软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2019,30(11):3518−3534 [doi: 10.13328/j.cnki.jos.005539]

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

基于运行特征监控的代码复用攻击防御*

张贵民[1,2,3], 李清宝[1,2], 张 平[1,2], 程三军[4]

[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)

[3](信息保障技术重点实验室,北京 100072)

[4](河南省人民检察院,河南 郑州 450000)

通讯作者: 张贵民, E-mail: zh.guimin@163.com

摘 要: 针对代码复用的攻击与防御已成为网络安全领域研究的热点,但当前的防御方法普遍存在防御类型单一、易被绕过等问题.为此,提出一种基于运行特征监控的代码复用攻击防御方法 RCMon.该方法在分析代码复用攻击实现原理的基础上定义了描述程序正常运行过程的运行特征模型 RCMod,并提出了验证程序当前运行状态是否满足 RCMod 约束规则的安全验证自动机模型.实现中,通过直接向目标程序中植入监控代码,使程序运行到监控节点时自动陷入,并由 Hypervisor 实现运行特征库的构建和安全验证.实验结果表明,RCMon 能够有效地防御已知的绝大部分代码复用攻击,平均性能开销约为 22%.

关键词: 代码复用攻击;运行特征;系统调用;插桩

中图法分类号: TP309

中文引用格式: 张贵民,李清宝,张平,程三军.基于运行特征监控的代码复用攻击防御.软件学报,2019,30(11):3518−3534. http://www.jos.org.cn/1000-9825/5539.htm

英文引用格式: Zhang GM, Li QB, Zhang P, Cheng SJ. Defending code reuse attacks based on running characteristics monitoring. Ruan Jian Xue Bao/Journal of Software, 2019,30(11):3518−3534 (in Chinese). http://www.jos.org.cn/1000-9825/5539.htm

Defending Code Reuse Attacks Based on Running Characteristics Monitoring

ZHANG Gui-Min[1,2,3], LI Qing-Bao[1,2], ZHANG Ping[1,2], CHENG San-Jun[4]

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

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

[3](Science and Technology on Information Assurance Laboratory, Beijing 100072, China)

[4](People's Procuratorate of Henan Province, Zhengzhou 450000, China)

Abstract: Code reuse attacks (CRAs) and their defense technologies have been the hot topic in network security field. However, current defense technologies usually focus on a single type of attacks and can be easily bypassed by other attacks. This paper presents a method called RCMon to defend CRAs based on running characteristics monitoring to overcome this problem. RCMon defines the running characteristics model (RCMod) according to the realize theory of CRAs and designs a safety verification automaton to verify whether current status meets the constraints in the RCMod. When RCMon is implemented, monitor code is instrumented into the target executable directly so that target program will trap in the Hypervisor when it runs to monitoring nodes, then the construction of running characteristics databse and safety verifications will be both performed by the Hypervisor. The experiment results show that RCMon can effectively detect and defense mostly CRAs, and induces average 22% performance penalty.

* 基金项目: 国家社会科学基金(15AJG012); 核高基国家科技重大专项(2013JH00103); 信息保障技术重点实验室开放基金(KJ-15-107)

Foundation item: National Social Science Foundation of China (15AJG012); CHB National Science and Technology Major Project of China (2013JH00103); Foundation of Science and Technology on Information Assurance Laboratory (KJ-15-107)

收稿时间: 2016-12-15; 修改时间: 2017-04-05, 2017-11-26; 采用时间: 2017-12-26

Key words: code reuse attack (CRA); running characteristics; system call; instrumentation

代码复用攻击(code reuse attack,简称 CRA)正严重威胁着网络系统的安全.CRA 并不向内存中植入任何恶意代码,而是首先在内存中的已有代码中寻找可用的指令序列(即 gadget),然后劫持控制流,使这些 gadget 得到执行从而实现攻击.因此,CRA 能绕过数据执行阻止技术(data executive prevention,简称 DEP)[1]和 W⊕X 等内存保护机制,典型 CRA 包括 return-into-libc(RILC)[2]、ROP[3]、JOP[4]、COOP[5]等.

CRA 防御方法总体可分为 3 类:代码随机化防御方法、控制流完整性保护方法和 CRA 特征检测方法.

(1) 代码随机化防御方法

代码随机化防御方法通过增加攻击者获取 gadget 的难度,来防御代码复用攻击.ASLR[6]通过随机化动态链接库和可执行文件的加载地址,使攻击者无法定位 gadget.之后,不同粒度的随机化方法[7,8]被先后提出,以增强 ASLR 方法的安全性.但攻击者仍可通过内存泄漏[9]找到 gadget,例如 JIT-ROP 攻击[10].为了应对内存泄漏+代码复用攻击,研究者提出了动态随机化方法.Isomeron[11]和 Remix[12]分别利用双函数副本执行和函数内基本块变换的方式实现对函数内部的随机化,但都无法防御函数级复用攻击;TASR[13]实现了对程序整个代码段内存位置的随机变化,但代码段内容不变.因此,一旦攻击者定位到该程序所在内存即可掌握其所有代码特征.另外,该方法仅对 C 语言程序实施保护,应用范围受限.

(2) 控制流完整性保护方法

控制流完整性保护方法通过阻止攻击者篡改控制流,使 gadget 序列得不到执行.CFI[14]基于控制流图(control-flow graph,简称 CFG)防御所有控制流劫持攻击,但开销较大,实用性差.为了降低开销,CCFI[15]、Context-sensitive CFI[16]和 CFI-KCraD[17]等方法通过结合对程序语义的分析,保证控制流在一定层度上的完整性,提高了方法的实用性.但控制流被劫持的风险依然存在[18],例如 CFI-KCraD 中的方法只能防御指令片段的复用,而无法防御函数级复用.

(3) CRA 特征检测方法

CRA 特征检测方法通过监控当前是否存在代码复用攻击的执行特征来检测和阻止攻击的实现.ROPecker[19]通过检测是否执行了超过某个数量(阈值)的 gadget 来检测代码复用攻击,其问题在于难以确定合理的阈值,且无法防御利用长 gadget 实施的攻击[20].ROPdefender[21]基于 call 指令与 ret 指令的匹配来防御 ROP 攻击,但 call 和 ret 指令并不总是成对出现,且该方法无法防御除 ROP 外的其他代码复用攻击,如 JOP、COOP 等.文献[22]提出通过检测 ret 指令的执行频率来推断 ROP 攻击,但无法防御 JOP、CPROP[23]等攻击.

除了上述防御方法外,SoftBound[24]、Baggy Bounds Checking[25]等方法通过检测内存错误,从源头上来防御攻击,但开销太大;文献[26]基于编译器来消除用于构建 gadget 的指令,该方法依赖源码且必须对所有程序模块进行处理,实现复杂;文献[27]通过阻止对代码的读操作使攻击者无法通过直接扫描内存获取 gadget,但攻击者仍可通过其他内存泄漏方式获取到 gadget 并构造攻击;CPI[28]通过保护代码指针的完整性来防御代码复用攻击,但该方法已被证明是可以被绕过的[29];DFI[30]基于静态分析的到达定义集合对数据进行保护,但需要源码且需要处理所有相关库和模块.

另外,还有通过代码和数据隔离的方法[31],可防御内存泄漏攻击,却无法防御 return-into-libc 攻击.

通过分析当前代码复用攻击的防御方法,发现依然还存在以下几个方面的问题.

1) 代码随机化方法通过阻止攻击者获取有效的 gadget 信息来预防代码复用攻击,尤其是动态随机化,但攻击者仍可借助侧信道攻击[32]等方式绕过该类防御方法.
2) 控制流完整性保护方法阻止攻击者执行 gadget 串,但研究表明,即使在细粒度控制流完整性保护下,攻击者仍可劫持控制流执行特定代码.
3) 基于 CRA 特征检测的方法在代码复用攻击运行阶段检测和阻止攻击的进一步执行,但无法涵盖当前各种新型代码复用攻击的所有特征,防御能力有限.

代码随机化方法和控制流完整性保护方法能够在一定程度上增加 CRA 实现的难度,但总会出现能够突破

防护的新型攻击;基于 CRA 特征检测的方法不关注如何阻止攻击的发起,而是着眼于如何尽快在攻击发起后检测并阻止其进一步执行,但问题在于该类方法无法跟上代码复用攻击技术更新的速度,对新型攻击无能为力.针对上述问题,本文提出一种基于程序运行特征监控的代码复用攻击防御方法(running characteristics monitoring,简称 RCMon).该方法首先对 CRA 的基本原理进行分析,并在此基础上定义了程序运行特征模型(running characteristics model,简称 RCMod).该模型由一系列程序关键节点处的运行特征模式组成,即各关键节点之间调用的关键系统调用的类型以及执行的总次数等统计特征.另外,还基于 RCMod 设计了防御 CRA 的安全验证自动机模型;然后,利用二进制插桩技术直接向被保护程序(目标程序)可执行文件中的所有关键节点处植入监控代码,使目标程序运行到关键节点时陷入到 Hypervisor;之后,由 Hypervisor 在训练阶段构建目标程序的运行特征库;最后,Hypervisor 按照安全验证自动机模型对目标程序的实时运行特征进行监控,从而实现对代码复用攻击的防御.

