软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2017,28(8):2026−2045 [doi: 10.13328/j.cnki.jos.005270]

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

软件二进制代码重用技术综述*

彭国军[1,2], 梁 玉[1,2,3], 张焕国[1,2], 傅建明[1,2]

[1](空天信息安全与可信计算教育部重点实验室(武汉大学),湖北 武汉 430072)

[2](武汉大学 计算机学院,湖北 武汉 430072)

[3](深信服科技股份有限公司,广东 深圳 518055)

通讯作者: 梁玉, E-mail: liangyu@whu.edu.cn

摘 要: 在当前的计算机系统架构和软件生态环境下,ROP(return-oriented programming)等基于二进制代码重用的攻击技术被广泛用于内存漏洞利用.近年来,网络空间安全形势愈加严峻,学术界、工业界分别从攻击和防护的角度对二进制代码重用技术开展了大量研究.首先介绍了二进制代码重用技术的基础.然后分析了二进制代码重用攻击技术的演变和典型攻击向量.同时,对基于控制流完整性和随机化的防护方法进行了讨论,对工业界最新的二进制代码重用防护机制 CET(control-flow enforcement technology)和 CFG(control flow guard)进行了剖析.最后讨论了二进制代码重用技术今后的发展方向,包括潜在的攻击面和防御机制增强的思路.

关键词: 信息安全;信息系统安全;软件安全;二进制代码重用;内存漏洞利用

中图法分类号: TP311

中文引用格式: 彭国军,梁玉,张焕国,傅建明.软件二进制代码重用技术综述.软件学报,2017,28(8):2026−2045. http://www.jos.org.cn/1000-9825/5270.htm

英文引用格式: Peng GJ, Liang Y, Zhang HG, Fu JM. Survey on software binary code reuse technologies. Ruan Jian Xue Bao/ Journal of Software, 2017,28(8):2026−2045 (in Chinese). http://www.jos.org.cn/1000-9825/5270.htm

Survey on Software Binary Code Reuse Technologies

PENG Guo-Jun[1,2], LIANG Yu[1,2,3], ZHANG Huan-Guo[1,2], FU Jian-Ming[1,2]

[1](Key Laboratory of Aerospace Information Security and Trust Computing (Wuhan University), Ministry of Education, Wuhan 430072, China)

[2](School of Computer, Wuhan University, Wuhan 430072, China)

[3](Sangfor Technologies Inc., Shenzhen 518055, China)

Abstract: Within the current commercial system achitecture and software ecosystem, code reuse techniques, such as ROP (return-oriented programming), are widely adopted to exploit memory vulnerabilities. Driven by the serve situation of cyberspace security, academical and industrial communities have carried out a great amount of research on binary code reuse from both defensive and offsensive perspevtives. This paper discusses the essence and basics of binary code reuse, along with an analysis of its technique roadmap and typical attack vectors. Corresponding defences and mitigations based on control flow integrity and memory randomization are analyzed as well. Dissections on CET (control flow enforcement technology) and CFG (control flow guard), two latest industrial techniques for binary code reuse mitigation, are presented. The future of binary code reuse, including protential attack vectors and possible mitigation strategies, is also discussed at the end of this paper.

* 基金项目: NSFC-通用技术基础研究联合基金(U1636107); 国家自然科学基金(61332019, 61202387, 61373168); 国家重点基础研究发展计划(973)(2014CB340600)

Foundation item: United Basic Research Foundation of NSFC-General Technology (U1636107); National Natural Science Foundation of China (61332019, 61202387, 61373168); National Basic Research Program of China (973) (2014CB340600)

收稿时间: 2016-08-31; 修改时间: 2016-11-04; 采用时间: 2017-01-29; jos 在线出版时间: 2017-03-17

CNKI 网络优先出版: 2017-03-17 14:37:33, http://kns.cnki.net/kcms/detail/11.2560.TP.20170317.1437.005.html

Key words: information security; information system security; software security; binary code reuse; vulnerability exploitation

软件作为信息系统的关键组成,控制着计算机系统设备的工作方式,其安全性关系到整个信息系统的安全、可靠和稳定.而软件作为人类智力活动的表达和产物,在其规模和复杂程度迅速增长的同时,就不可避免地存在漏洞(和缺陷)[1−3].在当前严峻的网络空间(cyberspace)安全形势下,软件漏洞以其高威胁、难防御、普遍存在等特点,被作为一种战略资源,广泛用于攻防博弈中[4−6].而本文则是在当前系统架构和软件生态环境下,从攻击和防护两个角度对软件漏洞利用中的二进制代码重用关键技术展开探讨和研究.

自 1988 年 11 月第 1 个缓冲区溢出漏洞攻击案例——莫里斯蠕虫(Morris worm)之后,针对软件漏洞的攻击与防护技术就已被广泛研究[7].回顾过去 30 年的发展历程,研究人员通过向编译器、操作系统和处理器这 3 个层面引入新安全特性,实现在源代码安全检查、二进制代码生成和软件运行过程中对软件进行保护,提高软件的安全性,试图消除或降低攻击者利用软件漏洞对信息系统进行攻击的可能性.

然而,攻防技术相生相长.虽然在硬件、操作系统和软件不同维度的防护下,传统的完全依赖于代码注入(code injection)的控制流劫持类漏洞的攻击方式已能被较好地防御,但攻击者仍可利用跳转指令将内存空间中已有的、分散的代码片段(code blocks)链接,构造出其所期望的、具有攻击目的的功能逻辑,最终实现在不引入外部代码的情况下,实施控制流劫持后的恶意攻击[8].这种利用内存中已经存在的代码片段实现新的功能逻辑的程序执行方式,就是二进制代码重用技术.当前,内存不可执行(non-executable memory,简称 NX)或数据执行保护(data execution prevention,简称 DEP)、动态代码签名(dynamic code-signing)等安全特性被广泛部署于主流操作系统中,使得漏洞攻击的难度显著提高.在这种情况下,代码重用技术的“无需注入具有可执行能力攻击代码”的特性使其能够有效绕过这些新型安全机制,其在漏洞利用中的关键性作用更加凸显.从已公开发布和披露的案例来看,代码重用技术已成为控制流劫持类漏洞利用中的一个必要环节.

最近 10 年,网络空间安全形势的愈加严峻,学术界、工业界都对二进制代码重用进行了广泛且深入的研究.这些研究涵盖基于二进制代码重用的攻击思路、相应的防护方法与检测机制等方面.本文从 3 个方面进行分析和探讨:(1) 二进制代码重用技术的基本原理、本质思想,不同平台架构、操作系统环境下的应用以及在攻防博弈下的进化;(2) 针对二进制代码重用攻击方法的防护和缓解机制的研究,涵盖从软件本身、程序运行环境(如操作系统)等角度进行的安全性防护以及针对攻击行为的异常检测方法;(3) 工业界对重要学术成果的转化和应用情况,如微软发布的 CFG(control flow guard)和 Intel 拟推出的处理器级别的代码重用防护技术——CET (control-flow enforcement technology).最后,本文对二进制代码重用未来的发展方向从攻击、防护和代码逻辑混淆等角度进行展望和探讨.

1 二进制代码重用关键技术

1.1 代码重用基本原理

在缓冲区溢出、UAF(use-after-free)等常见漏洞利用中,攻击者往往能够成功地控制线程调用栈中的返回地址或保存程序跳转目标(代码指针)的寄存器,形成控制流劫持.通过将控制流重定向到其所期望的任何有效代码区域,即可造成任意代码执行的高风险威胁.在早期的漏洞利用中,攻击者直接将所劫持的控制流重定向到已精心布局(注入)在栈或堆中的外部代码(shellcode),进行后续执行.但随着不可执行栈、NX(no-execute)、DEP 等内存不可执行技术的引入,使得攻击者直接注入到内存中的 shellcode 不具备执行权限,从而攻击失效.而 Apple 在 iOS 系统中引入的代码签名机制则比 DEP 等更为严格,只有包含已签名代码的内存页才具有执行权限,这也同样限制了外部注入的代码执行[9,10].

如果攻击者将控制流的跳转目标指向内存中某函数的入口处,并在栈上布局了该函数的调用参数,则可以实现对该函数的调用.若被调用的是类似 *system*()或 *execv*()等安全敏感的函数,则可在不注入代码的情况下造成有效攻击.但在实际攻击过程中,对单个函数的调用往往是不够的,攻击者通过使用位于函数末尾的类 ret 的指令片段,让控制流跳转的目标地址始终来自于其布局在栈上的数据,从而实现内存中分散指令片段的连续调

用,形成具有特定功能的大规模代码片段重用攻击.

图 1 是一个二进制代码重用示例,其功能是通过代码重用的方式实现了两个值的加法运算,并将结果写入指定内存地址 0x400000.通过在调用栈上精心布局指向内存中已有指令片段的指针和指令运行期间需要引用的数据,使得每个指令片段在完成其功能执行后能够通过 ret 指令"返回"到栈上预置数据指向的代码片段,即下一个指令片段,从而实现在具备不同功能的指令片段上的连续运行,最终实现特定的完整功能逻辑.

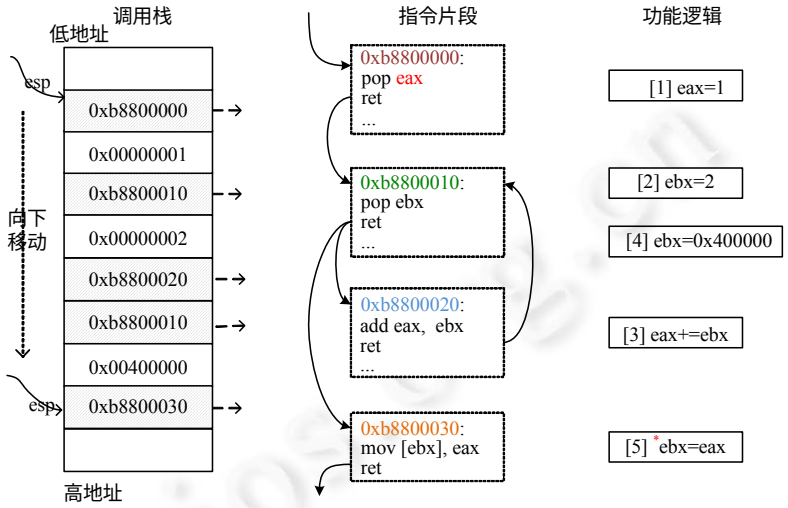


Fig.1 An example of binary code reuse and connected code blocks

图 1 二进制代码重用示例及指令片段连续调用

结合图 1 所示的例子,在典型的基于代码重用的执行方式中涉及如下 4 个要素.

(1) 程序调用栈:由攻击者控制,用于提供指令片段的指针和所需数据,是链接不同指令片段、确保连续执行的枢纽.

(2) 指令片段地址:被精确布局到调用栈中,使控制流正确跳转到目标指令片段.

(3) 指令片段:代码重用中的基本功能单位——指令,一般称为 gadget.根据 Schwartz 在文献[11]中的描述,gadget 需具备如下特性:

① 控制流保持:具备控制流转移能力,且当执行完该 gadget 中的指令序列后,控制流需跳转到一个可控的地址,以继续执行 gadget-chain 中的后续 gadget.

