已在文中以明确方式标明。

特此申明。

本人签名： 日 期： 年 月 日

关于使用授权的声明

本人完全了解北京理工大学有关保管、使用毕业设计（论文）的规定，其中包括：①学校有权保管、并向有关部门送交本毕业设计（论文）的原件与复印件；②学校可以采用影印、缩印或其它复制手段复制并保存本毕业设计（论文）；③学校可允许本毕业设计（论文）被查阅或借阅；④学校可以学术交流为目的,复制赠送和交换本毕业设计（论文）；⑤学校可以公布本毕业设计（论文）的全部或部分内容。

本人签名： 日 期： 年 月 日

指导老师签名： 日 期： 年 月 日

基于二次汇编的ROP攻击防御技术研究与实现

摘　要

随着计算机和软件技术的不断发展，使用软件已经是人们日常生活中必不可少的一部分。与此同时，也有越来越多的软件漏洞被挖掘出来，这些漏洞一旦被黑客利用，将极大地威胁系统的安全。面向返回编程(Return-oriented programming)是一种功能强大且用途广泛的代码重用攻击，攻击者可以基于现有的指令片段(gadget)引导程序控制流执行非预期功能，由此引发一系列的安全事件。于是使用有效的ROP攻击防御措施对于系统安全的保护有重要意义和价值。

本文对现有的ROP攻击防御技术进行讨论和分析，提出了一种基于二进制重写技术的ROP攻击防御方法，对Linux的32位和64位可执行文件进行保护。当前的ROP攻击防御方法一部分需要程序的源码，在编译器层面进行修改，然而如今的很多商用软件并不开源；另一部分使用动态监控的方法，会造成很大的性能开销。本文通过使用二次汇编框架Ramblr，可以有效的避免二进制文件没有源码的问题，在此基础上提出了指令替换，无效指令填充和自由跳转保护三种方案。通过对含有0xc2这样可能被解析为retn指令的指令进行等效指令替换或无效指令分隔，减少gadget数量，达到缓解ROP攻击的效果；以及在直接跳转和间接跳转前进行数据校验，保护控制流的完整性，防止攻击者直接调用跳转指令进行控制流转移。

实验部分，通过ROP攻击防御测试，上述方案可以有效地减少gadget数量，并且能够防御基于栈溢出的ROP攻击。结果表明，balabala。

**关键词：**软件安全、面向返回编程、二次汇编、Ramblr

The Subject of Undergraduate Graduation Project (Thesis) of Beijing Institute of Technology

Abstract

Key Words: ROP attack defense, reassemble, Ramblr

目录

[摘　要 I](#_Toc38704180)

[Abstract II](#_Toc38704181)

[第1章 绪论 1](#_Toc38704182)

[1.1 研究背景和意义 1](#_Toc38704183)

[1.2 国内外研究现状 2](#_Toc38704184)

[1.2.1 ROP攻击防御技术 2](#_Toc38704185)

[1.2.2 二次汇编技术 4](#_Toc38704186)

[1.3 论文主要研究内容 4](#_Toc38704187)

[1.4 论文结构 5](#_Toc38704188)

[第2章 相关理论知识 6](#_Toc38704189)

[2.1 ROP攻击技术 6](#_Toc38704190)

[2.1.1 面向返回编程的背景知识 6](#_Toc38704191)

[2.1.2 面向返回编程的原理 11](#_Toc38704192)

[2.2 二次汇编技术 13](#_Toc38704193)

[2.2.1 二次汇编技术的相关知识 13](#_Toc38704194)

[2.2.2 二次汇编技术的原理 15](#_Toc38704195)

[第3章 基于二次汇编的ROP攻击防御方法设计 17](#_Toc38704196)

[3.1 设计目标 17](#_Toc38704197)

[3.2 设计方案 18](#_Toc38704198)

[3.2.1 指令替换的设计 18](#_Toc38704199)

[3.2.2 静态插桩的设计 21](#_Toc38704200)

[3.2.3 重编译的设计 22](#_Toc38704201)

[第4章 基于二次汇编的ROP攻击防御实现 23](#_Toc38704202)

[4.1 指令替换 23](#_Toc38704203)

[4.2 静态插桩 26](#_Toc38704204)

[4.3 重编译 29](#_Toc38704205)

[第5章 测试及结果分析 32](#_Toc38704206)

[5.1 测试环境 32](#_Toc38704207)

[5.2 测试方案 32](#_Toc38704208)

[5.2.1 性能 32](#_Toc38704209)

[5.2.2 gadgets数量 32](#_Toc38704210)

[5.2.3 ROP防御效果 32](#_Toc38704211)

[5.3 测试结果分析 32](#_Toc38704212)

[结　论 33](#_Toc38704213)

[总结 33](#_Toc38704214)

[展望 33](#_Toc38704215)

[致谢 34](#_Toc38704216)

[参考文献 35](#_Toc38704217)

第1章 绪论

1.1 研究背景和意义

随着网络技术的快速发展，计算机正不断受到越来越复杂的攻击者的威胁。在网络中，攻击者发送恶意制作的数据包，这些数据包利用软件错误来获得未经授权的控制。防止此类错误存在的研究一直在进行中，但是到目前为止，仍未能为可利用的安全漏洞问题提供一个周全的方法。但是，针对攻击本身的不同方面的解决方案已经取得了一些成功，这导致了恶意攻击者和安全研究人员之间的“军备竞赛”。

最早的攻击技术之一是代码注入攻击，攻击者将精心编写的代码写入有漏洞的程序的内存中，然后利用一个漏洞将控制流重定向至代码所在的内存，从而执行代码。攻击者注入的代码能够损害数据库的完整性和安全性，还能够窃取数据来绕过访问和身份验证控制。针对这一问题，早在1961年，Burroughs 5000推出时就为可执行空间保护提供了硬件支持，在实现带标记的体系架构时，内存中的每个字节都带有一个关联的隐藏标记位，用于指定其是代码或数据，确保了内存可写或可执行。Microsoft公司在2004年将这种技术首次应用在Windows XP Service Pack 2，可执行的空间保护在Windows中也称为数据执行保护（Data Execution Prevention，DEP），开启此功能后，计算机将拒绝执行位于用户可写内存区域中的任何代码，从而防止攻击者将代码写入堆栈并通过返回地址覆盖跳转到堆栈。但是，早期的数据执行保护依旧不是完美的，攻击者可以使用代码重用攻击，例如return-into-libc技术，通过栈溢出覆盖返回地址将控制流转移到libc库中现有的代码片段，调用system函数或mprotect函数创建可写的可执行内存区域来绕过DEP。于是Windows系统又添加了地址空间布局随机化（Address Space Layout Randomization, ASLR）技术，用于将虚拟地址随机分配给正在运行的程序中的代码和数据，防止攻击者利用事先从程序中获取的静态地址来执行代码。

面向返回编程（Return-Oriented Programming, ROP）是一种基于代码重用的攻击，它也是栈溢出的高级形式，最初由Shacham在2007年提出[1]。通过这种方法攻击者可以劫持程序的控制流，并执行计算机内存中已经存在的指令片段（gadget），从而绕过DEP。像栈溢出一样，面向返回编程可以滥用缓冲区溢出漏洞来执行恶意指令，而不受安全措施DEP的阻碍。



