操作系统内核模糊测试技术综述

李 贺[1,2],张 超[2],杨 鑫[1,2],朱俊虎[1]

[1](数学工程与先进计算国家重点实验室,郑州 450002)
[2](清华大学 网络科学与网络空间研究院,北京 100083)
E-mail: chaoz@tsinghua.edu.cn

摘 要: 模糊测试作为一种高效的漏洞挖掘方法,在操作系统内核安全领域得到了广泛应用. 内核模糊测试的应用促进了操作系统内核和驱动程序安全防护水平的显著提升. 目前,针对不同平台上操作系统使用模糊测试技术进行漏洞挖掘已经成为研究热点. 文章对现有的内核模糊测试方法进行研究,综述了内核模糊测试发展情况和技术思想,并尝试对内核模糊测试进行分类,总结了近年来内核模糊测试中使用的新技术. 最后讨论了目前研究中的问题,并对内核模糊测试未来发展趋势进行了展望.

关 键 词: 操作系统内核; 模糊测试; 漏洞挖掘; 驱动程序
中图分类号: TP309 **文献标识码:** A **文 章 编 号:** 1000-1220(2019)09-1994-06

Survey of OS Kernel Fuzzing

LI He[1,2],ZHANG Chao[2],YANG Xin[1,2],ZHU Jun-hu[1]
[1](State Key Laboratory of Mathematical Engineering and Advanced Computing,Zhengzhou 450002,China)
[2](Institute for Network Sciences and Cyberspace,Tsinghua University,Beijing 100083,China)

Abstract: As an efficient vulnerability discovering method,fuzzing testing has been widely used in operating system kernel security. Kernel fuzzing significantly improves the security of operating system kernel and driver programs. At present,the use of fuzzing techniques for operating systems on different platforms for vulnerability discovering has become a research hotspot. This paper studies the existing kernel fuzzing methods,summarizes the development of kernel fuzzing and technical ideas,and attempts to classify kernel fuzzing. Summarizes the new technologies used in kernel fuzzing in recent years. Finally,the problems in the current research are discussed,and the future development trend of kernel fuzzing testing is prospected.

Key words: OS kernel; fuzzing; vulnerability discovering; driver

1 引 言

操作系统作为当今社会信息基础设施中最为基础的软件设施,从其诞生之初就与安全研究有着密不可分的联系,是软件安全最早的研究对象之一[1,3]. 内核作为操作系统最重要的组成部分,其安全性一直是计算机安全研究的热点. 随着近年来技术的发展,针对 Windows 内核、Linux 内核和 XUN 内核等主流 PC 和服务器操作系统的漏洞挖掘研究也越来越多,出现了大量研究成果. 自 2007 年以来,iOS 和 Android 等移动终端系统的市场份额的不断扩大,针对移动终端操作系统内核的研究也成为安全研究一个重要关注对象.

内核漏洞有其独特的优良性质. 首先,一般的操作系统平台都有着数量巨大的用户群,这使得内核漏洞具有很强的通用性. 其次,操作系统内核代码极其庞大,拥有大量的遗留代码和纷繁复杂的子模块,增加了漏洞发生的可能性. 再次,由于内核特权级高,内核漏洞可以实现获取内核资源的访问权限、获取超级用户权限、关闭安全防护功能等高危攻击行为,并且能够为 rootkit 等后门驻留打开突破口. 最后,内核重视运行性能,默认情况下安全防护机制较少. 尤其是在实际工作中还需要维持生产环境的稳定,甚至需要关闭很多内核防御机制,给攻击方留下了很大的可利用空间.

模糊测试作为实践效果良好的漏洞挖掘方法,在内核漏洞挖掘领域得到了广泛的应用. 以虚拟化、硬件调试器、云计算为代表的新技术都被应用到了内核模糊测试漏洞挖掘框架中,主流操作系统都有专门针对其内核的模糊测试漏洞挖掘工具. 从发展历史来看,内核漏洞模糊测试从一开始简单的随机生成测试例,发展到输入规范化、虚拟化运行系统、覆盖率反馈、多系统支持、云平台集群化运行等诸多新特性.

本文主要对近年来内核模糊测试的发展进行回顾,总结内核模糊测试的技术特点,并对其面临的问题进行梳理. 对典型工具的技术思想进行综述,并专门讨论了驱动模糊测试技术. 最后对本文进行总结,讨论了内核模糊测试今后的发展方向.

2 内核模糊测试技术概述

2.1 模糊测试

收稿日期: 2019-03-01 收修改稿日期: 2019-04-09 基金项目: 国家自然科学基金联合基金项目(U1736209)资助. 作者简介: 李 贺,男,1989 年生,硕士研究生,研究方向为系统安全; 张 超(通讯作者),男,1986 年生,博士,副教授,CCF 会员,研究方向为软件与系统安全、物联网与区块链应用安全、软件分析技术、AI 与安全; 杨 鑫,男,1991 年生,硕士研究生,研究方向为系统安全; 朱俊虎,男,1974 年生,博士,教授,研究方向为网络安全.