② 功能性:通过执行 gadget 中的指令序列能够实现特定操作,如从内存读数据或进行加法运算等.

③ 移动栈指针:为了保证控制流的持续可控,通常执行 gadget 的过程中需要移动栈指针 esp,使得 gadget 能够继续引用布局在栈上的其他数据,尤其后续跳转地址.

④ 不改变程序状态:gadget 在执行过程中不会产生一些不确定的结果,使得程序执行状态发生意外改变.

(4) 新引入的控制流:由代码片段的连续执行形成的新的功能逻辑.

基于代码重用技术的攻防对抗,正是以上述 4 个因素为关键点进行相应研究的展开,本文将在第 3 节、第 4 节中详述.

1.2 代码重用的图灵完备性

代码重用的概念在早期被称为返回导向式编程(return-oriented programing,简称 ROP)方法,说明其具备程序语言一样的能力[12,13].为了说明特定的代码重用方式在图灵机模型下能够进行任何操作,通常需要证明该代码重用方式的图灵完备性(turing completeness).一旦证明该代码重用攻击方法是图灵完备的,则从理论上说明

攻击者基于此种代码重用攻击模型能够实现任意攻击目的的程序行为[13−15].

Shacham 在 2007 年首次证明了基于 ROP 的代码重用方式是图灵完备的[13].证明代码重用方式具有图灵完备性,需证明该重用方法下的 gadget 能够实现如下功能:内存读/写、数据处理、控制流跳转、系统调用和函数调用.下面结合 kernel32.dll(Windows 32,X86)中的 gadget(ROP gadgets search: http://2www.ropshell.com/ropsearch?h=b921fb870c9ac0d509b2ccabbbbe95f3)分别进行描述.

(1) 内存读/写

一般情况下,内存读写主要有 3 种形式:加载常量到寄存器(load cons)、从指定内存加载数据到寄存器(load mem)、向指定内存写入数据(write mem).故要证明具备内存读/写能力,需存在上述 3 种类型的 gadget:通过类似于 pop REG 的 gadget 即可实现从栈到寄存器的常量加载;在 X86 下,通过 mov 指令可实现从内存加载数据到寄存器,如“mov eax, [ebp+0x10]; pop ebp; ret 0xc”;同理,内存写入 gadget 形如“mov [ebx+2], eax; ret”.

(2) 数据处理

若要具备完整的数据处理能力,则需要具有算术运算和逻辑运算的 gadget.其中,算术运算 gadget 需要包括用于进行算数加法、减法、乘法和除法的 4 类 gadget,而逻辑运算包括能够进行逻辑与、或、非和异或这 4 类 gadget.表 1 给出了相应的 gadget 示例.

Table 1 Gadgets for data processing (arithmetic operation and logic operation)

表 **1** 数据处理 gadgets 示例(算数运算和逻辑运算)

Gadget 功能	示例 gadget
算数加	add al, 0x3b; ret}
算数减	sub al, 0x3b; ret}
算数乘	mul [edx+4]; shrd eax, edx, 0x18; ret}
算数除	div esi; ret}
逻辑与	sbb eax, eax; and eax, [ebp+8]; pop ebp; ret 4}
逻辑或	push ebp; or [esi+0x5d], bl; ret 0xc}
逻辑非	not [ecx]; add [ebx−0xd37b], cl; jmp [esi−0x7d]}
逻辑异或	xor [eax+0x5c], ebp; ret}
逻辑与	sbb eax, eax; and eax, [ebp+8]; pop ebp; ret 4}
逻辑或	push ebp; or [esi+0x5d], bl; ret 0xc}

(3) 控制流转移

控制流转移分为直接跳转(unconditional-jump)和条件跳转(conditional-jump).对于前者,使用函数末尾类似于“ret”的返回指令进行任意目标的无条件跳转.对于条件跳转则较为复杂,Shacham 在文献[13]中给出了间接实现条件跳转的一种思路.

a) 使用指令 neg 对特定值进行求补码运算,操作数是否为 0 可以通过进位标志 CF(carry flag)体现出来.由 neg 指令特性可知:如果操作数为 0,则 CF 为 0;否则,操作数为非 0,CF 被置为有效,即 CF 值为 1.

b) CF 位反映了条件判定结果,通过使用移位指令或者带进位的加法即可取出 CF 值.一种简单的办法是使用类似于 xor ecx, ecx; adc cl, cl; ret 的指令片段将 CF 的值放入寄存器 ecx.

c) 将 ecx 向左移位 2n 位,ecx 的值则为 0 或 4^n,其中,4^n 是两个分支目标之间的 offset.只需要通过 gadget 序列实现功能 mov eax, offset; add esp, eax; ret,则实现由步骤 a)中的特定值来决定最终的跳转目标的功能,即条件跳转.

上述步骤中的后两步可能需要其他多个基础 gadget 的操作来完成,但该思路最终能够实现条件跳转.

(4) 函数调用、系统调用

对于 X86 和 ARM 等架构,其函数调用过程本身就是借助于栈实现的,其位于栈上的参数、返回地址等是攻击者可控的,故攻击者很容易地通过控制流转移 gadget 实现函数调用.早期的代码重用方式 ret2libc 就已实现函数调用.在系统环境下,直接或间接地进行系统调用是实现复杂功能的必要条件.在系统中,系统调用一般都是通过在特定寄存器中设置系统调用号,然后执行系统调用指令来实现系统调用的.例如,在 Windows 系统中,调用过程封装在函数 ntdlll!KiFastSystemCall 中,其内部实现为 mov edx, esp; sysenter; retn,由上使用 sysenter 进入

内核.对于特定功能的系统调用,通常则是封装在系统调用对应的功能函数中,例如 ntdll!NtReadFile,一般形如 mov eax,SYSCALL_NO;mov edx,offset SharedUserData!SystemCallStub;call dword ptr [edx];ret 24h.

所以,当需要进行特定系统调用时,使用 ret2libc 的方式调用相应的库函数;亦可在 eax 中设置系统调用号后,调用 KiFastSystemCall 实现.故系统调用的本质也是函数调用.

(5) Stack Pivot

在攻击过程中,如果将所有的用于组织代码重用攻击的数据都保存到栈上,则一方面有可能触发操作系统对栈的保护机制;另一方面也有可能导致栈上关键数据被覆盖,使得结束攻击者的执行流之后无法继续执行原有程序流程.因此,攻击者通过修改栈指针 esp 的值,使其指向攻击者完全可控的堆空间.esp 指向的堆上内存区域则称为伪造栈(forged stack),而用于改变 esp 值、进行栈切换的指令被称为 stack pivot 指令.例如,使用“xchg eax, esp; ret”的 gadget-chain 作为第 1 个 gadget,实现将栈切换到 eax 所指向的内存区域.Stack pivot 指令虽然不是图灵完备性中的必要条件,但这类指令的存在,极大地降低了构造代码重用攻击的难度.

上面仅讨论了基于代码重用的程序执行方式的图灵完备性,但是在实际漏洞攻击中一般不要求必须图灵完备,例如,较为复杂的条件转移 gadget 就很少用到.

1.3 代码重用自动化

基于代码重用的程序执行方式中有两个关键点:(1) 找到所需的 gadget,即 gadget 搜索;(2) 选择合适的 gadget 将其组装、拼接实现相应的功能逻辑,即编译过程.下面将对这两方面已有研究展开介绍.

1.3.1 Gadget 搜索

目前公开的 gadget 搜索工具有桌面工具 ROPGadget[16]和基于 ROPEME[17]提供的在线服务 ropshell[18].这些工具中的 gadget 搜索算法都是基于 Shacham 在文献[13]中提出并命名的 Galileo 搜索算法,其基本思想是:

(1) 在可执行文件二进制代码(指令序列)中顺序匹配类 ret 指令;

(2) 匹配成功后,从该 ret 指令位置开始进行反向的线性反汇编;

(3) 直到遇到无法反汇编的指令或跳转(分支)指令,或者到达预设值的最长 gadget 序列,则结束反汇编,从当前位置到 ret 指令间的指令序列即为一个 gadget;

(4) 从 ret 的下一条指令继续步骤(1)的过程,直到所有二进制代码分析结束.

通过上述流程,最后获得可以从二进制文件中搜索到的 gadget 集合 G.文献[13]中的 Galileo 搜索算法的匹配对象仅为 X86 平台上的 ret 指令,但实际上在后续的研究中可以作为 gadget 中控制流转移的指令还包括类似于 CALL [REG]的指令[14,15,19],对于 ARM 平台则有 bx REG 或 pop {…,pc}的 gadget 控制流转移指令[20,21].因此,一种更加通用的搜索方式是:根据具体的代码重用方法构建其用于匹配 gadget 的控制流转移指令集合 GT(对应于不同的控制流转移方式,见第 3.1 节),集合中的每一个指令像 ret 一样作为单一的匹配对象来搜索全部潜在 gadget 集合 G.

1.3.2 Gadget 自动化编译

获取 gadget 之后的下一个问题就是如何组织和利用搜索到的 gadget.对此,自 Shacham 于 2007 年正式提出 Return-oriented Programing 之后,gadget-chain 自动化编译相关研究就开始展开[11,22−24],思路基本一致:将 gadget 用某种中间语言(intermediate language,简称 IL)重新表示,然后使用编译器中的指令匹配思路,在提取到的 gadget 集合 G 上选择 gadget 来表示用户输入的用于功能描述的中间语言,确定最终执行的 gadget 指令序列.

典型的实现方案有基于 REIL(reverse engineering intermediate language)的 gadget 自动化搜索框架和基于 QooL 的 gadget-chain 自动化生成框架 Q[11].下面以 Q 为例,对其 gadget-chain 自动化编译的步骤进行介绍.

(1) 功能分类:根据 gadget 功能来分类定义指令集,使得每个 gadget 的执行相当于执行一条指令.Q 对 gadget 功能分类见表 2,其中,$M[addr]$表示访问地址 $addr$ 处的内存,$\diamond_b$表示任意二进制操作,$a\leftarrow b$ 表示将 b 的值赋给 a,$X\diamond_b\leftarrow Y$ 是 $X\leftarrow X\diamond_b Y$ 的缩写.

(2) Gadget 搜索:将所提取的 gadget 集合翻译为中间语言表达式,并按 gadget 类型进行归类.Q 在判定某个 gadget 是否属于特定功能的 gadget 类时,采用了程序验证(program verification)领域的程序最弱前

置条件(weakest precondition of a program)计算方法[25].对于 gadget $\mathcal{L}$ 和后置条件 $\mathcal{B}$ 的最弱前置条件 $WP(\mathcal{L},\mathcal{B})$,是一个判断 gadget 执行结束后是否满足条件 $\mathcal{B}$ 的布尔值.如果 $WP(\mathcal{L},\mathcal{B})\equiv$true,则认为 gadget 属于与后置条件 $\mathcal{B}$ 对应的功能类型.

(3) IL 功能描述:用户基于高级语言或直接使用中间语言描述所需功能.

