BIOP:自动构造增强型ROP攻击

邢 骁 陈 平 丁文彪 茅 兵 谢 立

(南京大学软件新技术国家重点实验室 南京 210093)
(南京大学计算机科学与技术系 南京 210093)

摘 要 针对传统的代码注入和代码复用等攻击技术的缺陷,返回导向编程(Return-Oriented Programming,ROP)提出了复用以ret指令结尾的短指令序列,实现图灵完备编程的思想.ROP攻击可以绕开现有的针对代码注入的防御,且相比于传统代码复用技术,构造功能更为强大.但ROP攻击使用的ret指令结尾的指令序列具有明显的特征,这些特征导致ROP攻击容易被检测到.现有的ROP改进技术使用jmp指令结尾的短指令序列构造攻击,虽然消除了以ret指令结尾的特征,但同时引入了新的特征,且并不具有实用性.文中提出了一种分支指令导向(Branch Instruction-Oriented Programming,BIOP)攻击技术,使用jmp指令或call指令结尾的短指令序列构造攻击.相比于以前的工作,BIOP不引入新的特征,能有效避免现有的防御技术.同时我们分析并解决了构造攻击时寄存器的副作用,提出控制指令序列概念解决构造时内存冲突,实现自动化构造BIOP攻击.作者设计了一个自动化构造BIOP工具,构造了大量实际的BIOP shellcode,实验结果表明BIOP攻击可以绕过现有的ROP防御技术.

关键词 返回导向编程;指令序列;寄存器副作用;自动化;信息安全;网络安全
中图法分类号 TP309 **DOI号** 10.3724/SP.J.1016.2014.01111

BIOP: Automatic Construction of Enhanced ROP Attack

XING Xiao CHEN Ping DING Wen-Biao MAO Bing XIE Li

(*State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing* 210093)
(*Department of Computer Science and Technology, Nanjing University, Nanjing* 210093)

Abstract Return-Oriented Programming (ROP) is a technique which leverages the instruction snippets in existing libraries/executables to construct Turing Complete programs. Such technique can generate the shellcode which evades most code injection defenses. However, ROP attack is usually composed with gadgets which are ending in ret instruction without the corresponding call instruction. Based on this fact, several defense mechanisms have been proposed to detect the ROP malicious code. In this paper, we present Branch Instruction-Oriented Programming (BIOP) attack which uses the gadgets ending in jmp instruction or call instruction. This new technique, which uses jmp instruction or call instruction to replace the ret instruction, breaks the hypothesis of existing defense tools. Meanwhile we propose a tool to automatically construct the real-world BIOP attack, which as demonstrated in our experiment can bypass most of the exiting ROP defenses.

Keywords Return-Oriented Programming (ROP); gadget; side effect of register; automation; information security; network security

---

收稿日期:2011-03-31;**最终修改稿收到日期**:2014-01-09.**本课题得到国家自然科学基金**(61073027,60773171,90818022,61272078,61321491)**和国家"八六三"高技术研究发展计划项目基金**(2007AA01Z448,2011AA1A202)**资助**.**邢 骁**,男,1988年生,硕士,工程师,主要研究方向为系统安全.E-mail:xingxiao1122@gmail.com.**陈 平**,男,1985年生,博士,主要研究方向为系统安全.**丁文彪**,男,1988年生,硕士,主要研究方向为系统安全.**茅 兵**,男,1967年生,博士,教授,博士生导师,中国计算机学会(CCF)会员,主要研究领域为系统安全、软件安全.**谢 立**,男,1942年生,教授,博士生导师,主要研究领域包括分布式计算和并行处理.

1 引 言

传统的攻击方法通常使用代码注入和代码复用技术.现在流行操作系统中普遍部署的数据执行保护技术(如 Linux 系统中 PAX 补丁①和 Windows 系统中的 DEP②),可以对代码注入技术实施有效的防御.传统代码复用技术如 return-to-libc 攻击技术[1],依赖于库中的函数功能,攻击能力受到严重限制.且现有的防御方法如 Libsafe③,通过替换特定的库函数有效的防御该攻击.

返回导向编程(Return-Oriented Programming,ROP)技术是由 Shacham[2]在 2007 年提出的代码复用技术,它复用现有的库和可执行代码中以 ret 结尾的指令序列(gadget),通过将地址和数据压入栈,使程序从一个指令序列跳到另一个指令序列,实现任意的恶意行为.与传统的攻击方法相比,ROP 无需引入恶意代码,可以有效绕开数据执行保护技术,同时 ROP 更细粒度复用短指令序列,增加了攻击构造能力和检测难度,被认为是一种更强大的攻击技术④.

但由于 ROP 使用以 ret 结尾的指令序列构造攻击,这种明显的特征使 ROP 攻击容易被检测到,常用的检测方法有:(1)正常的程序中,call 指令和 ret 指令存在一一对应关系,而 ROP 使用单独的以 ret 结尾指令序列,call 指令和 ret 指令不匹配.利用这种特性,文献[3-4]分别从硬件和软件角度对 ROP 攻击进行检测;(2)ROP 攻击常常频繁复用多个 ret 结尾的短指令序列,文献[5-6]利用这个特性,检测是否存在多个连续的 ret 指令序列,以检测 ROP 攻击;(3)消除系统中库函数或可执行代码中的 ret 指令,使 ROP 攻击无法找到可用 ret 指令序列.文献[7]提供一种消除 ret 指令的编译器,该编译器编译的内核可以有效防止 ROP rootkit.

为了弥补 ROP 的这些缺陷,攻击者们对 ROP 技术进行一定程度的改进,如文献[8]中使用含 pop-jmp 这种类似 ret 功能的指令序列代替原有的以 ret 结尾的指令序列,文献[9]中试图寻找以 jmp 结尾的指令序列构造攻击,这类技术被称作跳转导向编程(Jump-Oriented Programming,JOP)技术.这两种方法虽然都可以绕过现有 ROP 防御,但仍然存在缺陷:(1)这两种构造方法仍具有明显特征,通过对 ROP 检测技术简单的修改即可以检测到这类攻击.如文献[8]中使用 pop-jmp 固定指令,可以通过检测这种 pop-jmp 的频率或语义分析即可以检测此类攻击,文献[9]中使用调度指令序列(调度指令序列)分发控制流,每次实现功能的指令序列执行完毕以后,都会跳回调度指令序列,继而转到下一个指令序列执行.这种控制流特征在正常程序中并不会出现,根据这种特征,动态检测很容易发现此类攻击;(2)由于使用固定类型的指令(pop-jmp、调度指令序列),这两种方法构造攻击的能力有限.它们都没有考虑构造时寄存器的副作用,只能手工构造特定的攻击.寄存器副作用指不同指令序列在使用同一个寄存器时会存在冲突,如文献[8]中使用 gadget1(pop eax;…),gadget2(…;jmp eax)进行构造时,gadget1 会给 eax 寄存器赋值,而这个值会在 gadget2 中使用.如果这两个 gadget 中间存在一个指令序列对 eax 的值进行改动,将会影响 gadget1 和 gadget2 之间的协作,攻击者需要寻找其他 gadget 来代替其中的某几个指令序列;同样,文献[9]中使用 add ebp, edi; jmp[ebp-0x39]作为调度指令序列,实现功能的指令序列中则无法对 ebp 和 edi 进行改动.寄存器副作用限制了这两类攻击的构造能力,使这两类方法都无法大规模自动化的构造攻击.我们会在第 3 节详细的分析寄存器副作用问题及其解决方法.

为了解决 ROP 及 JOP 技术的缺陷,构造隐蔽性更强、能力更为强大的攻击,我们试图解决以下问题:

(1)消除攻击代码的特征.我们期望用于构造攻击的指令序列不受限于特定的指令形式,如 ROP 使用以 ret 结尾的指令序列,JOP 中 pop-jmp、调度指令序列等,且程序控制流和正常程序类似.使检测此类攻击变的更为困难.

