

学校代码: 10004

密级: 公开

北京交通大学

BEIJING JIAOTONG UNIVERSITY

硕士学位论文

基于内核函数监控的 Linux 系统防护方法的
研究与实现

作者姓名 刘 晨

学科专业 计算机科学与技术

指导教师 翟高寿 副教授

培养院系 计算机与信息技术学院

二零一八年三月

北京交通大学

硕士学位论文

基于内核函数监控的 Linux 系统防护方法的研究与实现

Study and Implementation of System Protection by Monitoring Abnormal
Invocation of Linux Kernel Functions

作者：刘 晨

导师：翟高寿

北京交通大学

2018 年 3 月

学位论文版权使用授权书

本学位论文作者完全了解北京交通大学有关保留、使用学位论文的规定。特授权北京交通大学可以将学位论文的全部或部分内容编入有关数据库进行检索，提供阅览服务，并采用影印、缩印或扫描等复制手段保存、汇编以供查阅和借阅。同意学校向国家有关部门或机构送交论文的复印件和磁盘。学校可以为存在馆际合作关系的兄弟高校用户提供文献传递服务和交换服务。

（保密的学位论文在解密后适用本授权说明）

学位论文作者签名：刘晨

签字日期：2018年6月12日

导师签名：翟高寿

签字日期：2018年4月12日

北京交通大学
硕士学位论文

基于内核函数监控的 Linux 系统防护方法的研究与实现

Study and Implementation of System Protection by Monitoring
Abnormal Invocation of Linux Kernel Functions

作者姓名: 刘 晨

学 号: 15120418

导师姓名: 翟高寿

职 称: 副教授

学位类别: 工学

学位级别: 硕士

学科专业: 计算机科学与技术

研究方向: 操作系统安全

北京交通大学

2018 年 3 月

致谢

本论文是在我的导师翟高寿老师悉心指导下完成的。翟老师严谨的治学态度、踏实勤恳的作风以及深厚的学术功底使我感触颇深、受益良多，故在此对翟老师表示由衷的感谢。攻读硕士学位期间，从起初的专业方向学术调研与选题，到研究工作的开展与实践以及最后所有研究工作的完成，翟老师都给予了我莫大的帮助和耐心的指导，特别是每次科研遇到困难举步维艰时，翟老师提出的意见与建议总能帮助我找到解决问题的突破口。这些使我的专业知识水平和科研工作能力都得到了很大的提高，也将让我受益终生，再次对翟老师表示衷心的感谢。

此外，在论文撰写期间，我也得到了清华大学向勇老师的指导，以及实验室同学和朋友们的帮助。正是由于他们的关心，我才能以最快的速度获取知识与灵感，在此对他们也表示深深的感谢。

最后，我要感谢我的家人，正是由于他们一如既往的支持和鼓励，我才能够潜心进行科研工作，专心完成学业。

摘要

伴随计算机技术的广泛使用,信息安全问题日益突出,信息系统安全越来越受到人们的重视。鉴于操作系统在计算机系统中所承担的关键作用,整个计算机系统的安全在很大程度上依赖于操作系统的安全。近些年, Linux 操作系统凭借自身独特的优势,迅速地提升了其在服务器操作系统市场上的份额。但也正因如此,针对 Linux 系统的恶意攻击也在不断增加。与此同时, Linux 内核的漏洞也随其代码量的快速增长而持续地被曝光出来。而内核作为操作系统中最为核心的部分,其安全则是操作系统乃至整个计算机系统安全的基础前提和根本保障。所以, Linux 操作系统安全尤其是内核安全已成为计算机系统安全领域研究的焦点之一。

本文以运行 Linux 系统的服务器为研究对象,在分析研究现有各类 Linux 系统防护方法的基础上,提出了一种基于内核函数监控的系统防护方法,试图通过限制相关服务进程所能访问的内核函数范围,加大恶意攻击的难度进而增强 Linux 内核安全,同时通过对内核函数各种异常调用情况的分级分类实时处理,从而提升整个服务器系统的安全水平。其间,内核函数监控集的获取利用了 ftrace 动态跟踪技术,而内核函数的监控及异常情况分析处理则综合运用了 Kprobes 运行时调试技术和 Zabbix 报警响应机制,并且,内核函数监控系统原型是通过分析监控模块构成要素及特征模式而自动构建的。此外,本文还提出了一种内核函数安全风险分级标准,并给出了原型应用环境下的相应划分示例。进一步说,在分析和参考有关系统调用的安全分级的基础上,结合相关已知内核漏洞等因素将内核函数划分为高、中、低三个潜在安全风险等级,然后在异常情况处理机制中设立相对应的三种处理方式,从而支持各种异常情况的分级分类处理。

针对有关 Linux 系统防护原型的功能测试与性能测试的实验结果表明,本文方法能够及时检测到相关服务进程对内核函数的异常调用情况,并给以适当的报警或拦截处理,而且由此带来的额外开销完全可以承受,从而验证了有关方法的可行性和有效性。与内核安全防护的其它研究工作相比,本文方法所涉内核防护覆盖范围更大且无需重新编译构建内核映像,并切实做到了监测与防护的有机结合。

关键词: 操作系统安全; 内核安全; 系统防护方法; 服务进程; 内核函数调用
分类号: TP316; TP309

ABSTRACT

With the widespread use of computer technology, the problem of information security is becoming highlighted. Information system security is paid more and more attention by people. In view of the key role that operating system plays in computer system, the security of the entire computer system relies heavily on operating system security. In recent years, the Linux operating system has rapidly gained shares in the server operating system market with its unique advantages. Because of this reason, however, targeted malicious attacks have increased. At the same time, with the rapid growth of the amount of codes in the Linux kernel, its vulnerabilities are continuously exposed. The kernel is the core part of the operating system and its security is the fundamental premise to guarantee the security of operating system and the entire computer system. Therefore, Linux security especially its kernel security has become one of the research focuses in the fields of computer system security.

As for the server running Linux system, a system protection method by the way of monitoring kernel functions is proposed in this paper based on the analysis of various existing Linux system protection methods. It limits the kernel functions that can be accessed by the related daemons and increases the difficulty of malicious attacks so as to enhance the security of Linux kernel. Moreover, some real-time categorical processing is introduced for various abnormal invocations to the kernel functions so that the security level of the entire server system is promoted. During the implementation of the prototype, the dynamic tracing technology of **ftrace** is utilized for the extraction of the kernel functions to be monitored, while the runtime debugging technology of **Kprobes** and the alarm response mechanism of **Zabbix** are used for monitoring the kernel functions and dealing with abnormal invocations respectively. Moreover, the prototype for monitoring kernel functions are constructed automatically based on the basic elements and characteristic patterns of related monitoring mechanisms.

In addition, this paper also presents a classification standard about security risk for kernel functions, and gives examples of corresponding classification under the prototype application environment. In another word, the kernel function is divided into three levels of high, medium and low potential security risks based on the analysis and reference to the security classification of system calls and combined with related known kernel vulnerabilities and other factors. Then the corresponding three kinds of processing ways in the handling mechanism are set up to support the classification of various abnormalities

processing.

The experimental results of functional tests and performance tests on Linux system protection prototype show that the proposed method can indeed detect the abnormal invocations of the kernel functions timely followed by some appropriate alarming or interception measures. Furthermore, the overloads are not too much such that the method is verified to be feasible and effective. Compared with other research work about kernel security, this method can protect broader kernel coverage and it does not need to recompile and reconstruct the kernel image while kernel monitoring and protection mechanisms are integrated organically.

KEYWORDS: security of operating systems; security of kernel; system protection; daemon; invocation of kernel functions

CLASSNO: TP316; TP309

目录

摘要	V
ABSTRACT.....	VII
1 绪论	1
1.1 研究背景.....	1
1.2 国内外研究现状.....	2
1.3 研究内容与技术路线.....	3
1.4 论文组织结构.....	4
2 研究基础	7
2.1 FTRACE 动态跟踪技术	7
2.1.1 Ftrace 概述	7
2.1.2 Ftrace 组成及跟踪原理	8
2.1.3 Ftrace 配置与使用	10
2.2 KPROBES 内核调试技术.....	12
2.2.1 Kprobes 概述.....	12
2.2.2 Kprobes 探测原理	13
2.2.3 Kprobes 启用及相关接口.....	16
2.3 ZABBIX 报警响应机制.....	17
2.4 本章小结.....	19
3 内核函数潜在安全风险等级的研究与划分	21
3.1 系统调用分类及潜在安全风险分级.....	21
3.1.1 系统调用概述	21
3.1.2 总体分类与分级标准	22
3.1.3 系统调用安全风险等级分级示例	22
3.2 内核函数潜在安全风险分级.....	27
3.2.1 高风险系统调用对应的内核函数	28
3.2.2 存在已知漏洞的内核函数	34
3.2.3 安全敏感内核函数	34
3.2.4 低安全风险内核函数	35
3.3 本章小结.....	36

4	系统防护原型的设计与实现.....	39
4.1	总体设计	39
4.2	待监控内核函数集获取方法	41
4.2.1	内核函数白名单获取方法	41
4.2.2	频繁被调内核函数集获取方法	42
4.3	基于 KPROBES 的内核函数实时监控及自动构建	43
4.4	内核函数异常调用情况分类处理	45
4.4.1	基于内核监控模块的异常处理	45
4.4.2	基于 Zabbix 的异常告警与处理	47
4.5	原型构建	49
4.5.1	针对特定服务进程的内核函数白名单的获取	49
4.5.2	频繁被调内核函数集的跟踪选取	50
4.5.3	内核函数监控集的计算确立	52
4.5.4	监控模块的自动构建	52
4.6	本章小结	53
5	原型测试与结果分析	55
5.1	功能测试与分析论证	55
5.1.1	特权服务进程攻击防护测试验证	55
5.1.2	非特权服务进程攻击防护分析论证	58
5.2	性能测试	59
5.2.1	时延测试	59
5.2.2	带宽测试	61
5.3	相关工作比较	61
5.4	本章小结	62
6	总结与展望	63
6.1	工作总结	63
6.2	研究展望	64
	参考文献	65
	附录 A	69
	作者简历及攻读硕士学位期间取得的研究成果	71
	独创性声明	73
	学位论文数据集	75

1 绪论

1.1 研究背景

在互联网时代,计算机已成为人们工作与生活中不可或缺的一部分,而操作系统作为计算机系统软硬件之间的桥梁和计算机软件层的基石,它不仅负责系统中软硬件之间的交互与衔接,同时也为各类应用程序提供运行平台与构建基础,起着极为重要的作用,所以其可靠性与安全性就变得格外的重要。

Linux 操作系统因其完备的功能、强大的性能、良好的兼容性以及开源免费等优点,已被人们广泛应用在各类计算机特别是服务器上。但随着 Linux 操作系统在服务器市场上的份额逐渐增大,它也吸引来大量有针对性的恶意攻击行为,其存在的安全问题也日渐凸显出来。

顾名思义,服务器的主要作用是对外提供某些服务,其上开启的端口与相应的服务进程是它对外暴露的接口,也几乎是外界访问系统的唯一入口。所以一个企图入侵服务器的攻击者往往会选择系统中某个服务进程,并使用某些手段尝试对其进行控制。成功之后,攻击者便能以之作为跳板来达到篡改敏感信息甚至窃取超级管理员权限进而控制整台机器等非法目的。一般来说,服务进程的功能较为单一,故而其正常运行所需的内核函数相当有限。但当它被恶意控制之后,攻击者往往需要借助内核中的其它功能代码或是内核漏洞来进一步实施对系统的恶意攻击。

然而,伴随 Linux 操作系统的日渐完善,其内核代码量也在急剧增长。这导致内核变得庞大臃肿的同时,也给系统带来了较大的安全隐患,并为攻击者达成其目的提供了更多的便利条件。一方面,为了实现对老旧功能和各类设备的兼容,内核保留了大量过时和冗余的代码。另一方面,由于一些功能代码的设计缺陷总是难以避免,内核也因此引入了许多漏洞。常见漏洞公开库 CVE^[1]是安全业界权威的“漏洞字典”,近些年由它曝光的 Linux 内核漏洞在数量上总体呈现出逐年增长的趋势,其中既包括可被利用来探查内核中敏感数据的漏洞,也包括可被用来窃取系统最高权限的高危漏洞。而这些已公开甚至仅仅掌握在部分人手中未公开的内核漏洞也就成为了攻击者入侵系统的有力工具。

鉴于此, Linux 操作系统在提供强大功能与兼容性的同时如何保证较高的安全性就成为了一个亟待解决的问题。另外,考虑到安全需从根本上进行保障,而不仅仅是依靠杀毒软件、防火墙等外部拦截手段,所以提出一种深入内核层面的 Linux 系统防护方法是很有必要的。

1.2 国内外研究现状

异常检测、访问控制和沙箱技术是在操作系统安全增强或加固过程中经常采用的安全防护机制。

基于主机的异常检测系统^[2-7]以事先定义的正常行为作为评判标准来判断进程当下的行为是否正常。相关方法往往根据进程运行时产生的系统调用及相关信息,并结合高效的统计学理论、数据挖掘以及人工智能等技术^[8]来形成对正常行为的描述,可以检测出未知的攻击。然而,这类系统大多数情况下是基于用户空间设计的,并没有深入到内核层面,且通常容易被攻击者绕过^[9]。

访问控制技术^[10]使用预设的安全策略,对数据和文件等资源加以控制,防止无权用户或低权限用户访问关键、敏感的数据,具有性能开销小、效率高等特点。SELinux 是 Linux 下著名的强制访问控制机制。进程在这种体系的限制下,只能根据预设的主体对客体的访问规则来访问其运行时需要的客体,通常只要进行合理的配置,这种机制就能够提供较高的安全性,但它也存在配置比较复杂,易用性较差^[11]等无法忽视的缺点。另外,实施访问控制的客体通常是系统中的文件、内存段页、存有敏感信息的结构体数据、硬件设备以及网络端口等,如果需要限制访问的对象是内核中的某些函数甚至是某一段代码,那么采用传统的访问控制技术并不合适。

沙箱技术^[12, 13]通常会结合访问控制的思想,根据预设的安全策略,构建可控的沙箱环境,将程序的所有或部分行为限制在沙箱内部,从而完全或部分隔离可疑程序的运行,所以具有较好的安全防护效果。但是,构建可控环境可能会带来较大的系统开销,而且,如何制定合理可行的安全策略,其本身也是一大难点或问题。

chroot^[14]是一种沿用至今的经典沙箱技术,它能够指定程序执行时的根目录,实现对进程活动范围的限制,从而将不可信程序与实际的文件系统隔离开,最典型的例子就是限定 ftp 服务存储文件的根目录。但使用 chroot 为进程设置的“监狱”也并非牢不可破,如果相关进程本身具有超级管理员权限,或者是能够在限制范围内利用内核漏洞等手段成功实现提权,就可以逃出 chroot 所指定的目录。

seccomp 技术则通过一定的规则限制进程对系统调用的使用范围,最初由 Andrea Arcangeli 设计实现并于 Linux2.6.12 加入内核^[15],是一种典型的基于系统调用的沙箱技术。但由于其对目标进程的影响和限制过于强烈,带来了如不能动态分配共享内存^[16]等副作用,使用不当有可能会使程序原本正常的功能无法使用,对使用者的要求较高。

程香鹏等人提出的 Linux 内核沙箱模块^[17]利用了内核内置的 LSM 访问控制框架^[18],针对进程敏感操作所涉及的关键内核函数设置自定义钩子函数,以检查进

程当前行为的合法性。不过，由于 LSM 框架本身的局限性，该方法只能在预设钩子的那些关键位置实施权限检查等操作，而并不能囊括千变万化的恶意代码执行逻辑以及内核漏洞利用方式。另外，LSM 在 2.6.22 内核之后不再支持运行时注册安全模块，这就意味着添加或修改安全模块需要重新编译和重构整个内核。

相比之下，内核裁剪与最小化技术^[19, 20]则可以从根本上降低内核面临的安全风险，同时能很好地解决上述系统安全防护机制所存在的一些问题。这种技术立足于从系统实际需要的具体功能出发，尽量删减其它非必要的功能代码，从而定制出当前功能需求下的最小化内核，故而能在很大程度上提高内核乃至整个系统的安全性。不过，相关裁剪操作需要重新编译内核，实际应用起来并不方便，所以该技术更加适用于功能需求固定的嵌入式系统，而不适用于服务器环境。

Anil Kurmus 等人所提出的内核攻击面缩减思想^[21]，通过限制程序所能访问的内核代码的数量来实现在内核运行时实施“裁剪”操作，进而避免传统的裁剪技术需要重新编译内核的缺点。基于这一思想实现的可量化的运行时攻击面缩减框架 kRazor^[22]，突破了目前常见系统防护方法大多仅停留在系统调用这一粗粒度监控层面的限制，深入到了内核监控层面。然而，鉴于其出发点主要在于实现对内核攻击面缩减和安全提升水平的量化，所以只是验证了内核监测及阻止恶意攻击的可行性，并未就相关内核异常情况的具体处理展开进一步的研究和探讨，算不上是一种真正意义上的能投入使用的 Linux 防护系统，离实际应用尚存在一定的距离。虽然 kRazor 确实存在一些局限性，但本文仍深受其攻击面缩减思想的启发，并在 kRazor 攻击面缩减流程的基础之上进行了改进与创新，试图设计与实现一个较为完善的 Linux 服务器防护系统。

1.3 研究内容与技术路线

本文在综合考虑前述各类系统防护方法基本思想及优势与不足的基础上，结合服务器的实际应用场景，以构建切实可行且完整有效的 Linux 服务器安全防护系统为目的，提出了基于内核函数监控的 Linux 系统防护方法。其中主要涉及到的研究内容如下：

(1) 提出一种内核函数潜在安全风险等级分级标准并给出相关示例。首先提出以系统调用所处的安全风险等级为主要参考依据来确定相关内核函数的潜在安全风险等级的思想。然后对 Linux 特定版本内核中三百多个系统调用进行了功能分类与潜在安全风险等级分级，同时调研与分析了该版本中内核漏洞情况。最后参照这些因素分析给出了一些内核函数的潜在安全风险等级的分级示例。

(2) 设计了基于内核函数监控的 Linux 系统防护方法。具体来说，首先借助

fttrace 动态跟踪技术确定系统中哪些内核函数需要被监控。然后自动构建基于 Kprobes 内核插桩调试技术的内核监控模块来实时监控它们被特定服务进程调用的情况。最后结合 Zabbix 报警响应机制以及(2)中相关分级情况实现对异常内核函数调用情况进行分级分类的拦截与处理,以增加整个防护系统的实用性和灵活性。

(3) 构建了基于内核函数监控的 Linux 防护系统。通过在实际的服务器环境下进行内核函数跟踪采集,包括获取服务器上所运行的服务进程(vsftpd 与 xrdp)对应的内核函数白名单和选取系统中频繁被调用的内核函数名单,确定出当前系统中待监控的内核函数集,并且根据该集合自动生成了内核监控模块。最后配置 Zabbix 来配合与拓展内核监控模块的功能,从而搭建出完整的防护系统。

(4) 对原型系统进行了功能与性能的相关测试。在功能测试方面,根据系统中可能被攻击者控制的服务进程本身所具有的权限,分两类进行区分验证和说明。对于本身具有特权的服务进程 vsftpd 而言,通过向其中植入后门代码并触发执行来模拟 vsftpd 被攻击者以某种手段控制之后所开展的恶意行为,同时根据防护系统的拦截与处理情况来说明本文防护系统的有效性。而对以普通用户身份运行的服务进程 xrdp 而言,则从理论上进行相关有效性阐述和论证。在性能测试方面,对防护系统部署前后的系统性能,包括常见操作时延与系统带宽进行了测试对比,以量化本文方法对系统性能的影响程度。

在实验设计方面,本文以处于真实使用状态的 Dell R730 服务器作为原型构建与测试的环境,其上运行内核版本为 Linux 4.4 的 Ubuntu Server 16.04 LTS 操作系统。防护原型则基于 GCC 5.4.0 编译器,采用 C 语言和 Shell 脚本语言编程实现。

1.4 论文组织结构

论文的组织结构如下:

第一章:绪论。主要介绍了研究背景、国内外相关研究工作现状以及主要研究内容与技术路线。

第二章:研究基础。介绍了与本文研究相关的关键技术及机理,包括 fttrace 内核动态跟踪技术、Kprobes 运行时内核调试技术以及 Zabbix 报警响应机制等。

第三章:内核函数潜在安全风险等级的研究与划分。结合系统调用的分类和全风险分析及内核函数已知漏洞等因素,讨论了内核函数的安全分级标准。

第四章:系统防护原型的设计与实现。结合实际问题,阐述了本文解决问题的切入点,并详细介绍了基于内核函数监控的 Linux 系统防护方法的总体设计与实现方法。最后,通过具体实验,说明了原型系统的构建与实现过程。

第五章:原型测试与结果分析。对防护系统进行了功能测试与性能测试,并对

有关结果进行了分析论证，说明了原型系统及相关方法的有效性和实用性。

第六章：总结与展望。对本文研究工作进行总结，并说明工作的不足之处以及未来需要进一步研究的努力方向。

2 研究基础

本章主要介绍与实现原型系统相关的一些研究技术基础。只有对它们的使用及深层原理进行透彻的理解,才能更好地支持本文防护方法的设计与实现。具体包括为确定 Linux 服务器系统中内核函数的监控范围而选用的 ftrace 动态跟踪技术、为实施核心的监控环节而采用的 Kprobes 内核调试技术,以及为提升防护方法灵活性和可扩展性而结合的 Zabbix 报警响应机制。

2.1 Ftrace 动态跟踪技术

2.1.1 Ftrace 概述

动态跟踪技术试图以一种非破坏性的方式对运行时的程序进行必要信息的跟踪与采集,从而帮助人们定位并排查程序在线上的各类问题。Dtrace^[23]是 Unix 世界著名的动态跟踪技术,也是该领域的的开山之作。它最初诞生于 Sun 公司的 Solaris 操作系统中,目的在于提高工程师解决系统运行时问题的效率。其实现过程得到了 Solaris 每个子系统开发人员的大力支持,跟踪范围囊括了整个内核态与用户态。但也正因为其功能强大齐全,且与内核各部分之间的联系过于紧密,导致 Dtrace 的实现机理格外的复杂。曾经有人试图将 Dtrace 移植到 Linux 中来,但最终因上述种种原因而舍弃了许多高级功能,导致移植结果离生产要求相去甚远。

出于同样的需求与目的, Linux 内核在 2.6.27 之后^[24]也加入了动态跟踪技术,即 ftrace。“ftrace”的本意是“function tracer”,这是因为它的诞生主要是为了跟踪与采集内核运行时,函数的调用与执行情况。相比 Dtrace 而言, ftrace 实现代码量少,原理简单且灵活稳定,从而得到了 Linux 内核开发者的喜爱与支持。此后 ftrace 逐渐提供了各类跟踪功能。例如,查看进程上下文切换的相关信息、系统中断被禁用的时间以及高优先级进程从被唤醒到被系统调度执行期间的最大延迟时间等等。这些功能可以很好的辅助内核开发和研究人员对内核进行调试,并及时发现内核中的各类问题。另外, ftrace 也具有很好的拓展性,它提供了一些简洁易用的接口来允许开发者以插件的形式添加和使用更多种类的跟踪器(tracer),正因如此,其逐渐发展成为了一个内核跟踪框架。目前一些常见的 ftrace 跟踪器及相关功能描述见表 2-1。如未明确说明,表中提到的函数均指内核函数。

