上下文敏感的控制流完整性保护的改进方法

沈钦涛 张 丽 罗 磊 马 俊 余 杰 吴庆波
(国防科学技术大学计算机学院 长沙 410000)

摘 要 面对控制流劫持攻击的威胁,业界使用控制流完整性保护技术来保障进程的执行安全。传统的控制流完整性验证保护机制依赖于动态二进制改写技术,在分析、实施等过程中难度较大,且有可能带来二进制兼容的问题。通过研究近几年提出的上下文敏感的控制流保护技术PathArmor,分析了其检测进程控制流的时机。然后针对PathArmor只在进程做系统调用时才进行检测的机制,提出了改进的方法。该方法依据内核页错误中断处理机制,通过修改用户页面的保护属性主动触发可执行页面的执行错误;接着,修改页错误中断处理过程,钩挂do_page_fault以处理主动触发的执行错误。用户进程代码和数据的完整性得以保证的同时,得到了更多陷入内核接受检查的机会。在Nginx,bzip2,SQLite等典型应用环境下的实验结果表明,改进的方法能够明显增加系统安全分析的粒度,更好地保护程序的控制流。

关键词 控制流完整性,执行路径,硬件特性,控制流保护,内核陷入
中图法分类号 TP311 **文献标识码** A **DOI** 10.11896/j.issn.1002-137X.2017.11.013

Improved Method of Context-sensitive Control Flow Integrity Protection

SHEN Qin-tao ZHANG Li LUO Lei MA Jun YU Jie WU Qing-bo
(School of Computer, National University of Defense Technology, Changsha 410000, China)

Abstract Facing the threat of control flow hijacking, the industry uses control flow integrity protection technology. It is difficult to achieve the goal for those traditional control-flow integrity protection mechanisms which depend on dynamic binary rewriting technology, and it's not easy for analysis and implementation. It may also bring out the problem of binary compatibility. The recently proposed context-sensitive control-flow integrity, PathArmor, only verifies the control flow when tasks are calling system functions. To achieve enhanced protection, an improved method was proposed in this paper. The improved method means to trigger more page fault intentionally by modifying the pages' protection flag of the target task, with the kernel's mechanism of page fault. Then it hooks the origin system IDT (Interrupt Description Table) and creates new do_page_fault function to handle the generated page fault. With doing some experiments on typical application like nginx, bzip2, SQLite and so on, the result shows that the counts for tasks to be verified increase significantly, and it can get better protection with the improved method.

Keywords Control-flow integrity, Execution path, Hardware feature, Control-flow protection, Kernel trap

1 概述

程序中的代码漏洞给进程的执行安全带来了极大的威胁,控制流劫持是一种常见的攻击技术。随着系统保护的不断增强,控制流劫持技术不断发展,在返回到库函数(return-to-libc)[1-2]等技术的基础上,发展出了返回导向编程(Return-Oriented Programming, ROP)[3]技术,给系统安全带来了极大的威胁。ROP利用二进制程序的漏洞,把已有的、具备特定功能的代码片段(Gadget)串接并精心构造成图灵完备的程序,从而达到执行恶意代码的目的。

为了保护进程的执行过程,早在2005年业界就提出了控制流完整性(Control-Flow Integrity, CFI)[4]的概念,依据分析程序的控制流图(Control-Flow Graph)明确进程接下来要执行的代码,从而验证进程是否正按照预期的路径执行。文献[4]使用插入二进制代码的技术实现了系统保护。该系统首先分析二进制程序的控制流图,获取程序内部的代码信息;然后通过在函数调用的位置插入标号和在函数返回前验证标号的方式,使控制流能够正确返回调用的位置。

目前,控制流完整性保护仍然是系统安全方向的一个关注热点。Bin-CFI[5]使用二进制代码改写技术,在静态分析的基础上使用二进制改写技术在原二进制代码中插入保护段,很好地处理了代码中的函数指针问题。CCFIR[6]采用了动态

到稿日期:2016-10-27 返修日期:2016-12-28 本文受自然科学基金项目(61303191,6130319,61402503)资助。
沈钦涛(1993-),硕士生,主要研究领域为系统安全,E-mail:qintaoshen@ubuntukylin.com;张 丽(1992-),女,硕士生,主要研究领域为静态分析、计算机安全;罗 磊(1984-),男,博士,主要研究领域为模式识别;马 俊(1982-),男,博士,主要研究领域为数据安全;余 杰(1982-),男,博士,副研究员,主要研究领域为网络安全与测量;吴庆波(1969-),男,博士,研究员,主要研究领域为基础软件、操作系统。