(2)从可执行程序或库中抽取出可用指令序列,利用这些指令序列自动的构造多种功能的攻击.现有的 JOP 技术都无法自动化构造,从而造成这类

① The pax project: http://pax.grsecurity.net/
② Microsoft. Windows XP Service Pack 2、Windows XP Tablet PC Edition 2005 以及 Windows Server 2003 中数据执行保护(DEP)功能的详细说明. http://support.microsoft.com/kb/875352
③ Libsafe. Avaya Labs Research Libsafe. http://www.research.avayalabs.com/gcm/usa/en-us/initiatives/all/nsr.htm&-Filter=ProjectTitle:Libsafe&-Wrapper=LabsProjectDetails&-View=LabsProjectDetails
④ Alexander Sotirov. Security Research. http://www.phreedom.org/

攻击并不实用.

为此，我们提出了一种自动化构造分支指令导向编程(Branch Instruction-Oriented Programming，BIOP)技术，它使用间接转移指令(间接跳转指令 indirect jump、间接调用指令 indirect call)结尾的指令序列代替 ROP 中以 ret 结尾的指令序列. 相比于 ROP 以及现有的 ROP 改进技术，这种新型代码复用技术不引入新的特征，攻击控制流与正常程序控制流类似，使检测更加困难. 同时我们分析了构造 BIOP 时遇到的问题，如寄存器副作用，内存冲突等. 我们提供了一种自动化构造 BIOP 的方法，指定构造规则以消除寄存器副作用，并引入控制指令序列以解决构造时内存冲突问题，使 BIOP 攻击具有实用性.

本文的贡献如下：

(1) 提出了一种新型的 BIOP 代码复用攻击，它打破了 ROP 攻击需要以 ret 结尾的局限，不引入新的特征，使现有 ROP 防御技术失效.

(2) 分析了 BIOP 攻击的可行性和完备性，并抽取 libc-2.3.5.so 库和 libgcj.so.5.0.0 库中的可用指令序列. 我们发现寄存器副作用是限制构造攻击的重要因素，提出自动构造 shellcode 的一般方法和规则，以有效的避免寄存器副作用. 同时我们提出控制指令序列，使用控制指令序列可以解决内存冲突、简化构造方法、优化构造代码.

(3) 设计实现构造攻击的自动化工具，并使用该工具构造大量的 BIOP shellcode. 实验证明我们的工具具有很强的实用性.

本文第 2 节详细介绍 BIOP 攻击并分析攻击的可行性，同时给出 BIOP 指令序列搜索算法；第 3 节介绍构造 BIOP 攻击的条件和自动化构造 BIOP shellcode 工具的原理；第 4 节给出使用工具构造 shellcode 的结果和构造的 shellcode 避开检测的能力；第 5 节介绍相关工作；第 6 节总结全文.

2　BIOP 概述

2.1　BIOP 基本组成单元

在这里，我们将 jmp 指令和 call 指令统称为转移指令，BIOP 的基本组成单元由以转移指令结尾的指令序列构成. 由于我们通过转移指令来控制程序的执行，而直接寻址指令跳转的偏移是固定的，所以这里我们只考虑两种情况：分别以 indirect call 指令和 indirect jmp 指令作为 BIOP 指令序列的结尾.

如图 1 所示，图 1(a)描述了传统 ROP 攻击，它执行一系列以 ret 结尾的短指令序列，这些指令序列中 ret 指令并没有调用指令与其相匹配. 现有的 ROP 防御技术通过检测这类特征可以有效的阻止 ROP 攻击. 如果将构造 ROP 攻击的以 ret 结尾的指令序列更换为以 jmp 或 call 结尾的指令序列，则可以有效绕过现有的 ROP 防御技术. 注意，简单的使用 jmp 或 call 来代替 ret 可能会产生问题，如使用 pop-jmp 来代替 ret，虽然可以消除 ret 的特征，但是 pop-jmp 的特征同样明显，防御者只需要对 ROP 防御技术进行简单的修改就可以有效防御此类攻击. 为此，我们提出 BIOP 攻击，它的基本组成单元是以 jmp 指令和 call 指令结尾的指令序列，如图 1(b)所示. BIOP 通过 jmp 指令和 call 指令来传递控制流，消除 ROP 中使用孤立 ret 指令的缺陷. 同时，BIOP 中包含一类联合指令序列，这类指令序列是由一个含 call 指令的指令序列和一个含 ret 指令的指令序列构成，call 指令用来调用 ROP gadget，然后由 ROP 指令序列中的 ret 指令返回. 这样，在 BIOP 中，每一个 ret 指令都有一个 call 指令与之对应. 由图 1 可以看出，传统的 ROP 攻击中 ret 指令作用是将控制流传递给下一个指令序列，所以 ROP 中由栈来控制程序的执行流程. 在 BIOP 中，控制流的传递由间接转移指令完成，控制流的传递依赖于间接转移指令中的寄存器的值. 所以 BIOP 的构造不依赖于栈的控制而依赖于寄存器，每次控制流从一个指令序列传递到另一个指令序列时必须先将 jmp 指令后的寄存器赋值，这导致指令序列之间具有一定的约束关系. 同时我们发现新型的 BIOP 攻击与库中现有的程序控制流类似，不会产生明显的特征.

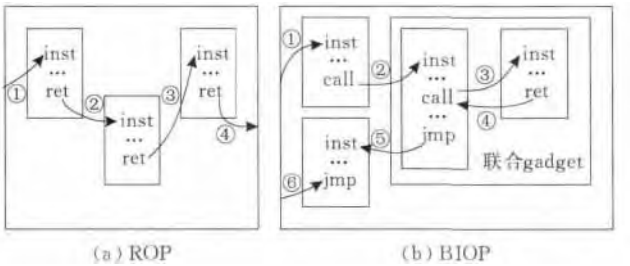

图 1　ROP 与 BIOP 控制流跳转的比较

2.2　构造 BIOP 可行性

本节我们研究普通的库和可执行代码中构造 BIOP 的可行性. x86 结构中，indirect call 指令和

indirect jmp 指令的第 1 个比特都是 0xff，对于 indirect call 指令，第 2 个比特是在 0x10 到 0x1f，0x50 到 0x5f，0x90 到 0x9f 和 0xd0 到 0xd7 的区间内，而 indirect jmp 指令的第 2 个比特是在 0x20 到 0x2f，0x60 到 0x6f，0xa0 到 0xaf 和 0xe0 到 0xe7 的区间内. 我们对两个常用的库 libc-2. 3. 5. so(C 库)和 libgcj. so. 5. 0. 0(java 运行动态链接库)进行分析，统计 ret 指令、indirect call 指令和 indirect jmp 指令的个数. 统计结果如图 2 和表 1 所示. 图 2 给出 libc-2. 3. 5. so 和 libgcj. so. 5. 0. 0 中 0xff 之后字节分布，表 1 给出了库中指令数目的统计. 由表 1 可看出，在大小为 1 489 572 字节的 libc-2. 3. 5. so 库和大小为 41 090 990 字节的 libgcj. so. 5. 0. 0 库中，indirect jmp 指令分别有 3723 个和 20 462 个，indirect call 指令分别有 3243 个和 31 940 个，数目少于 ret 指令，但仍可以提供足够的 gadget.

