代码重用攻击与防御机制综述

柳 童[1,2] 史 岗[1] 孟 丹[1]

[1] 中国科学院信息工程研究所 北京 中国 100093
[2] 中国科学院大学 北京 中国 100049

摘要 由于 C 与 C++等计算机程序中广泛存在的漏洞, 攻击者可以通过这些漏洞读取或篡改内存中的数据, 改变计算机程序原有的执行状态达到破坏的目的。为此研究者进行了不懈地努力并采取了一些卓有成效的保护机制, 例如数据不可执行与内存布局随机化, 这些防御机制对于早期的代码注入攻击起到了极好的防御效果, 然而计算机系统的安全性依然不容乐观。攻击者在无法通过向内存中注入自己的代码并执行的方式完成攻击后, 开始利用内存中原有的代码, 通过控制它们执行的顺序来达到自己的目的, 这种攻击方式称为代码重用攻击, 它具有极大的威胁性, 能够绕过多种现行的安全措施, 并成为攻击者的主流攻击方式。为此, 研究界针对代码重用攻击的研究也逐渐增多。本文简述了代码重用攻击的起源, 攻击实现的方式, 系统化地总结了现有的防御机制并对这些防御机制进行了评价。对代码重用攻击的根本原因进行了简要的分析, 并提出了一种新的防御机制设计思路。

关键词 计算机系统安全; 内存攻击; 代码重用攻击
中图法分类号 TP309.2 DOI 号 10.19363/j.cnki.cn10-1380/tn.2016.02.002

A Survey of Code Reuse Attack and Defense Mechanisms

LIU Tong[1, 2], SHI Gang[1], MENG Dan[1]

[1]Institute of Information Engineering, Chinese Academy of Science, Beijing 100093, China
[2]University of Chinese Academy of Science, Beijing 100049, China

Abstract Due to the wide existence of vulnerabilities in computer programs such as C and C++, computer systems is vulnerable to be tampered by adversary changing the original running states. Researchers have made great efforts and take some effective protection mechanisms, for instance, Data Execution Prevention and Address Space Layout Randomization. These security mechanisms have a great effect against the primitive attack patterns like code-injection attack. However, the security of computer system is still not optimistic. Though the adversary could not inject their own codes into the memory then run them ever again, they began to use the original benign codes in the memory, manipulate them to achieve malicious purpose by changing their order of operating, which is called code-reuse attack. And it is able to bypass a variety of security mechanisms of commodity computer systems, thus it has become a major threat and the main pattern of hacking. For this reason, researches about code-reuse attack have been taken up in recent years. This paper illustrates the origin of code-reuse attack and achieved way of attack, summarizes the existing defense mechanisms and simply evaluates these defense mechanisms systematically. Meanwhile, this paper analyzes briefly the basic reason of code reuse attack and puts forward an new idea of defense mechanism designing.

Key words computer system security; memory security; code-reuse attack

1 引言

计算机的使用在当代社会已经普及到工业、教育、医疗、金融、军事等各个行业。然而, 针对计算机系统的攻击也数不胜数, 造成的损失数以亿计。这是由于在 C 与 C++等计算机程序中普遍存在漏洞, 例如能够对内存的直接存取; 对用户的输入没有自动的边界检查机制, 以及对已经释放的指针再利用等[1]。使攻击者能够利用这些漏洞通过溢出没有边界检查机制的缓冲区来注入恶意代码并篡改内存的函数返回地址, 改变程序控制流使注入的代码得到执行[2], 这就是最早的代码注入攻击。在此之上又发展出了整数溢出、格式化字符串溢出等攻击方式, 实现原理基本相同。计算机系统安全机制的设计者针对

通讯作者: 柳童, 博士研究生, Email: liutong9017@iie.ac.cn。
收稿日期: 2016-2-24; 修改日期: 2015-4-14; 定稿日期: 2016-4-18

这种攻击方式采用了许多卓有成效的防御机制, 例如数据不可执行(Data Execution Prevention , DEP)[3-5], 它将原有在内存中无差别存储的内容分为数据段与代码段, 并划分了不同的权限: 数据不可执行, 代码不可修改。这使攻击者以溢出缓冲区等形式注入的恶意代码不会再被允许执行, 从而抑制了代码注入攻击。

然而这些漏洞仍然在被利用。近年来, 另一种攻击方式开始兴起。一些攻击者开始利用程序在内存中原有的代码, 通过利用漏洞改变它们的执行顺序来达到攻击的目的, 这就是所谓的代码重用攻击(Code-reuse Attack)。从最早的 Return-to-Libc 攻击[6,7]逐渐发展出了 ROP(Return Oriented Programming) [8,9]、JOP(Jump Oriented Programming)[10]等攻击方式。这些攻击方式依然能够达到与传统的代码注入攻击相同的攻击效果, 能够使攻击者执行任意代码。因此目前对于计算机系统安全性的保护依然不容乐观, 研究界对于代码重用攻击的防御提出了各种各样的安全机制, 然而都存在安全性或性能、兼容性缺陷, 无法达到令人满意的效果。本文对代码重用攻击的实现机制与主要方法进行了概述, 并对针对代码重用攻击的防御机制的实现与存在的问题进行了详细的介绍。在分析了代码重用攻击的根本原因基础上提出了一种新的针对代码重用攻击的防御机制设计思想。

本文结构如下: 第二部分对代码重用攻击的实现机制与国内外研究界当前对于代码重用攻击的研究现状进行了概述; 第三部分和第四部分分别对当前主流的防御机制两大设计思想进行了综述; 第五部分对这些防御机制进行了总结并提出了一种全新的能够提高计算机系统安全性的安全机制设计思想; 第六部分对全文进行总结。

2 代码重用攻击概述与国内外研究现状

代码注入与代码重用都属于对计算机系统漏洞进行利用的攻击方式。他们的区别在于攻击者执行恶意代码的方式, 前者通过直接向内存中注入恶意代码并执行; 后者利用内存中原有的代码, 通过篡改控制指令执行的数据, 改变了指令原有的执行顺序, 使其按照攻击者的意志执行, 达成攻击目标。现有的安全机制, 例如数据不可执行(DEP), 对代码注入攻击起到了极大的抑制作用。因此, 代码重用攻击得到了攻击者的青睐, 成为目前攻击方式的主流, 它能够绕过现行计算机系统的大部分安全机制。目前代码重用攻击有 Return-to-Libc、ROP、JOP 等具体形式。

2.1 Return-to-Libc 攻击

若攻击者能够获悉程序所使用的库函数的地址, 便可以用其地址来覆盖程序中某个函数的返回值, 并将调用库函数所需要的参数以正确的顺序添加至覆盖区内, 这样程序在该函数返回后就会执行相应的库函数, 例如调出一个 shell。这种攻击方式能使攻击者可以使用任意的参数调用内存中任意的库函数[11,12]。

虽然 Return-to-Libc 攻击能够达成上述的效果, 但在使用上还是有局限性, 不属于图灵完备的攻击方式[9], 原因有两点: 一是在 Return-to-Libc 攻击中, 攻击者能够一个接一个调用任意的库函数, 然而这仍然只能允许他执行原有的线性代码, 无法满足进行任意行为的需求; 二是攻击者只能够调用已经加载到内存中的库函数, 它们功能有限, 因此限制了攻击者的能力。

然而也有证明 Return-to-Libc 攻击的图灵完备性的成果[13], 通过传统 Return-to-Libc 中利用的函数产生的副作用完成特定的操作。找出那些副作用可用的函数来作为执行某项功能的函数配件。再利用与下文讲述的 ROP 攻击相同的原理将这些函数串联起来, 完成图灵完备的攻击。