模糊测试(Fuzz testing,Fuzzing)作为目前较为有效的软件漏洞挖掘工具,在新世纪以来得到了学术界和工业界的广泛应用. 模糊测试的核心思想是自动化或半自动的生成输入数据到目标程序中,监测目标程序运行是否发生异常. 通过对造成程序运行失常的输入执行过程进行分析,从而找出目标程序中的隐藏缺陷. 从测试例生成来说,模糊测试可以划分为基于生成(Generation-based)和基于变异(Mutation-based)两种技术路线,也有将其称为基于语法(Grammar-based)和基于反馈(Feedback-based)的模糊测试. 基于生成的模糊测试偏向于利用目标程序输入语法、目标静态分析等知识生成大致符合目标软件输入格式的测试例,而基于变异的模糊测试则使用遗传算法等方法迭代生成表现更好的测试例. 从引导方式上来说,模糊测试可以划分为基于随机的模糊测试(blind fuzzing)和基于覆盖率引导的模糊测试. 在实践中,模糊测试确实能够挖掘出内存错误、数据竞争、逻辑错误等多种类型的漏洞.

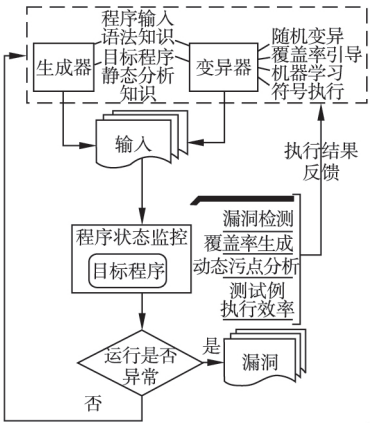

图 1　模糊测试

Fig. 1　Fuzzing test

如图 1 所示,有研究[11,26-28,31-33]将模糊测试与静态分析、符号执行、污点分析、机器学习等技术结合,加强模糊测试平台漏洞挖掘能力,黎军[6]等人对模糊测试进行系统性的总结. 模糊测试主要的缺点是覆盖率较低,生成的测试例一般只能覆盖小部分目标程序代码,并且大量测试例基本上是无效的. 实践中,模糊测试在漏洞挖掘领域取得良好效果,已经成为业界主流的漏洞挖掘方法.

2.2　内核模糊测试面临的挑战

与普通软件的安全研究相比,针对操作系统内核的安全研究一直面临着代码规模过于庞大、功能复杂、部分模块相关文档较少、调试运行不便等诸多问题,而针对内核的模糊测试也在此列. 与针对普通软件的模糊测试相比,实现系统内核的高效模糊测试需要克服以下几个问题.

2.2.1　操作系统内核代码规模带来的问题

2018 年 9 月整个 Linux 系统代码行数已经超过了 2500 万行[1],Windows 在 2003 年(NT 5.2)就已经达到了 5000 万行[5]. 庞大的代码规模使得模糊测试的目标模块和系统调用接口数目非常多. 并且 Linux、Windows 等系统都能在不同硬件架构上运行,这也进一步加剧了模糊测试面临的规模问题. 由于内核代码的规模,研究者很难对内核的各个模块具体运行有广泛又细致的理解,也导致内核模糊测试过程中测试例初始种子挑选、崩溃样本分析等人工参与的环节需要消耗大量人力来提取相关信息.

2.2.2　操作系统内核运行环境带来的问题

模糊测试中必不可少的一个环节就是需要对被测目标软件运行状态进行监控,发现软件的异常状态. 而操作系统内核位于计算机体系底层,特权级高,不仅需要处理硬件管理事务,还要对上层用户程序的运行提供支撑,这就使得内核作为被监控程序运行存在一定的技术困难. 早期的模糊测试采用直接在物理主机上运行,显而易见的存在漏洞触发状态记录、漏洞调试上的巨大困难,因此大部分的内核模糊测试都不同程度上利用虚拟化技术来监控运行内核. 内核负责整个系统任务切换和资源分配等基础功能,运行状态非常复杂多变,提取漏洞触发状态语义比较困难,尤其是数据竞争和逻辑错误两种漏洞更为明显. 另外,作为高权限运行实体,操作系统内核还会出现二次获取[24,25](double fetch)和二次写入(double write)等独特的逻辑漏洞.

