软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, [doi: 10.13328/j.cnki.jos.000000]

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

基于 IPT 硬件的内核模块 ROP 透明保护机制*

王心然[1],刘宇涛[1],陈海波[1]

[1](上海交通大学 并行与分布式系统研究所,上海 闵行 200240)

通讯作者: 王心然, E-mail: wangxinran498@gmail.com

摘 要: Return-Oriented Programming(ROP)是一种流行的利用缓冲区溢出漏洞进行软件攻击的方法,它通过覆写程序栈上的返回地址,使程序在之后执行返回指令时,跳转到攻击者指定位置的代码,因而违反了程序原本期望的控制流.控制流完整性(Control-flow Integrity,简称 CFI)检查是目前最流行的 ROP 防御机制,它将每条控制流跳转指令的合法目标限制在一个合法目标地址集合内,从而阻止攻击者恶意改变程序的控制流.现有的 CFI 机制大多用于保护用户态程序,然而当前已经有诸多针对内核态的攻击被曝出,其中 Return-oriented rootkits[33] (ROR)就是在有漏洞的内核模块中进行 ROP 攻击,达到执行内核任意代码的目的.相较于传统的基于用户空间的 ROP 攻击,ROR 攻击更加危险.根据 Linux CVE 的数据统计,在 2014-2016 年中,操作系统内核内部的漏洞有 76%出现在内核模块中,其中基本上所有被公布出来的攻击都发生在内核模块.由此可见,内核模块作为针对内核攻击的高发区,非常危险.另一方面,当前鲜有针对操作系统内核的 CFI 保护方案,而已有的相关系统都依赖于对内核的重新编译,这在很大程度上影响了它们的应用场景.针对这些问题,本文首次提出利用 Intel Processor Trace (IPT)硬件机制,并结合虚拟化技术,对内核模块进行透明且有效的保护,从而防御针对其的 ROP 攻击.实验表明该系统具有极强的保护精确性、兼容性和高效性.

关键词: ROP;CFI;内核模块;IPT

中图法分类号: TP311

Transparent Protection of Kernel Module against ROP with Intel Processor Trace

WANG Xin-Ran[1], LIU Yu-Tao[1], CHEN Hai-Bo[1]

[1](Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University, Minhang 200240, China)

Abstract: Return-oriented programming (ROP), in which attackers corrupt program stack in order to hijack the control flow of the program, is a popular way to attack memory corruption bugs. Control flow integrity (CFI) is a popular approach which thwarts attackers tampering with execution flow, in a way that enforces the legal targets of each indirect branches. While published CFI approaches mainly focus on protecting user programs, the OS kernel is still vulnerable to various attacks. E.g, return-oriented rootkits(ROR), which can launch ROP attacks in vulnerable kernel modules, is able to execute arbitrary code in kernel. Compared with traditional user-level ROP, ROR is more dangerous because it happens in kernel space. According to Linux CVE from 2014 to 2016, 76% of kernel bugs appear in kernel module and almost all of the published attacks happen in kernel modules, which infers that kernel modules happen to be the most dangerous area in the kernel space. However currently there are still very few number of kernel-level CFI protection mechanisms, and all of the existing ones require source-code level modification and kernel recompilation, which restricts the usage scenarios of the commodity systems. Facing of these problems, we propose to leverage Intel Processor Trace(IPT), and present the first system which can prevent against ROP attacks in kernel modules base on virtualization without relying on the source code of kernel and kernel modules. The evaluation proves the precision, transparency and efficiency of our system.

Key words: ROP;CFI;kernel module;IPT

收稿时间: 0000-00-00; 修改时间: 0000-00-00; 采用时间: 0000-00-00; jos 在线出版时间: 0000-00-00
CNKI 在线出版时间: 0000-00-00

系统安全一直是国内外学者重点研究的领域,围绕系统安全展开的攻防战也层出不穷.Return-Oriented Programming(ROP)是其中比较重要的一种攻击方法,它通过攻击程序存在的缓冲区溢出的漏洞,来覆写程序的栈上的返回地址,使得程序在之后执行返回指令时被迫跳转到攻击者指定的位置的代码,从而使得攻击者可以执行任意代码,进而窃取用户数据,篡改系统代码执行流等等.由于 ROP 攻击利用的是现有的代码,没有注入新的代码,因此它可以绕过数据执行禁止保护(DEP).围绕 ROP 攻击,研究人员提出了一系列防御机制,其中控制流完整性(Control-flow Integrity ,CFI[1])保护是最流行的一个方向.由于 ROP 攻击需要将被攻击者的控制流重定向到攻击者指定的位置,而 CFI 可以阻止控制流的任意跳转.例如最早的 CFI 规则保证返回指令被限制到只能跳转到一条 call 指令的后面,而 call 指令被限制只能跳转到一个函数的开始.

然而，这种粗粒度的 CFI(Coarse-grained CFI)机制仅能用于防备最基本的 ROP 攻击,通过精心地构造栈和选择函数,攻击者仍然能够执行任意代码.因此研究人员提出一系列细粒度的 CFI(Fine-grained CFI)机制.它们使用程序控制流图(Control-Flow Graph,CFG)来更加细粒度地限制每个控制流跳转指令可以跳转的目标地址,从而阻止较为高级的 ROP 攻击.

除了用户态程序,操作系统内核同样也被暴露在 ROP 攻击的风险之下.内核代码行数远超应用程序,出现漏洞从而被攻击者利用的可能性也随之增加.对此研究者们提出了一系列保护内核完整性的机制[5],它们可以确保内核代码不会被攻击者恶意修改.例如,通过给内核模块添加签名,并在内核模块加载时检查签名的合法性,可以确保加载的内核模块是没有被修改过的.或者通过设置 W^X 保护,限制任意内存区域不能同时为可写和可执行状态,来防止攻击者在运行时注入并执行恶意代码.这些保护方法可以在一定程度上保护内核不受代码注入的攻击,但是它们都不能防御利用已有代码的 ROP 攻击.

另一方面,内核模块作为内核地址空间的重要组成部分,在很大程度上决定了内核的功能性和安全性.例如,当前绝大多数的设备驱动都是以内核模块的形式运行,而这些驱动程序一般是由第三方的厂商自行编写,对于内核维护人员来说,极难保证它们的质量,因此在这些内核模块中极易产生漏洞,从而被攻击者利用.事实上,在 2014 年到 2016 年间,Linux 内核被公布出的所有漏洞(CVE)中,有 76%发生在内核模块.同时,所有在 2016 年公布的针对 Linux 内核的攻击基本上全部发生在内核模块中.而所有的这些攻击中,针对内核模块的 ROP 攻击可以绕过内核完整性检查,破坏整个内核程序的控制流.例如,Return-Oriented Rootkits[33](ROR)就是通过利用内核模块的漏洞进行相应的 ROP 攻击,具有极强的破坏性.因此可以说,内核模块作为 Linux 系统漏洞的高发地,保护其不受 ROP 攻击显得至关重要.

针对基于内核模块的 ROP 攻击,研究人员提出了一些保护内核控制流完整性的方案.[3][4] [6] [7] [34],但是他们都依赖于分析甚至修改内核的源码,并且需要在部署方案之前对内核进行重新编译.这在很大程度上限制了这些保护机制的应用场景.例如,在云服务平台中基于 IaaS 的服务模式下,租户可以向云服务提供商购买资源来运行自建的虚拟机.在该应用场景中,云服务提供商无法获取内核源码并对其进行重新编译.另一方面,由于大部分内核模块都是由第三方直接提供二进制可执行文件,而无法获得相应的源码并进行重新编译.因此,如何实现一套基于二进制的防御内核模块 ROP 攻击的透明保护机制是一个值得探讨的问题.针对这个问题,本文首次提出利用 Intel Processor Trace (IPT)硬件机制,并结合虚拟化技术,对内核模块进行透明且有效的保护,从而防御针对其的 ROP 攻击.

在实现本系统的过程中有很多挑战.针对基于二进制的内核模块分析,我们提出了一套静态分析框架,结合内核模块的特点对其进行细粒度的分析;针对 IPT 解码慢的问题,我们参考 FlowGuard[30]的做法,将 CFG 重构为与 IPT 记录格式相匹配的 ITC-CFG,加速解码匹配过程;针对检查控制流时机的问题,我们修改了扩展页表错误处理函数,实现了一套滑动窗口(sliding window)的机制,在内核模块和内核核心切换的过程中及时触发控制流完整性的检查.经过实践,本系统可以有效防御针对内核模块的 ROP 攻击,并具有极强的保护精确性、兼容性和高效性.