2.2 ROP(Return-Oriented Programmi ng)攻击

OP 是在 Return-to-Libc 攻击的基础上逐步发展起来的, 能够实现任意程序行为(图灵完备)的攻击方式[9]。ROP 攻击与函数调用和返回机制有极大的联系。以 x86 架构程序为例, call 与 ret 指令在程序的执行过程中总是一一对应的。在执行 call 指令时, CPU 会将 call 指令的下一条指令地址压栈作为返回地址, 然后跳转到 call 所指示的位置。在 ret 指令执行时, CPU 会自动将预先保存在栈中的返回地址弹给 eip

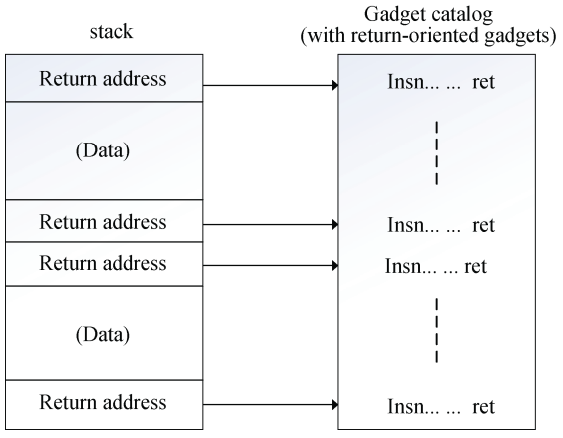


图 1 ROP 攻击模型

寄存器, 继续执行原有 call 指令之后的指令。这个操作不会对 call 的下一条指令的正确性进行检查或保障。ROP 攻击就是利用了这种缺陷。如图一所示, 首先在原有代码中搜索一些短小的指令序列, 这些序列都能完成一定的功能, 例如运算或赋值, 然后以 ret 指令为结尾, 称为配件(gadget)。攻击者依据要执行的配件的顺序与所需的参数, 将这些配件的地址与参数进行拼接, 构建一条配件链(gadget chain)。并通过栈溢出漏洞将这条配件链注入到栈中覆盖当前或其他某个函数的返回地址。一旦返回地址被覆盖的函数返回, 那么 CPU 就会按照栈中攻击者存放的配件地址链进行跳转, 不断的从一个配件返回并跳转到下一个配件执行[14]。ROP 攻击极为灵活, 在发展演变的过程中出现了许多变种与新的实现方式, 并配合内存泄露攻击, 能够绕过多种专门针对它的防御机制[15-17]。

2.3　JOP(Jump-Oriented Programming)攻击

JOP 攻击采用与 ROP 类似的配件链来实现攻击, 但不依赖栈完成对程序流的控制[10]。如图二所示, JOP 采用以 jmp 指令为结尾的配件。将配件地址链保存在另外一块任意数据区中, 称为调度表(dispatch table), 采用了一个专门的配件作为一个指向调度表的指针, 称为调度配件(dispatcher), 类似于普通程序的 eip 指针, 它对某个寄存器进行已知操作, 例如自增, 然后跳转到这个寄存器指示的地址。其他所有的能够完成特定功能的配件在最后都跳转到调度配件。而在 ROP 攻击中, 配件执行的控制序列是以配件地址链的形式存储在栈上的, 利用栈指针来指向配件。与 ROP 相同, JOP 攻击的实现也是需要指令序列的, 这些序列就是配件的地址, 但由于采用了调度配件, 配件地址序列可以存放在内存的任何可读写区域, 而不是必须在栈上。这就给了攻击者更大的灵活性和空间。他们可以将配件链放置于内存中任意一块可写的数据区。

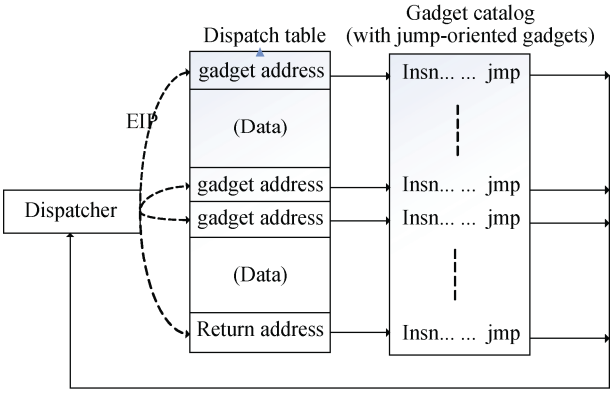


图 2　JOP 攻击模型

2.4　其他类型代码重用攻击

除了上述三种常见的代码重用攻击之外, 有研究者和黑客也相继提出了其他类型的代码重用攻击。

SROP[18]利用类 UNIX 操作系统用进程栈保存信号帧的机制, 通过伪造信号帧来引导进程进入到攻击者设定的代码区域中执行实现攻击目的。

COOP[19]利用了 C++面向对象编程的特性, 利用篡改类中的虚函数表来使一系列虚函数按照攻击者设定的顺序执行完成攻击目标。

2.5　国内外研究现状

由于代码重用攻击具有极强的破坏性, 极高的灵活性, 能够绕过现行的绝大多数防御机制。已经渐渐成为黑客进行攻击的主要手段。研究界对于代码重用攻击防御机制的研究也得到极大的重视。

在 2011-2015 年间 S&P (IEEE Symposium on Security and Privacy)、CCS(ACM Conference on Computer and Communications Security)以及 Usenix (Usenix Security Symposium)安全方面世界三大顶级会议对于代码重用攻击的相关论文发表数量如图三所示, 可以看出, 对于这类攻击的研究与防范逐渐成为研究界的重点。

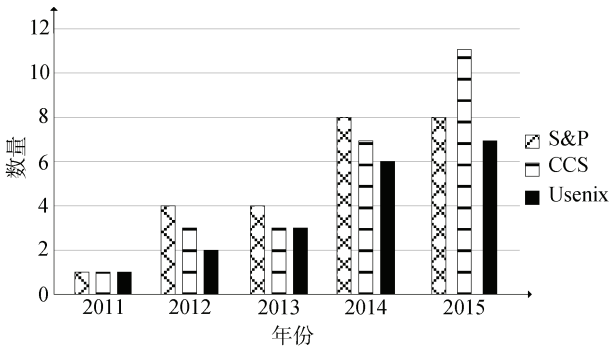


图 3　信息安全三大顶级会议历年相关文献数目统计

代码重用攻击的实现需要两大重要环节, 一个是控制流的劫持, 另外一个是对内存中的代码位置的获悉。首先得到内存中的代码位置, 构建控制程序执行的配件地址链, 而后通过对溢出漏洞将配件链注入到内存的数据区中, 最后劫持控制流使得程序按照攻击者注入的配件顺序执行。因此防御措施也就分别针对这两个重要环节, 监测程序的异常行为与增加程序的不可知性。

攻击者在实行代码重用攻击时, 要实现恶意的功能, 势必改变原有程序的执行路线, 因此研究者通过添加一些监控机制对这种改变做出侦测用以判断代码重用攻击的发生。

实现代码重用攻击的另外一个要素是对内存中

程序指令位置的精确知晓, 由于攻击者已经无法注入自己的代码, 因此只能使用内存中原有的代码, 他们通过选择配件, 并精心构造配件地址链用来控制这些配件的执行实现自己的恶意目的。因此研究者通过增加内存的不可知性来阻止攻击者得知内存中代码的布局, 使其无法构造配件链。

