软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2017,28(10):2583−2598 [doi: 10.13328/j.cnki.jos.005317]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

ROP 图灵完备的普遍可实现性*

袁平海[1,2], 曾庆凯[1,2]

[1](南京大学 计算机科学与技术系,江苏 南京 210023)

[2](计算机软件新技术国家重点实验室(南京大学),江苏 南京 210023)

通讯作者: 曾庆凯, E-mail: zqk@nju.edu.cn

摘 要: 返回导向编程(return-oriented programming,简称 ROP)被广泛用于软件漏洞利用攻击中,用来构造攻击代码.通过更新 ROP 构造技术,证实了图灵完备的纯 ROP 攻击代码在软件模块中是普遍可实现的.ROP 构造功能代码的难点是实现条件转移逻辑.通过深入分析条件转移机器指令的执行上下文发现,对这些指令的传统认知存在一定的局限性.事实上,在已有代码中存在少量的条件转移指令,它们的两个分支的开始部分都是可复用的代码片段(称为 gadgets),而且这两个 gadgets 会从不同的内存单元中取得下一个 gadget 的地址,因此,以这些条件转移指令开始的代码片段可以帮助 ROP 实现条件转移逻辑.把这种代码片段称为 if-gadget.在 Linux 和 Windows 系统上的实验结果表明, if-gadget 普遍存在,即使在代码量很小的日常可执行程序中也存在.在 Binutils 程序集上的实验结果表明,引入 if-gadget 后,构造图灵完备的 ROP 代码要比用传统方法容易得多.在 Ubuntu 这样的主流操作系统上,由于可执行程序上默认没有实施防御 ROP 攻击的保护机制,因此,攻击者可以在这些软件模块中构造纯 ROP 攻击代码来发动攻击.由此可见,ROP 对系统安全的威胁比原来认为的严重得多.

关键词: 软件漏洞利用;返回导向编程;图灵完备计算;条件转移逻辑

中图法分类号: TP311

中文引用格式: 袁平海,曾庆凯.ROP 图灵完备的普遍可实现性.软件学报,2017,28(10):2583−2598. http://www.jos.org.cn/1000-9825/5317.htm

英文引用格式: Yuan PH, Zeng QK. Universal availability of ROP-based Turing-complete computation. Ruan Jian Xue Bao/ Journal of Software, 2017,28(10):2583−2598 (in Chinese). http://www.jos.org.cn/1000-9825/5317.htm

Universal Availability of ROP-Based Turing-Complete Computation

YUAN Ping-Hai[1,2], ZENG Qing-Kai[1,2]

[1](Department of Computer Science and Technology, Nanjing University, Nanjing 210023, China)

[2](State Key Laboratory for Novel Software Technology (Nanjing University), Nanjing 210023, China)

Abstract: Return-Oriented programming (ROP) is widely applied in modern software vulnerability exploitations. This work demonstrates that Turing-complete ROP code is universally available in everyday software. A big challenge for applying ROP is to construct the functionality of conditional jumps. Because conditional branch instructions are abandoned as they are deemed no use for achieving this functionality, existing works resort to some awkward methods which suffer from high risk of failure. By analyzing the execution context of conditional branch instructions, this study finds that the traditional viewpoint on these instructions only partially reveals the truth. In fact, there are some conditional branch instructions in which two branches each starts a reusable gadget, and these two gadgets fetch the next gadget from different memory cells. Hence, the code snippets beginning at these conditional instructions can implement the conditional jumps for ROP code. Such a code snippet is named if-gadget. Experimental results show that if-gadgets are

* 基金项目: 国家自然科学基金(61772266, 61572248, 61431008, 61321491); 国家科技支撑计划(2012BAK26B01)

Foundation item: National Natural Science Foundation of China (61772266, 61572248, 61431008, 61321491); National Key Technology R&D Program of China (2012BAK26B01)

收稿时间: 2017-01-20; 修改时间: 2017-03-24, 2017-06-07; 采用时间: 2017-06-14

widely available in executables of Linux and Windows platforms. Evaluations on programs of Binutils demonstrate that, Turing complete ROP code can be achieved with the help of if-gadgets while existing techniques even fail to gather Turing complete gadgets. On platforms such as Ubuntu, because the executables running on them do not support ASLR by default, attackers can construct Turing-complete ROP code on these executables and then mount an attack. Therefore, ROP-based attacks pose a great threat to modern platforms, which is far more serious than originally thought.

Key words: software vulnerability exploitation; return-oriented programming; Turing-complete computation; conditional jump

软件漏洞依然普遍存在,甚至有些被广泛使用的动态库也一直受到软件漏洞的困扰.以 glibc 为例,这两年的漏洞已经超过 15 个,其中,CVE-2015-8779 是栈上的缓冲区溢出漏洞,可以让攻击者用长的目录名来触发漏洞并执行任意代码[1].在软件漏洞利用攻击中,攻击者成功获取了程序的控制权后,可能会在受害系统上创建后门或安装恶意软件,以便进一步窃取隐私信息或实施金融敲诈.因此,保护日常软件免于漏洞利用攻击是系统安全的一个重要课题.

当前,数据执行保护(data execution prevention,简称 DEP)[2]和地址空间布局随机化(address space layout randomization,简称 ASLR)[3]等安全机制已被主流系统广泛配置.这些安全机制可以有效地缓解漏洞利用攻击.DEP 通过设置进程页表,可去除数据段所在页的可执行属性.该机制可以有效地阻止传统的代码注入攻击;该攻击以输入数据的方式在数据段上注入 shellcode 并执行之.为了绕过 DEP,攻击者发明了代码复用攻击技术;这类攻击通过复用受害进程地址空间中已有的代码来构造攻击代码.返回导向的编程(return-oriented programming,简称 ROP)是一种典型的代码复用技术[4],它通过串接以 RET 指令结束的代码片段(称为 gadget)来构造恶意代码.ASLR 是缓解 ROP 攻击的有效策略.ASLR 将软件模块加载到随机选择的地址空间,使得 gadget 的地址不可预测,因此可以显著提高 ROP 的攻击门槛.在 Linux 系统上,几乎所有的动态库都支持 ASLR.但是,由于性能问题[5]和兼容性问题,很多 Linux 发行版本中的可执行程序默认地并不支持 ASLR(当可执行程序被编译为“位置无关的可执行程序”后,能够支持 ASLR.但是,在很多 Linux 发行版本中,这并不是默认配置.例如,在 Ubuntu 系统中,大部分可执行程序不支持 ASLR.Windows 系统下的情况似乎更为严重,由于系统设计机制的限制,很多重要动态库的加载地址只能在系统启动时做一次随机化.本文以 Linux 为例讨论安全问题,所提及的技术在 Windows 系统上也是通用的).

由于用 shellcode 构造恶意代码要比 ROP 技术容易得多,因此攻击者现在普遍采用 ROPcode+shellcode 的模式构造攻击代码.具体地,ROP 代码用来绕过 DEP 保护机制,shellcode 则用来实现更多的恶意企图.典型地,ROP 代码调用 mprotect 函数为 shellcode 所在的数据段增加可执行属性,之后再执行 shellcode.防御者已经找到了有效的方法来防范这种攻击.由于日常可执行程序通常不会使用动态生成的代码,即没有为内存段动态增加可执行属性的需求,因此可以约束对 mprotect 系统函数的调用以遏制这种攻击模式.事实上,Linux 系统正在应用该策略来限制对内存页属性的动态修改[6].该机制启动后,将阻止进程进行以下操作:(1) 改变未被初始化为可执行的内存页的可执行状态;(2) 使只读的可执行页可再次写入;(3) 在分配的动态内存中创建可执行页;(4) 让重定位后只读的数据页可再次写入.该机制启动后,一种可能的攻击方式是复用不支持 ASLR 的可执行程序中的 gadgets 来构造纯 ROP 攻击代码.考虑到用 ROP 构造复杂代码的困难性,以及可执行程序的代码量通常较少的事实,这种攻击普遍认为很难实现.这也是默认情况下 Ubuntu 等操作系统中的可执行程序不支持 ASLR 的一个重要理由.

用 ROP 技术构造功能代码要比用程序语言编写功能代码难得多.ROP 代码的执行流是通过 RET 指令(或等价的指令序列)串接起来的.该指令从栈上取得控制流转移的目标地址,即下一个 gadget 的地址.ROP 代码需要控制栈指针寄存器(stack pointer register,简称 SP)的值,让 RET 指令从指定的栈上内存单元(在攻击者劫持程序控制流前,有效载荷即 payload 已经存放到栈上)取出目标地址.ROP 技术构造功能代码的最大挑战是实现条件转移逻辑功能.传统观点认为,条件转移机器指令只能修改指令指针寄存器(instruction pointer register,简称 IP)的值,而不能有条件地改变 SP 的值,因此它们不能帮助 ROP 代码实现条件转移逻辑[4,7,8].在 x86 架构下,攻击者用已有方法实现简单的条件转移逻辑也需要串接 11 个不同的 gadgets[4].在构造过程中,缺少某些类型的

gadgets 或者不能完全消除 gadgets 之间的副作用,都会导致构造失败.