改写的方式，在进程的地址空间中开辟了 Springboard 段地址空间，并结合使用地址空间布局随机化技术（Address Space Layout Randomization，ASLR）[7] 对间接转移的过程进行保护，即所有的间接转移地址都将重定向到该段中，并从该段地址空间内重新转移到正确的地址上。这些基于二进制改写技术的 CFI 不仅在分析和实现时面临一定的困难，而且存在二进制代码的兼容问题。

由于二进制改写技术可能面临以上的问题，业界学者开始关注如何利用系统的硬件特性来验证软件的运行时状态。Last Branch Recode 作为通用 x86 处理器平台具有的一个硬件特性，是硬件调试部件的一部分，主要用于记录指令流的最近 16 次转移的源地址和目标地址。ROPecker[8] 利用 LBR 动态检测进程的执行状态。在进程陷入内核时，ROPecker 检查 LBR 中的记录，通过转移指令的特征，判断所有的转移过程是否可能构成一个恶意 ROP 代码链。

PathArmor[9] 也是一种采用了 LBR 的 CFI 技术。它使用用户层程序对静态程序做了详细的分析，并记录可能的转移路径。不同于 ROPecker，PathArmor 验证了 LBR 中的记录是否按照预期的顺序发生，即验证了执行路径是否正确。

ROPecker 和 PathArmor 均摆脱了对二进制改写技术的依赖，不仅获得了使用硬件的性能优势，也避免了二进制改写带来的兼容性问题和对原程序完整性的破坏。ROPecker 等技术提高了攻击者的技术门槛，但仍有被攻破的可能性[10-11]。Path-Armor取得了更大粒度的保护效果，但在选择进程的安全验证时机的问题上，仅采用了较为简单的方式，这给攻击者劫持控制流提供了可乘之机。

本文在 PathArmor 的基础上，设计并实现了一种通过物理页的不可执行属性触发执行错误，使得进程更多地陷入内核的机制，增加了受保护进程接受安全验证的机会，增加了 PathArmor保护的粒度。本文通过对典型应用进行实验对比，检验了代码实现的效果，为后续的研究工作提供了基础。

2 上下文敏感的完整性：PathArmor

静态程序由指令和数据组合而成。一段能够顺序执行的代码的最大集合被定义为一个基本块。在运行时，控制流会在基本块之间不断转移，一个转移过程的来源地址到目的地址的连接被定义为一条执行边。进程执行的上下文是指控制流在当前状态之前执行过的执行边的有序集合和接下来进程按照预期将要执行的边的有序集合。

程序的控制流图可以抽象为一个有向图 $CFG=(V,E)$，其中 V 是能够顺序执行的基本块结点集合；E 代表基本块之间执行边的集合。通过控制流图能够判断进程在运行时控制流的转移过程。

进程是程序加载到内存后的状态，根据系统环境、输入输出等外部条件的差别会有不同的表现。假设基本块集合为 $V=(v_1,v_2,\cdots,v_{i-1},v_i,\cdots,v_m)$，进程控制流图中的某一条边 $e_i(e_i\in E)$ 表示从点 v_{i-1} 转移到点 v_i。那么进程 P 可以抽象为 $P=\{e_1,e_2,\cdots,e_i,\cdots,e_n\}$，表示在执行过程中基本块按照某种执行的先后顺序被连接起来。

传统的控制流完整性验证针对程序代码中的间接转移指令进行加强保护，如 ret，indirect call，indirect jump 等。不同于存储于静态不可写区域的直接转移的目标地址，这些间接转移的目的地址在运行时的堆栈空间中产生，容易被攻击者利用。

对于执行路径的控制流完整性验证，考虑了基本块执行的先后次序。在一个进程中，对于当前的执行边 e_i，一定存在一个已经执行过的边的集合（即执行路径）$E=(e_1,e_2,\cdots,e_{i-1})$，使得边顺序地在 E 之后开始执行。

传统的 CFI 是针对一条或多条后向执行边的检测，例如判断 ret 返回的地址是否正确，或者判断一系列 ret 的地址是否符合 ROP 的特征。相对于传统的 CFI 验证方法，上下文敏感的验证方法能够保证控制流的来路和去向，不仅可以保证执行边的目标地址是合法的，而且对其来路进行了验证分析，进一步强化了保护的粒度，能够大大提高控制流被攻击的门槛。