3 基于程序异常行为监测的防御机制

目前的针对程序异常行为的监测防御机制中, 大致分为两种方法, 一种是对 ROP/JOP 攻击中配件出现频率的监测; 另一种是对程序控制流正确性的保障与监测。

3.1 对于配件出现频率的检测

研究者在观察代码重用攻击(这里以 ROP 攻击为例)发生时的程序行为时发现, 发生 ROP 攻击的程序行为与正常程序行为有着明显不同。在 ROP 攻击的过程中, 攻击者通过将许多以 ret 指令为结尾的配件连接起来完成攻击, 这些配件大都较短, 在一定长度以内, 当 ROP 攻击发生时程序指令流中会出现大量 ret 指令。因此监测一段指令当中 ret 指令出现的频率, 就成为了一种监测 ROP 攻击的方式。然而这种方式只能对 ROP 攻击产生效果, 无法防御 Return-to-Libc 攻击。

Pappas 等人[20]利用 Intel 处理器中自带的 LBR (Last Branch Recording)机制来对程序中的间接跳转进行检测, 保证所有的 ret 指令返回目标指令都位于一条 call 指令之后, 这种方式可以防御简单的 Return-to-Libc 与 ROP 攻击; 利用对 LBR 中存储的跳转指令的地址进行分析, 分析这些地址之间的紧密程度来判断是否出现了一串以 ret 为结尾的配件链, 从而判断是否出现了 ROP 攻击。Cheng[21]等人采用了相同的对配件链的检查思路, 并采取了对于间接跳转更为频繁和严格的检查。Pappas[20]等人与 Cheng[21]等人都采用了对配件的出现频率进行监控的方法, 通过比对遭到攻击的程序与正常程序之间的差别来监测攻击的发生。具体涉及到两个变量: 被认作是配件的指令序列长度 L_g 和被认作 ROP 攻击发生的配件连续出现的次数 L_c。通过改变这两个变量可以提高或降低 ROP 攻击检测的敏感程度。

这种方式存在的问题是无法确定检查的敏感度。假如提高敏感度, 那么有可能正常的程序也会被判定发生 ROP 攻击, 使假警报剧增; 假如降低敏感度, 则 ROP 攻击就有可能逃过监测。如何选择 Lg 和 Lc 对是需要解决的问题。更严重的是, 目前有些攻击者已经能够找到一些长配件, 这些长配件的长度大于被系统认作 ROP 配件的长度, 并把这些配件加入到实施 ROP 攻击的配件链中, 就可以完全逃过这种防御机制的监测[15,16,22]。

总体来说, 这种监测思想存在较大的绕过可能性, 无法有效满足目前的安全需求。

3.2 基于保障控制流完整性思想的防御机制

类比与对配件出现频率进行监测的方法, 保障间接转移指令目标地址的正确性是目前更为有效的方法, 这里需要指出的是, 转移指令分为直接转移和间接转移两大类, 直接转移指令的目的地址是在编译链接阶段就已经确定并写入二进制代码中的, 在程序运行时无法更改; 间接转移指令在运行时对具体的寄存器或内存进行间接寻址, 这些间接转移指令的目标可能是动态变化的, 因此存在被攻击者篡改并劫持的可能性, 故所有保障控制流完整性的机制都是针对间接转移指令的。

3.2.1 细粒度 CFI 的最早提出与探讨

文献[23]是最早提出保障控制流完整性(Control-Flow Integrity,CFI)思想的文章, 首先将要执行的代码进行静态分析, 画出控制流图, 预先计算出所有的间接转移指令的可能的目标地址, 并为之配唯一的 ID。改写程序的二进制代码, 在每一条间接跳转指令执行之前增加检查逻辑, 一旦出现违反控制流图的间接跳转就会报错并终止程序。这种为每一个转移目标地址分配唯一 ID 的实现思想我们称之为细粒度 CFI。它在被提出时被认为具有极佳的安全性, 能够检测出任何代码重用攻击并使用纯软件方式实现, 得到了研究界广泛的重视与探讨, 并出现了不少具体的实现方案。

XFI[24]利用 CFI 思想保证控制流的正确性, 并结合了其他例如内存访问控制, 影子栈等机制, 将传统的应用程序包装成 XFI 模块, 在运行时提供保护与监控, 并通过设置外部接口与操作系统交互。然而所有的应用程序都被封装成块, 则被设计用于共享的动态链接库就无法使用, 导致运行时内存有一定的空间损耗。

Hypersafe[25]是最早提出的对虚拟机监控器进行 CFI 保护的安全机制。其对监控器代码进行分析, 对所有代码指针的可能调用位置进行计算, 并维持指针索引表, 在每次指针解引用时查表, 若出现非法目标地址就会报错。对返回地址采用类似的方法。但是只能对直接运行在硬件上的 1 类监控器进行保护, 在运行与宿主操作系统之上的 2 类监控器上无法实现。

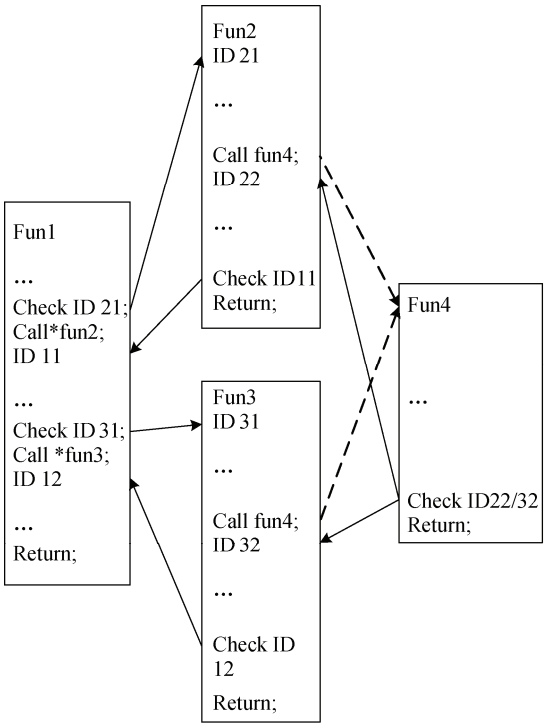


图 4　细粒度 CFI 实现简图

文献[26]是最早提出的基于智能手机平台上的CFI实现方案, 针对苹果公司的面向对象C语言并且应用程序加密存储。对程序的二进制代码进行解密, 反汇编, 绘制控制流图再对二进制代码进行改写。而文献[27]是属于对其的改进, 在编译器阶段进行 CFI 设置, 提高了应用程序的灵活性。

上述文章都属于对传统 CFI 思想在不同平台上的实现与应用, 没有本质上的改进与创新。

在提出了近10年以后, CFI思想才在商业领域得到了初步的应用, 微软公司提出了 CFG (Control-Flow Guard)技术, 通过编译器在编译阶段对代码的分析。在程序运行时的每一个间接调用进行检查, 看目标地址是否是一个函数的开头[28]。这种方式不属于传统细粒度 CFI 思想并且对间接跳转与返回指令没有保护, 存在被绕过的可能。

细粒度 CFI 在提出之后之所以一直没有得到广泛的应用, 是由于其本身还存在不少问题, 比如:

1. 针对诸如内核程序上百万行的代码, 以及其他的大规模的程序, 如何精确的画出其控制流图, 仍然是一个亟待解决的问题。

2. 传统的细粒度 CFI 要求在程序载入内存之前需要对所有的程序模块进行分析并重写二进制代码加入检查指令, 包括动态链接库, 那么每个程序就只能在载入时带着它们自己重写好后的库了, 无法与其他的程序共享, 这与动态链接库的设计初衷相悖[29]。

3. 纯软件的 CFI 实现方式性能损耗较高。在20%至50%左右[23]。

针对细粒度 CFI 存在的这些性能与实现上缺陷, 研究界也进行了许多探讨与改进。

文献[29]针对经典 CFI[23]对动态链接库无法支持的缺陷, 将 ID 与地址绑定存在另外一块内存区中。建立两个表, 一个映射间接转移指令地址与其ID, 称为分支表; 另一个表映射各个ID的转移指令的合法目标地址, 称为目标表。对每一个间接转移指令, 分别查两个表, 根据指令自身地址在分支表中查其ID, 再在目标表中根据转移指令的目标地址查 ID, 若匹配, 说明通过。若不匹配则报错。经典 CFI 要求一次性画出全部程序的 CFG, 但是该方法将程序分为多个模块, 分别为每个模块画出 CFG, 在加载多个模块运行时, 就会将各个模块的CFG合并。这种运用ID表的方式方便动态的添加与维护。能够较好的解决传统二进制代码改写的无法兼容共享库的缺陷。