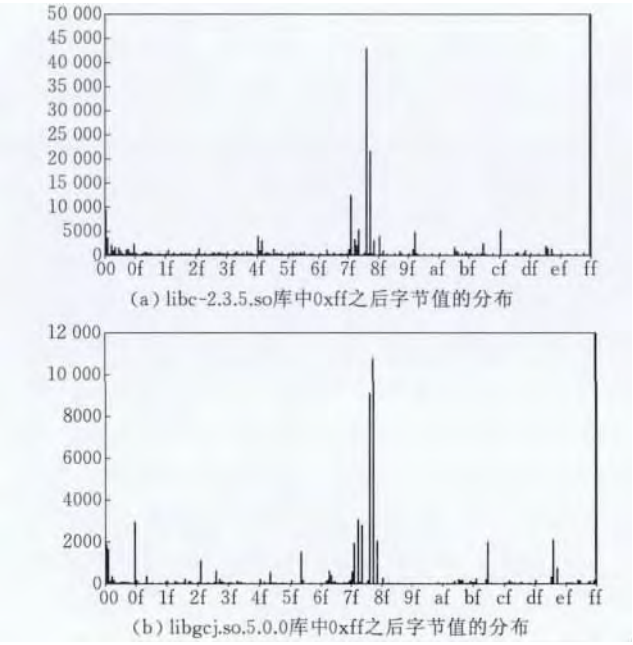

图 2 库中 0xff 之后字节值的分布

表 1 库中转移指令数目统计

		libc-2. 3. 5. so		libgcj. so. 5. 0. 0	
		数目	占总百分比/%	数目	占总百分比/%
0xc3(ret 指令)		5790	0. 39	58299	0. 14
0xff	call	3243	0. 22	31940	0. 08
	jmp	3723	0. 25	20462	0. 05
库大小		1489572	100	41090990	100

2. 3 BIOP 指令序列快速搜索算法

为了快速有效找到可用指令序列，我们提出 BIOP 指令序列快速搜索算法，如算法 1 所示. 对于输入的一段二进制代码，算法构建一个指令序列树集合，集合中每个树根节点为指令序列尾部的 jmp 或 call 指令，代码中每条指令是它上一条指令的父节点，当某条指令前面二进制代码序列可以被解释成多条不同指令时，这条指令将会有多个不同的孩子节点，每个节点存放这些不同的指令.

算法 1. BIOP 指令序列搜索.

```
JOP gadget Searching Algorithm
input: textseg          output: G
    create a set G
    for pos from 1 to textseg_len do
        if bytes at pos decode as a unconditional
        branch instruction branch_insn then
                put branch_insn into G
                BuildTree(pos, branch_insn)
        end if
    end for

Procedure BuildTree(index pos, instruction parent_insn)
        for step from 1 to max_insn_len do
            if bytes [(pos-step)…(pos-1)] decode as
            a valid instruction insn then
                if Viable(insn) is true then
                    ensure insn as the child of parent_insn;
                        BuildTree(pos-step, insn)
                end if
            end if
        end for

Procedure Viable(instruction insn)
        if insn is a branch and insn
        is not a call instruction then
                return false
        end if
        return true
```

该算法描述如下：首先创建一个集合 G，用于放置指令序列树，树的根节点为非条件转移指令序列. 然后，pos 指针指向二进制代码的第一个 byte，并由此位置依次向后搜索，当 pos 指针指向的 byte 序列可以被解释成一条非条件转移指令时，将这个指令序列放入集合 G 中，调用 BuildTree() 函数构建非条件转移指令树. 构建的方法是从 pos 指针指向的位置开始，依次向前搜索 1 到 max_insn_len 个比特，这里 max_insn_len 是指所在机器结构(如 x86 系统)中最大指令的比特数. 如果从 pos-1 位置到 pos-step 位置的二进制序列可以被解释成一个合法指令 insn，则调用 Viable() 函数判断它是否是转移指令，因为转移指令可以改变控制流，BIOP 只能由指令序列最后的非条件转移指令控制程序执行，不允许 insn 为转移指令. 注意，这里将 call 指令排除在分支指令外，因为 call 指令可以构成联合指令序列. 如果 insn 不是 call 之外的间接转移指令，那么将 insn 作为 parent_insn(即 pos 所指向的非条件转移指令)的一个孩子节点，然后递归调用 BuildTree

来构建 insn 的孩子节点.

算法生成的指令序列中，相邻两条指令的后一条指令在树中是前一条指令的父亲节点. 例如指令序列 00 58 58 ff e1，可得到如图 3(a)一个子树. 通过这个子树，可得到图 3(b) 4 个 BIOP 指令序列. 所以，使用该算法得到的树集合，从树的某个节点到根节点的指令序列就是一个 BIOP 指令序列.

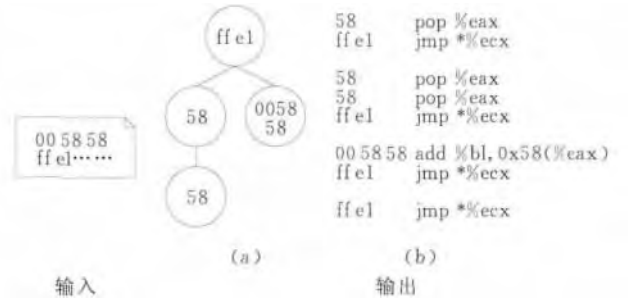

图 3　搜索算法生成子树

2.4　BIOP gadget 实例

本节我们研究在 libc-2.3.5.so 和 libgcj.so.5.0.0 两个库中找到的 BIOP 指令序列，并且依据它们可以实现的功能对其分类. 通过实例可看出，库中存在的指令序列可以实现大部分的 x86 操作，如数据传送、算术运算、逻辑运算、分支语句和系统调用等等.

2.4.1　数据传送

数据传送指令的功能是将数据或地址传送到寄存器或存储器单元中，如 pop %edx 指令，可以使用如下两个指令序列实现：

```
5e          pop   %esi
ff 66 00    jmp   *0x0(%esi)
gadget(1)
5a          pop   %edx
f5          cmc
ff 66 83    jmp   *-0x7d(%esi)
gadget(2)
```

上述两个指令序列运行如图 4 所示，首先栈内设好传递给 esi 寄存器的值和递给 edx 寄存器的值 0xdeadbeef，传递给 esi 寄存器的值是内存空间某个地址. 当程序执行 gadget(1)时，pop %esi 将栈顶值传递给 esi，此时 esi 指向的内存中存放 gadget(2)地址，当执行 jmp *0x0(%esi)时，程序跳转到 gadget(2)，执行 pop %edx，将 0xdeadbeef 传递给 edx 寄存器. -0x7d(%esi)中存放着下一个要跳转的指令序列地址，执行完 gadget(2)，程序自动跳转到下一个设定的指令序列.

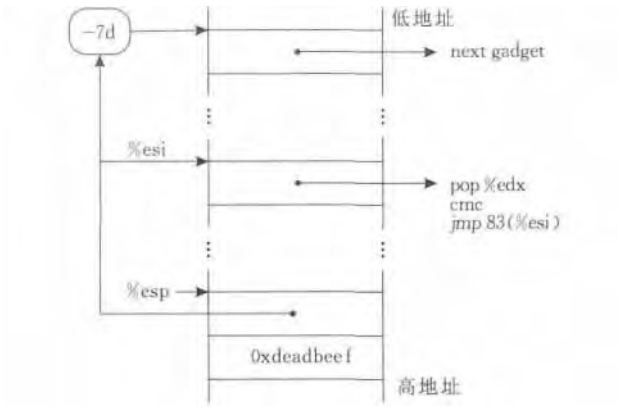

图 4　传递数据到 edx 寄存器

数据传送指令中的标志传送指令用于传送标志器寄存器 eflag 中的标志位，以便设置、修改和保存 eflag 寄存器中的标志. 如 libgcj.so.5.0.0 中的 gadget(3)：

```
9c          pushf
ff 62 00    jmp   *0x0(%edx)
gadget(3)
```

2.4.2　算术运算和逻辑运算

对于算术运算和逻辑运算，先使用数据传送指令序列将操作数传入寄存器或内存中，然后使用对应的指令序列完成算术运算和逻辑运算. 例如 libgcj.so.5.0.0 中的 gadget(4)：

```
0b d5       or    %ebp, %edx
ff 20       jmp   *(%eax)
gadget(4)
```