想要达到对执行路径进行验证的目的，需要解决 3 个方面的问题：1）动态且高效地监测进程的执行路径；2）分析获得二进制程序的控制流图；3）对程序的执行路径做有效性验证。PathArmor 已经为上文所述工作奠定了一定基础。PathArmor 实现了以下 3 个模块。

1）内核模块：能够动态使能 LBR 寄存器；实现 IOCTL() 接口，便于用户层程序使用该模块；进行了内核函数的截取重写，保护安全相关的系统调用；处理进程执行过程中发生的信号、回调、创建子进程、进入动态链接库等过程。

2）路径分析模块：基于 DynInst 技术分析程序的二进制代码，用于获取控制流图。

3）动态检测组件：采取主动分析策略，工作于用户层，存储静态分析的结果；与内核模块通信获取实时信息，验证 LBR 记录的路径并将验证的结果反馈给内核模块。

经过原文中对系统的分析检测，PathArmor 取得了不错的实际保护效果。

3 设计与实现

原来的 PathArmor 系统已经完成了静态分析和硬件驱动的工作。在实际检测进程运行上下文是否正确时，PathArmor 钩挂了安全敏感系统调用，如 read，write，execve，mmap，mprotect 等，在内核层面保证了用户进程在使用这些函数前首先验证 LBR 中记录的进程执行的上下文是否在预期的范围内。

但是由于二进制程序中的基本块之间的调用关系非常复杂，而 LBR 记录的数量有限（硬件层面只有 16 组寄存器可以使用），因此如果恶意攻击劫持控制流的位置距离检测点的跳转次数超出了 LBR 记录的范围，就可以“刷新”LBR 内容。因而即使此时执行到了检测点并且对 LBR 进行了验证，也难以发现 LBR 记录之前是否存在恶意的控制流劫持。即如果攻击者能够在更早的位置篡改控制流，当前的 PathArmor 的检测机制是不能发现的。

基于以上分析，PathArmor 并不能全面地对用户进程进

行监控保护。因而在系统调用之外的位置增加更多的检测点是很有必要的。本文在现有的 PathArmor 的基础上，通过人为创造 Page Fault，并修改原 Linux 内核中 Page Fault 处理过程的方法，在进程因为缺页错误而陷入内核时添加 PathArmor 的检测过程，从而增加了 PathArmor 对用户进程的检测次数，加强了 PathArmor 的保护粒度。

首先，替换掉原有的 Page Fault 处理函数。由于 Linux 系统内核提供的 Probe 方法不能钩挂 Page Fault 函数，本文在内核空间申请了一张新的页面并构造新的中断处理表，通过修改原来的 Page Fault 处理函数指针使之指向新函数的位置。

过程 1　初始模块，修改中断处理表

```
Begin
  Initialize Module
  Ori_IDT:=Default_idtr.address
  Page:=__get_free_page(GFP_KERNEL)
  New_IDT.address:=Page
  Memcpy(New_IDT,Old_IDT,Default_idtr.size)
  New_IDT[PGFAULT_NR]:=New_Do_Page_Fault
  Store_Old_IDT(Ori_IDT)
  Load_New_IDT(New_IDT)
End
```

其次，若要完成前文的想法，则需创造条件以主动触发 Page Fault。本文利用了操作系统保护页面的 NX 机制，其需要利用内存页面(Page)的保护属性触发页执行错误，因此需要对人为触发的页错误进行针对性的处理，并不断修改 PTE 的 X 属性使进程在页面切换时能够源源不断地产生新的页执行错误。这一过程需要在内核模块中完成。在模块初始化时，按照本文的需求对 Do_Page_Fault 进行改写，如过程 2 所示。

过程 2　New_Do_Page_Falut

```
Begin
  Address:=0,Pte:=NULL
  Address:=get_cr2() //获取出错地址
  IF CurrentTask is Monitored:
    Pte:=get_pte(Address)
    IF Pte is in Memory://如果页面已经加载到内存
      IF Address is to execute and Pte is NX:
        Validate_LBR()
        //修改执行属性，执行页面跳转到其他代码页时将触发
        Make_All_Pages_NX(CurrentTask)
        //把本页面标记为可执行
        Make_The_Page_EXEC(Pte)
      END IF
    ELSE://先使用原来的 Page Fault 逻辑把页面加载到内存
      Ori_Do_Page_Fault()
      IF Address is to execute:
        Validate_LBR()
        Make_All_Pages_NX(CurrentTask)
        Make_The_Page_EXEC(Pte)
      END IF
    END ELSE
  END IF
  Ori_Do_Page_Fault()
RETURN
```