文献[30,31]针对传统 CFI 使用纯软件实现导致损耗高的缺点, 使硬件层次对上层的 CFI 机制进行支持, 添加了关于 CFI 检查的指令以替代传统的软件实现, 能够大大缩减代码指令长度并减小了性能损耗。但这种方式会使兼容性降低, 没有进行硬件修改的硬件无法对上层代码进行 CFI 支持。

文献[32]将沙箱处理的相关技术与CFI进行组合, 对 CFI 繁琐的检查机制进行整合简化, 提高了 CFI 检索的速度, 降低了性能损耗。

文献[33]将CFI在编译器层次上实现, 对C++中的虚函数表进行保护, 采用静态分析, 对虚函数表进行分析, 对目标码进行修改, 在调用虚函数之前加入一个验证函数进行确认。该方法在一定程度上提高了 CFI 的兼容性并减少了性能损耗。但是仅仅能够对通过函数指针进行的函数调用进行保护, 无法对运行时的栈、堆等数据结构进行保护。

3.2.2　粗粒度 CFI 的提出与否定

针对经典的细粒度 CFI 实现较困难, 损耗较高的缺陷。学界提出了一类简化版的 CFI 实现方案。它们通过牺牲一定安全性来达到可以接受的性能损耗与实现难度。与先前的细粒度CFI相比, 这些实现方案都不需要画出程序的控制流图, 可以称之为粗粒度CFI。

CCFIR[34]重写目标码, 收集所有合法的跳转指令目标地址并将其以随机序存储于一块独立的安全内存中, 强制所有的间接跳转指令只能通过这个内存区域完成控制流的转移。CFIMon[35]通过静态分析得出合法的跳转地址, 并利用处理器中的分支追踪存

储机制(LBR 等)来实时分析控制流的完整性。文献[36]将间接转移指令替换为到一个外部处理程序的直接跳转, 并维持一个合法跳转表, 每次间接跳转, 处理程序会查这个表, 只有在表中的目标地址才能作为这个间接跳转的目的地。文献[37]首次将 CFI 思想在操作系统内核上实现。改写操作系统代码, 加入检查逻辑。文献[38]采用静态分析与动态监测相结合的方式检查间接跳转指令的目标是否位于合法函数区间。

上述几种方法虽然实现方法各异, 但是他们都有一个共同的特点, 就是不需要画出程序详细的控制流图并且对于间接跳转目的地址不需要分配唯一 ID。仅仅是对程序进行静态分析, 根据一定的规则找到合法的跳转地址, 例如, 所有的 ret 指令的合法目标都是一条 call 指令下一条指令; 所有的间接调用目标地址都必须是一个函数的开头。这就导致任意间接转移指令与他们可能的目标地址之间不再遵从逻辑上的一对一关系, 而是一对多。一条 ret 指令可以返回到任意一个 call 指令的下一条指令, 而不必是原本的调用者; 一条 call 指令可以进入任意一个函数的开头, 而不用根据逻辑进行判定究竟应当跳转到哪一个函数的开头。

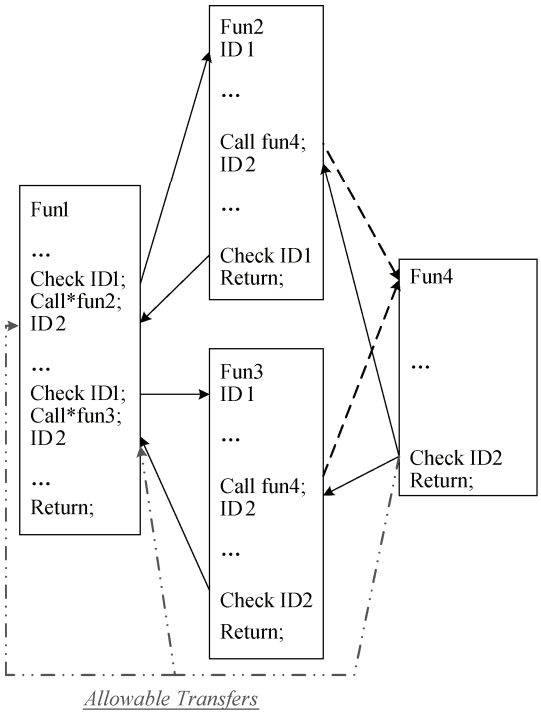


图 5 粗粒度 CFI 的缺陷

粗粒度 CFI 这种对间接跳转目标地址不加以再区分的管理方式, 导致了攻击者能够通过精心设计的配件, 组合配件链, 能够做到继续实施代码重用攻击[16,39]。具体使用的配件特点如下: 一种配件以一条 call 指令的下一条指令为起始, 以 ret 指令为结尾, 这种配件可以绕过粗理度 CFI 对 ret 指令必须调到调用点的检查机制; 另一种配件是以一个函数开头为起始, 以一个间接跳转指令为结束的配件, 这种配件可以绕过粗粒度 CFI 对间接调用指令必须调到函数开头的检查机制。攻击者将这两种配件搭配使用, 就可以完全绕开粗粒度 CFI 的检查完成攻击。因此, 粗粒度 CFI 也被证明无法满足安全需求而被研究界淘汰。

3.2.3 细粒度 CFI 安全性弱点的提出与改进

随着研究界将研究的重点重新转回细粒度 CFI, 研究者又发现了细粒度 CFI 中存在的安全漏洞。

“控制流弯曲”[40]利用细粒度 CFI 仅仅保障所有跳转都按照事先通过静态分析确定的控制流图进行, 不会对进程的上下文语义进行分析的缺陷。挑选了一些经常使用的并且会对内存中的数据产生修改的函数(例如 printf()、memcpy()等)通过控制这些函数的参数对内存中的数据进行修改, 完成了不逾越控制流图的控制流劫持, 完成攻击者所需要的功能。

传统细粒度 CFI 的另一个缺陷是无法对系统产生的信号或中断的返回进行保护, 在类 UNIX 操作系统中, 这些信号或中断发生时, 进程原状态大多数是保存在栈上的。SROP[36]利用这种机制, 通过伪造信号帧来并篡改栈上保存的现场状态引导进程进入到攻击者设定的代码区域中执行实现攻击的目的。文献[41]与[42]也针对细粒度 CFI 在实现上的漏洞提出了绕过的方法。

针对细粒度 CFI 存在的安全缺陷, 研究者也对其进行了相关的改进。

文献[43]对所有能够改变程序控制流的对象进行认证(包括函数返回地址, 函数指针等), 在每次这些对象从内存中存储的时候计算一个验证值, 将其存于一个寄存器中, 每次这些对象被读取时检查这个值是否与先前相同。

文献[44]提出了上下文相关的 CFI, 利用 intel CPU 中的 LBR(Last Branch Record)机制, 每当程序要执行重要的系统调用时(例如 exec()等), 则对 LBR 中保存的先前的跳转指令进行 CFI 判断, 若全部符合, 则继续执行, 若有不符合 CFG 的跳转则会报错停止执行。这种方法能够解决传统细粒度 CFI 无法对上下文的跳转合法性进行判断的缺陷。

总体来说, 粗粒度 CFI 安全性不足不能满足实际需求。细粒度 CFI 机制尚未完全成熟, 仍然存在安全与性能问题需要克服, 若能达到低耗, 高兼容性实现, 将会产生较好的保护效果。