本文做出了如下贡献:

- 首次提出了利用 IPT 硬件特性的内核模块 ROP 攻击透明防护机制,利用虚拟化技术,不依赖内核的源

码,不需要重新编译内核.

- 提出了一套细粒度生成内核模块控制流图的静态分析方法.
- 在 Linux 内核上成功部署了本系统,并经过测试表明,本系统可以精确、透明、高效地防护发生在内核模块中的 ROP 攻击.

本文第 1 节介绍了攻击的背景及国内外相关的研究现状,以及 IPT 硬件的特点.第 2 节总体介绍了本系统的架构,以及攻击的威胁模型和假设.第 3 节介绍了如何分析内核模块,并生成控制流图.第 4 节介绍了如何在运行时触发并检查内核模块的控制流完整性.第 5 节对本系统的安全性和性能进行了评测.第 6 节对本系统的局限性进行了讨论.第 7 节对全文进行了总结.

8 背景及国内外研究现状

8.1 面向内核的攻击

Return-Oriented Programming(ROP)攻击是一种控制流劫持攻击,通过内存操作上的一些漏洞(如,缓冲区溢出)来覆写程序栈上的返回地址,使得在下次执行返回指令时,程序的控制流会跳转到攻击者希望的目标地址.如果目标地址处是以返回指令结尾的代码片段,则又会从被覆写的栈上读出另一个返回地址,再次跳转到攻击者希望的下一个目标地址.攻击者用这种方式重用内存中已有的代码片段,并将小的代码片段串联起来,组合成攻击者想要的形式,可以达到执行任意代码的目的.ROP 攻击已被证明是图灵完备的.

内核一直暴露在攻击者的攻击之下,而面向操作系统的内核的攻击又是十分危险的,因为攻击者一旦攻破就可以以内核的权限执行任意内容.内核攻击的常见形式是修改关键数据以非法提高用户态进程权限,现有的内核完整性的保护机制可以防御这类攻击.攻击者为了绕过这些防御方法,开发出了新的攻击:Return-oriented Rootkits[33].它是一个面向内核的 ROP 攻击,通过利用在内核模块中出现的缓冲区溢出漏洞,来覆写内核栈上的返回地址,进行 ROP 攻击,重用内核中已有的代码,劫持控制流后在内核中埋下木马,永久影响内核的功能.该攻击可以完整的绕过基于内核完整性保护的方法.可以说内核完整性保护”和“控制流完整性保护”在内核安全机制方面是两个不同的方向，可以互相利用，但都无法被对方所取代。现在的防御 ROP 攻击的系统大多保护的是用户态程序,鲜有对内核及内核模块的保护.

8.2 控制流完整性(CFI)

控制流完整性(Control-Flow Integrity,简称 CFI),是一套可以防御控制流劫持攻击的理论.它在程序中加入检查指令,在每条间接转移指令前检查它的目标地址.间接转移指令指的是目标地址为动态确定的跳转指令,包括间接函数调用(如 call *%rax),间接跳转(如 jmp *%rax)和返回(ret)指令,可以阻止控制流劫持攻击,比如 ROP 及其变种.检查目标地址是否合法的方法有很多,最常用的是将间接跳转中用到的函数指针读出来,然后检查它是否真正指向之前定义的值.所以,对一个程序实行 CFI 保护的步骤包括:1、正确地计算每个间接跳转指令的合法跳转目标地址的集合;2、在这些间接跳转指令之前加上检查目标地址是否合法的指令;3、为各个合法目标地址加上合适的标签,方便检查.

然而,早期的 CFI 被认为是粗粒度(coarse-grained)的.它的检查策略十分宽松,如允许返回指令跳转到任意一条 call 指令之后,允许间接函数调用指令跳转到任意一个函数的开始.这种粗粒度的 CFI 可以保护系统不受最原始的 ROP 攻击,但是研究证明,精心构造的 ROP 攻击可以构造出符合这种保护机制的攻击链,这种粗粒度的 CFI 是不能防御 ROP 攻击的.为此,细粒度(fine-grained)的 CFI 被提出,它拥有更精细的检查机制,如限制每条返回指令只能返回到调用它的函数,限制每条间接函数调用指令只能跳转到一个事先定义的有限的函数集中,如所有和用到的函数指针的函数类型相匹配的函数.

可以看出,CFI 是一个很有吸引力的防止控制流劫持攻击的方案,但是实行 CFI 也有很多挑战.除了上面提到的安全性问题,还有在部署在实际环境中的性能问题.现在的 CFI 保护系统,通常都需要在这几个要素之间进行取舍:1、准确性,即这个 CFI 系统的保护必须足够精细,可以抵御足够多的 ROP 攻击,减小 ROP 攻击所能利

用到的攻击面;2、高效性,即部署这个系统对程序造成的额外性能开销不宜过大,否则将失去实用性;3、透明性,即系统需要对程序来说是不可见的,并且系统可以较为容易的部署在现存的操作系统和硬件上,和现有的大多数系统所兼容.

上文提到的在程序中的跳转指令前添加检查指令的方法,即使用二进制覆写来实施保护的方法在一定程度上保护了系统的透明性,然而这种方法不可避免的会破坏程序二进制文件的完整性,会导致和一些现存的安全保护机制不兼容(如,Windows 7 系统的共享库保护和远程监控系统).基于二进制的,即不依赖程序源代码的CFI 保护系统无法提供很精细的保护策略,每条间接跳转指令都有一个较大的合法跳转目标地址集合,系统的安全性会牺牲一部分,更容易遭受到攻击.与此相对,很多基于编译器的保护,即依赖程序的源代码的系统,可以实施更精细的保护策略,如对返回指令执行影子堆栈的保护,对间接函数调用指令实行更详细的基于源代码的分析.但是,它们不支持共享库,因为我们无法拿到共享库的源代码;同时它们也不支持已部署好的应用程序.所以,我们更希望一个透明的、基于二进制但并不将其修改的、精确的、高效的保护系统.

现在已有很多 CFI 系统被提出,如[13-16]皆为基于编译器的 CFI 系统,[8],[11],[17-21]均为基于二进制的CFI 系统.利用硬件来构建 CFI 系统也是比较流行的选择,如[22],[23]都设计了新的硬件保护 CFI,而[24-30]都是利用已有的硬件来实行 CFI 的保护.本文也是利用现有的 IPT 硬件来实现了 CFI.

但是,上面提到的系统均为对用户程序的保护,对内核的控制流完整性保护少之又少.在对内核的保护上,HyperSafe[2]在虚拟机管理器上实现了对间接函数调用的 CFI 保护,但是它只保护 hypervisor,不会保护在hypervisor 上运行的操作系统中的发生的控制流劫持攻击,同时也不能防御 ROR 攻击.KCoFI[3]使用了粗粒度的CFI 策略保护了 FreeBSD 中的操作系统,但是仍然会被精心制造的 ROP 攻击所攻破.第一个细粒度保护内核控制流完整性的系统是由 Ge et al[9]提出的,但是由于一些限制使得这个系统只能支持 FreeBSD 和 MINIX 的内核.PaX RAP [10] 使用了加密返回地址和严格的函数原型比对的方式来实施细粒度的 CFI 保护.一个基于硬件虚拟化的内核完整性监控方法[34]使用虚拟化方法来保护内核的完整性和控制流完整性,并基于 Xen 加以实现.但是,如上所提到的系统无一例外的需要内核的源代码进行分析,并在一定程度上加以修改,再重新编译内核.这违反了我们想要透明性的保护的目标.

8.3 Intel Processor Trace(IPT)

Intel Processor Trace (IPT) 在 Intel Core M 和第五代 Intel 处理器被引入,并在现在的主流英特尔处理器中均有配备.每个处理器核都有它自己的 IPT 硬件部分,并将程序的控制流用包的形式记录下来.注意,具有管理员权限才能配置 IPT.管理员可以设置将其开启或关闭,也可以设置只在满足一定条件时记录程序的控制流.IPT 记录的包被存储在提前设置好的内存区域内.当这块内存区域被填满时,硬件会发出一个中断,通知软件进行处理.软件解码器通过将记录的包和一些其他信息如程序的二进制文件相结合,来精确还原出程序原本的控制流.被 IPT 记录的包是经过高度压缩的,并直接写入物理内存,不经过 TLB 和缓存,所以 IPT 在运行时带来的额外性能开销非常小.