在进程加载完成后，把被保护进程的代码页面初始化为不可执行状态。物理页的可执行属性使用 PTE 的最高位表示，内核提供了对页面 X 标志位的操作接口：set_memory_x 和 set_memory_nx。过程 3 实现了对进程所有代码段页面的 NX 标记。

过程 3　INIT_TASK 初始化进程可执行页面

```
Old_Pte:=NULL,Now_Pte:=NULL
Start_Code:=0,End_Code:=0
Begin
  CurrentTask Initialize
  Start_Code:=CurrentTask.mm.start_code
  End_Code:=CurrentTask.mm.end_code
  i:=Start_Code
  //从代码段开始到结束，每次循环标记一个页面
  WHILE i<=End_Code:
    Now_Pte=get_pte(i)
    IF Now_Pte in Memory and Now_Pte ! =Old_Pte:
      Make_The_Pte_NX(Now_Pte)
      //Old_Pte 是为了防止重复标记页面造成性能损耗
        Old _Pte:=Now_Pte
    END IF
    i +=PAGE_SIZE
End
```

本文通过以上过程实现了检测点的插入，并能够在不影响进程代码完整性的情况下实时创造出新的检测点：Linux 在加载一个进程时，并不会把程序的所有内容直接加载到内存(程序比较大时)，而是在需要时再通过缺页错误(Page Fault)把该部分的代码映射到内存中；进程在正常执行的过程中有很多次缺页过程，但这对用户进程而言是透明的。本节描述的方法不仅利用了系统原本的缺页处理过程，也能主动触发页面数据执行错误，给系统添加了更多的安全检测的机会。

4　实验检测与评估

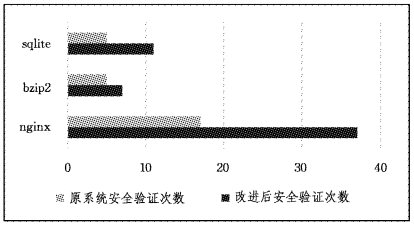
本文通过 Nginx 上的应用就优化之后系统对受保护进程的 LBR 验证次数进行了评估和对比。实验平台如下：处理器为 Intel Core i5，内存为 12BG，操作系统为 Ubuntu Kylin 14.04.3 x86_64，内核版本为 linux-image-3.16.0-77-generic，硬盘为 120G SSD。实验选取了 nginx，bzip2 和 sqlite 作为测试程序，详细情况如表 1 所列。

表 1　测试样例程序

程序	大小	版本	测试条件
Nginx	2.3MB	0.8.54	Web 服务
bzip2	294kB	1.0.6	压缩文件
sqlite	92.8kB	2.8.17	使用 sqlite

基于上述软硬件系统环境，实验的结果如图 1 所示。进程在运行过程中会产生多次页错误，本文只处理代码段缺页

的情况，并没有关注数据寻址引发的缺页。从实验结果可以看出，通过处理代码段页错误的方式能够有效增加安全验证的次数。相比于原来的 PathArmor 系统只在安全相关的系统调用的位置检测验证 LBR 状态，本文提出的方法在实验集中能明显地增加进程执行时安全检测的次数，sqlite，bzip2 和 nginx 的验证次数分别增加了 120%，25%和 117%。



注：sqlite 和 nginx 的测试条件为启动正常服务，bzip2 的测试条件为压缩大小约为 300kB 的文件

图 1 实验结果

从图 1 中看出，bzip2 增加的验证次数最少。bzip2 程序较小，代码段能够占据的内存页面较少；bzip2 在功能上侧重于压缩算法，代码比较紧凑，跨页面的寻址较少。sqlite 编译后的大小仅为 92.8kB，代码段引起的缺页多于 bzip2，但其功能模块多于 bzip2。nginx 获得了更多的验证机会，一方面因为其尺寸较大，引起的正常缺页过程较多；另一方面源于 nginx 的功能模块复杂，它在工作时存在频繁的跨页面调用过程。

PathArmor 的性能受到多方面因素的影响，如程序系统使用 Libc 库函数的频率、函数指针的数量、cache 的命中率等。在 SPEC2006 测试集下，默认配置的 PathArmor 的最低性能负载为 3%[9]。