攻击者实施代码复用攻击或其他攻击的目的都是为了执行某些特定操作.RCMon 的研究基于这样一种假设,即攻击者在实施恶意操作时难免要通过系统调用和操作系统进行交互.由于不借助任何系统调用的攻击所能实施的操作有限,如非控制数据攻击[33],这类攻击并不在本文的讨论范围.

RCMon 的设计目标是:

1) 不需要程序源码和定制的编译器,不改变系统配置和模块,可兼容于遗留代码(legacy code);
2) 能够防御当前已知的各类 CRA(详见第 3.1 节表 5),对未知的新型 CRA 也具有一定的防御能力;
3) 在对目标程序进行保护时,要使目标程序、操作系统和其他应用程序的开销均保持在可接受范围.

本文第 1 节描述 RCMon 的总体架构及相关模型设计.第 2 节论述 RCMon 的具体设计和实现方法.第 3 节实现 RCMon 原型系统并对其进行测试.第 4 节讨论 RCMon 的局限性.第 5 节对全文工作进行总结.

1 总体架构

RCMon 的总体架构如图 1 所示,由目标程序插桩阶段(Instrumentation phase)、训练阶段(Training phase)和运行阶段(Running phase)这 3 部分组成.

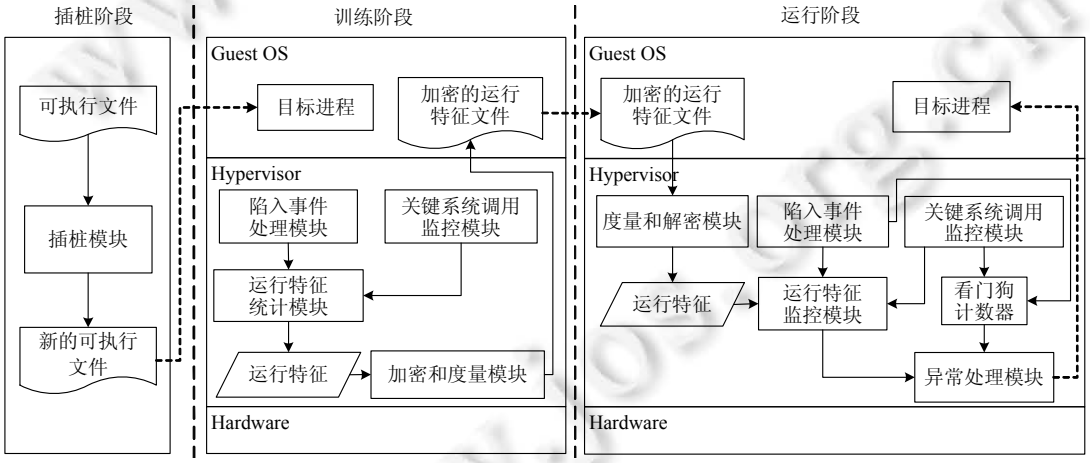

Fig.1 Overall architecture of RCMon

图 1 RCMon 方法总体架构

插桩阶段直接处理目标程序的可执行文件,由插桩模块(instrumentation module)将特定指令序列(即监控指令)植入到目标程序中的某些关键节点,其中,监控指令中含 vmcall 指令[34]以及标识程序关键节点的重要信息,最终生成一个新的可执行文件;训练阶段是在一个已开启了虚拟机监控器 Hypervisor 的客户机环境中运行由插桩阶段生成的新可执行文件,该 Hypervisor 能够监控程序运行过程中执行的所有关键系统调用和 vmcall 指

令,从而获得系统调用执行信息和目标程序的关键节点信息,然后由运行特征统计分析模块(RCs statistic module)根据这些信息建立目标程序的运行特征库,并通过加密和度量模块(encryption and measurement module)处理后存储在特征文件中,作为下一步验证目标程序行为特征是否正常的依据,最后通过多次训练以提高特征库的完备性;运行阶段,首先由度量和解密模块(measurement and decryption module)负责验证目标程序特征文件的完整性并解密该文件,然后将该文件加载到 Hypervisor 内存中,并加载运行目标程序.在目标程序运行过程中,Hypervisor 根据监控得到的关键系统调用执行信息,在每次监控指令标识的关键节点处验证目标程序在该节点处的运行特征是否正常:若不正常,则表明程序遭到攻击,立即终止程序运行;若正常,则根据运行特征库中对下一步要执行的关键系统调用的信息设置每个关键系统调用对应的看门狗计数器(watchdog counter,简称 WDC),由它实现对每次关键系统调用的合法性进行验证.通过这两种安全机制,实现对代码复用攻击的有效防御.

1.1 程序运行特征模型

CRA[2−5,10,23,35,36]的本质都是通过串连以间接跳转(indirect jump)指令、间接调用(indirect call)指令以及 ret 指令结尾的 gadget 来实现对内存中已有代码的复用,而根据复用代码的粒度不同,存在指令片段复用和函数级复用两种类型,如图 2 所示.在图 2 中,实线箭头表示正常的控制流,当攻击者发现当前内存中存在 *gadget*1~*gadget*5 这 5 个可利用的指令序列时,则通过篡改栈、寄存器或者虚函数表指针等方式实现代码复用攻击;图中虚线箭头表示该攻击的执行过程,图中 $instruction1_i$ 和 $instruction2_{i+k}$ 为 indirect jump 指令,该攻击示例中既包含了对程序自身函数 *func*1 和 *func*2 中指令片段的复用,也包含了对库函数 *func*4 的函数级复用.根据前文假设,*gadget*1~*gadget*5 中必定存在某个 gadget 调用了相关系统调用以实现攻击目标,而且由于 gadget 间的跳转执行改变了程序的原有执行过程,这势必将改变目标程序原有的系统调用执行过程.因此,通过对异常系统调用的及时检测和阻止可实现对攻击的有效防御,RCMon 正是基于该论断进行研究的.

