软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2019,30(9):2772−2790 [doi: 10.13328/j.cnki.jos.005516]

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

运行时代码随机化防御代码复用攻击*

张贵民[1,2], 李清宝[1,2], 曾光裕[1,2], 赵宇韬[1,2]

[1](解放军信息工程大学,河南 郑州 450001)

[2](数学工程与先进计算国家重点实验室,河南 郑州 450001)

通讯作者: 张贵民, E-mail: zh.guimin@163.com

摘 要: 代码复用攻击日趋复杂,传统的代码随机化方法已无法提供足够的防护.为此,提出一种基于运行时代码随机化的代码复用攻击防御方法 LCR.该方法在目标程序正常运行时,实时监控攻击者企图获取或利用 gadgets 的行为,当发现监控的行为发生时,立即触发对代码进行函数块级的随机化变换,使攻击者最终获取或利用的 gadgets 信息失效,从而阻止代码复用攻击的实现.设计实现了LCR原型系统,并对提出的方法进行了测试.结果表明: LCR能够有效防御基于直接或间接内存泄漏等实现的代码复用攻击,且在 SPEC CPU2006 上的平均开销低于 5%.

关键词: 代码随机化;代码复用攻击;内存泄漏

中图法分类号: TP309

中文引用格式: 张贵民,李清宝,曾光裕,赵宇韬.运行时代码随机化防御代码复用攻击.软件学报,2019,30(9):2772−2790. http://www.jos.org.cn/1000-9825/5516.htm

英文引用格式: Zhang GM, Li QB, Zeng GY, Zhao YT. Defensing code reuse attacks using live code randomization. Ruan Jian Xue Bao/Journal of Software, 2019,30(9):2772−2790 (in Chinese). http://www.jos.org.cn/1000-9825/5516.htm

Defensing Code Reuse Attacks Using Live Code Randomization

ZHANG Gui-Min[1,2], LI Qing-Bao[1,2], ZENG Guang-Yu[1,2], ZHAO Yu-Tao[1,2]

[1](PLA Information Engineering University, Zhengzhou 450001, China)

[2](State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou 450001, China)

Abstract: As code reuse attacks (CRA) are becoming increasingly complex, legacy code randomization methods have been unable to provide adequate protection. An approach called LCR is present to defense CRA by living code randomization. LCR real-time monitors all suspicious operations which aim to find or utilize gadgets. When above events occur, LCR randomizes the function blocks of the target process in the memory so that gadgets' information known by attackers become invalid and attacks composed of these gadgets will fail. Finally, a prototype system of LCR is implemented to test the proposed method. Experiment results show that LCR can effectively defense CRAs based on direct or indirect memory disclosure, meanwhile introduces low run-time performance overhead on SPEC CPU2006 with less than 5% on average.

Key words: code randomization; code reuse attack; memory disclosure

代码复用攻击(code reuse attack,简称 CRA)已成为当前一种广泛应用的主流攻击.CRA 首先在内存中的已有代码中寻找可用的指令序列(即 gadgets),然后劫持控制流,使这些 gadgets 得到执行,从而实现攻击.由于并不植入任何代码,因此能够轻易绕过数据执行阻止技术(data executive prevention,简称 DEP)[1].

地址空间布局随机化技术(address space layout randomization,简称 ASLR)[2]是防御代码复用攻击的一类重

* 基金项目: 国家社会科学基金(15AJG012); 国家"核高基"科技重大专项(2013JH00103)

Foundation item: National Social Science Foundation of China (15AJG012); National Science and Technology Major Project of China (2013JH00103)

收稿时间: 2016-09-14; 修改时间: 2016-11-25, 2017-02-08, 2017-06-06; 采用时间: 2017-08-24

要技术,已在当前的大部分操作系统中得到应用.ASLR 通过对代码段和数据段的基地址的随机化,使攻击者无法定位相关指令序列.但之后提出的暴力破解攻击[3]和基于内存泄漏漏洞[4]实现的攻击,成功突破了 ASLR 防护技术.为进一步提高 ASLR 的防御能力,更细粒度的随机化技术[5−8]被不断提出,通过进一步增加代码或指令的随机性,阻止攻击者获得 gadgets 信息.但随后,Snow 等人提出了即使在细粒度 ASLR 下也能实现的 JIT-ROP 攻击[9].JIT-ROP 攻击的成功在于现有 ASLR 普遍采用程序运行前的一次代码随机化,而运行过程中代码在内存中的位置则相对固定,攻击者可借助内存泄漏漏洞动态地分析内存中的代码以获取 gadgets 的信息.

为更好地防御复杂先进的代码复用攻击,如 JIT-ROP 攻击等,本文提出一种基于运行时代码随机化的代码复用攻击防御方法 LCR.其基本原理是:允许攻击者获取内存信息,但在攻击者利用这些信息实施攻击之前使这些信息失效,从而导致攻击失败.该方法监控对目标程序代码段的读数据操作和该程序执行的输入类系统调用,当发生上述操作时,则对程序代码中各函数块在内存中的位置进行随机变换,使攻击者通过直接内存泄漏或间接内存泄漏[10]获得的内存信息无效,从而阻止攻击者根据泄漏的信息实现代码复用攻击.最后,本文借助硬件虚拟化技术 Intel VT[11]设计实现了 LCR 的原型系统,并对该方法的有效性和性能开销进行了测试,结果表明:LCR 能够有效防御包含 JIT-ROP 在内的大多数代码复用攻击,且在 SPEC CPU2006 上的平均开销低于 5%.

本文第 1 节描述 LCR 的威胁模型和假设,明确防御对象和实施条件.第 2 节简要介绍 LCR 的总体架构.第 3 节论述 LCR 的具体设计和实现方法.第 4 节实现 LCR 原型系统并对其进行测试.第 5 节探讨 LCR 的局限性.第 6 节对比分析 LCR 和其他相关的研究工作.第 7 节对全文工作进行总结.

1 威胁模型和假设

LCR 旨在保护目标程序免遭代码复用攻击,即使是如 JIT-ROP 这样功能强大的攻击也无法在有 LCR 保护的情况下得以实现.LCR 的威胁模型和假设为:

(1) 目标程序中至少含有一个内存损坏漏洞,攻击者能够借助该漏洞实现对目标程序内存的任意读写以及对目标程序控制流的劫持;
(2) 攻击者可以获取和分析目标程序的二进制文件;
(3) 操作系统已开启 ASLR 和 DEP 机制,攻击者不能通过植入代码实现攻击;
(4) 系统的硬件设备是可信的;
(5) 只考虑针对应用程序的代码复用攻击,针对内核层和虚拟层的攻击不在考虑范围内.

本文与其他相关代码复用攻击的防御方法[12,13]所采用的攻击模型和假设基本一致.

2 总体架构

LCR 的目的是使攻击者通过内存泄漏等方式得到的关于程序代码的内存信息失效,而利用这些失效的信息,攻击者无法成功发动代码复用攻击.在目标程序运行过程中,LCR 动态地对目标程序代码的内存布局进行随机化变换,其总体架构如图 1 所示.

LCR 总体分为 3 个阶段.

- 一是预处理阶段:重新编译目标程序,获取与随机化处理相关的程序代码信息;
- 二是运行时监控阶段:在发现对程序代码所在内存页的读数据操作和执行输入类操作时,触发对目标程序代码的随机化处理,实现对代码复用攻击的有效防御;
- 三是运行时代码随机化处理阶段:采用对目标程序代码段中各函数块所处内存位置重排序的方式,打乱原有代码中所有指令的内存地址,使攻击者在变化前获取的内存分布信息失效,当攻击者利用这些失效的信息发动代码复用攻击时,导致攻击失败;同时,该阶段还要完成对所有相关指针、代码等内容的更新操作,以保证变换后程序仍能正常运行.

本文借助 Intel VT 硬件虚拟化技术[11]对 LCR 进行了设计和实现.LCR 的核心功能模块,即触发事件监控模块和随机化处理模块,均在一个轻量级 Hypervisor 中实现.

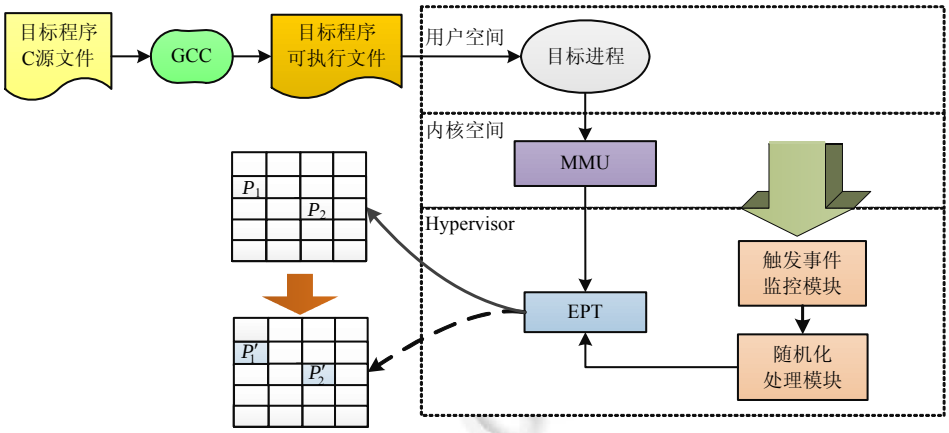


Fig.1 Overall architecture of LCR

图 1 LCR 的总体架构

3 LCR 的设计与实现

3.1 预处理

在对目标进程代码段中的函数块进行随机化处理后,函数块的绝对地址以及各函数块之间的相对位置都将改变,因此,某些指令中使用的相关地址参数和偏移量以及指向各函数的指针类型变量都需要进行更新.预处理阶段获取需要更新的指令和变量的信息,为随机化后的更新提供依据.

为尽量降低随机化后内容更新的复杂度,采用了地址无关代码(position-independent code,简称 PIC)和地址无关可执行(position-independent executable,简称 PIE)技术,在编译时增加-fpic 或-pie 选项即可.PIC/PIE 对常量和函数入口地址的操作都基于基寄存器+偏移量的相对地址的寻址方式,有利于代码随机化的实现.采用 PIC/PIE 技术在一定程度上简化了随机化变换后的处理,但仍要对目标进程的某些代码和数据进行更新操作.