2.4.3　分支语句

在 BIOP 中我们使用类似 jmp *(reg1,reg2,1) 或 call *(reg1,reg2,1)的指令序列实现分支指令. reg1 中放置地址偏移，它的值随着标志寄存器不同而不同，reg2 放置基地址. 如 libgcj.so.5.0.0 中的 gadget(5)：

```
ff 24 32    jmp *(%edx, %esi, 1)
gadget(5)
```

图 5 是一个简单的条件跳转实例，它可以实现 jz 指令或 jnz 指令，用到指令序列序列如下：

```
9c          pushf
ff 62 00    jmp   *0x0(%edx)
gadget(3)
5a          pop   %edx
f5          cmc
ff 66 83    jmp   *-0x7d(%esi)
gadget(2)
```

```
0b d5       or      %ebp, %edx
ff 20       jmp     *(%eax)
gadget(4)
ff 24 32    jmp     *(%edx, %esi, 1)
gadget(5)
```

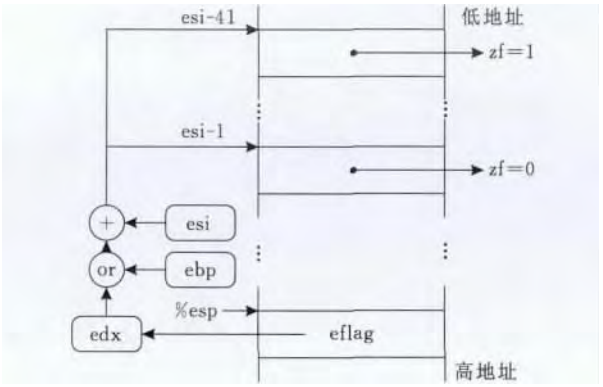

图 5 实现 jz 指令或者 jnz 指令功能

我们假设数据传输指令序列已经设置好 eax、edx、ebp 和 esi 的值，同时将 ebp 的值设置为 0xffffffbf. 程序运行时，首先将标志寄存器的值入栈，然后传递给寄存器 edx，将 edx 的值与 ebp 的值进行或操作，并将结果保存在 edx 中. 当保存在 edx 中的标志寄存器值的 zf 位为 0 时，得到的值为 0xffffffbf，jmp *(%edx, %esi, 1)相当于执行 jmp *−0x41(%esi). 当 zf 位为 1 时，得到的值为 0xffffffff，相当于执行 jmp *−0x1(%esi). 这样就根据 zf 位的不同而跳转到不同的地方. 注意，使用这种方法实现 jc 或 jnc 等与 cf 标志位相关的指令时，会出现 jmp *−0x2(%esi)和 jmp *−0x1(%esi)地址重叠的情况，针对这种情况，我们使用左移位指令序列或者乘法运算指令序列去放大地址之间的距离.

2.4.4 系统调用

在 BIOP 中我们一般使用联合指令序列实现系统调用功能，如 gadget(6)：

```
ff 56 54      call *0x54(%esi)
ff 65 e8      jmp *−0x18(%ebp)
65 ff 15 10 00 00 00
              call %gs:0x10(,0)
c3            ret gadget(6)
```

首先使用数据传送指令序列设置 esi 和 ebp 寄存器的值，0x54(%esi)内存放的是 lcall %gs:0x10(,0)的地址，−0x18(%ebp)中存放下一个指令序列地址. 然后用数据传送指令序列将需要传递的参数传递给对应的寄存器，执行这个指令序列就可以实现系统调用功能.

2.5 BIOP 构造形式

图 6 描述了 BIOP 的构造形式，BIOP 由多个以 call 或 jmp 结尾的指令序列串联起来构成，每个指令序列在执行完毕以后都会跳转到下一个指令序列开头，多个指令序列内部的指令的组合可以实现特定的语义. 首先，我们看到这类指令序列并不存在特殊的指令类型，如文献[8]中必须使用 pop 指令为以后的 jmp 指令中寄存器赋值从而确定跳转地址的情况. 在这里，给 jmp 指令中寄存器赋值的指令可以有多种类型，如 mov、add 等. 其次，区别于文献[9]中使用调度指令序列管理分发控制流，BIOP 攻击的控制流与普通程序的控制流类似，每个指令序列执行完毕后，都会经过 call 或 jmp 指令将控制流传递给下一个指令序列. 而在普通程序中，使用 call 指令或 jmp 指令跳转到其他指令序列这种情况也普遍存在. 所以 BIOP 攻击的程序更类似于正常程序，不易被检测到.

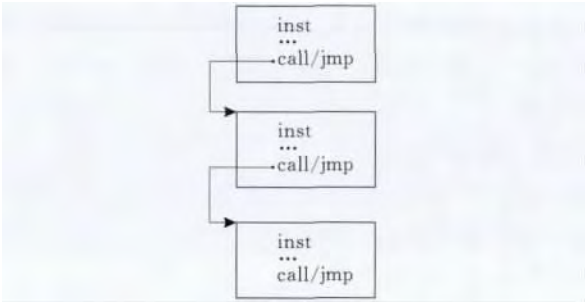

图 6 BIOP 构造形式

3 自动构造 BIOP 攻击

本节将对自动 BIOP 构造时出现的问题进行探讨，并给出解决方法. 同时在本节我们会介绍自动化构造工具的基本原理.

3.1 BIOP 寄存器副作用对攻击构造的影响

在构造 BIOP 攻击时，会出现多个指令序列使用到同一个寄存器，导致寄存器在赋值和使用时出现数据不一致的问题. 这类问题同样在传统的 ROP 和 ROP 改进工作中出现，传统的方法是人工分析构造的指令序列序列是否可行，导致 ROP 和 JOP 不具有实用性. 为此，我们在本节深入探讨造成这种原因的主要因素. 我们把构造中寄存器使用时冲突，称之为寄存器副作用. 导致寄存器副作用的原因有

3 种，(1) 用某个寄存器传递控制流和将该寄存器用作其他用途时产生的冲突；(2) 两个指令序列用同一个寄存器传递控制流跳时产生的冲突；(3) 指令序列本身某些指令会改变已设寄存器的值. 我们以文献[9]的调度指令序列为例，在程序中的每个功能指令序列都会跳转到调度指令序列，然后由这个调度指令序列决定下一个要使用的功能指令序列. 如下是某个理想的调度指令序列：

```
83 c2 04 add     $0x4, %edx
ff 22            jmp     *(%edx)
```

对于第 1 种原因，有多种可能的情况. 如在这个指令序列中，需要使用 edx 寄存器确定指令序列要跳转的地址. 如果程序中需要用到 edx 作为参数的系统调用，那么我们需要一个数值传递指令序列将参数传递给 edx 寄存器，此处如果仍然使用调度指令序列，就会产生错误. 这在很大程度上限制了构造的能力.

对于第 2 种原因，例如在程序中使用了如下 gadget(8). 由于都是用 edx 寄存器实现指令序列的跳转，所以调度指令序列和 gadget(8) 不能在一起使用.

```
8b 61 10     mov 0x10(%ecx), %esp
ff e2        jmp     %edx
gadget(8)
```

同样，我们也用如下的 gadget(9) 来解释第 3 种原因. 当使用这个指令序列对 al 进行与操作时，cdq 指令会改变 edx 值，导致调度指令序列不能继续使用.

```
24 e8        and     $0xe8, %al
99           cdq
ff e1        jmp     *%ecx
gadget(9)
```

所以使用类似调度指令序列会严重影响 JOP 构造的能力. 同样，这种寄存器副作用也存在于 BIOP 程序中.

3.2　消除寄存器副作用

由于 BIOP 程序中存在的寄存器副作用，导致 BIOP 的构造具有一定的困难和挑战，为了解决这个问题，我们在本节给出两个构造攻击的规则以消除寄存器副作用和一个指令序列排列算法以实现自动化选择和排列指令序列.

规则 1.　每个寄存器在使用之前必须有对应的赋值.