Table 1 An example of IPT

表 1 IPT 监测控制流的例子

编号	执行流	IPT 记录的包
1	0x400100 callq fun1	
2	0x400106 jg 0x400150 // not taken	TNT(0)
3	0x40010a jmpq 0x400150	
4	0x400150 ...	
	fun1:	
5	0x400200 jmpq *%rax // %rax = 0x400300	TIP(0x400300)
6	...	
7	0x400300 cmp %rsi, %rax	
8	# timer interrupt	FUP; TIP(handler)
9	0x400304 jg 0x400400 // taken	TNT(1)

	...	
10	0x400400 leaveq; retq	TIP(0x400106)

表 1 举例展示了 IPT 如何记录被监测程序的控制流.IPT 只在执行目标地址不能静态确定的控制流跳转指令的时候才记录包,在执行非条件的直接转移指令时不会记录包.在表 1 中第 1 条指令为直接函数调用(callq),第 3 条指令为直接跳转(jmpq),它们都属于非条件直接转移,不会有包被记录.每个条件跳转指令(jg, je 等)会产生一个 TNT 包,这个包只有 1 个比特,代表这个条件跳转有没有被采用.记录 1 则代表条件跳转分支被采用(如指令 9),0 则代表这条跳转没有生效(如指令 2).间接转移指令(包含 callq *, jmpq *, ret)会产生一个 TIP(Target IP)包,包内记录了指令跳转的目标地址,如指令 5 和指令 10.此外,如果该跳转为一个长跳转(中断,虚拟机下陷等),则会在生成一个 TIP 包的同时生成一个 FUP(Flow UPdate)包,表示这是一个长跳转,如系统在指令 8 处收到一个 timer interrupt,则在生成包含中断处理函数的地址 TIP 包的同时,会生成一个 FUP 包.每运行一条指令,IPT 记录的包平均大小小于 1 比特.

此外,管理员可以配置很多参数来决定 IPT 记录包的条件,如当前执行任务的特权级别 (Current privilege level, 简称 CPL),页表基地址寄存器 (CR3),程序的指令指针 (Instruction pointer, 简称 IP)的范围等.使用 IPT 时,可以很方便的过滤出目标程序产生的所有包.由于我们只关心内核模块和内核的控制流执行情况,而它们都是运行在 CPL 0 特权级别下的,所以我们将 IPT 记录包的条件设置为 CPL 等于 0,避免记录大量的我们不关心的用户态的控制流跳转情况.

IPT 还可以记录更复杂的控制流转移情况,如进程之间的上下文切换、中断的处理等.在当前的页表基地址寄存器(CR3)进行切换时,IPT 会记录一个 PIP(Page Information Packet)包,来记录切换后的 CR3.PIP 包中有一个比特为 Non-Root 位,用来标记切换 CR3 之后是否处于 Root 模式下,以区分当前是否运行在虚拟机中.此外,在系统发生中断时,IPT 会记录一个 FUP(Flow Update)包,表示控制流发生了长跳转,之后会紧接着记录一个 TIP 包来记录这个长跳转的目标地址.

然而,在监测记录控制流时被节省下来的额外开销被移到了解码时.由于记录的包是经过高度压缩的,我们必须使用软件解码,将收集到的包和程序的二进制相结合,才能重新构造出程序准确、完整的控制流.这是十分繁琐并耗时的操作,使用 IPT 在 SPECCPU 2006 上进行测试时,如果当存储 IPT 的包的内存区域被填满的时候就停止程序执行然后将收集到的包结合程序的二进制解码,会造成 200-500 倍的额外性能开销.

IPT 提供了很多有利于监测控制流的功能,并且在记录控制流时有优秀的性能.同时,解码和还原控制流的低性能也为 CFI 带来了不小的挑战.这是因为 IPT 本身设计的目的是用于线下分析,如软件故障排除等,它们对线下解码和还原控制流的时间没有过多要求,只要保证线上软件运行时的性能.所以,为了在 CFI 中使用 IPT 的迅速、完整的记录控制流的优势,我们必须解决解码慢的问题.

9 系统设计综述

本系统目标为保护虚拟机中的内核模块不会受到来自攻击者的 ROP 攻击,保证被保护的内核模块的控制流不被恶意篡改.为了达成这一目的,我们需要先在开启保护前对被保护的内核模块进行基于二进制的静态分析(不需要依赖内核模块的源代码)来生成程序的控制流图(Control flow graph, 简称 CFG),之后在模块运行的时候动态检查模块的控制流是否合法.为了解决 IPT 的解码慢导致检查控制流时花费时间过长的问题,本系统先生成传统的 CFG,然后将传统的 CFG 进行重构,使之符合 IPT 记录的包的样式,来加速解码检查控制流的过程.本系统可以精确、高效、透明地保护内核模块不受 ROP 攻击的影响.

9.1 程序控制流图的生成

传统的静态分析中,生成的 CFG 是把程序以基本块(Basic block)为单位分割,将所有控制流跳转指令以边的形式体现出来.这些边连接了跳转的源地址和目标地址.在程序运行时,只有被边相连的两个基本块才能进行

遵从边的方向的控制流跳转.一个 CFG 中包括了被分析程序的所有可能被执行到的控制流跳转指令,这些跳转指令可以组合出程序中所有可能执行的控制流.CFG 被目前的绝大多数的保护 CFI 的系统所使用,是保护 CFI 的基本策略.

然而,我们使用的硬件 IPT 只记录间接转移指令和条件跳转指令,直接转移指令是不被记录的.因此,使用传统的 CFG 时,我们必须将 IPT 记录到的包和程序的二进制结合起来进行解码,还原程序的完整的控制流之后,才能与传统 CFG 进行比对.如上节所述,这会造成非常大的额外开支.为此,本系统将参考 FlowGuard[30]中的设计,将传统的 CFG 进行重构,删去传统 CFG 中所有直接转移指令的边,并将剩余的间接转移指令的目标基本块连接起来.为了方便,我们称它为 ITC-CFG (Indirect targets connected CFG).经过这种重构操作后的得到的 ITC-CFG 中包含的信息量和之前的 CFG 是完全等价的:它不会将合法的控制流误报为非法的,也不会漏报一个非法的控制流,认为其合法.重构之后的 ITC-CFG 是由间接转移指令的目标基本块相连的图,和 IPT 只记录间接转移指令的目标地址的特性相符;在动态控制流检查时,可以直接将 IPT 记录的包和 ITC-CFG 进行比对,不需要再依赖程序的二进制进行解码,大大减少了额外性能开支.

9.2 控制流检查

本系统的目标是保护内核模块的控制流完整性,所以本系统选择在控制流跳出内核模块时进行控制流的检查,确保在进入内核核心之前的控制流是合法的,保证在受到控制流劫持攻击时,系统受到实质危害前可以检测到攻击并加以应对.在检查时,本系统从内存中读出 IPT 记录的包,然后根据其中 PIP 包中包含的信息,从中挑选出虚拟机内核模块中产生的包,再将其与 ITC-CFG 进行比对,检查控制流是否合法.如果比对失败,则本系统认为虚拟机遭到了 ROP 攻击,中止虚拟机运行并依据实际情况选择后续处理方案(如根据用户或管理员的意愿重新部署等).

9.3 系统架构综述

本系统的架构如图 1 所示.

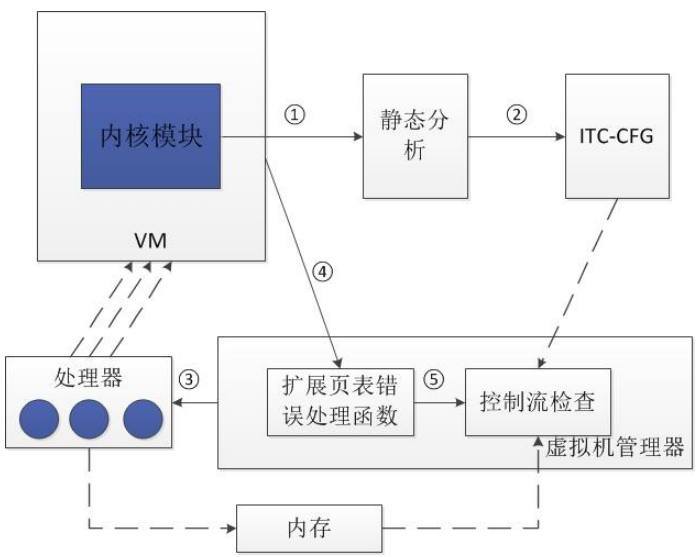

Fig.1 System design

图 1 系统架构

被保护的内核模块的二进制被传入静态分析器中进行分析(步骤 1),生成 CFG 后经过重构处理最终生成

ITC-CFG(步骤 2).生成的 ITC-CFG 存储在虚拟机监控器(Hypervisor)中,等待运行控制流检查模块时使用.之后,虚拟机监控器在处理器中配置 IPT 并使其开始记录被保护的虚拟机的控制流(步骤 3),并将记录的包存入内存中.为了在控制流从被保护的内核模块跳出时能够被检测到,本系统修改了扩展页表错误处理函数(EPT violation handler),使控制流在内核模块和内核核心中进行切换时,会触发一个扩展页表错误(EPT violation).这个扩展页表错误会触发调用修改过的扩展页表错误处理函数(步骤 4).在修改过的扩展页表错误处理函数中,会触发控制流检查(步骤 5),将 IPT 记录的包和 ITC-CFG 进行比对.如果比对失败,则判定为检测到攻击,进行后续处理.