2.2.3　操作系统开发方式带来的问题

由于 Windows、MacOS 等操作系统内核的大部分代码都不对外开放,非厂商相关研究者针对这些闭源操作系统的内核漏洞挖掘就必须对闭源模块进行逆向分析,并且无法实现基于源码插桩的覆盖率引导模糊测试. 此外,Linux 等开源操作系统为了实现更好的功能,更新速度很快,Linux v4 在 2018 年[2] 就从 4.15 发展到了 4.20,发布了 40 多个最终测试版本,这也给内核模糊测试工作带来了很大压力.

2.3　内核模糊测试基本框架

内核模糊测试的威胁模型将用户态或硬件输入到操作系统内核的数据看作是威胁来源,即从低权限的用户空间、外部硬件到高权限的内核空间中的输入是不可信的. 目前,用户态输入作为威胁来源的模糊测试已经有了比较多的研究,但将硬件输入作为威胁来源的研究还较少,文献[39]对这个方面进行了讨论.

以图 2 为例,内核模糊测试将生成的畸形数据输入到被测内核,并期望发生内核运行异常. 但由于操作系统内核的特殊性,内核模糊测试在测试例生成、输入、内核运行监控等方面需要更高的技术实现水平. 除了直接使用物理机以及将内核代码库做用户态模糊测试,其他内核模糊测试工具都同程度上应用了虚拟化方式运行目标系统. 一般来说,内核模糊测试主要的评价标准包括内核代码覆盖率、漏洞触发识别能力以及内核监控运行额外开销等等.

3　内核模糊测试技术分类和典型工具

除了传统软件测试中基于对目标软件知识依赖程度而划分的白盒、灰盒及黑盒测试,内核模糊测试还有很多的分类标准. 系统内核的攻击面基本上可以认为是传入内核的各种数

[1] https://phoronix.com/misc/linux-20180915/index.html

[2] https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/refs/tags

据接口,包括系统调用参数、硬件输入、用户态共享到内核态的内存对象等. 从这些攻击面来看,内核模糊测试可以分为面向系统调用和面向系统数据处理接口两种. 从引导方式来看,内核模糊测试工具也可以分为随机模糊测试(blind fuzzing)和覆盖率引导的模糊测试. 除此以外,也可以用针对的目标操作系统、漏洞类型将内核模糊测试工具进行分类. 文献[7]对

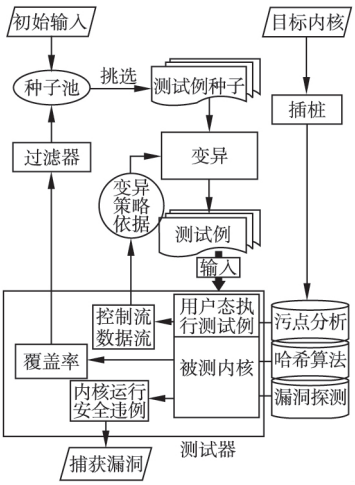

图2 基于覆盖率反馈的内核模糊测试

Fig. 2 Cover age-guided kernel fuzzing

针对系统调用的模糊测试工具进行了总结,根据工作原理将内核模糊测试工具分为了基于随机生成、基于类型知识、基于钩子以及基于反馈驱动四种类型. 另外,还有一类是基于内核漏洞状态感知的模糊测试工具,这种模糊测试的测试例输入可以是运行正常的程序,通过对内核状态的监控,进而捕获信息泄露、内存错误等类型的漏洞. 值得注意的是,随着移动互联网和物联网的发展,模糊测试也成为挖掘这些平台漏洞的通用方法[18,37].

3.1 面向内核数据处理接口模糊测试工具

操作系统内核需要处理很多从硬件以及用户空间中传入的各种输入数据,从内核安全的角度来看,这些数据输入的接口都可以看做是系统内核的攻击面. 部分模糊测试工具利用AFL[3]等针对普通软件模糊测试工具将用户态生成的随机数据输入到系统文件镜像挂载、音频视频处理等接口中,期望发现漏洞. 比较典型工具有KernelFuzzing[4]、kAFL[13] TriforceLinuxSyscallFuzzer[5]、JANUS[12]等等. kAFL创造性的使用Intel PT[6](Processor Trace)和Intel VT-x两个硬件特性,实现了在虚拟机中使用PT技术对内核进行动态跟踪,大大的提高了针对内核模糊测试的效率. 通过随机填充、比特翻转、字节翻转、代数变换、兴趣字段探测、段拼接等方法生成文件系统镜像,通过不断的挂载镜像文件对文件系统进行漏洞挖掘. 同时,kAFL也使用AFL的位图(Bitmap)来记录测试例输入后内核程序控制流边,实现了覆盖率反馈机制. JANUS是一个专门针对文件系统进行模糊测试的工具. 这个工作创造性地采用了Linux Kernel Library将Linux文件子系统转换到用户态单独测试,减少了操作系统中其他子系统的影响,极大的提高了漏洞复现能力. JANUS还使用对文件镜像和文件操作两个维度进行模糊测试,有效增加了代码覆盖率.