我们根据对某个寄存器的操作将指令序列分为赋值指令序列和应用指令序列两类，赋值指令序列将某个数据传入该寄存器，应用指令序列则使用该寄存器完成特定操作. 如 3.4.3 节中实现 jz 或 jnz 功能的指令序列组，对于 edx 寄存器来说，gadget(2) 是赋值指令序列，gadget(4) 相对于 gadget(5) 是赋值指令序列，相对于 gadget(2) 是应用指令序列，gadget(5) 是应用指令序列.

规则 2.　使用寄存器前，最近给该寄存器赋值的指令序列的赋值结果，必须是使用该寄存器时所需要的值.

如对于 gadget(9) 是赋值指令序列，而紧随其后是需要用 eax 值为 0x1 的系统调用指令序列，由于任何数和 0xe8 进行与操作的结果不为 0x1，这种情况违反了规则 2. 通过满足规则 2 确保每次指令序列所用到的寄存器的值都是符合预期的.

为了实现 BIOP 构造的自动化，我们提供了一个指令序列排列算法(算法 2)来解决这个问题. 如图 8 所示.

算法 2.　gadget 排列算法.

```
Load Gadgets Selection Algorithm
Lr: Load registers in gadget
Li: Limited registers in gadget
R: The candidate registers need to be set
T: Gadgets set which contains the selected
    load gadgets for R
G: All load gadgets get form lib
input: R, G;          output: T
while R !=NULL do
  The rule of select gadget in G as follows:
  from MAX to 1, registers of gadget.Lr∩R do
    If gadget.Li∩R==NULL do
      Remove registers in both Lr and R
      Put gadget in T
    end if
  end from
end while
```

在算法 2 中，我们对搜索到的每一个具有 Load 功能的指令序列定义赋值寄存器集合 Lr 和受限的寄存器集合 Li. 以 gadget(2) 为例，gadget(2).Lr 为 {edx}，gadget(2).Li 为 {esi}. 注意，如果在指令序列中某个受限寄存器之前已经有给这个寄存器赋值的指令，则不把这个寄存器加入 Li 中. 如在 gadget(1) 中的 Lr 为 esi，Li 为 $\varnothing$.

我们对所有的指令序列按赋值寄存器与待赋值寄存器集合并集元素个数从多到少进行搜索，找出 gadget.Li 与集合 R 并集为空的指令序列，将赋值的寄存器从集合 R 中移除，并将指令序列放入集合

T，当 R 中不再存在寄存器时，T 中存放的是实现赋值功能的指令序列. 这个算法保证在每次将某个寄存器从 R 中移除时，都已经给这个寄存器赋值，在此之后的赋值指令序列都不会改变这个寄存器.

3.3 BIOP 的图灵完备性

Shacham[2]认为 ROP“从直觉上”看是图灵完备的，通过一些技巧可以实现任意功能. 实际上 ROP、JOP 与 BIOP 的能力受限于系统中的函数库或者可执行代码，如果某个功能在库或可执行代码中找不到对应的 gadget，同时也无法使用其他技巧来实现时，可能这个操作就无法用 ROP 或 BIOP 来实现. 通过上述讨论我们知道 BIOP 的构造存在一些限制，如系统调用指令序列，我们并没有找到 syscall-jmp 这类指令序列，所以我们只有通过联合指令序列实现这类功能. 但是如果某个系统调用需要用到多个寄存器传递参数，那么使用联合指令序列来实现系统调用也不是一个理想选择. 当然，这是一个开放性问题，是否有其他技巧可以实现类似功能，我们在此并不讨论.

3.4 控制 gadget

本节我们提出一种特殊的控制指令序列，用于解决 BIOP 构造中可能出现的地址冲突问题和内存安排问题. 这类指令序列对于寄存器是一种自省控制方式，它可以在指令序列内部同时实现对寄存器的赋值和使用该寄存器控制将要跳转的地址，理想状态下并不影响其他的寄存器.

图 7 是控制指令序列的原理图. 在图 7 中，gadget(1)是 libgcj. so. 5. 0. 0 库中一个典型的控制指令序列，它利用栈对 esi 进行赋值同时使用 esi 控制指令序列的跳转，且它很少影响其他寄存器.

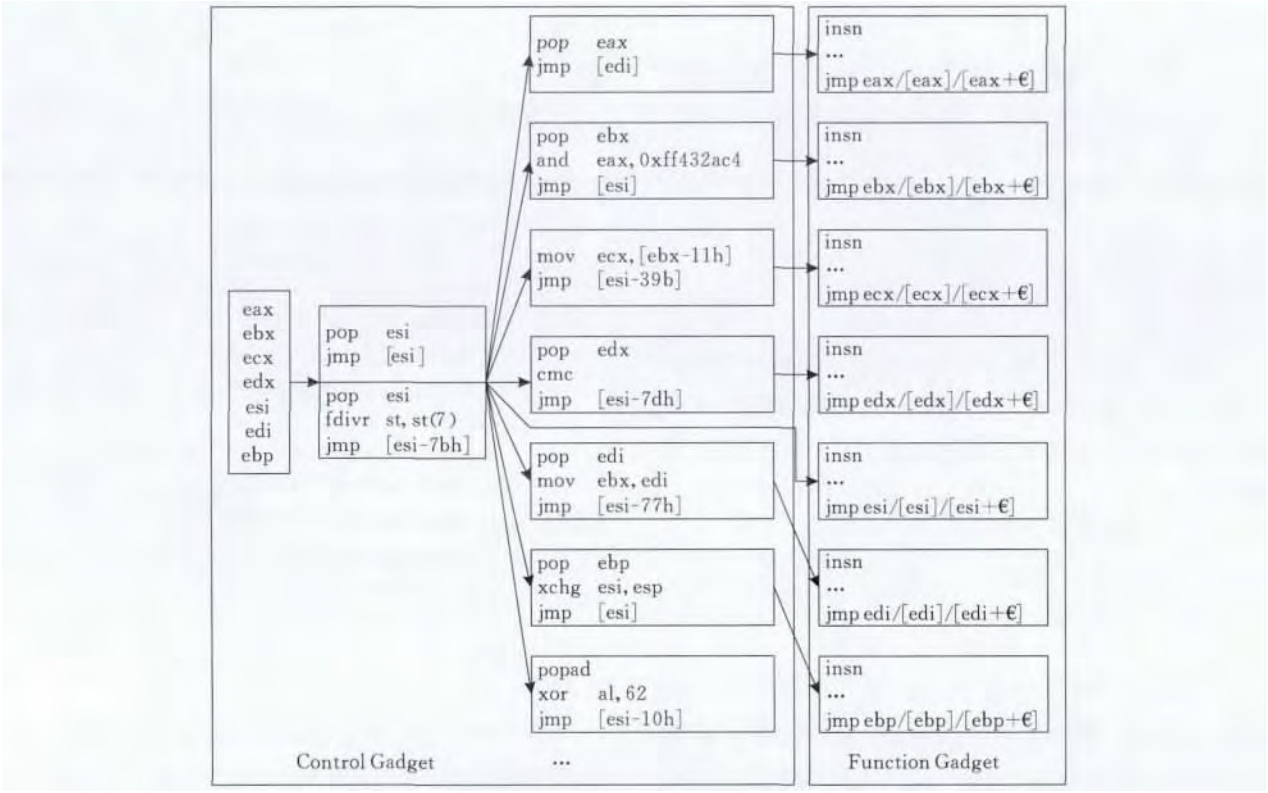

图 7 控制 gadget 原理

在构造 ROP 与 BIOP 时，还存在一类地址冲突问题，例如 gadget(10)和 gadget(11)，由于 jmp 的跳转地址均存放在－0x77(%esi)地址空间中，所以在连续使用这两个指令序列时会出现地址冲突问题，如果使用以 call 指令结尾的指令序列，call 指令运行时会将指令地址入栈，从而占用当前栈的空间，使此指令序列之后的指令无法从这个栈内取值. ROP 中通过栈控制程序执行，并不存在这类问题，但是 BIOP 中，由于 BIOP 指令序列的数量有限，这类问题便显得尤为突出. 通过在 gadget(10)和 gadget(11)之间插入控制 gadget(1)，改变 esi 寄存器的值，从而解决地址冲突.

```
97          xchg    %edi, %eax
38 f4       cmp     %dh, %ah
ff 66 89    jmp     *-0x77(%esi)
gadget(10)

89 7b fe    mov     %edi, -0x2(%ebx)
ff 66 89    jmp     *-0x77(%esi)
gadget(11)
```

在 BIOP 中，同时还存在内存空间安排问题，例如 gadget(11)和系统调用 gadget(6)，他们都是用 esi 寄存器控制程序跳转，如果连续使用这两个指令序列，那么在 esi－0x76 到 esi＋0x53 这段内存中的某些空间将会被闲置. 通过使用联合 gadget(1)，改变 esi 的值，可以使闲置空间缩短为 esi＋0x1 到 esi＋0x53.