9.4 威胁模型和假设

本系统防御的是攻击者针对内核模块的 ROP 攻击.具体举例来说,一个攻击者知道某台虚拟机上的某个内核模块的详细构造,知道其中存在缓冲区溢出漏洞.之后攻击者通过和虚拟机进行远程交互,精心构造和虚拟机之间交互的信息,来触发虚拟机的内核模块中的缓冲区溢出漏洞,实行 ROP 攻击,从而劫持、篡改虚拟机的控制流.

本系统假设虚拟机管理器和内核核心本身是可信的,针对它们的攻击不在本文讨论的范围内;被保护的内核模块是非恶意的,但是会有漏洞供攻击者利用.此外,本系统认为静态分析生成 ITC-CFG 的过程是可信的.最后,本系统认为被保护的内核模块没有在加载后修改自身代码的逻辑,这在目前市面上的内核模块中都是没有出现过的.这样可以保证静态分析生成的 ITC-CFG 是可信的、有效的.

10 控制流图的生成

本系统通过静态分析,以被保护的内核模块的二进制作为输入,生成它的 CFG,并参考 FlowGuard[30]的做法,将其重构为与 IPT 记录包的形式兼容的 ITC-CFG,以用于运行时控制流的检查.

10.1 传统的CFG的生成

程序控制流图(Control Flow Graph, CFG)是一个由点和有向边组成的图,代表了一个程序在运行时控制流可能经过的所有路径.在 CFG 中,每个点代表一个基本块,即一段在中间不包含控制流跳转指令,也不包含其他控制流跳转指令的目标的连续代码.控制流跳转指令作为基本块的最后一条指令,而它们跳转的目标则作为基本块的第一条指令.有向边代表了一条控制流跳转指令产生的控制流转移,将跳转指令所在的基本块和跳转的目标所在的基本块连接起来,由源基本块指向目标基本块.利用 CFG,我们可以知道一个程序所拥有的所有控制流转移指令所可能跳转到的目标地址集合.

本系统通过静态分析来生成 CFG,用于标识每一条跳转指令都有哪些合法的目标地址.在静态分析时,我们只需要分析被保护的内核模块的二进制,而不需要该模块的源代码或者是内核核心的二进制和源代码.和之前的 CFI 工作相同,并参考[11],[12],本系统生成的 CFG 是保守的,即每个间接转移指令的合法目标地址的集合可能会多于实际会跳转的目标.这是因为只使用二进制而不依赖于源代码进行分析时,没有足够的信息来精确分析每个间接转移指令的合法目标地址.为了避免将合法的控制流误报为非法,在分析时必须将所有可能的跳转目标地址都标为合法,来保证被保护的内核模块可以正常运行.

在对用户态应用程序进行分析的时候,由于它会依赖一些库(如 libc),我们不能仅仅分析单个程序的二进制,还需要找到它所依赖的所有库,并一并分析所有这些库的二进制,将分析的结果汇总进一个大的 CFG 中.然而,本系统保护的是内核模块,内核模块并没有依赖的库,它只依赖于内核核心.所以我们只需要分析被保护的内核模块的二进制即可.

对程序静态分析的过程,就是一条一条地处理被分析程序的二进制,根据当前处理的指令种类来执行不同的操作.分析的过程就是明确出程序的所有的基本块和跳转边.二进制中包含有函数表,可以得到其中包含的所有函数及它们的地址.分析从二进制的每个函数的第一条汇编指令开始逐条处理,在遇到非跳转指令时,只需要简单的将这条指令包括到当前的基本块中即可.在遇到跳转指令时,需要将这条跳转指令添加到当前的基本块

中,然后结束当前基本块并准备建立一个新的基本块;然后寻找这条跳转指令所有可能的目标地址,再将这些目标地址标记为一个跳转目标.如果某个跳转目标在一个基本块的中央,则将这个基本块以这个目标为界一分为二,使得跳转目标为一个新的基本块的开始.

在分析并生成CFG的过程中,每遇到一条跳转指令时,为了使合法的控制流转移不被误报成非法的,必须在构建CFG时将其所有可能的跳转目标地址找到,并以边的形式将这条跳转指令在CFG中体现.下面将分段叙述如何处理各种不同种类的控制流跳转.

(1) 直接转移指令

直接转移指令包括目标地址为一个固定值的直接函数调用指令(call)、直接跳转指令(jmp).它们的共同特点是目标地址被固定地写在了二进制中,所以我们可以直接从二进制中拿到它们的目标地址,然后将这一跳转作为一条边加入CFG中.

然而,在内核模块的二进制被加载到内核中之前,有些目标地址并不能这样获取.由于内核模块是动态加载的,所以一些调用内核核心中函数的指令具体目标地址在编译内核模块时是未知的.这些地址依赖于内核,存储在内核的符号表中.在编译时,这些直接转移指令的目标地址被填为0,用来占位.在模块加载到内核中时,内核会去查找内核中的符号表,并告诉内核模块,再将这些占位符替换为在内核中真正的目标地址.如果在内核模块文件加载到内核中之前分析它的的二进制,就会得不到这些跳转指令的目标地址(会读出0).所以,本系统选择在模块加载完成后,从虚拟机的内存中读出被加载后的内核模块的二进制,然后再对其进行静态分析.这样一来,静态分析处理的就是完整的、可执行的内核模块,可以生成完整的CFG.

(2) 间接函数调用指令

间接函数调用(Indirect call)指的是一条目标地址依赖于某个寄存器值的call指令,如call *%rax,一般由通过函数指针的函数调用编译而来.它们的目标地址是在运行时动态地根据当时的某些值决定,因此在静态分析时不能分析出它们的准确的目标.所以我们必须找到所有的可能的目标地址,来确保生成的CFG是完整的、保守的.然而,内核模块中的间接函数调用的目标可能是一个内核核心中的函数.在我们不依赖内核核心的二进制的情况下,是无法找到内核核心中的目标函数的.目前本系统的目标是防御通过内核模块攻击内核的ROP攻击,因此我们不需要关心内核模块调用内核核心中函数的情况,只需要确保内核模块通过执行返回指令进入内核核心时控制流的完整(详见第4.2节).但是,我们仍然需要防止攻击者通过修改函数指针,来非法调用内核模块内部的其他函数.因为攻击者可以通过非法调用内核模块内部的函数来修改一些寄存器的状态,然后再返回到内核核心中合法的地址,让内核核心继续执行时使用被攻击者控制的寄存器.具体来说,如果一个函数地址出现在如下地方之一,则我们认为它是一个合法的间接函数调用的目标地址,所有的间接函数调用都可以调用它:

- 一个出现在内核模块二进制的.text段的函数地址.如果一个函数地址被直接赋值给一个函数指针,那么这个地址就会出现在.text段.
- 一个出现在内核模块二进制的.data段的函数地址.如果一个函数指针从一个数据结构或变量中读取了一个函数地址,那么这个地址会出现在.data段.

在静态分析过程中,本系统扫描模块的二进制,从中提取满足上述某个条件的函数地址,将它们加入到间接调用目标列表中.每个间接函数调用都被允许调用列表中所有的函数.

(3) 返回指令

对于返回即return指令,一般来说,我们只需要在向CFG中加入函数调用的边时,同时加入相应的返回边,即将目标函数的所有返回指令都和调用这个函数的call指令的下条指令相连.