(4) 功能分配:使用编译器思想选择不同功能的 gadget 类型表示用户功能描述,相当于编译器的指令选择过程.Q 使用最大匹配算法尝试选择出可能的表示.

(5) Gadget 映射:从每个功能类型的 gadget 中选择合适的实际 gadget 来实现当前所需的 gadget 功能,实现从 gadget 功能到实际 gadget 的映射.

通过上述步骤,Q 可以根据用户输入的 IL 描述和提供的二进制文件(或指令序列)生成最终可用 gadget 链.

Table 2 Gadgets categories in Q

表 2 Q 对 gadget 的功能分类

Gadget 类型	输入	参数	定义	功能描述
NoopG	–	–	–	不改变内存或寄存器内容的空指令
JumpG	*AddrReg*	*Offset*	*EIP*	跳转到目标地址
MoveRegG	*InReg*,*OutReg*	–	*OutReg*←*InReg*	向寄存器赋值
LoadConstG	*OutReg*,*Value*	–	*OutReg*←*Value*	向寄存器赋值
ArithmeticG	*InReg*1,*InReg*2,*OutReg*	$\circ_b$	*OutReg*←*InReg*1$\circ_b$*InReg*2	进行算术运算
LoadMemG	*AddrReg*,*OutReg*	#*Bytes*,*Offset*	*OutReg*←*M*[*AddrReg*+*Offset*]	从内存加载数据
StoreMemG	*AddrReg*,*InReg*	#*Bytes*,*Offset*	*M*[*AddrReg*+*Offset*]←*InReg*	将数据写入内存
ArithmeticLoadG	*OutReg*,*AddrReg*	#*Bytes*,*Offset*,$\circ_b$	*OutReg*$\circ_b$←*M*[*AddrReg*+*Offset*]	内存中数据进行运算后保存到寄存器
ArithmeticStoreG	*InReg*,*AddrReg*	#*Bytes*,*Offset*,$\circ_b$	*M*[*AddrReg*+*Offset*]$\circ_b$←*InReg*	将寄存器中数据进行运算后写入内存

虽然类似于 Q 的自动化代码重用工具能够实现 gadget 组装、拼接等繁琐工作,但就目前研究进展情况而言,代码重用自动化的效率仍然比较低,其性能瓶颈主要在于从提取的所有 gadget 中选择合适的 gadget,并编译出目标功能逻辑.

2 攻防对抗形势下代码重用技术的演进

长期以来,基于代码重用的程序执行方式被广泛用于漏洞攻击中,用来绕过代码不可执行、动态代码签名等安全机制.图 2 从时间维度给出了代码重用攻击的演变历程.

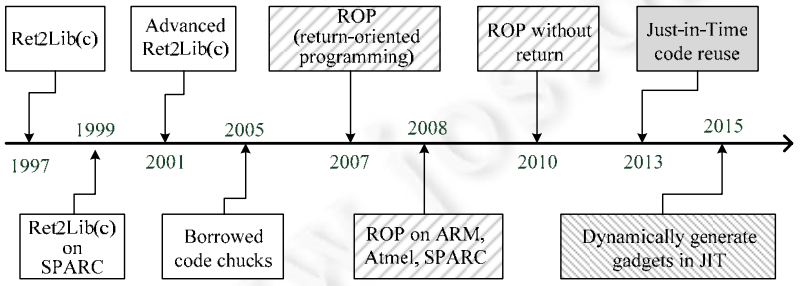


Fig.2 Evolution of binary code reuse-based attacks

图 2 二进制代码重用攻击的演变历程

整个历程分为 4 个阶段.

- 阶段 1,以 Ret2Libc 为核心思想的函数级别代码重用,跳转目标一般为函数开始位置[26−29].

- 阶段 2,以具备控制流转移能力的代码块级别的代码重用为核心.证明了基于代码重用执行方式的图灵完备性;经历从返回导向编程(ROP)到直接跳转导向编程(jump without return,简称 JOP)的发展;应用场景也扩展到 ARM,SPARC 等架构上[13,22,24,30−32].
- 阶段 3,以 Just-in-time code reuse 为代表的交互式环境下,具备对抗随机化能力的代码重用攻击[33−35].
- 阶段 4,动态生成所需的指令片段来进行代码重用攻击[36].

上述演变历程中,一方面将代码重用扩展到不同架构、平台上,例如,从早期 X86 架构扩展到当前广泛用于移动互联网设备的 ARM 架构上,但更重要的是另一方面——对代码重用攻击方式的进化.代码重用攻击涉及两个关键过程:

(1) 利用 gadget 的控制流转移特性来拼接具备不同功能的 gadget;

(2) 在内存代码页中定位并获取所需的 gadget.

在攻防对抗形式下,代码重用攻击方式主要是从这两个角度不断进化的,即扩展控制流转移方式和拓宽 gadget 获取方式.本节将主要从这两个方面详细分析,最后也介绍了在对抗代码重用攻击检测方面的进展.

2.1 控制流转移方式

根据第 1.1 节中对 gadget 的定义——gadget 必须具备控制流转移能力,而这种控制流转移能力是基于不同的程序分支指令实现的.

2.1.1 基于函数返回指令的控制流转移

利用函数末尾的返回指令实现控制流转移的方式在代码重用攻击中使用最早且最广泛,典型技术有 ret2libc 和 ROP 等.Ret2libc 是代码重用攻击思想,最早由 Solar 于 1997 年在 Bugtraq 邮件列表上提出来:介绍了通过覆盖函数返回地址将控制流重定向到 libc 库中的目标函数(如 setuid,system 等)的方法,从而解决栈上代码不可执行的问题[26].在该方法以及后续研究中,分别针对 x86/x64,SPARC 提出了更加完善的基于 ret2libc 的攻击利用方法,将函数粒度的 gadget 调用转变为短指令序列.例如,使用包含函数返回指令的短指令序列“pop reg;ret”对寄存器的赋值操作,使用 ret *n* 指令实现对函数的栈上参数的布局[27,28].2005 年,Krahmer 在文献[29]中实现的工具,能够通过 register-pop 代码序列构建任何的形式的参数.实际上,这与 ROP 已经非常接近了.

ROP 技术的出现,为基于代码重用的攻击提供了完备的理论支持.2007 年,Shacham 提出了 Return-to-libc without Function Calls(on X86)的方法,该方法与此前的基于 return-into-lib(c)方法相比的最大进步就是无需调用函数即可实现相应的功能[13].传统的方法使用“pop-ret”等短指令来连接不同的函数;Shacham 则证明了可以使用类似的短指令实现任何所需的功能,即证明了短指令的图灵完备性.

接下来,基于 Shcham 的理论,ROP 的代码重用方法被广泛推广,被逐步证明并应用到更多的指令集、系统架构等上面,如 ARM,SPARC 等[21,30,31,37].但对于不同的架构,对应的函数返回指令不尽相同.例如在 ARM 架构上,程序主要通过 pop {…,pc}和 pop {…,lr}; bx lr 的方式进行函数返回,但其功能类似于 ret 指令,可以作为代码重用攻击中的控制流转移指令.

2.1.2 基于函数调用或跳转指令的控制流转移

随着对基于 ROP 的代码重用的研究的深入,用于 ROP 中的短跳转指令也不仅仅局限于“pop-ret”,而是扩展到各种直接跳转指令和间接跳转指令上.Checkoway 等人在 2010 年提出了 ROP without return 的思路[15].通过分析典型 ROP 中 return 指令所承担两个角色:① 实现控制流跳转;② 更新寄存器状态,Checkoway 等人证明了使用 pop+jump 的指令组合成功实施 ROP 攻击的可行性;同时,他们也证明了 ROP without return 的图灵完备性.同时,Bletsch 等人和 Chen 等人提出了类似的不使用 return 指令的代码重用方法 Jump-Oriented Programing (JOP)[14,19].具体来说,Bletsch 等人首先指出,由于编译器优化的原因,使得 Checkoway 的方案[15]中所使用的 pop+jmp 类型的 gadget 数量非常少,很难实际应用;其次,pop+jmp 的方案依旧严重依赖于栈操作指令 pop,根据栈指针 sp 的变化来加载数据.而 Bletsch 提出的 Jump-oriented Programing 则使用了两类以 jmp 或 call 间接跳转指令结尾的 gadget 实现代码重用攻击,结合图 3 中 JOP 代码重用攻击的示例来看,

- 一类是作为跳转枢纽的 dispatcher 类指令,例如 add ebx, 4; jmp [edx],用于从位于攻击者可控内存区域

的 dispatch table 中获取跳转目标;

- 另一类是具备功能作用且能够跳转回 dispatcher 指令的功能性 gadget,例如 add eax,[ebx];jmp [edi],是一个用于进行加法运算的功能性 gadget.

相对于 pop+jmp 的代码重用方法,JOP 方案有更多满足条件的 gadget,且不依赖于栈进行控制流跳转.此外,国内学者茅兵、刑骁等人也提出了基于分支指令的 ROP 方案 BIOP(branch instruction-oriented programming).该方案是对 JOP 方案的改进和完善,使用 jmp 指令或 call 指令结尾的短指令序列作为 gadget 构造 ROP 攻击.该方法由于不引入 ret 指令,能够躲避一些检测方法[32].无论是 pop+jmp 还是 JOP,这种 ROP without Return 的思路不仅可以用于 X86,而且也可以应用于 ARM 等架构平台上[20].

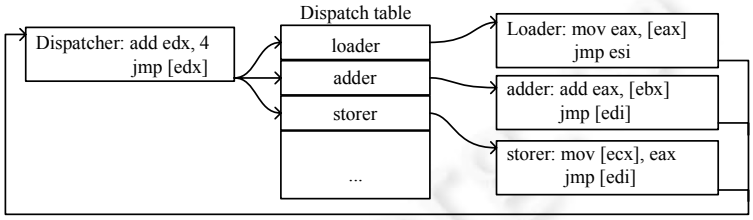


Fig.3 A code reuse example based on JOP

图 3 基于 JOP 的代码重用示例

2.2 Gadget获取方式

随着攻防形式的演变,攻击者为了绕过针对代码重用的各种防护策略,gadget 的获取方式也从传统的二进制离线分析扩展到内存在线分析和动态构建、生成 gadget 的方式.其中,“二进制离线分析”基于程序静态分析方法完成,而“内存在线分析”和“动态构建、生成 gadget”则依赖于程序运行时的动态分析.

(1) 二进制离线分析

一般情况下,二进制代码重用所需的 gadget 都是通过逆向分析将被漏洞程序加载到内存中的模块,从模块的代码段中搜索获得.基于二进制离线分析的方式获取 gadget 的典型工具有 ROPgadget,ROPshell 等.搜索工具通过这种方式获得的是 gadget 的模块内偏移地址,在最终组装 gadget 链时还需要加上实际的模块基地址.

(2) 内存在线分析

相对于传统的 ROP,Ret2Libc 等基于静态分析来构造代码重用的方法,Snow 等人在 2013 年提出了 Just-in-time code reuse 的动态代码重用方法[33,34].这种代码重用思路的重点是:在浏览器、文档阅读器等支持脚本运行环境的软件中,通过脚本与软件的实时交互获取内存分布情况,进而基于内存中的指令分布来动态地构造 ROP 链.在 Blind ROP 研究中,斯坦福大学的 Bittau 等人则是利用 Liunx 中一个服务崩溃后会自动重启,且服务重启后内存布局不发生变化的特性,实现了通过多次触发漏洞、动态调整基于代码重用的 payload,最终实现在不直接获悉内存地址的情况下,利用 ROP 成功攻击的案例[35].