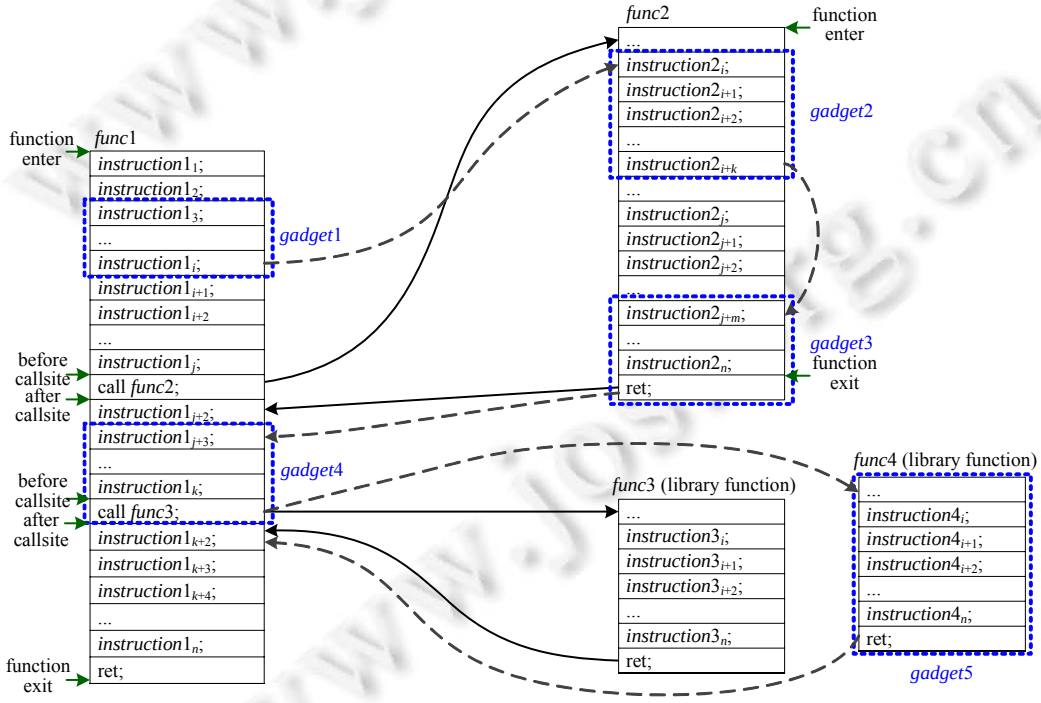

Fig.2 An example of CRA

图 2 CRA 示例

为了检测异常行为,必须首先界定什么是正常行为.如果借助若干个程序执行过程中的关键节点将程序的整个执行过程分成多个不同的执行域,那么每个区域都有自己需要的系统调用的类型和数量,将其称为该区域的运行特征.当关键节点的选取恰当时,通过验证每个区域的运行特征,即可实现对代码复用攻击的检测和阻止.基于对 CRA 实现方式的分析,由关键节点划分的区域必须能够覆盖 CRA 采用的以 indirect jump,indirect call 和 ret 指令为结尾的指令片段,且要尽量实现就近覆盖.最直接的想法是以所有的上述 3 类指令作为关键节点进行划分,但考虑到监控效率和实现复杂度,RCMon 选择采用 indirect call 和 ret 指令作为关键节点,并辅以调用点前后位置节点,实现对所有已经用于以及将来可能会用于 CRA 中的转移指令进行覆盖.图 2 中标出了各个关键节点的信息,即对应 indirect call 和 ret 指令的每个函数的入口(function enter,简称 FEN)节点和出口(function exit,简称 FEX)节点,以及函数调用点前(before callsite,简称 BC)节点和调用点后(after callsite,简称 AC)节点.之所以引入 BC 和 AC 节点,是由于 RCMon 为了达到不改变系统环境和模块的目的,没有对动态链接库中的函数(如图 2 中的 *func*3 和 *func*4)进行插桩,因此无法监控库函数的入口和出口以及库函数中又间接调用的其他库函数的入口和出口.通过增加 BC 和 AC 节点,RCMon 将库函数中的所有执行过程抽象为一个整体,不仅达到了不改变系统环境和模块的目的,也实现了对库函数执行行为的有效监控.通过上述 4 类关键节点的划分,除 indirect call 和 ret 指令外的其他可能指令(包括 indirect jump 指令)构建的所有 gadget,都只能存在于由 FEN 和 BC、FEN 和 FEX、AC 和 FEN 以及 BC 和 AC 划分的区域中.

基于上述分析,构建了程序的运行特征模型 RCMod 来描述程序的正常运行特征.为了描述 RCMod 的构建方法,首先给出描述程序的运行特征的模式定义.

定义 1. 运行特征模式(running characteristic pattern,简称 RCP)是一个 4 元式(*procID*,*fID*,*NT*,*SCCA*),其中,

- *procID*:该运行特征所属程序的 ID 编码,即标识某个被保护程序的整数值,由用户指定.
- *fID*:关键节点所处函数体的 ID 编码,即标识某个被保护程序中不同函数体的整数值,由算法自动分配.
- *NT*:关键节点类型.当 *NT* 为 1~4 时,依次表示当前关键节点为 BC,FEN,FEX 和 AC 节点.
- *SCCA*:系统调用特征序列.假设运行特征中涉及的系统调用共有 $N_{sc}(N_{sc}\geqslant 1)$种,则 $SCCA=\{scca[i][j]|i\in[0,N_{sc}-1],j\in[0,1]\}$,其中,$scca[i][0]$和 $scca[i][1]$分别表示程序到达本节点时第 i 种系统调用已经执行了的总次数和在该节点到下一节点之间的区域内将要执行的次数.

由定义 1 可知,一个 RCP 给出了某个程序在某个关键节点处的运行特征.一个程序往往由多个函数实现,包括对某些库函数的调用,必然引入一定数量的关键节点.假设一个程序共含有 k 个关键节点,则至少含有 k 个 RCP,因为当存在对某个函数的循环多次调用时,每次循环时与该函数相关的各个节点的 $scca[i][0]$都将发生变化,即生成新的 RCP.该程序的运行特征模式空间$\Omega=\{RCP_1,RCP_2,RCP_3,\ldots,RCP_n\}(n\geqslant k,n\in\mathbf{N}^+)$,利用程序一次运行中的运行轨迹经过的所有关键节点的 RCP 的集合即可描述该程序本次运行过程的运行特征,详见定理 1.

定理 1. 设 F 为Ω的子集族且为 C 族(即 F 为Ω的所有子集构成的子集族),那么程序一次运行的运行特征一定可由集合 R 表示,且 $R\in F$.

证明:从程序的所有运行轨迹中任选一条轨迹,假设该条轨迹中含有的运行特征模式为 $RCP_i,RCP_j,RCP_k,\ldots,RCP_m(1\leqslant i\leqslant j\leqslant k\leqslant\ldots\leqslant m\leqslant n;i,j,k,\ldots,m\in\mathbf{N}^+)$,此时:

集合 $R=\{RCP_i,RCP_j,RCP_k,\ldots,RCP_m\}$即表示了该程序本次运行的运行特征;

因为 $RCP_i,RCP_j,RCP_k,\ldots,RCP_m\in\Omega$,所以 $R\subseteq\Omega$;

因为 F 为Ω的子集族且为 C 族,所以 $R\in F$.定理 1 得证.

R 描述了程序一次执行过程中的所有运行特征,因此,R 即是 RCMod 模型的具体表现形式. □

1.2 安全验证自动机模型

在运行阶段,RCMon 采用两种安全机制:基于关键节点陷入事件触发的后向安全验证(backward safety verification,简称 BSV)和基于 WDC 的前向安全验证(forward safety verification,简称 FSV).

BSV 是在由监控指令导致的关键节点陷入时进行的第 1 类验证,将陷入节点的 *procID*、*fID*、*NT* 以及当前各个关键系统调用执行的总次数 $scca[i][0](0\leqslant i\leqslant N_{sc}-1)$与事先训练获得的特征库中的值进行匹配验证.如无

匹配项,则说明该节点之前的运行过程已经遭到攻击,此时终止该程序的执行;否则,继续执行 FSV.BSV 机制保证了当前关键节点的到来是合法的,只有通过 BSV 机制验证过的指令序列(或整个函数)才能成功执行.

但 BSV 存在一定的局限性.假设目标程序遭到代码复用攻击并将控制流劫持到动态链接库中的代码,如果之后控制流不再回到目标程序的自身代码执行,即一直到攻击完成都不会再到达含监控代码的下一个关键节点,那么 BSV 是无法防御该类攻击的;另外,如果攻击者劫持一段时间控制流后再次到达另一个关键节点,虽然此时 BSV 能够检测到攻击,但攻击可能已经执行了想要执行的恶意操作.因此,采用另外一种验证机制 FSV 来弥补 BSV 的不足.

在论述 FSV 机制前,首先对本文提出的 WDC 进行说明.看门狗也叫做看门狗计时器(watchdog timer,简称 WDT),是单片机中的一种安全机制.WDT 是一个计时器电路,在单片机开始工作后启动,微处理器控制单元(microprocessor control unit,简称 MCU)正常情况下会定期输出一个信号给 WDT,称为喂狗,目的是使 WDT 清零.而一旦受到干扰导致程序跑飞时,MCU 将无法在规定时间内执行喂狗操作,最终导致 WDT 超时.此时,WDT 就会给 MCU 一个复位信号,使 MCU 复位,从而防止 MCU 死机.本文提出的 WDC 正是借鉴了 WDT 的思想.WDC 在这里是一个计数器而不是计时器,它接受一个数值作为临界值.当启动后,WDC 的数值通过事件触发的方式增长,所关注事件(本文为所监控的关键系统调用)每触发一次,计数器加 1.WDC 与 WDT 的不同点还包括:WDC 只有在恰好等于临界值时接收到喂狗操作才说明系统的运行正常,否则都会发出异常警告.

FSV 在经过后向验证后执行,将找到的匹配项中的各个 $scca[i][1](0\leqslant i\leqslant N_{sc}-1)$的值设置为对应的各 WDC 的临界值.RCMon 为每个受保护的目标进程维护着 N_{sc} 个 WDC,它们分别由相应的关键系统调用的执行作为计数器加 1 的触发事件,即所关注的系统调用每执行一次,WDC 的值加 1,并以关键节点陷入事件作为喂狗操作.若 WDC 在达到临界值前接收到了喂狗操作,即下一个关键节点提前到达,则说明在本应执行该 WDC 对应的系统调用的某处指令并没有得到执行,程序遭到攻击;若 WDC 超过临界值,则说明此时执行了多出正常范围的系统调用,程序遭到攻击;而只有当 WDC 正好在临界值时接收到喂狗操作,才表明程序运行过程符合其运行特征.

FSV 的作用是保证目标程序在该节点到下个节点之间执行的所有关键系统调用符合运行特征,可实时检测攻击的发生.但 FSV 无法防御某些函数级复用攻击,例如,当攻击者复用程序本身具有的函数时,由于所有自身函数的入口和出口节点及其内部的其他节点都是对该函数自身行为的约束,因此当该函数完整执行时,FSV 无法检测到该类复用攻击.而当结合 BSV 之后,RCMon 则可对所有函数执行的合法性进行验证,此时,复用函数无法在攻击者设定的位置执行,即可实现对该类攻击的有效防御.