4 基于增加程序不可知性思想的防御机制

由于代码重用攻击的实现要依赖对内存中已有指令的使用。攻击者需要知道这些指令的确切位置。因此通过增加程序不可知性的方法也可以达到阻止攻击的发生, 使用随机化是一个典型并有效的方法。

4.1 随机化安全机制

ASLR(Address Space Layout Randomization)[45]是一个典型且已经得到广泛应用的基于随机化的防御机制。其主要思想是通过对进程中的代码段、数据段、堆、栈所占的页面进行随机化排布来使攻击者无法得知这些段的具体位置从而无法实施攻击。这种方式与数据段不可执行机制(DEP)相结合, 起到了很好的防御效果, 不仅能够对传统的注入攻击产生防御效果, 也能提高代码重用攻击实施的难度。

但是 ASLR 仍然存在不足与缺陷, 具体有三点:

1. 随机化的粒度仅仅到页一级, 页内的数据仍然是顺序存储的, 且参与随机化的内存地址位数不足, 能够被暴力攻破[46,47];

2. 由于动态链接库的共享机制, 其位置对所有的程序都是可知的, 这就导致了这些代码无法参与细粒度随机化处理。防御机制设计者只能选择放弃动态库的共享或放弃对这部分代码的随机化处理[48];

3. ASLR 面临内存泄露攻击的威胁, 由于程序代码段一般都会占用多个页面, 页面之间都有跳转指令维持其之间的联系。攻击者若能够通过悬空指针等漏洞对任意内存地址数据进行读取, 可以找到页面之间的相互联系, 就可以达到知晓代码段页面的分布位置, 实现去随机化[49]。

由于 ASLR 极为广泛的使用和极为突出的弱点与缺陷, 目前大多数的研究都是着眼于对 ASLR 的改进或以 ASLR 为标尺。在 ASLR 的基础上又产生了许多新颖的内存随机化实现机制。

文献[50]针对 ASLR 粒度较粗的缺点, 对代码的放置顺序进行随机排布, 粒度达到了单个指令级。建立一个记录代码顺序的表, 对每一条指令, 记录其后继指令。

文献[51]在不影响程序运行正确性的情况下, 对指令进行小范围的随机化处理, 能够做到消除无意识配件, 重排函数调用与返回的寄存器保存和弹出指令。能够在一定程度上减少攻击者可以使用的配件数量。但是这种方法的覆盖不够广泛, 无法完全消除配件。

文献[52]对二进制代码中的代码段复制为两份, 将原有的代码段当做纯数据, 不可执行, 对复制的部分进行分块, 并在此程序装载时随机化, 并运行, 使用相关的算法保证代码段中的转移指令的正确性。以此实现内存布局的细粒度随机化。

文献[53]将进程的所有代码数据段分割成任意大小的块, 然后将这些块随机排布的整个进程地址空间。利用二进制代码重写技术进行重新汇编, 保持控制流指令正确。

文献[48]对于 ASLR 的兼容性问题, 使共享代码实现随机化。首先重写目标代码, 利用一个间接查找表使控制流转移重定向, 使得所有跳转目标都会通过这个表来查找。利用 x86 的分段机制保护间接查找表, 使用特定的段选择寄存器 FS, 只有修改过的合法跳转指令才能找到间接查找表。但这种方式仅仅能够在 x86-32 架构上实现, 并且无法防御内存泄露的攻击[54]。

文献[55]与[56]分别在函数段与程序块的粒度对内存中代码进行随机化分布, 增加了参与随机化的内存地址位数, 提高了随机化粒度。

不论实现方式如何, 上述通过随机化增加程序不可知性的安全机制, 都面临一个共同的威胁, 那就是内存泄露。内存泄露能够帮助攻击者获得配件的地址, 对代码重用攻击的实现起到了极大的帮助作用。代码重用加内存泄露是目前最具威胁性的攻击方式。

4.2 内存泄露的威胁与应对

传统的 DEP 等权限划分的安全机制仅仅是将内存页面的写操作与执行操作分离开来。无论代码段还是数据段都是默认可读的, 这就导致了攻击者通过悬空指针等程序错误对内存页面进行读取、扫描。在代码段或数据段中得到程序随机化的分布布局, 从而绕过 ASLR 等随机化安全机制, 进行攻击。

文献[49]能够使用内存泄露漏洞, 实时对随机化布局之后的程序代码段进行扫描, 获悉每个代码页确切的起始地址, 计算出可以用作配件的代码段的地址, 动态搭建配件链实施攻击。

内存泄露大致分为两种方式: 一是直接泄漏: 读取代码段, 找到直接跳转和直接调用的目标地址, 收集这些地址进行分析, 就可以得出代码页面的分布位置; 二是间接泄漏: 读取数据段中的函数指针(例如虚函数表), 返回地址可以达到相同的目的[57]。

传统的写/执行权限分离思想无法防御内存泄露的攻击, 内存泄露又会导致随机化安全机制的失效。因此防御针对内存泄露也成为了学界研究的重点。针对这种威胁, 最初的防御思想是分离读权限。但是

针对目前的硬件结构来说, CPU 无法对页面的读权限进行剥离, 需要通过上层的操作系统进行支持, 并且传统的代码段中是包含数据信息的, 如何对代码段中的数据进行剥离也是实现的难点之一。

文献[58]将读与执行权限分离。通过改写 MMU 中的页错误处理机制实现。假如进程对内存空间中代码区中一个不存在的页面进行取指操作, 则说明属于正常的缺页异常, MMU 会从硬盘中载入这个页并将页存在相关位置。但如果对这个代码页进行的不是取指操作, 则说明是对代码页的读取, 属非法, 会导致进程停止。能够对直接泄露产生防御效果。然而这种防御机制无法对间接泄露产生防御效果, 无法保护数据段中包含代码段地址的函数指针等不会被泄露。

文献[57]将代码段与数据段严格区分, 确保代码段只可执行而不再可读。将代码段中的数据进行修改为只执行(例如 switch 转换表, 之前的是跳转的地址, 文章改成跳转到该地址的指令。)实现代码只可执行可以防止直接内存泄露。再将数据段中的函数指针和返回地址都改为代码段中的一个中转站, 这个中转站通过指令的形式跳转到对应的目标地址, 通过这种方式防御间接内存泄露。这种防御方式在目前来说能够对内存泄露起到有效的防御效果。

由于对内存代码段中的数据与代码强行进行分离的方法实现过于困难, 有研究者通过对代码段进行复制并添加不同的方式实现对内存泄露攻击的防御。

文献[59]将可执行的代码段复制作为数据段。当有对代码段中的数据进行读取操作时, 就将这段数据用随机数代替, 并对 MMU 进行重定向操作, 使之指向复制的数据段。将对数据和代码的操作分割在不同的区域, 在可执行区的数据在读取后将不会再与原来相同。

文献[54]先对二进制代码进行修改, 将其复制为两份, 一份是原有状态, 一份是经过随机化排布的, 在每个函数的调用和返回阶段, 随机选取运行哪一份的代码, 这样即使攻击者能够通过泄漏得到内存布局, 也无法使构造的 ROP 链得到执行。

文献[60]将程序段分为两个大段, 代码段和数据段。AG 作为中间的转接层。将程序中所有的能够控制代码执行的位置称为代码地址, 例如基地址, GOT、PLT 的入口、栈中的返回地址, 跳转表等。每当有代码地址将向数据区泄露时, AG 对其进行加密作为一个标识, 每当有标识被用于控制流跳转时, AG 对其进行解密。对栈上的数据进行了保护, 将函数返回地址等数据保存在原有的位置。将栈上的其他数据用另一个寄存器指示。