(3) 动态构建、生成 gadget

动态构建、生成 gadget 的方式和内存在线分析的方法都是动态代码重用,前者侧重于动态获悉已加载模块的内存分布,后者侧重于动态生成所需的代码片段,取消了对程序内存中其他模块的依赖性.具体来说,在动态构建、生成所需 gadget 方面,2015 年,Athanasakis 等人提出通过定义特定的常量,使得浏览器中的 JIT 引擎动态生成所需的代码片段.此类代码片段具有较易定位的特点,并且能够绕过当前绝大多数防御机制,因此,可以更加有效地用于代码重用攻击[36].

2.3 代码重用攻击检测的绕过

除了从代码重用本身出发开展相关研究之外,研究人员也从对抗代码重用攻击检测的角度提出了一些专门针对检测技术(详见第 3.2 节)进行绕过的代码重用变形方案[38−41].

在当前针对软件系统漏洞攻击与防护的严峻对抗形势下,研究人员已从代码重用过程中控制流转移方式和 gadget 获取方式等增强和拓宽了代码重用技术的灵活性和可靠性,但在后续研究中,无论是出于学术研究还是黑客获取利益等目的,代码重用作为一种十分有效的漏洞攻击方法仍会被广泛关注,并促进其从不同维度去发展和进化.

3 二进制代码重用攻击的对抗策略与应用

二进制代码重用作为一种攻击技术,在漏洞利用中被广泛应用的同时,自然也催生了对其相应对抗策略的研究.本节一方面从代码重用攻击的防护和检测两个角度出发介绍了学术界近年来的研究;另一方面,对 CFG (control flow guard)和 CET(control flow enforcement)这两项近期被工业界实践和应用的防护方案进行了剖析和探讨.

3.1 代码重用攻击的防护

基于代码重用的漏洞利用攻击方式,通常需要两个前置条件[42−45].

(1) 内存中存在足够的、合适的能够完成功能需求的代码块(gadget),且通过在该代码块间进行控制流转移和控制流传递(保持)能够动态地引入新的功能性控制流;

(2) 能够准确定位到程序运行期间所需的 gadget,即,需要获悉 gadget 在程序运行期间的内存地址,该地址将作为控制流的跳转目标.

根据代码重用攻击所需前置条件,结合代码重用攻击对控制流跳转的影响以及对特殊指令和指令地址高度依赖的特性,当前针对代码重用攻击的防护思路和方法如图 4 所示.

- 一方面,以阻止非预期的控制流跳转为出发点,构建控制流完整性保护体系(control flow integrity enforcement);
- 另一方面,以降低攻击者对内存布局的知悉情况为出发点,引入增强型随机化策略(randomization)[44].

接下来,本文将对这两种防护思路进行详细分析.

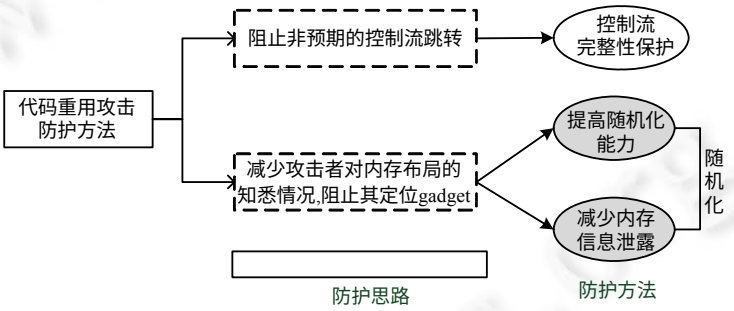


Fig.4 Defense strategies for code reuse attacks

图 4 代码重用攻击的防护思路

3.1.1 基于控制流完整性的防护方法

代码重用攻击方法是在劫持程序原有控制流之后,通过连续地执行一个个分散的 gadget 形成新的控制流、功能逻辑、实现所期望的功能,达到预期攻击目的.对于代码重用,

(1) 从宏观上来看,攻击者在原有程序上构造的控制流跳转和动态引入的功能逻辑都不是程序开发者预先设计和期望的.这种非预期的程序行为破坏了原有程序的控制流完整性,导致了程序行为异常.

(2) 从微观上来看,代码重用攻击所使用的大部分代码片段均是函数末尾指令或其他跳转指令跟随的代码块,执行的入口点也并非相应代码块所预期的一个跳转来源,即对一个函数或者原始代码块而言,代码重用的执行方式破坏了其正常的控制流转移和执行流程.

基于控制流完整性(control flow integrity,简称 CFI)的防护思路通过限制程序运行时的控制流转移,使应用程序的所有控制流转移均处于事先定义的预期控制流图(control flow graph,简称 CFG)内[46−49].具体来说,通过编译时的源码分析或基于二进制代码的静态分析,或运行时 profiling 来获取程序的预期控制流图;程序运行时,在控制流转移的指令附近进行额外的跳转目标验证,保证当前跳转目标处于预期控制流图内[40,46,47,50].因此,通过保证控制流的完整性,可有效防御代码重用攻击.

控制流完整性(CFI)在 2005 年由 Abadi 等人在文献[46,49]中提出.文献[49]为 CFI 提供了理论基础,使用程序语言理论分析和定义修改后程序的执行行为.文献[46]给出了基于二进制文件静态分析和二进制重写的一个 CFI 实现.具体来说,对应用程序中函数调用的目的地址、函数返回的目的地址进行编号、生成标识符 ID,在控制流转移前对函数调用的目的地址、函数返回的目的地址的 ID 进行校验,看是否属于合法跳转集合,如果不属于 CFG,则报错或抛出异常.

严格意义上的 CFI 需要对每个控制流转移的目标地址进行检查,保证所有的跳转都始终维持在相应的合法集合中.此外,如果要提供更好的防御效果,则还需从程序行为的角度,结合程序上下文进行分析.但这种方法粒度较细,且额外引入的指令较多,对系统性能造成了明显的影响[48].所以,后续的研究主要侧重于研究更实用的粗粒度的控制流完整性方案——在尽量不损失安全性的前提下提高效率,同时增强多架构多系统下的适用性.

Zhang 等人在 2013 年提出了将 CFI 与随机化结合的实现方法 CCFIR(compact control flow integrity and randomization)[51].该方法将间接调用指令和函数的返回指令的跳转目标进行区分,阻止非预期的跳转;同时,将所有的跳转指令的检查代码统一放在一个特定随机化过的内存区域,使得攻击者无法继续正常跳转或者绕过随机化,有效避免了大量插桩引发的性能损耗.此外,CCFIR 的实现方案支持增量部署,使其更具实用性.

Zhang 等人提出了一种针对 COTS(commercial off the shelf)程序的方法 binCFI.该方法相对于 Abadi 在文献[46]中给出的 CFI 方案、Zhang 的 binCFI 方案[51],不需要依赖于编译器,也不依赖于二进制程序中的重定位信息(relocation)、符号信息(symbol)和调试信息(debug)等,却能够应用更多复杂的、较为底层的二进制文件,提供更加全面的 CFI 应用方法,能够有效地保护包括二进制程序、共享库和加载器(loader)在内的模块内控制流转移以及模块间的控制流转移[52].具体来说,在细致地静态分析过程中,识别出作为间接跳转(indirect control flow transfer,简称 ICF transfer)目标的代码指针,并将其分类为编译时已经确定的常量指针、运行时动态计算出来的可计算型指针、指向异常处理代码指针、指向导出表的指针以及返回地址,然后分别对这几类进行细化分析,对不同类型的间接控制转移指令收集其合法跳转目标地址集合,设计相应的间接跳转替换,减少可被用于代码重用的 gadget.binCFI 通过反汇编、代码修改、再编译成目标代码(code in object file)、最后作为新的节放到原始二进制文件中作为代码节的方式与指令重写的方式相比,不改变原有代码,实现更便捷,对原代码透明.优点就是可以在不损失安全性、性能损耗很小的情况下很好地应用于系统程序和第三方程序.

binCFI 和 CCFIR 只是将间接调用指令和间接跳转指令的跳转目标进行验证.但在文献[53]中,Göktas 通过使用两种特殊的 gadget,即 entry point (EP) gadget 和 call site (CS) gadget,实现对 binCFI 和 CCFIR 等 CFI 的绕过,成功实施代码重用.对此,Mashtizadeh 等人提出了 CCFI(cryptographic CFI)方案.CCFI 一方面对跳转目标进行更细粒度的分类,即函数指针、函数返回地址、方法指针、虚表指针;另一方面,在程序运行期间保存代码指针时,对运行上下文信息加密保存,在程序返回时,则解密上下文信息并对比,进而判断跳转的合法性.额外的上下文信息比对,增强了 CFI 在对抗代码重用攻击上的安全性和可靠性[48].

Niu 等人在 2014 年提出了模块化 CFI(modular CFI,简称 MCFI).MCFI 允许模块在加载和链接时利用模块中的辅助信息生成新的 CFG,或者对 CFG 进行更新,以保证模块间的调用仍然遵循 CFI[54].MCFI 使 CFI 的实用性得到了增强.在 RockJIT 中,Niu 等人将 MCFI 的思想运用在浏览器中 JIT 引擎的安全性增强上[55].Mohan 等人的 O-CFI 则是将随机化与 CFI 相结合,使得攻击者无法获悉存在可被劫持的 CFG 中的边,实现了对代码重用攻击的抑制[56].在多架构、多系统应用方面,Davi 等人则将 CFI 应用在 ARM 架构上[57],Pewny 等人则是将其应用于 iOS[58].

除了上述通过验证跳转目标地址的 CFI 的实现方法之外,还有一类则是基于函数调用栈的特性,通过 shadow stack 实现对 call 和 return 的地址进行验证,从而阻止防护代码重用攻击[42,46,59−62].Lucas 等人在 2011 年实现了工具 ROPdefender,通过将函数调用的返回地址单独保存在一个称为 shadow stack 的内存空间,在函数返回时进行验证[60].ROPdefender 通过 Pin[63]进行运行时的指令监控,当调用 call 时,除了进行 call 指令本身的操作之外,同时将返回地址入栈到 shadow stack.当执行 ret 返回指令时,则验证当前的目标返回地址与 shadow stack 中栈顶的地址是否一致:如果不一致,则说明存在对函数的不完整调用——ROP 攻击一般所用的 gadget 代码块仅仅是函数的末尾部分.考虑到基于 Pin 的实现方法在一般场景中很难应用,同时,Davi Lucas 只对返回型 gadget 有效防护,但对于其他面向 call 的代码重用无效,因此,Qiao 等人在 2015 年提出了兼容性更好、防护更加全面的方案,并且将原来的 shadow stack 进化为 RCAP-stack(return capability stack).文中对 ret 和 call 的目标地址进一步细致分析,对用于代码重定位的 call 和用于跳转的 ret 指令进行单独处理,同时采用了应用性更好的基于二进制重写的实现方案[42].总的来说,该方案是对 Shadow-stack 思路的一个精细化研究和完善,提升了防御 ROP 攻击的能力.