RCMon 的安全验证机制 BSV 和 FSV 可抽象为一个带输出的有限自动机模型 $M=(Q,\Sigma,\Delta,\delta,\lambda,q_0)$,其中,

- $Q=\{S_0,S_1,S_2,S_3,S_4\}$为该模型的所有状态;
- $\Sigma=$\{get information of every critical system call, verification failed, verification successful, continue execution, exceed the critical value, below the critical value\}为该模型的输入字母表;
- $\Delta=$\{vmcall in critical node, kicking the dog\}为该模型的输出字母表;
- δ为状态转移函数,$\delta: Q\times\Sigma\to Q$;
- λ为输出函数,$\lambda: Q\to\Delta$;
- q_0 为初始状态,$q_0=S_0$.

模型中的 S_0 表示经过插桩处理后的目标程序的正常执行状态;S_1 表示执行后向验证状态;S_2 表示执行前向验证状态;S_3 表示关键系统调用的执行状态;而 S_4 则表示异常状态,即目标程序遭到攻击.

模型 M 对应的状态转移过程如图 3 所示.当程序执行到定义的关键节点时,监控指令得到执行,当其中的 vmcall 指令执行时使程序陷入到 Hypervisor 中,并同时将节点信息(例如 *procID*,*fID* 以及 *NT*)传递给 Hypervisor,此时,状态机从 S_0 进入 S_1,并在状态 S_1 执行后向验证.若后向验证失败,则由状态 S_1 直接进入状态 S_4,产生警告,中断目标程序的运行;否则,由状态 S_1 进入状态 S_2,在 S_2 中执行前向验证.只有当出现 WDC 超过临界值或在喂狗操作提前到达时,才由状态 S_2 进入状态 S_4,产生警告,并中断目标程序的运行;否则,由状态 S_2 进入状态 S_0,目标程序继续运行.

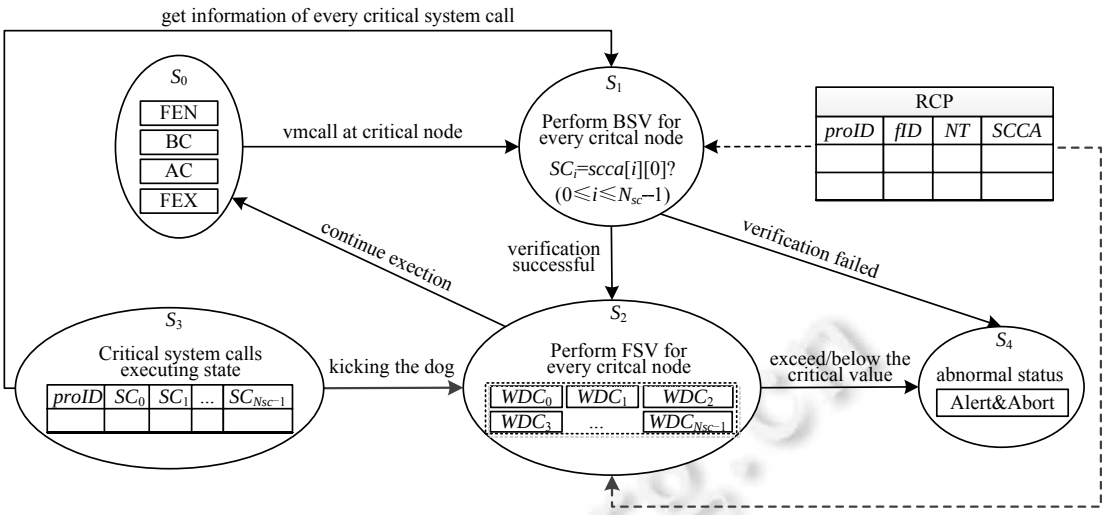

Fig.3 Model of safety verification automaton

图 3 安全验证自动机模型

2 监控指令植入与关键系统调用监控方法

2.1 监控指令植入

向目标程序中植入监控指令,是构建程序运行特征库和检测程序异常行为的重要一环.根据模型 RCMod,需要在目标程序中函数调用位置的前后以及所有本地函数的入口和出口处植入监控指令,如图 2 所示.为了使 RCMod 的实现不依赖于目标程序源码和编译器,且实现一次插桩、永久监控,RCMon 基于静态二进制插桩框架 Dyninst[37]实现上述功能.

基于 Dyninst 实现的监控代码植入算法如图 4 所示.首先,通过 Dyninst API 获取目标程序中所有能够进行插桩的函数信息,同时为每一个可插桩函数分配一个 *fID*,用于标识不同函数;然后,在每一个函数区域内分别定位函数入口、出口和函数调用点;之后,构建 4 类关键节点监控代码 snippet;最后,分别将构造的监控代码植入到对应的关键节点并生成新的可执行程序.

植入的监控代码需要实现两方面功能:一是报告当前节点的标识信息,包括 *procID*,*fID* 和 *NT*;二是在执行到关键节点时产生陷入.为了提高信息的传递效率,研究中采用寄存器作为传递信息的媒介,分别将 *procID*,*fID* 和 *NT* 保存到寄存器 EAX、ECX 和 EDX 中,然后,通过执行特权指令 vmcall 实现陷入,Hypervisor 通过读取 3 个寄存器获得当前关键节点的信息.为了避免在使用 3 个寄存器时对它们内容的保存和恢复的复杂性,实现时,将监控指令置于独立的函数中实现.

4 类不同节点处调用的监控函数的具体定义如图 5 所示,其中,参数 *functionID* 即图 4 所示算法中为每个函数分配的 *fID*,*procID* 则设定为唯一标识该程序的某个整数值,*NT* 直接根据对应的节点设置为 1~4.最后,将上述监控函数封装在一个动态链接库(算法 1 中的 *mylib.so*)中,利用 *loadLibrary* 函数将该库加载到目标程序的内存空间,供目标程序调用.

通过上述处理后,最终生成一个新的含监控代码的可执行文件.该文件可完全脱离 Dyninst 独立运行.监控代码植入后,程序控制流的变化情况如图 6 所示(注:图 6 是以 *funcX* 为本地函数为例进行说明的,当 *funcX* 为库函数时,则只有 BC 和 AC 节点,不存在 BEN 和 BEX 节点以及与 *user2vmmenter*,*user2vmmexit* 函数之间的控制流转移).可见,目标程序的执行过程已经完全处在 RCMon 的监控之下.

Algorithm 1. The algorithm for instrumenting monitoring code.

Input: *objbinary*,*mylib.so*(*dependency lib*).
Output: *objbinary_rcmon*.

```
Bpatch bp; int j;
BPatch_addressSpace*appBin=bp→openBinary(objbinary);
BPatch_image*appImage=appBin→getImage(·);
std::vector<BPatch_function*>allInstrumentedFuncs=appImage→getProcedures(false);
std::vector<BPatch_function*>::iterator func; int i=0;
funcInfor fi=new funcInfo[allInstrumentedFuncs.getlength(·)];
for (func=allInstrumentedFuncs.begin();fun!=allInstrumentedFuncs.end();++func){
    fi[i].name=func.getName(·); fi[i].id=i; i++;
}
appBin→loadLibrary(mylib.so);
for (func=allInstrumentedFuncs.begin();fun!=allInstrumentedFuncs.end();++func){
    j=0;
    while (fi[j].name!=func.getName(·)) j++;
    int fID=j;
    std::vector<BPatch_point*>*point1,*point2,*point3;
    points1=func→findPoint(BPatch_entry);
    points2=func→findPoint(BPatch_exit);
    points3=func→findPoint(BPatch_subroutine);
    std::vector<BPatch_snippet*>monitorargument;
    BPatch_snippet*argument=new BPatch_constExpr(fID);
    monitorargument.push_back(argument);
    std::vector<BPatch_function*>monitorfunc;
    appImage→findFunction("user2vmmenter",monitorfunc[0]);
    appImage→findFunction("user2vmmexit",monitorfunc[1]);
    appImage→findFunction("user2vmmbefore",monitorfunc[2]);
    appImage→findFunction("user2vmmafter",monitorfunc[3]);
    BPatch_funcCallExpr monitorCall1(*(monitorfunc[0]),monitorargument);
    BPatch_funcCallExpr monitorCall2(*(monitorfunc[1]),monitorargument);
    BPatch_funcCallExpr monitorCall3(*(monitorfunc[2]),monitorargument);
    BPatch_funcCallExpr monitorCall4(*(monitorfunc[3]),monitorargument);
    app→insertSnippet(monitorCall1,*points1);
    app→insertSnippet(monitorCall2,*points2);
    app→insertSnippet(monitorCall3,*points3,BPatch_callBefore);
    app→insertSnippet(monitorCall3,*points2,BPatch_callAfter);
}
```

```
class funcInfo{
    string name;
    int fID;
}
```