3.1.1 代码段中相关内容

为便于分析,这里引入模块的概念,将一个采用动态链接程序的每个可执行文件和共享对象都看做是该程序的一个模块,而将采用静态链接的程序看做一个独立模块.

(1) 函数块内的代码引用

由于随机化处理时将函数块作为一个整体进行移动,函数块内顺序执行部分不受影响,因此主要考虑非顺序执行的内容,即分支指令,包括 CALL,JCC(如 JA,JAE,JC,JRCXZ 等),JMP 和 LOOP[11]这 4 类.分支指令只能跳转到同一个函数内部的某条指令或者另一个函数的入口,正常情况下不能跳转到其他函数内部.在这里仅考虑内部跳转,跳转到其他函数入口的情况在情形(2)中讨论.而上述指令在函数块中使用时,都通过由相对于当前指令指针值(存放在 EIP 寄存器中)的偏移值确定目的指令.由于随机化前后函数块内的指令顺序不变,因此指令间的相对偏移也不变,所以该类指令在随机化后不需要更新.

(2) 模块内的函数调用

同一模块内的函数调用通过 CALL 指令或跳转指令实现.对于 CALL 指令和 JMP 指令,当以寄存器直接或间接寻址时,由于 LCR 并不改变目标进程的执行流程,寄存器中的内容与随机化变换前一致,因此该类指令不需要修正;而当采用相对于指令指针或程序代码段基地址的偏移值定位目的指令时,则需要根据该指令的下一条指令与被调用函数在随机化变换之后的位置关系计算新的偏移值.对于 JCC 和 LOOP 指令,均采用相对于当前指令指针值的偏移,同样需要重新计算偏移值.

为了便于在随机化后执行上述操作,需提取相关指令的地址信息,同时记录目标程序代码中各个函数块在

随机化变换前后的布局信息.为此,在预处理阶段,通过分析目标程序的汇编指令获取代码段中的各个函数的名称、起始地址和长度,并构造一个原始的函数布局信息表.而在每次随机化变换时,都将生成一个新的布局信息表,并按照该表对代码进行随机化变换.通过新旧布局信息表即可计算出所需的各类数据.例如,假设某次变换时原始表和新表见表 1,那么原来跳转到 func1(0x55c)的跳转指令使用的偏移值只需更新为指令指针值与地址 0x582 的偏移值即可.

Table 1 Function blocks layout information table

表 1 函数布局信息表

函数名称	起始地址	长度(byte)	函数名称	起始地址	长度(byte)
func1	0x55c	0x26	main	0x55c	0x26
func2	0x582	0xf	func1	0x582	0x26
func3	0x591	0x34	func2	0x5a8	0xf
func4	0x5c5	0x6c	func3	0x5b7	0x34
func5	0x631	0xb2	func4	0x5eb	0x6c
main	0x6e3	0x26	func5	0x657	0xb2

注:左表为原始函数布局信息表,右表为新布局信息表

(3) 模块间的函数调用

模块间的函数调用依赖程序链接表(procedure linkage table,简称 PLT)和全局偏移表(global offset table,简称 GOT)实现.ELF 文件将 GOT 拆分成“.got”和“.got.plt”两个表,分别保存全局变量和函数的地址.下面以程序 exam.c 为例(exam.c 的源代码以及部分汇编代码、符号表及调试信息如图 2 所示),分析在 x86 架构下采用 PIC/PIE 进行编译生成共享库(共享库默认即为 PIC)和可执行代码时的函数调用机制.

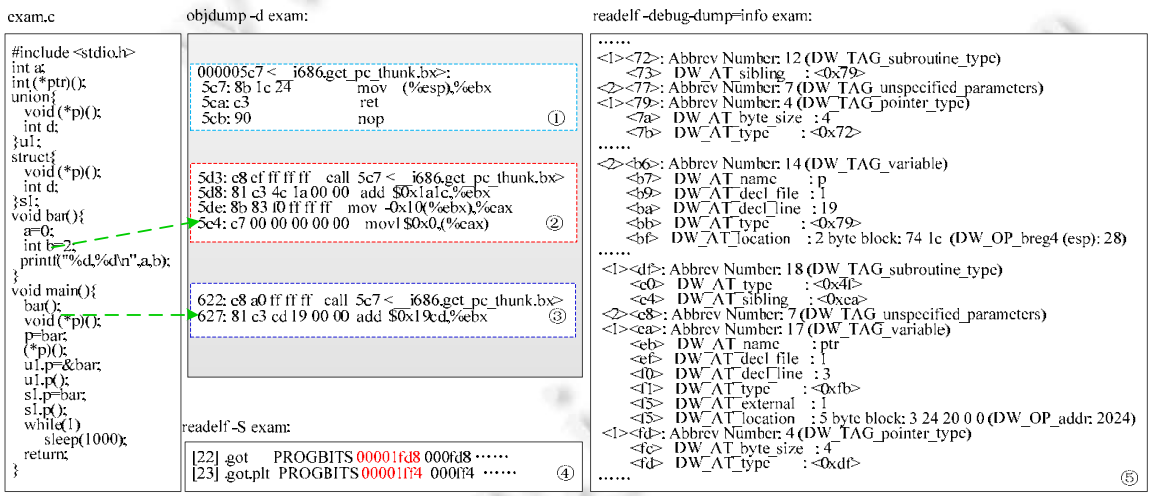


Fig.2 Relevant files for exam

图 2 程序 exam 的相关文件

由汇编代码可知:exam 调用共享库函数时,首先执行图 2③中的两条指令.由图 2①中函数__i686.get_pc_thunk.bx 的定义可知,该函数获取 0x627 处的指令在内存中的实际内存地址并存放于 ebx 中.通过符号表(如图 2④所示)可知,.got.plt 与 0x627 处指令的偏移正好为 0x19cd.因此,0x627 处指令获取.got.plt 的实际内存地址并存放在 ebx 中.程序之后对共享库中内容的调用都基于相对于.got.plt 的偏移实现.由于 LCR 并不改变数据段(含.got.plt),在随机化后该偏移值不变,因此只需重新计算调用__i686.get_pc_thunk.bx 函数的那条指令的下一条指令(本例中为 0x627)与.got.plt 地址(本例中为 0x1ff4)的相对偏移(本例中为 0x19cd),并用该值替换原参数,即可保证程序仍能够定位到.got.plt.

(4) 访问数据的指令

对数据的访问同样分为模块内和模块间两种情况:模块内的数据访问采用相对寻址方式实现,由于一个模块中的代码段和数据段的相对位置固定,因此每一条指令和模块内部数据之间的相对位置也固定,所以只需在当前指令地址的基础上加一个固定的偏移量即可访问相应数据,如图 2 中②所示对变量 *a* 的访问方式;模块间的数据访问由于数据地址要等到其所在模块装载时才能确定,故该类数据访问基于 GOT 实现.虽然两类数据访问方式在理论上有所区别,但实际实现中都是首先获取.got.plt 的地址,然后以该地址与要访问数据所在内存地址之间的偏移作为指令访问数据的偏移量.因此,在随机化后只需重新计算当前指令地址的下一条指令与.got.plt 地址的相对偏移(本例中为 0x1a1c),并修正相关指令(本例中为 0x5d8 处指令)中的偏移值即可.

(5) 指令指针寄存器(EIP)

EIP 确定下一步要执行指令的地址,在随机化变换后,需要对该值进行更新以保证后续指令正常执行.首先读取 EIP 的值,然后基于该值和函数布局信息表判断其所属的函数块以及该值和函数块入口地址的偏移值,然后根据该函数块随机化变换后的新地址加上该偏移值计算新的 EIP 值,并更新指令指针寄存器.

(6) 回调函数相关指令

通过把一个函数的指针作为参数传递给另一个函数(如 atexit,signal,sigaction 以及其他以函数指针作为参数的函数等),当该指针被用来调用其所指向的函数时,该函数就称为回调函数.当目标程序中包含回调函数时,必定存在相应指令将该函数的指针(即内存地址)作为参数传递给其他函数(通常为 movl 指令),即注册指令.在执行代码随机化后,回调函数的地址将改变,此时,注册指令的参数若不进行相应更新,当回调函数被调用时,将导致程序 Crash.因此,必须对这类指令进行更新.而根据随机化操作发生时注册指令是否已经执行,可将该类指令分为未执行和已执行两类.如图 3 所示,指令 Func1(fp3,argu1,argu2)和 Func2(fp6,argu3,argu4)分别将 fp3 和 fp6 指向的函数 F3 和 F6 注册为回调函数,且前者属于随机化操作时已执行注册指令,后者属于未执行注册指令.

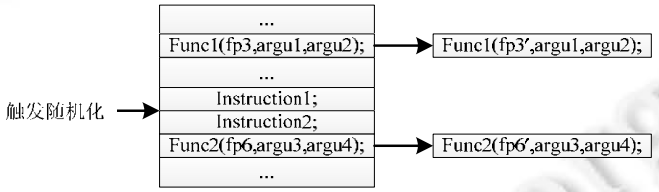


Fig.3 An example program

图 3 例子程序

对于尚未执行的注册指令,只需在随机化发生时定位到该指令,然后通过原函数布局信息表定位该指令传递的函数指针所对应的函数,再在新函数布局信息表中找到该函数变换后的新地址,并以该值替换原指令中的指针参数.假设图 3 中程序的原内存布局和随机化变换后的新内存布局分别如图 4 左上图和图 4①所示,Func2 在随机化操作后执行时,将直接按更新后 F6 的新地址 fp6′进行回调函数的注册,从而保证程序正常运行.

而对于已执行的注册指令,如图 3 中的 Func1,由于该指令已执行完毕,此时即使将其参数 fp3 更新为 F3 的新地址 fp3′也不起作用,当程序按照 fp3 回调函数 F3 时,仍将造成程序 Crash.为此,提出两种解决方法.