文献[61]与文献[58]类似, 利用 MMU 通过软件实现对代码可读属性与可执行属性的分离。利用页错误处理程序区分程序的非法读操作与正常的指令获取(Instruction Fetch)。

总结来说, 基于随机化的安全机制在确保内存不被泄露并且随机熵足够高的情况下能够起到较好的防御效果。

5 思考与讨论

本小节对现有的对于代码重用攻击的防御方式进行系统化的总结, 并提出了一种能够从根本上提升对于代码重用攻击防御效果的设计思路。

对于计算机系统的攻击与防御是一对不断上升与交错的矛盾双方: 对于计算机系统的攻击催生着防御方式的改进; 防御方式的改进有促使黑客或研究者开发与使用新的更有效的攻击方式。

而在对于计算机防御机制的设计与实现方面。性能与安全又成为了一对矛盾的两个方面。要提高系统的安全性必然伴随着性能或兼容性的损耗。能够平衡这一对矛盾, 在可接受的性能与兼容性损耗限度内达到令人满意的安全效果的防御机制才得以广泛的采用。DEP 与 ASLR 可以成为这种防御机制的典型, 它们二者组合使用可以做到在较低的损耗的情形大幅度提高攻击者成功实施攻击的难度。

但这两种防御机制在代码重用攻击与内存泄露攻击的双重威胁之下其安全效能已经不再适用。DEP 仅仅能够防御计算机系统不受代码注入的攻击, 无法对代码重用攻击产生防御效果; ASLR 本身具有诸多弱点并且面临内存泄露的威胁。研究界对于这种新兴的攻击方式开始了不懈地探索与研究。

5.1 程序行为监控

针对代码重用攻击的防御, 思路上最简单的方法是对于程序运行时行为的监测, 有效实施这种防御机制的难点与关键在于快速区别正常运行的程序与遭到攻击的程序之间运行状态的差异。

早期的研究者对指令流中出现配件的频率进行监控。这种方法能够对最原始的 ROP 攻击产生较好的防御效果, 但其本身仍然存在问题, 严格地监控会导致大量虚假警报, 而降低敏感度会使攻击行为被忽视。且攻击者已经开始使用超过被认作的配件的长指令段以此稀释配件出现的频率将其绕过。

对于配件出现频率的监控无法进行, 研究者开始转而控制进程跳转行为的合法性, 传统细粒度 CFI

越来越得到重视。这种机制安全性较好高, 但存在先天性的性能损耗高, 无法使用动态链接库等缺陷。于是研究者开始研究能够实现的 CFI 方案, 出现了粗粒度 CFI 实现方案。

粗粒度 CFI 不需要事先精确分析程序的控制流, 采用了较为宽松的 CFI 检查机制, 得到了性能上的提升, 但是其检查机制过于宽松, 无法保障函数在返回时跳转到的是原先的调用点, 也无法保障函数调用时, 所进入的函数体就是原本应当调用的函数, 攻击者就可以利用精心设计的配件加以绕过。

在粗粒度的 CFI 实现方案被否定以后, 研究界继续细粒度 CFI 的研究。对于细粒度 CFI 安全性的质疑被提出, 难点依然无法解决: 无法准确判断某个跳转属于程序正常的操作还是被攻击者劫持, 有攻击者能够实现不违反控制流图的攻击; 由于目前操作系统自身的代码没有经过 CFI 强化, 对于从操作系统到用户代码的转移没有检查, 因此攻击者可以通过伪造系统调用或信号帧来实现对控制流的劫持。

针对程序行为监控这种安全机制的基本思想是使程序按照一定的规则运行, 一旦检测出程序有违反规则的行为则认作攻击发生。两个关键点分别是监控的实施与规则的设定。有些攻击者通过安全机制的监控盲区进行攻击[18]; 有些攻击者则利用规则的漏洞进行攻击[40]。因此, 能否全面地对程序的行为进行监控与能否设定安全高效的规则, 是决定安全机制有效与否的关键。而由于计算机程序的庞杂性、程序运行时的状态多样性, 使得这种监控的实施与规则的设定具有很高的挑战性。

5.2 增加程序的不可知性

代码重用攻击的另一个防御思想是通过增加程序不可知性使攻击者无法得知程序代码的布置方式从而无法构建配件地址链。主要的实现方式是进行随机化排布。然而在进行随机化的同时还应当保证程序自身运行的正确性。这个过程可以成为去随机化(Derandomization), 有些安全机制例如 ASLR[45]通过生成地址无关代码与程序装载时的随机化分布来自动实现随机化程序的逆随机化; 有些需要一定的内存占用与时间开销来完成随机化代码的去随机化例如 ILR[50]使用额外的指令流程图控制每条指令的执行。攻击者可以利用诸如内存泄露等方式获得 ASLR 随机化后的内存布局或是 ILR 的指令流程图, 进而实现去随机化, 找到要用的配件链完成代码重用攻击。

评价一个随机化安全机制有两个关键指标: 程序正常去随机化的难度以及攻击者掌握去随机化方法的难度。一个好的随机化安全机制必然是程序自身能够实现简便的去随机化过程, 而攻击者很难进行去随机化。

对于内存泄露的防御也得到了研究者的重视, 主要的防御思想是分离并限制读权限, 例如使代码段不可读, 仅可执行等。但是传统的程序代码段是包含数据的, 如何将代码段中的数据剥离也是等待解决的问题。除此之外也有对代码段进行冗余化并采取原份与备份不同处理的方式。

总体来说, 本文认为能够克服诸多限制的细粒度 CFI, 以及能够有效防范内存泄露的随机化机制可以较好地防御代码重用攻击。

5.3 代码重用攻击根本原因分析与安全机制设计思路

代码重用攻击的一个重要的原因是控制流被劫持, 许多防御机制针对如何保护控制流的完整性, 致力于发现并制止控制流的被劫持。但这种防御措施仅仅是被动地检测, 实际的根本问题是控制流为什么会被劫持。从计算机系统结构级考虑, 发生传统缓冲区溢出攻击的根本原因是在程序的执行过程中, 对程序的代码段和数据段采取了相同的权限管理, 所有的页面都是可读可写并可执行的。这就导致了攻击者利用软件漏洞将自己要执行的恶意代码以数据形式注入到内存中, 并通过修改控制流使得这些恶意代码以代码的形式得到了执行。对于数据与代码权限与处理方式的混淆导致了这种情况的发生。研究界认识到了这种问题, 并提出了数据不可知性思想, 即一段内存页不能既可修改, 又可执行。并采取相应的实际防御机制, 即 DEP, 使得这个问题得到了很好的解决。

然而攻击仍在发生, 只是攻击者不再将恶意代码以数据的形式注入到内存中去, 而是利用内存中原有的代码, 通过控制它们执行的顺序来达到自己的目的, 也就是所谓的代码重用攻击。数据与指令具有极大的关联性, 真正控制指令执行的还是数据。这些控制指令执行的数据在内存中是以地址或指针的形式存储的。这些数据与传统的整数、浮点数等数据类型都被保存在内存的数据区中。那么类比针对传统注入攻击的分类并采取权限分离的防御思想, 我们可以对内存中的内容进行再次分类。具体可以分为普通数据, 例如传统的 char、int 类型数据等; 普通指令, 即原有意义上的指令; 数据地址, 例如指向普通数据的指针, 数组等; 指令地址, 例如函数指针, 函数的返回地址、基地址、虚函数表等。那么如此分类之后, 代码重用攻击的根本原因也就显而易见

了——指令地址被篡改。然而攻击者能够注入自己数据的地方仅仅是在内存的普通数据段与数据地址段中, 但是由于这些数据段与指令地址在内存中都是按照传统数据这种方式进行无差别存储与管理的, 它们之间没有逻辑上或空间上的分隔。这就给了攻击者篡改指令地址的机会, 导致了代码重用攻击的发生。