Fig.4 Algorithm for instrumenting monitoring code

图 4 监控代码植入算法

```
void user2vmmbefore(int functionID){asm volatile("vmcall"::"a"(procID),"c"(functionID),"d"(1));}//for BC
void user2vmmenter(int functionID){asm volatile("vmcall"::"a"(procID),"c"(functionID),"d"(2));}//for FEN
void user2vmmexit(int functionID){asm volatile("vmcall"::"a"(procID),"c"(functionID),"d"(3));}//for FEX
void user2vmmafter(int functionID){asm volatile("vmcall"::"a"(procID),"c"(functionID),"d"(4));}//for AC
```

Fig.5 Definitions of instrumented functions

图 5 植入函数的定义

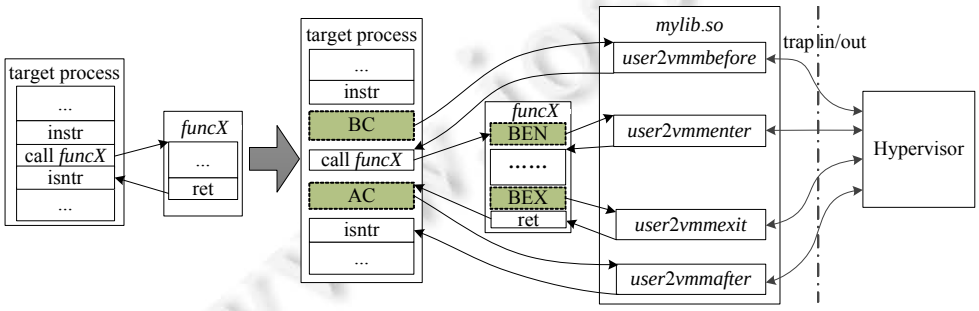

Fig.6 Control flow before and after instrumenting code

图 6 代码植入前后的控制流

2.2 关键系统调用监控

实现 RCMon 必须对系统调用进行监控,从而构建模型 RCMod,实现对目标进程的运行时监控.Linux 操作系统中存在数百个系统调用,若对所有系统调用都进行监控,会导致较大的时间开销和空间开销(更多内存用于存储 scca 数组和 WDC).在保证监控强度不变的条件下,为了降低监控带来的开销,RCMon 选取了只与系统安全和各类攻击行为相关性较高的关键系统调用进行监控.

文献[38]为了实现对恶意软件行为的描述,通过对 472 种 Linux 下的恶意软件进行分析,总结提炼出了恶意软件经常使用的 64 种系统调用;TASR[13]为了更好地防御内存泄漏,在总结分析各类攻击实现机制的基础上提出了最小攻击间隔理论,并将所有的输入类和输出类系统调用作为关键系统调用进行监控.RCMon 在借鉴已有研究成果的基础上,对各类系统调用进行了如下安全性分析.

(1) 是否能够用于改变系统内某类资源(含硬件资源和文件、进程等软件资源)的状态;

(2) 是否能够用于获取更高操作权限;

(3) 是否能够用于查看或传输各类信息.

通过上述分析,RCMon 最终选择监控 82 种(N_{SC}=82)与安全相关的关键系统调用,见表 1,分别对应于模型中的第 0 种~第 81 种系统调用.

Table 1 Critical system calls

表 1 关键系统调用

Resource type	System calls	Sum
File	sys_read,sys_write,sys_open,sys_creat,sys_fchdir,sys_execve,sys_chdir,sys_chmod, sys_writev,sys_utime,sys_pwrite64,sys_lseek,sys_preadv,sys_ftruncate,sys_fchmod, sys_quotactl,sys_chown,sys_ftruncate64,sys_lchown,sys_fchown,sys_fadvise64, sys_pwritev,sys_utimes,sys_fadvise64_64,sys_openat,sys_readv,sys_pread64	27
Process	sys_kill,sys_sigreturn,sys_rt_sigaction,sys_capset,sys_tkill,sys_tgkill,sys_exit,sys_sigaction	8
Pipe	sys_pipe,sys_pipe2	2
Memory	sys_brk,sys_old_mmap,sys_mremap,sys_mprotect	4
Network	sys_ioctl,sys_old_select,sys_listen,sys_socketpair,sys_epoll_create,sys_mbind,sys_socket, sys_epoll_create1,sys_socketcall,sys_sendmmsg,sys_recvmmsg,sys_sendfile,sys_sendto, sys_send,sys_sendmsg,sys_sendfile64,sys_bind,sys_connect,sys_recvfrom,sys_recvmsg,sys_recv	21
Module	sys_delete_module, sys_init_module	2
System	sys_sysctl,sys_setrlimit,sys_getrusage,sys_uselib,sys_ioperm,sys_iopl, sys_reboot,sys_swapon,sys_sysinfo,sys_uname,sys_newuname,sys_olduname	12
User	sys_setuid,sys_setgid	2
Message	sys_msgget,sys_msgctl,sys_msgsnd,sys_msgrcv	4

为了高效地监控上述关键系统调用,且不改变操作系统内核,RCMon 采用一种基于动态指令替换的监控方法实现该功能[35],即在系统运行过程中,通过底层的 Hypervisor 修改内核代码内存,将要监控的关键系统调用对应的处理函数入口处(可通过事先分析操作系统的导出符号表 system.map 得到各个关键系统调用处理函数的入口地址)的若干指令替换为 vmcall 指令,使得关键系统调用执行时产生陷入,从而达到监控的目的.同时,还需要在 Hypervisor 中对被 vmcall 替换的那些指令进行模拟,保证系统调用的正常执行.

以 Ubuntu12.04 使用的 3.2.0-29-generic-pae i386 的内核为例,该内核代码中每个关键系统调用处理函数的入口处均为长度为 3 个字节的“push %ebp; movl %esp, %ebp”两条指令,由于 vmcall 指令的长度也为 3 个字节,因此对于该发行版内核来说,只需将函数入口处的头 3 个字节替换为 vmcall 指令即可.

指令替换后,所有关键系统调用处理函数在执行时都会由于 vmcall 指令而陷入 Hypervisor,此时获取客户机指令寄存器的值并进行分析:若该值为某个关键系统调用处理函数的入口地址,则说明当前进程调用了该系统调用;否则,vmcall 指令来自于对目标程序关键节点的监控指令,此时执行运行特征库构建或安全验证.最后,在 Hypervisor 中仿真执行被替换指令的功能.上述监控机制的实现过程如图 7 所示.

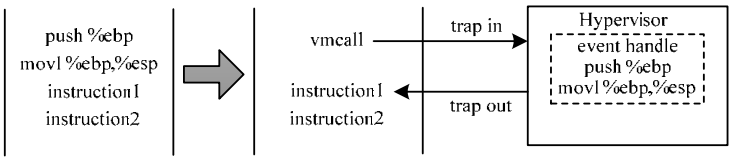

Fig.7 Monitoring mechanism of critical system calls

图 7 关键系统调用监控机制

2.3 监控优化

由 RCMon 的实现过程可知,在 Hypervisor 中执行的安全验证过程是该方法最核心和最复杂的部分,同时也引入了较大开销,对这部分的优化,对于降低 RCMon 的监控开销具有重要作用.当采用第 2.1 节和第 2.3 节中的方法完成对目标程序运行特征的构建后,通过分析获得的运行特征,发现存在以下特点:a) 目标程序正常运行过程中往往不会包含对所有 82 种关键系统调用的使用,而是存在某些未被使用的系统调用;b) 相邻关键节点 RCP 中的 $scca[i][0](0\leqslant i\leqslant N_{sc}-1)$局部或全部相同(在两个关键节点间执行所有 82 种关键系统调用的可能性很小).基于上述特点,提出两种对 RCMon 运行时监控过程的优化方法以降低其监控开销.

(1) 全局优化

对于目标程序正常运行过程中不会使用的关键系统调用,只在目标程序第 1 个函数的入口节点处对这些关键系统调用对应的 WDC 进行设置(全部设置为 0),从而保证目标程序一旦执行这些系统调用即可导致 WDC 超过临界值而报警.而在程序其他节点处执行 BSV 和 FSV 时,则不再对这些系统调用对应的 *SCCA* 中的项进行对比验证,也不再重置其对应的 WDC,即在程序入口处设置的 WDC 将作用于程序的整个运行过程.目标程序未使用的系统调用可通过分析由训练阶段得到的运行特征获得,并通过运行特征文件告知 Hypervisor 中的运行特征监控模块.

(2) 局部优化

当存在相邻两个或多个关键节点中的 $scca[i][0](0\leqslant i\leqslant N_{sc}-1)$完全一致的情况时,表明在这些关键节点之间未执行任何关键系统调用.此时,将这些连续的节点看作一个整体,仅在第 1 个和最后一个节点处执行 BSV 和 FSV,中间节点则不再进行任何验证.为实现该项优化,首先在运行特征中找到每个符合条件的连续节点的起始节点和最终节点,然后反汇编插桩后的新可执行程序,并根据中间关键节点的 *fID* 和 *NT* 的值定位到在这些节点处植入的调用监控函数的代码,然后使用 nop 指令将其替代,从而避免目标程序在这些节点处产生陷入.