1) 方法 1:在随机化操作前,首先分析已经执行的注册指令及其注册的回调函数;然后,在随机化操作时将已注册函数进行定位和锁定,保证这些函数的内存地址在随机化操作后不发生改变,只对剩余函数进行随机化.其实现机制如图 4②所示:在随机化前后保证函数 F3 的内存位置不发生变化,即 fp3=fp3′;而对于尚未注册的函数 F6 以及其他函数,则进行正常的随机化.该方法可保证随机化操作不影响已注册回调函数的执行过程,避免 Crash;
2) 方法 2:首先,仍要在随机化操作前对已经注册的回调函数进行定位;然后,对所有函数(包括已注册函数)进行随机化.但在随机化过程中,向已注册回调函数的原指针指向的内存地址处写入一条 jump 指令,将已注册的回调函数的新内存地址作为 jump 的目的地址.如图 4③所示:在原注册回调函数的内存

地址 fp3 处写入 jump fp3′,从而保证已注册回调函数仍能跳转到正确的内存位置处执行.

方法 1 简单高效,但部分代码的内存位置在随机化操作后并未发生改变,这在一定程度上为攻击者留下了可乘之机;方法 2 既能解决已注册回调函数在随机化操作后导致的 Crash 问题,也保证了随机化方法原有的安全性,避免了方法 1 导致的安全性降低的问题.但方法 2 的实现还涉及多个方面的问题,包括植入 jump 指令后内存布局被打乱;随机化后相关数据的更新方式均需改变;另外,在下一次随机化时,如何对待之前植入的 jump 指令也是需要解决的问题.本文目前采用方法 1,并将在未来工作中对方法 2 进行研究和实现.

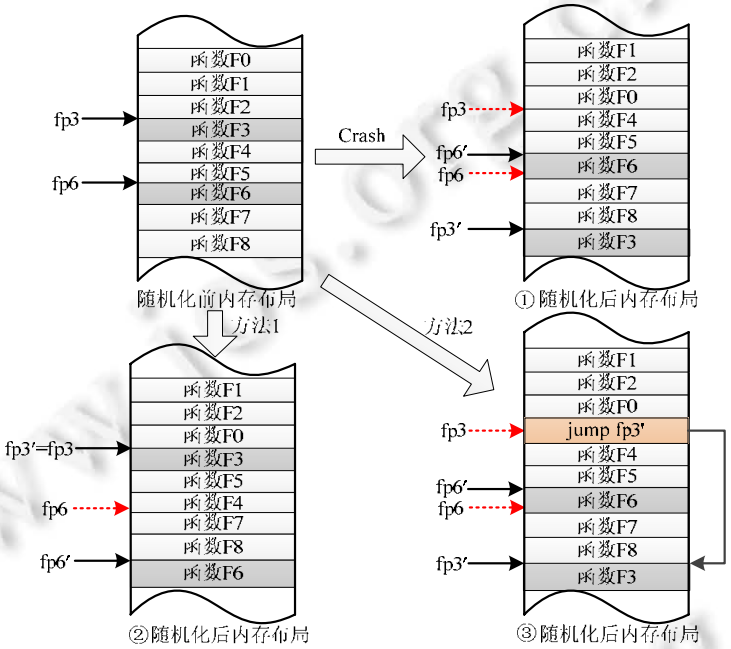


Fig.4 Method to handle callback functions

图 4 回调函数处理方法

3.1.2 数据相关内容

(1) 函数指针类型变量

LCR 改变函数的入口地址,因此所有指向这些函数的函数指针类型变量均需要进行修正.由于在 C 标准中,函数指针类型无法由其他类型转换得到,因此可以直接分析目标程序中含有的函数指针类型的全局变量和局部变量.这里,通过在编译时添加-gdwarf-version 选项(version 为 DWARF 的版本号,本文使用-gdwrf-4)生成 DWARF 格式的程序调试信息来实现该分析.程序 exam 的部分调试信息如图 2⑤所示.

DWARF 中,每条记录是一个调试信息入口(debugging information entry,简称 DIE),当某个 DW_TAG_variable 类型的 DIE,又同时属于指针类型(即 DW_TAG_pointer_type)和子过程类型(即 DW_TAG_subroutine_type)时,该 DIE 对应的变量即为函数指针类型.然后,由该变量的 DW_AT_location 属性即可获得该变量内存地址的计算方法.例如图 2⑤所示,ptr 的实际地址即程序加载地址加上 0x2020.不同的 DW_AT_location 属性对应各自的变量地址的计算方式[14],因此,通过遍历 DIE 即可找到所有函数指针类型变量及其地址的计算方法.

另外,需要特别注意的是:当存在结构体 struct 和联合体 union 类型变量时,若存在同时属于 DW_TAG_pointer_type 和 DW_TAG_subroutine_type 的成员变量(即 DW_TAG_member),该成员变量即为一个函数指针类型变量.更新结构体变量时,只需对该变量的函数指针类型的成员变量进行更新.当联合体中存在函数指针类型变量时,需要结合程序源代码和汇编代码分析定位所有使用其作为函数指针的指令及其所处的函数块,只需在这些指令执行时对联合体变量的值进行更新.

随机化变换后,首先根据函数指针类型变量的地址计算方式获得其地址,然后由原值和函数布局信息表判断出其要指向的目的函数,最后由随机化变换后该函数的新地址替换原值.

(2) 函数返回地址

函数返回地址是被调用函数执行完成后的后续指令的地址.随机化变换后,返回地址也应当更新为变化后的新地址,以保证后续指令的正常执行.返回地址保存在栈中,而且总是在前栈帧 ebp 之前入栈.因此,EBP 寄存器中的值加上返回地址的长度,即为当前函数的返回地址在栈中的位置.通过返回地址确定其对应的指令所属的函数块以及该指令到函数入口的偏移值,然后根据该函数变换后的新地址加上偏移值,即得到了新的返回地址,最后将该值写入到原返回地址所在的内存单元中.之后,读取出前栈帧 ebp,并以该值为基础更新下一个返回地址.循环上述过程,直到发现返回地址位于 main 函数时停止.此时,目标进程的函数调用栈已遍历完毕,所有的返回地址也已更新完成.

(3) 跳转表

若目标程序中含有 switch/case 语句,且当 case 语句较多时,C 编译器会自动生成一个跳转表(jump table)并存放在数据区中,通过该表进行跳转,使 switch/case 语句的执行效率得到提高.跳转表表项中存放的是 case 语句的入口地址相对于.got.plt 的偏移 offset.由于.got.plt 的地址在随机化前后不变,而 case 语句的入口地址将发生改变,因此当目标程序含有跳转表时,表项(即 offset 的值)必须进行更新.首先定位到各个 case 语句的入口地址;然后,基于该值和函数布局信息表判断其所属的函数块以及该值和函数块入口地址的偏移值;再根据该函数块随机化变换后的新地址加上该偏移值计算新的 case 语句入口地址,并基于该值与.got.plt 的偏移得出 offset 的值;最后,使用该值更新跳转表的相应表项.

3.2 运行时监控

代码复用攻击通常需经过 gadgets 获取和利用两个阶段.由于 ASLR 的广泛应用,通过直接或间接内存泄漏获取 gadgets 信息已成为最有效的 gadgets 获取方式:直接内存泄漏就是直接读取(或扫描)程序代码的内存页获取 gadgets;而间接内存泄漏则通过代码指针等数据(例如栈中的函数指针和返回地址等)推断出可执行代码的内存地址,不需要读取代码页.LCR 分别采用读后即变(change after reading,简称 CAR)和用前即变(change before using,简称 CBU)两种机制,使攻击者利用直接或间接内存泄漏无法获取到正确有效的 gadgets 信息,从而无法实施代码复用攻击.

3.2.1 CAR

CAR 机制监控攻击者对目标程序代码所在内存页的读数据操作,并触发函数代码块内存布局的随机变换,使攻击者通过直接内存泄漏的方式无法获得有效的代码内存分布信息.

该种监控通常有两种实现方式.一是监控系统中所有的读操作,判断其目的地址是否为目标程序的代码页.该方式实现简单,但开销较大;二是将目标程序的代码页设置为不可读,当读这些内存页时导致访问异常,通过异常处理实现监控.该方式只监控目标内存页的读操作,大大降低了监控开销复杂度.

CAR 借助 Intel VT 硬件虚拟化技术中提出的扩展页表技术(extended page-table,简称 EPT)[11]实现了仅对目标内存页读操作的监控.

为保证目标程序一开始执行就得到保护,CAR 通过监控 *do_execve*(·)内核函数(kernel function,简称 KF)(在 Linux 操作系统中,该函数完成具体的程序加载工作)的执行过程获知目标进程是否已加载完成.这里给出监控 *do_execve*(·)等内核函数的方法.通过分析发现,编译器会在每个函数块入口插入如下两条指令:“push %ebp; movl %esp, %ebp”,用于保存原 ebp 并修改 ebp 指针指向栈顶,且这两条指令的长度与 VMCALL 指令的长度相同(均为 3 个字节).因此在系统运行过程中,利用 Hypervisor 将这些函数入口处的头两条指令替换成 VMCALL 即可使这些函数执行时陷入 Hypervisor,从而实现监控.其中,内核函数的入口地址可通过 System.map 文件找到,最后,需在 Hypervisor 中仿真执行被覆盖的两条指令(“push %ebp; movl %esp, %ebp”)保证程序正常运行.

当监控到 *do_execve*(·)函数陷入到 Hypervisor 且由参数获知加载对象为目标程序时,保存栈中的返回地址,然后用一个非法的内存地址(假设为 Addr)替换该值.因此,当 *do_execve*(·)执行完成返回时,将由于非法内存访问

导致 EPT violation 异常再次陷入 Hypervisor.此时,标志着目标程序已加载完毕.

目标程序加载完毕后,立即将其代码页属性设置为仅可执行.首先,通过在 Hypervisor 中重构出目标进程的 task_struct 结构获取 start_code 和 end_code 的值,从而得到其对应的内存页;然后,清空并重新构建 EPT,此时将目标进程代码所在内存页的属性设置为仅可执行;最后,将 *do_execve*(·)的返回地址恢复为事先保存的正确值,保证程序仍能正常运行.