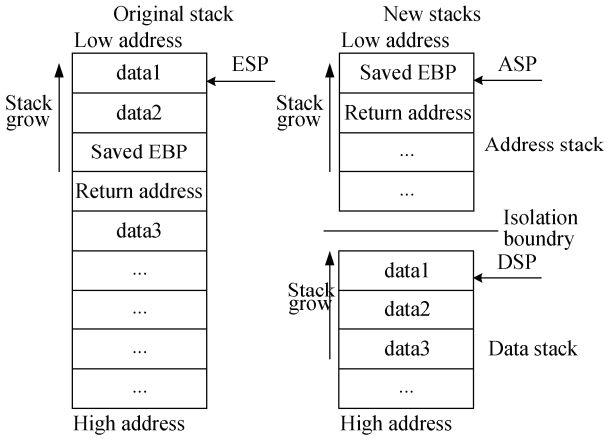


图 6　新的栈结构示意图

依据这种宏观思想, 本文提出了一种初步的设计思路。如图六所示, 对程序运行时栈的结构进行重构, 将原有的栈一分为二: 一个用于保存压入的普通数据, 称之为数据栈; 一个用于保存所有的函数返回地址和基地址, 称之为地址栈。为两个栈分别维护一套栈顶指针与栈帧指针。将地址栈的权限设置为只有函数的调用与返回阶段才会进行压栈与弹栈操作。将地址栈的位置置于数据栈的地址减小方向, 这样即使攻击者利用漏洞对栈中的缓冲区进行溢出, 也不会覆盖到地址栈中函数的返回地址。

为了初步实现这种设计思路, 本文给出两种方案: 第一种在硬件上实现, 增加两条机器指令分别对新的数据栈进行压栈与弹栈操作, 从原有的寄存器中保留两个用于保存数据栈的栈顶地址与栈帧地址, 将原有的压栈与弹栈指令改为对新的数据栈进行的操作, 原有的栈作为地址栈。或是对 call 指令与 ret 指令进行修改, 将压入原有栈中的返回地址压入新的地址栈中, 原有的栈作为数据栈; 第二种是用软件实现, 与文献[60]中的 AG-Stack 类似, 由编译器用 mov 与 sub(add)指令组合来模拟对新开辟的数据栈的压(弹)栈操作, 原有的栈作为地址栈。采用软件实现的方式与硬件实现相比, 能够产生较少的兼容性损耗和较高的性能损耗。硬件实现需要对底层的指令架构进行修改, 兼容性变差, 但是产生的性能损耗较低。这样将栈结构分开并处以不同的处理方式, 可以有效避免攻击者利用漏洞对保存在栈中的函数返回地址等指向指令的数据进行篡改, 从而大幅增加了实现代码重用攻击的难度。

本小节提出的设计思路仅仅是一种构想, 具体实现的细节仍然有待完善。

导致计算机系统不安全的原因是没有相应权限的访问与修改: 攻击者能够读取本来不应该被读取的部分, 导致了内存泄露攻击; 通过在普通数据区的溢出修改了控制程序运行的指令地址段, 导致了代码重用攻击。这些都是权限划分不清, 处理方法模糊造成的安全隐患。那么针对这些问题就可以通过权限与处理方式再细化的思想予以解决。假如通过安全机制的设计能够实现普通数据与指令地址在逻辑上或空间上隔离, 将有效的阻止代码重用攻击的发生。

6　总结

本文介绍了代码重用攻击的起源与三种主要的实现方式。系统化的阐述了目前应对这种攻击的两类防御思想: 针对程序行为的监控思想以及增加程序的不可知性的思想, 以及它们在研究与实现过程中遇到的困难与解决方式。其中, 对程序行为监控的思想又可以分为针对配件频率的监控与程序控制流完整性的监控; 增加程序不可知性的思想以 ASLR 为代表, 介绍了实现过程中遇到的问题以及能够使这种安全机制失效的内存泄露攻击, 并介绍了针对内存泄露攻击的防御机制。最后, 对这两类防御思想进行了深入且详细的分析, 并提出了一种新的能够提升对代码重用攻击防御效果的安全机制设计思想。

参考文献

[1] V. van der Veen , N. dutt-Sharma , L. Cavallaro , and H. Bos. “Memory errors: The past, the present, and the future,” in *Proc. The 15th International Symposium on Research in Attacks, Intrusions, and Defenses. (RAID’12)*, pp. 86-106, 2012.

[2] “Smashing the stack for fun and profit.” Aleph One. http://insecure.org/stf/smashstack.html, 2000.

[3] M. Russinovich. “Windows internals.” *Washington, DC, Microsoft*, 2009.

[4] V. De Ven Arjan. “New security enhancements in red hat enterprise linux v.3, update 3”. *Raleigh, North Carolina, USA: Red Hat*, 2004.

[5] Microsoft. “Data Execution Prevention (DEP).” http:// windows.microsoft.com/en-us/windows-vista/data-execution-prevention-frequently-asked-questions#, 2001.

[6] C0ntex. “Bypassing non-executable-stack during exploitation us-

ing return-to-libc", http://css.csail.mit.edu/6.858/2014/readings/return-to-libc.pdf, 2005.

[7] Nergal. "The advanced return-into-lib(c) exploits (pax case study)." *Phrack Magazine*, 58(4): 54. http://phrack.org/issues/58/4.html Dec.2001.

[8] T. Kornau "Return oriented programming for the ARM architecture." *Bochum:Ruhr-University* , 2009.

[9] H. Shacham. "The geometry of innocent flesh on the bone: Returninto-libc without function calls (on the x86)." In Proc. *the ACM Conference on Computer and Communications Security (CCS'07)*, pp. 552-561, 2007.

[10] T. Bletsch, X. Jiang, V.Freh, and Z. Liang. "Jump Oriented Programming:A New Class of Code-Reuse." in Proc. *the 6th ACM Symposium on Information, Computer and Communications Security(ASIACCS '11)*. pp. 30-40, 2011.

[11] S. El-Sherei, Return-to-libc,https://www.exploit-db.com/docs/28553. pdf.

[12] Wikipidia, "Return-to-Libc Attack." https://en.wikipedia.org/wiki/Return-to-libc_attack. Aug. 2015.

[13] M. Tran, M. Etheridge,X. Jiang, T. Bletsch, and N. Peng. "On the expressiveness of return-into-libc Attacks" in *Proc. the 14th Recent Advances in Intrusion Detection(RAID'11)* pp. 121-141, 2011.

[14] R. Roemer ,E. Buchanan ,H. Shacham,. "Return-Oriented Programming System,Languages, and Applications"*ACM Transactions on Information and System Security (TISSEC)-Special Issue on Computer and Communications Security* : Volume 15 Issue 1, March 2012.

[15] N. Carlini ,D. Wagner. "ROP is still dangerous: Breaking modern defenses" in *Proc. the 23rd USENIX Security Symposium. (Usenix'14)* pp. 385-399, 2014.

[16] L. Davi, A. Sadeghi, D. Lehmann. and F. Monrose, "Stitching the Gadgets On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection" in *Proc. of the 23rd USENIX Security Symposium.(Usenix'14)*, pp. 401-416, 2014.

[17] S. Checkoway, L. Davi, A. Dmitrienko, and H. Shacham. "Return-oriented programming without returns" in *Proc. the ACM Conference on Computer and Communications Security (CCS'10)*, pp. 559-572, 2010.

[18] R. Bosman, H. Bos. "Framing Signals:A Return to Portable Shellcode" in Proc. *the 2014 IEEE Symposium on Security and Privacy(SP'14)* ,pp. 243-258, 2014.