通过上述处理,在未降低 RCMon 方法安全性的条件下,减少了目标程序陷入 Hypervisor 的次数,并降低了 Hypervisor 中执行安全验证的复杂度,实现了对 RCMon 性能的提升和优化.由第 3.2 节中的测试可知,实施优化后,RCMon 的性能提升了约 29%(即(31%−22%)/31%).

3 实验与结果分析

本节首先验证 RCMon 方法的有效性,即能否有效防御代码复用攻击,然后再测试该方法的监控开销.实验环境如下:主机 CPU 为 Intel Core™ i7-5500U CPU@2.40GHz;内存大小为 4GB;客户机操作系统采用的是 3.2.0-29-generic-pae i386 内核版本的 Ubuntu12.04;Hypervisor 是基于 Intel VT 技术设计的一个轻量级的虚拟机监控器,该监控器仅对无条件陷入事件[31]进行处理,目的是最大化地降低监控开销;另外,采用二进制文件插桩框架 Dyninst-9.1.0[37]实现对监控指令的植入.

3.1 有效性测试

为了测试 RCMon 的有效性,首先以一个具体实例 exam 程序来验证方法的可行性.

exam 的源码如图 8 所示,为了便于实现代码复用攻击,该程序 *func* 函数中含缓冲区溢出漏洞.首先编译得到可执行文件 exam,然后经过代码植入模块植入监控指令并生成新的可执行程序 exam_rcmon,并赋予该程序编号为 128(可任意指定,能够区分不同的被保护程序即可),即将图 5 中所示的全局变量 *procID* 初始化为 128.然后

经过训练过程,获得 exam_rcmon 的运行特征库,由于内容较多,表 2 仅列出了其部分运行特征.其中,(128,5,2,$SCCA_1$)为程序 exam_rcmon 中 *func* 函数的 FEN 节点的 RCP,(128,5,1,$SCCA_2$)为 *func* 中对 *read* 函数调用的 BC 节点的 RCP,(128,5,4,$SCCA_3$)为 *func* 中对 *read* 函数调用的 AC 节点的 RCP,(128,5,3,$SCCA_4$)为 *func* 的 FEX 节点的 RCP.

```
#include ⟨stdio.h⟩
#include ⟨unistd.h⟩
void func(·) {
    char buffer[128];
    read(STDIN_FILENO,buffer,256);
}
int main(int argc,char**argv) {
    func(·);
    write(STDOUT_FILENO,"hello world\n",12);
}
```

Fig.8 Source code of exam

图 8 exam 源码

Table 2 Running characteristics database of exam_rcmon

表 2 exam_rcmon 的运行特征库

procID	*fID*	*NT*	*SCCA*
128	4	2	*scca*[0][0]=7;*scca*[0][1]=0;*scca*[1][0]=1;*scca*[1][1]=1;*scca*[2][0]=123;*scca*[2][1]=0; *scca*[29][0]=1;*scca*[29][1]=0;*scca*[37][0]=1;*scca*[37][1]=0;*scca*[40][0]=12;*scca*[40][1]=0;
128	4	1	*scca*[0][0]=7;*scca*[0][1]=0;*scca*[1][0]=2;*scca*[1][1]=1;*scca*[2][0]=123;*scca*[2][1]=0; *scca*[29][0]=1;*scca*[29][1]=0;*scca*[37][0]=1;*scca*[37][1]=0;*scca*[40][0]=12;*scca*[40][1]=0;
…	…	…	…
128	6	1	*scca*[0][0]=7;*scca*[0][1]=0;*scca*[1][0]=18;*scca*[1][1]=1;*scca*[2][0]=123;*scca*[2][1]=0; *scca*[29][0]=1;*scca*[29][1]=0;*scca*[37][0]=1;*scca*[37][1]=0;*scca*[40][0]=12;*scca*[40][1]=0;
128	5	2	*scca*[0][0]=7;*scca*[0][1]=0;*scca*[1][0]=19;*scca*[1][1]=1;*scca*[2][0]=123;*scca*[2][1]=0; *scca*[29][0]=1;*scca*[29][1]=0;*scca*[37][0]=1;*scca*[37][1]=0;*scca*[40][0]=12;*scca*[40][1]=0;
128	5	1	*scca*[0][0]=7;*scca*[0][1]=1;*scca*[1][0]=20;*scca*[1][1]=1;*scca*[2][0]=123;*scca*[2][1]=0; *scca*[29][0]=1;*scca*[29][1]=0;*scca*[37][0]=1;*scca*[37][1]=0;*scca*[40][0]=12;*scca*[40][1]=0;
128	5	4	*scca*[0][0]=8;*scca*[0][1]=0;*scca*[1][0]=21;*scca*[1][1]=1;*scca*[2][0]=123;*scca*[2][1]=0; *scca*[29][0]=1;*scca*[29][1]=0;*scca*[37][0]=1;*scca*[37][1]=0;*scca*[40][0]=12;*scca*[40][1]=0;
128	5	3	*scca*[0][0]=8;*scca*[0][1]=0;*scca*[1][0]=22;*scca*[1][1]=1;*scca*[2][0]=123;*scca*[2][1]=0; *scca*[29][0]=1;*scca*[29][1]=0;*scca*[37][0]=1;*scca*[37][1]=0;*scca*[40][0]=12;*scca*[40][1]=0;
128	6	4	*scca*[0][0]=8;*scca*[0][1]=0;*scca*[1][0]=23;*scca*[1][1]=1;*scca*[2][0]=123;*scca*[2][1]=0; *scca*[29][0]=1;*scca*[29][1]=0;*scca*[37][0]=1;*scca*[37][1]=0;*scca*[40][0]=12;*scca*[40][1]=0;
128	6	1	*scca*[0][0]=8;*scca*[0][1]=0;*scca*[1][0]=24;*scca*[1][1]=2;*scca*[2][0]=123;*scca*[2][1]=0; *scca*[29][0]=1;*scca*[29][1]=0;*scca*[37][0]=1;*scca*[37][1]=0;*scca*[40][0]=12;*scca*[40][1]=0;
128	6	4	*scca*[0][0]=8;*scca*[0][1]=0;*scca*[1][0]=26;*scca*[1][1]=1;*scca*[2][0]=123;*scca*[2][1]=0; *scca*[29][0]=1;*scca*[29][1]=0;*scca*[37][0]=1;*scca*[37][1]=0;*scca*[40][0]=12;*scca*[40][1]=0;
…	…	…	…
128	10	4	*scca*[0][0]=8;*scca*[0][1]=0;*scca*[1][0]=32;*scca*[1][1]=1;*scca*[2][0]=123;*scca*[2][1]=0; *scca*[29][0]=1;*scca*[29][1]=0;*scca*[37][0]=1;*scca*[37][1]=0;*scca*[40][0]=12;*scca*[40][1]=0;
128	10	3	*scca*[0][0]=8;*scca*[0][1]=0;*scca*[1][0]=33;*scca*[1][1]=0;*scca*[2][0]=123;*scca*[2][1]=0; *scca*[29][0]=1;*scca*[29][1]=0;*scca*[37][0]=1;*scca*[37][1]=0;*scca*[40][0]=12;*scca*[40][1]=0;

在进行攻击测试时,通过内存泄漏分别找到在攻击中经常使用的 *libc* 库中的 *read*、*write* 和 *system* 函数的地址以及程序 exam_rcmon 中 *func* 函数的地址,然后利用缓冲区溢出漏洞篡改控制流,分别对上述函数进行复用攻击.上述 4 个测试的结果见表 3.

根据测试结果,4 种代码复用攻击均以失败告终.攻击中,将 *func* 的 ret 指令作为第 1 个 gadget 以实现向其他 gadget 的转移,因此直到 *func* 的 FEX 节点之前都是正常的.而在该节点后,由于对 ret 指令的目标地址进行了篡改,分别调用了 *read*,*write*,*system* 和 *func* 函数这 4 类指令序列.当执行 *write*、*read* 和 *system* 函数时,会分别调用 *sys_write*、*sys_read* 和 *sys_execve* 系统调用,这 3 类系统调用常用于各类攻击中.但由于根据(128,5,3,$SCCA_4$)设置的 3 类系统调用对应的 WDC 的临界值均为 0,因此当复用上述函数时,WDC 超过临界值,导致 FSV 失败.当复

用 *func* 函数时,由于 *func* 函数中会调用 *read* 函数,此时 *scca*[0][0]增加 1 变为 8,但特征库中并没有符合 *scca*[0][0]=8 且 *procID*、*fID* 和 *NT* 分别为 128、5 和 1 的 RCP 存在,因此 BSV 失败.另外,测试过程中没有发现任何误报.通过上述测试,验证了 RCMon 方法检测代码复用攻击的可行性.同时,借助 RCMon 还可以对攻击的发生位置进行定位,这有助于攻击发生后的软件安全漏洞和攻击实现机制的分析.