基于上述操作,目标程序一旦加载运行,其代码页即已被设置为仅可执行.因此,当执行对这些代码页的读操作时,将导致 EPT Violation 异常而陷入 Hypervisor.LCR 监控到异常后,分析导致该异常的原因,若是由读操作引起的,则修改要读取的代码页属性为可读,并将虚拟机控制结构(virtual machine control struct,简称 VMCS)中的 primary processor-based VM-execution control 字段的 monitor trap flag 位设置为 1(单步执行),之后,执行 VMRESUME 返回客户机.当客户机执行完读操作指令后,将再次陷入到 Hypervisor,此时将代码页的属性恢复为仅可执行,并同时触发一次代码随机化操作(具体实现方法将在第 3.3 节中阐述).

CAR 不阻止对目标进程代码的读操作,但通过在每次读操作后执行一次随机化,使攻击者之前读取到的内存信息失效,能够有效防御通过直接内存泄漏实现的代码复用攻击.另外,由于假设攻击者可获得目标程序的二进制文件,因此攻击者可在加载前获取相关代码信息,并在加载后直接进行篡改和利用,而不需再读取代码.针对该情况,CAR 除了由程序加载之后的读操作触发随机化外,还设定在程序加载完成后无条件触发一次随机化,使攻击者通过二进制文件获得的信息失效.但由于 CAR 机制只监控读代码操作,无法防御间接内存泄漏攻击.

CAR 是实时性的且在读后立即实施,但该机制开销并不高:首先,EPT 机制本身是基于硬件辅助实现的,因此基于该机制陷入陷出的效率远高于传统的影子页表技术;另外,正常情况下,对代码页的访问主要是取指操作,而将代码页设置为仅可执行属性时,取指操作并不会导致陷入,只有当攻击者试图构造攻击时,才有可能大量产生由读操作导致的陷入.上述分析与本文最后的性能测试结果也是一致的.

3.2.2 CBU

LCR 采用 CBU 机制的目的是实现对间接内存泄漏攻击的防御.CBU 机制通过在攻击者使用获取的内存信息发动攻击前变换原有代码内存布局的方式,使攻击者所利用的信息失效,导致攻击失败.

TASR[12]首次提出攻击实现的最小间隔理论,认为执行一个攻击操作的最短间隔即最近的输出操作和紧跟其后的输入操作之间的时间长度.如果随机化发生在每个最小间隔中间,攻击者就失去了利用已知的内存状态信息实现攻击的机会.但 TASR 并没有考虑那些不需要目标进程执行输出操作也能获得内存信息的攻击方式,如内存扫描方式,攻击者只需通过执行输入操作即可实现攻击.为了防御此类攻击,CBU 机制采用更宽松的随机化触发机制,即每次目标进程执行输入类操作都将触发一次随机化处理.另外,TASR 未考虑那些本身就含有恶意逻辑,不需要输入操作也可实现的来自恶意软件自身的攻击.而本文研究如何保护一个(非恶意的)软件不受到外部攻击,因此上述情况也不在本文考虑的范围内.

首先,通过对 32 位 linux 3.2.0-29 内核的分析得到输入类系统调用,并进一步分析获得每个系统调用实际执行的关键内核函数,其结果见表 2.通过对这些内核函数的监控,达到监控输入类操作的目的.实现方法已在第 3.2.1 节中论述.

当检测到上述内核函数陷入后,还需要判断该操作是否来自目标进程.为此,CBU 在监控到目标程序加载完成时(已在第 3.2.1 节中论述)获取该程序的基地址,即寄存器 CR3 的值,并将该值作为判断当前进程是否为目标进程的依据.若当前 CR3 的值与目标进程的基地址值相同,则表明操作的发起者为目标进程,此时触发对目标进程代码的随机化变换,否则不采取任何操作.

CBU 机制在攻击者使用获取的内存信息发动攻击前变换原有代码的内存布局,使攻击者所利用的信息失效,导致攻击失败.CBU 机制弥补了 CAR 机制无法防御间接内存泄漏的不足,而 CAR 机制又对那些并不一定需要执行输入操作的代码复用攻击(例如 JIT-ROP)提供了有效防御,两种机制共同保证了 LCR 防御代码复用攻击的能力.CBU 机制也是实时性的,它引入的性能开销与目标程序执行的输入操作的次数和频率有关,总体上比 CAR 机制引入的开销要大,尤其对于输入密集型程序,如本文第 4.2 节中测试的服务器程序 nginx,当对类似

nginx 这类输入密集型程序进行保护时,可考虑在满足一定安全性的基础上关闭 CBU 机制来换取较高的效率.

Table 2 System calls that perform input operation

表 2 输入类系统调用信息

系统调用名	系统调用号	关键内核函数
read(·)	3	vfs_read
recvfrom(·)	123	sock_recvmsg
recv(·)	98	sock_recvmsg
recvmsg(·)	184	__sys_recvmsg
*pread*64(·)	108	__sys_recvmsg
readv(·)	145	vfs_readv
msgrcv(·)	189	do_msgrcv
mq_timedreceive(·)	232	__audit_mq_sendrecv
preadv(·)	315	vfs_readv
recvmmsg(·)	319	__sys_recvmmsg

3.3 运行时代码随机化

3.3.1 代码随机化处理机制

当监控到触发事件时,Hypervisor 中的随机化处理模块负责完成对目标进程代码段中函数块布局的随机化处理.首先,基于原始函数布局信息表对代码中的函数进行重排序,构造新的函数布局信息表,并保证相邻变换中任何函数块都处于不同位置,见表 1.LCR 采用的随机化重排序算法如图 5 所示,其中,随机数的产生依赖于 Linux 内核的随机数发生器,该算法完成新函数布局表的生成、代码内存的变换以及相关内容的更新.

Algorithm 1. Code Randomization Reorder algorithm

Input: CFA(current function array), n(the number of functions),Max(blocks number when reordered)
Output: NFA(new function array)

```
int a[Max]; int number[Max]; int b[Max]; int s=0;
for i=0 to Max-1 a[i]=i;
if n<2 return;//only one function
//randomly divided CFA into Max partitions
if n>Max
      find(a, Max, n);//find Max-1 different integers belong to (0, n) randomly and store in a from small to large, a[0]=0
      //set the first function's address as the start address of every partition
else  Max=n;
for i=0 to Max-1
      temp[i].startaddress=CFA[a[i]].startaddress;
      for j=a[i] to a[i+1]-1
            temp[i].length=temp[i].length+CFA[j].length;
      number[i]=a[i+1]-a[i];//store the number of functions in every partition
reorder(a, b);//find one new sequence b randomly and ensure that all element have different locations in a and b.
//construct NFA and change code layout according to b
for i=0 to Max-1
      for j=a[b[i]] to number[b[i]]
            NFA[s].length=CFA[j].length; s++;
NFA[0].startaddress=CFA[0].startaddress;
for i=1 to n-1
      NFA[i].startaddress=NFA[i-1].startaddress+NFA[i-1].length;
move(temp[b[0]].startaddress, temp[b[0]].length, tempmemory);
for i=1 to Max-1
      move(temp[b[i]].startaddress,temp[b[i]].length, tempmemory+temp[b[i-1]].length);
move(tempmmemory, NFA[n-1].startaddress-NFA[0].startaddress+NFA[n-1].length, NFA[0].startaddress);
updatecontents(CFA,NFA);//update related contents
```

```
struct FuncLayoutInfo{
      int startaddress;
      int length;
}CFA[n],NFA[n],temp[Max];
```

Fig.5 Code randomization reorder algorithm

图 5 代码随机化重排序算法

算法中的 Max 是为了降低计算重排列的时间开销而引入的.由于在进行随机化操作时需要对 n 个函数进行全排列,并从中选择某个排列来重新布局内存,当 n 较大时,全排列的开销较大,而若先将 n 个函数划分成 Max 个部分,再对 Max 个部分进行全排列则可将时间复杂度从 $O(n!)$降低至 $O(Xax!)$.因此,Max 的引入对于提高算法性能、降低开销具有重要意义.

Max 在[2,n](n≥2)上取值,随机化算法的开销与 Max 的值为正相关关系.另外,当函数个数 n 一定时,Max 的值越小,就意味着平均每个分区中含有的函数数量越多,代码长度越长.此时,虽然每次随机化后代码的绝对地址仍然都会发生变化,但相对地址未发生变化的代码(即同一个分区内的代码)在所有代码中所占的比重将越大,即一次随机化变换后对原有内存布局的打乱程度会越小,那么就可能降低攻击者暴力破解的难度.所以在开销可以容忍的情况下,Max 值越大越好(Max 最大等于 n).但需要说明的是:Max 的大小并不影响 LCR 方法对由直接或间接内存泄露导致的代码复用攻击的防御能力,因为即使 Max 取最小值 2 时,也可实现对所有代码内存布局的随机化变换,达到防御内存泄露和代码复用攻击的效果.

构造新布局信息表后,按照该表对原代码内容进行变换.首先,按照当前函数布局信息表将目标进程代码中的各个函数依次拷贝到另一块临时内存区域.然后,再按照新函数布局信息表指定的各个函数的起始地址将临时内存区域中的函数写回到目标进程内存.由于进程代码页正常情况下不可写,为实现该操作,需将所在内存页的 EPT 属性修改为可写.当完成对所有函数块的写回后,根据新旧函数布局信息构造一个映射 f,实现从原内存地址到新内存地址的转化.如图 6 所示,利用该映射对第 3.1 节中提到的代码段中的指令参数和相关数据的内容进行更新,然后恢复代码页的不可写属性,并修改 RIP 寄存器,保证后续指令的正常执行.此时,清空并删除原函数布局信息表,而新函数布局信息表则作为下次变换的原始表.最后,由 Hypervisor 陷出到客户机继续运行目标进程.