表 2-1 常见的 ftrace 跟踪器

Table 2-1 Common tracer of ftrace

跟踪器	相关功能描述
nop	空操作跟踪器，ftrace 的初始状态，不进行任何跟踪。
function	函数跟踪器，用来获取函数调用的相关信息
function_graph	函数调用图跟踪器，可以清晰的展现出函数的调用层次关系以及返回情况
sched_switch	进程上下文切换跟踪器，用来跟踪系统中进程切换情况
irqsoff	中断跟踪器，跟踪禁止中断的函数及相关信息
preemptoff	抢占跟踪器，跟踪禁止内核抢占的函数及相关信息
preemptirqsoff	抢占与中断跟踪器，可以看做 irqsoff 与 preemptoff 二者的组合
wakeup	进程调度延迟跟踪器，记录高优先级进程从被唤醒到被系统调度执行期间的最大延迟时间
wakeup_rt	实时进程调度延迟跟踪器，与 wakeup 跟踪器类似，但跟踪对象为实时进程
blk	阻塞式输入输出跟踪器，跟踪记录 Block I/O 相关信息
mmiotrace	内存映射输入输出跟踪器，跟踪记录 Memory Mapped I/O 相关信息

2.1.2 Ftrace 组成及跟踪原理

Ftrace 主要由负责控制整个跟踪流程的核心框架和提供各类跟踪功能的跟踪器等两部分组成。其组成及典型跟踪流程如图 2-1 所示。

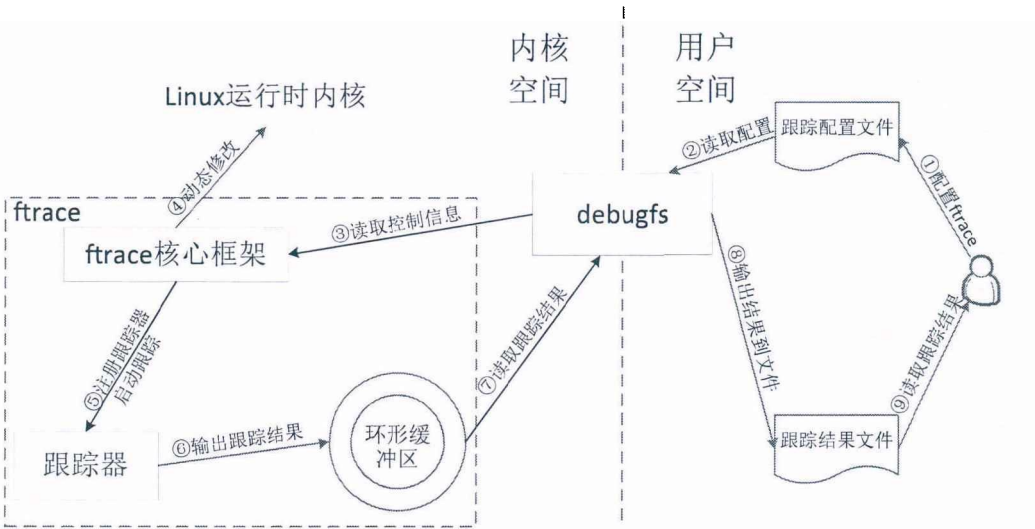


图 2-1 ftrace 组成及跟踪过程

Figure 2-1 Composition of ftrace and corresponding tracing process

由于相关机制存在于内核空间中，所以 ftrace 为了方便用户使用，采用了内核

中的 debugfs^[25]调试技术来支持用户与之进行交互。Debugfs 是 Linux 内核提供的一种类似于 procfs^[26]与 sysfs^[27]的虚拟文件系统，它们均只存在于内存中，用来实现内核空间与用户空间的数据交互。但与 procfs 和 sysfs 不同的是，debugfs 是专门设计来为系统调试提供服务的，它将 ftrace 的相关控制接口以文件形式（如表 2-2 所示）呈现给用户，供用户方便地配置和获取相关信息。

正如图 2-1 中过程所示，当用户修改这些接口文件来配置 ftrace 时，ftrace 核心框架将会借助 debugfs 实时读取相关配置信息，并据此来动态调整运行时内核以支持跟踪机制，然后注册当前指定的跟踪器。当用户开启跟踪后，跟踪器将会把跟踪结果输出到 ftrace 维护的内核环形缓冲区中，而 debugfs 则负责把缓冲区的内容同步输出到跟踪结果文件中，于是用户便可以读取相应文件来查看跟踪结果。

表 2-2 ftrace 提供的常用文件接口

Table 2-2 Common file interface provided by ftrace

文件名	相关功能描述
tracing_on	控制跟踪的开启与暂停
current_tracer	配置或查看当前正在使用的跟踪器
available_tracers	查看系统中支持的传统跟踪器
trace	跟踪结果文件，用来查看跟踪结果
trace_pipe	类似于 trace 文件，所不同的是查看过的记录之后不会再次显示
trace_options	提供一组选项来控制跟踪结果的输出内容及格式
tracing_cpumask	支持以掩码的方式指定要跟踪的 CPU 核心
set_ftrace_pid	指定 pid 以跟踪特定进程
set_ftrace_filter	设置跟踪过滤规则，支持简单形式的通配符表达式
set_ftrace_notrace	与 set_ftrace_filter 的效果相反，用来设置不需要被跟踪的函数等
available_filter_functions	查看所有允许被过滤的函数亦即 ftrace 能跟踪的所有函数
buffer_size_kb	调整或查看用来记录跟踪结果的环形缓冲区大小
buffer_total_size_kb	查看环形缓冲区总大小

而在跟踪原理方面，以函数跟踪器为代表的传统跟踪器^[24]的实现主要利用了 GCC 编译器提供的 mcount 特性（一种静态代码插桩技术）。当程序通过 gcc 命令进行编译且指定 -pg 调试选项时，GCC 会在编译过程中往每个函数的入口处加入对 mcount 函数的调用代码。此后，当程序执行过程发生函数调用时，mcount 函数会被调用执行。mcount 本是 libc 库中的函数，用来记录关于该程序执行及当前被调函数的一些信息，但由于内核的构建并不会与用户态模块进行链接，所以 ftrace 实现了自己的 mcount 函数来记录函数调用情况等相关跟踪数据。

进一步说，在内核编译构建时，如果 `CONFIG_FUNCTION_TRACER` 选项被选中，那么 GCC 便会开启 `-pg` 选项对内核进行编译，从而在每一个内核函数的入口处设置对 `mcount` 函数的调用^[28]。尽管这种方式能够实现大范围的跟踪覆盖，但也会导致大量的函数调用指令被加入内核中，进而会一定程度地降低内核的运行速度。为了降低跟踪对内核性能的影响，`ftrace` 支持了所谓的 `dynamic ftrace` 机制，该功能通过配置 `CONFIG_DYNAMIC_FTRACE` 编译选项来启用。

如果 `dynamic ftrace` 被开启，那么 GCC 在内核编译期间会调用一个名为 `recordmcount.pl` 的 Perl 脚本来分析每一个 C 程序目标文件中的 ELF 头信息以获取所有 `text` 段中调用 `mcount` 的位置信息，并在原始目标文件中以重链接（`re-link`）的方式新添加一个特殊的“`__mcount_loc`”段来存储上述位置。在内核最后的链接阶段，所有这些位置信息会被记录到一张表里。内核启动初期，`ftrace` 核心通过扫描这张表，将其中地址所指向的指令更新为 `nop` 指令从而完成初始化。因此当跟踪未启用时，`ftrace` 几乎不会给系统带来额外的开销。此后在必要时，`ftrace` 会将 `nop` 指令动态替换为 `mcount` 调用^[29]。配置 `set_ftrace_filter` 和 `set_ftrace_notrace` 过滤器就是 `dynamic ftrace` 技术的一种具体体现。以使用函数跟踪器为例，指定要跟踪或者无需跟踪的具体函数，实质上就是通知 `ftrace` 动态地将相关函数中的 `nop` 指令替换为 `mcount` 调用或者将 `mcount` 调用替换为 `nop` 指令的过程。该设计很好地提高了跟踪的性能与灵活性。

2.1.3 Ftrace 配置与使用

表 2-3 与表 2-4 分别列出了支持 `ftrace` 核心和相关跟踪器的内核配置选项。需要指出，本文实验环境使用的 Linux 发行版 Ubuntu 操作系统默认开启对 `ftrace` 的支持，无需重新编译内核。

表 2-3 支持 `ftrace` 基本框架的相关内核配置选项

Table 2-3 Kernel configuration options that support the basic framework of ftrace	
配置选项	说明
<code>CONFIG_DEBUG_FS</code>	启用 <code>debugfs</code> 文件系统
<code>CONFIG_TRACING</code>	启用内核跟踪机制
<code>CONFIG_TRACING_SUPPORT</code>	启用内核跟踪机制相关支持
<code>CONFIG_FTRACE</code>	启用 <code>ftrace</code> 核心框架
<code>CONFIG_DYNAMIC_FTRACE</code>	启用 <code>ftrace</code> 动态跟踪功能
<code>CONFIG_ENABLE_DEFAULT_TRACERS</code>	启用默认跟踪器

表 2-4 支持跟踪器功能的相关内核配置选项

Table 2-4 Kernel configuration options that support functionality of tracer

编译选项	说明
CONFIG_NOP_TRACER	支持 nop 跟踪器
CONFIG_FUNCTION_TRACER	支持 function 跟踪器
CONFIG_FUNCTION_GRAPH_TRACER	支持 function_graph 跟踪器
CONFIG_SCHED_TRACER	支持调度相关跟踪器
CONFIG_CONTEXT_SWITCH_TRACER	支持 sched_switch 跟踪器
.....	其它

在使用 ftrace 前,应提前挂载 debugfs 文件系统^[30]。官方提供的挂载方式如下:

```
# mount -t debugfs nodev /sys/kernel/debug/
```

挂载操作完成后, /sys/kernel/debug/下会出现一个 tracing 目录, 其中内容便是 ftrace 借助 debugfs 提供的所有控制接口, 用户可以使用常规的文件读写操作来进行查看与设置。该目录下常用的文件及相关功能描述已在表 2-2 中列出。

通过配置接口文件 current_tracer, 可设置当前 ftrace 采用的 tracer 以完成不同类型的跟踪需求。另外可以通过如下命令来查看当前系统支持的所有 tracer。相关文件中的内容已在表 2-1 中给出说明。

```
# cat /sys/kernel/debug/tracing/available_tracers
```

以获取系统中内核函数的被调信息为例, 启用函数跟踪器 (function tracer) 涉及到如下的配置命令:

```
# echo function > /sys/kernel/debug/tracing/current_tracer
```

buffer_size_kb 接口文件负责管理单个 CPU 核心所使用的缓冲区大小。默认情况下每个 CPU 的缓冲区大小相同, 而总的跟踪缓冲区大小 (buffer_total_size_kb) 则为 buffer_size_kb 中的数值乘以系统 CPU 核心数。另外, 由于 ftrace 采用的是环形缓冲区 (ring buffer), 所以缓冲区大小对于跟踪效果来说格外的重要。如果缓冲区设置过大, 会导致系统可用内存减少。如果缓冲区设置过小, 那么当跟踪到的信息过多时, 就可能导致记录发生循环覆盖的现象, 从而造成有效数据的丢失, 影响跟踪结果的完整性。需要指出的是, 在修改缓冲区大小前, 需要先将 current_tracer 设置为 nop, 亦即开始跟踪之前就应该确定合适的缓冲区大小。所以在实际的跟踪场景下, 需要根据跟踪的具体需求和类型, 提前调整好合适的环形缓冲区大小, 尽可能避免上述情况发生。具体调整方法类似于设置当前跟踪器, 可以使用 echo 命令将以 kb 为单位的数值写入 buffer_size_kb 来实现。此外, 诸如 set_ftrace_pid 等配置文件的查看与设置方法也较为类似, 故不再赘述。

在完成一系列配置工作之后,修改 `tracing_on` 文件内容为 1 或 0 即可开启或暂停跟踪。示例如下:

```
# echo 1/0 > /sys/kernel/debug/tracing/tracing_on
```

由于 `tracing` 目录下的 `trace` 文件的内容会与环形缓冲区保持同步,所以对该文件进行常规的读操作即可获取跟踪结果。命令读取方式示例如下:

```
# cat /sys/kernel/debug/tracing/trace
```

使用 shell 命令来操作 `ftrace` 是最为常见的使用形式,但需要指出的是 `ftrace` 也提供了一些函数接口(声明位于 `include/linux/kernel.h` 或 `kernel/trace/trace.h` 等文件中),用来支持在内核编程中实现相关功能。其中, `tracing_start()` 用来开启对跟踪的相关支持,此后需要继续设置跟踪器以及开启跟踪。`tracing_set_tracer()` 函数用来指定跟踪器,并对跟踪器进行注册操作。`tracing_on()` 与 `tracing_off()` 的作用则分别等价于用命令向 `tracing_on` 文件输入 1 和 0,用来在代码中灵活控制跟踪的开启与暂停。而 `tracing_stop()` 会将 `ftrace` 重置为初始状态,此前指定的跟踪器将被注销,如需重新开启跟踪,需再次调用 `tracing_start()` 及后续函数。`tracing_update_buffers()` 与 `tracing_resize_ring_buffer()` 分别将环形缓冲区恢复为默认大小和设置为指定大小。而 `trace_printk()` 函数类似于内核中的 `printk()` 函数,所不同的是后者向内核日志缓冲区中输出信息,而前者则向 `ftrace` 的跟踪缓冲区中进行输出。

2.2 Kprobes 内核调试技术

2.2.1 Kprobes 概述

Linux 内核从本质上讲也是程序,开发人员进行开发与维护的过程中,同样需要对其进行调试,以保证功能的正确性与可靠性。对于应用程序而言,人们可以依赖系统层面提供的一些支持来实现诸如控制台输出以及打断点等调试技术,从而查看变量取值、函数调用堆栈和程序执行流程等必要调试信息。但由于内核不同于常规的程序,其体积庞大,并且自身就是构建与运行其它程序的基础平台,也不可能像应用程序那样暂停运行等待用户调试,所以对运行时内核的调试工作需要一些特殊的技术进行辅助。

传统的内核调试采用在源代码中插入 `printk()` 调用语句的方式。尽管其可以在内核的任何位置实现相应调试信息的获取,但由于每一次对调试点的新增或修改,都需要重新编译内核,所以这种方式具有相当的局限性且极其低效。为此, Linux 内核自 2.6.9 版起^[31]便集成了一种所谓 Kprobes (Kernel Probes) 的运行时内核调试技术。这是一种非破坏性的二进制插桩技术,且一般以可加载内核模块方式来动态

地加载或卸载，故而不再需要重新编译内核。除了少数特定的函数以及自身实现代码之外，Kprobes 几乎能在运行时内核的任意位置插入探针（probe）^[32]，并设置由使用者自定义的处理函数，因此极大地方便了人们对内核的开发与研究。

Kprobes 在许多常见的 CPU 架构上都有相应的实现。其目前具体支持的架构以及支持情况如表 2-5 所示。

表 2-5 Kprobes 支持的 CPU 架构
Table 2-5 CPU architectures supported by Kprobes

CPU 架构	说明
i386	完全支持，且支持跳转优化
x86_64 (AMD-64, EM64T)	完全支持，且支持跳转优化
ia64	不支持对 slot1 指令的探测
ppc/ppc64	完全支持
sparc64	尚未实现 kretprobe (return probe)
arm	完全支持
mips	完全支持
s390	完全支持

2.2.2 Kprobes 探测原理

Kprobes 提供了 kprobe、jprobe 与 kretprobe 等三种类型的探针，分别支持对任意位置的探测、对指定函数的入口的探测以及对指定函数返回的探测，其中 jprobe 与 kretprobe 基于 kprobe 实现，它们是对 kprobe 的封装和特殊化^[33]。

(1) kprobe

kprobe 结构体定义于源码文件/linux/kprobe.h 中，是整个 Kprobes 内核调试机制的基础。下面的代码段列出了其中较为关键的结构体成员。

```
struct kprobe {
    kprobe_opcode_t *addr;           /* 待探测地址 */
    const char *symbol_name;         /* 待探测函数名 */
    unsigned int offset;              /* 地址偏移量 */
    kprobe_pre_handler_t pre_handler; /* 指向前置处理函数的指针 */
    kprobe_post_handler_t post_handler; /* 指向后置处理函数的指针 */
    kprobe_fault_handler_t fault_handler; /* 指向错误处理函数的指针 */
    kprobe_opcode_t opcode;          /* 探测点处的原指令 */
    struct arch_specific_insn ainsn; /* 与平台相关的原指令副本 */
    ..... /* 其它 */
};
```

所谓 kprobe 探针其实就是该结构体的一个实例，用户可设置 `symbol_name`、`offset` 以及 `addr` 等字段来指定待探测的内核位置。其中 `symbol_name` 字段用来指定待探测内核函数的符号名。`offset` 字段则用来告知 Kprobes 待探测位置相对于函数入口地址的偏移量。当意图探测相关函数内部的某个位置时，需要计算并设置该字段。但值得注意的是，在 i386, x86_64 等 CISC 类型的架构中，Kprobes 注册探针时并不会检查探测点地址是否超过了指令边界，所以 `offset` 字段的使用需要格外的谨慎。此外，也可以直接设置 `addr` 字段来指定探测位置。事实上，在注册探针的过程中，Kprobes 首先会通过内核符号表查到 `symbol_name` 中存储的符号对应的地址，然后在此基础上加上 `offset` 中设置的偏移量从而得到真正的探测点地址，最后该地址会被填入当前结构体实例的 `addr` 字段中。需要指出的是，如果 `symbol_name` 与 `addr` 均被设值，则注册 kprobe 探针将直接返回错误信息。

`pre_handler`、`post_handler` 和 `fault_handler` 等三个函数指针成员用来指定内核执行探测点处原指令之前需要执行的前置处理函数、内核执行探测点处原指令之后需要执行的后置处理函数，以及探测过程发生错误时需要执行的错误处理函数。它们是 Kprobes 机制预设的回调钩子，也是实现内核运行时调试的关键。遵照如下的特定原型标准来按需定义及实现相关处理函数，并将它们“挂在”对应的钩子上即可完成相关处理函数的设置。

```
/* 前置处理函数的类型定义 */
typedef int (*kprobe_pre_handler_t) (struct kprobe *, struct pt_regs *);
/* 后置处理函数的类型定义 */
typedef void (*kprobe_post_handler_t) (struct kprobe *, struct pt_regs *, unsigned long flags);
/* 错误处理函数的类型定义 */
typedef int (*kprobe_fault_handler_t) (struct kprobe *, struct pt_regs *, int trapnr);
```

当注册一个探针时，Kprobes 会首先备份该探针结构体指定位置 (`addr`) 处的原指令到 `opcode` 字段中，并以一条中断指令（如 i386 架构中的 `int3`）将原指令的第一个字节或几个字节替换掉。此后，一旦 CPU 执行到该指令位置，Kprobes 就能够借助中断以及事先在内核中的异步通知机制 `notifier_call_chain` 中注册的中断处理函数取得 CPU 控制权，并保存当前的寄存器内容等上下文环境，然后回调 `pre_handler` 指针所指向的自定义前置处理函数，再单步执行 `ainsn` 字段中与机器架构相关的原指令副本，并通过 `debug` 中断重获 CPU 控制权，继而调用 `post_handler` 指针指向的后置处理函数，最终让系统回到原本的执行路径上继续执行。相关探测处理过程如图 2-2 所示。

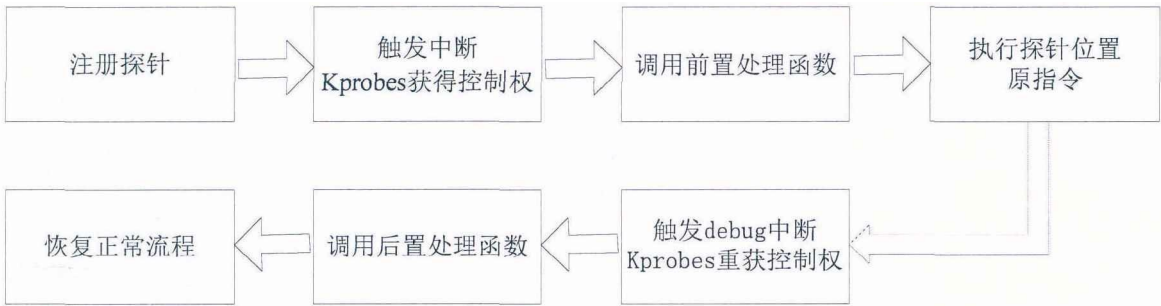


图 2-2 kprobe 探针基本探测原理

Figure 2-2 Basic detection principle of kprobe

(2) jprobe

与 kprobe 不同的是，jprobe 探针仅能在函数入口处进行探测且一个函数只能注册一个 jprobe 实例，它常常用来获取传入函数的参数。其结构体定义如下：

```
struct jprobe {
    struct kprobe kp;
    void *entry; /* 指向入口处需调用的处理函数的指针 */
};
```

从上述定义可以得知，jprobe 探针结构体只是在 kprobe 的基础上进行了简单的封装，并添加了一个 entry 成员，用来指定被探测函数入口点处的处理函数。该处理函数需要与被探测函数具有一样的参数列表。而待探测函数则通过结构体实例中 kp 成员的 symbol_name 字段来指定。以探测 do_fork()为例，entry 所指向的函数原型可定义如下：

```
/* 该函数的参数列表与实际的do_fork()函数参数列表完全一致 */
static long my_do_fork(
    unsigned long clone_flags,
    unsigned long stack_start,
    struct pt_regs *regs,
    unsigned long stack_size,
    int __user *parent_tidptr,
    int __user *child_tidptr);
```

注册 jprobe 探针时，由 Kprobes 机制实现的 setjmp_pre_handler()函数会被设置成 jprobe 内嵌的 kprobe 探针 kp 的 pre_handler。按照 kprobe 探针的触发机制，该函数将在 CPU 运行到探测点时被调用，它负责将此时寄存器和栈中的相关数据进行备份，然后将控制权转交给 entry 指针所指向的处理函数。由于该处理函数与被探测函数具有一样的参数列表，所以用户在其中可以很方便的获取到将要传入被探测函数的参数。此后处理函数在返回前调用 jprobe_return()，还原此前的寄存器与栈的状态并把控制权交回被探测函数，以此保证被探测函数能够正常运行。需要注意，处理函数在返回前必须调用 jprobe_return()，否则内核无法回到原执行流程。

另外，由于函数参数除了通过寄存器传递，也可能通过栈传递，所以 jprobe 在保存现场时，同时备份了寄存器与栈的相关内容。jprobe 探针的基本探测原理见图 2-3。