然而,现在的编译器大多使用的尾调用优化(tail-call optimization)会破坏这种简单的一对一的关系.尾调用优化指的是,如果一个函数B的结尾是一个函数调用指令,调用了函数C,那么它可能会被替代为一条直接跳转指令,跳转到函数C的开始.在函数C中重用了函数B的栈,看起来就像函数C是被函数B的调用者,即函数A调用的.在函数C返回时,会直接返回到函数A,节省一次函数调用和返回的开销.然而,函数A实际上并没有调用函数C,我们在静态分析时无法分析出函数C会返回到函数A.本系统参考现有的方法进行处理:在分析每个

函数时(例如,函数 A),计算它接下来所有可能的控制流,找到所有可以被这个函数执行的、在另一个函数中的(例如,函数 B)、并且目标地址为第三个函数的首地址(例如,函数 C)的跳转指令.找到这些跳转指令后,将他们的目标函数的所有返回指令的地址和函数 A 中调用函数 B 的 call 指令的地址在 CFG 中相连,表示函数 C 的所有返回指令被允许直接返回到函数 A 中调用函数 B 的地方.

(5) 间接跳转指令

间接跳转(Indirect jump)指的是一条目标地址依赖于某个寄存器值的 jmp 指令,如 jmp *%rax.有两种场合会使用到间接跳转:一种是间接函数调用和尾调用优化的结合,我们用(2)和(3)中已经提到的方法分析它;另一种是由 switch-case 语句编译成的跳转表.跳转表被存储在.rodata 段中,我们使用间接跳转指令中的相对偏移计算出跳转表在.rodata 段中的位置,然后读到跳转表包含的目标地址,再将这些目标地址在 CFG 中与间接跳转指令相连.

10.2 CFG的优化与重构

对以上方法生成的 CFG,我们通过 TypeArmor[32]的做法,通过匹配函数调用时的参数数量来进一步限制间接函数调用的目标地址集合.例如,在 x86 架构上,通过基于二进制的静态分析,我们分析一个函数中对参数寄存器的读操作,从而推断出这个函数消耗了多少个参数;而函数调用指令在执行之前,往往需要通过写参数寄存器来准备参数,同样地我们可以在函数调用处分析出这个调用准备了多少个参数.注意,由于我们的分析是基于二进制的,所以做不到精确分析每个函数使用的参数数量,只能得出某个函数最多使用了多少参数.在生成 CFG 的过程中,我们添加一条间接函数调用的边时,被调用的函数消耗的参数数量必须小于等于调用函数处准备的参数数量.即,一个准备了 2 个参数的函数调用处,不能调用消耗了 3 个及 3 个以上参数的函数.

生成 CFG 后,由于 CFG 的形式是以各种跳转将每个基本块相连,和 IPT 记录包的形式不符,会造成在控制流检查时性能低下.我们采用 FlowGuard[30]中的做法,将其重构为和 IPT 记录包形式兼容的 ITC-CFG (Indirect Target Connected CFG),即删除所有直接跳转的边,留下那些作为间接转移指令的目标的基本块,再将它们连接起来.这样重构过的 CFG 是和原来的 CFG 完全等价的,因为直接跳转指令的行为都是确定的,不会产生新的边或者删除原有的边.重构之后的 CFG 和 IPT 记录包的形式相仿,可以简单高效地进行比对.

11 控制流的实时保护

本系统在虚拟机管理器中控制保护系统的开启和关闭.本系统还修改了扩展页表错误处理函数,用于在合适的时机触发控制流检查器,进行控制流检查,确保控制流是合法的.

11.1 系统加载与开启

在要被保护的内核模块在虚拟机中加载完毕时,虚拟机管理器可以获取到被加载模块的地址和二进制,然后将它们读出并传入静态分析程序中.本系统的内核模块将静态分析生成的 ITC-CFG 传入控制流检查器中,供动态检查控制流时使用.之后,虚拟机管理器配置开启 IPT,记录被保护的虚拟机的内核模块的控制流.通过控制 IA32_RTIT_CTL MSR 寄存器来控制记录包的条件:将 TraceEn 位和 BranchEn 位设为 1 来开启控制流记录,将 OS 位设为 1 并将 User 位设为 0 来只记录内核中的控制流,将 FabricEn 位设为 0 来直接将记录的包发送到内存中.最后,本系统将被保护的内核模块设置为不可执行,保护开始.

11.2 实时控制流保护

本系统的目标是确保被保护的内核模块的控制流完整性.所以,我们必须在合适的时机触发控制流检查:要确保在攻击者影响到内核的关键数据前进行检查,同时检查又不能太频繁,否则带来的额外性能开销过大.综合考虑,本系统选择在控制流从被保护的内核模块切换到内核核心中的时候进行控制流检查.因为内核模块中攻击者做出的影响往往要等到控制流转移到内核核心中时才会生效.而在这个检查点,攻击者还没有将控制流转移到内核核心中,我们可以确保被保护的系统不会受到实质性的危害,保证了系统的安全性.同时,检查不会被太过频繁地触发,保证了系统的效率.

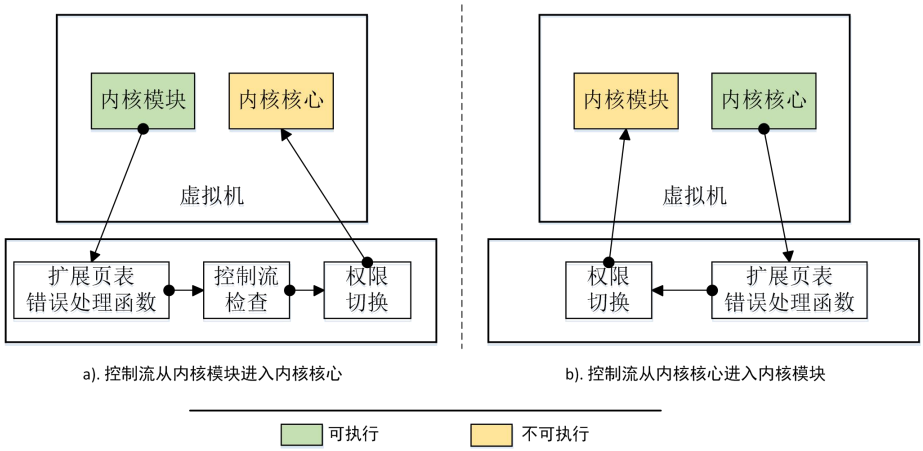

Fig.2 Check control flow using EPT violation

图 2 利用扩展页表错误触发控制流检查的过程

如上节所说,本系统在开启保护的时候,将被保护的内核模块设置为不可执行.这样一来,在控制流到达被保护的内核模块时,会产生一个扩展页表错误,调用扩展页表错误处理函数,进行权限切换,将被保护的内核模块设置为可执行,将内核核心部分设置为不可执行,然后将控制流交还给虚拟机的内核模块,如图 2(b).注意,这里的内核核心还包括了除了被保护的内核模块的其他内核模块.之后,当控制流从被保护的内核模块转移到内核核心时,会触发另一个扩展页表错误.这次在扩展页表错误处理函数中,满足控制流检查的触发条件(控制流从内核模块跳转到内核核心),函数会调用控制流检查器.检查通过后,它也会进行权限切换,将内核核心部分设置为可执行,将内核模块设置为不可执行,再将控制流交还给虚拟机继续运行,回到和开启保护时相同的状态,如图 2(a).

我们希望在控制流跳出内核模块的时刻触发控制流检查.因此,本系统在满足以下条件时触发检查:触发本次扩展页表错误的地址不在被保护的内核模块内部,且上一次触发扩展页表错误的地址在被保护的模块内.即,当前的控制流在内核模块中并即将进入内核核心时,在控制流跳出内核模块的瞬间进行检查.

在控制流检查器中,我们需要将 IPT 记录的包和之前生成的 ITC-CFG 进行比对.首先,检查器从 IPT 记录的缓冲区中读出 IPT 记录的包.IPT 使用的是环形缓冲区,在每次读取缓冲区时会记录当前的 offset.由于内核模块经常会调用内核中的函数,和内核核心的交互比较频繁,所以两次检查之间的间隔很短,在此期间 IPT 记录的数据不会超过缓冲区的长度,把还未被读取的数据覆盖(至少在我们的实验过程中，这种情况并没有发生过).当然为了完全避免缓冲区被填满所造成的 false negative,我们可以设置相关的中断,在缓冲区填满时产生 buffer-filled PMI interrupt 中断，来避免该问题.注意,IPT 不只记录虚拟机生成的包,同时也会记录宿主机(Host)的控制流.我们需要挑选出控制流在虚拟机中时 IPT 记录的包.如 4.1 节所述,我们在开启 IPT 时,设置了只记录内核状态下产生的包.在页表基地址寄存器(CR3)被修改时,IPT 会记录一个 PIP 包,表示页表映射发生了修改.在 PIP 包中,有一个 Non-Root 位.如果 PIP 包在非 root 的环境下生成,则 Non-Root 位被设置为 1,反之则为 0.非 root 环境下运行的是虚拟机,所以我们只需要提取出在一个 Non-Root 位为 1 的 PIP 包(代表进入非 root 环境,即进入虚拟机)和一个 Non-Root 位为 0 的 PIP 包(代表进入 root 环境,即返回宿主机,退出虚拟机)中间的那些包,即可得到在虚拟机中产生的所有包.