在基于 CFI 的代码重用攻击对抗策略的学术研究基础上,近期,工业界基于 CFI 和 shadow stack 实现了安全机制 CFG 和 CET,试图构建从程序到运行环境的完整生态系统,以对抗代码重用攻击.具体细节将在第 3.3 节中分析.

3.1.2 基于随机化的防护方法

基于随机化的防护方法的出发点在于降低攻击者对内存信息的知悉情况,使得代码重用等严重依赖内存布局的攻击方法失效.通过对内存中模块、数据对象、代码指令等的地址和布局进行随机化,攻击者难以定位内存中关键对象、无法构造出可以有效执行的代码重用链(gadget chain),即打破了攻击者必须获悉所需 gadget 地址这一代码重用的必要条件,进而达到代码重用防护效果.在攻防对抗的实践过程中,基于随机化的防护方法的研究主要从两个方面展开:一方面是尽可能提升随机化能力本身;另一方面,则是在随机化的基础上防止和减少内存信息泄漏.本节将分别论述.

(1) 随机化能力提升

对随机化能力的提升,主要包括地址空间布局随机化(address space layout randomization,简称 ASLR)和指令随机化两个方面.地址空间分布随机化最初是由 PaX Team 于 2001 年提出,后来逐步被现在的绝大多数操作系统所广泛应用,如 Linux,Windows 7/8/10,iOS,Android(>4.0)[9,64−67].在应用程序启动时、重新加载模块时,ASLR 机制将会以一定的熵为应用程序随机确定程序加载的基地址、模块基地址、堆和栈的基地址,使得加载到内存中的模块内指令和堆栈上的数据地址无法预测.由于 ASLR 的随机化粒度为模块级别,所以一旦模块内的某个地址 $Addr_{\text{leaked}}$ 被泄露,则由该地址的固定偏移 $Offset_{\text{leaked}}$ 可计算得出当前模块的基地址,进而模块内指令或者堆栈上数据的实际地址 $Addr_{\text{target}}$ 可通过固定偏移 $Offset_{\text{target}}$ 计算得出,使得现有广泛使用的 ASLR 存在单指针泄露威胁(single pointer leakage threat).计算过程为 $Addr_{\text{target}}=Addr_{\text{leaked}}-Offset_{\text{leaked}}+Offset_{\text{target}}$.

此外,出于性能优化等目的,在部分系统上,同一模块在不同进程中的基地址是相同的.例如,Android 上的应用程序进程均是由 Zyogte 进程 fork 出来的,使得 libc 等众多系统库的基地址是相同的,攻击者可以通过协同攻击实现地址信息泄露和代码重用攻击的组合攻击[68].即通过应用程序 A 获得所需模块的基地址,通过其他方法将该地址泄露给应用程序 B,应用程序 B 则可以基于此地址定位到所需 gadget 的真实地址,构造出有效的代码重用攻击载荷.类似的问题也存在于 Linux 系统上的部分重要服务上,如 ngix 和 mysql 等,这些服务的主程序崩溃重启后,仍然使用相同地址空间映射,使得攻击者可以通过暴力尝试方式实现地址推测,获悉模块基地址,进而实施代码重用攻击[35].因此,在内存信息泄露漏洞的辅助下,ASLR 可以被轻易地绕过,使其防护代码重用攻击的能力弱化.

基于上述因素,自 2012 年起,针对代码重用攻击防御的研究开始转向如何通过实施细粒度的指令随机化和杜绝指令信息泄露来实现对代码重用攻击的防护.ILR[69],Binary Stirring[70],Smashing the gadgets[71]等研究通过采用重新排列函数顺序、指令顺序重排、等效指令替换、寄存器重新分配等方法,在不改变原有程序语义的前

提下,使得内存中 gadget 的语义发生变化、位置发生变化等,从而实现降低攻击者对程序运行时内存的知悉程度[33,34].图 5 给出了部分指令随机化策略的示例.Snow 等人在 Just-in-time Code Reuse 一文提出的动态生成 ROP gadget 链的方法有效地绕过了上述细粒度随机化,但考虑到该方法要通过在线读取内存中大量的可执行代码来实现代码重用攻击,所以攻击场景具有一定的局限性.梁玉等人则利用 ARM 指令特性,通过二进制重写技术(binary rewriting),在函数首尾的栈操作指令中随机地插入成对的空闲寄存器来实现栈帧布局随机化[45,67].例如,指令“push {r5,r6,lr}…pop {r5,r6,pc}”则可通过随机插入 r2 和 r7 变为“push {r2,r5,r6,r7,lr}…pop {r2,r5,r6,r7,pc}”.在程序运行时,随机化后的代码能够向栈帧中引入随机大小的 padding(r2 和 r7 中的数据)来改变栈帧中数据的相对偏移,从而使得布局在栈帧中的地址无法准确还原到寄存器 pc 中,进而使得代码重用攻击失败.

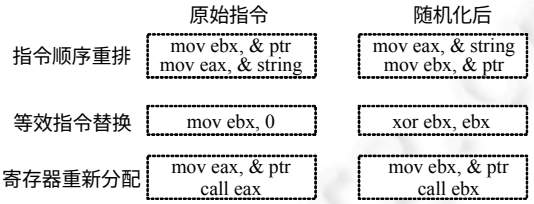


Fig.5 Examples of various instruction randomization strategies

图 5 部分指令随机化策略示例

除了指令的细粒度随机化外,研究人员也从随机化方式和随机化频率两个角度进行随机化能力提升[72].Oxymoron 利用 X86 的段特性实现了对库基地址的随机化,分页内存管理模式使得 Just-in-time Code Reuse 攻击中的页代码泄露失效[73];Isomeron 则是将程序在内存中布局两份,在函数返回跳转时,随机确定跳转目标的所在的程序布局[74];Lu 等人则实现了在程序每次 fork 子进程时都进行一次随机化,从而解决了 fork 的子进程与其父进程内存布局一致的缺陷[75].

(2) 缓解内存信息泄漏

此外,研究人员从防止和减少内存信息泄漏的角度,提出了新的防止可执行内存中的代码被读取的解决方案.防止可执行内存被任意读取的研究有 XnR[76],Readactor[77]等,这些方法或通过软件模拟 MMU(memory management unit,内存管理单元,负责虚拟地址映射为物理地址以及提供硬件机制的内存访问授权),或通过硬件与虚拟化(基于虚拟化的 MMU)相结合的方法实现了代码内存页的可执行不可读的特性,使得即使软件中存在单指针内存泄露,攻击者也无法通过该内存泄漏实现整个代码页、代码节的指令泄露,进而攻击者也就无法构造有效的攻击.值得关注的是,Readactor 是一个相对比较全面的防御机制,能够有效防止 JIT-Code Reuse 中提到的直接内存泄漏,也能防止 Isomeron[74]中提到的间接内存泄漏,进而结合细粒度随机化,可有效防止静态、动态的代码重用攻击.

3.2 代码重用攻击的检测

针对代码重用攻击的检测方法则是根据控制流跳转规则建立行为异常模型,实施检测.根据检测异常的探测点的不同,本节从控制流异常和栈异常两个方面进行论述.

3.2.1 控制流异常

在典型代码重用中,引入新的控制流并对短指令块的返回式调用、函数的不完整调用等程序执行方式属于控制流异常.据此提出的实用性较强的代表研究有 CFIMon[38]、kBouncer[78]和 ROPecker[41]等方案.这 3 种方案均是利用系统的硬件设备进行辅助,在降低性能消耗的同时,从代码重用攻击跳转频率较高、函数的不完整调用特性出发实现代码重用攻击检测的.这种代码重用检测思路是基于启发式的,需要设定每个 gadget 的指令数阈值和 gadget 数阈值,如图 6 所示.CFIMon 利用了性能计数器(performance counter)捕获程序控制流,进行合法性判定;kBouncer 和 ROPecker 则是利用了 LBR(last branch record)对最近捕获的 16 次跳转进行分析和合法性判定.相比之下,ROPecker 提供了更加综合、更加完善的代码重用检测方案,除了针对跳转频率和 gadget 长度的

启发式检测之外,还对触发检测的时机进行了有效选择——基于敏感系统调用触发异常检测,从而很好地控制了性能开销.但这些代码重用攻击检测方案仅针对大多数通用的攻击有效,对于针对性攻击,基于启发式的检测方法还是存在被绕过的可能,例如,Isomeron,Size Does Matter 等研究中均给出了对此类启发式检测方法的bypass 实例[74,79].

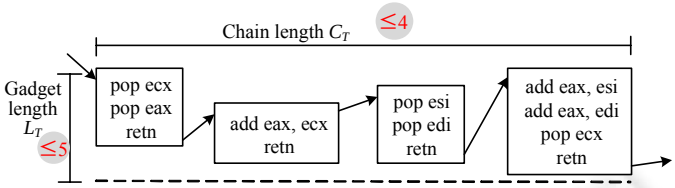


Fig.6 A heuristic detection strategy for code reuse attacks

图 6 基于启发式的代码重用攻击检测方法

3.2.2 栈异常

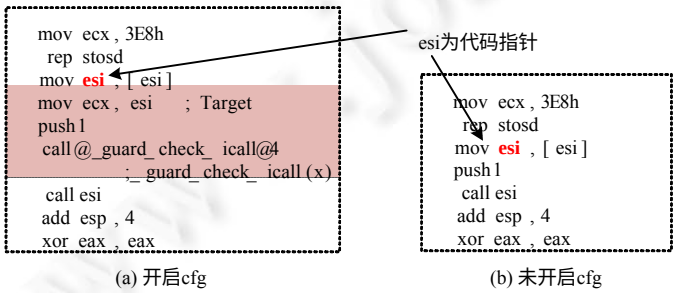
线程栈在程序函数调用过程中至关重要——保存了函数调用返回地址.利用线程栈的栈式结构,可以很容易地在栈上布局数据和程序跳转目标,实现 gadget 和函数的连续调用.为了方便布局攻击数据、尽可能地减小栈的破坏程度,攻击者通常会使用 stack pivot 指令将栈帧切换到一个位于堆上的伪造栈(forked stack)上.根据代码重用攻击过程中对栈或伪造栈的严重依赖性以及栈帧的异常切换,梁玉等人提出的 S-Tracker[80]和 Prakash[81]等人从栈完整性异常的角度对代码重用攻击进行检测.S-Tracker 对栈关键要素 esp,ebp 等进行了完整性定义,限制 esp 等指针的范围,在敏感行为发生时,触发对栈的检测;Prakash 等则是从代码重用中类似于 stack pivot 的关键 gadget 入手,使用 instrument 的方式对其修改,使得这类指令被调用时触发检测机制,通过判断栈指针完整性,实现对代码重用攻击的检测.

3.3 典型防护技术的实践与应用

对于防护方案,从理论、原型的提出到最终实际部署、应用是一个相对漫长的过程.近两年来,以 CFG 和 CET 为代表的基于控制流完整性的平台级防护方案在工业界得到应用和实践.