Apache2-utils 的性能测试工具 ab 能够以指定的速率和并发数请求测试目标服务器的响应速度。本文使用 ab 对 nginx-0.8.54 进行了性能测试，请求的数据为 nginx 服务器默认页面，结果如图 2 所示。

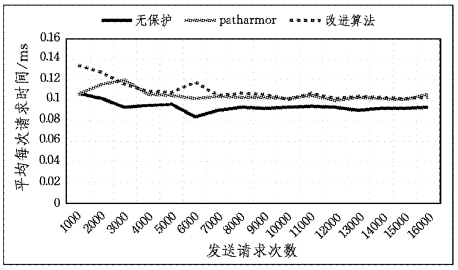


图 2 无保护、使用 PathArmor 和使用改进后的 PathArmor 的性能对比

经过本地环境测试，PathArmor 给 nginx 带来了额外的平均约为 12.3%的负载。本文改进算法的平均负载约为 16.8%。从图 2 可以看出，改进算法在初次测试时带来的性能损耗最大。经过查看调试信息，可以知道这是由于进程刚启动服务时会产生比较多的缺页中断而造成的。但是处理多次请求后，改进算法的性能接近于未改进的 PathArmor。一方面，在图 2 的测试条件下，进程在处理一部分请求后所需的代码和数据页面都已经被加载到内存中，没有产生新的缺页中断；另一方面，由于程序执行的局部性原理，进程在执行态的代码区域比较集中，并不产生多次跳转。同时，在静态分析阶段，当前版本的 PathArmor 避开了 Libc 库中的代码，并不监测进程进入 Libc 库后的控制流图的变化情况，因而当 nginx 使用 Libc 库处理函数时，并没有因当前执行代码页面的转移而验证 LBR 状态。因此，在服务进程多次响应同样的请求后，其性能接近于未改进的 PathArmor。

结束语 本文基于 PathArmor 控制流保护系统，针对进程的运行时验证次数做了一些改进。实验表明，改进后的系统对进程的正确性验证的次数较之前有显著增加，这给了较少使用指定系统调用的用户程序提供了更多的安全保障。经过实际测试，新算法产生了一定的性能损耗。

在实际生产环境中，基本块之间的转移状况会更加复杂。下一步工作将继续探讨针对不同用途和规模的程序，如何更加完善地利用页错误，达到利用 LBR 多次合理陷入内核进行安全分析的目的，同时设计能进一步控制性能损耗的机制。

参 考 文 献

[1] DESIGNER S. Return-to-libc attack[M]. Bugtraq, 1997.

[2] SHACHAM H. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)[C]// Proceedings of the 14th ACM Conference on Computer and Communications Security. ACM, 2007: 552-561.

[3] ROEMER R, BUCHANAN E, SHACHAM H, et al. Return-oriented programming: Systems, languages, and applications[J]. ACM Transactions on Information and System Security, 2012, 15(1): 1-34.

[4] ABADI M, BUDIU M, ERLINGSSON U, et al. Control-flow integrity[C]// Proceedings of the 12th ACM Conference on Computer and Communications Security. ACM, 2005: 340-353.

[5] ZHANG M, SEKAR R. Control Flow Integrity for COTS Binaries[C]// Usenix Security Symposium. 2013: 337-352.

[6] ZHANG C, WEI T, Chen Z, et al. Practical control flow integrity and randomization for binary executables[C]// 2013 IEEE Symposium on Security and Privacy (SP). IEEE, 2013: 559-573.

[7] TEAM P X. PaX address space layout randomization (ASLR) [OL]. http://pax.grsecurity.net/docs/aslr.txt.

[8] CHENG Y, ZHOU Z, MIAO Y, et al. ROPecker: A generic and practical approach for defending against ROP attack[C]// Network & Distributed System Security Sympoisum. 2014.

[9] VAN DER VEEN V, ANDRIESSE D, GÖKTAŞ E, et al. Practical context-sensitive cfi[C]// Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. ACM, 2015: 927-940.

[10] GOKTAS E, ATHANASOPOULOS E, BOS H, et al. Out of control: Overcoming control-flow integrity[C]// 2014 IEEE Symposium on Security and Privacy (SP). IEEE, 2014: 575-589.

[11] DAVI L, LEHMANN D, SADEGHI A R, et al. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection[C]// USENIX Security Symposium. 2014.