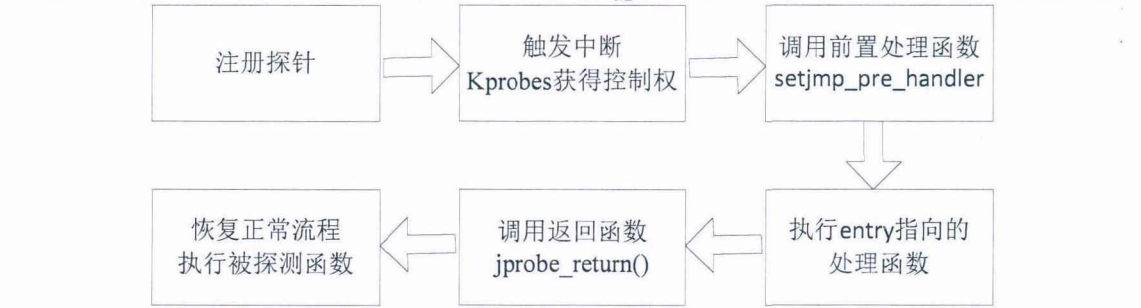


图 2-3 jprobe 探针基本探测原理

Figure 2-3 Basic detection principle of jprobe

(3) kretprobe

kretprobe 一般用于获取指定函数的返回值，其结构体定义如下：

```
struct kretprobe {
    struct kprobe kp;
    kretprobe_handler_t handler;    /* 指向返回后需调用的处理函数的指针 */
    ..... /* 其它 */
};
```

与 jprobe 类似，kretprobe 探针结构体中也包含了一个特殊的 kprobe 结构体 kp。它被置于被探测函数入口处，并且其中包含的 pre_handler 和 post_handler 等回调函数由 Kprobes 机制本身来实现。当被探测函数被调用时，kp 中设置的 pre_handler 被调用执行。其最主要的工作是备份被探测函数的原返回地址，并将其替换为一个所谓“trampoline”区域的地址。当被探测函数正常执行完成并返回时，控制流跳转到“trampoline”区域，并会触发一个由 Kprobes 机制在内核初始化时期置于该区域的 kprobe 探针。其在保存被探测函数返回后的上下文信息（包括返回值）之后，将控制流转到探针结构体成员 handler 所指向的函数。因此，用户可在该自定义处理函数中借助于 regs_return_value() 函数来获取到事先已被保存的返回值。最后 Kprobes 借助于已备份的原返回地址恢复正常的执行流程，其中原理与 kprobe 探针恢复正常执行的流程近似，故不再赘述。

2.2.3 Kprobes 启用及相关接口

虽然 Kprobes 以动态加载内核模块的方式来使用，但其核心机制仍然需要内核提供支持，即编译内核前需要进行适当的配置。用来支持 Kprobes 功能的内核配置选项见表 2-6。需要说明，Ubuntu 发行版默认开启 Kprobes，无需重新编译内核。表 2-7 则列出了由 Kprobes 提供的常用内核编程接口原型及相关功能说明。

表 2-6 支持 Kprobes 的相关内核配置选项

Table 2-6 Kernel configuration options that support Kprobes

配置选项	说明
CONFIG_KPROBES	启用 Kprobes
CONFIG_MODULES	开启模块运行时插入支持
CONFIG_MODULE_UNLOAD	开启模块运行时卸载支持
CONFIG_KALLSYMS	启用内核符号表
CONFIG_KALLSYMS_ALL	支持内核符号表中包含所有符号信息

表 2-7 Kprobes 接口函数原型

Table 2-7 Prototype of interface functions for Kprobes

接口函数原型	功能
int register_kprobe(struct kprobe *kp);	kprobe 探针注册函数原型
int pre_handler(struct kprobe *p, struct pt_regs *regs);	kprobe 探针前置处理函数原型
void post_handler(struct kprobe *p, struct pt_regs *regs, unsigned long flags);	kprobe 探针后置处理函数原型
int fault_handler(struct kprobe *p, struct pt_regs *regs, int trapnr);	kprobe 探针错误处理函数原型
void unregister_kprobe(struct kprobe *kp);	kprobe 探针注销函数原型
int register_jprobe(struct jprobe *jp);	jprobe 探针注册函数原型
void unregister_jprobe(struct jprobe *jp);	jprobe 探针注销函数原型
int register_kretprobe(struct kretprobe *rp);	kretprobe 探针注册函数原型
int kretprobe_handler(struct kretprobe_instance *ri, struct pt_regs *regs);	函数返回后调用的处理函数原型
void unregister_kretprobe(struct kretprobe *rp);	kretprobe 探针注销函数原型

2.3 Zabbix 报警响应机制

Zabbix 是一个开源的企业级分布式监控解决方案^[34]。它能够监控目标机器的各类系统参数，并提供具有高可配置性的告警与响应机制来帮助系统管理员及时的发现问题，解决问题，进而保障被监控系统的正常运行。通过适当的配置，Zabbix 可以通过邮件、微信甚至是短信等媒介第一时间向系统管理员发出告警信息，并且支持预设不同的处理方案来应对系统可能出现的各类异常情况。

一个典型的 Zabbix 配置及告警过程如下：

(1) 首先，配置监控项 (item)。即被监控机器上需要被监控的对象，包括网络带宽使用情况、CPU 负载状态以及指定日志文件内容等等。

(2) 然后，按需设置监控项上发生的特定事件的触发器 (trigger)。即指定一个或者多个监控项共同触发报警需满足的特定条件，譬如被监控系统网络带宽超限、CPU 负载过大、当前登录用户数量发生变化以及日志文件新增记录中出现特定关键字等等。当特定条件满足，trigger 便会被触发，管理页面将出现报警信息。

(3) 最后，可以为某个触发器事先绑定响应动作 (action)。当这个触发器被触发时，其上绑定的 action 会随之执行，从而进行相关处理。一个触发器可以绑定一个或多个响应动作，比如远程执行处理脚本、给管理员发送邮件告警等等。

在 Zabbix 的监控体系中，用户一般可以配置一个监控端 (安装 zabbix-server) 以及一个或多个被监控端 (安装 zabbix-agent)。如果有必要，监控端与被监控端可以是同一台机器，甚至也可以形成多对多的关系。一种基本的 Zabbix 监控系统部署架构如图 2-4 所示。

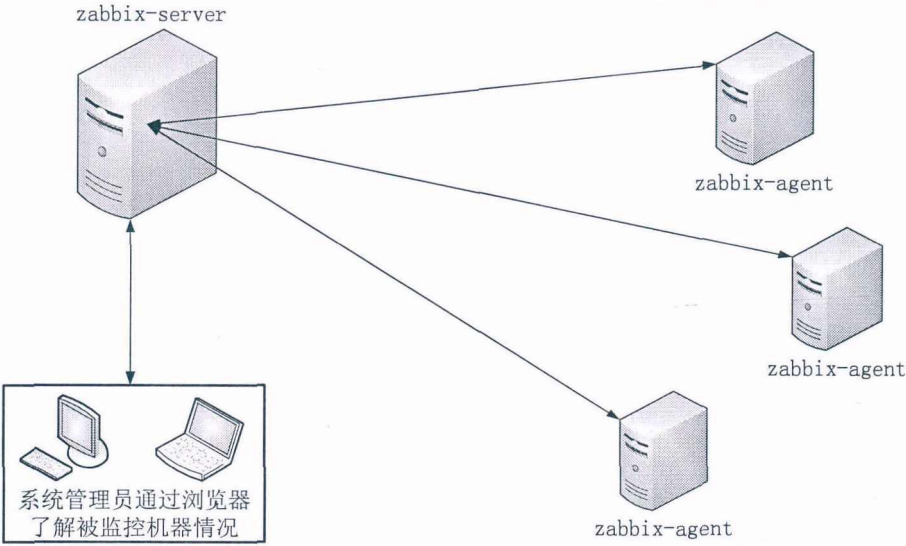


图 2-4 Zabbix 监控系统基本部署架构

Figure 2-4 Basic deployment architecture of Zabbix monitoring system

被监控端中安装的 zabbix-agent 负责采集机器的各种监控项的情况以及执行监控端发送的处理命令。其与监控端之间的通信可配置为主动和被动两种，主动模式下 zabbix-agent 会主动将采集到的信息发送给 zabbix-server，而被动模式则是后者主动进行读取。

监控端则采用了 B/S 模式。zabbix-server 会将来自被监控机器的相关监控数据存储在 MySQL 数据库中进行持久化，而 web 服务器则负责将这些数据以网页的形式展示给用户，同时提供可视化的配置支持，并将用户设置的相关配置数据写入

数据库中。在这种架构下，管理员可以在任何有网络的地方使用浏览器清晰便捷地了解被监控对象各类关键数据的情况，并及时介入对异常情况的处理中。

2.4 本章小结

本章针对原型设计与构建过程中所涉及到的关键技术进行了较为详细的介绍，主要包括 `ftrace` 动态跟踪技术、`Kprobes` 内核调试技术和 `Zabbix` 报警响应机制。这些技术均为本文研究与实现基于内核函数监控的系统防护方法提供了必要的理论与实践的支撑。

3 内核函数潜在安全风险等级的研究与划分

本章试图给出一种内核函数的潜在安全风险等级分级标准并进行相应的划分，从而更好地支持本文原型系统对服务进程异常情况的分级分类处理。但事实上，由于内核函数数量过于庞大，想要直接完成这一目标并不现实。考虑到处于用户空间的攻击者在控制服务进程之后的各种恶意行为总是会直接或间接地通过组织和调用系统调用来实施开展，所以提出以系统调用的潜在安全风险等级为主要参考依据来确定它们背后实际实现功能的相关内核函数对应的潜在安全风险等级的思想，并以此开展了对所有系统调用的安全风险分级工作以及相关内核函数的潜在危险等级分级研究。特别地，对于高安全风险级别来说，则会额外考虑一些存在已知安全漏洞的内核函数以及安全相关的敏感内核函数。

3.1 系统调用分类及潜在安全风险分级

3.1.1 系统调用概述

系统调用（System Call）是内核提供给用户空间的功能交互接口。位于用户空间的程序要实现与系统相关的各类高级功能，就需要直接或者间接使用内核提供的这些功能函数来获得支持。直接使用系统调用的方法是通过一个名为 `syscall` 的特殊系统调用，以系统调用号作为其入参来指定调用特定的函数。而间接的使用方式则是借助 Linux 下的标准 C 库（`glibc`）中对原始系统调用进行封装之后的包装函数。一般来说，包装函数在名称上与原始名称相同，也更方便用户使用。但需要说明，并不是所有的系统调用在 `glibc` 中都有对应的封装函数。

以 `x86_64` 架构的 Linux 4.4 内核为例，其对外提供的系统调用共 326 个，在源码根目录下的 `/arch/x86/entry/syscalls/syscall_64.tbl` 中列出（示例见表 3-1）。相关内核空间入口函数原型定义于源码根目录下的 `/include/linux/syscalls.h` 中。

表 3-1 `x86_64` 架构系统调用表部分内容

Table 3-1 Part of `syscall_64.tbl`

系统调用号	兼容性	系统调用函数名	对应内核入口函数
0	common	read	sys_read
1	common	write	sys_write
2	common	open	sys_open

表 3-1 x86_64 架构系统调用表部分内容（续）

Table 3-1 Part of syscall_64.tbl (continued)

系统调用号	兼容性	系统调用函数名	对应内核入口函数
3	common	close	sys_close
...
19	64	readv	sys_readv
20	64	writew	sys_writew
...
324	common	membarrier	sys_membarrier
325	common	mlock2	sys_mlock2

3.1.2 总体分类与分级标准

徐明等人曾在其异常检测研究^[35]中把 Linux 2.4 版本内核中的 241 个系统调用按照功能和危险程度划分为文件、进程、网络、模块、信号与其它六个类别以及四个危险等级，并主要针对每一类的第一级也就是潜在危险程度最高的系统调用（称之为关键调用）进行异常调用的监控。本文受其分类思想的启发，结合 Linux 内核子系统（虚拟文件系统、进程调度、进程间通信、网络接口以及内存管理）的划分情况^[36-39]，在已有分类标准^[35, 39]上进行了改进与扩充，提出了自己的系统调用分类以及潜在安全风险等级分级原则，并据此针对 x86_64 架构的 Linux 4.4 版本内核中的系统调用进行了分类与分级。

具体做法是，先排除实际未实现的少量系统调用（总是返回-1），并将剩余部分（共 315 个）按其提供的功能划分为文件相关操作、进程相关操作、进程通讯相关操作、网络相关操作、内存相关操作以及其它等六个功能大类。然后以总体的潜在安全风险分级原则为基础，结合每一大类自身的特性，给出具体类别下的分级依据。其中总的分级原则是，一般能够直接帮助攻击者达成提升权限、控制系统或破坏系统稳定性与完整性等目的的高危敏感操作属于第一级，也就是高风险等级。除高风险操作外，那些有可能被用来组织恶意行为或是辅助完成恶意行为的系统调用应归入第二级，即中等风险等级。而第三级别则主要包括一些如信息查看等类型的低风险或无风险操作。在总体分级原则的基础上，不同类别的系统调用的具体的分级标准将在下一小节以示例分析的方式进行详细解释说明。

3.1.3 系统调用安全风险等级分级示例

对于文件相关操作而言，那些可能导致文件内容发生变化或是可能修改如文件权限、属主（组）等敏感属性的系统调用一般来说都具有较高的安全风险。譬如 `open` 系统调用是获取各类文件操作的入参——文件描述符的关键函数，`truncate` 系列函数可以截断文件内容、控制文件的长度，`sync` 系列操作则可将内存中的内容同步到磁盘对应文件中。它们均可能导致敏感文件内容发生变化，所以被视为高风险等级系统调用。此外，`chmod` 与 `chown` 能分别修改文件的访问权限以及属主（组），而对于 `rename` 系列与 `utime` 系列系统调用而言，前者能够移动文件的访问路径并支持重命名操作，这类似于删除原位置处的相应文件，后者则可以改变文件上一次修改的时间戳。二者均可导致文件关键信息的变化，故均属于第一级别。特别指出，`unlink` 系列函数虽然主要用来撤销文件链接，但是当操作目标是某文件的最后一个硬链接时，该文件将会从硬盘上彻底删除，所以其也应属于第一级别。

除文件高风险操作以外，那些有可能被用来组织恶意行为的文件相关系统调用则具有中等风险。以 `create` 系统调用为例，它可以创建一个常规的文件并返回其文件描述符，并不会对已有的敏感文件产生影响，但不排除其有被用来恶意填充磁盘，浪费系统资源的可能。另外，`userfaultfd`、`eventfd` 及 `signalfd` 等函数则用来创建错误处理，事件处理和信号处理专用的特殊文件，并返回文件描述符。它们（包括 `create`）与 `open` 操作的打开现有文件的性质不同，故被定为第二级。创建软连接或硬链接相关操作类似于 `create`，而 `rmdir` 则只能删除空目录，亦属于第二级。但需说明，相较 `utime` 而言，`setxattr` 系列函数操作的仅是文件的非敏感属性，所以并没有被划分到文件类别的第一级中。

另外，`stat` 系列函数获取文件的各类基本信息，`access` 检查用户对指定文件的访问权限，`sysfs` 查看文件系统类型信息。本文将这些获取文件状态、属性等信息、查看文件权限、获取 `io` 优先级、获取当前工作目录以及获取文件系统相关信息信息等基本信息查看类的操作划分为第三级。另外，第三级还包含如开启文件预读取等几乎不存在安全风险的文件相关操作。

需要指出，本文始终站在某函数本不应该被调用，但是却发生了调用行为的角度来评价相关操作的潜在安全风险等级。在这种情况下，每一个系统调用都应该被独立对待。而徐明等人的分类标准则认为，需要文件描述符作为入参的系统调用（如 `write` 和 `read`）在使用前需要先调用 `open` 来打开相应文件。故只将 `open` 划分为关键调用进行监测。但实际上如果使用 `open` 对于目标进程而言本身就是合法的，而使用 `read` 或者 `write` 等系统调用是非法的，那么相关恶意行为就可能不会被检测到。所以本文将 `write` 系列调用也置于第一级，而 `read` 则因仅涉及读取操作而被归入第二级别中。`readlink` 用来获取符号链接路径，应与 `read` 置于同一级别。相关分级结果如表 3-2 所示。

表 3-2 文件类系统调用潜在安全风险等级划分

Table 3-2 Classification of potential security risk for file-related system calls

风险等级	系统调用名称
1	chmod, chown, fchmod, fchmodat, fchown, lchown, fchownat, rename, renameat, renameat2, utime, utimes, utimensat, futimesat, open, openat, open_by_handle_at, write, pwrite64, writev, pwritev, truncate, ftruncate, fallocate, sync_file_range, sync, syncfs, fsync, fdatasync, msync, unlink, unlinkat
2	close, dup, dup2, dup3, flock, ioctl, memfd_create, mknod, mknodat, mkdir, mkdirat, rmdir, creat, symlink, symlinkat, link, linkat, quotactl, ioprio_set, chdir, fchdir, chroot, fcntl, lseek, poll, ppoll, epoll_create, epoll_create1, epoll_ctl, epoll_wait, epoll_pwait, select, pselect, pread64, read, readv, preadv, readlink, readlinkat, sendfile, umask, setxattr, lsetxattr, fsetxattr, removexattr, lremovexattr, fremovexattr, io_setup, io_destroy, io_submit, fadvise64, userfaultfd, eventfd, eventfd2, signalfd, signalfd4, perf_event_open, name_to_handle_at, inotify_init , inotify_init1, inotify_add_watch, inotify_rm_watch, fanotify_init ,fanotify_mark
3	access, faccessat, newfstatat, fstat, fstatfs, getcwd, getdents, getdents64, lstat, readahead, stat, statfs, sysfs, ustat, getxattr, lgetxattr, fgetxattr, listxattr, llistxattr, flistxattr, io_getevents, io_cancel, lookup_dcookie, ioprio_get

在进程相关操作中，那些可能导致可执行代码被加载的函数，可能影响进程正常运行的函数，以及可能修改进程相关权限等敏感信息的函数均属于高风险函数。正如表 3-3 所示，除了如 `execve`、`kill` 以及 `setuid` 系列等危险系统调用外，其它较敏感的系统调用也被置于第一级别中。比如 `uselib` 可以实现恶意动态链接库的加载，而 `ptrace` 则允许进程以较高的权限来访问与控制其子进程甚至其它进程。此外 `ioperm` 系统调用可设置当前进程对端口的访问权限，`iopl` 则用来改变当前进程的 I/O 特权级别。两者均涉及到敏感权限的修改，具有较高的潜在安全风险，并且官方文档中也明确指出不建议用户使用 `iopl`，因为其能够授予进程不受限制的 I/O 端口高特权级别访问权限，这就使进程拥有禁用中断的能力，从而可能导致系统崩溃。

进程类的第二级则主要是一些有可能实现恶意行为但并不没有上述明显敏感操作的进程相关系统调用。`fork` 等调用虽然创建了新的进程，但如果后续未使用 `exec` 族函数为子进程加载新的可执行代码或未使用其它系统调用组织子进程的行为，那么其所执行的代码仍是继承于父进程的后续代码，不会产生直接的安全威胁。不过进程也有可能通过反复 `fork` 子进程来恶意消耗系统资源，但其危害程度相较

第一级函数而言并不算大，故被归为第二级。`seccomp` 的不当使用虽然可能导致进程被系统杀死，但其影响的只是当前调用它的进程，也属于第二级别。

低风险分级中则主要包括用来查看进程权限相关信息、运行与调度相关信息、资源使用情况与限制情况等与进程非敏感信息相关的系统调用。例如 `geteuid` 系列函数用来获取进程权限信息，`getcpu` 用来了解进程当前运行在哪个 CPU 上，`sched_getscheduler` 及相关函数能获取进程调度的相关信息，`sched_rr_get_interval` 获取轮转调度的时间片大小，`getrusage` 与 `getrlimit` 用来得到进程资源使用与限制情况。它们的使用几乎不存在安全风险。进程类系统调用分级结果如表 3-3 所示。

表 3-3 进程类系统调用潜在安全风险等级划分

Table 3-3 Classification of potential security risk for process-related system calls	
风险等级	系统调用名称
1	<code>execve</code> , <code>execveat</code> , <code>uselib</code> , <code>kill</code> , <code>tkill</code> , <code>tgkill</code> , <code>ioperm</code> , <code>iopl</code> , <code>ptrace</code> , <code>setuid</code> , <code>setgid</code> , <code>setpgid</code> , <code>setgroups</code> , <code>setsid</code> , <code>setreuid</code> , <code>setregid</code> , <code>setresgid</code> , <code>setresuid</code> , <code>setfsuid</code> , <code>setfsgid</code>
2	<code>fork</code> , <code>vfork</code> , <code>clone</code> , <code>brk</code> , <code>exit</code> , <code>exit_group</code> , <code>seccomp</code> , <code>pivot_root</code> , <code>sched_setparam</code> , <code>sched_setscheduler</code> , <code>sched_yield</code> , <code>setpriority</code> , <code>setrlimit</code> , <code>prlimit64</code> , <code>vhangup</code> , <code>shmctl</code> , <code>shmat</code> , <code>shmctl</code> , <code>shmdt</code> , <code>semget</code> , <code>semop</code> , <code>semtimedop</code> , <code>semctl</code> , <code>futex</code> , <code>arch_prctl</code> , <code>set_thread_area</code> , <code>modify_ldt</code> , <code>sched_setattr</code> , <code>sched_setaffinity</code> , <code>set_tid_address</code> , <code>process_vm_readv</code> , <code>process_vm_writev</code> , <code>setns</code> , <code>set_robust_list</code> , <code>migrate_pages</code> , <code>move_pages</code> , <code>unshared</code> , <code>capset</code> , <code>personality</code> , <code>prctl</code> , <code>wait4</code> , <code>waitid</code> , <code>nanosleep</code> , <code>clock_nanosleep</code>
3	<code>acct</code> , <code>capget</code> , <code>getegid</code> , <code>geteuid</code> , <code>getgid</code> , <code>getgroups</code> , <code>getpgid</code> , <code>getpgrp</code> , <code>getpid</code> , <code>gettid</code> , <code>getppid</code> , <code>getpriority</code> , <code>getresgid</code> , <code>getresuid</code> , <code>getsid</code> , <code>getuid</code> , <code>getrlimit</code> , <code>getrusage</code> , <code>sched_get_priority_max</code> , <code>sched_get_priority_min</code> , <code>sched_getparam</code> , <code>sched_getscheduler</code> , <code>sched_rr_get_interval</code> , <code>kcmp</code> , <code>sched_getaffinity</code> , <code>get_thread_area</code> , <code>getcpu</code> , <code>sched_getattr</code> , <code>get_robust_list</code>