Table 3 Results of attacking tests

表 3 攻击测试结果

复用代码	攻击是否成功	RCMon 报告信息	验证失败时的节点信息	是否存在误报
write	否	FSV 失败(WDC 超过临界值)	*procID*=128,*fID*=5,*NT*=3	无
read	否	FSV 失败(WDC 超过临界值)	*procID*=128,*fID*=5,*NT*=3	无
system	否	FSV 失败(WDC 超过临界值)	*procID*=128,*fID*=5,*NT*=3	无
func	否	BSV 失败(未找到匹配项)	*procID*=128,*fID*=5,*NT*=1	无

为了更好地验证 RCMon 的有效性,除上述测试外,我们还对 4 类真实的应用程序(nginx,proftpd,mcrypt 以及 TORQUE)进行了测试.测试中,分别采用来自 http://www.elis.ugent.be/□svolckae 的 4 种不同的 ROP 对上述程序实施攻击.

- 攻击 1 针对 Web 服务器 nginx,它首先读取栈中的 canary 值以及漏洞函数栈帧底部的返回地址,利用该地址计算出 nginx 的基地址,然后基于对 nginx 的已有知识构造 ROP 链,并利用 nginx 中的栈缓冲区溢出漏洞(CVE-2013-2028)使 ROP 链获得执行.
- 攻击 2 针对 ftp 服务器 proftpd,它首先通过扫描 proftpd 的可执行文件和 libc 库定位构造 ROP 链所需要的 gadget,然后从/proc/pid/maps 中读取 proftpd 和 libc 的加载地址来确定 gadget 的绝对地址,最后通过一个未授权的 FTP 链接将含有 ROP 链的 buffer 发送到 proftpd,并利用 proftpd 中的栈缓冲区溢出漏洞(CVE-2010-4221)使其得到执行.
- 攻击 3 针对 mcrypt(一个用于替换 crypt 的加密程序),它首先从/proc 接口获得 mcrypt 和 libc 库的加载地址来构造 ROP 链,然后通过一个 pipe 将 ROP 链发送给 mcrypt,并利用 mcrypt 中的栈缓冲区溢出漏洞(CVE-2012-4409)执行该 ROP 链.
- 攻击 4 针对 TORQUE 资源管理器服务器,它首先读取 pbs_server 进程的加载地址并构造 ROP 链,然后通过一个未授权的网络连接将该 ROP 链发送到 TORQUE,并利用 TORQUE 中的栈缓冲区溢出漏洞(CVE-2014-0749)使攻击得到执行.

测试前,首先分别处理各目标程序(植入监控代码),然后通过将处理后的程序在不同输入和操作条件下分别运行 20 次、40 次、60 次和 80 次(每次运行 30min)来构造程序运行特征库.然后,在关闭和开启 RCMon 特征监控两种情况下,分别使用上述 4 个 ROP 攻击对目标程序进行攻击.根据攻击是否成功,验证 RCMon 防御代码复用攻击的有效性.需要注意的是,由于植入监控代码的影响,原有的 4 个攻击在构造 ROP 链时所使用的地址都需要更新为代码植入后的新地址.最后,再在开启 RCMon 特征监控的情况下运行各目标程序 20 次(每次仍然运行 30min),以测试是否存在误报.最终的测试结果见表 4.

Table 4 Results of real applications tests

表 4 真实应用测试结果

目标程序	攻击是否成功		误报次数(20 次)	误报次数(40 次)	误报次数(60 次)	误报次数(80 次)
	关闭 RCMon	开启 RCMon				
nginx	√	×	3	2	2	1
proftpd	√	×	0	0	0	0
mcrypt	√	×	0	0	0	0
TORQUE	√	×	1	1	0	0

由表 4 可知,RCMon 能够有效防御针对 nginx 等应用程序的 4 种 ROP 攻击,无漏报;但 RCMon 存在误报,尤其是 nginx 误报情况较为明显.主要原因在于,该类程序运行特征的影响因素较多,不同服务状态和客户链接数量都会对服务器的执行过程产生影响.另外,从测试结果可以发现,在训练阶段的训练越充分,RCMon 产生误

报的可能性就越小.

另外,本文总结了当前已知的各类代码复用攻击技术,并根据其实现机制判断RCMon是否能够有效阻止这些攻击.分析过程中均假设这些攻击中含有对某些关键系统调用的执行过程或者这些攻击的实施改变了程序原有的关键系统调用执行情况,而这个假设对大多数攻击都是适用的[13,38].对于函数级复用攻击来说,不含任何关键系统调用的情况极少,另外,即使对于本身不含任何关键系统调用的指令片段复用攻击,在gadget跳转串联过程中也极有可能会干预原有的系统调用执行过程,如跳过某些执行关键系统调用的指令等.最终的分析结果见表 5.通过上述测试和分析可知,RCMon 可实现对已知各类代码复用攻击的有效防御.同时,还在表 5 中将RCMon与其他相关防御方法进行了防御能力的对比,对比结果进一步表明了RCMon的有效性.

Table 5 Analysis of RCMon's defensive ability

表 5 RCMond 的防御能力分析

攻击类型	攻击实质	ASLR[6−8]	Isomeron[11], Remix[12]	TASR[13]	CFI[14−17]	ROPecker[19] ROPdefender[21]	RCMon
return-into-libc[2]	函数复用	√	×	√	√	×	√
COOP[5]	虚函数复用	×	×	√	×	×	√
ROP[3]	gadget 复用 (以 ret 指令结尾)	√	√	√	√	√	√
JOP[4]	gadget 复用 (以 jump 指令结尾)	√	√	√	√	×	√
LOP[36]	函数复用	√	×	√	√	×	√
SROP[39]	gadget 复用且含 sigreturn gadget	√	√	√	√	×	√
CPROP[23]	gadget 复用(仅使用 call-preceded gadget)	√	√	√	√	×	√
JIT-ROP[10]	gadget 复用	×	√	×	√	×	√

3.2 性能测试

为了测试RCMon对整个系统性能的影响,本文采用SPEC CPU 2006的CFP 2006基准测试集中的部分测试程序对系统开销进行了测试.鉴于RCMon基于Hypervisor实现,本文分别测试了裸机、Hypervisor、Hypervisor下开启关键系统调用监控、测试程序部署未优化的RCMon以及部署优化后的RCMon这5种条件下的性能开销,测试结果如图9所示.

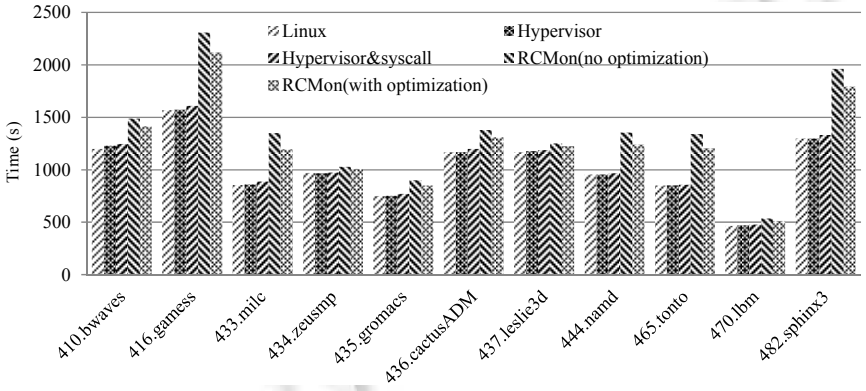

Fig.9 Results of performance overhead measurement

图 9 性能开销测试结果

由测试结果可知,本文所设计实现的Hypervisor由于采用了硬件辅助虚拟化技术Intel VT,并且尽可能地减少不必要的监控,使得该监控器在不监控内核函数时开销很小.当增加对关键系统调用的监控后,由于所有对关

键系统调用的执行都会产生陷入,并在陷入后执行对当前关键系统调用执行状态的更新和对函数入口原有的两条指令的模拟,因此开销略有增加.在对测试程序部署 RCMon 之后,由于不仅要监控系统中的所有关键系统调用,还要监控程序执行到每个关键节点处的陷入,并在陷入后根据当前节点信息到运行特征库中查找是否存在匹配项,当存在匹配项时,还要根据运行特征设置 WDC 对关键系统调用进行实时监控,因此,RCMon 引入后,测试程序的运行开销增加较多.与裸机(Linux 纯净系统)相比,当采用未优化的 RCMon 时,平均开销约为 31%(其中,434.zeusmp 开销最小,为 6%;而 465.tonto 开销最大,为 58%);当采用优化后的 RCMon 时,平均开销降至 22%,这也进一步验证了第 2.3 节中优化方法的有效性.