3.5　BIOP 构造自动化工具

通过上述分析，我们设计实现了一个自动化构造 BIOP 攻击的工具，原理如图 8 所示.

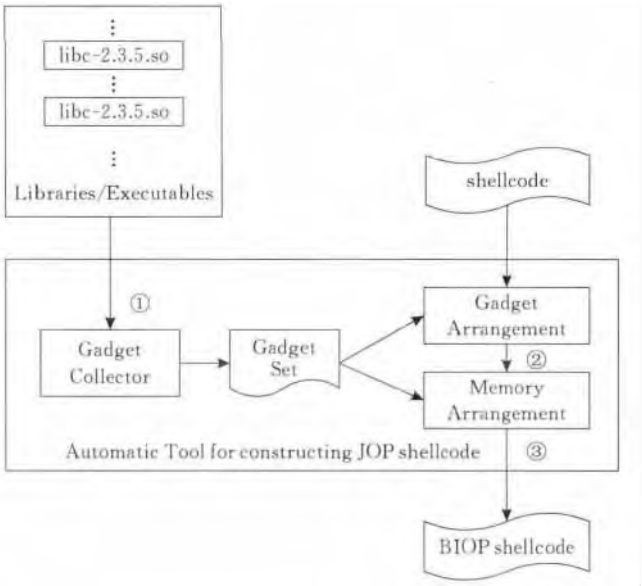

图 8　自动化构造 BIOP 原理

我们的工具主要由指令序列收集器(①)、指令序列排列器(②)和内存优化器(③) 3 部分组成. 指令序列收集器通过算法 1(BIOP gadget Searching Algorithm)从库或可执行文件中收集可用的指令序列. 指令序列排列器通过对 shellcode 的分析，找出对应功能的指令序列，并通过算法 2(Load Gadgets Selection Algorithm)实现各个功能中寄存器的赋值. 内存优化器使用控制指令序列解决指令序列排列器生成的指令序列中的内存冲突和空间安排问题，并生成所最终的 BIOP shellcode.

4　实验与评估

4.1　BIOP shellcode 实例

本节给出一个使用自动化工具构造的 shellcode 实例，功能是通过执行 execve 系统调用运行一个 shell. 这个 shellcode 具体实现如下：在执行系统调用之前，将系统调用号 0xb 传递给 eax 寄存器，执行程序的路径“/bin//sh”地址传递给 ebx 寄存器，参数向量传递给 ecx 寄存器(参数向量指向两个指针的队列，一个是指向“/bin//sh”字符串，另一个是空指针)，环境向量(空指针)传递给 edx 寄存器.

具体实现方法如图 9 所示，图中左侧给出 BIOP 的内存安排和内存地址的后 12 位. 右侧虚线代表了指令序列的执行循序. 通过一次缓冲区溢出，将返回值地址用 gadget(12)所在的地址改写，程序在返回时跳转到我们设定的指令序列，然后循环 3 次使用 gadget(12)和 gadget(13). 对于 gadget(13)，把 edi 和 ebx 指向的内存设定好相应的值，相加后会将结果放入 ebx 指向的内存中. 这在构造攻击中可以解决 0 比特的问题，因为在缓冲区溢出中，不允许有 0x0 的存在. 如给 eax 赋值为 0xb，首先使用 gadget(12)将 edi 设置为 0xeeeeeee0，ebx 指向的空间设置为 0x1111112b，这个空间以后将会给 eax 赋值，所以在执行 gadget(13)之后，ebx 指向空间结果为 0xb. 同样在 shellcode 中，第 2 次和第 3 次使用是将相应的内存空间赋值为 0x0. 使用 gadget(12)和 gadget(14)给 eax、ebx、ecx、edx、esi、edi、ebp 7 个寄存器赋好相应的值，在这里给 eax 赋值的内存恰好是在通过 gadget(13)设置的，在运行 gadget(14)后，eax 的值是系统调用号 0xb. 最后执行系统调用完成 execve 启动一个 shell. 在内存中，我们没有用到的闲置内存，全部用字符”A”填充.

```
61          popa
34 62       xor $0x62, %al
ff 60 f0    jmp *-0x10(%esi)
gadget(12)

03 0f       add %edi, (%ebx)
ff 66 00    jmp *(%esi)
gadget(13)

58          pop %eax
ff 67 00    jmp (%edi)
gadget(14)
```

BIOP shellcode 二进制代码如图 10 所示.