3.3.1 CFG

CFG(control flow guard)是微软对控制流完整性的一个平台级实现,通过代码编译和程序运行时(runtime)相结合的方式实现对 call 的间接跳转目标的限制,从而有效缓解包括代码重用在内的攻击方式.具体来说,微软在 Visual Studio 2015 中引入了 CFG,并通过代码生成选项"/guard:cf"开启对间接跳转指令的插桩.图 7 给出了插桩前后的对比.自 Windows 10 开始,微软正式引入了操作系统级的 CFG 支持.图 7(a)中额外调用的 _guard_check_ icall 在支持 CFG 的操作系统上,会通过调用 ntdll!LdrpVaildateUserCallTarget 对跳转目标进行验证;在不支持 CFG 的系统中是一个空调用,不引发额外的操作.操作系统通过位图 CFGBitmap 存储了当前调用的有效跳转目标集合,在 ntdll!LdrpValidateUserCallTarget 当中首先获取 CFGBitmap 对象,然后索引到对应的值,从而判断当前跳转是否有效:如果无效,则终止进程[82,83].



(a) 开启cfg (b) 未开启cfg

Fig.7 Comparison of code blocks generated by Visual Studio with CFG on and off

图 7 Visual Studio 开启 CFG 前后生成的代码对比

3.3.2 CET

2016 年 6 月,Intel 发布了称为 Control-flow Enforcement Technology(CET)的技术预览白皮书,正式公开了其准备推出的主要用于防护 ROP,COP/JOP(call/jump oriented programming)的芯片级解决方案.该方案的核心思想源自学术界早先提出的 shadow stack,即在系统中的特定内存区域构造一个专门用来存储间接跳转数据的栈[47,84].具体来说,当一个子函数被调用时,像传统方式一样,其返回地址将会被保存到线程栈上;同时也会被保存到 shadow stack 上;在程序后续执行中若遇到返回指令,则处理器需要保证线程栈上的返回地址与影子栈中的地址相匹配.如果二者不匹配,处理器则会抛出异常,使得操作系统能够捕获该异常并停止程序执行.

在 CET 技术中,Intel 从 CPU 级别对 shadow stack 提供了安全特性支持.Shadow stack 仅用于控制流转移操作,其中保存了函数返回地址等,与一般的数据栈相对独立.为了防止 shadow stack 被篡改,Intel 通过 CPU 的内存管理单元 MMU 在页表保护中提供了新的扩展属性,使得页内存除了具备读、写、执行保护外,还可被标记为 shadow stack 页.被标记为 shadow stack 的内存页,普通软件和指令无法直接对其进行写操作,只有通过特定的控制流转移指令和 Intel 提供的专门的 shadow stack 操作指令,才能实现对 shadow stack 的写操作,从而从硬件体系结构提供了 shadow stack 这一安全特性.

此外,Intel 还提供了一个新寄存器 SSP(shadow stack pointer).该寄存器始终指向 shadow stack 栈顶.当 CET 功能打开时,执行函数返回指令时,CPU 将自动地从 SSP 寄存器指向的 shadow stack 栈顶处取保存的返回地址,并与普通栈中的返回地址进行对比:二者如果不一致,CPU 则抛出异常,操作系统捕获该异常后进行后续处理.

和 CFG 一样,CET 技术也需要通过平台支撑才能完全实现防护效果.Intel 为 CET 技术提供了新指令 ENDBRANCH,该指令可用于标记合法的间接跳转目标.只有开发者使用支持 CET 技术的编译器来生成目标程序,才会使得目标程序中的返回地址等合法间接跳转目标被 ENDBRANCH 标记.在程序运行时,CPU 内部构建了一个状态机来跟踪 call/jmp 等间接跳转指令:当执行到 call/jmp 时,状态机由 IDLE 进入 WAIT_FOR_ENDBRANCH 状态.该状态下,其下一条指令必须是 ENDBRANCH:如果不是 ENDBRANCH,则说明跳转目标是非法的,触发 CET 保护,抛出异常;如果是 ENDBRANCH,则状态机再次进入 IDLE 状态.通过上述方式,实现对基于 COP/JOP 等代码重用方式的防护.

完善的防护方案需要构建从整个生态系统出发,尽可能地减少攻击面,提升安全性.CFG 和 CET 的设计和实施过程中,都是从底层硬件支撑、系统层程序执行环境保障、编译器层程序代码生成等环节进行了安全性和兼容性考虑,使其可快速地、最大化地被实际部署.但是,考虑到提供 CET 的 Intel 芯片目前尚未发布,且攻防对抗是一个持续、演进的过程,新的攻击方法存在绕过 CET,CFG 这类防护策略的可能性,所以二进制代码重用技术仍然是软件安全领域所面临的一个重要威胁.

4 总结与展望

代码重用作为一种程序执行方式,以其灵活的代码组织方式、功能逻辑动态生成的特点,在过去被广泛用于漏洞攻击当中.严峻的网络空间安全形势使得代码重用技术的研究价值凸显,学术界、工业界的研究人员在近 10 年中,从攻击和防护两个方面开展了深入的研究.本文前面部分重点对已有研究的分析和总结,一方面探讨了代码重用技术在攻击过程中的关键要素、逻辑组织方式及其攻防博弈下的演变历程;另一方面,从防护和缓解攻击的角度分析了已有防御技术的思路,如基于控制流完整性的防护思路和基于随机化的防护思路.此外,文中也对最新的来自工业界防护方法 CFG,CET 进行了分析,为研究实用性更强的防护技术提供了启发.

代码重用技术除了用于攻击之外,这种动态生成程序逻辑的程序执行方式亦可用于代码混淆,尤其是对关键功能、关键逻辑的代码混淆和隐藏[85,86].例如,在程序运行过程中,通过服务器推送的特定数据输入,动态生成所需要的功能逻辑,实现关键功能执行.这种方式的一个关键是在正常程序中构建一个风险可控的“漏洞”,使得程序控制流有机会被隐藏的功能逻辑使用.因此,研究如何利用基于二进制代码重用的程序执行方式进行代码逻辑混淆、代码隐写等应用,是一项可探索的工作.

随着防御技术的不断演进,代码重用攻击不局限于编译、发布的程序,可能更加倾向于动态生成代码的重

用.目前,针对 JIT 等方式动态生成的二进制代码的防护相对薄弱,这将成为攻击者的新的攻击面.因此,针对动态生成代码的攻击防护,也就是代码重用攻击对抗中的一个新的战场.

此外,从控制流完整性和随机化的角度研究代码重用攻击防御方法仍将继续.在控制流完整性防护方面,一方面是完善已有的防御方法,如 CET,CFG,使其更加完备;另一方面,继续探索如何高效地实现更细粒度的 CFI.对于随机化能力增强的研究可从 3 个方面展开:(1) 引入新的随机化机制,弥补现有随机化机制中的不足;(2) 对内存中不同的数据对象进行不同程度的随机化,除了使其具备防范代码重用攻击之外,还可提升其整体安全性;(3) 提高随机化的熵,增强随机化的效果.此外,邬江兴院士提出的拟态计算机理论则是从整个计算机体系结构出发,提高运行环境或执行机构的不确定性[87].对于该理论,从攻击者的角度来看,一类与运行环境、执行程序实体关系紧密的攻击方式将可能失效.因此,研究新的计算机体系、从根源上解决代码重用等众多内存漏洞攻击,也是可供探索的方向.

典型代码重用攻击需要配合内存信息泄漏,而这其中的信息泄露既包括程序漏洞引发的内存任意读缺陷,也包括由软件、硬件设计缺陷导致的侧信道信息泄露.因此,如何从硬件、系统、软件等层面有效防止程序运行信息泄漏、防止内存中敏感数据泄漏,是近期研究的一个方向.在数据中心、云环境中,针对硬件特性引发的侧信道信息泄露的研究也值得关注.

二进制代码重用在当前的计算机系统体系结构和软件生态环境下暂时无法消除,基于代码重用的攻防博弈仍将继续.因此,无论是为了在网络空间安全博弈中占据主动权,还是出于防护的目的,都有必要对二进制代码重用技术做进一步的研究.

References:

[1] Mei H, Wang QX, Zhang L, Wang J. Software analysis: A road map. Chinese Journal of Computers, 2009,32(9):1697−1710 (in Chinese with English abstract). http://cjc.ict.ac.cn/quanwenjiansuo/2009-9/mh.pdf [doi: 10.3724/SP.J.1016.2009.01697]

[2] Chen X, Gu Q, Liu SW, Liu SL, Ni C. Survey of static software defect prediction. Ruan Jian Xue Bao/Journal of Software, 2016, 27(1):1−25 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/4923.htm [doi: 10.13328/j.cnki.jos.004923]

[3] Wang T. Research on binary-executable-oriented software vulnerability detection [Ph.D. Thesis]. Beijing: Peking University, 2011 (in Chinese with English abstract). http://d.wanfangdata.com.cn/Thesis/Y2024928

[4] Zhang HG, Han WB, Lai XJ, Lin DD, Ma JF, Li JH. Survey on cyberspace security. Scientica Sinica Informationis, 2016,46(2): 125−164 (in Chinese). [doi: 10.1360/N112015-00176]

[5] Zhang HG, Han WB, Lai XJ, Lin DD, Ma JF, Li JH. Survey on cyberspace security. Science China: Information Sciences, 2015, 58(11):1−43. [doi: 10.1007/s11432-015-5433-4]

[6] Liang Y. Research on key techniques of software binary code reuse [Ph.D. Thesis]. Wuhan: Wuhan University, 2016 (in Chinese with English abstract).

[7] Spafford EH. The Internet worm program: An analysis. ACM SIGCOMM Computer Communication Review, 1989,19(1):17−57. [doi: 10.1145/ 66093.66095]

[8] Wei Q, Wei T, Wang JJ. The evolution of exploitation and exploit mitigation. Journal of Tsinghua University (Science and Technology), 2011,51(10):1274−1280 (in Chinese with English abstract). [doi: 10.16511/j.cnki.qhdxxb.2011.10.015]

[9] APPLE. iOS security guide. 2015. https://www.apple.com/business/docs/iOS_Security_Guide.pdf

[10] Miller C, Blazakis D, Daizovi D, Esser S, Lozz V, Weinmann RP. iOS Hacker's Handbook. Indianapolis: Wiley, 2012.