图 1-1 国家信息安全漏洞库-漏洞新增数量统计图

图1-1是国家信息安全漏洞库[2]从2019年9月至2020年2月关于漏洞新增数量的统计图，近六个月来平均每月漏洞达到1432个，2020年2月份采集安全漏洞共1246个，其中缓冲区错误类型的漏洞最多，数量为150个，占比12.04%，并且在超危漏洞中缓冲区错误类型的漏洞也有20个，数量最多。这些漏洞可以为ROP攻击提供很好的利用条件。于是，及时的检测并防御ROP攻击就显得极为重要。基于此，本文提出基于二次汇编的ROP攻击防御技术，通过二次汇编技术可以对可执行文件进行反汇编生成汇编文件，并对汇编指令进行混淆和插桩，然后将汇编文件重编译成新的可执行文件。此过程不需要程序的源代码，并且可以在指令级别进行处理，在不对程序进行外部监控的情况下自动化的完成ROP攻击防御，减少了性能开销。

1.2 国内外研究现状

1.2.1 ROP攻击防御技术

ROP攻击通过将目标程序中现有的指令序列链接在一起，可以使远程攻击者执行具有图灵完备的计算，而无需注入任何恶意代码。正是由于其巨大的威胁，近年来产生了许多针对ROP攻击的防御方法。

一些安全研究人员在程序源代码的基础上进行ROP攻击的防御。

Kaan Onarlioglu等人[3]提出了基于减少gadget和保护自由跳转分支的防御方法，通过修改编译器使源代码进行编译生成中间代码时对中间代码进行处理，包括指令替换、无效指令（nop指令）以及自由跳转保护，以达到减少gadget数量和保护跳转指令不被攻击者滥用的目的。

Tyler Bletsch等人[4]提出了基于控制流锁定的防御方法，通过一个大小为1比特的变量k来标记此时状态是解锁、间接调用、来自间接调用函数的返回以及来自直接调用的函数返回，从而防止跳转指令被攻击者滥用。

以上方法的实现需要重新编译程序，意味着每个使用者不得不使用新的程序来增加安全性。然而重新构建一个新的程序往往是一个巨大的工程，并且如果源代码丢失，那么程序的安全性就更无法得到保障。于是一些安全研究人员在不修改程序的基础上通过外部监控的方式进行ROP攻击的防御。

Lucas Davi等人[5]使用二进制指令框架Pin来监控栈内容达到防御ROP攻击的目的。通过复制一个程序栈，在程序动态运行时，在处理器执行指令之前拦截指令并检查指令类型。若为call指令，则将返回地址复制一份压进栈，若为ret指令，则比较复制的栈与程序栈的栈顶内容是否相同，从而判断该跳转指令是否被攻击者使用。

Vasilis Pappas等人[6]提出了一种基于异常控制流转移的动态检测方法。利用处理器提供的上次分支记录技术（Last Branch Recording，LBR），可以获取上一次执行分支的记录，从而在系统调用返回时，判断之前是由call指令还是ret指令跳转到系统调用的，若是call指令则是善意的系统调用，否则是恶意的系统调用。

Ivan Fratrić等人[7]提出了六种不同的检测方法来判断一个函数调用是否是恶意的。包括检验栈指针是否在正确的边界内、检查栈上是否有特定函数的入口地址、验证返回地址是否有效（可执行并且前面有call指令）、验证调用栈是否有效、模拟程序执行流程以及特定函数检验（VirtaulProtect和LoadLibrary函数）。

Yueqiang Cheng等人[8]使用LBR技术和预运行的方法。在预运行阶段导出所有指令的偏移、类型和对齐方式，并收集程序和共享库中可能的gadget，以此构建一个数据库;在运行阶段通过滑动窗口机制限制可利用的指令数量，并通过LBR技术识别先前执行的指令和传入的指令是否是ROP攻击的gadget。

以上的方法大部分依赖于ret-call指令之间的联系，并通过CPU的LBR技术在程序动态运行时获取控制流的转移信息，从而判断是否受到ROP攻击。然而外部监控的方法会带来很大的性能开销，并且Nicholas Carlini等人[9]已经对kBouncer和ROPecker两种防御方法提出了破解的思路及实现方法。

1.2.2 二次汇编技术

二次汇编是指将可执行文件反汇编得到汇编文件后，再将汇编文件重汇编成可执行文件。Shuai Wang等人[10]于2015年率先提出了静态二次汇编的方法，并实现了名为Uroboros的开源工具。但是，作为第一个实现二次汇编的项目，Uroboros存在许多的局限使得它无法处理许多可执行文件。Ruoyu Wang等人[11]在Uroboros工具提供的二次汇编方法的基础上进行完善，实现了一个完成度更高的工具Ramblr。Ramblr使用二进制分析框架angr[12]来进行控制流图（Control Flow Graph，CFG）恢复，并封装为了angr的子模块。相较Uroboros，Ramblr具有更高的成功率和很好的可拓展性，所以本文的ROP攻击防御方法基于Ramblr进行实现。

1.3 论文主要研究内容

本文主要研究基于二次汇编的ROP攻击防御技术，将二次汇编框架与一些ROP攻击的防御方法相结合。利用二次汇编的可直接对指令进行处理的特点，使用指令替换和无效指令填充的方法减少gadget的数量，缓解ROP攻击；使用跳转指令保护的方法保证控制流的完整性，防止攻击者任意调用跳转指令。本文的主要工作如下：

第一：本文提出了一种新的ROP攻击防御的思路，可以避免程序源码难以获取的问题。与现有的ROP攻击防御方法不同的是，本文的方法基于二次汇编实现，可以自动化地对程序进行反汇编和重汇编操作，不需要外部监控，生成的程序在功能上与原始程序完全相同。

第二：本文使用了一种适合二次汇编框架的ROP攻击防御方法。借助二次汇编技术可获取程序的汇编指令并对其直接进行操作的特点，提出了指令替换、无效指令填充和自由跳转指令保护三种方法，最大限度的减少gadget数量以及保护控制流的完整性。

第三：本文使用linux的coreutils工具包作为测试样本，结果表明经二次汇编处理之后的程序较原始程序在大小上小近xx%，在性能开销上xxxxx

1.4 论文结构

本文的主要研究内容是设计并实现了一种基于二次汇编的ROP攻击防御方法，总体分为五大章节，组织结构为：

第一章：绪论。本章介绍了论文的研究背景和研究意义、国内外的研究现状、论文的主要研究内容和论文的组织结构。

第二章：相关理论知识。本章首先介绍了面向返回编程的背景知识，包括堆栈缓冲区溢出的原理和利用方法。其次介绍了gadgets的查找方法和ROP的基本利用。最后介绍了二次汇编技术。

第三章：基于二次汇编的ROP攻击防御方法设计。本章首先介绍了基于二次汇编的ROP攻击防御方法的设计目标。然后对设计方案进行了阐述，包括指令替换、静态插桩以及重编译三个模块的设计原理、功能以及初步的效果展示。

第四章：基于二次汇编的ROP攻击防御方法实现。本章详细介绍了基于二次汇编的ROP攻击防御实现方法，包括指令替换、静态插桩和重编译的主要功能、使用的技术以及实现细节。