收集到需要检查的包之后,控制流检查器将它们和 ITC-CFG 进行比对.在 ITC-CFG 中,每个节点都有一个或数个合法目标地址,代表下一个间接跳转目标地址可能的值.在控制流检查的过程中,检查器逐个读出记录的 TIP 包的地址,然后去 ITC-CFG 中查找对应的节点.如果找到了对应节点,则在节点中记录的目标地址的数组中查找下一个 TIP 包记录的地址.如果任一查找失败,则代表控制流检查失败,检查器将结果通知虚拟机管理器,交

由其进行后续处理.

除了检查在被保护的内核模块内的控制流,我们还需要确保控制流从被保护的内核模块返回到内核核心时,这条返回指令的目标地址是合法的.因为攻击者执行 ROP 攻击时,往往会覆写栈上的返回地址来恶意跳转到内核核心中某个关键函数来实现目的.所以除了被保护模块内的控制流,从这个模块中跳出的那次跳转的合法性也必须被确保.然而,我们并不依赖于内核核心的源代码或者二进制,无法直接对内核核心做出分析.本系统使用类似影子堆栈(Shadow stack)的方式来检查返回值:在虚拟机管理器内维护一个函数调用栈,当控制流从内核核心跳转到被保护的内核模块中时,如果目标地址为某个函数的开头,则内核核心调用了内核模块的函数,将存在内核的栈中的栈顶的值即返回地址读出,存入我们维护的函数调用栈中.在被保护的内核模块执行返回指令且目标地址为内核核心中的地址时,将目标地址和函数调用栈上的最新地址进行匹配,如果相同则这个返回指令为合法的,否则检查器将通知虚拟机管理器检测到了 ROP 攻击.如果被保护的内核模块通过间接函数调用跳转到内核核心中,由于本系统的目标只是防御 ROP 攻击,我们认为这个调用是合法的,不检查这一次调用而只检查内核模块内部的控制流.有关安全性的评测详见 5.1 节.

但是有一种情况例外:当控制流在被保护的内核模块中时,系统接收到一个中断(如 Timing interrupt),内核核心需要去处理这个中断,暂停内核模块的执行.由于发生中断时,程序正在执行的地址是未知的,所以由于中断会产生很多未知的控制流挑战.幸运的是,IPT 硬件帮我们解决了这个问题.发生中断时,IPT 会在记录一个 TIP 包的同时记录一个 FUP(Flow-update)包,代表控制流的状态产生了更新.如果控制流检查器在读取 IPT 记录的包时,发现最新的 TIP 包之前紧接着一个 FUP 包,则代表最新的跳转是由中断产生的.这时候,我们只检查被保护模块内的控制流,而不检查从模块中跳出的指令的合法性,因为这是一个中断的处理,我们不认为攻击者有能力在虚拟机中制造中断.

使用 sliding window 的机制进行实时控制流检查，控制流在每次从内核模块中跳转到内核核心时都会读取 IPT 记录的缓冲区。IPT 使用的是环形缓冲区，在每次读取缓冲区时会记录当前的 offset。由于内核模块经常会调用内核中的函数，和内核核心的交互比较频繁，所以两次检查之间的间隔很短，在此期间 IPT 记录的数据不会超过缓冲区的长度，把还未被读取的数据覆盖（至少在我们的实验过程中，这种情况并没有发生过）。当然为了完全避免缓冲区被填满所造成的 false negative，我们可以设置相关的中断，在缓冲区填满时产生 buffer-filled PMI interrupt 中断，来避免该问题。

12 系统评测

我们的机器使用支持了 IPT 的 Intel Skylake 处理器,现在市面上可以方便的买到并且已被广为使用.我们已实现了本系统的原型.所有的测试在上述机器上运行,处理器为 Intel i7-6700K,操作系统为 Ubuntu 16.04,内核版本为 Linux 3.16.38.使用的虚拟机管理器为 Linux 所带 kvm 模块,使用的虚拟机为 Debian 8 操作系统,内核版本为 Linux 4.3.3.静态分析 CFG 的实现基于 Dyninst 框架开发.

12.1 安全性评测

为了确保安全性,本系统要求能够反汇编内核模块代码以及找到函数边界等信息.我们以 Dyninst 框架为基础,并在此之上编写了我们的模块分析器.Dyninst 框架中提供了多种分析二进制程序的接口,包括在 BIN-CFI[11] 中提到的线性分析和递归分析等.我们在 ext4 文件系统,e820 网卡驱动,ata 磁盘驱动上做了实验,结果表明我们的分析器可以准确分析出模块中包含的所有函数及其边界.

本系统可以防御真实的在内核模块 ROP 攻击.为了验证这点,我们手动编写了一个带有缓冲区溢出漏洞的内核模块,并编写了一个 ROP 攻击程序,利用内核模块的缓冲区溢出漏洞,来覆写掉内核栈上的返回地址,劫持控制流,最终达成向某一特定文件内写入任意值的目的.开启了本系统的保护之后,在被保护的内核模块执行到返回指令时,读取的返回地址是被修改的内核核心内的地址,控制流检查器检查出这个地址和函数调用栈上存储的值不一样,将结果通知虚拟机管理器,虚拟机管理器暂停这个虚拟机的执行并报告给用户,等待后续处理.

本系统可以确保在 ROP 攻击生效前检测到攻击并中止虚拟机运行,保证攻击不会造成危害.在先前的研究

中,有些攻击者会想办法避免触发控制流检查以隐藏攻击.然而这在本系统中是行不通的:假如攻击者想在被保护的模块内部寻找 gadgets,即用来在 ROP 攻击中执行的代码,修改一些寄存器和栈上的值,之后再返回到正常的返回地址,达到修改之后执行的函数参数的目的,进而实现一些攻击.然而在控制流返回到正常的返回地址时,由于它跳出了被保护的内核模块,所以会触发控制流检查,模块内部的 gadgets 会被发现,攻击不可能顺利完成.还有一种情况是,攻击者希望一直在模块内部进行跳转,来修改一些关键内存或寄存器,使控制流不跳出模块,从而使攻击不被发现.然而,在模块内部能够做的事情很有限:即使攻击者修改了很多关键数据,如果控制流不从模块中跳出去,那么那些数据也不会被用到,攻击也不会造成实际效果;而且,系统每隔一定时间会发出中断,在跳到中断处理函数的时候即跳出了被保护的模块,会触发控制流的检查.

还有一种可能的攻击是,攻击者通过内核模块的内存漏洞去修改属于内核核心部分的栈,而不修改内核模块部分的栈.这样一来,内核模块执行返回指令时,会返回到正确的内核核心中地址,但是内核核心接下来会使用攻击者修改过的栈.然而,现在并没有这种攻击发表,并且也超出了本文的防止在内核模块内发生的 ROP 攻击的范畴.这种攻击可以通过换栈来防御:我们为内核模块准备一个单独的栈,在切换至内核模块执行时,扩展页表错误处理函数将虚拟机的栈指针映射到一个新的页上,和之前的内核栈区分开,防止攻击者修改内核栈;之后在控制流返回内核核心时再将栈切换回内核栈,保证正常执行.

由于本系统的目标是防御内核模块中的 ROP 攻击,在内核模块通过间接函数调用来调用内核核心中的函数中时,并没有进行检查.由于我们不依赖于内核核心的源码或二进制,我们无法对内核核心进行分析,也就无法得出内核模块可能会调用内核核心中哪些函数.现在被公布出的在内核中的控制流劫持攻击只有 ROP 攻击一种,在内核模块中修改函数调用地址来进行控制流攻击的难度较大,不易实现,防御 ROP 攻击已经提供了极强的安全性.本系统可以防御所有内核模块中的 ROP 攻击,有力地阻止攻击者利用内核模块中的漏洞来攻击内核.