[11] Schwartz EJ, Avgerinos T, Brumley D. *Q*: Exploit hardening made easy. In: Proc. of the 20th USENIX Conf. on Security. Berkeley: USENIX Association, 2011. 25−25. http://dl.acm.org/citation.cfm?id=2028067.2028092

[12] Wkipedia. Return-Oriented programming. 2015. https://en.wikipedia.org/w/index.php?title=Return-oriented_programming&oldid=669727753

[13] Shacham H. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In: Proc. of the 14th ACM Conf. on Computer and Communications Security (CCS 2007). New York: ACM Press, 2007. 552−561. [doi: 10.1145/1315245.1315313]

[14] Bletsch T, Jiang X, Freeh VW, Liang ZK. Jump-Oriented programming: A new class of code-reuse attack. In: Proc. of the 6th ACM Symp. on Information, Computer and Communications Security (ASIACCS 2011). New York: ACM Press, 2011. 30–40. [doi: 10.1145/1966913.1966919]

[15] Checkoway S, Davi L, Dmitrienko A, Sadeghi AR, Shacham H, Winandy M. Return-Oriented programming without returns. In: Proc. of the 17th ACM Conf. on Computer and Communications Security (CCS 2010). New York: ACM Press, 2010. 559–572. [doi: 10.1145/1866307.1866370]

[16] Salwan J. ROPgadget. 2015. https://github.com/JonathanSalwan/ROPgadget

[17] Le L. Payload already inside: Data re-use for ROP exploits. In: Proc. of the BlackHat USA 2010. Las Vegas, 2010. https://media.blackhat.com/bh-us-10/whitepapers/Le/BlackHat-USA-2010-Le-Paper-Payload-already-inside-data-reuse-for-ROP-exploits-wp.pdf

[18] Ropshell. Free online ROP gadgets search. 2016. http://2www.ropshell.com/

[19] Chen P, Xiao X, Bing M, Li X, Shen X, Yin X. Automatic construction of jump-oriented programming shellcode (on the x86). In: Proc. of the 6th ACM Symp. on Information, Computer and Communications Security (ASIACCS 2011). New York: ACM Press, 2011. 20–29. [doi: 10.1145/1966913.1966918]

[20] Davi L, Alexandra D, Sadeghi AR, Winandy M. Return-Oriented programming without returns on ARM. Bochum: Ruhr University Bochum, 2010. http://www.trust.informatik.tu-darmstadt.de/fileadmin/user_upload/Group_TRUST/PubsPDF/ROP-without-Returns-on-ARM.pdf

[21] Kornau T. Return oriented programming for the ARM architecture [MS. Thesis]. Bochum: Ruhr-Universität Bochum, 2010. https://www.zynamics.com/downloads/kornau-tim--diplomarbeit--rop.pdf

[22] Buchanan E, Roemer R, Savage S, Shacham H. Return-Oriented programming: Exploits without code injection. In: Proc. of the BlackHat USA 2008. Las Vegas, 2008. https://www.blackhat.com/presentations/bh-usa-08/Shacham/BH_US_08_Shacham_Return_Oriented_Programming.pdf

[23] Roemer RG. Finding the bad in good code: Automated return-oriented programming exploit discovery [MS. Thesis]. San Diego: University of California, 2009. http://www.cs.ucsd.edu/~rroemer/doc/thesis.pdf

[24] Dullien T, Kornau T, Weinmann RP. A framework for automated architecture-independent gadget search. In: Proc. of the 4th USENIX Conf. on Offensive Technologies (WOOT 2010). Berkeley: USENIX Association, 2010. 1−10. http://www.cs.ucsd.edu/~rroemer/doc/thesis.pdf

[25] Flanagan C, Saxe JB. Avoiding exponential explosion: Generating compact verification conditions. In: Proc. of the 28th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages. New York: ACM Press, 2001. 193–205. [doi: 10.1145/360204.360220]

[26] Solar D. Bugtraq: Getting around non-executable stack (and fix). 1997. http://seclists.org/bugtraq/1997/Aug/63

[27] Mcdonald J. Defeating solaris/SPARC non-executable stack protection. 1999. https://www.thc.org/root/docs/exploit_writing/sol-ne-stack.html

[28] Nergal. The advanced return-into-lib(c) exploits (PaX case study). Phrack Magazine, 2001,58(4). http://phrack.org/issues/58/4.html

[29] Krahmerk S. x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique. 2005. http://users.suse.com/~krahmer/no-nx.pdf

[30] Buchanan E, Roemer R, Shacham H, Savage S. When good instructions go bad: Generalizing return-oriented programming to RISC. In: Proc. of the 15th ACM Conf. on Computer and Communications Security (CCS 2008). New York: ACM Press, 2008. 27–38. [doi: 10.1145/1455770.1455776]

[31] Qian Y. ROP attack and defense technology based on ARM [MS. Thesis]. Shanghai: Shanghai Jiaotong University, 2012 (in Chinese with English abstract). http://cdmd.cnki.com.cn/Article/CDMD-10248-1013022062.htm

[32] Xing T, Chen P, Ding WB. BIOP: Automatic construction of enhanced ROP attack. Chinese Journal of Computers, 2014,37(5): 1111−1123 (in Chinese with English abstract). http://d.wanfangdata.com.cn/Periodical/jsjxb201405012 [doi: 10.3724/SP.J.1016.2014.01111]

[33] Snow KZ, Monrose F, Davi L, Dmitrienko A, Liebchen C, Sadeghi A. Just-in-Time code reuse: On the effectiveness of fine-grained address space layout randomization. In: Proc. of the 2013 IEEE Symp. on Security and Privacy (SP). 2013. 574–588. [doi: 10.1109/SP.2013.45]

[34] Snow KZ, Davi L. Just in time code reuse. In: Proc. of the Blackhat USA 2013. Las Vegas, 2013. http://media.blackhat.com/us-13/US-13-Snow-Just-In-Time-Code-Reuse-Slides.pdf

[35] Bittau A, Belay A, Mashtizadeh A, Mazières D, Boneh D. Hacking blind. In: Proc. of the 2014 IEEE Symp. on Security and Privacy (SP). 2014. 227−42. [doi: 10.1109/SP.2014.22]

[36] Athanasakis M, Elias A, Michalis P, Georgios P, Sotiris I. The devil is in the constants: Bypassing defenses in browser JIT engines. In: Proc. of the 2015 Network and Distributed System Security Symp. (NDSS 2015). 2015. [doi: 10.14722/ndss.2015.23209]

[37] Qian Y, Wang YJ, Xue Z. ROP attack and defense technology based on ARM. Information Security and Communications Privacy, 2012,(10):75−77 (in Chinese with English abstract). [doi: 10.3969/j.issn.1009-8054.2012.10.036]

[38] Xia Y, Liu Y, Chen H, Zang B. CFIMon: Detecting violation of control flow integrity using performance counters. In: Proc. of the IEEE/IFIP Int'l Conf. on Dependable Systems and Networks (DSN 2012). Boston: IEEE, 2012. 1−12. [doi: 10.1109/DSN.2012.6263958]

[39] Carlini N, Wagner D. ROP is still dangerous: Breaking modern defenses. In: Proc. of the 23rd USENIX Security Symp. (USENIX Security 2014). San Diego: USENIX Association, 2014. 385−399. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/carlini

[40] Carlini N, Barresi A, Mayer M, Wagner D, Gross TR. Control-Flow bending: On the effectiveness of control-flow integrity. In: Proc. of the 24th USENIX Security Symp. (USENIX Security 2015). Washington: USENIX Association, 2015. 161−176. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/carlini

[41] Cheng YQ, Zhou ZW, Yu M, Ding XH, Deng RH. ROPecker: A generic and practical approach for defending against ROP attacks. In: Proc. of the 2014 Network and Distributed System Security Symp. (NDSS 2014). 2014. [doi: 10.14722/ndss.2014.23156]

[42] Qiao R, Zhang MW, Sekar R. A principled approach for ROP defense. In: Proc. of the 31st Annual Computer Security Applications Conf. (ACSAC 2015). New York: ACM Press, 2015. 101−110. [doi: 10.1145/2818000.2818021]

[43] Szekeres, L, Payer M, Wei T, Song D. SoK: Eternal war in memory. In: Proc. of the 2013 IEEE Symp. on Security and Privacy (SP). San Francisco: IEEE Computer Society, 2013. 48−62. [doi: 10.1109/SP.2013.13]

[44] Schuster F, Tendyck T, Liebchen C, Davi L, Sadeghi AR, Holz T. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications. In: Proc. of the 2015 IEEE Symp. on Security and Privacy (SP). 2015. 745−62. [doi: 10.1109/SP.2015.51]

[45] Liang Y, Peng GJ, Luo Y, Zhang HG. Mitigating ROP attacks via ARM-specific in-place instruction randomization. China Communications, 2016,13(9):208−826. [doi: 10.1109/CC.2016.7582313]

[46] Abadi M, Budiu M, Erlingsson Ú, Ligatti J. Control-Flow integrity. In: Proc. of the 12th ACM Conf. on Computer and Communications Security (CCS 2005). New York: ACM Press, 2005. 340−353. [doi: 10.1145/1102120.1102165]

[47] Abadi M, Budiu M, Erlingsson Ú, Ligatti J. Control-Flow integrity principles, implementations, and applications. ACM Trans. on Information and System Security, 2009,13(1):4:1−4:40. [doi: 10.1145/1609956.1609960]

[48] Wu CG, Li JJ. The evolution of control flow integrity. 2016 (in Chinese). http://www.inforsec.org/wp/?p=495

[49] Abadi M, Budiu M, Erlingsson Ú, Ligatti J. A theory of secure control flow. In: Lau KK, Banach R, eds. Proc. of 7th Int'l Conf. on Formal Engineering Methods (ICFEM 2005). Berlin, Heidelberg: Springer-Verlag, 2005. 11−24. [doi: 10.1007/11576280_9]

[50] Arden O, George MD, Liu J, Vikram K, Askarov A, Myers AC. Sharing mobile code securely with information flow control. In: Proc. of the 2012 IEEE Symp. on Security and Privacy (SP). 2012. 191−205. [doi: 10.1109/SP.2012.22]

[51] Zhang C, Wei T, Chen ZF, Duan L, Szekeres L, McCamant S, Song D, Zou W. Practical control flow integrity and randomization for binary executables. In: Proc. of the 2013 IEEE Symp. on Security and Privacy (SP). 2013. 559−73. [doi: 10.1109/SP.2013.44]

[52] Zhang M, Sekar R. Control flow integrity for COTS binaries. In: Proc. of the 22nd USENIX Security Symp. (USENIX Security 2013). Washington: USENIX Association, 2013. 337−352. [doi: 10.1145/2818000.2818016]

[53] Goktas E, Athanasopoulos E, Bos H, Portokalidis G. Out of control: Overcoming control-flow integrity. In: Proc. of the 2014 IEEE Symp. on Security and Privacy (SP). 2014. 575−89. [doi: 10.1109/SP.2014.43]

[54] Niu B, Tan G. Modular control-flow integrity. In: Proc. of the 35th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI 2014). New York: ACM Press, 2014. 577−587. [doi: 10.1145/2594291.2594295]

[55] Niu B, Tan G. RockJIT: Securing just-in-time compilation using modular control-flow integrity. In: Proc. of the 2014 ACM SIGSAC Conf. on Computer and Communications Security (CCS 2014). New York: ACM Press, 2014. 1317−28. [doi: 10.1145/2660267.2660281]

[56] Mohan V, Larsen P, Brunthaler S, Hamlen K, Franz M. Opaque control-flow integrity. In: Proc. of the 2015 Network and Distributed System Security Symp. (NDSS 2015). 2015. [doi: 10.14722/ndss.2015.23271]