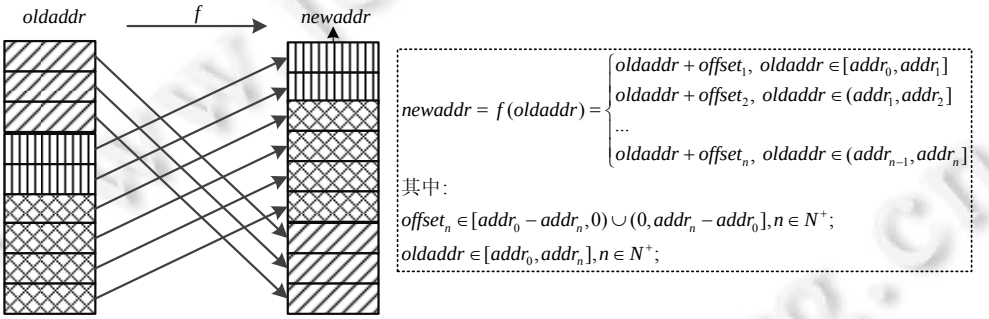


Fig.6 Mapping relationship between original and new memory layout

图 6 新旧内存布局之间的映射关系

为了更有效地防御代码复用攻击,LCR 在随机化操作时,除了.text 节外,还将.plt,.init 和.fini 节中的函数块纳入到随机化的范畴,实现了对 ELF 文件中所有可执行部分的随机化.其中,对.plt 段随机化能有效防御 return-into-libc 攻击,而.init 和.fini 节是所有可执行文件都具有的内容,其中所含的 gadgets 更容易被攻击者利用.另外,为了不让攻击者发现函数布局信息表,该表始终存放在 Hypervisor 中,且每次随机化操作完成就彻底删除原始函数布局信息表,只保留新函数布局表用于下次随机化操作.

3.3.2 安全性分析

LCR 通过对原函数块位置的变换,使攻击者之前获得的信息失效,无法以基于内存泄漏的方式实现代码复用攻击.但攻击者通过分析二进制文件,仍有可能得到目标程序的函数信息,若在此基础上实施暴力破解攻击,即枚举函数块变换后所有可能的布局方式,并分别发起攻击,其成功的可能性还是存在的.假设目标程序代码中共含 n 个函数块,攻击者尝试 K 次获得成功.另外,假设当尝试失败导致目标进程崩溃时,该程序将自动重启.

当攻击者不了解 LCR 的随机化机制(相邻两次排列各个元素的位置均不相同)时,需要尝试每一种排序,则每一次成功的概率都是 $p=\frac{1}{n!}$,那么此时 K 的期望值为

$$E(K)_1 = P + 2\times(1-P)\times P + 3\times(1-P)^2\times P + \dots + K\times(1-P)^{k-1}\times P \Rightarrow E(K)_1 = \frac{1}{P} \Rightarrow E(K)_1 = n!$$

即,攻击者平均需要尝试 $n!$ 次才能成功.当攻击者了解 LCR 的随机化机制时,攻击者只需要对那些与当前排

列相比元素的位置全部发生变化的排列进行尝试,最大化地缩小事件空间.此时,设满足条件的新的排列总数为 N,至少有一个元素在原来位置上的排列数为 N',则 $N=n!-N'$.攻击者只需在 N 个可能排列中进行尝试即可,其中,

$$N' = C_n^1 \times (n-1)! - C_n^2 \times (n-2)! + C_n^3 \times (n-3)! - \ldots + (-1)^{k+1} \times C_n^k \times (n-k)! + \ldots + (-1)^{n+1} \times C_n^n.$$

此时,每一次尝试的成功率 $P = \frac{1}{N} \Rightarrow P = \frac{1}{n! - N'}$,$K$ 的期望值为 $E(K)_2 = \frac{1}{P} \Rightarrow E(K)_2 = N$,即,此时攻击者平均需要尝试 N 次才能成功.

由上述分析可知:目标程序的函数块数目 n 越大,攻击者实现暴力破解攻击的难度越大.例如,当 n=10 时,$E(K)_1$=3628800,$E(K)_2$=1334961,而大部分应用程序的函数块数量远大于 10.可见,LCR 方法的安全性是有理论保证的.另外,以上评估均在假设攻击者尝试失败导致目标进程崩溃后进程将自动重启的条件下进行,但在实际应用中,可通过相关设置阻止进程自动重启,还可通过对程序异常崩溃情况的分析检测攻击,从而进一步提高目标程序防御代码复用攻击的能力.在这种情况下,攻击者攻击成功的可能性将更低.

4 方法评估

在 32 位 Ubuntu12.04 系统上对 LCR 进行了实现.系统内核版本为 3.2.0-29-generic-pae,其中,物理地址扩展机制(physical address extension,简称 PAE)默认开启,gcc 版本为 4.6.3,而 Hypervisor 是基于 Intel VT 技术设计的一个轻量级的虚拟机监控器,该监控器除对无条件陷入事件[11]进行处理外,只对 LCR 中涉及到的触发事件进行监控,目的是最大化地降低监控开销.计算系统采用 4 核 Intel 处理器(Core™ i7-3770,主频 3.40GHz)和 16GB 内存(RAM).

4.1 有效性测试

4.1.1 随机化有效性测试

本次测试中通过对比随机化前后内存布局的变化来验证 LCR 随机化处理的有效性.

为了更具体地体现 LCR 随机化处理过程,以程序 exam 为例进行了测试:首先,利用 ROPgedget[15]分析 exam 的二进制文件获得 gadgets 的信息列表,如图 7 所示,其中,每个 gadget 对应的 16 进制编码也已在图中列出;然后,在未实施 LCR 保护的情况下运行 exam,并假设攻击者可通过一定方式得到 exam 本次加载的虚拟地址 0xb770e000(攻击者可通过 hook 系统的程序加载过程、提取目标程序的 task_struct 结构等方式获得该信息),并通过地址转换获得了其对应的物理地址 0x3dbfdf000.测试中,由 Hypervisor 提取出 exam 可执行代码对应的内存内容(0x3dbfdf438~0x3dbfdf751,对应_init,.plt,.text 和.fini 这 4 个节),如图 8(a)所示.可见,此时攻击者可以利用图 7 中获得的地址信息从 exam 的内存空间中成功定位到 gadgets.

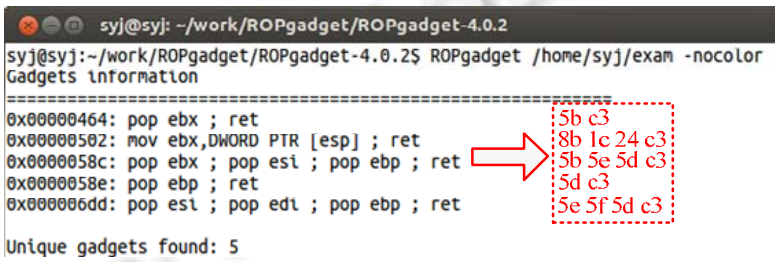
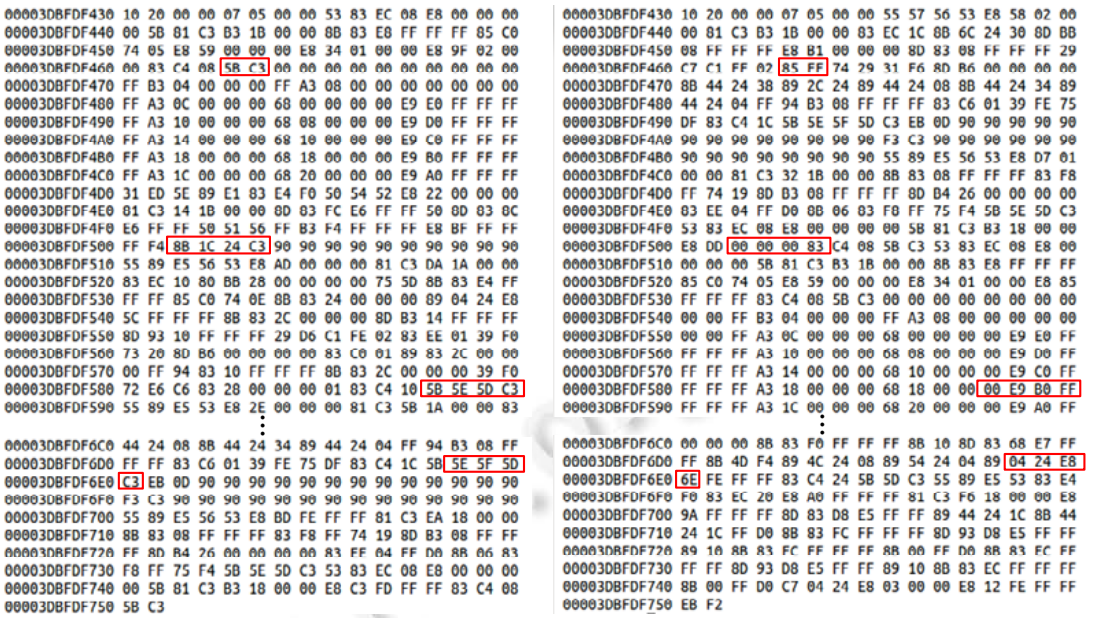


Fig.7 Running results of ROPgadgets

图 7 ROPgadgets 的运行结果



(a) (b)

Fig.8 Memory of executable code in exam

图 8 exam 中可执行代码对应的内存内容

之后,利用 LCR 对 exam 实施保护,由 Hypervisor 在每次随机化处理完成时读取一次 exam 进程的可执行代码对应的内存内容,某一次读取的结果如图 8(b)所示.为了更好地对比随机化效果,在测试中同时保留了原始函数布局信息表和新函数布局信息表,通过对比发现,本次随机化操作实现了 0x3dbfdf438~0x3dbfdf67f 和 0x3dbfdf680~0x3dbfdf751 两部分内容的位置互换.此时,当攻击者再次按照之前获取的 gadgets 地址信息去读取内存内容时,已无法得到有效的 gadgets.说明随机化处理后原有信息失效.