第五章：测试及结果分析。

结论：总结和展望。

第2章 相关理论知识

2.1 ROP攻击技术

2.1.1 面向返回编程的背景知识

在正式介绍面向返回编程的原理之前，这里先对面向返回编程的背景知识进行大致的介绍，包括堆栈的定义、堆栈缓冲区溢出、堆栈溢出的利用方法以及基础的保护技术。有了背景知识的掌握后，理解面向返回编程的原理将会更加容易。

2.1.1.1 堆栈的定义

在现代计算机的操作系统中，每个进程都有它专属的内存区域，称为堆栈（stack）。进程可分为三大区域，分别是代码段、数据段和堆栈。代码段是一个只读的数据段，它包含了程序的指令；数据段包含了程序中初始化和未初始化的数据，比如静态变量；堆栈则存储函数中动态分配的变量，用来传递函数的参数以及在函数执行结束后返回数据。堆栈是一个充当元素集合的抽象数据类型，它的功能就像一堆盘子，最后放在最上面的盘子也是要取出的第一个盘子。这种从堆栈取元素的排序方式称为后进先出。

每次函数调用时，操作系统会分配堆栈的一小部分来存储它的数据，这一小部分称为栈帧，由函数参数、函数的局部变量以及用于恢复前一个栈帧的数据。尽管从理论上来说堆栈的功能很简单，但仍然很可能出现错误。与堆栈交互时遇到的最常见的错误之一就是堆栈缓冲区溢出。这个错误与变量在堆栈的存储方式有关，它也是引起漏洞的关键。在下一节堆栈缓冲区溢出原理中，本文将研究导致此错误的原因及其工作原理。

2.1.1.2 堆栈缓冲区溢出的原理

缓冲区溢出是一种程序运行时的错误，发生在程序尝试对超出其预期数据结构的内存地址进行写操作的时候。通常是写入数据的大小超出缓冲区大小，这可能会破坏相邻的内存数据，并导致程序崩溃或错误的结果。缓冲区溢出错误的一个常见例子就是数组索引超出范围，该错误使攻击者可以通过代码注入劫持程序控制流，导致程序受到攻击。

为了更直观地说明缓冲区溢出的原理，接下来将使用一个例子来介绍。程序源码如图2-1，使用32位系统编译，其功能非常简单，使用gets函数接收字符串并输出该字符串。字符串名为buffer，大小为8字节。



图 2-1 缓冲区溢出例子源码

程序正常执行时，栈的内容如图2-2，可以看到使用字符串“Hello”作为输入时，栈内容没有异常。但是，如果使用的字符串过长而无法放入预期的缓冲区，那么字符串的剩余部分溢出到堆栈上，并从而覆盖其它的内容。程序发送缓冲区溢出时，栈的内容如图2-3，可以看到输入的字符串为16个字符A，由于大小超出预期，并且C语言中的gets函数没有边界检查，导致栈上的上一个函数的栈帧以及返回地址被字符A所覆盖。这里由于返回地址被覆盖为无效地址，程序将异常终止，但攻击者可以精心构造输入，利用此漏洞接管程序的控制流，实现任意代码执行。



图 2-2 程序期待的输入 图 2-3 程序发生缓冲区溢出

2.1.1.3 堆栈缓冲区溢出的利用方法

本小节将介绍两种不同的技术，这些技术为面向返回编程奠定了基础。第一项也是最基本的技术称为堆栈粉碎（Stack Smashing）。它是第一项利用堆栈缓冲区溢出的技术，并且在Aleph One发表他著名的文章[13]之后该项技术得到了广泛的认可。第二项将介绍的技术是返回到共享库（Return-to-libc），它被视为面向返回编程的前身，与其有很多相似之处，并且也是基于代码重用的技术，但它的通用性要差一点。

**堆栈粉碎技术**是我们可以通过堆栈缓冲区溢出进行的最基本的利用方法。使用这项技术，可以通过调用堆栈来重定向程序的控制流。当一个函数被调用时，调用者会将返回地址压入堆栈，而当函数执行结束时，被调用者会将返回地址弹出堆栈，并将控制权转移给该地址。如果攻击者可以控制该地址，那么就可以任意地重定向控制流，使得攻击者能够执行任意代码。

如图2-3所示，如果我们输入足够长的字符串，栈上存储的返回地址将被部分输入覆盖，这导致当函数结束时，控制权就会移交给地址0x49494949，即字符串“AAAA”的十六进制。在大多数情况下，0x49494949会超出程序的内存范围，并且程序将会异常终止。但如果这个地址包含有效指令，程序将从该地址继续执行。这意味着攻击者可以用包含有效指令的地址覆盖返回地址，从而执行任意指令。

要使用堆栈粉碎技术对程序进行利用，攻击者必须构造一个字符串，该字符串由三部分组成：填充偏移的无效字符串、新的返回地址和配合新的返回地址的有效负载（payload）。攻击者首先应当计算到达返回地址所在栈上的位置所需的偏移。偏移取决于许多因素，比如缓冲区的长度和堆栈的实现原理。这里提出两种方法来计算偏移，一是单步执行代码，二是通过报错暴力破解。从图2-2可以看出，从缓冲区的开始到返回地址的开始的偏移是12个字节。攻击者得到偏移之后，需要用一条指令的地址覆盖栈上存储的返回地址，该指令的功能可以是控制寄存器的值，也可以是转移控制流，但相同点是它们都需要以跳转指令结尾。这里的示例如图2-4所示，我们将堆栈上的返回地址覆盖为jmp esp指令所在的地址，在程序执行到函数返回时，使用栈上新的返回地址，执行jmp esp指令，跳转到栈顶元素所处地址，即shellcode。



图 2-4 构造堆栈粉碎的字符串

然而在现代操作系统中，由于采用了堆栈不可执行和其它内存空间的保护措施，使得直接在堆栈上执行代码几乎不可能。因此，需要改用基于程序已有指令的新技术，最早实现这一思想的技术之一就是返回到共享库技术。

**返回到共享库技术**是面向返回编程的前身，并且是最早发现的代码重用攻击之一，它由Alexander Peslyak于1997年首次发表[14]。返回到共享库攻击通过调用共享库中已经存在的函数来绕过堆栈不可执行的安全措施，它和将攻击者编写的代码压入堆栈并执行的堆栈粉碎技术不同。C的标准共享库是一个寻找现有函数十分方便的地方，其中包含许多不同功能的函数，并且被不同的应用程序广泛使用。这也是为什么返回到共享库技术被命名为Return-to-libc的原因，其中的字母c代表的就是c语言。利用这项技术的一个很大的限制就是我们只能在漏洞利用中调用已经存在的函数，在下一节面向返回编程的原理中，将研究一种更高级的技术称为面向返回编程，这项技术为我们提供了更多选择，并为在现代操作系统中绕过堆栈不可执行措施提供了更大的灵活性。

2.1.1.4 针对堆栈缓冲区溢出的保护措施

第一种保护措施是可执行的空间保护（Executable Space Protection）。它利用了硬件功能的不可执行位，是一种CPU用于在处理器指令和存储之间隔离内存区域的技术。如果硬件不支持的情况下，可以使用软件仿真来实现此功能，但需要一些性能开销。此外存在其它可执行空间保护方案，例如通过一个比特位将内存页标记为可写或可执行。在Windows操作系统中，该措施是数据执行保护方案的一部分。从Windows XP Service Pack2开始，这种方法已经在Windows操作系统的每个版本中实现。由于该措施将内存区域标记为不可执行，因此无法对受该措施保护的应用程序执行堆栈粉碎攻击。因此，攻击者必须利用返回共享库或面向返回编程技术来绕过内存页的不可执行保护。