12.2 性能评测

我们对本系统进行了性能测试,用于多方面评测本系统所带来的额外开销.

5.2.1 Linux 命令的性能测试

我们在虚拟机中测试了 Linux 自带的一些命令,并记录他们的运行时间,之后再开启本系统的保护,重新测试相同的命令,将它们的结果比较,得到额外的性能开销.每个测试都被重复 10 次,并取平均值作为该条测试的结果.我们选择保护被监控虚拟机的 ext4 文件系统内核模块,并测试在保护 ext4 时运行各个基准测试的性能,并算出了造成的额外性能开销的几何平均值.

我们选取了一些有代表性的 Linux 命令进行测试,使用默认配置运行这些命令,并使用 time 命令计算运行所用的时间.之后开启本系统保护,再次运行相同保护并记录所用的时间,并比较两次运行所用时间.我们选择的命令有:用于查看当前文件夹下所有文件的 ls,用于解压缩的 tar,用于文件操作的 dd,以及用于浏览网页的 curl.通过结果发现,对于频繁使用文件操作的 ls、tar、dd 命令,所造成的额外性能开销比较大,而主要开销在网络传输的 curl 命令则开销很小.平均的额外性能开销为 38.75%,如图 3 所示.

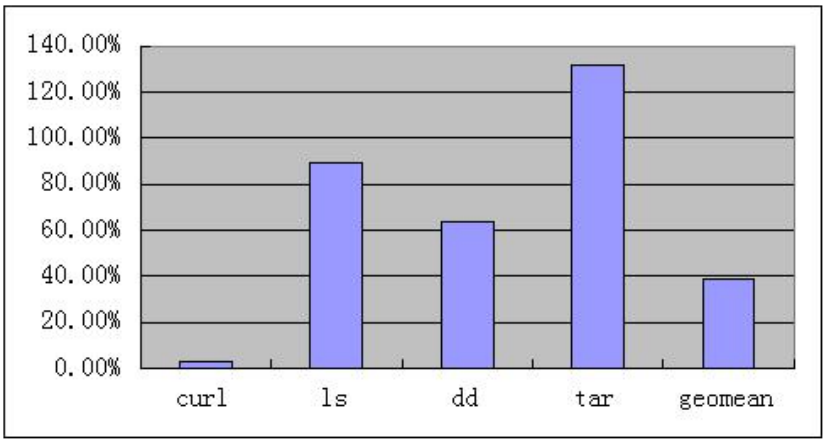

Fig.3 Overhead when running Linux utilities

图 3 开启本系统时运行 Linux 命令造成的额外性能开销

5.2.2 服务器程序的性能测试

因为本系统目标为保护客户的虚拟机,而服务器程序是攻击者最常使用的攻击载体,也是客户虚拟机上运行较多的程序,所以本系统的一个主要应用场景是在运行着服务器程序的客户虚拟机上实施保护.

为了测试本系统会造成的最大的性能开销,我们选取的被保护的内核模块是服务器程序中利用最频繁的网卡驱动模块,并测试在保护它时运行各个服务器程序时的性能开销.我们在 Web 服务器 nginx-1.6.3,FTP 服务器 vsftpd-3.0.3,SSH 服务器 OpenSSH-7.1p1 上进行了测试.在 nginx 上,我们使用 Apache 基准测试(Apache benchmarks,ab)来模拟 10 个并发的客户同时产生 20K 个请求,其中每个请求都请求一个大小为 20B 的文件.在 FTP 服务器上,我们使用 pyftpbench 来下载 10MB 大小的文件.在 OpenSSH 上,我们使用了登陆并执行一般命令的脚本来进行测试.每种测试同样进行了 10 次并取平均值.图 4 展示了开启本系统时运行服务器程序造成的额外性能开销.如图可以看出,造成的平均额外性能开销为 61%.

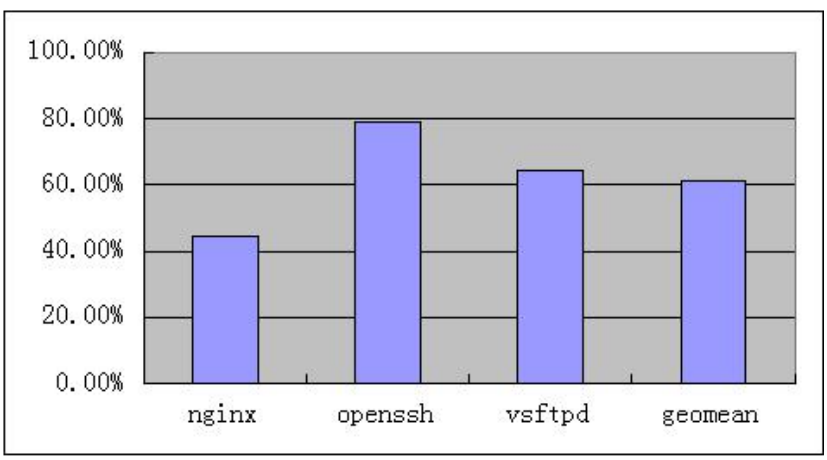

Fig.4 Overhead when running server applications

图 4 开启本系统时运行服务器程序造成的额外性能开销

13 讨 论

13.1 对返回指令的保护

本系统目前只支持对返回指令的保护,只防御通过篡改返回地址来劫持控制流的 ROP 攻击,不能防御所有的控制流劫持攻击.这是因为本系统设计的目标是不依赖内核核心的源代码和二进制,无法对内核核心进行分析,所以本系统无法获得较精确的内核中的合法跳转目标地址集合.现在流行的控制流劫持攻击大多数都为ROP攻击,并且现有的公布出来的针对操作系统内核的控制流攻击全部为ROP攻击.因此,我们认为本系统基本能够防御针对内核的控制流劫持攻击,已提供足够强的安全性.除本系统之外,也有一些保护 CFI 的系统只保护返回指令,不保护间接函数调用指令.如 PT-CFI[31]就是一个这样的系统.这篇文章中提到,目前还没有一个完美的保护间接函数调用指令的机制,因为各种分析方法都或多或少存在漏洞,无法准确分析出每一条间接函数调用指令的合法目标地址.

除开操作系统内核核心,内核的地址空间还包括除了被保护的内核模块的其他内核模块.我们已经做了一些工作来分析其他的内核模块.经过分析发现,我们可以通过被保护的内核模块的信息获得它所依赖的内核模块和依赖它的内核模块,然后获得这些内核模块中合法的跳转目标地址.被保护的内核模块的间接函数调用只能调用如下几个地方:1、该内核模块内部;2、某个该内核模块所依赖的内核模块内部;3、某个依赖该内核模块的内核模块内部;4、内核核心中.(1)的情况已经通过静态分析生成的 CFG 解决,而在(2)和(3)中的合法跳转目标地址我们可以也通过静态分析得到.所以,如果被保护的内核模块的一条间接函数调用指令的目标的地址为内核模块的地址范围,而非内核核心的地址范围,就可以确保它是合法的.然而,即使我们保护了调用其他内核模块中的函数调用的合法性,攻击者仍然可以直接调用内核核心中的函数来实行攻击.我们尝试对内核核心的二进制做了一些分析,但是由于内核本身过于庞大,而且其中有很多非常规的函数调用方式,仅仅使用基于二进制的分析无法很好的限制非法的函数调用.所以,完全的基于二进制来保护内核的控制流完整性是一项比较艰巨的工作.

13.2 额外性能开销

如 5.2 节所述,本系统目前造成的额外性能开销相对较大,经测试在开启本系统后对 Linux 基本命令造成的额外性能开销约为40%,而对服务器程序造成的最大额外性能开销约为60%.这主要有两个原因:第一,我们在进行性能测试的时候,选取的是会有较高负载的内核模块.如果我们运行的程序和被保护的内核模块关系不大,如在保护 ext4 文件系统模块时运行 curl 来浏览网页,则只会造成 2.99%的额外性能开销;第二,本系统对返回指令做了最精细的保护,保证本系统可以防御所有的ROP攻击,所以需要较频繁地进行控制流检查,来保证虚拟机的安全性.同时,我们也做到了极好的透明性和兼容性:我们不需要内核核心的源代码和二进制,使本系统可以用于几乎所有的操作系统;我们也不需要对内核核心和内核模块的二进制进行修改,使本系统可以用于几乎所有的虚拟化云服务场景.我们用相对较大的额外性能开销,换来了极强的安全性、兼容性和透明性,我们认为这是值得的.