为使结果更具有代表性,按照上述方法对 SPEC CPU2006 中用 C 语言实现的基准程序进行了测试(400.perlbench 除外,因存在运行时翻译代码,编译时无法直接获得完整的代码信息),在编译时使用-pie 和-gdwarf-4 选项生成可执行代码.测试中,首先利用 ROPgadget 提取各基准程序中含有的 gadgets 信息,然后分别在采用和未采用 LCR 保护的情况下运行基准程序,并在运行过程中获取可执行代码的内存快照,据此分析随机化变换前后 gadgets 的变化情况.当采用 LCR 保护时,为测试 LCR 的动态随机化效果,测试中随机选择 3 个时间点获取内存快照进行分析,3 个时间点获得的内存快照中均没有定位到任何可用的 gadgets.测试结果见表 3.

Table 3 Validity tests of the randomization

表 3 随机化有效性测试

测试样例	Gadgets 数量(个)	未采用 LCR 时可定位数量(个)	采用 LCR 时的可定位数量(个)		
			T1	T2	T3
401.bzip2	13	13	0	0	0
403.gcc	69	69	0	0	0
429.mcf	10	10	0	0	0
445.gobmk	50	50	0	0	0
456.hmmer	29	29	0	0	0
458.sjeng	19	19	0	0	0
462.libquantum	16	16	0	0	0
464.h264ref	27	27	0	0	0
433.milc	19	19	0	0	0
470.lbm	7	7	0	0	0
482.sphinx3	28	28	0	0	0

通过上述测试表明:LCR 能够有效实现对目标进程内存空间中的可执行代码区域的随机化变换,攻击者通过静态分析和内存泄漏等方式无法获取有效的 gadgets 信息.

4.1.2 代码复用攻击防御能力测试

为测试 LCR 对代码复用攻击的防御能力,首先编写了含有栈缓冲区溢出漏洞的目标程序 targetProc,然后借助 ROPgadget 构造针对 targetProc 的 ROP 攻击 payload,用于生成新的 shell.在 targetProc 程序运行过程中,通过输入操作将该 ROP 攻击的 payload 注入到 targetProc 的栈中,并篡改控制流使 payload 得到执行.分别在采用和未采用 LCR 保护的情况下进行上述测试,结果见表 4.当采用 LCR 保护时,ROP 攻击失败,因为当 LCR 对该程序的可执行代码进行随机化后,注入的 payload 中所使用的 gadgets 地址信息已失效,因而此时仍然利用这些失效的地址串联执行最终导致程序崩溃,无法实现攻击目标.

Table 4 Tests results of the ROP attack

表 4 ROP 攻击测试结果

	是否开启 LCR 保护	攻击结果
测试 1	否	启动了新的 shell
测试 2	是	未启动 shelll 且程序崩溃

另外,分析和总结了当前已知的各类代码复用攻击技术.由于无法获得它们的实现代码,因此通过对它们实现机制的分析来判断 LCR 能否防御这些攻击,分析结果见表 5.上述分析都在满足本文的攻击模型和假设条件下进行.

Table 5 Analysis LCR ability to defend CRAs

表 5 LCR 代码复用攻击防御能力分析

攻击类型	攻击原理	LCR 能否防御
return-into-libc 攻击[16]	利用.plt 中的函数地址泄漏 libc 库中的其他函数地址,进而实现攻击	√
ROP 攻击[17]	利用 ret 指令串联多个 gadgets 实现攻击	√
JOP 攻击[18]	利用 jmp 等跳转指令串联 gadgets 实现攻击	√
LOP[19]	采用一个含 loop 语句的 gadgets 串联其他 gadgets 实现攻击	√
CPROP[20]	仅使用 call-preceded gadgets 构造 ROP 攻击	√
JIT-ROP 攻击[9]	动态搜索 gadgets,并通过 JIT 编译生成 payload 实现攻击	√
COOP 攻击[21]	通过修改指向虚函数表的指针实现对虚函数的复用攻击	×

4.2 性能测试

LCR 基于虚拟机监控器实现,不仅会对被保护的目标程序产生影响,也会对整个系统的性能产生影响.因此,为了充分说明 LCR 带来的各类性能开销,分别在裸机、Hypervisor、Hypervisor 下开启内核函数监控和 Hypervisor 下开启 LCR(其中,随机化重排序算法中的 Max 设置为 10)这 4 种环境下进行了测试,所有测试均使用表 4 中列出的 SPEC CPU2006 中的 C 语言基准测试程序,并且使用-pie 和-gdwarf-4 编译选项.

根据图 9 所示的测试结果可知,本文设计的轻量级虚拟机监控器 Hypervisor 在不监控内核函数时开销很小.这是因为借助 Intel VT,只需处理无条件陷入事件(该类事件执行频率很低),对不需要陷入的其他操作都采用直接穿透的方式由客户机操作系统负责完成.开启内核函数监控后,当执行表 2 中的 7 类关键内核函数时会发生陷入,开销稍有增加,但由于只在陷入后模拟执行“push %ebp; movl %esp, %ebp”两条指令,引入的开销仍然很小.启动 LCR 后,由于要在每次触发事件发生时对目标程序代码的内存进行随机化变换,在一次变换中,每一页都要经过两次读写操作,从而引入了较多运行开销,与未开启 Hypervisor 时相比,最小和最大开销分别为 0.22%(433.milc)和 18.23%(456.hmmer),而平均开销为 4.89%.可见,LCR 虽然引入了部分开销,但仍满足实用性要求[22].

为了更深入地分析 LCR 的性能开销,测试中统计了每个基准程序运行过程中监控到的内核函数的执行次数和触发 CBU 的次数,以及各基准程序可执行代码对应的内存区的总页数(start_code~end_code)和有效页的总页数(非零页).表中每个数据的分母表示监控到的执行总数,而分子表示来自于基准程序的执行次数,即触发 CBU 的次数.由于测试中未监控到 vfs_readv,do_msgrcv,__sys_recvmsg 和__audit_mq_sendrecv 这 4 个内核函数

的执行,因此未予未列出.统计结果见表 6.

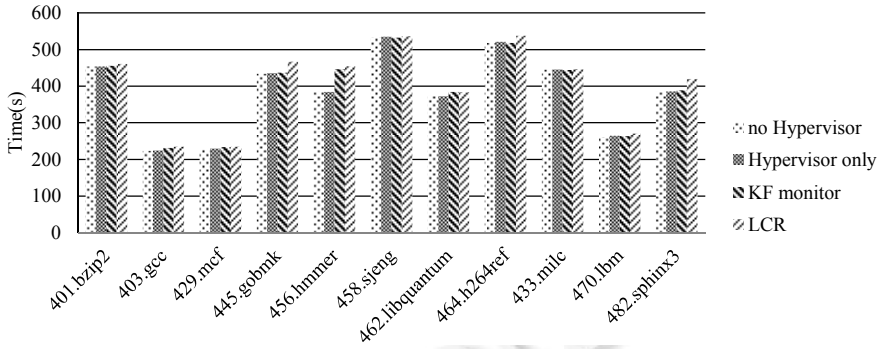


Fig.9 Running overhead for SPEC CPU2006 under four different conditions

图 9 SPEC CPU2006 在 4 种不同条件下的运行开销

Table 6 Statistics and analysis of the test data

表 6 测试数据的统计分析

测试样例	vfs_read	sock_recvmsg	__sys_recvmsg	CBU 触发次数	总页数	有效页	处理的总页数	开销(%)
401.bzip2	885/169499	4/145435	3/63065	892/377999	15	10	8920	1.55
403.gcc	68/69756	21/133512	23/49209	112/252477	801	54	6048	4.70
429.mcf	799/161922	2/133512	1/49209	802/344643	4	3	2406	3.08
445.gobmk	605/174288	0/140983	0/69027	605/384298	433	50	30250	7.60
456.hmmer	13418/274125	0/224576	0/109662	13418/608363	74	5	67090	18.23
458.sjeng	18/409893	0/346642	0/138302	18/894837	38	15	270	0.75
462.libquantum	18/364151	0/365351	0/108457	18/837959	11	9	162	2.68
464.h264ref	1284/243444	0/210696	0/90882	1284/545022	146	12	15408	3.67
433.milc	7/282105	5/244859	13/109073	25/636037	32	7	625	0.22
470.lbm	645/16488	1/20106	2/109073	648/145667	4	3	1944	3.84
482.sphinx3	3100/427957	7/402045	3/145881	3110/975883	48	44	136840	9.11

由表 6 可知,433.milc 开销小是由于其触发的 CBU 次数少,且每次随机化处理的内存页也很少.目标程序的开销与随机化过程中处理的总内存页数(即 CBU 触发次数和有效页的乘积,忽略加载时的一次随机化)的关系如图 10 所示,两者基本呈正比关系,即:当目标程序执行过程中含有越多的输入操作且可执行代码量越大时,该程序采用 LCR 保护后的开销也会越大.实际测试结果中存在个别不符合正比关系的情况,是因为每个程序运行过程中系统执行关键内核函数的总数具有一定偶然性,当该数量较大时,也会对程序的运行开销产生一定影响.

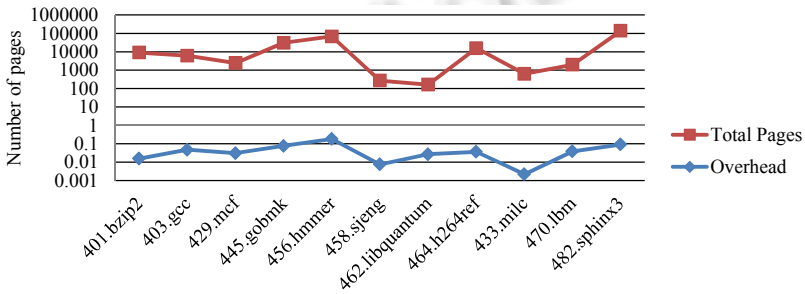


Fig.10 Relationship between running overhead and total pages which need to be handled

图 10 运行开销和处理的总页数之间的关系

由于 LCR 的 CBU 机制在输入类操作时触发代码随机化,因此,为评估 LCR 对高 I/O 程序的性能影响,采用 webbench-1.5[23]对 nginx 服务器程序在不同环境下的处理能力进行了测试.测试命令为 webbench-c 100-t 30 http://127.0.0.1/,即并发数为 100,时间为 30s,测试结果如图 11 所示.在开启对所有输入类操作的监控后(即 KF

monitor 环境下),nginx 处理能力明显下降;而开启 CAR 和 CBU 之后(即 LCR(CAR&CBU)环境下),nginx 处理能力下降 64.3%.测试发现:nginx 在测试的 30s 中,约产生了 12 次 vfs_read 操作和高达 9 363 次 sock_recvmsg 操作,共触发约 9 375 次随机化操作.正是短时间内集中触发的大量随机化操作,使 nginx 处理能力下降.