第二种保护措施是堆栈金丝雀（Stack Canary）。它是放置在堆栈上缓冲区与控制数据之间的值，以监控堆栈缓冲区溢出。当缓冲区溢出破坏了堆栈内容时，canary值也将被覆盖。在函数执行结束时，会进行canary的验证，由于缓冲区溢出导致canary被破坏，因此将出现不匹配的情况。

第三种保护措施是地址空间布局随机化（ASLR）。它可以使程序的地址空间随机化，是一种计算机安全技术。这使得攻击者在执行漏洞利用时很难准确地跳转到正确的地址。ASLR技术会随机化进程的关键数据区域，如堆栈、共享库以及可执行文件的基址。并且基于使用概率来防止攻击，它依赖于攻击者猜测随机放置的数据的正确位置的几率非常低。由于攻击者必须知道要执行的指令的位置，这使得利用返回到共享库和面向返回编程的攻击变得更加困难。由于攻击者很难知道他们注入代码的准确位置，因此堆栈粉碎技术也很难付诸实践。

2.1.2 面向返回编程的原理

面向返回编程是一种基于代码重用的攻击，所以要实现面向返回编程，程序中必须存在我们要执行的指令，并且指令片段（gadgets）以返回指令结束。这意味着我们需要足够丰富的指令集来执行面向返回编程攻击。如果有足够大的指令集，意味着面向返回编程就是图灵完备的。接下来将从gadgets的结构、寻找gadgets的方法以及ROP的基础利用三部分阐述面向返回编程的原理。

2.1.2.1 gadgets的结构

gadgets是一条或多条指令组成的指令片段，并且以返回指令（retn）结束。通过将gadgets组合放到堆栈上，可以实现不同的功能，其中不同的gadgets组合成为ROP链（ROP-chain）。图2-5是一个简单ROP链的示例，它的功能是将寄存器eax和edx清零，然后将eax加3，edx加2，并把edx的值加在eax上。



图 2-5 gadgets示例-控制寄存器值

通过返回指令可以将每个gadgets链接在一起。当返回指令执行时，会将栈顶的元素弹出并跳转到元素值所在位置，在ROP链中，则会跳转到下一条指令的地址。建立ROP链时，有一些非常有用的gadgets。例如异或指令xor，当我们需要将寄存器清零但又要避免空字节时就可以用异或指令将寄存器自身异或，达到清零的目的；还有就是pop指令，它可以将栈顶的值弹出并放入寄存器中。

2.1.2.2 查找gadgets的方法

查找gadgets是构造ROP链必不可少的一部分。gadgets决定了攻击的类型和如何构造ROP链，通过对二进制文件及其导入的共享库进行静态分析，可以找到gadgets。在寻找gadgets时，应避免一些特殊值，如空字节（0x0）、换行符（0xA）和回车符（0xD）。因为这些值代表了C语言中字符串的结束，这导致当我们在利用像strcpy之类的函数时，导入的ROP链会被提前解析结束。在通过静态分析查找gadgets时，我们应寻找返回指令（0xC3），并从返回指令处向前构建gadgets，直到达到了不希望出现的指令或gadgets预定的最大长度为止。在寻找gadgets时，最好先将其构建的尽可能长，然后将其拆分为多个小的gadget。例如如果需要构造“a; b; c; retn”这样的gadgets，那么从逻辑上讲，也可以将其分隔成“a; b; retn”和“c; retn”两个小的gadgets。Shacham提出了一种算法[1]用于寻找gadgets。由于Intel x86的指令是不定长的，这使得我们可以从任意位置开始解析汇编代码，导致出现非预期的指令。如图2-6的前两行是经编译器产生的原有指令。然而当我们向后偏移两个字节，并以此为起点开始解析时，就会产生如图2-6的第4行和第5行的结果。



图 2-6 intel不定长指令示例

通过这种方法我们可以查找出更多可利用的gadgets，从而使面向返回编程的功能更加强大。现在已经有许多成熟的gadgets搜索工具可以直接使用，比如ROPgadget、ROPGenerator等。

2.1.2.3 ROP的利用示例

这里以Linux中的ret2syscall方法举例说明ROP链的作用，程序如图2-1所示。Linux系统中的系统调用通过int 80h实现，通过系统调用号来区分入口函数。这里我们要利用的是execve函数获取shell，那么系统调用号就是0xb，其输入的参数为字符串“/bin/sh”以及两个空值即0。所以需要控制寄存器eax、edx、ecx和edx分别为0xb、字符串“/bin/sh”的地址、0和0。



图 2-7 ROP链结构示例

ROP链结构如图2-7。可以看到，首先通过长度为12由字符A组成的字符串填充缓冲区并覆盖上一个栈帧；接着使用“pop eax； ret”指令的地址覆盖返回地址，并在其后附上0xb，当控制流执行到“pop eax”指令时，弹出的栈顶元素即0xb，达到控制寄存器eax的值的目的。此后的内容同理，控制寄存器edx、ecx和ebx，并最终返回到int 80h处，执行execve函数并获取shell。通过这个实例，可以更加直观的了解ROP链的工作原理。

2.2 二次汇编技术

2.2.1 二次汇编技术的相关知识

在阐述二次汇编的原理之前，先对这项技术的相关知识进行大致的说明，包括静态反汇编和二进制重写技术。

2.2.1.1 静态反汇编

反汇编顾名思义即汇编的逆过程，是指把机器语言解码成汇编代码。当前有许多关于正确及完全反汇编二进制文件的方法。线性扫描是其中最简单的技术，它是指对二进制文件的可执行区域从头到尾进行扫描，并将所有遇到的机器码解码为指令。大多数反汇编技术都基于更高级的方法，即递归遍历。该技术从二进制文件的入口点开始，解析每个控制转移的目标，并递归地沿着这些目标来解码遇到的机器码[15]。IDA Pro就是基于递归遍历方法的一个功能非常强大的二进制分析软件。

反汇编是二次汇编技术中的第一步，并且也是最基础的一步。反汇编的结果将用于程序控制流图的恢复，完整而准确的汇编代码能够有效地提高正确识别函数的概率。

2.2.1.2 二进制重写技术

二进制重写是指在保持现有功能的同时，将一个二进制文件静态或动态地转换为另一个二进制文件。通常在此过程中，可以选择将一个或多个新功能添加到转换后的二进制文件中。与动态二进制重写相比，静态二进制重写一般引入更低的开销，而后者在运行时会进行检测。因此，静态重写技术被广泛用于控制流完整性保护、二进制文件强化、安全策略增强以及二进制文件检测等方面[11]。

传统的静态二进制重写一般通过给插入的代码添加跳转的钩子，或者通过完整的二进制转换，将所有代码转换为中间表示并将其转换回机器码来进行。与原始文件相比，这两种方式生成的二进制文件都会产生大量开销。并且通过完整的二进制转换产生的文件通常与原始文件会有很大的不同。而二次汇编则没有这些缺点，因为通过二次汇编产生的文件是从原始文件恢复的汇编代码生成的，避免了额外的跳转和完整二进制转换。