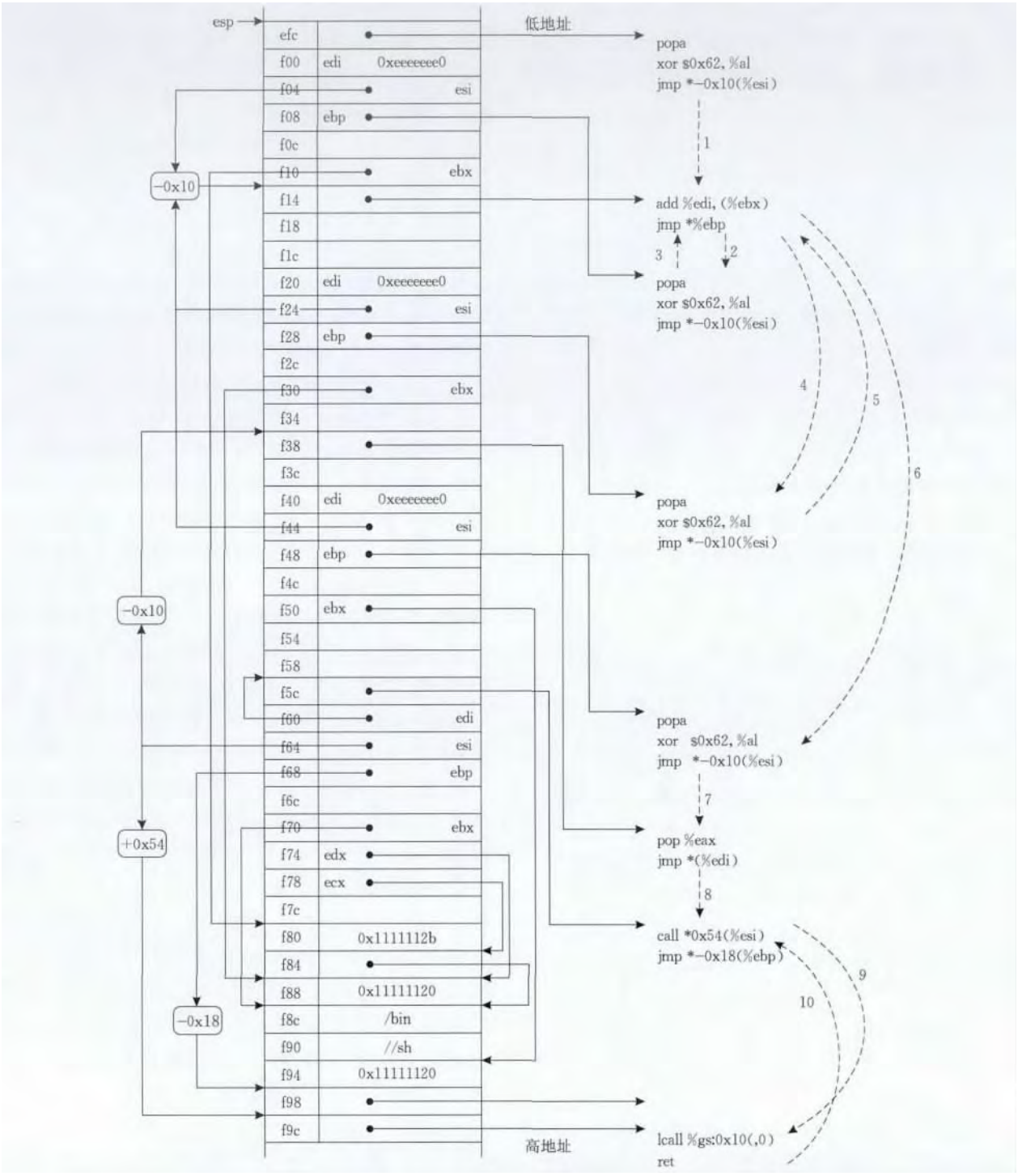

图 9 BIOP shellcode **构造**

```
[root@fedora13 test]# hexdump -C _bin_sh_shellcode
00000000  04 7a 0e 27 ee ee ee e0  4f ff ff 24 04 7a 0e 27  |.z.'....O..$.z.'|
00000010  aa aa aa aa 4f ff ff 80  05 52 4a 5d aa aa aa aa  |....O....RJ]....|
00000020  aa aa aa aa ee ee ee e0  4f ff ff 24 04 7a 0e 27  |........O..$.z.'|
00000030  aa aa aa aa 4f ff ff 88  aa aa aa aa 03 8c 1d c0  |....O...........|
00000040  aa aa aa aa ee ee ee e0  4f ff ff 24 04 7a 0e 27  |........O..$.z.'|
00000050  aa aa aa aa 4f ff ff 94  aa aa aa aa aa aa aa aa  |....O...........|
00000060  03 5a dd fa 4f ff ff ec  4f ff ff 48 4f ff ff b0  |.Z..O...O..HO...|
00000070  aa aa aa aa 4f ff ff 8c  4f ff ff 88 4f ff ff 84  |....O...O...O...|
00000080  aa aa aa aa 11 11 11 2b  4f ff ff 8c 11 11 11 20  |.......+O...... |
00000090  2f 62 69 6e 2f 2f 73 68  11 11 11 20 de ad be ef  |/bin//sh... ....|
000000a0  03 08 d1 a4                                       |....|
000000a4
```

图 10 BIOP shellcode **实例**

4.2 BIOP shellcode

我们从网站[①]中选取了 22 个 shellcode 进行构造，使用常见的 libc-2.3.5.so 库和 libgcj.so.5.0.0 库，实验环境是 2.6.15 版本内核的 Fedora 5 系统. 22 个 shellcode 中包括数据传输、算术运算、系统调用和条件转移等基本语句. 实验结果如表 2 所示.

表 2　BIOP shellcode

	功能	shellcode 大小(bytes)	BIOP shellcode	
			指令序列数目	大小(bytes)
1	chmod("//etc/shadow",666) exit(0)	30	14	324
2	killall5 shellcode	34	8	168
3	PUSH reboot()	30	8	168
4	/sbin/iptables -F	40	13	300
5	execve(rm -rf /) shellcode	45	15	381
6	execve("/bin//sh")	25	8	164
7	normal exit w/random return value	5	4	104
8	setreuid(getuid(),getuid()),execve("/bin/sh",0,0)	34	24	503
9	edit /etc/sudoers for full access	86	34	695
10	system-beep shellcode	45	22	495
11	iopl(3); asm(cli); while(1)	12	9	204
12	forkbomb	7	6	172
13	write(0,"Hello core !",12)	36	18	456
14	eject cd-rom(follows /dev/cdrom symlink)+exit()	40	25	531
15	anti-debug trick (INT 3h trap)+execve /bin/sh	39	9	204
16	set system time to 0 and exit	12	12	276
17	kill all processes	11	9	236
18	re-use of /bin/sh string in .rodata shellcode	16	7	212
19	File unlinker	18	12	276
20	dup2(0,0); dup2(0,1) dup2(0,2)	15	23	397
21	Ho' Detector	56	25	573
22	Radically Self Modifying Code	25	18	280

在我们构造的 shellcode 中，有的涉及了复杂的 shellcode 的设计. 包括使用多个系统调用、条件跳转和 loop 指令等. 如 setreuid(getuid(),getuid()), execve("/bin/sh",0,0)，需要涉及 3 个系统调用，dup2(0,0)、dup2(0,1)、dup2(0,2)，Ho' Detector 使用条件跳转指令，Radically Self Modifying Code 使用 loop 指令等. 由表 2 可以看出，随着 shellcode 复杂性的增加，构造 BIOP shellcode 的复杂度也在增大. shellcode 的长度越大，构造的 BIOP shellcode 的 gadget 数目和大小也越大，一般 BIOP shellcode 都在 100 比特之上.

4.3 BIOP 攻击效力

我们同时测试 BIOP 对现有的 ROP/JOP 防御技术的效力. 如表 3 所示，我们选取 3 种典型的 ROP 防御方法进行测试，分别是：(1) DROP[5]，检测运行时以 ret 结尾的短指令序列是否频繁出现；(2) ROPdefender[6]，检测运行时 call 指令与 ret 指令是否一一对应；(3) Return-less Compiler[7]，使用消除 ret 指令的编译器. 为了方便比较，我们也测试了传统的 ROP 攻击、Tyler 提出的 JOP 攻击对这 3 种防御技术的效力. 从表 3 可以看出，传统的 ROP 技术使用以 ret 结尾的 gadget 片段，这种明显的特征可以被这 3 种技术检测到. Tyler 的 JOP 技术[9]消除了 ret 指令带来的负面影响，可以有效绕过 DROP 和 Return-less Compiler 的防御方法，但是它所使用的 call 指令并没有相应的 ret 指令，基于这种特征的 ROPdefender 技术仍然可以检测到这类攻击. BIOP 技术避免使用连续的 ret 指令，同时使 call 指令与 ret 指令一一对应，从而可以有效的绕开所有这 3 种防御技术.

表 3　各类攻击效力的比较

	DROP	ROPdefender	Return-less Compiler
ROP shellcode	有效	有效	有效
Tyler's JOP	失效	有效	失效
BIOP	失效	失效	失效

5　相关工作

ROP 技术是一种全新的代码复用攻击方法，它被认为是未来主流攻击方法之一. ROP 攻击技术和防御技术近几年发展迅速，我们在此章介绍 ROP 的发展背景和相关工作，使读者对 ROP 的发展有更清楚的了解.

① milw0rm：http://www.milw0rm.com/shellcode/linux/x86

5.1 Return-to-libc 攻击

为了解决向 W⊕X 或 DEP 保护的内存页注入恶意代码失效的问题，攻击者尝试复用系统已经存在的函数级代码，return-to-libc[1] 是一种典型的代码复用攻击，它通过劫持控制流，跳转到 C 语言函数库 libc，复用 libc 中已有的函数，可以有效的绕开 W⊕X 和 DEP 技术. 然而，renturn-to-libc 攻击存在以下缺陷：(1) 只能顺序调用函数，不能实现图灵完备的行为，如分支操作、循环操作等；(2) 依赖于系统中存在的库函数，移除或修改 libc 中的特定函数将严重限制攻击能力.