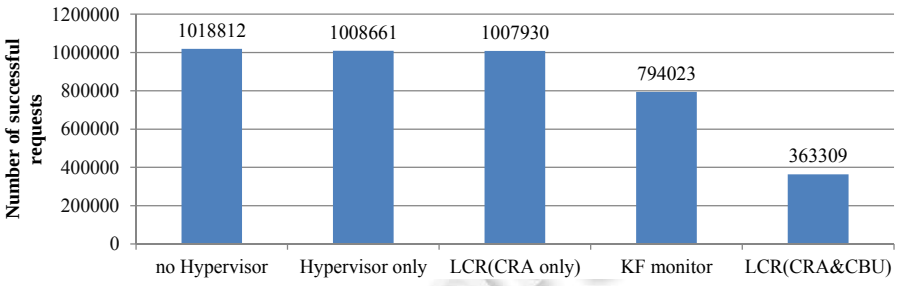


Fig.11 Processing capacity of nginx under five different conditions

图 11 nginx 在 5 种环境下的处理能力

nginx 不仅是一种高 I/O 程序,而且在工作时通常会出现 I/O 密集型操作,即,在短时间内执行大量输入/输出操作,因此,使用 LCR 对该类程序进行保护时开销会较大.在对性能要求较高的场合,存在两种可行的方案来降低 LCR 的开销.一种是通过仅对这类程序采取 CAR 保护机制,如图 11 中的 LCR(CAR only)所示,此时性能开销很小(约为 1.07%).这种通过牺牲部分安全性来换取高性能的做法也是安全领域经常采用的平衡策略.另外,由于目前大部分的代码复用攻击仍借助直接内存泄漏实现,因此,仅开启 CAR 机制仍能显著提高对代码复用攻击的防御能力;另一种是采取 CAR 和周期性随机化相结合的方式,这样既能降低部分开销,又能保证对间接内存泄漏攻击具有一定的防御能力.

5 LCR 的局限性

LCR 的局限性主要体现在以下几个方面.

(1) LCR 目前仅支持 C 语言程序.首先,C 程序是一种广泛使用的非类型安全语言,且存在较多内存破坏漏洞,对这类程序的安全防护非常具有现实意义;其次,C 语言中的函数指针无法通过其他数据类型转换得到,使对函数指针类型变量的跟踪和更新成为可能.本文提出的动态随机化代码复用攻击防御思想也可用于其他语言程序,只需解决对需要更新的函数指针等相关数据的定位和跟踪即可,涉及程序分析技术和程序动态跟踪技术的相关研究.由于本文重点在于提出和验证这样一种动态随机化的防御思想,并未研究对所有语言类型的支持;

(2) LCR 是以函数块而非 gadgets 为对象进行随机化,因此可能存在以下情况,即:函数块位置变了,但 gadgets 仍在原位置.例如,变换后,原函数 func1 的 ret 指令处变成了 func2 的 ret 指令.由于 LCR 随机化变换前后代码所在的内存空间不变,出现该类情况需满足以下条件:首先,不同函数中必须含有相同的 gadgets;另外,还要保证被该 gadgets 所在的新函数块分成两部分的原内存空间,每一部分都正好可以容下剩下的若干函数块.当含相同 gadgets 的两个函数块大小一致时,该情况出现的可能性较大;而当两者大小不一致时,则可能性很小.为应对这类情况,可通过在随机化变换时增加对原 gadgets 地址处的内容审查,若发现内容一致,则再触发一次随机化.研究发现,大多数代码复用攻击往往由多个 gadgets 配合实现.因此,除非使用的所有 gadgets 在随机化后都仍在原位置(概率很小),否则攻击仍会失败.为兼顾安全和效率,LCR 暂未对该类情况进行处理;

(3) 对于多线程程序,攻击者可能通过对子进程的分析获得相关的内存信息实现代码复用攻击.文献[24]已经对该类情况进行了详细分析,并通过在 fork 时再次随机化每个子进程的地址空间,实现了对该类攻击的防御.LCR 借助该方法实现对多线程目标程序的代码复用攻击防御;

(4) LCR仅对目标程序自身代码而未对动态链接库的代码进行随机化,因此,LCR无法防御那些不依赖读程序代码或者.plt 来获取动态链接库内存地址,且仅利用动态链接库代码实现的代码复用攻击.此类攻击可通过代码扫描(扫描动态链接库代码而非目标程序代码)或旁信道攻击[25]等方式来获取所需要的动态链接库代码地址.基于程序控制流的监控对该类攻击具有较好的防御效果,也是我们下一步要研究的重要内容.

6 相关工作

(1) 代码随机化防御方法

该类方法的核心思想是:增加获取 gadgets 的难度,使攻击者无法获得足够多且有效的 gadgets 来构造攻击.ASLR[2]最早提出了一种对动态链接库和可执行文件的加载地址随机化的方法,使攻击者无法定位 gadgets,但攻击者只需发现一个指令的地址即可获得其他所有指令的地址.为提高 ASLR 的安全性,更细粒度的随机化方法不断被提出,包括指令级随机化[5]、基本块随机化[6]、函数随机化[7]以及页面级的随机化[8].但上述方法都只在加载时执行一次随机化,攻击者仍可通过内存泄漏定位到 gadgets.与该类方法相比,LCR 采用动态运行时多次随机化,能够使攻击者通过直接或间接内存泄漏获得的信息失效,在能防御的内存泄露攻击类型和代码复用攻击的防御能力方面都得到显著提高.

JIT-ROP 攻击在程序运行过程中寻找 gadgets 加以利用,这使所有的加载时随机化方法失效.为了应对这类新型攻击,动态随机化方法被提出,LCR 就属于该类方法.Isomeron[13]同时运行程序的两个不同副本,使攻击者无法确定哪一个 gadget 会得到执行,即 Isomeron 仅提供两种可能的变换,而 LCR 是在第 3.3.2 节中分析的 $N=n!-N'$ 种可能性中选择一种进行代码的随机化.因此,相比于 Isomeron,LCR 的随机化熵值得到显著提高.文献[26]首次提出操作系统的动态随机化方法,但该方法采用周期性随机化,存在攻击窗口且实用性较差.Remix[27]动态随机变换程序每个函数体内基本块的顺序,但函数块本身位置不变,不能防御函数级复用攻击.文献[24]提出了一种通过在 fork 之后再次随机化子进程内存空间防御 clone-probing 攻击的方法,但该方法不能防御直接内存泄漏攻击.TASR[12]采用最短攻击间隔触发随机化,实现了对程序整个代码段内存位置的动态变化,但整个代码段内容不变,因此,一旦攻击者通过某种机制定位到该程序的内存,即可掌握其所有代码特征,无法防御只需一次或不需要输入操作的攻击(如 JIT-ROP 攻击);另外,TASR 的随机化模块位于用户层,易被破坏,安全性较低.相比上述动态随机化方法,LCR 首次提出了以 CAR 和 CBU 相结合的函数级动态随机化方法防御代码复用攻击,所能防御的代码复用攻击类型更多(TASR 无法防御只需一次或不需要输入操作的攻击;Remix 不能防御函数级复用攻击,文献[24]无法防御直接内存泄露攻击,文献[26]存在攻击窗口),能够提供更高的安全性.

(2) 控制流完整性保护方法

该类研究的目的是阻止攻击者篡改控制流,使 gadgets 序列得不到执行.CFI 是由 Abadi 等人.最早提出的[28],旨在基于控制流图(control-flow graph,简称 CFG)防御所有控制流劫持攻击.但该方法开销较大,实用性差.因此,目前最新的研究仍集中在如何提高该方法的实用性上.CCFIR[29],CCFI[30],PICFI[31],Context-sensitive CFI[32]和文献[33]等结合对程序语义的分析,降低对 CFG 精确度的要求,以达到降低监控开销的目的.但研究者已证明:即使细粒度的 CFI 类保护方法,也无法真正保证控制流的安全[34],控制流被劫持的风险依然存在.

(3) 其他防御方法

除了上述两类方法外,还有在编译时减少可构建 gadgets 的指令的方法,例如 G-free[35],但这类防御方法可防御的代码复用攻击类型有限;还有通过拒绝对代码的读操作[36]防御直接内存泄漏的方法,但无法防御间接内存泄漏;以及通过保护代码指针的安全性防御代码复用攻击的方法 CPI[37],但该方法已被证明是可以被绕过的[38].另外,还有通过代码和数据隔离的方法[10,39],可防御内存泄漏攻击,但实际的函数代码不变,可能遭到 return-into-libc 攻击.而 LCR 则可以有效阻止直接和间接内存泄漏攻击以及 return-into-libc 攻击.

7 总 结

本文提出了一种运行时代码随机化防御代码复用攻击的方法 LCR,并分别对预处理阶段、动态监控阶段和随机化处理阶段进行了详细论述.LCR 采用 CAR 和 CBU 两种机制,分别实现在攻击者获取 gadgets 信息后和利用 gadgets 前对目标进程代码中函数块的随机化变换,使攻击者通过直接或间接内存泄漏获得的 gadgets 信息失效,从而阻止代码复用攻击的实现.对 LCR 原型系统的测试表明:该方法能够有效实现在目标程序运行过程中对其代码内容的随机化变换,可防御大部分已知代码复用攻击,并且平均开销低于 5%,具有良好的实用性.