而在进程通信类系统调用中，可向指定进程发送信号数据的 `rt_sigqueueinfo` 与 `rt_tgsigqueueinfo` 系统调用被归为进程通讯类操作的第一级别。其它的信号操作均针对的是当前调用进程，属于第二级别。这其中就包括检查当前进程待接收信号（`rt_sigpending`），设置当前进程对具体信号的处理方式（`rt_sigaction`），检查与设置当前进程的信号掩码（`rt_sigprocmask`），暂停当前进程等待某信号的到来（`pause`）以及设置 `alarm` 信号定时器（`alarm`）等。此外，消息队列与管道通信等相关系统调用也划分在第二级别中。如表 3-4 所示，该类别无低风险级别函数。

表 3-4 进程通信类系统调用潜在安全风险等级划分

Table 3-4 Classification of potential security risk for system calls about process communication	
风险等级	系统调用名称
1	rt_sigqueueinfo, rt_tsigqueueinfo
2	rt_sigaction, rt_sigpending, rt_sigprocmask, rt_sigreturn, rt_sigsuspend, rt_sigtimedwait, sigaltstack, pause, alarm, msgget, msgsnd, msgrcv, msgctl, mq_open, mq_unlink, mq_timedsend, mq_timedreceive, mq_notify, mq_getsetattr, pipe, pipe2, tee, splice, vmsplice

网络相关高风险系统调用包括新建 socket 连接及相关操作、绑定或监听特定端口等。中等风险系统调用包括关闭连接、接收与发送数据等操作。低风险级别主要为查看 socket 相关信息等无害操作。具体分级见表 3-5。

表 3-5 网络类系统调用潜在安全风险等级划分

Table 3-5 Classification of potential security risk for network-related system calls	
风险等级	系统调用名称
1	accept, accept4, bind, connect, listen, socket, socketpair
2	recvfrom, recvmsg, recvmmsg, sendto, sendmsg, sendmmsg, setsockopt, shutdown, bpf
3	getpeername, getsockname, getsockopt

在内存相关的系统调用中，madvice 用来向内核提供对指定位置内存的使用建议，这有可能导致类似于释放还未使用完的内存映射等可能对其它进程产生影响的现象发生。故将其置于第一级别中。第二级别则主要包括内存映射相关操作、内存锁操作以及一些可能对当前进程造成影响的操作（如设置当前进程的内存策略、内存屏障以及内存页的访问保护等）。低风险级别则主要是一些获取进程信息类型的操作。如获取当前进程或线程的内存策略（get_mempolicy），查看当前进程的虚拟内存页面是否驻留内存中（mcore）。内存相关系统调用的分级如表 3-6 所示。

表 3-6 内存类系统调用潜在安全风险等级划分

Table 3-6 Classification of potential security risk for memory-related system calls	
风险等级	系统调用名称
1	madvise
2	mmap, mmap2, mremap, munmap, mprotect, set_mempolicy, mbind, mlock, mlock2, mlockall, munlock, munlockall, membarrier

表 3-6 内存类系统调用潜在安全风险等级划分（续）

Table 3-6 Classification of potential security risk for memory-related system calls (continued)

风险等级	系统调用名称
3	get_mempolicy, mincore

除了上述分类外，剩下的系统调用中也存在一些高风险操作。譬如获取、新增或管理内核加密机制的 key，加载新的内核模块或卸载已有内核模块，挂载新的文件系统或卸载已有文件系统，修改系统时间，内核时钟等系统关键参数，重启系统调用机制或重启系统与加载新内核，以及操作内核日志缓冲区等存在破坏系统可靠性、可用性的高潜在风险操作。中等风险的操作则包括定时器系列操作、设置 swap 路径及停止 swap 交换机制等可能对恶意行为有辅助作用的操作。最后，查看计时器信息、生成随机数、查看系统时间、日期以及内核基本信息等无安全风险的操作被归为其它功能类别的第三级别。分级情况见表 3-7。

表 3-7 其它系统调用潜在安全风险等级划分

Table 3-7 Classification of potential security risk for other system calls

风险等级	系统调用名称
1	request_key, add_key, keyctl, init_module, finit_module, delete_module, mount, umount2, settimeofday, clock_settime, adjtimex, clock_adjtime, _sysctl, restart_syscall, reboot, kexec_load, kexec_file_load, syslog
2	setitimer, timer_create, timer_delete, timer_settime, timerfd_create, timerfd_settime, setdomainname, sethostname, swapon, swapoff
3	getitimer, timer_gettime, timer_getoverrun, timerfd_gettime, getrandom, gettimeofday, time, times, sysinfo, clock_getres, clock_gettime, uname

3.2 内核函数潜在安全风险分级

对应于系统调用的潜在安全风险等级，本文同样将内核函数划分为高、中、低三个潜在安全风险级别。其中，实现高风险系统调用功能的某些内核函数、当前内核版本中存在已知漏洞的内核函数以及一些安全敏感内核函数属于高潜在安全风险等级。低风险系统调用的实现所涉及的内核函数或一些查询信息类内核函数属于低潜在安全风险等级。而剩余函数则可以参照并结合系统调用的分级标准来进行风险等级的判定，一般默认归为中等潜在安全风险等级。另外需要特别说明的是，

本节有关内容均基于 Linux 4.4 版本内核源码进行阐述。

3.2.1 高风险系统调用对应的内核函数

具体而言，实现高风险系统调用的内核函数判定为高潜在安全风险内核函数主要依据的原则是，其在相关系统调用功能的实现过程中担当关键作用。另外，鉴于系统调用在内核中的实现往往存在代码路径重叠的现象，所以风险等级的划定也应遵循“就高不就低”处理原则。所谓担当关键作用，主要指相关内核函数是某系统调用功能在内核中的启动入口或者实现了该调用的实质性功能。而“就高不就低”则指的是如果一个内核函数既在某个较高安全风险系统调用的实现中担当关键作用，同时又被另一较低安全风险级别的系统调用使用，则其仍会被认为是属于较高安全风险等级的内核函数。这里选取几个较典型的高风险等级系统调用进行内核源码静态分析，来具体说明高风险系统调用对应的高风险内核函数在本文的分类分级体系下的判断与选取方法。

(1) 内核模块相关系统调用

内核模块相关的三个系统调用（`init_module`、`finit_module` 和 `delete_module`）的定义位于 `/kernel/module.c` 中。其中 `init_module` 的定义如下所示：

```
SYSCALL_DEFINE3(init_module, void __user *, umod,
                unsigned long, len, const char __user *, uargs)
{
    int err;
    struct load_info info = { };

    err = may_init_module();
    if (err)
        return err;

    pr_debug("init_module: umod=%p, len=%lu, uargs=%p\n",
            umod, len, uargs);

    err = copy_module_from_user(umod, len, &info);
    if (err)
        return err;

    return load_module(&info, uargs, 0);
}
```

这是一个通过 `SYSCALL_DEFINE` 宏来定义的函数，在内核编译时，编译器会进行一系列预处理，最终生成 `sys_init_module()` 函数的原型。该函数是 `init_module` 系统调用在内核中的入口，当用户态进程使用 `init_module` 系统调用并借助 `int 0x80` 指令切换到内核态^[40]时，系统会直接通过系统调用表找到并调用 `sys_init_module()` 从而开始进行模块加载工作。宏替换后的函数原型如下所示。其中包含三个入参：

`umod` 传入编译好的模块目标文件 (*.ko) 在用户空间的内存地址, `len` 用来指明该文件的长度, `uargs` 则是指向用户空间中模块传入参数的指针。

```
asmlinkage long sys_init_module(void __user *umod, unsigned long len, const char __user *uargs);
```

在 `sys_init_module()` 的函数体中, `may_init_module()` 主要负责检查当前进程是否具有加载内核模块的权限 (`CAP_SYS_MODULE`) 以及当前内核是否允许加载模块, 而 `copy_module_from_user()` 函数则负责检查待加载模块的安全性并将其拷贝到内核空间, 最后读取该文件中的一些 ELF 信息临时保存到 `struct load_info` 类型的结构体对象 `info` 中。当准备工作完成后, 最后由 `load_module()` 函数完成 `init_module()` 系统调用的实际功能, 即把编译好的模块目标文件链接到运行时内核中, 并回调用户编写的模块初始化函数。`load_module()` 的部分代码如下:

```
/* Allocate and load the module */
static int load_module(struct load_info *info, const char __user *uargs,
                      int flags)
{
    struct module *mod;
    long err;
    char *after_dashes;
    .....
    err = elf_header_check(info);
    if (err)
        goto free_copy;

    /* Figure out module layout, and allocate all the memory. */
    mod = layout_and_allocate(info, flags);
    if (IS_ERR(mod)) {
        err = PTR_ERR(mod);
        goto free_copy;
    }

    /* Reserve our place in the list. */
    err = add_unformed_module(mod);
    if (err)
        goto free_module;

    .....
    err = check_module_license_and_versions(mod);
    if (err)
        goto free_unload;

    .....
    /* Link in to syfs. */
    err = mod_sysfs_setup(mod, info, mod->kp, mod->num_kp);
    if (err < 0)
        goto bug_cleanup;

    .....
    return do_init_module(mod);
    .....
}
```

其中, `elf_header_check()` 函数检查载入模块的二进制格式是否合法、架构是否匹配当前系统以及 ELF 版本是否正常。检查通过后 `layout_and_allocate()` 函数将根

据模块 ko 文件的 ELF 布局，为其中所有需要被加载的段（section）计算虚拟地址以及分配地址空间。这是模块链接到运行时内核的前提。然后 `layout_and_allocate()` 会调用 `move_module()` 函数将临时对象 `info` 中暂存的信息转移到真正代表模块的 `struct module` 类型结构体实例中并将其指针 `mod` 返回。该实例包含当前加载模块的所有必要信息。如模块名称、初始化函数指针和卸载函数指针等等。紧接着，`add_unformed_module()` 函数通过系列操作将前一函数返回的模块结构体实例加入内核相关数据结构中。其中包括调用向指定链表中挂载新节点的工具函数 `list_add_rcu()` 将模块结构体挂载到内核模块链表中，以及调用 `mod_tree_insert()` 函数将模块结构体插入内核模块红黑树中。此后 `check_module_license_and_versions()` 检查与校验模块的许可证和版本信息，`mod_sysfs_setup()` 在 `sysfs` 文件系统中注册该模块，而位于最后的 `do_init_module()` 函数的实质工作则是借助 `do_one_initcall()` 函数来回调由用户自定义的模块初始化函数。至此模块加载的主要工作全部完成。

另一个用来加载模块的系统调用 `finit_module` 实质上相当于是对 `init_module` 的重载，其允许用户直接将目标模块文件对应的文件描述符作为入参而不需要预先处理成内存中的二进制模块对象，以方便用户使用。相关定义如下：

```
SYSCALL_DEFINE3(finit_module, int, fd, const char __user *, uargs, int, flags)
{
    int err;
    struct load_info info = { };

    err = may_init_module();
    if (err)
        return err;

    pr_debug("finit_module: fd=%d, uargs=%p, flags=%i\n", fd, uargs, flags);

    if (flags & ~(MODULE_INIT_IGNORE_MODVERSIONS
                  |MODULE_INIT_IGNORE_VERMAGIC))
        return -EINVAL;

    err = copy_module_from_fd(fd, &info);
    if (err)
        return err;

    return load_module(&info, uargs, flags);
}
```

可以看到，函数执行的步骤与 `init_module` 基本一致。所不同的是，将模块数据拷贝到内核空间由 `copy_module_from_user()` 的“重载函数”`copy_module_from_fd()` 来实现。实际的模块加载工作则同样由 `load_module()` 函数来完成。`sys_init_module()` 以及 `sys_finit_module()` 的内核函数调用关系图如图 3-1 所示。

由以上静态分析可知，`init_module` 和 `finit_module` 系统调用在内核的实现过程中共涉及七个较为关键的函数。`sys_init_module()` 以及 `sys_finit_module()` 是它们在

内核中的启动入口。load_module()是实现模块加载的实质性函数，而其中调用的layout_and_allocate()、add_unformed_module()以及do_init_module()则是较为重要的功能环节。另外do_init_module()中调用的do_one_initcall()用来启动模块中的初始化函数，也属于担当关键作用的内核函数。

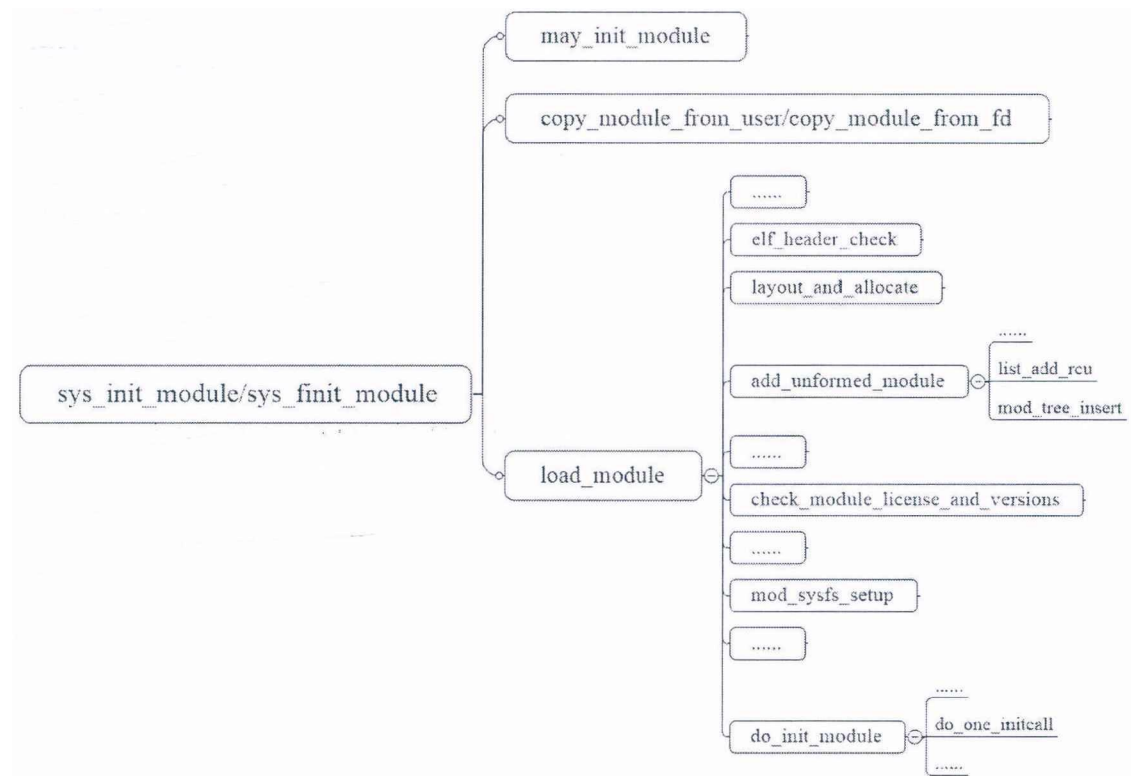


图 3-1 模块加载系统调用对应的内核函数调用关系图

Figure 3-1 Corresponding kernel function call graph of the module loading system calls

与上述系统调用功能相反，delete_module 用来卸载已加载模块，其主要执行流程以 sys_delete_module()函数为入口，先进行卸载权限检查、将模块名参数拷贝到内核空间等准备工作，之后由 find_module()根据模块名找到相应的 module 结构体实例，并判断模块是否处于可被卸载的状态、是否定义相关卸载函数等。然后调用 try_stop_module()函数停止模块运行并进行卸载。最后完成一些如释放模块结构体等善后工作。从上述实现过程可以看出，在模块卸载系统调用中担当关键作用的内核函数包括 sys_delete_module()、find_module()和 try_stop_module()。

(2) 文件相关系统调用

open 系统调用在内核中的入口函数 sys_open()的函数原型如下：

```
asmlinkage long sys_open(const char __user *filename, int flags, umode_t mode);
```

同样地，该函数的定义以 SYSCALL_DEFINE 宏的形式存在于内核源码中 (/fs/open.c)，其会在内核编译时进行展开。相关定义如下：

```

SYSCALL_DEFINE3(open, const char __user *, filename, int, flags, umode_t, mode)
{
    if (force_o_largefile())
        flags |= O_LARGEFILE;

    return do_sys_open(AT_FDCWD, filename, flags, mode);
}

```

可以看出, `open` 系统调用的实质性工作是由 `do_sys_open()` 内核函数来完成的。相关定义如下。在其函数体中, `build_open_flags()` 函数先进行一系列打开标志的检查和填充, 以及读写权限的检查等前置操作, 然后 `getname()` 函数将指定文件名从用户空间取到内核空间中。期间若不出错, `get_unused_fd_flags()` 函数会查询当前进程的已打开文件表并在其中将第一个尚未被使用的文件描述符分配给即将打开的文件。此后 `do_filp_open()` 将通过 `path_openat()` 调用 `get_empty_filp()` 函数来生成相关文件对应的 `struct file` 类型的结构体实例并返回指向该实例的指针, 从而基本完成文件打开过程。

```

long do_sys_open(int dfd, const char __user *filename, int flags, umode_t mode)
{
    struct open_flags op;
    int fd = build_open_flags(flags, mode, &op);
    struct filename *tmp;

    if (fd)
        return fd;

    tmp = getname(filename);
    if (IS_ERR(tmp))
        return PTR_ERR(tmp);

    fd = get_unused_fd_flags(flags);
    if (fd >= 0) {
        struct file *f = do_filp_open(dfd, tmp, &op);
        .....
        return fd;
    }
}

```

`file` 结构体实例保存着已打开文件的各种必要信息, 能够唯一代表一个已打开的文件, 是内核组织与管理系统中所以被打开文件的操作对象, 也是内核中其它各类文件操作的基本入参。所以生成 `file` 结构体实例是文件打开操作的核心, 与之直接相关的功能函数都承担着关键作用, 应归为高潜在安全风险内核函数。包括 `sys_open()`、`do_sys_open()`、`get_unused_fd_flags()`、`do_filp_open()`、`path_openat()` 以及 `get_empty_filp()`。

特别说明, `create` 系统调用属于系统调用分级中的中等风险级别, 与之相关的内核函数本应当也属于中等风险。但事实上其功能是直接通过调用 `sys_open()` 函数来实现的, 所以根据就高不就低原则, `create` 的实现中涉及的关键内核函数的风险等级仍应属于高风险函数。其内核入口函数 `sys_creat()` 的定义如下:

```
SYSCALL_DEFINE2(creat, const char __user *, pathname, umode_t, mode)
{
    return sys_open(pathname, O_CREAT | O_WRONLY | O_TRUNC, mode);
}
```

与之类似的例子还有 read 与 write。它们首先从 sys_***() (***)对应代表 read 或 write) 函数进入内核，然后在其函数体中通过 fdget_pos()与 file_pos_read()来获取用户空间传入的文件描述符所对应的 file 对象和相应文件的读写位置。随后进入 vfs_***()函数进行实际的读写操作。其中首先会进行文件读写权限检查等准备工作，然后调用实质性的功能函数 __vfs_***()。其借助传入的 file 对象所包含的 file_operations 结构体^[18]实例中已设置好的相应文件操作函数指针来回调与具体文件系统相关的底层读写函数，进而完成对磁盘文件的读写操作。由上述分析可知，在 read 与 write 调用的实现过程中，较为关键的内核函数包括 sys_***()、fdget_pos()与其实质功能函数 __fdget_pos()、以及 vfs_***()与其实质功能函数 __vfs_***()。其中 fdget_pos()虽然同时被中等风险等级系统调用 read 所使用，但由就高不就低的原则，该内核函数仍被认为具有高潜在安全风险。read 和 write 系统调用对应的内核函数执行路径如图 3-2 所示。为避免混淆，前者的执行路径用虚线表示。

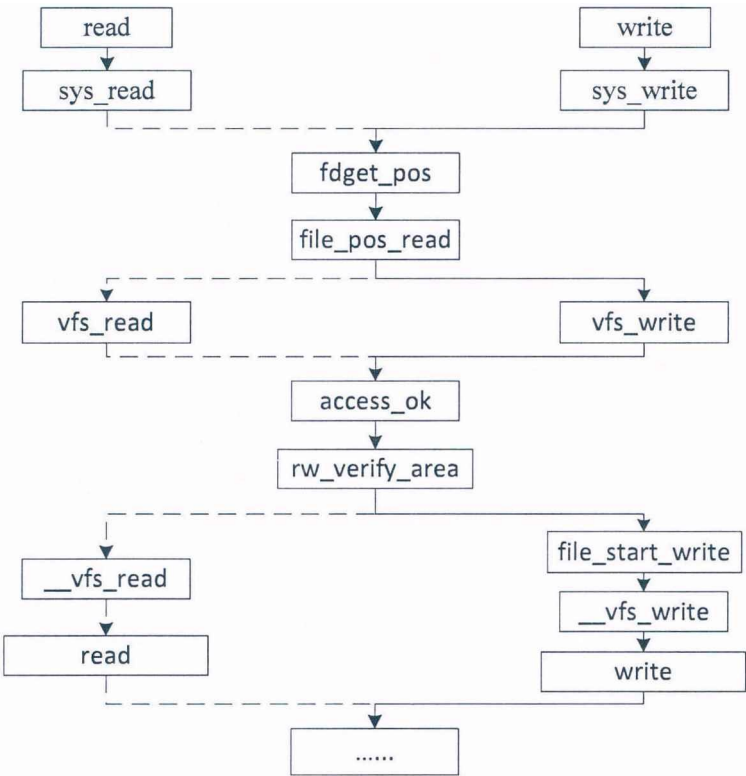


图 3-2 read 与 write 系统调用对应内核函数的执行路径

Figure 3-2 Execution path of the kernel function about read and write system call

由于篇幅等原因，在此深入分析每一个具有高潜在安全风险等级的系统调用

所涉及到的所有内核函数以及其中的具体实现逻辑并不现实。所以除前述示例外，本文仅另外列举出部分高风险等级系统调用对应的高风险内核函数。（见表 3-8）相关分析过程不再详述。

表 3-8 部分高风险等级系统调用对应的高风险内核函数

Table 3-8 High-risk kernel functions corresponding to some high-risk level system calls	
系统调用	担当关键作用的内核函数
chmod	sys_chmod, sys_fchmodat, chmod_common
execve	sys_execve, do_execve, do_execveat_common, do_open_execat, bprm_mm_init, exec_binprm, search_binary_handler, load_elf_binary, start_thread
kill	sys_kill, kill_something_info, kill_pid_info, __kill_pgrp_info, group_send_sig_info, do_send_sig_info, send_signal, __send_signal
setuid	sys_setuid, set_user, commit_creds
socket	sys_socket, sock_create, __sock_create, sock_alloc, inet_create, tcp_v4_init_sock, sock_map_fd, sock_alloc_file, alloc_file, get_empty_filp
bind	sys_bind, sockfd_lookup_light, sock_from_file
listen	sys_listen, sockfd_lookup_light, sock_from_file
accept	sys_accept, sys_accept4, sockfd_lookup_light, sock_from_file, sock_alloc, get_unused_fd_flags
settimeofday	sys_settimeofday, do_sys_settimeofday, do_settimeofday
syslog	sys_syslog, do_syslog, syslog_print_all