[57] Davi L, Koeberl P, Sadeghi AR. Hardware-Assisted fine-grained control-flow integrity: Towards efficient protection of embedded systems against software exploitation. In: Proc. of the 51st Annual Design Automation Conf. (DAC 2014). New York: ACM Press, 2014. 133:1−133:6. [doi: 10.1145/2593069.2596656]

[58] Pewny J, Holz T. Control-Flow restrictor: Compiler-based CFI for iOS. In: Proc. of the 29th Annual Computer Security Applications Conf. (ACSAC 2013). New York: ACM Press, 2013. 309−318. [doi: 10.1145/2523649.2523674]

[59] Chiueh TC, Hsu FH. RAD: A compile-time solution to buffer overflow attacks. In: Proc. of the 21st Int'l Conf. on Distributed Computing Systems. Mesa: IEEE, 2001. 409−417. [doi: 10.1109/ICDSC.2001.918971]

[60] Davi L, Sadeghi AR, Winandy M. ROPdefender: A detection tool to defend against return-oriented programming attacks. In: Proc. of the 6th ACM Symp. on Information, Computer and Communications Security (ASIACCS 2011). New York: ACM Press, 2011. 40−51. [doi: 10.1145/1966913.1966920]

[61] Frantzen M, Shuey M. StackGhost: Hardware facilitated stack protection—Vol.10. In: Proc. of the 10th Conf. on USENIX Security Symp. (SSYM 2001). Berkeley: USENIX Association, 2001. 5−5. http://dl.acm.org/citation.cfm?id=1267612.1267617

[62] Sinnadurai S, Zhao Q, Wong W. Transparent runtime shadow stack: Protection against malicious return address modifications. 2014. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.120.5702

[63] Iintel. Pin—A dynamic binary instrumentation tool. https://software.intel.com/en-us/articles/pintool

[64] PAX team. Address space layout raondomization (ASLR). 2001. https://pax.grsecurity.net/docs/aslr.txt

[65] Ubuntu. Ubuntu security features—Address space layout randomisation (ASLR). Ubuntu Wiki: 2016, https://wiki.ubuntu.com/Security/Features#Address_Space_Layout_Randomisation_.28ASLR.29

[66] Schulz P. Android security analysis challenge: Tampering dalvik bytecode during runtime. Bluebox Security, 2013, https://bluebox.com/android-security-analysis-challenge-tampering-dalvik-bytecode-during-runtime/

[67] Liang Y, Ma X, Wu D, Tang X, Gao D, Peng G, Jia C, Zhang H. Stack layout randomization with minimal rewriting of Android binaries. In: Kwon S, Yun A, eds. Proc. of the Information Security and Cryptology (ICISC 2015). LNCS 9558, Cham: Springer Int'l Publishing, 2016. 229−45. [doi: 10.1007/978-3-319-30840-1_15]

[68] Lee B, Lu L, Wang T, Kim T, Lee W. From zygote to morula: Fortifying weakened ASLR on Android. In: Proc. of the 2014 IEEE Symp. on Security and Privacy (SP). 2014. 424−439. [doi: 10.1109/SP.2014.34]

[69] Hiser J, Nguyen-Tuong A, Co M, Hall M, Davidson JW. ILR: Where'd my gadgets go? In: Proc. of the 2012 IEEE Symp. on Security and Privacy (SP). 2012. 571−85. [doi: 10.1109/SP.2012.39]

[70] Wartell R, Mohan V, Hamlen KW, Lin Z. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In: Proc. of the 2012 ACM Conf. on Computer and Communications Security (CCS 2012). New York: ACM Press, 2012. 157−168. [doi: 10.1145/2382196.2382216]

[71] Pappas V, Polychronakis M, Keromytis AD. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In: Proc. of the 2012 IEEE Symp. on Security and Privacy (SP). 2012. 601−615. [doi: 10.1109/SP.2012.41]

[72] Chen Y. A survey of address space layout randomization (ASLR) enforcement. 2016 (in Chinese). http://www.inforsec.org/wp/?p=1009

[73] Backes M, Nürnberger S. Oxymoron: Making fine-grained memory randomization practical by allowing code sharing. In: Proc. of the 23rd USENIX Security Symp. (USENIX Security 14). San Diego: USENIX Association, 2014. 433−447. https://www.usenix.org/node/184466

[74] Davi L, Christopher L, Sadeghi AR, Snow KZ, Monrose F. Isomeron: Code randomization resilient to (just-in-time) return-oriented programming. In: Proc. of the 2015 Network and Distributed System Security Symp. (NDSS 2015). 2015. [doi: 10.14722/ndss.2015.23262]

[75] Lu K, Nurnberger S, Backes M, Lee W. How to make ASLR win the clone wars: Runtime re-randomization. In: Proc. of the 2016 Network and Distributed System Security Symp. (NDSS 2016). 2016. http://www.cc.gatech.edu/~klu38/publications/runtimeaslr-ndss16.pdf

[76] Backes M, Holz T, Kollenda B, Koppe P, Nürnberger S, Pewny J. You can run but you can't read: Preventing disclosure exploits in executable code. In: Proc. of the 2014 ACM SIGSAC Conf. on Computer and Communications Security (CCS 2014). New York: ACM Press, 2014. 1342−1353. [doi: 10.1145/2660267.2660378]

[77] 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 2015 IEEE Symp. on Security and Privacy (SP). 2015. 763−80. [doi: 10.1109/SP.2015.52]

[78] Pappas V, Polychronakis M, Keromytis AD. Transparent ROP exploit mitigation using indirect branch tracing. In: Proc. of the 22nd USENIX Security Symp. (USENIX Security 2013). Berkeley: USENIX Association, 2013. 447−462. http://dl.acm.org/citation.cfm?id=2534766.2534805

[79] Göktaş E, Athanasopoulos E, Polychronakis M, Bos H, Portokalidis G. Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard. In: Proc. of the 23rd USENIX Security Symp. (USENIX Security 2014). San Diego: USENIX Association, 2014. 417−432. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/goktas

[80] Liang Y, Fu J, Peng G, Peng B. S-Tracker: Attribution of shellcode exploiting stack. Journal of Huazhong University of Science and Technology (Natural Science Edition), 2014,42(11):39−46 (in Chinese with English abstract). [doi: 10.13245/j.hust.141108]

[81] Prakash A, Yin H. Defeating ROP through denial of stack pivot. In: Proc. of the 31st Annual Computer Security Applications Conf. (ACSAC 2015). New York: ACM Press, 2015. 111−120. [doi: 10.1145/2818000.2818023]

[82] Tang J. Exploring control flow guard in Windows 10. 2015. http://documents.trendmicro.com/assets/wp/exploring-control-flow-guard-in-windows10.pdf

[83] Tencent PC Manager. Security features of Windows 10: The enforcement of exection flow. 2015 (in Chinese). http://www.freebuf.com/articles/security-management/58373.html

[84] Dang T, Maniatis P, Wagner D. The performance cost of shadow stacks and stack canaries. In: Proc. of the 10th ACM Symp. on Information, Computer and Communications Security (ASIA CCS 2015). New York: ACM Press, 2015. 555−566. [doi: 10.1145/2714576.2714635]

[85] Lu K, Xiong S, Gao D. RopSteg: Program steganography with return oriented programming. In: Proc. of the 4th ACM Conf. on Data and Application Security and Privacy (CODASPY 2014). New York: ACM Press, 2014. 265−272. [doi: 10.1145/2557547.2557572]

[86] Tang X, Liang Y, Ma X, Lin Y, Gao D. On the effectiveness of code-reuse based Android application obfuscation. In: Hong S, Park J, eds. Proc. of the Information Security and Cryptology (ICISC 2016). LNCS 10157, Cham: Springer-Verlag, 2017. 333−349. [doi: 10.1007/978-3-319-53177-9_18]

[87] Wu J. Mimic defense in cyberspace. Secrecy Science and Technology, 2014,(10):4−9 (in Chinese with English abstract). http://www.cnki.com.cn/Article/CJFDTotal-BMKJ201410001.htm

附中文参考文献:

[1] 梅宏,王千祥,张路,王戟.软件分析技术进展.计算机学报,2009,32(9):1697−1710. http://cjc.ict.ac.cn/quanwenjiansuo/2009-9/mh.pdf [doi: 10.3724/SP.J.1016.2009.01697]

[2] 陈翔,顾庆,刘望舒,刘树龙,倪超.静态软件缺陷预测方法研究.软件学报,2016,27(1):1−25. http://www.jos.org.cn/1000-9825/4923.htm [doi: 10.13328/j.cnki.jos.004923]

[3] 王铁磊.面向二进制程序的漏洞挖掘关键技术研究[博士学位论文].北京:北京大学,2011. http://d.wanfangdata.com.cn/Thesis/Y2024928

[4] 张焕国,韩文报,来学嘉,林东岱,马建峰,李建华.网络空间安全综述.中国科学:信息科学,2016,46(2):125−164. [doi: 10.1360/N112015-00176]

[6] 梁玉.软件二进制代码重用关键技术研究[博士学位论文].武汉:武汉大学,2016.

[8] 魏强,韦韬,王嘉捷.软件漏洞利用缓解及其对抗技术演化.清华大学学报:自然科学版,2011,51(10):1274−1280. [doi: 10.16511/j.cnki.qhdxxb.2011.10.015]

[31] 钱逸.基于 ARM 架构的 ROP 攻击与防御技术研究[硕士学位论文].上海:上海交通大学,2012. http://cdmd.cnki.com.cn/Article/CDMD-10248-1013022062.htm

[32] 邢骁,陈平,丁文彪,茅兵,谢立.BIOP:自动构造增强型 ROP 攻击.计算机学报,2014,37(5):1111−1123. http://d.wanfangdata.com.cn/Periodical/jsjxb201405012 [doi: 10.3724/SP.J.1016.2014.01111]

[37] 钱逸,王轶骏,薛质.基于 ARM 平台的 ROP 攻击及防御技术.信息安全与通信保密,2012,(10):75−77. [doi: 10.3969/j.issn.1009-8054.2012.10.036]

[48] 武成岗,李建军.控制流完整性的发展历程.2016. http://www.inforsec.org/wp/?p=495

[72] Chen Y.地址空间布局随机化(ASLR)增强研究综述.2016. http://www.inforsec.org/wp/?p=1009

[80] 梁玉,傅建明,彭国军,等.S-Tracker:基于栈异常的 shellcode 检测方法.华中科技大学学报(自然科学版),2014,42(11):39−46. [doi: 10.13245/j.hust.141108]

[83] 腾讯电脑管家.Win10 安全特性之执行流保护.2015. http://www.freebuf.com/articles/security-management/58373.html

[87] 邬江兴.网络空间拟态安全防御.保密科学技术,2014,(10):4−9. http://www.cnki.com.cn/Article/CJFDTotal-BMKJ201410001.htm

彭国军(1979－),男,湖北荆州人,博士,教授,CCF 专业会员,主要研究领域为恶意代码检测,可信软件.

张焕国(1945－),男,教授,博士生导师,CCF 高级会员,主要研究领域为信息安全,可信计算,密码学.

梁玉(1988－),男,博士,主要研究领域为系统安全,网络安全.

傅建明(1969－),男,博士,教授,博士生导师,CCF 高级会员,主要研究领域为系统安全,网络安全.