References:

[1] Andersen S, Abella V. Changes to functionality in Microsoft Windows XP service pack 2. Part 3: Memory Protection Technologies, Data Execution Prevention. 2015. https://technet.microsoft.com/en-us/library/bb457155.aspx

[2] PaX Team. PaX ASLR. 2003. http://pax.grsecurity.net/docs/aslr.txt

[3] Bittau A, Belay A, Mashtizadeh A, Mazieres D, Boneh D. Hacking blind. In: Proc. of the 2014 IEEE Symp. on Security and Privacy. Berkeley: IEEE CS Press, 2014. 227−242. [doi: 10.1109/SP.2014.22]

[4] Fu JM, Liu XW, Tang Y, Li PW. Survey of memory address leakage and its defense. Journal of Computer Research and Development, 2016,53(8):1829−1849 (in Chinese with English abstract). [doi: 10.7544/issn1000-1239.2016.20150526]

[5] Koo H, Polychronakis M. Juggling the gadgets: Binary-level code randomization using instruction displacement. In: Proc. of the 11th ACM on Asia Conf. on Computer and Communications Security. Xi'an: ACM Press, 2016. 23−34. [doi: 10.1145/2897845.2897863]

[6] Wartell R, Mohan V, Hamlen KW, Lin Z. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In: Proc. of the 19th ACM Conf. on Computer and Communications Security. Raleigh: ACM Press, 2012. 157−168. [doi: 10.1145/2382196.2382216]

[7] Gupta A, Habibi J, Kirkpatrick MS, Bertino E. Marlin: Mitigating code reuse attacks using code randomization. IEEE Trans. on Dependable and Secure Computing, 2015,12(3):326−337. [doi: 10.1109/TDSC.2014.2345384]

[8] Backes M. Oxymoron: Making fine-grained memory randomization practical by allowing code sharing. In: Proc. of the 23rd USENIX Conf. on Security Symp. San Diego: USENIX Association, 2014. 433−447. https://www.usenix.org/system/files/conference/usenixsecurity14/sec14-paper-backes.pdf

[9] Snow KZ, Monrose F, Davi L, Dmitrienko A. Just-in-Time code reuse: On the effectiveness of fine-grained address space layout randomization. In: Proc. of the 34th IEEE Symp. on Security and Privacy. Berkeley: IEEE CS Press, 2013. 574−588. [doi: 10.1109/SP.2013.45]

[10] 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 36th IEEE Symp. on Security and Privacy. San Jose: IEEE Press, 2015. 763−780. [doi: 10.1109/SP.2015.52]

[11] Intel. Intel 64 and IA-32 architectures software developer's manual. http://www.intel.cn/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf

[12] Bigelow D, Hobson T, Rudd R, Streilein W, Okhravi H. Timely rerandomization for mitigating memory disclosures. In: Proc. of 22nd ACM Conf. on Computer and Communications Security. Denver: ACM Press, 2015. 268−279. [doi: 10.1145/2810103.2813691]

[13] 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 22nd Annual Network and Distributed System Security Symp. San Diego: The Internet Society, 2015. [doi: 10.14722/ndss.2015.23262]

[14] DWARF4. http://dwarfstd.org/doc/DWARF4.pdf

[15] ROPgadget. https://github.com/JonathanSalwan/ROPgadget/releases

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

[17] 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. Alexandria: ACM Press, 2007. 552−561. [doi: 10.1145/1315245.1315313]

[18] Bletsch T, Jiang X, Freeh VW, Liang Z. Jump-oriented programming: A new class of code-reuse attack. In: Proc. of the 6th ACM Symp. on Information, Computer and Communications Security. Hong Kong: ACM Press, 2011. 303−307. [doi: 10.1145/1966913.1966919]

[19] Lan B, Li Y, Sun H, Su C, Liu Y, Zeng QK. Loop-oriented programming: A new code reuse attack to bypass modern defenses. In: Proc. of 14th IEEE Trustcom/BigDataSE/ISPA. Helsinki: IEEE CS Press, 2015. 190−197. [doi: 10.1109/Trustcom.2015.374]

[20] Carlini N, Wagner D. ROP is still dangerous: Breaking modern defenses. In: Proc. of the 23rd USENIX Conf. on Security Symp. San Diego: USENIX Association, 2014. 385−399. https://www.usenix.org/system/files/conference/usenixsecurity14/sec14-paper-carlini.pdf

[21] 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 36th IEEE Symp. on Security and Privacy. San Jose: IEEE CS Press, 2015. 745−762. [doi: 10.1109/SP.2015.51]

[22] Szekeres L, Payer M, Wei T, Song D. SoK: Eternal war in memory. IEEE Symp. on Security and Privacy, 2013,12(3):48−62. [doi: 10.1109/SP.2013.13]

[23] webbench-1.5. http://home.tiscali.cz/~cz210552/webbench.html

[24] Lu KJ, Nürnberger S, Backes M, Lee W. How to make ASLR win the clone wars: Runtime re-randomization. In: Proc. of the 23rd Annual Network and Distributed System Security Symp. (NDSS 2016). San Diego: The Internet Society, 2016. [doi: 10.14722/ndss.2016.23173]

[25] Seibert J, Okhravi H. Information leaks without memory disclosures: Remote side channel attacks on diversified code. In: Proc. of the 21st ACM Conf. on Computer and Communications Security. Scottsdale: ACM Press, 2014. 54−65. [doi: 10.1145/2660267.2660309]

[26] Giuffrida C, Kuijsten A, Tanenbaum AS. Enhanced operating system security through efficient and fine-grained address space randomization. In: Proc. of the 21st USENIX Security Symp. Bellevue: USENIX Association, 2012. 475−490. https://www.usenix.org/system/files/conference/usenixsecurity12/sec12-final181.pdf

[27] Chen Y, Wang Z, Whalley D, Lu L. Remix: On-demand live randomization. In: Proc. of the 6th ACM Conf. on Data and Application Security and Privacy. New Orleans: ACM Press, 2016. 50−61. [doi: 10.1145/2857705.2857726]

[28] Abadi M, Budiu M, Erlingsson U, Ligatti J. Control-Flow integrity. In: Proc. of the 12th ACM Conf. on Computer and Communications Security. Alexandria: ACM Press, 2005. 315−326. [doi: 10.1145/1102120.1102165]

[29] Zhang C, Wei T, Chen Z, Duan L. Practical control flow integrity and randomization for binary executables. In: Proc. of the IEEE Symp. on Security and Privacy. 2013. 559−573. [doi: 10.1109/SP.2013.44]

[30] Mashtizadeh AJ, Bittau A, Boneh D, Mazieres D. CCFI: Cryptographically enforced control flow integrity. In: Proc. of the 22nd ACM Conf. on Computer and Communications Security. Denver: ACM Press, 2015. 315−326. [doi: 10.1145/2810103. 2813676]

[31] Niu B, Tan G. Per-Input control-flow integrity. In: Proc. of the 22nd ACM Conf. on Computer and Communications Security. Denver: ACM Press, 2015. 914−926. [doi: 10.1145/2810103.2813644]

[32] Victor VDV, Andriesse D, Goktas E, Gras B. Practical context-sensitive CFI. In: Proc. of the 22nd ACM Conf. on Computer and Communications Security. Denver: ACM Press, 2015. 927−940. [doi: 10.1145/2810103.2813673]

[33] Chen ZF, Li QB, Zhang P, Wang Y. A kernel code reuse attack detection technique for Linux. Ruan Jian Xue Bao/Journal of Software, 2017,28(7):1732−1745 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5058.htm [doi: 10.13328/j.cnki.jos.005058]

[34] Carlini N, Barresi A, Payer M, Wagner D. Control-Flow bending: On the effectiveness of control-flow integrity. In: Proc. of the 24th USENIX Conf. on Security Symp. Washington: USENIX Association, 2015. 161−176. https://www.usenix.org/system/files/conference/usenixsecurity15/sec15-paper-carlini.pdf

[35] Onarlioglu K, Bilge L, Lanzi A, Balzarotti D, Kirda E. G-Free: Defeating return-oriented programming through gadget-less binaries. In: Proc. of the 26th Annual Computer Security Applications Conf. Austin: ACM Press, 2010. 49−58. [doi: 10.1145/1920261.1920269]

[36] Backes M, Holz T, Kollenda B, Koppe P, Nurnberger S, Pewny J. You can run but you can't read: Preventing disclosure exploits in executable code. In: Proc. of the 21st ACM Conf. on Computer and Communications Security. Scottsdale: ACM Press, 2014. 1342−1353. [doi: 10.1145/2660267.2660378]

[37] Kuznetsov V, Szekeres L, Payer M, Candea G, Sekar R, Dawn S. Code-Pointer integrity. In: Proc. of the 11th USENIX Symp. on Operating Systems Design and Implementation. Broomfield: USENIX Association, 2014. 147−163. https://www.usenix.org/system/files/conference/osdi14/osdi14-paper-kuznetsov.pdf

[38] Evans I, Fingeret S, Gonzalez J, Otgonbaatar U, Tang T, Shrobe H, Sidiroglou-Douskos S, Rinard M, Okhravi H. Missing the point (er): On the effectiveness of code pointer integrity. In: Proc. of the IEEE Symp. on Security and Privacy. 2015. 781−796. [doi: 10.1109/SP. 2015.53]

[39] Lu K, Song C, Lee B, Chung SP, Kim T, Lee W. ASLR-Guard: Stopping address space leakage for code reuse attacks. In: Proc. of the 22nd ACM Conf. on Computer and Communications Security. Denver: ACM Press, 2015. 280−291. [doi: 10.1145/2810103.2813694]

附中文参考文献:

[4] 傅建明,刘秀文,汤毅,李鹏伟.内存地址泄漏分析与防御.计算机研究与发展,2016,53(8):1829−1849. [doi: 10.7544/issn1000-1239.2016.20150526]

[33] 陈志锋,李清宝,张平,王烨.面向 Linux 的内核级代码复用攻击检测技术.软件学报,2017,28(7):1732−1745. http://www.jos.org.cn/1000-9825/5058.htm [doi: 10.13328/j.cnki. jos.005058]

张贵民(1987－),男,山东济南人,博士生,主要研究领域为信息安全,可信计算.

曾光裕(1966－),女,副教授,主要研究领域为信息安全,可信计算.

李清宝(1967－),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,可信计算.

赵宇韬(1992－),男,硕士,主要研究领域为网络信息安全.