3.2 针对系统调用接口的模糊测试工具

模糊测试技术是为了测试Linux系统的健壮性而发明[2]的. 针对系统调用接口的模糊测试工具可以追溯到1991年tsys程序,这个程序循环生成不正常的系统调用参数,通过syscall函数将这些参数交给系统调用执行,从而实现触发系统崩溃. 由于系统调用数目较多,大部分模糊测试工具都将目标聚焦到部分系统调用上. Perf_fuzzer[23]是一个专门针对Linux系统Perf_events工具接口进行模糊测试的工具,使用参数和系统调用等知识来生成测试例,主要用于解决Linux内核引入perf工具以后导致的内核频繁崩溃问题. Trinity[7]将系统调用参数的数据类型、参数值接受范围、接受的特定参数值等参数格式知识加入到测试例生成过程中,大大的提高了测试例质量. IMF[7]重点关注了系统调用之间的依赖关系对于漏洞挖掘的影响. IMF通过对MacOS上运行的程序进行动态跟踪,记录系统调用的执行顺序和参数使用. 将存在参数数据依赖和系统调用顺序依赖的系统调用识别出来,生成系统调用依赖模型,并使用模型来生成用于模糊测试的系统调用序列.

Syzkaller[8]是Google开发的内核漏洞模糊测试平台,规模相当庞大,能够对多种操作系统平台进行模糊测试工作. Syzkaller使用一系列的系统调用模板生成符合一定规范的系统调用序列,将测试例生成部分和测试例执行部分都放到虚拟机中,生成可以合法访问的缓存区等内存对象参数. 此外,Syzkaller还对整个虚拟机管理做了很大的改进,使得Syzkaller可以在QEMU、Google云平台、KVM等多种虚拟化及硬件调试平台上运行,并且能够大规模集群化的运行在大量主机上. 值得注意的是,Syzkaller实现了一个自动化地对Linux内核最新版本进行模糊测试的一个Syzbot[9]组件,并运行在Google公司的云平台上,已经自动化地挖掘出3000多个各类系统内核缺陷. Syzkaller是一个优秀的内核漏洞挖掘平台,其Linux内核上实现的源码插桩覆盖率记录、运行集群管理、漏洞捕捉技术、系统调用函数模板、漏洞复现代码自动生成等工作极大的方便了内核漏洞挖掘. 目前有大量研究都是针对Syzkaller平台在各个方面进行的改进,如静态分析系统调用之间依赖关系提升模糊测试覆盖率的Moonshine[24],以及结合静态分析专门挖掘设备驱动漏洞的DIFUZE[30]等等.

3.3 基于感知的内核模糊测试工具

基于感知的内核模糊测试工具的主要思路是通过虚拟化等方法,提取内核活动中可能存在漏洞状态的语义,捕获内

[3] http://lcamtuf.coredump.cx/afl/
[4] https://github.com/oracle/kernel-fuzzing
[5] https://github.com/nccgroup/TriforceLinuxSyscallFuzzer
[6] https://software.intel.com/en-us/node/721535
[7] https://github.com/kernelslacker/trinity
[8] https://github.com/google/syzkaller
[9] https://syzkaller.appspot.com

核漏洞. 这种方法在检测信息泄漏类型的漏洞方面有比较好的效果. Bochspwn[12] 利用 Bochs 模拟器模拟执行 Windows 等目标系统,对系统内核数据进行预先的污染,并对内核空间传递到用户空间的数据进行检查,通过这种污点跟踪的技术,Bochspwn 不仅能对能够对内核流出到文件系统的泄露进行探测,还能探测到二次获取和二次写入等逻辑漏洞. 与之类似,Digtool[14] 实现了一套虚拟机监控程序,利用硬件虚拟化和 Intel PT 等功能,极大的提升了监控系统调用接口和内核活动的能力,可以方便的收集整个内核的运行情况.

3.4　结合代码静态分析的内核模糊测试

在内核漏洞挖掘领域,代码静态分析作为软件安全白盒测试中最为重要的方法,出现了很多研究成果. 通过结合代码静态分析覆盖率高、检测结果可靠性(Soundness)强等优点,可以有效的补足模糊测试的缺点,提升模糊测试效果. Moonshine[26] 与 IMF 思路较为相似,使用静态分析工具对系统调用的内核代码实现进行依赖性分析,发掘内核状态读写方面依赖,然后基于这些依赖从 trace 中提取系统调用测试序列,输入到 Syzkaller 中执行,提高了模糊测试的代码覆盖率. Razzer[40] 主要针对内核中的数据竞争漏洞进行挖掘,利用 LLVM[10] 对 Linux 代码进行静态分析,找到代码中可能发生数据竞争的指令对. 通过定制化地修改 Syzkaller 和 QEMU,实现了生成包含潜在发生数据竞争代码的测试例,精确地执行数据竞争中特定指令对,并验证其竞态访问后果,从而找到可以发生竞争的代码.