对于本系统目前性能开销相对较大的问题,我们也拟出了一些优化方案.如,在每次控制流跳出内核模块的时候,在虚拟机内部检查这个跳转的源地址和目标地址对是否在之前已经遇到过.如遇到过,则直接认为这个跳转是合法的,不触发虚拟机下陷从而提高效率.但是这种做法必须向虚拟机中加入新的内核模块,无法维持我们提供的保护的透明性.再如,我们可以将一些被经常调用的函数(如中断处理函数)放在保护范围之外,即调用它们的时候我们不会触发扩展页表错误,减少虚拟机下陷.但是这样的做法给了攻击者将控制流跳转指令重定向到这些函数内部并实施攻击的机会,会影响系统的安全性.可以看出,目前对本系统尝试做的性能优化的方案都会一定程度上的牺牲本系统提供的保护精确性、透明性、兼容性.

14 总 结

本文针对目前流行的ROP攻击的问题,以及鲜有保护操作系统内核不受ROP攻击的系统的现状,提出了一套基于硬件的使用虚拟化手段来保护针对内核模块的 ROP 攻击的系统.本系统分为内核模块静态分析、控制流记录、动态检查几个模块.利用 IPT 高效记录的特点,本系统可以准确的捕捉被保护虚拟机的控制流.通过修改扩展页表错误处理函数,在控制流跳出内核模块的时候触发一个扩展页表错误,在扩展页表错误处理函数中触发控制流检查,将记录的控制流与静态分析生成并重构的 ITC-CFG 进行比对,可以精确的确保被保护虚拟机的返回指令是合法的,保证虚拟机不受 ROP 攻击.测试结果表明,本系统可以精确地检测到 ROP 攻击,并不会影响正常状态下虚拟机的运行,提供了极强的安全性、兼容性、高效性。

References:

[35] Martín Abadi , Mihai Budiu , Úlfar Erlingsson , Jay Ligatti, Control-flow integrity, Proceedings of the 12th ACM conference on Computer and communications security, November 07-11, 2005, Alexandria, VA, USA.

[36] Wang, Zhi, and Xuxian Jiang. "Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity." Security and Privacy (SP), 2010 IEEE Symposium on. IEEE, 2010.

[37] J. Criswell, N. Dautenhahn, and V. Adve, “KCoFI: Complete Control-Flow Integrity for Commodity Operating System Kernels,” in Proceedings of the 2014 IEEE Symposium on Security and Privacy, ser. SP ’14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 292–307. [Online]. Available: http://dx.doi.org/10.1109/SP.2014.26

[38] N. L. Petroni Jr and M. Hicks, “Automated detection of persistent kernel control-flow attacks,” in Proceedings of the 14th ACM conference on Computer and communications security. ACM, 2007, pp. 103–115.

[39] Arvind Seshadri , Mark Luk , Ning Qu , Adrian Perrig, SecVisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity OSes, Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles, October 14-17, 2007, Stevenson, Washington, USA [doi>10.1145/1294261.1294294]

[40] Ge X, Talele N, Payer M, et al. Fine-Grained Control-Flow Integrity for Kernel Software[C]//2016 IEEE European Symposium on Security and Privacy (EuroS&P). IEEE, 2016: 179-194.

[41] Moreira J, Rigo S. Control-Flow Integrity for the Kernel[J].

[42] 王明华, 尹恒, 苏璞睿, 冯登国. 二进制代码块: 面向二进制程序的细粒度控制流完整性校验方法. 信息安全学报. 2016(2):61-72.

[43] Ge, Xinyang, et al. "Fine-grained control-flow integrity for kernel software." Security and Privacy (EuroS&P), 2016 IEEE European Symposium on. IEEE, 2016.

[44] P.Team.Rap:Riprop.https://pax.grsecurity.net/docs/PaXTeam-H2HC15-RAP-RIP-ROP.pdf", 10 2015. [Online; accessed 7-August-2016].

[45] Zhang, Mingwei, and R. Sekar. "Control Flow Integrity for COTS Binaries." Usenix Security. Vol. 13. 2013.

[46] Zhang, Mingwei, et al. "A platform for secure static binary instrumentation." ACM SIGPLAN Notices 49.7 (2014): 129-140.

[47] B. Niu and G. Tan, “Modular control-flow integrity,” in PLDI, 2014.

[48] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, ´ U. Erlingsson, L. Lozano, and G. Pike, “Enforcing forward-edge control-flow integrity in gcc & llvm,” in USENIX Security, 2014.

[49] A. J. Mashtizadeh, A. Bittau, D. Boneh, and D. Mazi`eres, “Ccfi: Cryptographically enforced control flow integrity,” in CCS, 2015.

[50] W. Arthur, B. Mehne, R. Das, and T. Austin, “Getting in control of your control flow with control-data isolation,” in CGO, 2015.

[51] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou, “Practical control flow integrity and randomization for binary executables,” in S&P, 2013.

[52] M. Payer, A. Barresi, and T. R. Gross, “Fine-grained control-flow integrity through binary hardening,” in DIMVA, 2015.

[53] R. Qiao, M. Zhang, and R. Sekar, “A principled approach for rop defense,” in ACSAC, 2015.

[54] V. Mohan, P. Larsen, S. Brunthaler, K. W. Hamlen, and M. Franz, “Opaque control-flow integrity.,” in NDSS, 2015.

[55] V. van der Veen, E. G ¨ oktas, M. Contag, A. Pawlowski, X. Chen, S. Rawat, H. Bos, T. Holz, E. Athanasopoulos, and C. Giuffrida, “A tough call: Mitigating advanced code-reuse attacks at the binary level,” in S&P, 2016.

[56] M. Kayaalp, T. Schmitt, J. Nomani, D. Ponomarev, and N. Abu-Ghazaleh, “Scrap: Architecture for signature-based protection from code reuse attacks,”in HPCA, 2013.

[57] S. Arnautov and C. Fetzer, “Controlfreak: Signature chaining to counter control flow attacks,” in RDS, 2015.

[58] A. Vasudevan, N. Qu, and A. Perrig, “Xtrec: Secure real-time execution trace recording on commodity platforms,” in HICSS, 2011.

[59] L. Yuan, W. Xing, H. Chen, and B. Zang, “Security breaches as pmu deviation: detecting and identifying security attacks using performance counters,” in APSys, 2011.

[60] Y. Xia, Y. Liu, H. Chen, and B. Zang, “Cfimon: Detecting violation of control flow integrity using performance counters,”in DSN, 2012.

[61] V. Pappas, M. Polychronakis, and A. D. Keromytis, “Transparent rop exploit mitigation using indirect branch tracing,” in USENIX Security, 2013.

[62] Y. Cheng, Z. Zhou, Y. Miao, X. Ding, H. DENG, et al., “Ropecker: A generic and practical approach for defending against rop attack,” 2014.

[63] V. van der Veen, D. Andriesse, E. G ¨ okta , s, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida, “Practical context-sensitive cfi,” in CCS, 2015.

[64] Liu, Y., Shi, P., Wang, X., Chen, H., Zang, B., & Guan, H. (2017, February). Transparent and efficient cfi enforcement with intel processor trace. In High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on (pp. 529-540). IEEE.

[65] Gu, Yufei, Qingchuan Zhao, Yinqian Zhang, and Zhiqiang Lin. "PT-CFI: Transparent backward-edge control flow violation detection using intel processor trace." In Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy, pp. 173-184. ACM, 2017.

[66] Van der VEEN V, GÖKTAS E, CONTAG M, et al. A tough call: Mitigating advanced code-reuse attacks at the binary level[C]//Security and Privacy (SP), 2016 IEEE Symposium on. IEEE. [S.l.]: [s.n.], 2016: 934 - 953.

[67] Hund, Ralf, Thorsten Holz, and Felix C. Freiling. "Return-Oriented Rootkits: Bypassing Kernel Code Integrity Protection Mechanisms." USENIX Security Symposium. 2009.

[68] 李珣, 黄皓. 一个基于硬件虚拟化的内核完整性监控方法. 计算机科学. 2011;38(12):68-72.

附中文参考文献:

[8] 王明华, 尹恒, 苏璞睿, 冯登国. 二进制代码块: 面向二进制程序的细粒度控制流完整性校验方法. 信息安全学报. 2016(2):61-72.

[34] 李珣, 黄皓. 一个基于硬件虚拟化的内核完整性监控方法. 计算机科学. 2011;38(12):68-72.