3.2.2 存在已知漏洞的内核函数

除高潜在安全风险系统调用所对应的担当关键作用的内核函数外，高潜在安全风险等级中还应当包括一些存在已知漏洞的内核函数。这些内核函数可以通过归纳整理由 CVE 曝光的影响当前版本内核的安全漏洞的方式来获取。进一步说，此类内核函数由于涉及公开的漏洞，容易被攻击者知晓与利用，所以同样存在较高的安全隐患。其中部分函数的漏洞甚至能帮助攻击者直接取得系统的超级管理员权限。一旦它们因系统没有及时地安装相应的安全补丁而被攻击者成功利用，系统安全就极易受到侵害。Linux 4.4 版本涉及的部分漏洞及相关说明示例见附录 A。

3.2.3 安全敏感内核函数

另外，一些涉及身份权限、安全机制等安全攸关的内核函数，如前人相关研究

工作中已分析过的安全敏感内核函数^[41]（如表 3-9 所示）也应该被划分到高潜在安全风险等级中。

表 3-9 内核敏感函数^[41]

Table 3-9 Sensitive kernel functions ^[41]

内核敏感函数	功能
commit_creds	为当前进程装载新的权限集
prepare_kernel_cred	为内核提供权限集
update_mmap_min_addr	允许内核低地址映射
selinux_disable	禁用 SELinux
security_init	重置安全模块
reset_security_ops	重置安全操作指针

3.2.4 低安全风险内核函数

进程类系统调用中的 `getpid` 是一个常见且典型的信息获取类操作，其对应的内核的入口函数为 `sys_getpid`。该函数中会调用 `task_tgid_vnr()` 读取 `current` 指针所指向的代表当前进程的结构体实例（`struct task_struct` 类型）中所存储的 `pid` 字段。获取过程涉及一系列信息获取类函数，如 `task_tgid()`、`pid_vnr()`、`pid_nr_ns()` 以及 `task_active_pid_ns()` 等。由于这些函数只涉及到读取数据而没有修改数据，所以应属于低风险等级内核函数。其它相关系统调用诸如 `getuid`，`getgid` 以及 `gettid` 等的实现均与 `getpid` 系统调用类似。`getpid` 在内核中的函数调用关系图如图 3-3 所示。

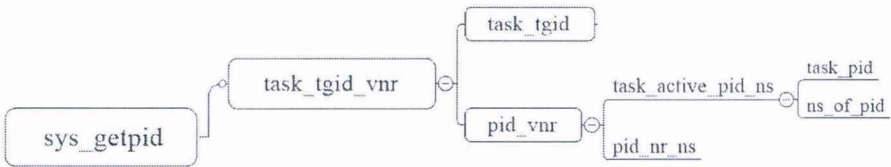


图 3-3 getpid 系统调用的内核函数调用关系

Figure 3-3 Invocation relationship of kernel functions for getpid system call

为了进一步说明低安全风险内核函数的选取方法，以实现稍微复杂的 `get_mempolicy` 系统调用为例。其内核函数调用关系如图 3-4 所示。其中实现功能的主要函数为 `get_mempolicy`、`sys_get_mempolicy`、`do_get_mempolicy`、`find_vma_intersection`、`lookup_node` 以及 `get_policy_nodemask`。这些函数同样也都基本以获取信息为主，存在较低或基本不存在安全风险，除了这些函数之外，函数调用关系图中涉及的其它函数也均会依照 `get_mempolicy` 系统调用的风险等级优

先归为低潜在安全风险内核函数级别中。当然，如果这些函数同时也在高、中等风险系统调用的实现中担当关键作用，那么就应当依照“就高不就低”原则，将它们归到相应的内核函数风险级别中。

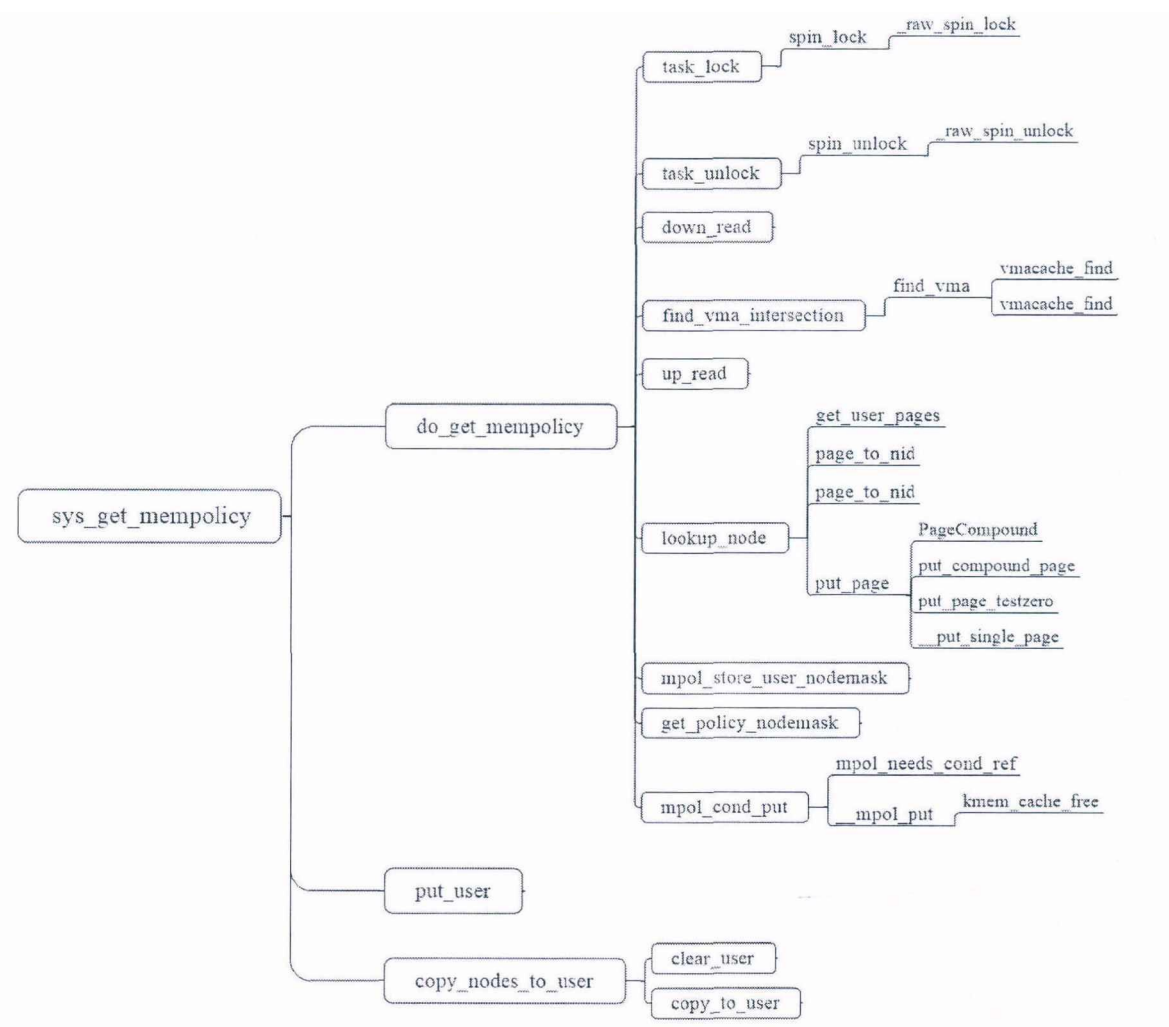


图 3-4 get_mempolicy 系统调用的内核函数调用关系

Figure 3-4 Invocation relationship of kernel functions for get_mempolicy system call

一些其它的例子比如 `gettimeofday`、`time`、`uname` 以及 `mincore` 等，其相关主线函数 `sys_gettimeofday`、`do_gettimeofday`、`getnstimeofday64`、`__getnstimeofday64`、`sys_time`、`get_seconds`、`sys_newuname`、`utsname`、`sys_mincore`、`do_mincore` 以及相关辅助函数也同样属于低潜在安全风险级别。

最后需要指出的是，系统调用的函数调用关系覆盖不到的那些仅供内核在内部使用的函数如不符合高风险内核函数相关标准，应优先置于中等风险级别中。

3.3 本章小结

Linux 内核代码规模庞大，故而无法直接对其中所有函数进行详细分析和归类整理。但考虑到系统调用的特殊地位，所以本章选择以系统调用的功能分类与风险分级为切入点，以特定内核版本为例先对其提供的所有系统调用进行了功能分类与潜在安全风险分级，然后以此为主要参照标准给出了内核函数潜在风险等级的分级依据并进行了相应划分工作。其中已确定安全风险等级的内核函数（Linux 4.4 内核）包括，高安全风险等级 186 个（高安全风险系统调用对应的关键内核函数 72 个，存在已知漏洞内核函数 108 个，安全敏感内核函数 6 个）以及低安全风险等级内核函数 21 个。需要特别说明的是，由于时间与精力等原因，本章只是以示例分析的方式具体说明了内核函数的风险等级确定方法，而并不是对所有的内核函数都进行了相关等级划定。那些未明确划分等级的内核函数在原型系统的异常情况分级分类处理中均将被视为中等潜在安全风险等级函数。也就是说，分级的完善与否，可能导致相关异常发生时采取的处理方式存在部分差异，但这并不会影响到原型系统对异常内核函数调用情况的监测效果。

4 系统防护原型的设计与实现

本章将阐述基于内核函数监控的 Linux 系统防护方法的具体设计及实现过程，是论文的核心部分。相关系统防护方法总体由三部分组成，即待监控内核函数的获取、对待监控内核函数的实时监控以及对异常调用情况的分级分类处理。所以本章首先将依次介绍上述三部分的设计细节和实现方法，然后据此在具体的应用环境下进行本文原型系统的构建与部署工作。

4.1 总体设计

服务进程（如 vsftpd、xrdp 以及 mysqld 等）作为服务器对外暴露的接口，往往是服务器所有安全保障环节中至关重要的一环。按照常见的控制流劫持攻击方式^[42]，对服务器的一般入侵过程应当是，攻击者先扫描目标机器上所有打开的端口以及运行在这些端口上的服务进程版本，同时查看对应版本的服务进程是否存在已知可利用的漏洞，并尝试劫持目标服务进程的控制流，然后利用被控制进程本身已具有的权限进一步渗透系统，最终达成各类非法目的。

由于系统中的服务进程可能以不同的权限运行，所以它们被劫持后危害系统的方式往往也会有所不同。对于以普通用户身份运行的服务进程（如 xrdp、mysqld 等）而言，攻击者可能会以它们的本地权限为跳板，设法构造特定的触发条件来利用某些内核漏洞（如 CVE 中已曝光的大量漏洞）完成恶意行为。而对于以超级管理员身份运行的服务进程（如 vsftpd、sshd 等）来说，攻击者则可以直接利用其本身已有的特权对系统实施各类危害操作。

但不论采取哪种方式，针对系统的恶意攻击最终总会通过执行内核函数来加以实施，并且处于内核级别的监控往往较难被恶意程序绕过^[43]。所以可以说监控内核函数的非法调用情况是增强内核安全和操作系统安全的重要环节。具体来说，服务进程本身的功能特殊性决定了它们正常运行所需要的内核函数往往十分有限，然而它们却通常拥有触发所有内核函数的能力，故而会为攻击者破坏服务器系统的机密性、完整性和可用性提供更多的可乘之机。进一步说，被攻击者非法利用的内核函数所体现的是攻击者自身的非法意图，它们也并不是相应服务进程正常运行所必需的。为此，设法合理有效地限制服务进程所能访问的内核函数范围便成为构建服务器系统防护机制的一种理想的可选方案。

类似于异常检测系统的原理，即事先给出特定服务进程在正常运行情况下所需的内核函数列表，然后把相应进程对其它内核函数的访问全部视为异常行为，且

发生异常行为将会触发告警、拦截等处理动作。基于此思想，本文设计了一种基于 Kprobes 内核调试技术与 Zabbix 报警响应机制的内核函数实时监控防护模型。前者主要用来探测服务进程对内核函数的异常调用情况，并当即处理其中危险程度高的异常情况，后者则配合与拓展前者功能，以支持更为灵活的告警与处理方式。监控模块以内核可加载模块的形式即插即用运行于内核态。而确定模块所监控的内核函数则借助 ftrace 动态跟踪技术来完成。整个系统防护模型如图 4-1 所示。

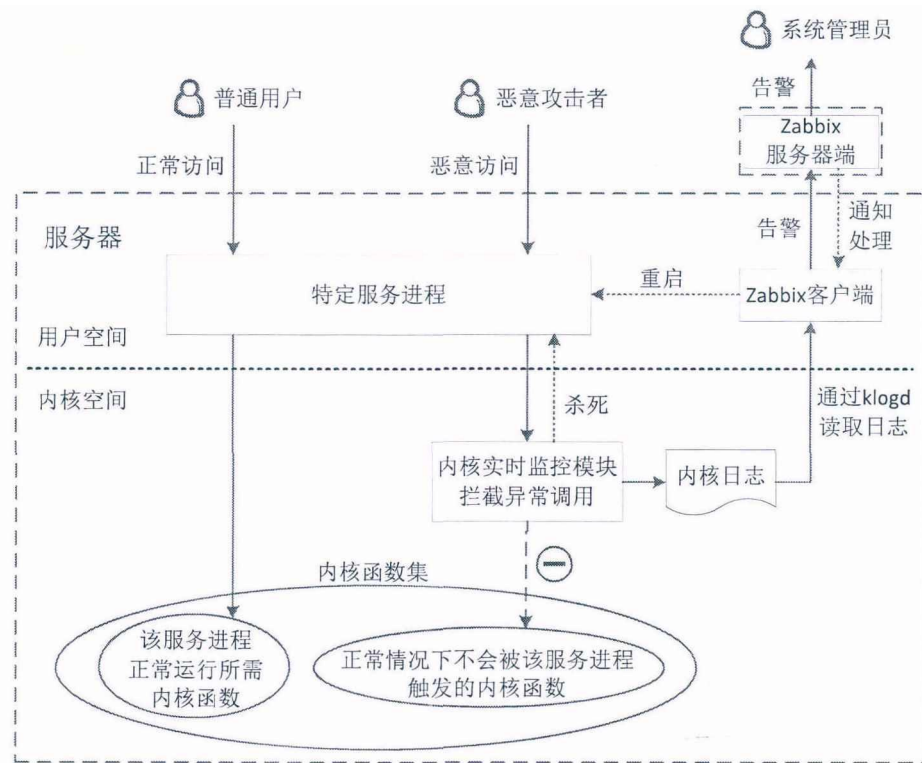


图 4-1 系统防护模型

Figure 4-1 Model for system protection

为实现上述系统防护模型，首先需要跟踪服务器系统中指定服务进程在正常运行期间对内核函数的调用情况，以获取服务进程本身功能真正所需的内核函数的列表，称其为服务进程的内核函数白名单。对于白名单中的这些内核函数，通常可以不进行监控。其次，考虑到在实际应用场景中，采用插桩机制对内核函数进行监控可能带来的系统开销，所以还需要监测跟踪和获取服务器系统日常运行期间频繁调用的内核函数的列表，对于这些函数一般也不应进行监控，以避免导致监控模块对系统性能产生严重的影响。于是，从内核函数总集中剔除白名单内核函数和被频繁调用的内核函数，便构成对应服务进程应当监控的内核函数即待监控内核函数的集合。鉴于待监控集合中的内核函数均非对应服务进程在正常情况下需要使用的内核函数，故而此类内核函数的调用属于异常行为即异常情况。接下来，就

可以构建用于系统监控的内核可加载模块和启动对所有待监控内核函数的持续的实时监控。当异常情况发生时，内核监控模块根据异常所涉及的相关函数所属的潜在安全风险等级采取一系列应对措施，比如在内核日志中记录相应异常调用信息、立即杀死发生异常调用的服务进程、通知系统管理员并重启对应服务进程等。

整个实验设计、原型构建和实施过程简要归纳说明如下：

- (1) 利用 `ftrace` 动态跟踪技术，获取指定进程的内核函数白名单。
- (2) 利用 `ftrace` 动态跟踪技术，获取被系统频繁调用的内核函数的列表。
- (3) 计算确定针对特定服务进程的待监控内核函数列表。

(4) 基于 `Kprobes` 内核调试机制，构建监控模块和启动实时监控，并在异常调用情况发生时，联合 `Zabbix` 监控系统根据不同情况进行相应处理。

4.2 待监控内核函数集获取方法

待监控内核函数集的获取是防护系统开展实时监控的基础。其将通过从所有内核函数集中剔除掉服务进程白名单内核函数以及系统中频繁被调内核函数得到。

4.2.1 内核函数白名单获取方法

内核函数白名单获取过程如图 4-2 所示。首先，主控模块获取待跟踪进程的进程标识符 (`pid`) 并配置 `ftrace` 以开启对相应进程所触发的内核函数进行跟踪。接下来，主控模块将反复读取 `ftrace` 的跟踪结果，并调用简化处理模块对相应结果进行处理，进而累积性地得到被跟踪进程的内核函数白名单。在整个过程中，只需要在一开始手工启动主控模块，之后再无需人工介入，从而实现了白名单的全自动获取。图中， T_{\min} 表示为获取内核函数白名单而设定的下限时间即最短收集时间。另外，主控模块采用 C 语言实现，而其余模块为 `shell` 脚本程序。

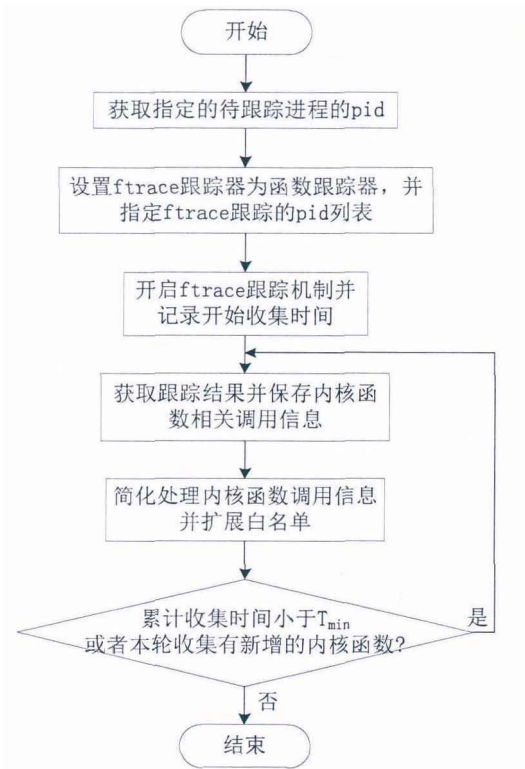


图 4-2 指定服务进程的内核函数白名单的获取流程

Figure 4-2 Process of obtaining a white list of kernel functions for a particular daemon

4.2.2 频繁被调内核函数集获取方法

尽管监控期间采用插桩技术在内核中所加入的代码本身开销极小，但由于需要进行监控的内核函数数量巨大，所以为了减少内核监控模块可能给系统带来的额外性能开销，故而需要放弃对某些内核函数的监控，特别是那些在短时间内可能被调用成百上千次的函数，对它们的监控往往会造成较大的性能开销。相关函数构成了频繁被调内核函数集。

频繁被调内核函数的采集工作同样借助 ftrace 来完成。与白名单获取过程不同的是，其并不指定 ftrace 跟踪的进程标识符，也即根据整个系统运行所涉的全部内核函数被调用的情况来进行采集。鉴于这一过程会产生大量的函数调用数据，而保存跟踪结果的环形缓冲区很快就会被填满，因此如果不及时地取出跟踪结果，便很可能会因记录覆盖而导致数据丢失。所以为了使数据的采集尽可能准确、完整，这一阶段应适当增加环形缓冲区的大小，同时根据系统具体的负载情况选择一个固定的时间片来周期性地获取有关内核函数及对应的调用频次信息。待整个跟踪过程完成后，再根据实际跟踪结果，确定一个调用次数的阈值，进而利用 shell 脚本文程序处理结果文本，从中抽取出频繁被调的内核函数。需要指出的是，只要某个内

核函数在某一时间片内的调用频次大于选定的阈值，它将被记为频繁被调用的内核函数。

4.3 基于 Kprobes 的内核函数实时监控及自动构建

内核函数实时监控是系统防护方法的关键环节，其需要在监控对象即待监控内核函数集确定之后进行。具体做法是借助 Kprobes 提供的相关功能接口，使用 C 语言编写内核可加载模块，为所有待监控内核函数注册 kprobe 探针，展开对它们的实时调用监控。根据本文第二章所介绍的 kprobe 探针实现机制，一旦某个被监控的内核函数被指定的服务进程所触发，相关探针中预设的异常情况处理函数 pre_handler 就会被调用执行，进而实施相关拦截与处理。

鉴于需要监控的内核函数较多，所以监控模块的代码编写工作量非常庞大。不过，考虑到相关内核函数的监控设计彼此间具有较强的相似性和规律性，故而选择以自动生成的方式来构建监控模块。为此，本文在 kprobe 探针的基本模块结构的基础上，设计了如下内核函数实时监控模块的主体代码框架。并以该框架为参考模板，设计和编写了相应的 Shell 脚本程序，用来自动生成监控模块源码文件。

```
/* 为每个待监控内核函数定义一个kprobe结构体 */
static struct kprobe ion_ioctl_kp = {.symbol_name = "ion_ioctl",};
static struct kprobe aa_audit_kp = {.symbol_name = "aa_audit",};
.....

/* 基于潜在安全风险等级划分的kprobe结构体指针数组 */
static struct kprobe* highRiskFunc_kps[] = {&ion_ioctl_kp, ...};
static struct kprobe* mediumRiskFunc_kps[] = {&aa_audit_kp, ...};
static struct kprobe* lowRiskFunc_kps[] = {...};

/* 不同潜在安全风险等级对应的异常处理函数 */
static int pre_handler_emerg(struct kprobe *p, struct pt_regs *regs){...}
static int pre_handler_warning (struct kprobe *p, struct pt_regs *regs){...}
static int pre_handler_notice(struct kprobe *p, struct pt_regs *regs){...}

/* 模块初始化函数 */
static int __init kprobe_init(void){...}

/* 模块注销函数 */
static void __exit kprobe_exit(void){...}
```

上述实时监控框架由探针定义、潜在安全风险等级分级数组、异常情况处理函数以及模块初始化与卸载函数等四部分组成。具体而言，对于每一个待监控的内核

函数，都需要模块为其定义一个对应的 `kprobe` 结构体实例，并通过其中的 `symbol_name` 成员以函数名方式来指定相关内核函数。除此之外，监控框架中还设立了代表不同潜在安全风险级别的三个 `kprobe` 结构体指针数组，用来统一组织对应风险等级的待监控内核函数的 `kprobe` 结构体。此后设置的三个与危险等级相对应的前置处理函数则分别用来实现对相应风险等级的所有待监控函数的异常调用情况的处理。最后部分的模块初始化函数在模块被加载时，将会根据待监控函数的潜在风险级别即其 `kprobe` 结构体实例所关联的数组，为其绑定相应的前置处理函数，并完成所有 `kprobe` 结构体的注册工作。而模块注销函数则主要负责在监控模块被卸载时注销所有的 `kprobe` 结构体。