3.5　针对驱动程序的模糊测试

内核驱动程序模糊测试主要针对的 read、write、ioctl 等系统调用进行模糊测试. 而没有专门对应系统调用函数的 Windows 系统,也可以采用相似的方法,将模糊器生成的数据传入到设备控制、输入和输出接口中[10-14].

DIFUZE[30] 使用代码静态分析方法,提取 IOCTL 系统调用接口的数据结构和控制命令信息,生成更为符合 IOCTL 系统调用接口规定的测试例,进而提高模糊测试的覆盖率. 在 Syzkaller 的基础上,DIFUZE 实现了一整套利用驱动程序知识提高模糊测试能力的框架. Charm[38] 主要实现了一种能够在 X86 体系的主机上对 ARM 移动设备系统驱动进行模糊测试的架构,通过虚拟化等方法将 Android 设备上的驱动程序在 X86 平台上重新编译运行,实现了利用 X86 平台上内核模糊测试工具对 Android 设备驱动进行模糊测试. PeriScope[39] 对硬件不可信输入造成的影响进行了研究,将驱动程序接收外部设备数据的接口作为模糊测试的目标,将硬件输入数据作为测试例,对采用 MMIO 和 DMA 通信的驱动程序进行分析,并基于 AFL 和定制的 KCOV[15] 实现了整个的模糊测试循环.

4　内核模糊测试中的关键技术

在内核模糊测试框架的搭建中,针对目标操作系统内核,需要整体设计测试目标接口、测试例生成方法、测试例输入执行方式、反馈机制等方面. 内核模糊测试框架的搭建与普通应用程序模糊测试比较类似,有研究将用户态模糊器作为输入生成器,也有部分研究另起炉灶,直接使用基于系统调用接口知识的方式生成测试例. 为了实现高效的漏洞挖掘,还需要重点考虑被测内核运行方式、内核漏洞触发状态感知、以及实现覆盖率引导等几个方面的问题.

4.1 虚拟化技术

一般来说,操作系统内核只能运行在计算机体系特权级较高的层级上. 2015 年以前的内核模糊测试研究[16,19,22] 主要为利用钩子(HOOK)技术对 Windows、MacOS 等操作系统部分函数进行修改,实现输入测试例和监控系统崩溃功能. 但为了实现对内核状态的较为方便监控,就需要采用虚拟化技术将让被测内核降级到用户态运行. 在内核模糊测试中一般使用软件虚拟化和硬件虚拟化平台运行被测内核,常见的平台有 QEMU 和 KVM 等. 文献[17,20]研究了使用虚拟化技术运行被测目标系统,文献[13,40]等研究采用定制虚拟化工具的方法实现黑盒系统的覆盖率生成、执行漏洞触发关键代码等功能. Panda[16] 研究了利用虚拟化技术记录漏洞生命周期,完整收集漏洞状态语义,极大的方便了内核漏洞的调试. 另外,基于感知的内核模糊测试工具基本上就是围绕虚拟化技术展开研究的. 从开始的系统加载到漏洞状态检测,此类工具都需要透过虚拟机或模拟器对内核的运行状态进行监控.

4.2　内核漏洞状态感知技术

内核漏洞的感知技术也是内核漏洞研究的一个重要方面,并且极大的决定着内核漏洞挖掘的效率. 内核漏洞感知技术既需要精确地捕捉到内核漏洞触发时的寄存器状态、堆栈状态等底层硬件信息,还要对漏洞发生时相关内存对象分配释放信息、内核栈信息、线程信息等上层漏洞语义进行记录. Linux 系统上实现了一系列的内核安全违例检查工具,这些工具都受到了 PaX[17] 项目以及用户态程序漏洞检测技术的启发,利用雷区(Red Zone)等技术对 Linux 内核中内存读写、线程创建与销毁等可能导致漏洞的行为进行探测. 其中以 KASAN[18](Kernel Address Sanitizer)和 UBSAN[19](Undefined-Behavior Sanitizer)最为成熟,已经可以直接应用到 Linux 内核中,可以通过对内存对象读写监控、内存对象周围监控、内存延迟回收等内核活动实现内存错误探测,发现内核中存在

[10] https://llvm.org
[11] https://github.com/Cr4sh/ioctlfuzzer.
[12] https://github.com/debasishm89/iofuzz
[13] https://docs.microsoft.com/en-us/windows-hardware/drivers/devtest/how-to-perform-fuzz-tests-with-iospy-and-ioattack
[14] https://code.google.com/archive/p/ioctlbf/
[15] http://simonkagstrom.github.io/kcov
[16] https://github.com/panda-re/panda
[17] https://pax.grsecurity.net/
[18] https://www.kernel.org/doc/html/latest/dev-tools/kasan.html
[19] https://www.kernel.org/doc/html/latest/dev-tools/ubsan.html

的堆栈移除、UAF(Use-after-free)和越界读写漏洞,并打印输出漏洞信息. KMSAN[20](Kernel Memory Sanitizer)和 KTSAN[21](Kernel Thread Sanitizer)还都处于原型开发阶段,主要用于未初始化内存使用和竞态条件错误的探测. 另外,针对信息泄漏漏洞,可以配合内核数据污染,采用污点分析的方法进行漏洞挖掘.