本工作通过更新 ROP 实现条件转移逻辑的方案,证实了图灵完备的纯 ROP 攻击代码在软件模块中是普遍可实现的,即上文提及的攻击也是普遍可行的.通过深入分析条件转移机器指令的执行上下文我们发现,对这些指令的传统认知存在局限性.尽管大部分条件判断指令不可用于帮助 ROP 代码实现条件转移逻辑,但仍然存在着少部分条件判断指令:它们的每个分支的开始部分都是一个可复用的经典 gadget,这两个 gadgets 会从不同的内存单元(内存单元的内容可能已经被加载到寄存器中)中取得下一个 gadget 的地址.因此,这样的代码片段可以帮助 ROP 实现条件转移逻辑.我们把这种以条件判断指令开始的代码片段称为 if-gadget.引入 if-gadget 后实现条件转移逻辑要比用传统方法简单得多.典型地,只需要两个代码片段就能实现该功能.本文阐述 if-gadget 实现条件转移逻辑的原理,讨论引入 if-gadget 后,gadget 的查找算法、payload 布局以及 if-gadget 的调度.

在 Linux 和 Windows 上的实验结果表明,if-gadget 在软件模块中普遍存在.甚至在一个简单的 HelloWorld 程序中,都存在若干 if-gadgets.在 Binutils 程序集上的实验结果表明,引入 if-gadget 后,在每个可执行程序中都能找到图灵完备的 gadgets,且可实现图灵完备的功能.与之相比,用传统方法几乎无法在这些程序中找到图灵完备的 gadget 集合.由此可见,图灵完备的 ROP 攻击代码在软件模块中是普遍可实现的.另外,在 Ubuntu 等主流系统上,攻击者可以复用不支持 ASLR 的可执行程序中的 gadgets 来构造出纯 ROP 攻击代码.因此,ROP 攻击对现实系统的安全威胁要比原来认为的严重得多.

1 背景知识与动机

1.1 ROP代码复用技术

ROP 通过控制栈指针寄存器的值来控制执行流,payload 由 gadgets 的地址以及喂给 gadgets 的参数组成.图 1 给出了一个简单的 ROP 攻击实例,它调用 system 库函数来执行“uname-a”命令,之后调用 exit 库函数结束进程.假设攻击者已经利用缓冲区溢出漏洞把 payload 填充到栈上,且图中偏移为 0 的内存单元覆盖了原来的函数返回地址.当原程序中的函数执行 ret 指令返回时,将执行 ROP 代码中的第 1 个 gadget,即 payload 的第 1 个数据 0x40067f 指向的代码片段“pop %rdi;ret”.该 gadget 执行完后,寄存器 rdi 的值将指向字符串“uname-a”,同时其末尾的 ret 指令取出下一个 gadget 的地址并执行.此时,代码片段“jmpq *0x200ae2(%rip)”被执行.这个 gadget 将调用库函数 system.该函数的参数是要执行的命令所对应的字符串,在 x86-64 架构上,这个参数存放在寄存器 rdi 中,因此,system 将执行命令“uname-a”.执行完该命令后,将执行下一个 gadget,即“pop %rdi;ret”,之后,ROP 代码将调用库函数 exit(0x21)结束进程.

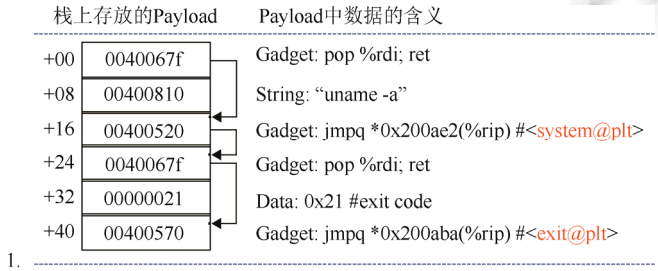

Fig.1 A ROP instance. The stack grows downwards

图 1 ROP 攻击实例.栈向地址空间生长

ROP 实现条件跳转的传统方法很复杂.Shacham 等人[4]特意为此设计了一个方案,其核心内容是把条件标志的值传递到栈指针寄存器 SP 上.他们通过 3 个步骤来实现该功能:(1) 进行某些操作来设置或清除感兴趣的标志;(2) 把寄存器 EFLAGS 上的标志传递到一个通用寄存器上,以便将该标志隔离开来;(3) 使用该标志来产生一个期望的增量(当标志清除时,增量为 0),再将该增量加到 SP 上,这样就可以有条件地修改 SP 的值.为了证

实该方案,Shacham 等人在 glibc 中找到了 11 个不同的 gadgets,并将它们串接起来以便实现一个简单的条件转移功能.其他工作也采用类似方案实现条件转移功能[7,8].需要强调的是,这些工作都是在大型软件模块上实施有效性评估的,因为这样的模块可以提供足够丰富的 gadgets 以便于构造功能代码.

串接数十个强相互依赖的 gadgets 是一个非常有挑战性的工作.通常,每个 gadget 除了完成预想的计算外,还会额外地读写内存或更新寄存器内容.这些额外的操作,我们称其为副作用.有些副作用并不能简单地克服.如果一个 gadget 的副作用会冲刷前面的 gadgets 设置的寄存器值,那么这些 gadgets 就不能一起工作.随着需要串接的 gadgets 数量的增多,找到可以一起工作的组合将变得非常困难.因此,有的 ROP 编译器尝试利用约束求解器来查找兼容的组合[9].即使如此,目前也没有实用的编译器可以为条件判断逻辑自动生成 ROP 代码.

1.2 动 机

默认配置下,Ubuntu 等主流系统中的可执行程序并不支持 ASLR.攻击者可以复用这些应用程序中的代码片段构造纯 ROP 攻击代码.由于很多实用程序和部分服务器程序都属于超级用户,而且有些程序拥有 SUID(set user ID)属性.攻击者在这些进程上实现攻击后,可以对系统安全造成严重破坏.本工作的主要动机是评估这种攻击的可实现性.该工作将对评价主流系统的安全状态有重要意义(实用程序是系统软件的一部分,用于分析、配置、优化和维护一个计算机系统.Linux 系统的软件源通常也提供一些常用的服务器程序.例如,Ubuntu 14.04 的软件源就包含有 xinetd 和 nginx 这两个软件.Xinetd 是新一代的网络守护进程服务程序,nginx 是一个高性能 HTTP 服务程序,但两者都不支持 ASLR).

实施该评估的有效方法是检测图灵完备 ROP 功能代码在日常软件,特别是代码量较小的日常用可执行程序上的可构造性.值得注意的是,已有好几项工作研究了 ROP 的图灵完备性[4,7,8,10].但它们的关注点是构造图灵完备的 gadget 集合,这只是实现图灵完备功能的前提条件.有些图灵基本功能需要串接多个 gadgets 才能实现.不能成功消除 gadgets 之间的副作用,就不能实现这些功能.但是,即使如此,这些工作也只能在代码量大的可执行文件和动态库中才能找到图灵完备的 gadgets.因此,这些工作不能在代码量相对较小的日常软件上给出有效结论.

1.3 图灵完备功能

一种计算机语言能够实现任意计算的前提条件是:它提供了一套图灵完备的指令集.最简单的图灵完备指令集只有一个指令[11],该指令的定义(位于下面所示左半部分)及伪代码(位于下面所示右半部分)如下.

```
subleq a,b,c    ;Mem[b]=Mem[b]−Mem[a]
                ;if (Mem[b]≤0) goto c
```

该指令 subleq 对 ***b*** 指向的内存单元做减法运算,减去 ***a*** 指向的内存单元的值.如果运算结果小于等于 0,则跳转到 ***c*** 指向的代码处.但是,该指令在已知的硬件架构上几乎都不存在,因此我们要把它转化成一个等价的、实际存在的指令集合.把该指令的原子操作分开,我们可以得到 3 个指令.

1. 在内存上的减法运算: Mem[b]=Mem[b]−Mem[a]
2. 与 0 做比较操作: Mem[b]≤0
3. 条件跳转: if (TURE) goto c

第 1 个指令使用两个内存操作数进行运算,但是现有的通用架构,如 x86,并不支持两个内存操作数.为此,我们需要增加实施内存读写操作的指令.另外,为了支持输入/输出操作,我们还需要增加一条指令实现系统调用.这样,我们还有如下 3 个指令.