代码自动生成脚本将以上述结构为基础，分段进行代码生成工作。相关生成过程如图 4-3 所示。

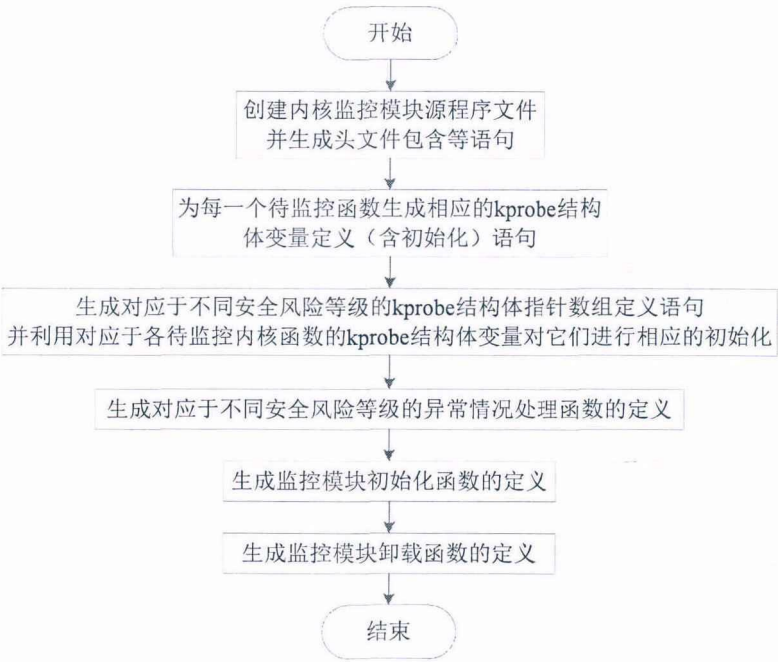


图 4-3 关于内核监控模块源码的自动生成

Figure 4-3 Automatic generation of source codes for kernel monitoring module

其首先创建模块源码文件，添加必要的头文件包含等前置代码。紧接着读取待监控内核函数集，以此逐条生成所有的 `kprobe` 结构体实例定义（变量名统一为 `***_kp`，`***`代表函数名），并将对应函数名填入 `symbol_name` 字段中。然后再参考提前设置好的潜在安全风险等级分级信息，将当前所有待监控函数拆分成高、中、低三个风险等级集合，并填入相应的等级数组中。此后脚本在模块源码中追加高、中、低三个风险等级的处理函数代码，以及模块初始化和卸载函数（部分代码示例如下所示）。其中，初始化函数将会使用循环遍历数组的方式分别为三个 `kprobe` 结

构体数组中的所有元素绑定相应等级的处理函数，并完成所有结构体的注册工作。而卸载函数则以同样的方式分别注销三个数组中所有的 `kprobe` 结构体。最后将模块版本声明等后置代码追加到源码文件中。至此模块生成工作全部完成。以上内核监控模块构建的全过程只需要提前准备好三个风险等级的函数参考列表即可全部自动完成（包括编译内核模块以及将其加载到内核）。此后如有必要，按需调整处理函数中的具体代码即可。

```
for(i=0; i<highRiskFunc_num; i++)
{
    // 为数组中的每一个kprobe结构体绑定与数组级别相同的处理函数
    highRiskFunc_kps[i]->pre_handler = pre_handler_emerg;
    ret = register_kprobe(highRiskFunc_kps[i]); // 注册kprobe探针，开始探测
    if (ret < 0) {
        printk(KERN_INFO "register_kprobe of %s_kp failed, returned %d\n",
                highRiskFunc_kps[i]->symbol_name, ret);
    }
    else {
        printk(KERN_INFO "Planted %s_kp at %p\n",
                highRiskFunc_kps[i]->symbol_name, highRiskFunc_kps[i]->addr);
    }
}
```

最后特别指出，虽然 `ftrace` 可以跟踪某些指定内核函数的调用情况，但却无法为每一个被跟踪函数绑定特定的触发条件^[44]，也不能在某个函数被调用时执行预设的处理代码，所以本文使用能够满足上述要求的 `Kprobes` 内核运行时调试技术来实现对特定内核函数集中函数的实时监控。

4.4 内核函数异常调用情况分类处理

根据服务进程所触发的被监控内核函数所属的潜在安全风险等级，本文防护原型结合内核监控模块和 `Zabbix` 机制提供了相应针对性的处理措施。内核监控模块负责生成关于内核函数异常调用的系统日志记录，并对存在较高可利用风险的内核函数所触发的异常情况，直接采取果断杀死等处理措施。同时，`Zabbix` 机制则主要负责辅助处理相关异常调用险情，并主动向有关管理人员报警。

4.4.1 基于内核监控模块的异常处理

在本文的异常处理机制下，由高潜在安全风险内核函数所触发的异常情况将

直接在内核监控模块中进行处理。进一步说，如果相关服务进程因调用到这些内核函数而产生异常情况，就应当在内核监控模块中借助高危险等级处理函数 `pre_handler_emerg()`，直接杀死触发异常调用的服务进程，同时产生带有紧急标签 [Emergency] 的内核日志记录，并借此触发 Zabbix 服务器端向系统管理员发送带有相关异常信息的告警邮件。

对于待监控函数集中具有中等潜在安全风险的内核函数，尽管它们本身并不存在已知安全漏洞也没有较明显的安全隐患，但仍存在被恶意利用的可能性。如果相关异常情况是由这些函数引起的，内核监控模块应利用对应的中等风险等级处理函数 `pre_handler_warning()`，立即暂停或杀死对应服务进程的执行进而阻止恶意行为的继续，同时产生带有警告标签 [Warning] 的内核日志记录，并触发 Zabbix 服务器端发送邮件通知系统管理员，同时回调被监控端上预设的处理脚本，恢复或重启相应服务进程。

最后，待监控函数集中低风险等级的内核函数与处理函数 `pre_handler_notice()` 相对应，它仅产生带有通知标签 [Notification] 的异常调用日志记录，并以此触发 Zabbix 发送邮件向系统管理员告警。

上述处理过程中提到的杀死或暂停进程是通过向相关进程发送信号来实现的。由于在 Linux 的所有信号中，只有 SIGKILL 与 SIGSTOP 不能被目标程序以自定义处理方式捕获到，故而也无法被绕过和忽略掉。所以监控模块中的处理函数主要使用这两个信号来停止异常进程的继续执行。相关 Linux 信号及简要对比见表 4-1。

表 4-1 部分可停止进程的 Linux 信号

Table 4-1 Some Linux signals that can stop the process		
编号	信号名称	相关解释
2	SIGINT	来自键盘的中断信号，通常在终端中由 Ctrl+C 组合键向进程发出
3	SIGQUIT	来自键盘的退出信号，通常在终端中由 Ctrl+\组合键向进程发出。进程退出后会生成 Core Dump 文件
5	SIGTRAP	暂停进程，一般由断点指令或其它 trap 指令产生，主要用于调试程序
9	SIGKILL	强制杀死进程，不可捕捉与忽略
15	SIGTERM	结束进程，与 SIGKILL 不同的是，该信号可以被捕捉处理或忽略，通常用来要求进程自己正常退出，是 Kill 命令的默认信号
18	SIGCONT	继续执行一个已停止的进程，往往与 SIGSTOP 或 SIGTSTP 配对使用
19	SIGSTOP	暂停进程，不可捕捉与忽略
20	SIGTSTP	暂停进程，与 SIGSTOP 不同的是，该信号可以被捕捉处理或忽略，通常在终端中由 Ctrl+Z 组合键向进程发出

需要说明，为了方便 Zabbix 解析异常调用日志信息并改善系统的可扩展性，

内核监控模块采用 Json 数据格式生成报警日志记录，其中包含异常情况所涉函数名、进程名、进程标识符以及调用参数与调用时间等内容，相关示例如下：

```
[Emergency]{"funcName":"test_arg","processInfo":{"pid":4623,"comm":"insmod"},"registerInfo":{"rdi":6,"rsi":5,"rdx":4,"rcx":3,"r8":2,"r9":1},"timestamp":"1493640848.141706","BeijingTime":"2017-05-01 20:14:08"}
```

另外，为了增强对异常情况的拦截力度，本文防护方法为部分高风险等级内核函数所触发的异常情况设计了另一种处理手段。即适当修改恶意操作代码路径中相关函数的入参或返回值，使相关操作因途中某函数返回代表错误的返回值从而提前结束。

以加载内核模块为例，如果一个正常情况下不会存在模块加载操作的服务进程使用 `finit_module` 系统调用请求将指定的模块文件加载到内核。那么当其执行到内核入口函数 `sys_finit_module()` 时，异常情况会被监控模块监测到。但此时通过发送信号杀死进程可能无法及时阻止即将执行的 `load_module` 函数完成实际的模块加载工作。考虑到后续代码执行路径中某些函数只有模块加载相关操作才会涉及到，即它们往往会与 `sys_finit_module()` 函数同时出现在上述服务进程对应的待监控名单中。所以监控模块在杀死异常进程的同时，可人为地让 `sys_finit_module()` 函数在完成模块加载的中间过程中的某个必经环节（模块加载相关函数）返回错误来达到相关拦截目的。譬如选择 `init_module` 与 `finit_module` 系统调用执行过程中的某一公共函数如在 `copy_module_from_user()` 与 `copy_module_from_fd()` 中都会调用的安全检查函数 `security_kernel_module_from_file()` 作为另一个拦截点，借助 `kretprobe` 探针让该函数返回一个非零值（如 -1，代表 Operation not permitted），结束内核对目标模块的加载过程。当然，除了修改返回值外，修改函数入参也能达到同样的效果。比如借助 `jprobe` 探针将传入 `copy_module_from_fd()` 函数，代表目标模块的文件描述符 `fd` 修改成一个无效描述符，致使该模块文件不存在于已打开文件列表中，模块加载过程则会因该函数返回相应错误而停止。当然，也可以使用监控框架中已有的 `kprobe` 探针的前置处理函数 `pre_handler()` 直接通过修改寄存器的值来实现对入参的修改，只不过此方式没有借助 `jprobe` 探针的实现方式直观。

4.4.2 基于 Zabbix 的异常告警与处理

在本文的原型系统设计中，系统管理员使用一台独立的笔记本作为 Zabbix 监控端来监控部署着内核函数监控模块的服务器（被监控端）。监控端与被监控端可连接在同一局域网中。相关架构设计如图 4-4 所示。

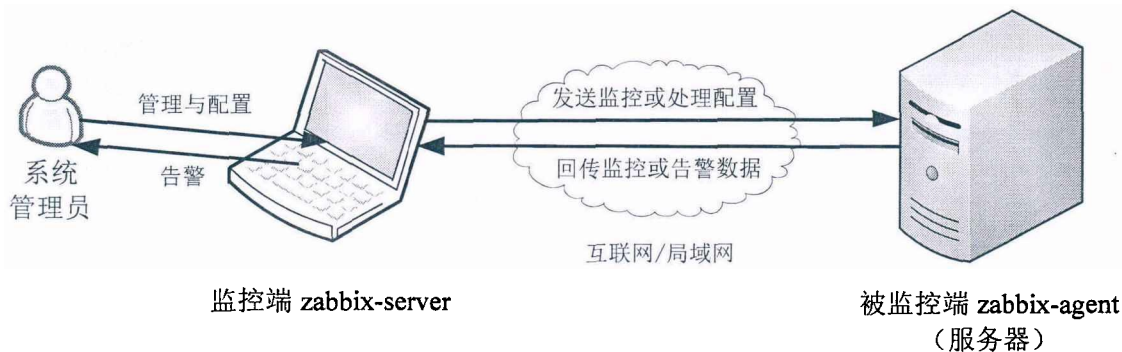


图 4-4 原型系统中的 Zabbix 监控架构

Figure 4-4 Zabbix monitoring architecture in the prototype system

为了实现相关异常告警和辅助处理功能，需要对 Zabbix 进行一系列配置。首先，在监控端与被监控端的有关配置文件中通过指定 ip 地址使二者产生联系。然后，在 Zabbix 的管理页面中将服务器的内核日志文件 (/var/log/kern.log) 设置为监控项并指定监控内容为包含 Emergency、Warning 或 Notification 字符串的新增记录。相关关键配置设置如下：

Item Key: log[/var/log/kern.log, (Emergency|Warning|Notification), , , skip,]

此后，设置 High、Warning 和 Information 等三个不同告警等级的触发器，分别与新增日志记录中出现[Emergency]、[Warning]和[Notification]告警标签相对应。以 Emergency 触发器为例，其关键配置如下：

Trigger Expresion: {kernelFuncMonitor:log[/var/log/kern.log, (Emergency | Warning | Notification), , , skip,].str([Emergency])}=1

最后，为触发器绑定相应的响应动作（包括发送告警邮件 sendMail 和执行处理脚本 execShell）。其中，sendMail 与所有触发器绑定，负责在告警产生后发送邮件通知系统管理员。execShell 只与 Warning 等级的触发器进行绑定，负责在告警产生后远程执行被监控服务器上预设的处理脚本。

这样，当内核监控模块产生新的异常调用日志记录时，Zabbix 便会根据对应记录的等级标签，触发对应的预设告警，并进行相应的处理。

使用内核模块和 Zabbix 相结合的方式异常告警与处理有如下优势和好处：

(1) 增加灵活性。由于 Zabbix 的管理端以网页的形式展示监控数据和支持监控配置，所以相较内核可加载模块而言，通过 Zabbix 查看监控数据以及修改处理配置更为方便，管理员可随时查看被监控端状态并根据具体情况更改对有关异常情况的处理方式。此外，Zabbix 支持多种告警方式，且更容易实现向管理员的主动告警，使人工介入更加及时。

(2) 提高可扩展性。借助 Zabbix 便于将来对异常调用日志的统计与分析的拓

展,而且 Zabbix 不仅可用于监控一台机器,还可应用于对一组机器的分布式监控。

4.5 原型构建

为了完成原型系统的构建与部署,本文在课题组实际投入使用的真实服务器环境中进行了从获取待监控内核函数列表,到构建内核实时监控模块等必要环节。相关服务器系统配置有两颗 Intel(R) Xeon(R) E5-2620v3 2.4GHz 处理器以及 32GB 内存,Ubuntu Server 16.04 LTS x86_64 操作系统,内核版本为 Linux 4.4.0,且运行的服务进程包括 ftp 服务进程 vsftpd,以及远程桌面服务进程 xrdp。防护系统的完整构建过程如图 4-5 所示。其中,Zabbix 的配置过程本节不再赘述。

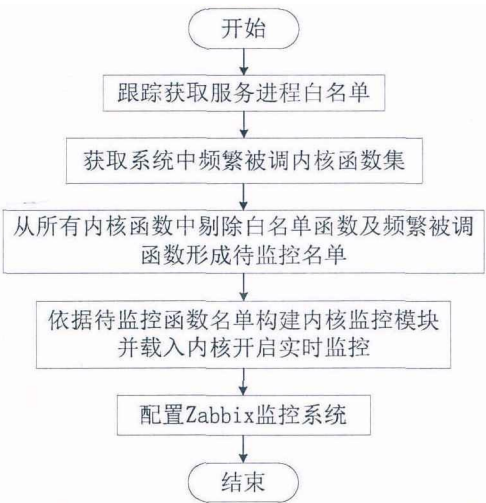


图 4-5 原型构建过程

Figure 4-5 Process for prototype construction

4.5.1 针对特定服务进程的内核函数白名单的获取

在服务器正常使用期间,利用 ftrace 机制,针对其上运行的两个服务进程 vsftpd 与 xrdp,开展了为期四个月的动态跟踪,并分别记录保存了它们调用的所有内核函数的相关数据。关于 vsftpd 的原始跟踪结果片段见表 4-2 所示,其中标志部分用来描述中断是否被禁止、进程状态、软硬件终端类型以及是否禁止抢占。

表 4-2 关于 vsftpd 的 ftrace 原始跟踪结果片段

Table 4-2 Original tracing results of ftrace for vsftpd

进程名-PID	CPU 编号	标志	时间戳	当前被调内核函数及主调函数
vsftpd-2096	[004]	d...	8127.453190	finish_task_switch <- __schedule
vsftpd-2096	[004]	8127.453191	lock_sock_nested <-inet_csk_accept

表 4-2 关于 vsftpd 的 ftrace 原始跟踪结果片段（续）

Table 4-2 Original tracing results of ftrace for vsftpd (continued)

进程名-PID	CPU 编号	标志	时间戳	当前被调内核函数及主调函数
vsftpd-2096	[004]	8127.453191	_cond_resched <-lock_sock_nested
vsftpd-2096	[004]	8127.453191	_raw_spin_lock_bh <-lock_sock_nested
vsftpd-2096	[004]	..s.	8127.453192	__local_bh_enable_ip <-lock_sock_nested
vsftpd-2096	[004]	8127.453192	finish_wait <-inet_csk_accept
vsftpd-2096	[004]	8127.453192	_raw_spin_lock_bh <-inet_csk_accept
vsftpd-2096	[004]	..s.	8127.453192	_raw_spin_unlock_bh <-inet_csk_accept
vsftpd-2096	[004]	..s.	8127.453193	__local_bh_enable_ip <-_raw_spin_unlock_bh
vsftpd-2096	[004]	8127.453193	release_sock <-inet_csk_accept

在此基础上，借助 Shell 脚本对原始跟踪结果数据的统计分析，最终提取出服务进程 vsftpd 与 xrdp 的白名单内核函数分别为 2224 个和 2165 个（本文所有函数名单统一以每行一个函数名的组织形式使用文本文档进行存储）。值得一提的是，跟踪期间发现白名单函数数量在刚开始的一段时间内迅速增加，但随着时间的推移，增速急剧下降，且到了跟踪中后期，白名单内容基本趋于稳定，不再增加新的函数。这一现象间接说明，服务进程在正常被使用的情况下所触发的内核函数是局限于一定范围内的，超出这一范围的函数调用很可能是异常情况。

4.5.2 频繁被调内核函数集的跟踪选取

关于频繁被调函数的选取，需要根据服务器内存大小、系统负载以及关于内核函数调用的监测数据量，适当调整环形缓冲区大小并选取一定大小的时间（譬如 15 秒）为跟踪时间片，从而尽可能避免相关调用记录的丢失。经过两个半月的持续监测和统计分析，得到关于系统运行的四十三万多份内核函数调用跟踪结果，具体示例如表 4-3 所示。

表 4-3 某时间片内的跟踪统计结果片段

Table 4-3 Part of tracing statistics in a time slice

内核函数名	被调次数
_cond_resched	15138
__schedule	7494
enqueue_entity	7136

表 4-3 某时间片内的跟踪统计结果片段（续）

Table 4-3 Part of tracing statistics in a time slice (continued)

内核函数名	被调次数
handle_mm_fault	6847
dequeue_entity	6806
mutex_lock	3861
.....
vsf_setpos	1

从表中可以看到，内核函数__schedule()在对应的时间片内被调用了 7494 次。事实上，该函数在每一个跟踪时间片中被调用的频次都非常大。其原因在于，__schedule()主要负责进程的调度，而服务器上的处理器调度和任务切换非常频繁，故而总是在较短的时间内被大量地调用。类似的特殊函数还有许多，显然，这些函数并不适合被插桩监控。

为最终确定频繁被调的内核函数，调用频次阈值的确立是关键所在，因为其将直接影响到有关防护系统的额外性能开销（与待监控函数数量正相关）及关于异常调用的漏报率（与频繁被调函数数量正相关）。故而，在实际构建防护系统时需在权衡两者关系的基础上选择一个合适的阈值，使得监控开销不至于过大且本应该监控但实际被剔除的内核函数不至于过多。为此，本文针对每份跟踪结果，从 2000 次开始，以 50 次为刻度单位，试探性选取调用频次阈值，并使用 shell 脚本来统计调用次数大于当前阈值的内核函数数量，相关统计分析结果示例如图 4-6 所示。

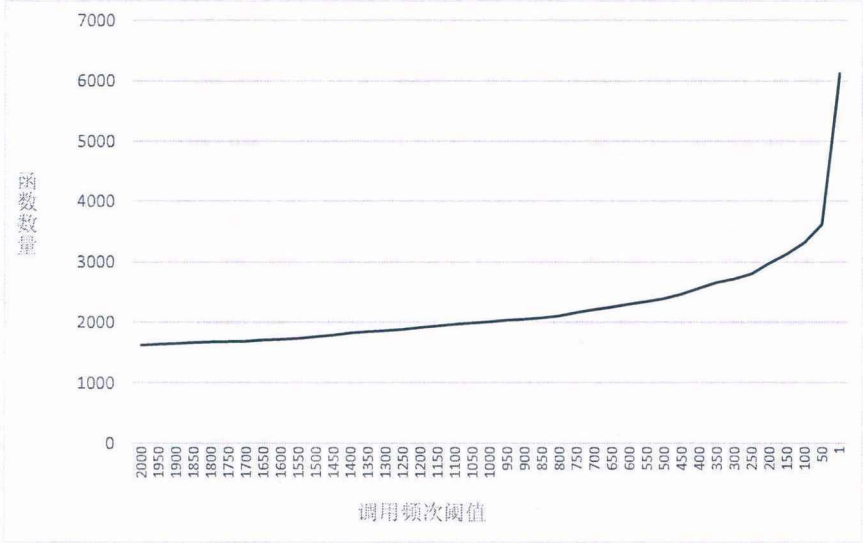


图 4-6 调用频次阈值与函数数量间关系

Figure 4-6 Relationship between the calling frequency threshold and the number of functions

从图中可以看到, 内核函数数量随调用频次阈值的减少而呈上升趋势, 且 50 次左右是所有调用频次阈值的一个拐点。进一步说, 当阈值选取在 50 次以下时, 函数数量随着阈值的下降而剧烈增长, 最终演变为在频繁被调函数整个监测跟踪期间, 所有被调用过的内核函数的总数 (6123 个)。换句话说, 50 次以下的阈值的小幅下调就能导致从待监控内核函数中剔除的函数数量的大幅增加, 从而导致异常调用漏报率的急剧上升。而如果选取 50 以上的次数为阈值, 函数数量随阈值增加而增长的速度较为缓慢, 故而漏报率的下降幅度并不大, 但在这种情况下, 实验证明系统的整体性能会剧烈下降, 其原因在于系统监控防护所带来的额外开销会迅速上升。因此, 本文选取 50 次作为阈值来依次分析每份跟踪结果以构建频调函数集, 最终汇总和统计分析获得 3618 个频繁被调函数。

4.5.3 内核函数监控集的计算确立

在确定服务进程 `vsftpd` 与 `xrdp` 各自的内核函数白名单, 以及整个系统的频繁被调函数名单之后, 便可进一步计算得到需要监控的内核函数列表。具体采用编写 `shell` 脚本进行文本数据处理的方式, 从内核函数全集 (共计 39003 个函数) 中剔除特定服务进程白名单函数以及频繁被调函数, 进而确定出与 `vsftpd` 对应的 35321 个待监控函数以及与 `xrdp` 对应的 35296 个待监控函数。