4.3 代码覆盖率计算技术

模糊测试中代码覆盖率指的是输入的测试例能够让目标程序的控制流途经代码的数量,一般计数的单位都是控制流上基本块. 目前内核模糊测试代码覆盖率计算主要方式包括源码插桩、Intel PT 控制流记录、单步执行、性能监控单元(PMU)取样、动态二进制翻译等方式[21]. KCOV 是 Syzkaller 项目组在 GCC 编译器中实现的一种针对 Linux 内核的源码插桩覆盖率检测工具,专门配合 Syzkaller 等模糊测试工具进行覆盖率测量. 通过开启特定的 Linux 内核编译选项,记录执行系统调用产生的内核代码覆盖率. KCOV 记录的数据包括内核代码基本块的地址和这些地址的数量. 通过将每一个地址与上一个地址进行哈希等运算生成控制流图中的边,然后以此计算整个测试例对内核代码的覆盖率. Intel Processor Trace(Intel PT)是 Intel 第五代(Broadwell) CPU 上添加的一个新特性,实现了使用硬件对程序控制流进行记录,提高了代码跟踪的效率. 通过使用 Intel PT 对内核控制流进行记录,通过解码获取控制流覆盖率的基本块,就可以高效地记录测试例对应的内核代码覆盖率,不需要对源码进行处理,并且相对于动态二进制翻译极大的提高了效率.

5 总结与展望

操作系统作为信息基础设施中地位不可动摇的可信计算基,使得针对操作系统内核漏洞的研究一直以来都是安全研究的热点. 对内核漏洞的深入研究也推动了内核安全防御机制的快速发展,而安全机制的发展反过来又促进研究者去发掘能够绕过安全机制的方法和漏洞. 内核模糊测试作为高效的内核漏洞挖掘工具,在近年来得到了较好的发展[22],各种新技术新做法都被应用到内核模糊测试中,并且取得了良好的效果. kAFL 在多个系统平台上发现了 8 个文件系统漏洞,IMF 在 Mac OS 上发现了 32 个内核漏洞,Razzer 在 Linux 系统内核中发现了 30 个竞态漏洞. 值得一提的是,截止 2019 年 4 月,Syzkaller 的自动化挖掘平台 syzbot 在 Linux 系统内核中发现了 1200 多个软件脆弱点,极大的提高了内核的安全性.

目前,实践中内核模糊测试的主要局限性还是在于操作系统内核复杂性导致的一系列问题上. 首先内核模糊测试还需要能力更强的漏洞检测方法. 由于内核运行状态较为复杂,大量的子系统事务相互影响,目前比较成熟的 KASAN 在检测内存错误也会存在无法准确还原漏洞语义的情况,而针对未初始化内存、数据竞争、未定义行为等漏洞的检测更是存在较大困难,这也使得漏洞调试、复现等环节收到了巨大的限制. 其次,目前的内核模糊测试工具除 Syzkaller 系列以外都没有正面面对操作系统内核规模庞大、接口众多的问题. Syzkaller 虽然能够一定程度自动化提取系统调用、系统库函数等内核调用接口,但仍然需要人工编写测试例生成的模版. 最后,模糊测试固有的测试例质量不高的问题. 目前 IMF、Moonshine 等工具虽然已经对这个问题进行了研究,但在解读内核中纷繁复杂的状态关系上仍然还有较大的改进空间.

由于技术实现难度等原因,内核模糊测试的发展相对于普通应用程序模糊测试还是有一定的滞后性,将普通应用程序模糊测试已有的方法应用在内核模糊测试上是一种比较经济的改进方法. 普通应用程序的模糊测试在 AFL 等工具的基础上使用了多种方法改进了代码覆盖率的表示形式. 如 AFL-Fast[5]、Vuzzer[29]、CollAFL[9]等工具利用定向模糊测试、有向图算法、静态分析等方法改进了原来基于控制流图边的覆盖率表示算法,提高了漏洞挖掘的有效性. 在测试例生成和执行方面,Angora[11]、T-Fuzz[10]等工具使用的利用梯度下降算法和强制执行流翻转算法提高了测试例的质量. 另外,随着神经网络技术的发展,也有研究[31-33]在模糊测试中应用深度学习技术提高模糊测试能力. 这些在普通应用程序模糊测试中行之有效的方法都可以尝试应用到内核模糊测试中. 从长期来看,内核模糊测试未来除了可以在传统的虚拟化的基础上进行更加细致的研究,采用类似于 JANUS 的方法也是一种非常好的选择,将内核子系统或驱动抽离单独对其进行模糊测试,提高漏洞复现能力和测试效率.