5.2 Return-Oriented Programming

由于 return-to-libc 攻击具有一定局限性，Shacham[2] 于 2007 年提出一种新型的代码复用技术——返回导向编程攻击，将复用的代码粒度从 return-to-libc 的函数级别缩小到指令序列，每个指令序列都是一个以 ret 指令结尾的指令序列，通过在栈内设置下一个要跳转的指令序列地址，攻击者可以将多个指令序列串联起来以实现恶意语义. Schacham 认为 libc 中的指令序列集合是图灵完备的，可以构造任意功能的攻击. 然而，ROP 使用以 ret 指令结尾的指令序列构造攻击，与正常程序相比，ROP 具有以下特征：(1) ret 指令并没有相应的 call 指令与其对应；(2) 频繁使用以 ret 指令结尾的短指令序列，这在正常程序流中非常少见.

5.3 Return-Oriented Programming 防御技术

针对 ROP 的特点，防御技术主要可以分为以下 3 类：(1) ROP 中 ret 指令并没有相应的 call 指令与其匹配，Francillon 等人[3] 使用一个嵌入式的微处理器保护 call/ret 栈不被任意数据改写，ROPdefender[5] 使用影子内存来检测对返回值地址的修改；(2) ROP 指令序列一般只有两三个指令长度，每个指令序列都以 ret 指令结尾，Chen 等人[6] 和 Davi 等人[4] 针对 ROP 的这种特性进行检测，当系统频繁的执行含 ret 指令的短指令序列时，则认为系统受到 ROP 攻击；(3) 消除库中 ret 指令可以有效消除 ROP 攻击的构造基础. 如 Li 等人[7] 设计了一种编译器可以在程序编译时候消除 ret 指令.

5.4 ROP 技术演化

为了克服 ROP 的缺陷，攻击者试图改进 ROP 攻击. Checkoway 等人[8,10] 提出使用含 pop-jmp 指令序列替换原来以 ret 指令结尾的指令序列. 但这种方法使用类似 ret 功能的 pop-jmp 指令（如 pop ebx，…，jmp [ebx]）. 虽然这种改进可以使现有 ROP 防御技术失效，但是没有从本质上解决 ROP 技术的缺陷. 因为虽然避免使用 ret 指令，但是引入的 pop-jmp 指令同样具有明显的特征，通过对栈的保护或对 ROP 动态检测技术进行修改可以很容易检测此类攻击.

Bletsch 等人[9] 提出 Jump-Oriented Programming(JOP) 消除对 ret 指令的依赖，但这种技术存在缺陷. 首先他们使用含 call 指令的指令序列，但并没有相应 ret 指令与其匹配，使用栈保护技术可以防御这类攻击[3,8]；其次，JOP 的构造具有寄存器依赖性，在某个指令序列中使用寄存器会影响到其他指令序列对该寄存器的使用，Bletsch 等人并没有对这个问题进行讨论，而且他们提出的调度指令序列加剧了这种依赖性，使 JOP 功能受到限制. 事实上，我们发现，在 Bletsch 等人使用了 add ebp，edi；jmp [ebp-0x39] 这个调度指令序列以后，将无法在以后的 gadget 中更改 ebp 和 edi 这两个寄存器值，导致 JOP 的构造能力受到严重限制. 我们在第 4 节中讨论了这个问题，同时我们也提出如何消除寄存器依赖性，可以构造能力更强大的攻击. 再者，调度指令序列的使用使程序流与正常程序流相比，具有明显的特征，通过动态检测很容易发现此类攻击. 最后，调度指令序列是一种理想的指令序列，现实的 libc 库中很难找到，从而使 JOP 的攻击取决于现有的库中是否可以找到有效的调度指令序列.

如上所述，已有的 ROP 及 ROP 改进技术都存在明显特征，易被检测，不利于实际的应用. 本文提出的 BIOP 技术消除了原有的 ROP 及 ROP 改进技术的特征，隐蔽性更强. 同时我们考虑了寄存器对构造攻击能力的影响，并分析解决了构造时寄存器副作用问题，为我们自动化构造攻击奠定理论基础. 我们设计了一个自动化构造 BIOP 的工具，使用我们的工具可以大规模有效的构造 BIOP shellcode.

6 结束语

本文提出了一类新型 ROP 攻击技术 BIOP，它通过复用现有库和可执行文件中含 jmp 指令和 call 指令的指令序列，从而有效绕开现有的 ROP 防御技术. 它不仅消除了 ROP 中 ret 和 call 非一一对应的缺陷，且不依赖于对栈的控制. 我们提出了一个指令序列快速搜索算法，并且分析了利用这些指令序列自动构造攻击时的难点和挑战，针对这些难点和挑战我们提出解决方法. 同时我们设计了一个自动化构造工具. 通过自动化工具，我们能方便构造各种语义的 shellcode. 实验结果表明，BIOP 具有与 ROP

相同的能力，并可以有效避开当前的 ROP 检测技术. 我们相信基于 BIOP 的攻击将会成为系统安全的新威胁.

参 考 文 献

[1] Desinger S. "return-to-libc" attack. Bugtraq, Aug, 1997

[2] Shacham H. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)// Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS). New York, USA, 2007: 552-561

[3] Francillon A, Perito D, Castelluccia C. Defending embedded systems against control flow attacks//Proceedings of the 1st ACM Workshop on Secure Execution of Untrusted Code (SecuCode'09). New York, USA, 2009: 19-26

[4] Davi L, Sadeghi A R, Winandy M. ROPdefender: A detection tool to defend against return-oriented programming attacks. Technical Report TR-2010-001, 2010

[5] Chen P, Xiao H, Shen X, et al. DROP: Detecting return-oriented programming malicious code//Proceedings of the International Conference on Informaion Systems Security. Kolkata, India, 2009: 163-177

[6] Davi L, Sadeghi A R, Winandy M. Dynamic integrity measurement and attestation: Towards defense against return-oriented programming attacks//Proceedings of the 2009 ACM Workshop on Scalable Trusted Computing. Chicago, USA, 2009: 49-54

[7] Li J, Wang Z, Jiang X, Grace M, Bahram S. Defeating return-oriented rootkits with "return-less" kernels//Proceedings of the 5th European Conference on Computer Systems. New York, USA, 2010: 195-208

[8] Checkoway S, Davi L, Dmitrienko A, et al. Return-oritend programming without returns//Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS). Chicago, USA, 2010: 559-572

[9] Bletsch T, Jiang Xuxian, Freeh V W, et al. Jump-oriented programming: A new class of code-reuse attack//Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security. Chicago, USA, 2011: 30-40

[10] Checkoway S, Shacham H. Escape from return-oriented programming: Return-oriented programming without returns (on the x86). University of California, San Diego: Technical Report, 2010

XING Xiao, born in 1988, M. S., engineer. His current research interest is system security.

CHEN Ping, born in 1985, Ph. D. His current research interest is system security.

DING Wen-Biao, born in 1988, M. S. His current research interest is system security.

MAO Bing, born in 1967, Ph. D., professor, Ph. D. supervisor. His main research interests include system security and software security.

XIE Li, born in 1942, professor, Ph. D. supervisor. His main research interests include distributed computing and parallel processing.

Background

Return Oriented Programming is a technique which leverages the instruction gadgets in existing libraries or executables to construct Turing Complete Programs. However, existing ROP defense techniques can detect the attack based on the fact that ROP attack is usually constructed with the gadgets ending with return instruction. In this paper, we propose BIOP, an improved ROP technique to construct the ROP shellcode without return, which can effectively bypass most of exitsting ROP defenses. Meanwhile, we implement a tool to automatilcally construct the real-world BIOP shellcode. Our work is supported in part by grants from the National Natural Science Foundation of China (61073027, 60773171, 90818022, 61272078 and 61321491), and the National High Technology Research and Development Program (863 Program) of China (2007AA01Z448, 2011AA1A202).