需要强调的是, 本文在具体确立待监控函数集时, 并没有采用编译器插件或静态分析内核源码等方式来获取内核中的所有函数, 而是直接借用 `ftrace` 在内核初始化时所记录的内核函数列表, 从而避免了需要重新编译内核等问题。

4.5.4 监控模块的自动构建

在确定特定服务进程对应的待监控函数集之后, 需要参照第三章中确定的内核函数潜在安全风险级别的分级情况, 使用 `Shell` 脚本 (主要借助 `grep` 命令的文件模板功能) 将所有待监控函数拆分成高、中、低三个潜在安全风险待监控内核函数列表。它们将作为模块自动生成脚本的入参来完成监控模块的自动构建与加载。

另外考虑到在具体构建原型的服务器环境中, 需要同时监控 `vsftpd` 与 `xrdp` 两个服务进程, 而它们的待监控内核函数集又不完全相同, 所以本文生成内核监控模块代码前, 对前述框架进行了扩展, 建立了二者公用的和各自独立的等三种情况的 `kprobe` 结构体指针数组及相应的异常处理函数, 并分别标以后缀 “`_common`”、“`_vsftpd`” 与 “`_xrdp`”。以中等潜在安全风险等级相关部分为例, 扩展内容如下。相关函数列表的拆分同样使用 `Shell` 脚本 (主要借助 `comm` 命令) 实现。

```
/* vsftpd与xrdp共有的中等风险待监控函数对应的kprobe结构体数组 */
static struct kprobe* mediumRiskFunc_kps_common[] = {};
/* vsftpd独有的中等风险待监控函数对应的kprobe结构体数组 */
static struct kprobe* mediumRiskFunc_kps_vsftpd[] = {};
/* xrdp独有的中等风险待监控函数对应的kprobe结构体数组 */
static struct kprobe* mediumRiskFunc_kps_xrdp[] = {};

/* 以上不同的kprobe结构体数组对应绑定相应后缀的不同中等风险处理函数 */
static int pre_handler_warning_common(struct kprobe *p, struct pt_regs *regs);
static int pre_handler_warning_vsftpd(struct kprobe *p, struct pt_regs *regs);
static int pre_handler_warning_xrdp(struct kprobe *p, struct pt_regs *regs);
```

4.6 本章小结

本章在上一章对内核函数潜在安全风险等级分级研究的基础上，设计并实现了本文的系统防护方法及相应的防护原型。具体来说，本章首先阐述了基于内核函数监控的 Linux 系统防护方法的总体设计框架和实现流程，接着详细说明了原型系统各组成部分及其构建过程各环节的具体细节与实现方法。包括待监控函数的获取方法、监控模块的框架设计及其代码的自动生成和对异常调用情况的分类处理办法等。最后完成了原型系统在实际服务器环境中的构建与部署。

5 原型测试与结果分析

本章主要介绍在系统防护原型构建完成后所进行的功能测试与性能测试等系列实验和结果分析。功能测试部分将主要针对服务进程可能存在的两种权限情况分别进行功能测试与分析论证。而性能测试则主要测试部署防护系统前后系统各性能指标的变化情况，并根据测试结果给出相关结论。

5.1 功能测试与分析论证

5.1.1 特权服务进程攻击防护测试验证

关于原型系统对特权服务进程的攻击防护，本文以 vsftpd 为例进行了测试验证。针对 vsftpd 的攻击一般从官方源代码所存在的恶意后门或漏洞等入手，或者通过代码注入、缓冲区溢出^[45]或 ROP（Return-oriented Programming）^[46]等改变程序控制流的方法来具体实施。考虑到一些真实存在的场景，本文准备了两份以后门方式组织的恶意代码并触发执行，进而根据已部署的防护系统所产生的报警与响应信息，来说明相关防护机制的有效性。

(1) Bind Shell 后门

“Bind Shell”是 2011 年 vsftpd 官方代码托管服务器被黑客成功入侵时曾被植入 2.3.4 版 vsftpd 源代码中的后门程序。本文以其为模板并将之植入到了目标服务器上运行的 vsftpd-3.0.3 中。该后门程序的功能是，当攻击者使用特定的条件对 vsftpd 发起访问请求时，vsftpd 会凭借自身的特权打开系统的 6200 端口，并在其上绑定一个以超级管理员身份启动的 shell。此后攻击者只要通过 telnet 命令来访问目标机器的 6200 端口，便可以利用这个 shell 得到系统的超级管理员权限。

相关功能测试实验表明，每当触发有关后门代码时，内核监控模块均及时将 vsftpd 杀死（参图 5-1 与 5-2 所示）从而阻止了 6200 端口的开启以及后续可能的攻击过程。并且产生了报警日志（参图 5-3 所示）。紧接着，Zabbix 重启 vsftpd 服务进程，并向管理员发送了相应的告警邮件。（参图 5-4 与图 5-5 所示）

```
root@DellR730:/home/lc/Experiment/vsftpd-3.0.3# ./vsftpd ./vsftpd.conf
■
root@DellR730:/home/lc/Experiment/vsftpd-3.0.3# ./vsftpd ./vsftpd.conf
已杀死
root@DellR730:/home/lc/Experiment/vsftpd-3.0.3# □
```

图 5-1 内核监控模块对 Bind Shell 后门异常情况的响应

Figure 5-1 Reaction for kernel monitoring module to backdoor of Bind Shell

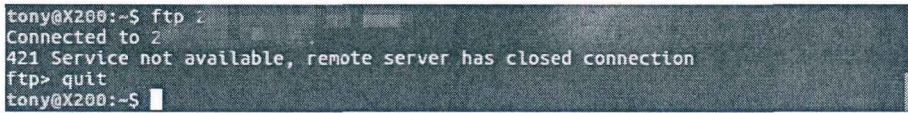


图 5-2 试图远程触发 Bind Shell 后门时的反馈

Figure 5-2 Feedback when trying to trigger backdoor of Bind Shell remotely

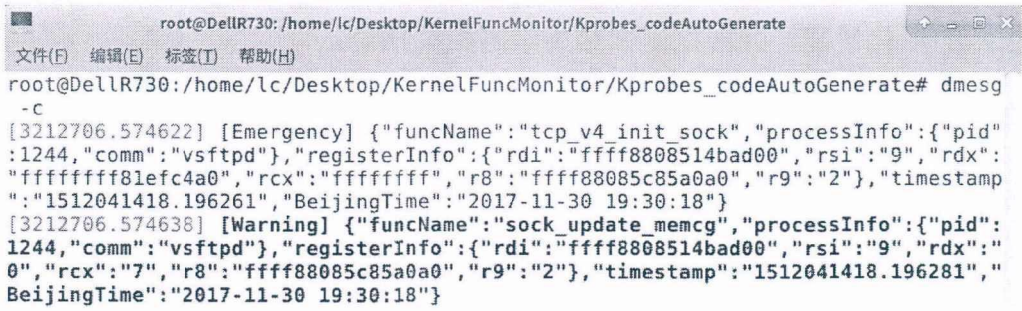


图 5-3 内核监控模块关于 Bind Shell 后门的报警日志

Figure 5-3 Alarm log about backdoor of Bind Shell generated by kernel monitoring module

2017-11-30 19:30:18		Emergency	PROBLEM	High	0	No	2		
Time	User	Details					Status	Info	
		Command:							
2017-11-30 19:30:19		Zabbix server sudo /home/lc/Desktop/KernelFuncMonitor/handler.sh 'Nov 30 19:30:18 DellR730 kernel: [3212706.574622] [Emergency] {funcName:"tcp_v4_init_sock",processInfo:{pid:1244,comm:"vsftpd"},registerInfo:{rdi:"ffff808514bad00",rsi:"9",rdx:"ffff808514bad00",rcx:"ffff808514bad00",r8:"ffff808514bad00",r9:"2"},timestamp:"1512041418.196261","BeijingTime":"2017-11-30 19:30:18"}'					Executed		
2017-11-30 19:30:19	Admin (Zabbix Administrator)	sendMail					Sent		

图 5-4 Zabbix 对 Bind Shell 后门异常情况的响应

Figure 5-4 Reaction of Zabbix to backdoor of Bind Shell

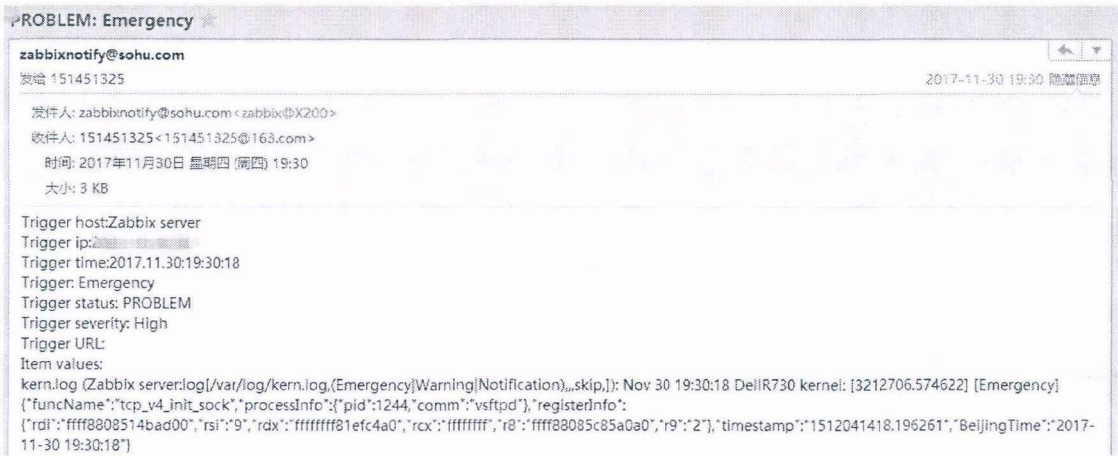


图 5-5 Zabbix 针对 Bind Shell 后门异常情况向系统管理员发送的告警邮件

Figure 5-5 Alarm mail about backdoor of Bind Shell sent to system administrator by Zabbix

需要强调的是，尽管额外植入的后门代码所使用的与网络相关的系统调用在 vsftpd-3.0.3 的源码中都曾出现过，但由于二者在创建 socket 时传入的参数并不相同，进而导致底层执行的内核函数也不尽相同，所以后门代码的执行因触发了 vsftpd 对应的监控函数集中的内核函数（见图 5-3）而被监控模块成功拦截。这也从侧面说明了基于内核函数的细粒度监控较难被恶意攻击绕过。

另外，Zabbix 中设置的 execShell 响应动作（重启服务进程）在原型设计中原本只与 Warning 等级的触发器绑定，但此处为了同时展示其在 Zabbix 中的响应效果（见图 5-4），也与 High 级别的触发器进行了绑定。

(2) 非法加载内核模块

为了更进一步地说明问题，本文验证了第四章已提及的另一种可能的攻击场景，亦即将加载一个恶意内核模块的后门代码植入到 vsftpd 中。当攻击者触发后门代码时，该恶意内核模块便被 vsftpd 加载到内核。由于其不易被管理员发现，故而可能长期驻留于内核中，并收集当前机器的某些敏感信息或进行其它恶意活动。

同样地，相关功能测试实验表明，每当触发有关恶意代码时，内核监控模块均可如预期地那样将异常进程杀死并产生相关报警日志，同时阻止恶意模块的加载（从图 5-6 中可以看到，security_kernel_module_from_file()函数如预期地返回了-1，导致模块加载失败），并重启 vsftpd。相关异常情况所产生的内核日志记录及 Zabbix 对异常情况的响应情况分别如图 5-6 及图 5-7、图 5-8 所示。

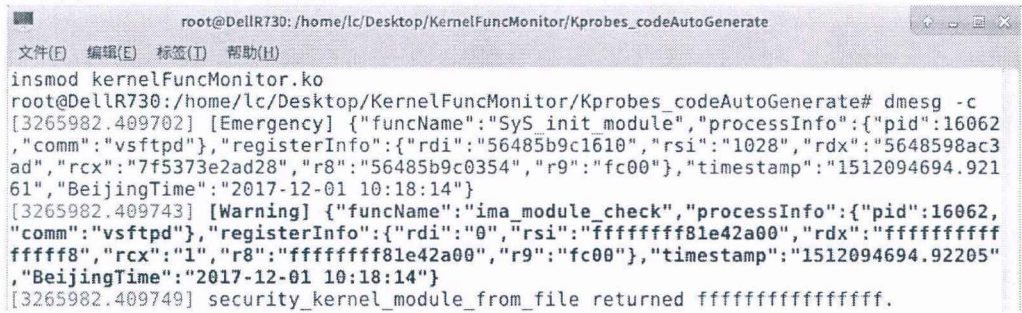


图 5-6 内核监控模块关于非法加载内核模块的报警日志

Figure 5-6 Alarm log about illegal kernel module loading generated by kernel monitoring module

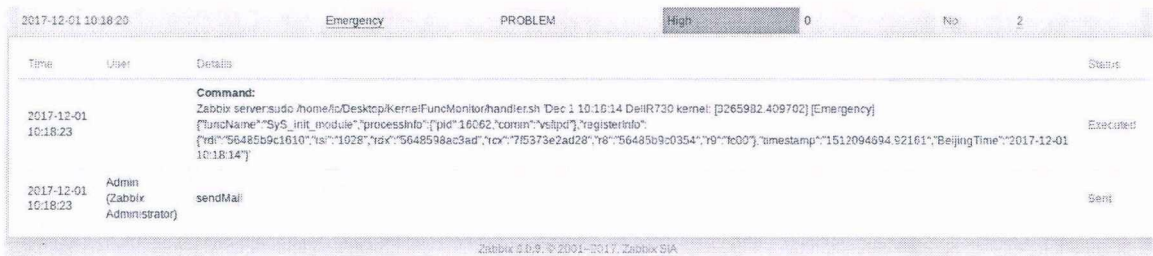


图 5-7 Zabbix 对非法加载内核模块异常情况的响应

Figure 5-7 Reaction for Zabbix when loading illegal kernel module

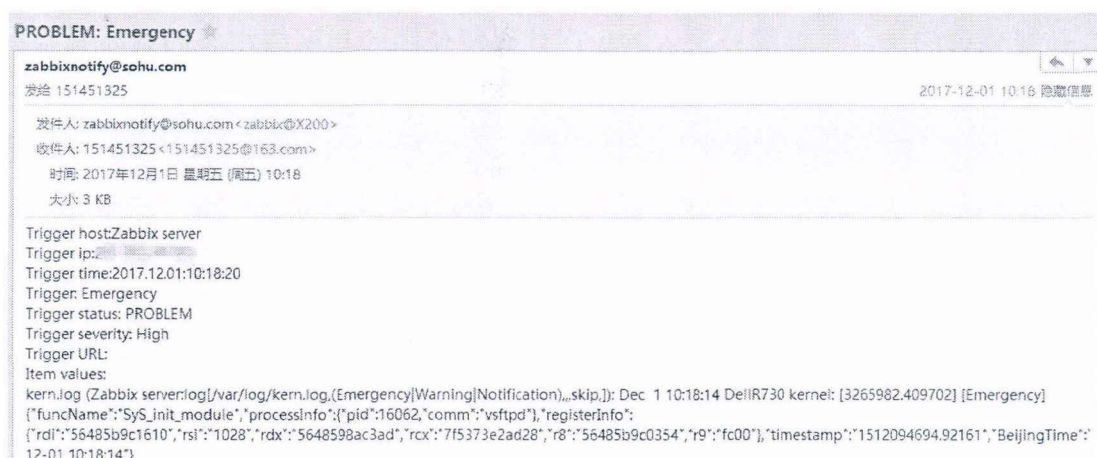


图 5-8 Zabbix 应对非法内核模块加载向系统管理员发送的告警邮件

Figure 5-8 Alarm mail sent to system administrator by Zabbix when loading illegal kernel module

由于 vsftpd 的正常功能几乎从未涉及到与内核模块加载相关的内核函数，所以 vsftpd 对这一系列内核函数的调用都会无一例外地触发报警与拦截。

5.1.2 非特权服务进程攻击防护分析论证

关于原型系统对非特权服务进程的攻击防护，本文以 xrdp 为例进行分析论证。对于以普通用户身份运行的服务进程 xrdp，恶意攻击者同样可以利用其自身漏洞（如 CVE-2017-6967 等）或后门等途径来控制它，进而获得本地普通权限。但为了达到其它非法目的，攻击者往往需要继续借助其它途径特别是利用系统中的内核漏洞来进一步提升权限。这里以两个较新的高危内核漏洞 CVE-2015-8660 和 CVE-2016-8655 为例来具体说明。

CVE-2015-8660 漏洞是由于 fs/overlayfs/inode.c 中的 ovl_setattr() 函数存在缺陷所导致的^[47]。该漏洞可以帮助已取得服务进程本地权限的攻击者，进一步窃取到服务器的超级管理员权限。但在本文系统原型防护下，由于 ovl_setattr() 函数存在于 xrdp 的待监控名单之列，所以一旦 xrdp 触发该函数，监控模块便会进行相应拦截，破坏掉漏洞触发条件，进而能成功阻止攻击者通过 xrdp 对系统实施侵害。

CVE-2016-8655 漏洞是由于 /net/packet/af_packet.c 中 packet_set_ring() 和 packet_setsockopt() 函数存在设计缺陷而导致的，属于条件竞争漏洞^[48]。攻击者可以构造两者特定的调用时序来触发该漏洞。如果 packet_set_ring() 在创建环形缓冲区时 packet 版本为 TPACKET_V3，那么其会初始化一个 timer_list 结构体。但如果在该函数返回之前，另一个线程通过 packet_setsockopt() 函数成功将 packet 版本修改为 TPACKET_V1，那么当套接字关闭后，此前产生的 timer_list 结构体并不会被删

除，从而导致该结构体中相关指针出现释放后重用（use after free）隐患。这一漏洞也能够帮助攻击者直接取得系统的超级管理员权限，但由于 xrdp 的正常运行过程并不需要 packet_set_ring()和 packet_setsockopt()这两个内核函数，所以如果 xrdp 试图去使用它们竞争上述特定的漏洞条件，那么防护系统将会及时发现和制止相关行为，从而能够成功阻止攻击者对相关漏洞的恶意利用。

随着 Linux 内核代码的不断扩充，类似的内核漏洞还有很多，表 5-1 列出了由 xrdp 触发的条件下，部分其它可被拦截的漏洞信息，能够实现拦截的原因均与上述例子类似，故不再赘述。由于本文方法实时监控所有 Linux 服务器中服务进程不该访问的内核函数并设置了相关异常情况拦截手段，基本上阻止了服务进程对其不该调用的内核函数的异常调用行为，从而可以有效地阻止相关恶意攻击的发生，进而提升系统的安全水平。

表 5-1 其它由 xrdp 触发时可被拦截的内核漏洞示例

Table 5-1 Other examples of kernel vulnerabilities that can be intercepted when triggered by xrdp			
漏洞编号	可能危害	所涉内核函数	受影响的内核版本
CVE-2016-8630	拒绝服务	x86_decode_insn()	4.8.7 之前
CVE-2016-7911	拒绝服务/权限提升	get_task_ioprio()	4.6.5 之前
CVE-2016-7910	权限提升	disk_seqf_stop()	4.7.1 之前
CVE-2016-7042	拒绝服务	proc_keys_show()	4.8.2 之前
CVE-2016-1583	拒绝服务/权限提升	ecryptfs_privileged_open()	4.6.2 之前
CVE-2016-0728	拒绝服务/权限提升	join_session_keyring()	4.4.1 之前

5.2 性能测试

为了确定本文方法所带来的额外系统开销，本文采用为 UNIX/POSIX 定制的轻量级系统性能测试工具 LMBench^[49]对部署防护系统前后的系统性能特别是时延和带宽等测评指标进行了测试对比。

5.2.1 时延测试

时延测试的性能指标包括进程上下文切换、文件操作、进程创建、信号处理以及内存访问等常见操作和各类系统调用的时间开销，具体测试结果参图 5-9 至图 5-11 所示。可以看出，在部署防护系统之后，系统调用 read 操作的额外开销相对较大，其它指标的变化则并不明显。

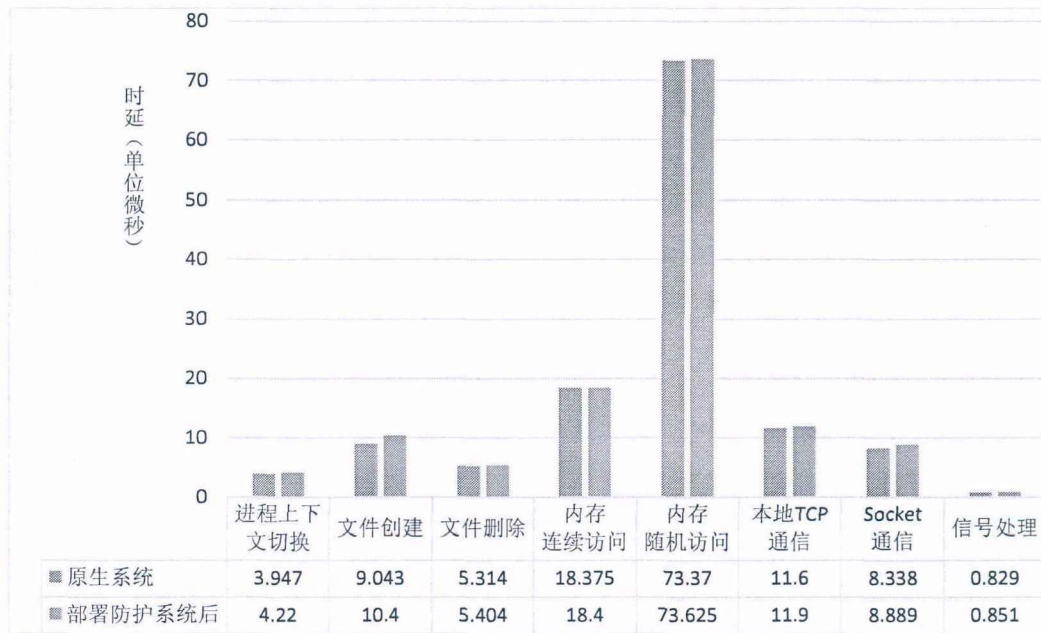


图 5-9 常见操作时延指标测试结果及对比

Figure 5-9 Test results and comparison of common operation delay

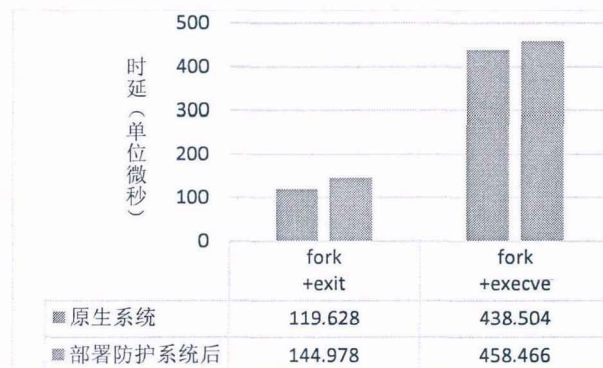


图 5-10 进程创建系统调用的时延指标测试结果及对比

Figure 5-10 Test results and comparison of delay of the system call that created the process