4. 加载内存到寄存器: load Reg,Mem[addr]
5. 存储寄存器到内存: store Reg,Mem[addr]
6. 系统调用指令: Syscall

这 6 条指令在所有的架构上几乎都存在,因此可以作为我们要搜索的最小 gadget 集合.这 6 条指令所对应的功能,就是本工作要用 ROP 技术实现的图灵基本功能.

2 条件判断指令执行上下文与 if-gadget 工作原理

2.1 条件判断机器指令执行上下文

程序语言,如 C/C++,在实现条件判断逻辑时,先测试条件的真假,再依据测试结果选择不同的执行分支.对应的机器语言代码则先执行条件测试指令来设置 EFLAGS 中的标志位,紧接着执行条件转移机器指令并依据相应标志的状态来修改指令指针的值.x86 架构提供了数十个条件转移机器指令,不同的指令测试不同的标志位或标志位的组合.如 JZ 测试 ZF 标志是否为 0;JA 测试 ZF 和 CF 是否同时为 0.当测试条件成立时,执行流跳转到真分支;否则,转向(fall-though)到假分支.

表 1 给出了一个段用 C 语言描述的代码以及与其对应的汇编指令序列.机器先执行 TEST 指令来测试 EAX 寄存器是否为 0,然后执行 JLE 指令来判断条件标志位的状态.如果条件成立(即为真),则跳转到 0x80483f9 处对 EAX 赋 1 后返回;否则,自动执行后续指令(fall-through),做乘法运算后返回.

Table 1 The execution context of a conditional branch instruction

表 1 条件判断指令的执行上下文

C 语言源代码	汇编代码
int demo1(int arg)	080483ed ⟨demo1⟩
{	80483ed: mov 0x4(%esp),%eax
int ret=1;	80483f1: test %eax,%eax
if (arg>0)	80483f3: jle 80483f9
ret=arg*arg;	80483f5: imul %eax,%eax
return ret;	80483f8: ret
}	80483f9: mov $0x1,%eax
	80483fe: ret

在该实例中,条件判断指令 JLE 的两个分支执行完后都产生相同的栈偏移,因此,两个 RET 指令都从相同的内存单元中取得控制流的转移目标地址.为此,以 JLE 开始的代码片段不能帮助 ROP 实现条件转移逻辑.在现实中,更常见的情况是至少有一个分支以直接控制流指令或非法指令结束(ROP 攻击会从合法指令的中间开始寻找 gadgets,因此时常会遇到以非法指令结束的情况).在 glibc 上的统计表明,在所有的 91 536 个条件控制流指令中(计数包含从合法指令中间开始的截断指令),有 90 989 个指令的至少一个分支是以直接控制流指令或非法指令结束的,即有超过 99.4%的条件判断指令不能帮助 ROP 实现条件转移逻辑.在剩下的条件转移指令中,大部分也不能帮助 ROP 实现条件转移逻辑.例如,有 192 个条件判断指令的两个分支都是经典 gadgets,但其中有 165 个属于表 1 所示的情况.但我们发现,有少量条件转移指令的两个分支可以从不同的内存单元取得下一个 gadget 的地址(该 glibc 来源于 Ubuntu 14.04 32bit 系统,版本号 2.19,运行在 x86 硬件架构上.代码量 0x1a7a60 字节).

表 2 给出了一个这样的实例.左栏给出了一段用 C 语言描述的代码.代码功能很简单,它只是简单地用函数指针实现函数调用:当函数指针不空时,调用子函数.

Table 2 A code snippet with an if-gadget

表 2 包含 if-gadget 的程序片段

C 语言源代码	汇编代码
typedef void (*ft)(void);	080483ff ⟨demo2⟩:
void demo2(ft fun)	80483ff: sub $0xc,%esp
{	8048402: mov 0x10(%esp),%eax
if (fun) fun();	8048406: test %eax,%eax
}	8048408: je 804840c
	804840a: call *%eax
	804840c: add $0xc,%esp
	804840f: ret

右栏给出了对应的汇编代码.在汇编代码中,条件指令 JE(地址 0x8048408)的真分支是指令序列“ADD $0XC,%ESP;RET”;假分支则是指令序列“CALL *%EAX;ADD $0XC,%ESP;RET”.从 ROP 攻击的视角来看,真分

支是一个以 RET 指令结束的经典 gadget,而假分支则是以“CALL *%EAX”结束的经典 gadget.因为这两个 gadgets 可以从不同的栈上内存单元获取下一个 gadget 的地址,因此,以该条件跳转指令开始的代码片段能够帮助 ROP 实现条件转移逻辑.下一节以表 2 中的代码片段为例,阐述 if-gadget 的工作原理并给出 if-gadget 的定义.

2.2 If-Gadget原理与定义

攻击者可以将表 2 中以 JE 开始的代码片段视为一个复合代码片段.在构造 payload 时,攻击者可以使用常见的经典 gadgets(如“POP *%EAX;RET”)把 EAX 的值设置成某一个 gadget 的地址,这样就能确保条件为真时与条件为假时,攻击代码能从不同的内存单元取得下一个 gadget 的地址,即分别从 0x804840f 处的 RET 指令访问的地址和“POP *%EAX”访问的地址中取出下一个 gadget 的地址.同时,攻击者可以根据实际情况,通过加法(ADD)/减法(SUB)/比较(CMP)/测试(TEST)等指令设置 EFLAGS 寄存器中的 ZF 标志位,之后再将控制流转移到如上所述的两个 gadgets,这样就能实现条件转移逻辑.这种构造方法实现了 ROP 代码的条件转移逻辑,此即 if-gadget 的工作原理.我们将这种可以实现条件转移逻辑的复合代码片段称为 if-gadget.

具体地,本工作定义了 3 种类型的 if-gadget.这些类型都是我们的 gadget 查找系统能够找到的.

If-Gadget 类型 I.

以条件转移机器指令 J_0 开始;

J_0 的两个分支都是经典 gadgets(定义见表 3),分别为 G_1 和 G_2;

G_1 和 G_2 从不同的内存单元取得下一个 gadget 的地址.

表 2 中以 0x8048408 处指令 JE 开始的代码片段就是这种类型的 if-gadget.其真分支是指令序列“ADD $0XC,%ESP;RET”,这是一个 AddConstantGadget 的经典 gadget(是 ArithConstG 的一个子类);假分支是“CALL *%EAX”,这是一个 RegFuncallG 类型的经典 gadget.由上述分析可知,这两个 gadget 会从不同的内存单元取得下一个 gadget 的地址,可以帮助 ROP 实现条件转移逻辑.定义中的约束非常重要.表 1 中以指令 JLE 开始的代码片段满足前面两个要求,但不满足后面的约束条件,即它的两个分支是从同一个内存单元取得下一个 gadget 的地址,因此它不是 if-gadget.另外,由该定义可见,if-gadget 与传统的经典 gadget 不同,它有两个出口.值得注意的是,if-gadget 的真分支(发生跳转的分支)的目标地址可能在条件转移机器指令之前,因此,if-gadget 并不一定是一个连续的代码块.

If-Gadget 类型 II.

以条件转移机器指令 J_0 开始;

J_0 的一个分支为经典 gadget G_0,另一个分支是以条件跳转机器指令 J_1 结束的基本块,该基本块无内存读写操作;

J_1 的执行条件可以被控制,且 J_1 的假(pass-through)分支为经典 gadget G_1;

G_0 和 G_1 从不同的内存单元取得下一个 gadget 的地址.

事实上,类型 II 是类型 I 的扩展.在利用这种 if-gadget 时,攻击者需要通过控制条件标志位使得 J_1 的假分支被执行,这样,J_1 将等效为一条无操作指令(no-operation),以 J_1 开始的代码块的行为语义与 G_1 等价.根据 if-gadget 类型 I 的定义,这种类型的代码片段可以帮助 ROP 实现条件转移逻辑.另外,在该定义中,我们要求以 J_1 结束的基本块无内存读写操作,该约束的目的是减少到达 G_1 的代码路径可能产生的副作用.攻击者在利用这种 if-gadget 时需要控制两个条件判断指令的执行.尽管有些复杂,但相对而言,这仍然是一个简单的约束求解问题.

If-Gadget 类型 III.

以条件转移机器指令 J_0 开始;

J_0 的两个分支分别是以条件跳转机器指令 J_1 和 J_2 结束的基本块;两个基本块都无内存读写操作;

J_1 和 J_2 的执行条件可控,J_1 的假分支为经典 gadget G_1,J_2 的假分支为经典 gadget G_2;

G_1 和 G_2 从不同的内存单元取得下一个 gadget 的地址.

该类型的 if-gadget 是前两种类型的扩展.施加的各种约束的目的就是要确保 J_0 的两个分支的行为语义分别与 G_1 和 G_2 等价.由于这类 if-gadget 需要满足的约束更多,在实际代码中出现得很少.