References:

[1] Anderson J P. Computer security technology planning study [R]. Anderson(James P) and Co Fort Washington PA,1972.

[2] Miller B P,Fredriksen L,So B. An empirical strudy of the reliability of UNIX utilities [J]. Communication of the ACM,1990,(33):32-44.

[3] Aslam T. A taxonomy of security faults in the unix operating system [D]. West Lafayette,Purdue University,1995.

[4] Johnson R,Wagner D. Finding user/kernel pointer bugs with type inference [C]//USENIX Security Symposium,2004:119-134.

[5] Maraia V. The build master: microsoft's software configuration management best practices [M]. Pearson Education India,2006.

[6] Li J,Zhao B,Zhang C. Fuzzing: a survey [J]. Cybersecurity,2018,1(1):1-13.

[7] Han H S,Sang K C. IMF: inferred model-based fuzzer [C]// ACM Sigsac Conference,ACM,2017:2345-2358.

[8] Lu K,Lee W,Nürnberger S,et al. How to make ASLR win the clone wars: runtime re-randomization [C]// Proceedings of the Network and Distributed System Security Symposium (NDSS),2016.

[9] Gan S,Zhang C,Qin X,et al. CollAFL: path sensitive fuzzing [C]//IEEE Symposium on Security and Privacy(SP),IEEE,2018:679-696.

[10] Peng H,Shoshitaishvili Y,Payer M. T-Fuzz: fuzzing by program transformation [C]//IEEE Symposium on Security and Privacy (SP),IEEE,2018:697-710.

[11] Chen P,Chen H. Angora: efficient fuzzing by principled search [C]//IEEE Symposium on Security and Privacy(SP),IEEE,

[20] https://github.com/google/kmsan

[21] https://github.com/google/ktsan

[22] https://moflow.org/Presentations/Evolutionary%20Kernel%20Fuzzing-BH2017-rjohnson-FINAL.pdf