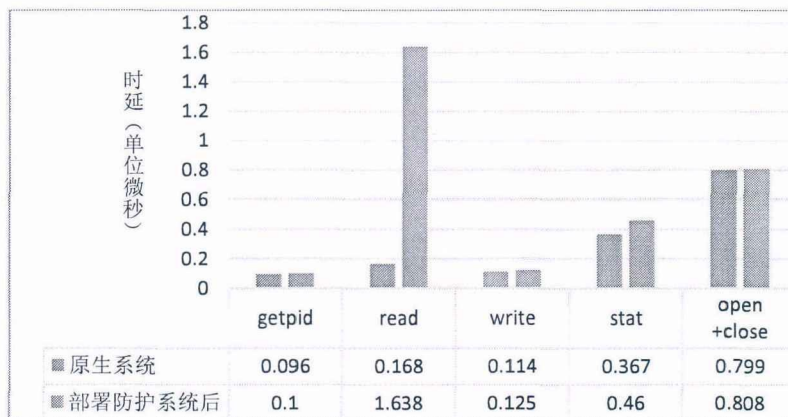


图 5-11 其它系统调用的时延指标测试结果及对比

Figure 5-11 Test results and comparison of other system call delay

事实上,LMbench 测试 read 系统调用时延开销的代码读取的目标是/dev/zero。这是 Linux 下的一个伪设备（文件），对其进行读取将始终得到所需读取长度个空字符（ASCII 中的 0）。通过对内核相关源码的分析得知，实际的读取由 read 系统调用中担当关键作用的__vfs_read()回调相关文件的 file_operations 结构体指针成员 read_iter 所指向的 read_iter_zero()函数（/drivers/char/mem.c）来完成。该函数中的迭代读取机制以循环的方式一次最多读取所谓 PAGE_SIZE 个字节，且每次读取完成后都会调用 cond_resched()函数主动放弃处理器，并进入就绪队列等待重新调度。所以读操作的时延测试结果实际上还包括了中间调度环节的各种其它因素的开销，而这些开销在加载内核监控模块之后被叠加放大，从而导致了 read 操作时延测试结果在部署防护系统前后存在较大的差距。

5.2.2 带宽测试

带宽测试包括缓存文件读取、内存拷贝、内存读写、管道通讯、TCP 连接等，具体测试结果参表 5-2 所示。不难看出，部署防护系统后，系统的一系列带宽虽然有所下降，但均在可接受范围内。

表 5-2 带宽指标测试结果及对比

Table 5-2 Test results and comparison about bandwidth

带宽（MB/s）	原生系统	部署防护系统后	额外开销占比（%）
管道通信	3854.330	3773.0	2.156
Socket 通信	9613.0	9510.0	1.083
TCP	5880.670	5394.0	9.022
文件读取	6792.630	6768.975	0.349
内存映射	11.8K	11.7K	0.855
内存拷贝	6545.130	6542.475	0.041
读内存	12K	11.67K	2.828
写内存	8579.500	8576.330	0.037

5.3 相关工作比较

总的来说，本文基于内核函数调用监控的系统安全机制建立在内核层面，相较于系统调用的异常行为监控来说具有更精细的监控与防护粒度。所以其能够实现对传统访问控制技术无法管控的函数代码的“访问控制”，并且更难被恶意攻击所绕过。此外，本文防护方法比基于 LSM 的内核沙箱技术^[17]具有更少的性能损耗，

而相对内核裁剪等技术则支持动态配置部署，无需重新编译和重构整个内核。

特别地，本文与运行时攻击面缩减框架 kRazor 相比，所做的改进与创新工作主要包括：

(1) kRazor 为了尽可能考虑某些本来合法但只有极少情况才会被服务进程触发的内核函数，而采用静态分析方法对内核函数白名单进行了较大程度的扩充，并由此把服务进程本来不会调用的某些函数排除在监控列表之外。这将会提升漏报率和造成潜在的安全隐患。本文则致力于构造一个实用的防护系统，严格按照统一标准获取待监控内核函数列表，使其尽可能接近理论列表。

(2) kRazor 在每次跟踪缓冲区被填满时按一定调用次数累加确定频繁被调即不适合探测的函数。但实际上，用来存放跟踪结果的环形缓冲区一般很快就会被新的记录所填满，故而正好填满的时机几乎很难准确把握，从而可能导致记录覆盖和跟踪数据丢失。为此，本文采用了更为可取的固定时间片累加获取方式，并适当提高缓冲区大小以尽量避免跟踪数据的覆盖和丢失问题。

(3) kRazor 对除频繁被调函数之外的内核函数都进行了监测，并在期间动态核实被调内核函数是否属于白名单，这显然会给系统带来许多不必要的开销。而本文则采取在监控开始前就将白名单函数从监控函数集中剔除的方式来避免这一问题。

(4) 最后特别需要指出的是，kRazor 只是一个攻击面缩减框架，并没有提供完备的异常情况处理机制，而只是在异常情况发生时不加区分地触发一个 `kernel oops`^[50]，即一种失败立停（fail-stop）性质的内核异常。此后内核可能出现停止运行等严重情况进而导致系统宕机，而这在实际应用中是无法容忍的。本文则构建和实现了综合性的异常情况处理系统，并做到了对所监控函数的分级分类处理，增强了有关方法和技术的实用性和有效性。

5.4 本章小结

本章对系统防护原型进行了功能测试、性能测试以及相关分析与对比工作。功能测试验证了本文方法的有效性，主要采取向特权服务进程 `vsftpd` 中植入后门代码并触发的方式进行，同时也结合了理论论证方式来说明防护方法对于非特权进程 `xrdp` 同样有效。而性能测试方面则使用了开源测试套件 `LMbench` 来测试评估部署防护原型前后系统的各类性能指标变化情况。相关结果表明，本文方法能有效地阻止针对服务器的恶意攻击，且带来的额外性能开销是完全可以接受的。

6 总结与展望

本文在研究前人相关工作的基础上提出了一种可实际应用于 Linux 服务器且具有较完善的异常情况分类处理措施的系统防护方法。相关防护原型基于 Kprobes 机制和内核可加载模块实现,将监控粒度细化到了内核函数层面,同时结合了内核函数潜在安全风险等级划分结果和灵活的 Zabbix 异常报警机制,以支持对内核函数异常调用的分级分类处理,既便于实际使用与部署,又可以使系统管理员及时发现有关服务进程的异常情况并加以适当的处理,从而能够大幅度提升服务器系统的安全水平。实验结果表明,相关方法可切实有效地阻止恶意行为对系统造成侵害,且带来的额外开销完全可以接受,所以具有一定的理论研究价值和实践应用价值。

6.1 工作总结

具体地,本文的研究工作主要体现在以下几个方面:

(1) 提出以系统调用的潜在安全风险等级作为主要参照来划分内核函数安全等级的思想,进而提出内核函数潜在安全风险等级分级标准,并以具体内核版本为例加以具体实施与说明。其中高安全风险内核函数兼顾考虑了高安全风险系统调用涉及到的关键内核函数、存在已知漏洞的内核函数以及一些前人分析研究过或公认的安全相关内核函数。该分类体系可供基于内核函数的相关监控防护方法借鉴与参考,并对完善异常处理机制提供一定的支持。

(2) 提出一套完备的基于内核函数监控的 Linux 系统防护方法,并设计与实现了其中的各部分细节。具体来说,首先设计了各类内核函数名单的确定方法,并以此在实际的应用环境中借助 `fttrace` 开展了相关函数集的采集工作。然后提出基于 Kprobes 机制的内核函数实时监控框架,并以其为模板设计了监控模块代码的自动生成方法,进而开展实际应用环境下的实时监控。最后以(1)中所述的内核函数潜在安全风险等级分级体系为基础,结合 Zabbix 监控系统设计并实现了防护原型对异常情况的分级分类处理机制。

(3) 对本文原型系统进行了必要的测试。首先设计了相关恶意后门实验验证了本文方法对特权服务进程的相关恶意攻击具有防护有效性。然后结合相关攻击利用方式以及内核漏洞触发原理,论证了方法对由非特权服务进程发起的恶意行为同样能进行有效防护。最后采用 `LMbench` 测试套件完成对原型系统的性能测试,并得出本文防护方法具有完全可接受的性能开销的结论。

6.2 研究展望

当然,现有方法在某些方面还存在一定的局限性和不足之处,需要继续研究与完善,具体包括:

(1) `ftrace` 所能跟踪的内核函数集合,并不等同于真正的内核函数总集,其未能跟踪的内核函数也可能存在安全隐患,未来可考虑采用更全面的方式来获取更完备的监控函数集。

(2) 为提高内核函数白名单的收集效率和缩短相应时间,可考虑采集过程与软件测试技术的有机结合,即通过设计完备的服务进程测试用例来实现短时间内对相应进程绝大部分正常功能操作的全面覆盖,从而快速获取对应的白名单。

(3) 为能在一定程度上防止攻击者仅通过白名单函数组织恶意行为和绕过有关系统防护机制,也可对白名单函数按照对应的调用模式(例如调用频次或调用序列)来实施监控,从而进一步完善系统防护体系。

(4) 因为时间与精力等原因,本文只是给出了一种内核函数潜在安全风险等级分级标准和相关示例分析与说明,并没有对所有内核函数进行更为精细与详尽的划分。要完成这一点,工作量是颇为可观的,所以未来可建立完善的由系统调用逐级深入的潜在安全风险等级划分与计算方法,并借助于自动化的静态分析手段,构建出完备的内核函数安全等级体系。

参考文献

- [1] Common Vulnerabilities and Exposures [OL]. <http://cve.mitre.org>.
- [2] Steven A. Hofmeyr, Stephanie Forrest, Anil Somayaji. Intrusion detection using sequences of system calls [J]. Journal of Computer Security, 1998, 6:151-180.
- [3] Shayan Eskandari, Wael Khreich, Syed Shariyar Murtaza, et al. Monitoring system calls for anomaly detection in modern operating systems [C] // IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), 2013.
- [4] Syed Shariyar Murtaza, Wael Khreich, Abdelwahab Hamou-Lhadj, et al. A Host-based anomaly detection approach by representing system calls as states of kernel modules[C] // IEEE International Symposium on Software Reliability Engineering, 2013.
- [5] 黄飞. 基于进程行为的主机异常检测系统[D]. 扬州:扬州大学, 2008.
- [6] Jonathon Ng, Deepti Joshi, Shankar M. Banik. Applying data mining techniques to intrusion detection[C] // 12th International Conference on Information Technology, 2015.
- [7] Gideon Creech, Jiankun Hu. A semantic approach to host-based intrusion detection systems using contiguous and discontiguous system call patterns [J]. IEEE Transactions on Computers, 2014, 63(4):807-819.
- [8] Lin Y, Zhang Y, Ou Y J. The Design and Implementation of Host-based Intrusion Detection System[C] // Third International Symposium on Intelligent Information Technology and Security Informatics. IEEE, 2010:595-598.
- [9] Ma W, Duan P, Liu S, et al. Shadow attacks: automatically evading system-call-behavior based malware detection [J]. Journal in Computer Virology, 2012, 8(1-2):1-13.
- [10] Z. Cliffe Schreuders, Christian Payne. Reusability of functionality-based application confinement policy abstractions[C] // International Conference on Information & Communications Security, 2008.
- [11] 李晨, 涂碧波, 孟丹, 等. 基于多安全机制的 Linux 应用沙箱的设计与实现[J]. 集成技术, 2014(4):31-37.
- [12] 赵旭, 陈丹敏, 颜学雄, 等. 沙箱技术研究综述[J]. 中原工学院学报, 2014, 25(4):1-5.
- [13] Hurtuk J, Baláz A, Ádám N. Security Sandbox Based on RBAC Model[C] // 11th IEEE International Symposium on Applied Computational Intelligence and Informatics.IEEE, May 12-14, 2016, Timișoara, Romania.
- [14] 钟国春. 构造 chroot 服务监禁[J]. 开放系统世界, 2004(11):36-40.
- [15] seccomp-维基百科[OL]. <https://en.wikipedia.org/wiki/Seccomp>, 2017-10-14.
- [16] Bo M, Mu D, Wei F, et al. Improvements the Seccomp sandbox based on PBE theory[C] // 27th International Conference on Advanced Information Networking and Applications Workshops. IEEE, 2013:323-328.
- [17] 程香鹏, 陈莉君. 基于 LSM 的沙箱模块设计与实现[J]. 计算机与数字工程, 2014, 42(8):1521-1525.
- [18] 林绅文. 基于 LSM 框架的安全增强型文件系统的研究[D]. 北京:北京邮电大学, 2008.
- [19] 黄义文. Linux 操作系统内核裁剪的分析[J]. 中国民航飞行学院学报, 2010, 21(3):56-59.

- [20] 徐晨辉. 嵌入式 Linux 内核裁剪及移植的研究与实现[D]. 上海:东华大学, 2009.
- [21] Kurmus A, Sorniotti A, Kapitza R. Attack Surface Reduction for Commodity OS Kernels: Trimmed garden plants may attract less bugs[C] // EUROSEC '11, April 10, 2011, Salzburg, Austria.
- [22] Kurmus A, Dech S, Tu B, Quantifiable Run-time Kernel Attack Surface Reduction[C] // Proc of the 11th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, 2014:212-234.
- [23] Bryan M. Cantrill, Michael W. Shapiro and Adam H. Leventhal. Dynamic Instrumentation of Production Systems[C] // General Track: Usenix Technical Conference, 2004, 7(2):15-28.
- [24] 刘明, Ftrace 简介[OL], <https://www.ibm.com/developerworks/cn/linux/l-cn-ftrace>, 2009-10-15.
- [25] debugfs 官方文档[OL]. <https://www.kernel.org/doc/Documentation/filesystems/debugfs.txt>.
- [26] procfs 官方文档[OL]. <https://www.kernel.org/doc/Documentation/filesystems/proc.txt>.
- [27] sysfs 官方文档[OL]. <https://www.kernel.org/doc/Documentation/filesystems/sysfs.txt>.
- [28] Daichi Fukui, Mamoru Shimaoka. Annotatable Systrace: An Extended Linux ftrace for Tracing a Parallelized Program[C] // The 2nd International Workshop Conference on Software Engineering for Parallel Systems, October 2015.
- [29] ftrace 官方文档[OL]. <https://www.kernel.org/doc/Documentation/kprobes.txt>.
- [30] 向勇, 汤卫东, 杜香燕, 等. 基于内核跟踪的动态函数调用图生成方法[J]. 计算机应用研究, 2015, 32(4):1095-1099.
- [31] Kernel Probes [OL]. <https://sourceware.org/systemtap/kprobes/>.
- [32] Jim K, Prasanna S P, Masami H. Kernel Probes (Kprobes) [OL]. <https://www.kernel.org/doc/Documentation/kprobes.txt>, 2017-07-15.
- [33] 黄杰, 翟高寿. 针对内核非控制数据攻击的在线监测方法研究[J]. 计算机应用与软件, 2017, 34(2):325-333.
- [34] Zabbix 3.0 官方文档[OL]. <https://www.zabbix.com/documentation/3.0/manual/introduction>.
- [35] 徐明, 陈纯, 应晶. 基于系统调用分类的异常检测[J]. 软件学报, 2004, 15(3):391-403.
- [36] 谭茁. 设备驱动非内核化通信机制的研究与实现[D]. 北京:北京交通大学, 2010.
- [37] Daniel P.Bovet, Marco Cesati. Understanding the Linux Kernel [M]. 0'REILLY. 2006.
- [38] Robert Love. Linux Kernel Development (Second Edition) [M]. Novell. 2006.
- [39] 杨娜. Linux 内核安全测试的研究[D]. 北京:北京交通大学, 2010.
- [40] Atiya Mumtaz. Survey Paper on Adding System Call in Linux Kernel 3.2+ & 3.16[J]. International Journal of Science and Research. 2015, 78.96:1155-1157.
- [41] 王曼丽. 基于 GCC 编译器插件的内核安全加固方法研究[D]. 北京:北京交通大学, 2015.
- [42] 武成岗, 李建军. 控制流完整性的发展历程[J]. 中国教育网络, 2016(4):52-55.
- [43] Wang C W, Chen C K, Wang C W, et al. MrKIP: Rootkit Recognition With Kernel Function Invocation Pattern [J]. Journal of information science and engineering 31, 2015, 12(1):455-473.
- [44] Rafik F, Michel D. Efficient Conditional Tracepoints in Kernel Space [J]. The Open Cybernetics & Systemics Journal. 2012, 6(1):11-25.
- [45] 蒋卫华, 李伟华, 杜君. 缓冲区溢出攻击: 原理, 防御及检测防御及检测[J]. 计算机工程, 2003, 29(10):5-7.
- [46] 黄志军, 郑滔. 基于 Return-Oriented Programming 的程序攻击与防护[J]. 计算机科学, 2012, 39(6):1-5.

- [47] CVE, CVE-2015-8660 [OL]. <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-8660>, 2015-12-23.
- [48] CVE, CVE-2016-8655 [OL]. <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-8655>, 2016-10-12.
- [49] Staelin, Carl. Imbench: an extensible micro - benchmark suite[J]. Software Practice & Experience, 2010, 35(11):1079-1105.
- [50] Yoshimura T, Yamada H, Kono K. Is Linux kernel oops useful or not? [C] // Eighth Usenix Conference on Hot Topics in System Dependability. 2012:2-2.

附录 A

CVE 中 Linux 4.4 版本存在的部分内核函数漏洞示例及说明

(1) CVE-2016-8650

Linux kernel 4.8.11 及之前版本中的 `lib/mpi/mpi-pow.c` 文件的 `mpi_powm()` 函数存在安全漏洞。该漏洞源于程序没有正确为 `limb` 数据分配内存。本地攻击者可通过 `add_key` 系统调用利用该漏洞造成拒绝服务（栈内存损坏）。

(2) CVE-2016-8630

Linux kernel 4.8.7 之前的版本中的 `arch/x86/kvm/emulate.c` 文件的 `x86_decode_insn()` 函数存在拒绝服务漏洞。本地攻击者可利用该漏洞造成拒绝服务（主机操作系统崩溃）。

(3) CVE-2016-7911

Linux kernel 4.6.5 及之前的版本中的 `block/ioprio.c` 文件的 `get_task_ioprio()` 函数存在竞争条件漏洞。本地攻击者可通过特制的 `ioprio_get` 系统调用利用该漏洞获取权限或造成拒绝服务（释放后重用）。

(4) CVE-2016-7910

Linux kernel 4.7 及之前的版本中的 `block/genhd.c` 文件的 `disk_seqf_stop()` 函数存在释放后重用漏洞。本地攻击者可利用该漏洞获取权限。

(5) CVE-2016-7042

Linux kernel 4.8.2 及之前版本中的 `security/keys/proc.c` 文件中的 `proc_keys_show()` 函数存在安全漏洞。该漏洞源于程序对超时数据使用错误的缓冲区大小。本地攻击者可通过读取 `/proc/keys` 文件利用该漏洞造成拒绝服务（栈内存损坏和错误）。

(6) CVE-2016-6187

Linux kernel 4.6.5 之前的版本中的 `security/apparmor/lsm.c` 文件的 `apparmor_setprocattr()` 函数存在安全漏洞。该漏洞源于程序没有验证缓冲区大小。本地攻击者利用该漏洞可获取权限。

(7) CVE-2016-2545

Linux kernel 4.4.1 之前版本的 `sound/core/timer.c` 文件中的 `snd_timer_interrupt()` 函数存在安全漏洞。该漏洞源于程序没有正确维持特定的链表。本地攻击者可借助特制的 `ioctl` 调用利用该漏洞造成拒绝服务（竞争条件和系统崩溃）。

(8) CVE-2016-1583

Linux kernel 4.6.2 及之前版本的 `fs/ecryptfs/kthread.c` 文件中的 `ecryptfs_privileged_open()` 函数存在安全漏洞。本地攻击者可借助精心构造的 `mmap` 调用来导致递归的缺页异常处理从而利用该漏洞获取权限，或造成拒绝服务（栈内存消耗或破坏）。

(9) CVE-2016-0728

Linux kernel 4.4.1 之前版本的 `security/keys/process_keys.c` 文件中的 `join_session_keyring()` 函数存在安全漏洞。该漏洞源于程序没有正确处理特定错误中的对象引用从而导致 `keyrings` 功能中存在引用泄露的可能性。本地攻击者可借助特制的 `keyctl` 命令利用该漏洞完成权限提升或造成拒绝服务（整数溢出和释放后重用）。

作者简历及攻读硕士学位期间取得的研究成果

一、作者简历

刘晨，男，1993 年 12 月生，湖北天门人，硕士研究生。现就读于北京交通大学计算机与信息技术学院计算机科学与技术专业，从事操作系统安全相关研究。

2015 年 9 月一至今，就读于北京交通大学计算机与信息技术学院计算机科学与技术专业，攻读工学硕士学位。

2011 年 9 月—2015 年 6 月，就读于廊坊师范学院数学与信息科学学院信息与计算科学专业，攻读理学学士学位。

二、发表论文

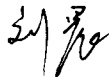
- [1] 翟高寿，刘晨，向勇. 基于内核函数监控的 Linux 系统防护方法的研究与实现[J]. 信息安全.

三、参与科研项目

- [1] 清华大学合作课题 Linux 内核加固与漏洞分析
- [2] 国外教材《基于螺旋式方法的操作系统教程》的翻译工作
- [3] 铁路工作人员考勤测酒仪软件系统的开发与维护工作
- [4] 铁路站场列车防溜逸预警系统的开发工作
- [5] 生物酶电商平台的开发工作

独创性声明

本人声明所呈交的学位论文是本人在导师指导下进行的研究工作和取得的研究成果，除了文中特别加以标注和致谢之处外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得北京交通大学或其他教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示了谢意。

学位论文作者签名：  签字日期： 2018 年 6 月 12 日

学位论文数据集

表 1.1： 数据集页

关键词*	密级*	中图分类号	UDC	论文资助
操作系统安全； 内核安全；系统 防护方法；服务 进程；内核函数 调用	公开	TP316； TP309		
学位授予单位名称*		学位授予单位代 码*	学位类别*	学位级别*
北京交通大学		10004	工学	硕士
论文题名*		并列题名		论文语种*
基于内核函数监控的 Linux 系统防 护方法的研究与实现				中文
作者姓名*	刘晨		学号*	15120418
培养单位名称*		培养单位代码*	培养单位地址	邮编
北京交通大学		10004	北京市海淀区西直 门外上园村 3 号	100044
学科专业*		研究方向*	学制*	学位授予年*
计算机科学与技术		操作系统安全	2.5	2018
论文提交日期*	2018.3			
导师姓名*	翟高寿		职称*	副教授
评阅人	答辩委员会主席*		答辩委员会成员	
	王宁		常晓林、任爽	
电子版论文提交格式 文本（ ） 图像（ ） 视频（ ） 音频（ ） 多媒体（ ） 其他（ ） 推荐格式： application/msword； application/pdf				
电子版论文出版（发布）者		电子版论文出版（发布）地		权限声明
论文总页数*	69			
共 33 项，其中带*为必填数据，为 21 项。				