动态二进制重写技术是指在执行二进制文件时对其进行转换，并以高代价的性能开销来保证对商用软件或去除符号表的文件进行完全覆盖的转换。常见的动态重写工具有Pin、DynamoRIO和Valgrind等。

2.2.2 二次汇编技术的原理

以Ramblr为例，简要介绍二次汇编技术的原理，包括反汇编和控制流图恢复、内容分类、符号化和重汇编。

2.2.2.1 反汇编和控制流图恢复

在一个目标文件可以被重汇编之前，需要先进行反汇编。Ramblr通过计算二进制文件的控制流图（CFG）并反汇编每一个已识别的基本块来实现这一步骤。此外Ramblr尝试识别并反汇编“死代码”，因为要实现二次汇编，反汇编尽可能多的代码是十分重要的。Ramblr的控制流图恢复基于angr二进制分析框架实现，如果angr对程序的控制流图恢复失败，Ramblr将停止继续运行并报告错误信息。

控制流图恢复从二进制文件的入口点开始，然后递归地遵循直接或间接控制流转换进行，直到将二进制文件的可执行区域（ELF文件的text段）全部遍历。

2.2.2.2 内容分类

Ramblr使用两种基本分析进行内容分类，即函数内数据依赖性分析和局部数值分析。函数内数据依赖性分析是指在特定函数的基本块上实施整体执行，在执行过程中恢复变量和常量之间的数据依赖关系，其中变量包括寄存器、堆栈变量和存储单元，并且执行的范围严格限制于当前函数。数值分析首次被提出是作为一种抽象解释技术，用来静态分析机器代码。Ramblr没有在整个二进制文件或函数上使用这种技术，而是设计了数值分析的约束版本即局部数值分析。这种方法仅在二进制文件的一部分上使用，比如基本块或循环等。

与传统的静态分析不同，Ramblr使用的两种分析方法受到严格的约束，使得可以快速实现并且结果易于处理。这些局部化分析通常用来分解跳转表、恢复原始数据类型以及检索简单循环中访问的数组大小。

2.2.2.3 符号化

在二进制文件的链接过程中，所有标签转换为绝对地址。在重汇编的时候，由于对目标文件的修改导致数据和代码在二进制文件的位置可能发生变化。如果二进制文件中有硬编码的指针地址或绝对跳转，则必须对其进行调整使其指向原来的数据或代码的位置。事实上，相对跳转也需要进行调整，因为在插入或删除代码会改变基本块之间的偏移。由此产生了符号化的方法，将这些硬编码的数字引用转换为符号，从而在重汇编时重新定位这些数据。

二进制文件中存在许多的立即数，但并不是所有的立即数都需要符号化。在代码中，可符号化的立即数必须是指令的一部分，用作指向代码或数据的指针。在数据中，可符号化的立即数是具有和机器位数相匹配的整数，它也用于指向代码或数据的指针。并且若立即数位于二进制文件的内存区域内，那么它就是可符号化的，而对于那些超出内存区域范围的立即数则是不可符号化的。

符号化还有许多需要考虑的细节，如有效内存区域的范围等，由于不是本文的研究重点，故这里不再赘述。

2.2.2.4 重汇编

重汇编过程相对前几个步骤来说显得十分简单直接。在符号化的结果中，为每个恢复的符号引用分配标签，并用相应的标签替换所有指令和数据中可符号化的立即数。生成的汇编代码会输出保存至单独的文件中，使用者可以根据需要添加补丁至文件中。最后，使用现成的编译器将所得到的汇编文件重编译为新的二进制文件。

# 第3章 基于二次汇编的ROP攻击防御方法设计

3.1 设计目标

本文针对当前ROP攻击防御方法性能开销过大以及部分方法需要程序源代码的问题，设计了一种基于二次汇编的ROP攻击防御方法。在二次汇编的框架中，基于减少gadgets数量和保护控制流的目标进行指令替换和静态插桩操作，实现了一个自动化的ROP攻击防御系统。该系统主要由以下三部分组成：

1. 指令替换

指令替换部分主要负责对机器码中含有可以被解析为跳转指令（如0xC3）的指令进行替换。通过对特定类型的指令设计等效的指令模板，在发现符合条件的指令时用等效指令对其进行替换，能够消除立即数被解析为跳转指令的可能，并减少gadgets数量。

1. 静态插桩

静态插桩部分包括无效指令填充和自由跳转保护。无效指令填充是指在指令对应的机器码中发现如0xC3这样的数据但没有相应的指令替换的模板时，在指令的前面填充无效指令，如nop指令，达到防止攻击者将其解析为可利用的gadgets的目的，并且需要在无效指令前添加相对跳转指令，减少因执行无效指令产生的性能开销。自由跳转保护是指在函数入口处使用canary对返回地址进行加密，并在函数结束时对返回地址进行解密，在间接跳转（jmp）时校验返回地址是否加密。达到保护控制流完整性的目的，防止攻击者直接调用跳转指令进行控制流转移。

1. 重编译

重编译部分负责对二次汇编框架产生的结果输出到文件并进行编译，产生新的二进制文件。新的二进制文件需要保持和原文件相同的架构类型（32位或64位），并且采取一样的保护方式，如是否开启地址随机化等。

3.2 设计方案

本文设计了一种基于二次汇编的ROP攻击防御方法，对二进制文件进行静态二进制重写，避免了需要程序源码的问题。从减少gadgets数量和保护控制流的完整性两个维度实现，使用指令替换、无效指令填充以及自由跳转保护三种方法对ROP攻击进行缓解和防御。本文实现的ROP攻击防御系统的流程如图3-1所示，二次汇编框架在接收到二进制文件后，首先进行控制流恢复、内容分类和符号化等一系列操作，然后在进行重汇编时，可以获取到二进制文件中分别以函数、基本块和指令为单位的数据。在此过程中，本系统对这些数据进行处理，其中指令替换和无效指令填充以指令为一个基本单位进行，自由跳转保护以函数为一个基本单位进行。当重汇编结束后，本系统进行最后一个步骤即重编译，将重汇编得到的输出保存为汇编文件并对其进行编译，得到新的二进制文件。



图 3-1 ROP攻击防御系统流程图

3.2.1 指令替换的设计

指令替换模块是通过设计指令的等效指令模板，在发现指令的机器码中含有0xC3这样可以被解析为返回指令的数据时将指令进行等效替换。由于二次汇编框架中的重汇编操作是以二进制文件、函数、基本块和指令这样的顺序从左至右逐级进行的。指令替换模块在指令进行重汇编时实现，包括对立即数中含有0xC3的指令以及操作数中含有0xC3的指令进行替换。

指令替换模块能够处理的指令类型有赋值、运算以及比较指令。对于立即数中含有0xc3的指令，采取对立即数拆分的方法，将原有立即数以0xc3为基准拆分为两个立即数之和，并结合指令类型进行相应的等效替换。以mov指令为例，如图3-2所示。汇编语法为at&t类型。可以看到原指令的立即数为0x1234c3ab，包含0xc3，故将立即数以0xc3为基准拆分为两个数0x1234c000和0x3ab。先将0x1234c000赋值给寄存器rcx，然后通过加法指令对rcx加上剩下的数0x3ab，达到等效替换指令并且消除立即数中含有0xc3的效果。