RCMon 的开销小于传统的内存安全保护方法(例如,SoftBound[24]的平均开销为 67%,Baggy Bounds Checking[25]的平均开销为 60%)、数据流完整性保护方法 DFI[30](平均开销高达 104%)以及控制流完整性方法 CCFI[15](平均开销 52%)等多类防御方法,略高于 CFI[14](平均开销 16%)以及 ASLR[6](的平均开销为 10%)等方法,但 RCMon 除了能够有效防御代码复用攻击以外,对所有含有系统调用恶意执行的攻击都具有一定的防御能力,防御的攻击类型更广,能够比这些防御方法提供更高的安全性.

4 RCMon 的局限性

由于 RCMon 不针对某个或某些具有特定特征的代码复用攻击类型进行防御,而是基于程序的系统调用运行特征防御代码复用攻击,是基于攻击的某种共性(即攻击者在实施恶意操作时难免要通过关键系统调用和操作系统进行交互)实施的检测和防御,因此无论是已知的还是未知的代码复用攻击,只要其实现过程包含对关键系统调用的使用或者其实现过程对原程序的关键系统调用执行过程造成了影响(即破坏了原程序的运行特征),RCMon 就能检测和防御.但 RCMon 仍然存在一些局限性.

(1) 如果一个代码复用攻击不使用任何关键系统调用且攻击实施过程不对原程序的关键系统调用执行过程产生任何影响,则可以绕过 RCMon 的检测,从而产生漏报.但这种情况下,代码复用攻击的发挥空间是非常有限的.首先,不借助任何关键系统调用的攻击所能实施的操作非常有限;另外,代码复用攻击的核心仍是对程序控制流的改变,而控制流改变的情况下要保证原程序关键系统调用执行过程不发生任何改变,这也使攻击变得更加难以实现.可见,虽然 RCMon 存在漏报某些攻击的可能,但该类攻击的危害性和实现的可能性都非常小,对 RCMon 的整体安全性的影响很小.而在第 3.1 节的测试中未发现任何漏报,这也在一定层度上验证了上述分析.

(2) 当在训练阶段构造的目标程序运行特征不够完备时,RCMon 可能会产生误报.由表 4 的测试结果可知,通过不断加大训练阶段的训练量,能够在较大程度上降低 RCMon 误报的可能.但这种方法效率低,而且存在很多不确定性.要在最大程度上消除误报,必须保证在训练阶段能够遍历目标程序所有的执行路径来构造完备的程序运行特征.拟在下一步工作中,利用动态符号执行技术来解决这一问题.动态符号执行能实现对程序的高路径覆盖,它以某个具体的输入来执行目标程序,并获取当前路径中的所有路径约束条件;然后通过对这些约束条件的修改,生成一条新的路径的约束条件,并用约束求解器获得一个新的输入;然后再以该输入执行目标程序.通过反复执行上述过程,即可实现对被测试对象所有路径的动态遍历.随着训练过程中的路径覆盖率的提高,就可构建更加完备的程序运行特征,从而消除误报.

5 总 结

针对当前代码复用攻击防御方法中存在的防御类型单一等问题,本文提出一种基于程序运行特征监控的代码复用攻击防御方法 RCMon.该方法首先定义并构造了程序的运行特征模型 RCMod(该模型包含了在程序中的函数调用位置前后和函数出入口这 4 类关键节点处的关键系统调用的运行特征信息),并基于该模型向目标程序的关键节点处植入监控代码,以实现基于 Hypervisor 的对目标程序运行过程的监控;然后,通过训练过程得到目标程序的运行特征库;最后,在实际运行过程中监控程序运行特征是否与事先得到的运行特征库中对应

特征一致,来判断程序是否遭到代码复用攻击,从而实现对代码复用攻击的有效防御.

RCMon 的验证过程包含基于特征库匹配的后向安全验证 BSV 和基于 WDC 的前向安全验证 FSV,分别实现了阶段性的安全验证和一个阶段内的实时安全验证,能有效防御函数级和指令级代码复用攻击,加强了目标程序的安全性.测试结果表明,RCMon 能够有效防御各类含关键系统调用执行过程的代码复用攻击,且实现过程不需要源码,实用性较好.另外,方法的平均性能开销也在可接受的范围内,约为 22%.

致谢 在此,由衷地感谢对本文研究工作提供基金支持的单位和参与本文评阅的审稿专家,同时向 Dyninst 研发团队以及其他为本文提供研究基础的前辈致敬.

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

[3] 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]

[4] 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]

[5] 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]

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

[7] 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]

[8] Gupta A, Habibi J, Kirkpatrick MS, Bertino E. Marlin: Mitigating code reuse attacks using code randomization. IEEE Trans. on Dependable and Secure Computing, 2014,12(3):326−337. [doi: 10.1109/TDSC.2014.2345384]

[9] 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]

[10] 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]

[11] 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]

[12] 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]

[13] 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]

[14] 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]

[15] 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]

[16] 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]

[17] 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]

[18] 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

[19] Cheng YQ, Zhou ZW, Yu M, Ding XH, Robert HD. ROPecker: A generic and practical approach for defending against ROP attacks. In: Proc. of the 21st Annual Network and Distributed System Security Symp. San Diego: The Internet Society, 2014. [doi: 10.14722/ndss.2014.23156]

[20] Göktas 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 Conf. on Security Symp. San Diego: USENIX Association, 2014. 417−432. https://www.usenix.org/system/files/conference/usenixsecurity14/sec14-paper-goktas.pdf

[21] 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. Hong Kong: ACM Press, 2011. 40−51. [doi: 10.1145/1966913.1966920]

[22] Wicherski G. Taming ROP on Sandy Bridge. SyScan, 2013. https://infocon.org/cons/SyScan/SyScan%202013%20Singapore/SyScan%202013%20Singapore%20presentations/SyScan2013_DAY1_SPEAKER05_Georg_WIcherski_Taming_ROP_ON_SANDY_BRIDGE_syscan.pdf

[23] 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

[24] Nagarakatte S, Zhao J, Martin MMK, Zdancewic S. SoftBound: Highly compatible and complete spatial memory safety for C. ACM SIGPLAN Notices, 2009,44(6):245−258. [doi: 10.1145/1543135.1542504]

[25] Akritidis P, Costa M, Castro M, Hand S. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In: Proc. of the 18th USENIX Security Symp. Montreal: USENIX Association, 2009. 51−66. https://www.usenix.org/legacy/event/sec09/tech/full_papers/akritidis.pdf

[26] 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]

[27] 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]

[28] 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

[29] 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 36th IEEE Symp. on Security and Privacy. San Jose: IEEE CS Press, 2015. 781−796. [doi: 10.1109/SP.2015.53]

[30] Castro M, Costa M, Harris T. Securing software by enforcing data-flow integrity. In: Proc. of 7th Symp. on Operating Systems Design and Implementation. Seattle: USENIX Association, 2006. 147−160. https://www.usenix.org/legacy/event/osdi06/tech/full_papers/castro/castro.pdf

[31] 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 CS Press, 2015. 763−780. [doi: 10.1109/SP.2015.52]

[32] 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]

[33] King M, Dave N. Automatic generation of hardware/software interfaces. In: Proc. of the 17th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems. London: ACM Press, 2012. 325−336. [doi: 10.1145/ 2150976.2151011]

[34] Intel. Intel 64 and IA-32 architectures software developer's manual. 2016. http://www.intel.cn/content/dam/www/public/us/en/documents/ manuals/64-ia-32-architectures-software-developer-manual-325462.pdf

[35] Wang XL, Wang ZL, Sun YF, Liu Y, Zhang BB, Luo YW. Detecting memory leak via VMM. Chinese Journal of Computers, 2010, 33(3):463−472 (in Chinese with English abstract). [doi: 10.3724/SP.J.1016.2010.00463]

[36] 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 the 14th IEEE Trustcom/BigDataSE/ISPA. Helsinki: IEEE CS Press, 2015. 190−197. [doi: 10.1109/Trustcom.2015.374]

[37] Dyninst9.1.0. 2016. https://github.com/dyninst/dyninst/releases

[38] Das S, Liu Y, Zhang W, Chandramohan M. Semantics-based online malware detection: Towards efficient real-time protection against malware. IEEE Trans. on Information Forensics and Security, 2017,11(2):289−302. [doi: 10.1109/TIFS.2015. 2491300]

[39] Bosman E, Bos H. Framing signals—A return to portable shellcode. In: Proc. of the 35th IEEE Symp. on Security and Privacy. Berkeley: IEEE CS Press, 2014. 243−258. [doi: 10.1109/SP.2014.23]

附中文参考文献:

[9] 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]

[17] 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]

[35] 汪小林,王振林,孙逸峰,刘毅,张彬彬,罗英伟.利用虚拟化平台进行内存泄露探测.计算机学报,2010,33(3):463−472. [doi: 10.3724/SP.J.1016.2010.00463]

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

张平(1969－),女,博士,副教授,主要研究领域为并行识别,并行编译,信息安全.

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.

程三军(1966－),男,高级工程师,CCF 专业会员,主要研究领域为计算机图形图像,网络安全.