在类型 II 和类型 III 的 if-gadget 定义中,我们要求 J_1(和 J_2)的假(false)分支为经典 gadget.当我们取消这个要求时,可能可以得到更多的 if-gadgets,但是这些新增 if-gadgets 的质量可能会很差,即不容易被操作.例如,如果我们反过来,在定义中仅考虑使用 J_1 的真(true)分支,即让真分支为经典 gadget,则有可能让 J_1 依赖 J_0,构成循环依赖.另外,如果允许 J_0 到 J_1 的路径上存在内存访问操作,则会让检查约束的过程变得复杂.

需要再次强调的是,各个定义中的约束是很重要的,满足约束的代码块才有实际意义.通过对 glibc 的观察,我们得到如下结果:(1) 有 192 个条件判断指令满足定义 I 的前两个要求,即它们的两个分支都是经典 gadgets,但满足约束的只有 27 个;(2) 有 242 个条件判断指令满足定义 II 的前两个要求,但只有 19 个指令满足后面的约束;(3) 有 94 个条件判断指令满足定义 III 的前两个要求,但仅有 4 个指令满足后面的约束.

3 图灵基本功能的构造

本节以 x86 架构为例,讨论各种图灵基本功能的实现方法.重点讨论引入 if-gadget 后,gadget 查找和调度以及 payload 的布局.这些方法与技术在其他架构下也是适用的.

3.1 图灵基本功能的构造方案

现代计算机架构都提供丰富的指令集,远远超过了实现图灵完备功能的要求.这样的指令集合有助于程序用简洁的机器语言实现给定功能.ROP 代码的构造也受益于这样的指令集.图灵基本功能中定义的“减法运算”有多种实现.在 x86 架构上,SUB 指令可以直接实现减法操作.如果实际需要的是一个加法操作,那么可以直接用 ADD 指令实现,而不需要用类似“SUB(x,SUB(0,y))”的方式来实现.对其他的算术逻辑运算更是如此.“比较操作”也有多种实现方法.在 x86 架构上,CMP、TEST、ADD 和 SUB 这些指令都可以修改条件标志位,因此,都可以实现与 0 进行比较的操作.

条件转移功能可以借助 if-gadget 来实现.典型地,一个 if-gadget 与一个常见的经典 gadget 结合能够实现条件转移逻辑.

加载内存内容到寄存器或者存储寄存器内容到内存,都可以用 MOV 指令来实现.因为 x86 的指令支持一个内存操作数,因此也可以使用带内存操作数的算术逻辑运算指令把内存内容加载到寄存器中,或者相反地,存储寄存器的内容到内存中.

用户态代码可以通过特殊指令(INT,SYSENTER 和 SYSCALL)实现系统调用.另外,在系统库(如 Linux 系统的 libc)中存在着丰富的封装函数,它们也可以实现系统调用功能.因此,ROP 代码可以调用库函数来实现系统调用.即使在部署了地址空间布局随机化安全机制的系统上,ROP 代码仍然可以采用这种通用的方法实现系统调用[10].我们倾向于用库函数实现系统调用,因为任何 gadget 都可将控制流传递给这些函数,实现系统调用.

3.2 Gadget类型与定义

表 3(其中,M[addr]表示访问 addr 指向的内存单元.◇b 表示二元算术逻辑运算)按行为语义定义了本工作用到的各类经典 gadgets(if-gadget 不在其中).注意,在 ROP 代码执行流下的“语义定义”与正常代码下不同.以 NopG 为例,它被定义为“无操作”是指它不改变 SP 和 IP 以外的寄存器和内存状态,其行为与执行 RET 指令等价.默认情况下,表中的 AddrR、InR 和 OutR 都不包含 IP 和 SP 寄存器. ArithG 代表了在两个寄存器上进行算术逻辑运算的所有 gadget 类型.第 1.3 节定义的“减法运算”和“比较运算”都可以用该类 gadget 实现.另外,ArithConstG 表示寄存器操作数与常数进行算术逻辑运算的 gadgets;ArithLoadG 表示 gadgets 完成寄存器操作数与内存操作数进行算术逻辑运算之后,将结果加载到寄存器中;ArithStoreG 则表示将结果存储到内存单元中.

与 Q[9]相比,我们增加了 JumpG2、MemJumpG、RegFuncallG 和 MemFuncallG 这 4 种 gadget 类型.通常,以间接 CALL 和间接 JMP 指令结束的代码片段会被映射到这几类 gadget 上.与其他 gadget 类型不同,它们允许栈指针在执行后减小一个常数值,并改写 payload 上已被使用过的数据;JumpG2 虽然不会让栈指针在执行后变小,但同样会改写 payload 上的旧数据.在对 payload 进行布局时,我们会对这几类 gadget 进行额外的处理,以防

止不必要的内存改写操作.

Table 3 The gadgets can be found by our algorithm

表 3 我们的查找算法能够找到的经典 gadget 类型

Gadget 类型	输入输出寄存器	参数	行为语义定义	实例
NopG	无	无	无操作	ret
JumpG1	AddrR	offset	IP=AddrR+offset	jmp eax
MoveRegG	InR, OutR	无	OutR=InR	mov eax,edx;ret
LoadConstG	OutR, value	无	OutR=value	mov ebp,0x21;ret
LoadMemG	AddrR, OutR	offset	OutR=M[AddrR+offset]	pop ebp;ret
StoreMemG	AddrR, InR	offset	M[AddrR+offset]=InR	mov [ebx+0x40],eax;ret
ArithG	InR_1, InR_2, OutR	$\diamond_b$	$OutR=InR_1 \diamond_b InR_2$	add eax,ebx;ret
ArithConstG	InR, OutR	value, $\diamond_b$	$OutR=InR \diamond_b value$	add eax,0x20;ret
ArithLoadG	AddrR, OutR	offset, $\diamond_b$	$OutR \diamond_b=M[AddrR+offset]$	add eax,[ebx+0x40];ret
ArithStoreG	AddrR, InR	offset, $\diamond_b$	$M[AddrR+offset] \diamond_b=InR$	add [ebx+0x40],eax;ret
JumpG2	AddrR	offset	IP=AddrR+offset	push ebx;ret
MemJumpG	AddrR	offset	IP=[AddrR+offset]	jmp [eax]
RegFuncallG	AddrR	offset	IP=AddrR+offset	call eax
MemFuncallG	AddrR	offset	IP=[AddrR+offset]	call [eax]

3.3 Gadget的查找

每一个 gadget 除了满足相应的语义定义外,还要满足一系列的约束条件.我们对每一个 gadget 有 4 个属性要求,这些属性与 Q[9]定义的类似.

1) 功能性.每一个 gadget 都有如表 3 定义的功能.每一种 gadget 的类型都是通过语义分析得到的,都必须满足定义中的最弱前置条件.

2) 控制保持.每一个 gadget 都可以将控制传递给下一个 gadget.这些 gadget 以间接 CALL、间接 JMP 或 RET 指令结束.

3) 已知的副作用.Gadget 必须不存在未知的副作用,包括改写任何不希望改写的内存.

4) 常量栈偏移.Gadget 需要栈指针在执行后增加一个常数值,新增加的 4 类 gadget 除外.新增的 gadget 类型允许栈指针在执行后减小一个常数值.

当查找完经典 gadgets 之后,就可以查找 if-gadgets.为了查找 if-gadgets,需要预先记录下目标软件中所有条件判断指令的起始地址及其分支的目标地址.另外,为了查找类型 II 和类型 III 的 if-gadgets,还需要记录下所有以条件判断指令结束的代码块的起始地址.

3.4 Payload的布局

通常,每个 gadget 执行过程中,会从栈上取得参数并获取下一个 gadget 的地址,这些参数信息都存储在 payload 中.Payload 布局就是为这些参数进行栈空间分配的过程.

顺序执行的 ROP 代码,它们的 payload 布局过程很简单.只要按照 gadget 执行的前后顺序,为每一个 gadget 的参数和自身地址逐个分配栈空间即可.值得注意的是,新增加的 4 种 gadget 类型在执行后产生负的栈偏移.当它们运行时,可能会修改 payload 上的原有数据,特别是修改指向 gadget 的指针.例如,RegFunCall 类型的代码片段“CALL *%EAX”会重写 payload 上指向自身的 gadget 指针.拥有循环功能的 ROP 代码需要极力避免这种情况的发生.一种有效的方法是在执行这类 gadget 之前执行一个“RET $n”的代码片段.这个代码片段会分配一块私有“栈帧”给下一个代码片段.对私有栈帧的重写不会破坏 payload 上的原有数据.在 x86 架构下,“RET $n”这类代码片段很多.