图 3-2 指令替换-立即数替换示例

对于操作数中含有0xc3的指令，为了能够有效的对其进行指令替换，这里首先介绍指令的操作数中出现0xc3的原因。以intel x64指令为例，指令格式如图3-3。其中Mod/R和SIB字节用于编码指令的寻址模式和操作数。当使用某些寄存器作为操作数时会导致Mod/R或SIB字节设置为和返回指令retn相同的操作码。



图 3-3 intel 64位的指令格式

举个例子，当赋值指令mov将寄存器rax指定为源操作数，并且目标操作数为寄存器rbx，此时Mod/R字节的值即为0xc3。类似的还有运算指令add，示例如图3-4。



图 3-4 add指令示例

当使用寄存器rdx作为基数，并将寄存器rcx作为索引时，可以看到SIB字节的值为0xca，而0xca可以被解析为远跳转的retn指令。这就导致了被攻击者利用其构造gadgets的风险。从以上两个例子可以看到，当寄存器rbx或rdx作为目标操作数时，包含它的指令就含有可以被解析为返回指令的数据。基于此，设计了基于寄存器替换的等效指令模板，通过引入寄存器r13或r14中转源操作数，使得产生的指令中不含有类似0xc3的数据。以mov指令为例，设计等效指令模板如图3-5。首先将寄存器r13入栈，其次将寄存器rax的值赋给r13，接着将寄存器r13的值赋给rbx，最后将寄存器r13出栈以恢复它原来的内容。可以看到替换指令的机器码均不含0xc3这样的数据，避免了可能被攻击者利用的风险。



图 3-5 指令替换-操作数替换示例

3.2.2 静态插桩的设计

静态插桩包括无效指令填充和自由跳转保护两部分。无效指令填充是指在指令前添加一串特殊的指令序列，通过这串指令序列可以强制一个指令片段的对齐执行，而不会由于攻击者从中间任意位置插入导致执行意料之外的指令序列。无效指令填充是为了对指令替换中没有相应模板的危险指令进行处理，使含有0xc3的指令全部得到替换或保护。如图3-6所示，实线框代表的是原始指令片段，其中第二条指令包含有0xc3。当从0x04开始解析指令时，执行的指令就变成虚线框上方所描述的指令片段，它包含了一个以返回指令ret为结尾的gadgets。但指令替换的设计中没有rolb指令的替换模板，这时就需要进行无效指令填充。



图 3-6 无效指令填充示例-原始指令片段

填充后的指令如图3-7所示，当在原有指令片段之间填充若干nop指令后，即使攻击者从0x04开始解析指令，由于nop指令的填充，使得第二条指令按照原来的解析方式执行，不会产生以ret指令结尾的gadgets。并且由于nop指令对程序的执行状态没有影响，在正常执行过程中，填充的nop指令也不会造成程序执行异常终止。



图 3-7 无效指令填充示例-填充后的指令片段

自由跳转保护是为了保护二进制文件中已存在的自由分支指令，包括每个函数末尾的返回指令ret，以及在函数中有时会出现的间接跳转指令jmp和call。如果在不更改程序行为的情况下，无法轻易的消除这些指令。此外通过语义上等效的指令片段直接对它们进行替换可能也无法解决问题。因为攻击者可以学习替换后的指令的规律，当找到指令所在位置后仍然可以执行替换后的指令来实现相同的功能。基于，此本文使用一个受现有堆栈保护机制（canary机制）启发的解决方案，通过使用一个短的指令片段来检测函数，来确保仅在从正确的入口点进入函数的情况下，才能执行自由分支指令或自由分支指令才能起到正确的作用。对于函数结束时的ret指令，使用一种有效的返回地址加密来保护ret指令。对于间接跳转指令jmp和call，在执行指令前进行返回地址是否加密的校验。从而避免攻击者直接调用自由分支指令进行控制流转移。

3.2.3 重编译的设计

重编译模块是为了将二次汇编产生的汇编代码输出保存至单独的文件中，并对其进行编译。这个过程需要校验原二进制文件的文件类型，机器位数以及采用的保护方式，并在编译时使用相应的参数，以保证生成的二进制文件与原文件在文件类型，保护方式以及功能上一致。如图3-8所示，通过开源工具checksec可以查看二进制文件的保护方式，可以看到Arch一行代表程序架构，“amd64”说明为64位；RELRO一行代表数据段保护技术，“Partial RELRO”说明为部分保护即GOT表可写；Stack一行代表栈溢出保护，“Canary found”说明开启了栈溢出保护；NX代表了内存执行保护，“NX enabled”说明堆栈不可执行代码；PIE代表程序运行时地址是否随机化，“PIE enabled”代表开启了地址空间布局随机化。得到二进制文件的保护方式后就可以在编译时添加相应参数以保证新的二进制文件与原文件保护方式一致。



图 3-8 二进制文件保护方式示例

# 第4章 基于二次汇编的ROP攻击防御实现

在上一章中，本文介绍了基于二次汇编的ROP攻击防御方法设计，包括指令替换、静态插桩以及重编译的设计目标、设计方案以及初步的效果展示。在这一章，本文将详细介绍上述方法的主要功能、使用的技术以及实现细节。

4.1 指令替换

指令替换模块通过对单条指令进行编译获取其机器码，判断其中是否存在类似0xc3的可以被解析为ret指令的数据，若存在则尝试对其进行指令替换。对于指令替换模块无法处理的指令，将对其进行静态插桩操作，即无效指令填充。指令替换的主要功能是通过等效的指令模板将含有0xc3数据的指令进行替换，从而减少gadgets数量，缓解ROP攻击，并将无法替换的指令交由静态插桩模块处理。

本文的方法基于开源的二次汇编框架Ramblr实现，从Ramblr的源码以及论文中可以知道二次汇编过程包括控制流图恢复、内容分类、符号化以及重汇编四个步骤。在重汇编过程中，二次汇编通过从二进制文件、函数、基本块以及指令四个级别由大到小，逐级进行反汇编。由于指令替换模块需要对单条指令进行分析，所以该模块的接口在指令反汇编的函数内实现。

指令替换模块的实现是通过在二次汇编框架的重汇编过程中添加接口，当单条指令完成反汇编后，调用接口完成指令替换的操作。该模块主要包括对指令中的立即数替换以及寄存器的替换两部分，流程图如图4-1所示。



图 4-1 指令替换流程图

首先对需要指令进行编译，获取指令的机器码。这里将指令添加汇编文件的起始标志“\_start”后保存为tmp文件夹下的一个临时文件，并对该文件进行编译。由于二次汇编框架反汇编的汇编格式默认为at&t，所以需要使用Linux中的汇编环境as对指令进行编译。编译之后使用Linux下反汇编可执行文件的命令objdump对产生的可执行文件进行反汇编得到汇编信息，经过对汇编信息的过滤得到指令的机器码。

获取到指令对应的机器码后判断机器码中是否含有0xc2、0xc3、0xca或0xcb其中之一。如图4-2所示，其中0xc2和0xc3均表示近距离返回指令，即返回的目标地址和当前所在地址位于同一段内，而0xca和0xcb则表示远距离返回指令。



图4-2 ret指令类别说明图

若指令的机器码中含有ret指令的机器码，则要进行下一步的判断，即指令的立即数中是否含有ret指令的机器码。若是，则进行基于立即数划分的指令替换，否则尝试进行基于寄存器中转的指令替换，若无法进行指令替换则传入静态插桩模块进行处理。