2018: 711-725.
[12] Jurczyk M. Detecting kernel memory disclosure with x86 emulation and taint tracking [R]. INFILTRATE, Miami, 2018.
[13] Schumilo S, Aschermann C, Gawlik R, et al. kafl: Hardware-assisted feedback fuzzing for OS kernels [C] // 26th USENIX Security Symposium, 2017: 167-182.
[14] Pan J, Yan G, Fan X. Digtool: A virtualization-based framework for detecting kernel vulnerabilities [C] //26th USENIX Security Symposium, 2017: 149-165.
[15] You W, Zong P, Chen K, et al. SemFuzz: semantics-based automatic generation of proof-of-concept exploits [C] //Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, ACM, 2017: 2139-2154.
[16] Yao Hong-bo, Yin Liang, Wen Wei-ping. Based on the fuzzing lead to a new mining method based on windows kernel vulnerability [J]. Netinfo Security, 2011, (12): 9-16.
[17] Zhao Yue-hua, Deng Yuan-hao. Research and implementation on monitoring race-condition vulnerablity in kernel based on virtual machine monitor [J]. Software Guide, 2015, (5): 161-164.
[18] He Yuan, Zhang Yu-qing, Zhang Guang-hua. Android driver vulnerability discovery based on black-box genetic algorithm [J]. Chinese Journal of Computers, 2017, 40(5): 1031-1043.
[19] Ni Tao. Research on key technologies for detection and exploitation of windows kernel vulnerabilities [D]. Zhengzhou: Information Engineering University, 2013.
[20] Deng Yuan-hao. Design and implementation on monitoring kernel-vulnerability system based on virtual machine monitor [D]. Zhenjiang: Jiangsu University, 2015.
[21] Xu Yong-jian. Research on detecting vulnerabilities in linux driver [D]. Beijing: Beijing University of Technology, 2015.
[22] Qin Bi-shi. Research on detection and analysis of vulnerability of windows device drivers [D]. Beijing: University of Chinese Academy of Sciences, 2015.
[23] Weaver V M, Jones D. Perf fuzzer: targeted fuzzing of the perf event open() system call [R]. Technical Report UMAINEVMW-TR-PERF-FUZZER, University of Maine, 2015.
[24] Xu M, Qian C, Lu K, et al. Precise and scalable detection of double-fetch bugs in OS kernels [C] //IEEE Symposium on Security and Privacy(SP), 2018: 270-287.
[25] Wang P, Krinke J, Lu K, et al. How double-fetch situations turn into double-fetch vulnerabilities: a study of double fetches in the Linux kernel [C] //USENIX Security Symposium, 2017: 1-16.
[26] Pailoor S, Aday A, Jana S. MoonShine: optimizing OS fuzzer seed selection with trace distillation [C] //27th USENIX Security Symposium, 2018: 729-743.
[27] Stephens N, Grosen J, Salls C, et al. Driller: augmenting fuzzing through selective symbolic execution [C] // Proceedings of the Network and Distributed System Security Symposium(NDSS), 2016: 1-16.
[28] Yun I, Lee S, Xu M, et al. QSYM: a practical concolic execution engine tailored for hybrid fuzzing [C] //27th USENIX Security Symposium, 2018: 745-761.
[29] Rawat S, Jain V, Kumar A, et al. Vuzzer: application-aware evolutionary fuzzing [C] //Proceedings of the Network and Distributed System Security Symposium(NDSS), 2017.
[30] Corina J, Machiry A, Salls C, et al. Difuze: interface aware fuzzing for kernel drivers [C] //Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, ACM, 2017: 2123-2138.
[31] Hu Z, Shi J, Huang Y H, et al. GANFuzz: a GAN-based industrial network protocol fuzzing framework [C] //Proceedings of the 15th ACM International Conference on Computing Frontiers, ACM, 2018: 138-145.
[32] Odena A, Goodfellow I. TensorFuzz: debugging neural networks with coverage-guided fuzzing [J]. arXiv preprint arXiv: 1807.10875, 2018.
[33] Godefroid P, Peleg H, Singh R. Learn&fuzz: machine learning for input fuzzing [C] //Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, IEEE Press, 2017: 50-59.
[34] Chen H, Xue Y, Li Y, et al. Hawkeye: towards a desired directed grey-box fuzzer [C] //ACM Conference on Computer and Communications Security, 2018: 2095-2108.
[35] Pham V T, Roychoudhury A. Coverage-based greybox fuzzing as markov Chain [C] //Acm Sigsac Conference on Computer & Communications Security, ACM, 2016.
[36] Schwarz M, Gruss D, Lipp M, et al. Automated detection, exploitation, and elimination of double-fetch bugs using modern CPU features [C] //Proceedings of the 2018 on Asia Conference on Computer and Communications Security, ACM, 2018: 587-600.
[37] Chen J, Diao W, Zhao Q, et al. Iotfuzzer: discovering memory corruptions in iot through app-based fuzzing [C] // Proceedings of the Network and Distributed System Security Symposium (NDSS), 2018.
[38] Talebi SMS, Tavakoli H, Zhang H, et al. Charm: facilitating dynamic analysis of device drivers of mobile systems [C] //27th USENIX Security Symposium, 2018: 291-307.
[39] Song D, Hetzelt F, Das D, et al. PeriScope: an effective probing and fuzzing framework for the hardware-OS boundary [C] // Proceedings of the Network and Distributed Systems Security Symposium(NDSS), Internet Society, 2019: 1-15.
[40] Jeong D R, Kim K, Shivakumar B, et al. Razzer: finding kernel race bugs through fuzzing [C] //Proceedings of the IEEE Press Symposium on Security and Privacy(SP), 2019: 296-310.
[41] Bai S, Li D, Huang M, et al. Synthesis of linux kernel fuzzing tools based on syscall [C] //DEStech Transactions on Computer Science and Engineering, 2017: 610-618.
[42] Xu W, Moon H, Kashyap S, et al. Fuzzing file systems via two-dimensional input space exploration [C] // IEEE Symposium on Security and Privacy(SP), 2019: 594-610.

附中文参考文献:

[16] 姚洪波, 尹　亮, 文伟平. 基于 FUZZING 测试技术的 Windows 内核安全漏洞挖掘方法研究及应用 [J]. 信息网络安全, 2011, (12): 9-16.
[17] 赵跃华, 邓渊浩. 基于硬件虚拟化的内核竞态漏洞监测技术研究与实现 [J]. 软件导刊, 2015, (5): 161-164.
[18] 何　远, 张玉清, 张光华. 基于黑盒遗传算法的 Android 驱动漏洞挖掘 [J]. 计算机学报, 2017, 40(5): 1031-1043.
[19] 倪　涛. Windows 内核漏洞检测与利用关键技术研究 [D]. 郑州: 解放军信息工程大学, 2013.
[20] 邓渊浩. 基于硬件虚拟化的内核漏洞监测系统的设计和实现 [D]. 镇江: 江苏大学, 2015.
[21] 徐永健. Linux 内核驱动中漏洞检测的研究 [D]. 北京: 北京工业大学, 2015.
[22] 秦弼时. Windows 设备驱动漏洞检测与分析技术研究 [D]. 北京: 中国科学院大学(工程管理与信息技术学院), 2015.