为了更容易理解该过程,图 2 给出了一个实现条件判断的 payload 实例.偏移 12 字节处的 payload 内容指向一个 if-gadget.其假分支需要调用的下一个 gadget 通过指令“CALL *%EAX”来调用,为此,我们使用“POP %EAX;RET”把该 gadget 的地址预先加载到 EAX 寄存器中;其真分支要调用的下一个 gadget 的地址存放在 payload 的末尾,将通过位于 0x804840f 处的 RET 指令来调用.由于 if-gadget 的假分支是一个 RegFunCall 类型的 gadget,所以我们使用了一个额外的 gadget(即“RET $4”)来分配局部栈帧.指令“RET $4”将取出 if-gadget 的

地址 0x8048408 并将 SP 值加 4,之后再执行 if-gadget.这样,在 if-gadget 执行时,它将拥有一个 4 字节的私有栈帧(偏移 16 字节处开始的 4 个字节);0x804840a 处的“CALL *%EAX”指令对这 4 个字节的修改不会对原有 payload 造成破坏.因此,这段 ROP 代码可以放在循环语句中反复执行.

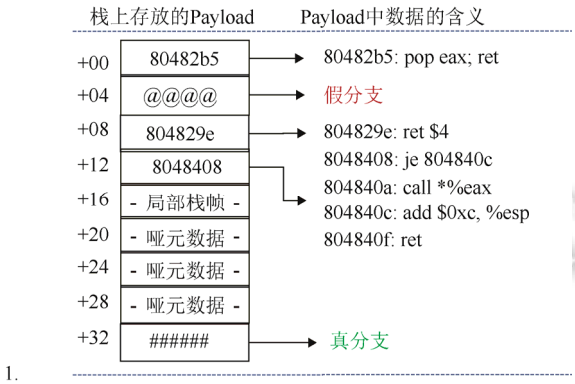

Fig.2 The layout of payload when using if-gadget to implement conditional jump

图 2 使用 if-gadget 构造条件判断功能所对应的 payload 布局

3.5 Gadget的调度

Gadget 调度将被分类的 gadget 组合起来,逐一实现每一个目标功能.我们通过迭代被分类的 gadgets 来找到实现一个目标需要的 gadget 序列.例如,如果要写一个值到内存,我们可以用两个类型为 LoadMemG 的 gadgets 来分别设置目标地址和数值,再用一个 StoreMemG 完成写内存操作.我们需要跟踪所有寄存器的状态,来保证一个在使用中的寄存器不会被后续执行的 gadgets 更新.

在实现条件判断功能时,需要考虑条件转化问题.在 x86 平台下,条件判断是通过测试 EFLAGS 寄存器上的标志位实现的.如果当前需要判断的标志并不能用已有的 if-gadget 直接测试,则需要将该标志移动到其他标志上.Shacham 等人[4]使用的方法是在 CF(carry flag)标志位上进行测试.如果需要测试其他标志,则需要通过按位逻辑运算将那些标志位转移到 CF 上.

引入 if-gadget 后,条件转化的解决方法要比 Shacham 等人提出的传统方法更简单.通常,在日常的可执行文件中能够找到好几种 if-gadgets,它们进行不同的条件测试.例如,在 x86 架构上运行的 Ubuntu 14.04 系统用 gcc 编译生成的 HelloWorld 程序中能够找到 4 个 if-gadgets,它们对 CF 和 ZF 这两个最常用的标志位进行测试.它们可以直接完成“>”“==”和“!=”这 3 种条件判断;如果对分支进行反转,则可以进行“<”条件判断;组合起来,能实现所有无符号比较操作.所谓的“分支反转”是将 gadget 的真(假)分支对应到实际代码为假(真)的逻辑.以表 2 中使用的 if-gadget 为例,如果要进行 JNE 判断,只需要让该 if-gadget 的假分支在执行后跳向攻击代码逻辑为真时所需要执行的代码块即可.最常见的 if-gadget 以 JE 或 JNE 开始.因此,在最坏情况下,我们可以通过按位逻辑运算把要比较的条件挪到 ZF 标志位上,再运行 if-gadget.

4 实 现

我们用 python 语言开发了一个可移植的通用的 gadget 查找系统.通过软件包 PyVex[12],该系统将机器代码提升为中间语言,然后再作语义分析和约束求解,得到经典的 gadgets.在处理二进制文件的时候,该系统也同时收集条件判断指令的信息;当找完所有经典 gadgets 之后,结合这些信息就能找到合法的 if-gadgets.图 3 是该系统的框图.因为是在中间语言的层次上作分析,该系统可以在多种硬件架构上的程序中查找 gadgets,包括个人电脑上常见的 x86 架构和移动终端流行的 ARM 架构.

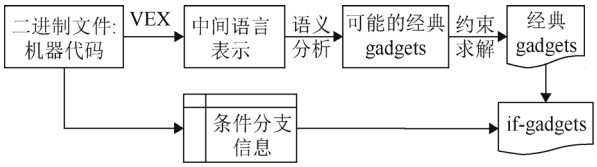

Fig.3 The framework of our gadget searching system

图 3 查找 gadget 的系统框图

4.1 约束条件

每一类 gadget 除了符合相应的语义定义外,还要满足对栈指针的修改约束和对内存的访问约束.总体上来说,栈指针 SP 的增量必须对齐到硬件架构的字长(例如,在 x86-32 架构上为 4 字节).默认情况下,表 3 中寄存器 AddrR、InR、OutR 都不包含 IP 和 SP 寄存器,但对于某些 gadget 类型,操作数可以使用 SP 寄存器.例如“MOV %ESP,%EAX;RET”是一个合法的 MoveRegG 类型的 gadget.这些 gadget 还需要满足对内存读写访问约束.多数类型的 gadgets 需要满足的内存访问约束条件是:只能读写旧栈值 SP_{old} 到新栈值 SP_{new} 之间的栈上区域,只有 LoadMemG 和 StoreMemG 等类型的 gadgets 才可以访问该区域之外的指定内存.

4.2 技术细节

本工作基于 pyrop[13]实现了一个 gadgets 查找系统.与原系统相比,我们改写了对 gadget 进行分类的大部分核心代码;修改了实施约束的代码;增加了查找 if-gadget 的代码.与 pyrop 一样,我们使用最弱前置条件来对经典 gadgets 进行分类:gadgets 除了满足语义定义外,还要满足内存访问约束.我们为每一个以间接控制流指令结束的代码片段生成一个可能的 gadget 集合,然后具体执行该代码片段若干次.每次具体执行都通过随机数来初始化每个使用到的寄存器和内存读操作,输出寄存器和内存写用来评估前置条件的真假.在多次具体执行中,一直存在的 gadgets 则被视为可复用的 gadgets.

任何必需的元数据都被视为 gadget 属性的一部分,这些数据已从代码片段中抽取出来.这些元数据包括输入寄存器、输出寄存器、被该 gadget 冲突的寄存器、下一个 gadgets 的地址相对于 SP_{old} 的偏移以及参数信息.这些信息是自动化调度 gadget 所不可缺少的,为我们在下一步工作中实现可实用的图灵完备 ROP 编译器奠定了基础.

为了查找 if-gadgets,我们记录下所有的条件判断机器指令的信息:包括当前指令的起始地址、两个分支的起始地址以及所实施的条件检测.为了找到类型 II 和类型 III 的 if-gadgets,我们还记录下所有可以用条件判断机器指令结束的代码块的起始地址.查找完所有的经典 gadgets 之后,我们会结合这些信息来查找 if-gadgets.

5 实 验

5.1 实验用例与参数设置

为了评估 if-gadget 在日常软件模块中的存在性,我们在 glibc 上进行了评估.同时,为了在代码量相对较小的可执行程序上进行评估,我们选取了 Coreutils 中的 10 个最常用的应用程序作为基准测试集,查找 if-gadget 在每个程序中的数量.这些软件模块都来源于 Ubuntu 14.04 的 32 位系统,运行于 x86 架构上.另外,我们在不同的硬件架构(即 x86 和 ARM)和不同的操作系统(即 Linux,Windows 和 MacOS)上,评估了 if-gadget 的存在性,并且观察了 if-gadget 的可移植性.

我们以 Binutils 为基准测试集,评估引入 if-gadget 后构造图灵完备 ROP 代码的可行性.为了与传统的方法进行对比,我们也在该测试集中评估了用 Shacham 方法构造图灵完备 gadget 集合的可能性.

5.2 If-Gadget在日常软件模块中的存在性

我们在 glibc 上统计了 if-gadget 的数量,总共找到 50 个可用的 if-gadgets.对攻击者来说,这个数量是非常丰

富的.但是,该结果并没有让人特别惊讶,毕竟 glibc 是一个大型的软件模块,拥有长度为 1 735 264 个字节的代码段.为了验证 if-gadget 在较小的软件模块中的存在性,我们进行了更多的实验评估.