对于基于立即数划分的指令替换，本文实现的方法能够处理的指令类型有mov、add、sub、cmp、and、or和xor共七种指令。这里为了方便阐述实现的原理，将上述指令根据指令模板中是否需要引入中转寄存器分为两类。需要引入中转寄存器的指令有cmp和and指令。以and指令为例，其等效模板示例如图4-3所示。由于与操作的特殊性，若对立即数划分出来的两个数进行连续的与操作，那么得到的结果与原指令不符。所以需要引入一个中转寄存器，通过将中转寄存器分别与两个由立即数划分得到的数进行赋值和加操作，使得中转寄存器的值和立即数相等，然后对目标寄存器和中转寄存器进行与操作，得到和原指令相同的结果。当然在使用中转寄存器之前需要将它的值压入栈，以便在使用之后恢复寄存器的值。



图 4-3 指令替换-add指令示例

不需要引入中转寄存器的指令即为剩下的五种指令，都可以通过两条指令进行等效替换。其中赋值指令比较特殊，第一条指令通过将立即数划分的一个数直接赋值给目标寄存器，第二条指令通过add指令将目标寄存器加上剩余的数。而对于其它的运算指令（add、sub、or和xor指令），使用两条和原指令相同类型的指令对目标寄存器和立即数划分的两个数进行操作即可。

对于基于寄存器中转的指令，即立即数中不包含0xc3但指令中包含的情况，本文实现的方法支持的指令类型和基于立即数划分的指令替换相同。在上一章的设计方案中我们知道，由于原始指令中的目标操作数为寄存器rbx或rdx，亦或是32位寄存器ebx或edx，导致指令的机器码中含有0xc3这样能够解析为ret指令的数据。所以需要使用其它的寄存器中转源操作数，使得指令中不含0xc3。这里使用的指令模板和基于立即数划分中需要引入中转寄存器的指令模板类似，都是通过push指令将中转寄存器压入栈，然后将源操作数赋给中转寄存器，最后使用原指令对中转寄存器和目标寄存器进行操作，并恢复中转寄存器的值。只不过这里使用的中转寄存器为r13或r14两种。由于在上一章的3.2.1小节已有示例图，这里不再赘述。

最后，经过指令替换后的指令片段通过接口传回重汇编中的指令反汇编模块，并替代原有指令。

4.2 静态插桩

静态插桩模块通过在二次汇编的重汇编过程中分析指令，并在函数和指令级别进行插桩操作，对于指令替换模块无法操作但需要进行保护的指令，使用无效指令填充的方式，保证指令执行时的完整性，对于函数中的直接跳转指令ret以及间接跳转指令jmp和call，使用自由跳转保护的方式，在函数入口处对返回地址进行加密，在跳转前进行校验或解密，保证控制流的完整性。

静态插桩的实现和指令替换类似，通过在二次汇编的重汇编中添加接口，包括函数级别的反汇编以及指令级别的反汇编，在反汇编过程结束后将相应的汇编指令传入静态插桩模块，经过插桩操作后将汇编指令传回并覆盖原有的汇编指令。

对于无效指令填充，是通过在指令前插入若干无效指令使得指令在被攻击者利用时能够正常执行，而不会被分隔为多个指令解析。所谓的无效指令是指对程序运行没有任何影响的指令，常见的无效指令有nop指令，对同一个寄存器进行赋值操作的指令，如mov eax eax等。本文使用的无效指令为nop指令，因为该指令的结构十分简单，仅由一个字节0x90组成。无效指令填充方法实现的关键在于nop指令的数量，若nop指令的数量太少，那么攻击者依然可以发现一些意想不到的指令，这些指令会覆盖nop指令以及后续指令中任意数量的字节，在这种情况下执行过程将以不对齐的方式进行。而如果nop指令的数量太多，则会影响程序的性能。在IA-32架构中，当一条指令同时具有地址偏移和一个完全由0x90组成的立即数时，该指令的最大长度为8个字节。另外，该指令可以有一个ModR/M字节、一个SIB字节或一个值为0x90的操作码，三者只能同时取其一。根据以上分析，本文将无效指令nop的数量设置为9。另外，二进制文件中可能有许多指令需要填充无效指令，若直接在指令前添加9个nop指令，会使产生的二进制文件在执行时产生不必要的开销。因此，本文提出的无效指令替换的方法在nop指令序列前会加入一个相对跳转指令，跳转的偏移为11，包括跳转指令本身的大小2个字节以及9个nop指令的大小，从而在二进制文件正常执行时跳过nop序列。示例如图4-4所示。



图 4-4 无效指令填充示例图

对于自由跳转保护，是通过在函数的入口处加密返回地址，在直接跳转ret指令前解密返回地址，在间接跳转前校验返回地址，若攻击者直接调用跳转指令，将导致返回错误的地址或校验不通过，从而保护控制流的完整性。这里使用的加密方式是异或加密，因为异或运算的特点是对一个数来说，使用同一个数对其进行两次异或运算其结果不变。本文的自由跳转保护方法使用canary当做异或加密的密钥，canary的特点是随着程序的每一次运行它的值都会改变，并且难以被预测。对于每一个经过二次汇编的重汇编过程得到的函数，若函数以ret指令结束，则对其进行返回地址保护。首先在函数入口处插入如图4-5所示的汇编指令序列，

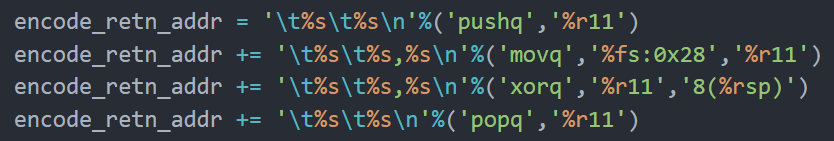


图 4-5 返回地址保护示例图

示例图中的指令架构为64位，对于32位的返回地址保护与其类似不再举例。可以看到，由于canary来自fs寄存器的偏移0x28处，若将其直接与栈上的返回地址进行异或操作，不符合汇编指令的设计。所以需要使用中转寄存器r11，通过将其压入栈以保存它原始的值，然后将canary取出并赋值给寄存器r11，最后对栈上的返回地址与寄存器r11进行异或，并将栈顶值弹出以恢复寄存器r11。这里返回地址在栈上的位置之所以是rsp+0x8，是因为在函数入口处返回地址应当位于栈顶即rsp处，但由于在使用返回地址之前将寄存器r11的值入栈，使得rsp指向了r11的值，产生了大小为8的偏移。完成在函数的入口处对返回地址的加密后，应当在函数的结束处即ret指令前添加对返回地址的解密，解密所需要的指令序列和加密使用的相同，只需将返回地址与canary再次异或即可。攻击者在使用ret指令时，ret指令通常只是作为gadgets的一部分，它与ret指令前的一些对寄存器进行操作的指令共同配合完成一个小功能。所以若攻击者直接调用以ret结尾的gadgets，使得没有加密过的返回地址与canary进行异或操作，导致控制流转移到一个无法预测的地方。

由于在返回地址保护时保证了以ret指令结束的函数的入口处对返回地址进行了加密，若函数中出现了间接跳转指令如jmp eax或call eax，则在这条指令前添加一段汇编指令序列。以64位可执行文件中的jmp rax指令为例，添加的指令序列如图4-6所示。