Coreutils 中的应用程序实现了最基本的文件和文本操作,提供了用户与 Shell 交互的功能,这些程序在每一个 Linux 系统中都会提供.我们以 Coreutils 中的 10 个最常用的应用程序为基准测试集,评估 if-gadget 在日常二进制可执行程序中的存在性.这些程序包括 cat、ls、rm 和 ping 等,涵盖了广泛的应用功能.同时,我们也在程序 bash 中统计了 if-gadget 的数量.bash 是一个命令语言解释器,是最常用的 Shell,是 Linux 系统中最重要的程序之一.

表 4 列出了在这些程序中 if-gadgets 的数量.为了展示该数量与代码量的关系,我们也在表中列出了每个程序的代码量大小.数据表明,在这些程序中都存在 if-gadget,即使在只有 23KB 的 echo 程序中也有 4 个 if-gadgets.在应用程序 bash 中存在的 if-gadget 最多,多达 35 个.其代码量也相对较大,有 984KB.大体上,if-gadget 的数量随着代码量的增大而增多.

Table 4 The number of if-gadgets found in Coreutils programs

表 4 工具集 Coreutils 的最常用的可执行程序中的 if-gadget 数量

程序名	代码基大小(字节)	If-Gadget 数量(个)	程序名	代码基大小(字节)	If-Gadget 数量(个)
cat	42 200	4	ping	34 572	5
cp	120 812	4	ps	83 712	5
echo	22 968	4	pwd	24 132	4
grep	175 104	4	rm	52 232	7
ls	105 440	6	bash	964 216	35
mv	114 064	6			

我们发现,在每一个可执行程序中都有 4 个 if-gadgets 是来自于库函数.在库函数 register_tm_clones 和 deregister_tm_clones 中各有 2 个 if-gadgets.由于这两个库函数会静态链接到几乎所有的可执行程序中,因此,在 Linux 系统中,if-gadget 的存在性是普遍的.至少,在进行该实验的硬件架构上(即 x86-32)是如此.

5.3 If-Gadget的存在性与平台的相关性

我们进一步在 Windows(Windows 10 教育版)和 MacOS(OSX EI Capitan 10.11.6)操作系统下评估了 if-gadget 的普遍存在性.我们采用的策略是查看静态链接到程序中的运行时代码中是否存在 if-gadget.该策略给出的是一个“保守”结果:即如果这些代码中不存在 if-gadget,则不能给出有效结论;否则,表示 if-gadget 普遍存在.我们在这两个操作系统下编译了 HelloWorld 程序.在 Windows 下,用 VS 2015 编译生成了一个调试版本和一个发行版本.在 MacOS 下,用 gcc 进行编译.这些编译过程都采用默认选项.Windows 系统下的调试版本和发行版本各有 6 个和 4 个 if-gadgets;而 MacOS 下则没有.由此,可以得到这样的结论:在 Windows 系统下的可执行程序中,if-gadget 普遍存在;在 MacOS 系统中,由于静态链接到可执行文件的代码太少,不能给出有效结论.

为了进一步评估 if-gadget 的存在性与硬件架构的关系,我们在 x86-32、x86-64 和 ARMv7 下做了进一步测试:观察同一个 if-gadgets 的存在性是否因硬件架构不同而有所改变.

在该测试中,x86 硬件架构上都运行 Ubuntu 14.04 操作系统;在 ARM 上运行 Debian Jessie 8 操作系统.这两个操作系统都是流行的 Linux 系统的发行版本,而且可以配置成使用相同的静态链接库版本.在每个系统中,都用 gcc 的默认选项来编译一个 HelloWorld 应用程序,并在生成的二进制文件中查找 if-gadgets.可以发现,从静态链接库中引入的 if-gadgets 有可移植性,即同一个 if-gadget 在不同硬件架构上都存在.这是一个很有意思的观察结果,可以为开发跨平台的 ROP 代码提供有价值的参考信息.

表 5 给出了函数 register_tm_clones 中的一个 if-gadget 在不同架构下的出现情况.在 x86-32 架构下,该 gadget 从地址 0x80483b0 开始,真分支以 RET 指令结束,假分支以 CALL *%EDX 结束.在 x86-64 架构下,该 gadget 从地址 0x4004cc 开始,真分支以 RETQ 指令结束,假分支以 JMP *%RD*X* 结束.在 ARM 上的出现非常简单,该 gadget 从地址 0x10390 开始,真分支以指令 bx lr 结束,假分支以 bx r3 结束.

Table 5 The appearances of the same if-gadget across different CPU architectures

表 5 同一个 if-gadget 在不同硬件架构上的出现

x86-32	x86-64	ARM v7
80483a8: ret	4004c2: pop %rbp	10390: cbz r3, 10394
80483a9: mov $0x0,%edx	4004c3: retq	10392: bx r3
80483ae: test %edx,%edx	4004c4: mov $0x0,%edx	10394: bx lr
80483b0: je 80483a8	4004c9: test %rdx,%rdx	
80483b2: push %ebp	4004cc: je 4004c2	
80483b3: mov %esp,%ebp	4004ce: pop %rbp	
80483b5: sub $0x18,%esp	4004cf: mov %rax,%rsi	
80483b8: mov %eax,0x4(%esp)	4004d2: mov $0x601040,%edi	
80483bc: movl $0x804a020,(%esp)	4004d7: jmpq *%rdx	
80483c3: call *%edx		

5.4 图灵完备ROP代码的可构造性

图灵完备 ROP 代码在大型软件模块中的可构造性已被证实[4,7,8,10].本节主要评估图灵完备 ROP 代码在代码量较小的可执行程序上的可构造性.如第 1 节所述,该评估对评价当前主流系统的安全状态有重要意义.

在 Ubuntu 14.04 系统下,我们以 Binutils 为基准测试集,评估可执行程序中图灵基本功能的可构造性.Binutils 是一组对二进制文件进行操作的工具,广泛用于二进制文件的编译生成和分析中.我们统计了可以实现第 1.3 节所述的 6 种基本功能的 gadget 的数量,表 6 给出了其中 4 种功能对应的 gadget 的出现.由于比较运算通常可以由算术逻辑运算来实现,因此表中只列出了进行算术运算的 gadgets 的数量(即第 3 列“Arith”).由于系统调用几乎可以用任何 gadget 调用库函数实现,因此可实现方式与可复用 gadget 数量相同,在表中省略了该信息.第 4 列(“Load”)表示加载内存内容到寄存器;第 5 列(“Store”)表示存储寄存器内容到内存;第 6 列是 if-gadget 的数量.由表 6 可见,在每一个可执行程序中,每一个图灵基本功能都存在若干个实现.特别地,在代码量仅有约 20KB 的 addr2line,c++filt,elfedit,size,strings 等程序中,都可以实现各种图灵基本功能.因此,引入 if-gadget 后,图灵完备的 ROP 代码是普遍可构造的.

为了与传统构造方法在实现图灵完备的难易程度上进行对比,我们以 Shacham 方法为例,对传统构造方法进行评估.由于实现条件判断逻辑是传统构造方法的难点,因此,我们仅以对该功能的实现为例,作最弱前置条件评估.具体地,先用 ROPgadget[5]在每个程序中查找所有可复用的 gadgets,然后统计 Shacham 方法中实现条件跳转逻辑所需的各类 gadgets 的出现.我们侧重于统计 Shacham 方法中的关键 gadgets 的信息,并且总是试图查找所有可能的出现.例如,Shacham 实例中用“adc %cl,%cl;ret”这样的 gadget 来把条件标志位存储到寄存器中.为此,我们对所有以“adc Reg,Reg”开始的代码片段进行计数,其中,Reg 表示硬件架构提供的所有可能的通用寄存器,如 EAX,AX 和 AL.

表 6(其中,左侧给出了 4 种基本图灵功能所对应的 gadget 的出现.为了与统方法进行对比,右侧给出了 Shacham 方法实现条件转移逻辑功能所需要的关键 gadgets 的出现次数.NEGReg,ADCReg,NEGMem,ANDRegMem 和 ADDMemESP 分别表示以命令 NEGL Reg;ADC Reg,Reg;NEGL [Reg*];AND Reg1,[Reg2]和 ADD [Reg],ESP 开始的可复用代码片段.每个命令模板中,Reg 表示 x86-32 硬件架构提供的所有可能的通用寄存器)给出了 Shacham 方法需要的 5 个关键 gadget 类型的出现次数(二进制流相同的代码片段只计数 1 次).只有在 dwp 和 ld.gold 这两个较大(都有大于 2MB 的代码)的程序中才可以找到较多的 gadgets.尽管如此,在所有程序中都没有找全这些关键 gadgets,所有程序中都缺少对内存操作数取反的 gadget(如“NEGL 0x5e(%EAX)”).虽然该操作可用其他方法实现(例如,可以先把内存内容加载到寄存器,再用 NEGL 指令处理,然后把结果存到内存单元),但是这会增加需要串接的 gadgets 的数量,使得消除副作用的后续处理变得更难.由此可见,用传统方法在代码量相对较小的软件模块中构造图灵完备功能 ROP 代码非常困难.

Table 6 The appearances of gadgets for achieving Turing-complete functionalities in Binutils programs

表 6 在 Binutils 程序集上构造图灵完备功能

程序名	代码基大小(KB)	引入 if-gadget 后,实现图灵完备功能所对应的各类 gadgets 的出现				Shacham 方法实现条件转移所需关键 gadgets 的出现(二进制流相同的 gadget 只计数 1 次)				
		Arith	Load	Store	If-gadget	NEG-Reg	ADC-Reg	NEG-Mem	ANDR-egMem	ADDM-emESP
addr2line	20.5	820	425	1	5	0	0	0	0	0
ar	49.6	2 036	1 095	12	10	0	1	0	0	0
as	338.9	15 401	7 942	267	32	2	10	0	1	1
c++filt	18.1	744	418	2	4	0	0	0	0	0
dwp	2 272.2	149 180	80 174	2 385	155	4	7	0	7	18
elfedit	24.0	1 343	750	6	4	0	0	0	0	0
gprof	91.2	3 516	2 000	23	7	0	0	0	0	0
ld.bfd	1 011.2	17 590	9 702	173	41	4	2	0	0	4
ld.gold	2 438.4	157 120	83 467	2 970	205	4	5	0	28	20
nm	32.0	1 302	665	4	4	1	0	0	0	0
objcopy	206.0	10 526	6 375	111	9	1	0	0	0	1
objdump	326.0	15 084	8 236	138	11	0	1	0	0	2
ranlib	49.6	2 060	1 096	12	10	0	0	0	0	0
readelf	399.4	8 781	4 079	44	7	1	0	0	1	4
size	21.8	940	524	4	4	0	0	0	0	0
strings	20.3	899	481	2	4	0	0	0	0	0
strip	206.0	10 533	6 375	112	9	1	0	0	0	1

5.5 实验结论

通过这些实验,我们可以得到如下重要结论.

- 在 Linux 和 Windows 操作系统的日常软件中,包括最简单的 Helloworld 应用程序中,if-gadget 普遍存在.因此,引入 if-gadget 后,将 ROP 构造中的一个最难问题变成了一个普通的简单问题.
- 有些 if-gadget 有可移植性,即含有 if-gadget 的代码,在不同硬件架构上的版本都含有该 if-gadget.这为构造跨平台的 ROP 代码提供了可能.
- 在一个软件模块中,if-gadget 的数量大体上随着软件代码量的增加而增多.在拥有大于 200KB 代码量的软件中,应该存在足够丰富的 if-gadget 可以帮助 ROP 代码实现条件转移逻辑.
- 引入 if-gadget 后,在日常软件模块中,特别是代码量相对较小的日常可执行程序中,可以普遍性地用 ROP 技术实现图灵完备功能.这是本工作的核心结论,该结论证实了攻击者可以在可执行程序中构造纯 ROP 代码进行攻击.因此,ROP 攻击对现实系统的威胁比原来认为的严重得多.

6 相关工作

6.1 ROP攻击与防御

自从 ROP[4]在 2007 年提出以来,相关的攻击和防御技术一直成为研究的热点.ROP 攻击可以在众多的硬件架构上进行,包括 x86[4]、ARM[14]和 SPARC[15].

另外,不用以 RET 指令(或语义等价)结束的代码片段,而用其他间接控制流指令(特别是 JMP 指令)结束的代码片段,也能构造攻击代码[10].在支持脚本语言的软件(如浏览器)上,攻击者可以动态地查找 gadget 并构造攻击代码,此即 JIT-ROP 攻击[16].该攻击可以有效地绕过地址空间随机化(ASLR)保护机制.由于浏览器是最常用的软件之一,对这类软件的攻击会造成大范围的危害.因此,如何保护浏览器等软件防范 JIT-ROP 攻击受到了大量关注[17−19].Blind-ROP 证实[20],即便是在没有被攻击目标程序的二进制文件的情况下,仍然可以进行漏洞利用攻击.由此可见,ROP 的攻击能力和对防御技术的忍耐性强于其他已有的攻击类型.

已有大量的防御 ROP 攻击的技术.G-free[21]消除所有非法的间接控制流指令(从合法指令的中间开始的指令),同时保护合法的控制流指令免于滥用.“Return-less kernel”[22]则试图消除内核代码中所有可能的 RET 指令;用一个间接 JMP 指令来取代 RET 指令;所有返回地址都用索引值来替代,JMP 指令通过索引值来获得合法的返回地址.这两个工作都能很好地从根源上缓解 ROP 攻击,但它们都需要重新编译源代码.

ASLR 是缓解 ROP 攻击的有效方法[3].但是,在 32 位系统上,只有 16 比特可用于随机化.由于随机化的熵比较低,暴力攻击在几分钟内就能计算出被攻击模块实际加载的地址[23].目前,在 64 位系统上进行攻击也成为可能.随后,有若干工作提出了细粒度的地址空间随机化,以提高随机化的熵.ASLP 以函数为单位进行随机化[24],IRL 在指令级别[25].但是这些方案并不能在程序的每一次执行过程中进行随机化,为了解决这个问题,Wartell 等人[26]构造了一个工具 STIR,该工具在程序的每一次执行前,对可执行文件的代码块进行随机化.信息泄露是 ASLR 的一个弱点.当存在一个可以被反复利用的信息泄露漏洞时,攻击者可以在运行时动态地收集 gadgets 并构造 ROP 代码[16].相对应地,出现了一些防止信息泄露的工作[27−30].另外,旁信道也是攻击 ASLR 的一种可行方法[31−33].但是,在很多 Linux 系统上,如 Ubuntu,由于性能和兼容性问题,可执行文件通常并不支持 ASLR[5].因此,攻击者可以在这些程序中查找 gadget 进行 ROP 攻击.

6.2 图灵完备的gadgets

ROP 在理论上是图灵完备的.为了证实该结论,Shacham 等人[4]在提出 ROP 概念时,在 libc 上构造了图灵完备的 gadgets 集合.Checkoway 等人[7]发现,x86 和 ARM 上存在一些与 RET 行为等价的指令;在实现 ROP 编程时,可用以这些指令结束的代码片段构造功能代码.JOP[10]证实了可以用其他间接控制流指令(特别是间接 JMP 指令)结束的代码片段来构造攻击代码;JOP 与 ROP 有同样的表达能力.Microgadgets[8]用短小的代码片段(只有 2~3 字节)在 x86 架构上构造图灵完备的 gadgets 集合.值得注意的是,它们在实现条件转移逻辑时都将条件标志位转化成一个数值,再将该数值加到 SP 上.这些方法都需要使用数十个不同类型的 gadgets 来实现该基本功能.

6.3 ROP构造工具

目前,已有好几种工具可以用来查找可复用的代码片段,并给出一些构造 ROP 链的建议信息.Q[9]可以在一个二进制文件中为给定功能自动地构造 payload,但 Q 不开源.pyrop[13]是一个开源的 ROP 编译工具,给定目标功能和二进制文件,可自动构造 payload.但是,Q 和 pyrop 都不支持构造图灵完备的 ROP 代码.ROPC[34]用传统方法来构造 ROP 代码,并演示了图灵完备的 ROP 代码的编译过程,但它不能在实际的二进制文件上工作.mona.py[35]和 ROPgadget[36]的主要功能集中于实现 gadget 的查找,会给出一些构造 ROP 的简单建议信息.

7 总结与结论

返回导向的编程(ROP)被广泛用于当前的软件漏洞利用攻击中.尽管主流操作系统采用了一些有效措施来防范 ROP 攻击,但仍然存在一种可能的攻击方式,即攻击者通过复用不支持 ASLR 的可执行程序中的 gadgets 来构造纯 ROP 攻击代码.

本工作通过更新 ROP 的构造技术,证实了这种攻击是普遍可实现的.本工作突破了传统 ROP 构造思想对条件判断机器指令的认知,引入了一种新的可复用的代码片段,称为 if-gadget.在 Windows 和 Linux 上的实验结果表明,if-gadget 在日常软件模块中普遍存在.在 Bintuitls 程序集上的实验结果表明,引入 if-gadget 后,构造图灵完备功能要比用传统方法容易得多,在每个程序中都能用 ROP 技术实现图灵完备功能.

因此,本工作证实了 ROP 攻击对现实系统的威胁比原来认为的严重得多.在 Ubuntu 这样的主流系统中,一些实用程序和服务器程序都不支持 ASLR,而且部分程序还拥有 SUID 属性.攻击者可以通过攻击这些程序来获取系统用户权限,甚至超级用户权限.当攻击成功后,攻击者可以对系统安全造成严重破坏.