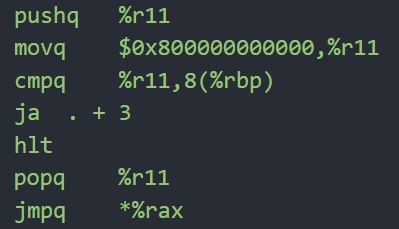


图 4-6 间接跳转保护-jmp指令

由于64位Linux系统中用户空间的地址范围最大为0x00007FFFFFFFFFFF，而canary的大小为8个字节，且高两个字节均不为0，所以经过加密的返回地址一定大于用户空间地址的最大值即0x00007FFFFFFFFFFF。本文的间接跳转保护正是依据这一点实现，首先将中转寄存器r11压栈，并将寄存器r11赋值0x800000000000，然后对栈上的返回地址即rbp+8的内容与寄存器r11比较，若返回地址大于等于寄存器r11的值，则跳转到pop指令恢复寄存器r11的值，否则执行hlt指令，程序停止运行。对于32位的可执行文件来说，其运行时地址的次高位字节一定为0xf7，所以可以依照和64位时类似的做法，比较返回地址的次高位字节是否为0xf7，若是那么表示返回地址没有进行加密操作，即攻击者直接调用了间接跳转指令，需要执行hlt指令使程序停止运行，否则跳转到下一条正常指令继续执行。通过插入图4-6所示的指令片段，使攻击者无法直接调用以间接跳转指令jmp结尾的gadgets，而会经过一个返回地址的校验，从而保护了程序控制流的完整性。

最后，经过静态插桩处理后的指令片段由函数或指令反汇编内的接口传回，并覆盖原有的指令序列。

4.3 重编译

重编译模块通过读取原文件的文件头获取二进制文件的位数，通过Linux的开源工具checksec获取二进制文件的保护方式，并在编译时添加相应参数，以保证生成的二进制文件的保护方式与原文件相同。重编译流程图如图4-7所示。

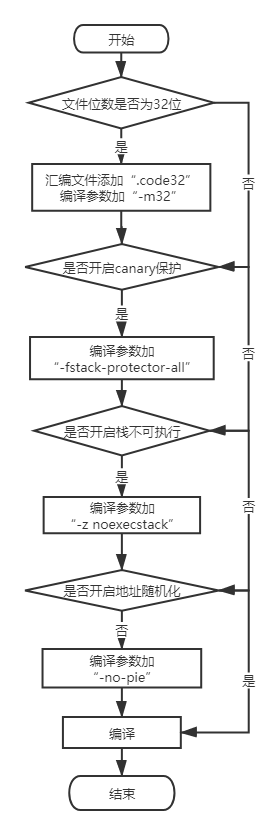


图 4-7 重编译流程图

对于Linux的可执行文件，其文件头的第五个字节标识了可执行文件的位数，若为0x01则表示文件位数为32位，若为0x02则表示文件位数为64位。需要注意的是，由于本文的方法基于64位的系统实现，若可执行文件为32位，则需要在汇编文件的文件头加上一行字符串“.code32”，以标识汇编文件为32位，并在编译参数上添加“-m32”。对于可执行文件保护方式，本文使用开源工具checksec自动化的获取并识别，它是使用shell语言编写的脚本，通过readelf命令获取可执行文件的信息后过滤关键信息来识别文件的保护方式 。经过checksec获取文件保护方式的信息后，需要对返回的信息进行识别过滤。若返回的信息中包含“Canary found”字符串，说明可执行文件开启了堆栈溢出保护，则在编译参数中加入“-fstack-protector-all”；若返回的信息包含“NX enabled”字符串，说明可执行文件开启了栈不可执行代码保护，则在编译参数中加入“-z execstack”；若返回的信息中心包含“No PIE”字符串，则在说明可执行文件没有开启地址空间随机化，则在编译参数中加入“-no-pie”。最后，使用gcc编译器及上述添加的编译参数对产生的汇编文件进行编译，得到经二次汇编处理的可执行文件。

# 第5章 测试及结果分析

本文在前两章中介绍了基于二次汇编的ROP攻击防御方法的设计方案以及实现细节，本章通过在Linux系统中使用coreutils工具包作为测试用例，来测试上述方法的实际

5.1 测试环境

测试过程使用的操作系统是deepin系统，版本为15.9，编程语言的环境为python 3.6。使用的二次汇编框架为Ramblr，它已经集成于开源工具angr中。

5.2 测试方案

本文的测试从可执行文件的性能、gadgets数量以及ROP防御效果三个维度进行，对二次汇编处理前后以及经ROP防御的二进制文件进行分析对比。

5.2.1 性能

二进制文件的性能测试包括文件的大小、执行特定任务所需的时间两部分。目的是对比经二次汇编处理前后的二进制文件的大小变化，以及经ROP防御前后二进制文件在执行同一任务所花费的时间开销的变化。

5.2.2 gadgets数量

gadgets数量的测试是为了对ROP防御处理前后的二进制文件中含有的gadgets数量进行分析对比，以测试指令替换对gadgets数量的影响。

5.2.3 ROP防御效果

ROP防御效果方面的测试是通过

5.3 测试结果分析

# 

结　论

## 总结

## 展望

# 致谢

参考文献

1. H. Shacham, “The Geometry of Innocent Flesh on the Bone: Returninto-libc Without Function Calls (on the x86),” in Proceedings of the 14th ACM Conference on Computer and Communications Security, 2007.
2. 国家信息安全漏洞库：<http://www.cnnvd.org.cn/>
3. Onarlioglu K, Bilge L, Lanzi A, et al. G-Free: defeating return-oriented programming through gadget-less binaries[C]//Proceedings of the 26th Annual Computer Security Applications Conference. 2010: 49-58.
4. Bletsch T, Jiang X, Freeh V. Mitigating code-reuse attacks with control-flow locking[C]//Proceedings of the 27th Annual Computer Security Applications Conference. 2011: 353-362.
5. Davi L, Sadeghi A R, Winandy M. ROPdefender: A detection tool to defend against return-oriented programming attacks[C]//Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security. 2011: 40-51.
6. Pappas V. kBouncer: Efficient and transparent ROP mitigation[J]. Apr, 2012, 1: 1-2.
7. Fratrić I. ROPGuard: Runtime prevention of return-oriented programming attacks[R]. Technical report, 2012.
8. Cheng, Yueqiang, et al. "ROPecker: A generic and practical approach for defending against ROP attack." (2014): 1.
9. Carlini, Nicholas, and David Wagner. "{ROP} is Still Dangerous: Breaking Modern Defenses." *23rd {USENIX} Security Symposium ({USENIX} Security 14)*. 2014.
10. Wang, Shuai, Pei Wang, and Dinghao Wu. "Uroboros: Instrumenting stripped binaries with static reassembling." *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. Vol. 1. IEEE, 2016.
11. Wang, Ruoyu, et al. "Ramblr: Making Reassembly Great Again." *NDSS*. 2017.
12. angr: <https://github.com/angr/angr>
13. One, Aleph. "Smashing the stack for fun and profit (1996)." *See http://www.phrack.org/show.php* (2007).
14. Designer, Solar. "Getting around non-executable stack (and fix)." *http://ouah.bsdjeunz.org/solarretlibc.html* (1997).
15. J. Kinder, “Static Analysis of x86 Executables,” Ph.D. dissertation, 2010.