在理论上,该工作也丰富了对 ROP 属性的认知.已有工作证实了 ROP 是图灵完备的,且可以用短小的 gadget 来构造图灵完备的 gadget 集合.本工作则探索了图灵完备 ROP 代码的可实现性与软件模块大小之间的关系,证实了 ROP 图灵完备的普遍可实现性,特别地,证实了在代码量相对较小的日常可执行程序中也可以构造出图灵完备的 ROP 代码.

References:

[1] CVE. 2017. http://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=glibc

[2] Team P. PaX non-executable pages design & implementation. 2003. http://pax.grsecurity.net/docs/pageexec.txt

[3] Team P. PaX address space layout randomization (ASLR). 2003. http://pax.grsecurity.net/docs/aslr.txt

[4] Shacham H. The geometry of innocent flesh on the bone: Return-Into-Libc without function calls (on the x86). In: Proc. of the CCS 2007. ACM, 2007. 552–561. http://doi.acm.org/10.1145/1315245.1315313

[5] Payer M. Too much PIE is bad for performance. 2012. http://e-collection.library.ethz.ch/view/eth:5699

[6] PaX (http://pageexec.virtualave.net). 2014. https://grsecurity.net/PaX-presentation

[7] Checkoway S, Davi L, Dmitrienko A, Sadeghi AR, Shacham H, Winandy M. Return-Oriented programming without returns. In: Proc. of the CCS 2010. ACM Press, 2010. [doi: 10.1145/1866307.1866370]

[8] Homescu A, Stewart M, Larsen P, Brunthaler S, Franz M. Microgadgets: Size does matter in Turing-complete return-oriented programming. In: Proc. of the WOOT 2012. 2012. 64–76. https://www.usenix.org/conference/woot12/woot12-final9.pdf

[9] Schwartz EJ, Avgerinos T, Brumley D. Q: Exploit hardening made easy. In: Proc. of the USENIX SEC 2011. 2011. https://www.usenix.org/legacy/events/sec11/tech/full_papers/Schwartz.pdf

[10] Bletsch T, Jiang X, Freeh VW, Liang Z. Jump-Oriented programming: A new class of code-reuse attack. In: Proc. of the ASIACCS 2011. ACM, 2011. 30–40. http://doi.acm.org/10.1145/1966913.1966919

[11] Nürnberg PJ, Wiil UK, Hicks DL. A grand unified theory for structural computing. In: Hicks DL, ed. Proc. of the Int'l Symp. on Metainformatics. LNCS 3002, Berlin, Heidelberg: Springer-Verlag, 2003. 1–16. [doi: 10.1007/978-3-540-24647-3_1]

[12] Shoshitaishvili Y, Wang R, Hauser C, Kruegel C, Vigna G. Firmalice-Automatic detection of authentication bypass vulnerabilities in binary firmware. In: Proc. of the NDSS 2015. 2015. [doi: 10.14722/ndss.2015.23294]

[13] Stewart J, Dedhia V. ROP compiler. 2015. http://css.csail.mit.edu/6.858/2015/projects/je25365-ve25411.pdf

[14] Kornau T. Return oriented programming for the ARM architecture. 2010. http://www.zynamics.com/downloads/kornau-tim-diplomarbeit-rop.pdf

[15] Buchanan E, Roemer R, Shacham H, Savage S. When good instructions go bad: Generalizing return-oriented programming to RISC. In: Proc. of the CCS 2008. ACM, 2008. 27–38. [doi: 10.1145/1455770.1455776]

[16] Snow KZ, Monrose F, Davi L, Dmitrienko A, Liebchen C, Sadeghi AR. Just-in-Time code reuse: On the effectiveness of fine-grained address space layout randomization. In: Proc. of the SP 2013. 2013. [doi: 10.1109/SP.2013.45]

[17] Niu B, Tan G. RockJIT: Securing just-in-time compilation using modular control-flow integrity. In: Proc. of the CCS 2014. ACM, 2014. 1317–1328. [doi: 10.1145/2660267.2660281]

[18] Davi L, Liebchen C, Sadeghi AR, Snow KZ, Monrose F. Isomeron: Code randomization resilient to (just-in-time) return-oriented programming. In: Proc. of the NDSS 2015, vol. 2015. 2015. [doi: 10.14722/ndss.2015.23262]

[19] Maisuradze G, Backes M, Rossow C. What cannot be read, cannot be leveraged? revisiting assumptions of JIT-ROP defenses. In: Proc. of the USENIX SEC 2016. 2016. 139–156. https://www.usenix.org/conference/usenixsecurity16/sec16_paper_maisuradze.pdf

[20] Bittau A, Belay A, Mashtizadeh A, Mazieres D, Boneh D. Hacking blind. In: Proc. of the SP 2014. 2014. [doi: 10.1109/SP.2014.22]

[21] Onarlioglu K, Bilge L, Lanzi A, Balzarotti D, Kirda E. G-Free: Defeating return-oriented programming through gadget-less binaries. In: Proc. of the ACSAC 2010. 2010. 49–58. [doi: 10.1145/1920261.1920269]

[22] Li J, Wang Z, Jiang X, Grace M, Bahram S. Defeating return-oriented rootkits with return-less kernels. In: Proc. of the 5th European Conf. on Computer Systems. 2010. 195–208. [doi: 10.1145/1755913.1755934]

[23] Shacham H, Page M, Pfaff B, Goh EJ, Modadugu N, Boneh D. On the effectiveness of address-space randomization. In: Proc. of the CCS 2004. ACM, 2004. 298–307. [doi: 10.1145/1030083.1030124]

[24] Kil C, Jun J, Bookholt C, Xu J, Ning P. Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software. In: Proc. of the Computer Security Applications Conf. IEEE Computer Society, 2006. 339–348. [doi: 10.1109/ACSAC.2006.9]

[25] Hiser J, Nguyen-Tuong A, Co M, Hall M, Davidson J. ILR: Where'd my gadgets go. In: Proc. of the SP 2012. 2012. 571–585. [doi: 10.1109/SP.2012.39]

[26] Wartell R, Mohan V, Hamlen KW, Lin Z. Binary stirring: Self-Randomizing instruction addresses of legacy x86 binary code. In: Proc. of the CCS 2012. ACM, 2012. 157–168. [doi: 10.1145/2382196.2382216]

[27] Backes M, Nürnberger S. Oxymoron: Making fine-grained memory randomization practical by allowing code sharing. In: Proc. of the USENIX SEC 2014. 2014. 433–447. https://www.usenix.org/conference/usenixsecurity14/sec14-paper-backes.pdf

[28] Gionta J, Enck W, Ning P. HideM: Protecting the contents of userspace memory in the face of disclosure vulnerabilities. In: Proc. of the CODASPY 2015. ACM, 2015. 325–336. [doi: 10.1145/2699026.2699107]

[29] Crane S, Liebchen C, Homescu A, Davi L, Larsen P, Sadeghi AR, Brunthaler S, Franz M. Readactor: Practical code randomization resilient to memory disclosure. In: Proc. of the SP 2015. 2015. 763–780. [doi: 10.1109/SP.2015.52]

[30] Tang A, Sethumadhavan S, Stolfo S. Heisenbyte: Thwarting memory disclosure attacks using destructive code reads. In: Proc. of the CCS 2015. ACM, 2015. 256–267. [doi: 10.1145/2810103.2813685]

[31] Hund R, Willems C, Holz T. Practical timing side channel attacks against kernel space ASLR. In: Proc. of the SP 2013. IEEE, 2013. 191–205. [doi: 10.1109/SP.2013.23]

[32] Seibert J, Okhravi H, Söderström E. Information leaks without memory disclosures: Remote side channel attacks on diversified code. In: Proc. of the CCS 2014. ACM, 2014. 54–65. [doi: 10.1145/2660267.2660309]

[33] Evtyushkin D, Ponomarev D, Abu-Ghazaleh N. Jump over ASLR: Attacking branch predictors to bypass ASLR. In: Proc. of the 49th Annual IEEE/ACM Int'l Symp. on Microarchitecture (MICRO). IEEE, 2016. 1–13. [doi: 10.1109/MICRO.2016.7783743]

[34] Pakt. ROPC: A turing complete ROP compiler. 2013. https://github.com/pakt/ropc

[35] Corelan. mona. 2016. https://github.com/corelan/mona

[36] Jonathan S. ROPgadget. 2016. https://github.com/JonathanSalwan/ROPgadget

袁平海(1982－),男,江西分宜人,博士生,主要研究领域为软件安全,系统安全.

曾庆凯(1963－),男,博士,教授,博士生导师,CCF 高级会员,主要研究领域为信息安全,分布计算.