[19] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A. Sadeghi. and T. Holz. "Counterfeit Object-Oriented Programming: On the difficulty of Preventing Code Reuse Attacks in C++ Applications," in *Proc. the 2015 IEEE Symposium on Security and Privacy (SP'15)*. pp. 745-762, 2015

[20] V. Pappas, M. Polychronakis, and A. D. Keromytis. "Transparent ROP Exploit Mitigation using Indirect Branch Tracing," in *Proc. the 22nd USENIX Security Symposium*. *(Usenix'13)*, pp. 447-462, 2013.

[21] Y. Cheng,Z. Zhou, and M. Yu. "ROPecker: A generic and practical for defending against ROP attacks" in Proc. *the 2014 Network and Distributed System Security (NDSS'14)* Symposium. 2014

[22] E. Athanasopoulos,M. Polychronakis,H. Bos, "Size Does Matter: Why Using Gadget-Chain Length to Prevent Code-Reuse Attacks is Hard" in *Proc. of the 23rd USENIX Security Symposium.(Usenix'14)*, pp. 417-432, 2014

[23] M. Abadi, M. Budiu, J. Ligatti, and U. Erlingsson. "Control-Flow Integrity" in *Proc. the 12th ACM Conference on Computer and Communications Security(CCS'05),* pp.340-353, 2005.

[24] M. Abadi, M. Vrable,M. Budiu, and U. Erlingsson. "XFI Software Guards for System Address Spaces" in *Proc. the 7th Symposium on Operating System Design and Implementation*, *(OSDI'06)*, pp. 75-88, 2006.

[25] Z. Wang, X. Jiang. "HyperSafe A Lightweight Approach to Provide Lifetime Hypervisor Control Flow" in *Proc. the 2010 IEEE Symposium on Security and Privacy (SP'10)*. pp. 380-395 , 2010.

[26] L. Davi, R. Dmitrienko, M. Egele, T. Fischer, T. Holz, R Hund, S. Nurnberger, and A Sadeghi."*MoCFI*: A Framework to Mitigate Control-Flow Attacks on Smartphones" in *Proc. of the Network and Distributed System Security Symposium(NDSS'12)*. 2012.

[27] J. Pewny, T. Holz. "Control-flow Restrictor: Compiler-based CFI for iOS". In *Proc.* the *Annual Computer Security Applications Conference (ACSAC'13)*, pp. 309-318, 2013.

[28] J. Tang. "Exploring Control Flow Guard in Windows 10", http://sjc1-te-ftp.trendmicro.com/assets/wp/exploring-control-flow-guard-in-windows10.pdf, Jan. 2015.

[29] B. Niu,G. Tang. "Modular Control-Flow Integrity" in *Proc. the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation.(PLDI'14)*, pp. 577-587, 2014.

[30] L. Davi, P. Koeberl, and A. Sadeghi. "Hardware-Assisted Fine-Grained Control-Flow Integrity:Towards Efficient Protection of Embedded Systems Against Software Exploitation" in *Proc. the Design Automation Conference (DAC'14)*, pp.1-6, 2014.

[31] M. Budiu, U. Erlingsson, and M. Abadi. "Architectural Support for Software-Based Protection" in Proc. the 1st workshop on Architectural and system support for improving software dependability. (ASID'06), pp. 42-51, 2006.

[32] B. Zeng, T. Gang. and G. Morrisett. "Combining Control-Flow Integrity and Static Analysis for Efficient and Validated Data Sandboxing" in Proc. *the 18th ACM Conference on Computer and Communication Security(CCS '11).*pp. 29-40, 2011.

[33] C. Tice,T. Roeder, S.Checkoway, U.Erlingsson, L. Locano, G. Pike, and P. Collingbournel. "Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM" in *Proc. the 23rd USENIX Security Symposium,(Usenix'14),* pp. 941-955, 2014.

[34] C. Zhang, T. Wei. Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou. "Practical Control Flow Integrity & Randomization for Binary Executables" in *Proc of the 2013 IEEE Symposium on Security and Privacy.(SP'13)*, pp. 559-573, 2013.

[35] Y. Xia,Y. Liu,H. Chen, and B. Zang. "CFIMon: Detecting Violation of control flow integrity using performance counters" in Proc. *the 2012 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'12)*, pp. 1-12, 2012.

[36] M. Zhang,R. Sekar. "Control flow integrity for COTS binaries." in *Proc. the 22nd USENIX Security Symposium.(Usenix'12)*, pp. 337-352, 2013.

[37] J. Criswell, N. Dautenhahn, and V. Adve. “KCoFI: Complete control-flow integrity for commodity operating system kernels.” in Proc. The 2014 *IEEE Symposium on Security and Privacy,(SP’14)*, pp. 292-307, 2014.

[38] W. Dai, Z. Liu, Y.H. Liu. “Function Pointer Attack Detection with address integrity checking” *Journal of Computer Applications*. Vol. 35, No. 2, pp. 424-429, 2015.
(代伟, 刘智, 刘益和, “基于地址完整新检查的函数指针攻击检测”, *计算机应用*, 2015,35(2):424-429.)

[39] E. Athanasopoulos.H. Bos,G. Portokalidis, and E. Goktas. “Out of Control Overcoming Control-Flow Integrity” in *Proc. the 2014 IEEE Symposium on Security and Privacy.(SP’14)*, pp. 575-589, 2014.

[40] N. Carlin,A. Barresi,D. Wagner, M. Payer, and T. R. Gross. “Control-Flow Bending:On the Effectiveness of Control-Flow Integrity” in *Proc of the 24th USENIX Security Symposium. (Usenix’15)*, pp. 161-176, 2015.

[41] C. Liebchen, M. Nergro, P. Larsen, M. Conti, S. Crane, L. Davi, M. Franz, M. Qunaibit, A. Sadeghi. “Losing Control : On the Effectiveness of Control-Flow Integrity under Stack Attacks” in *Proc. the 22nd ACM Conference on Computer and Communication Security (CCS’15),* pp. 952-963, 2015.

[42] I. Evans, F. Long, H. Shrobe, U. Otgonbaatar, and M. Rinard. “Control Jujutsu: On the Weaknesses of Fine-Grained Control Flow Integrity” in *Proc. of the 22nd ACM Conference on Computer and Communication Security (CCS’15)*, pp. 901-913, 2015.

[43] A. Bittau, D. Boneh,D. Mazieres, and A.J.Mashtizadeh. “CCFI: Cryptographically Enforced Control Flow Integrity” in *Proc. the 22nd ACM Conference on Computer and Communication Security (CCS’15)*, pp. 941-951, 2015.

[44] D. Andriesse,V. Veen, B. Gras, H. Bos, L. Sambuc, A. Slowinska, and C. Giuffrida. “Practical Context-Sensitive CFI” in *Proc. the 22nd ACM Conference on Computer and Communication Security (CCS’15)*, pp. 927-940, 2015.

[45] PAX Team. “Address Space Layout Randomization” http://pax.grsecurity.net/docs/aslr.txt, 2003.

[46] H. Shacham,M. Page, B. Ptaff, E. Goh, N. Modadugu, and D. Boneh. “On the Effectiveness of Address-Space Randomization” in *Proc. the 11th ACM Conference on Computer and Communication Security. (CCS’04)*, pp. 298-307, 2004.

[47] G. Roglia, L. Martignoni, R. Paleari, and D. Bruschi. “Surgically Returning to Randomized Lib(c)” in *Proc. the 2009 Annual Computer Security Application Conference (ACSAC ’09)*, pp. 60-69, 2009.

[48] M. Backes, S. Nurnberger. “Oxymoron: Making Fine- Grained Memory Randomization Practical by Allowing Code Sharing” in *Proc. the 23rd USENIX Security Symposium. (Usenix’14)*, pp. 433-447, 2014.

[49] K. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A. Sadeghi. “Just-In-Time Code Reuse:On the Effectiveness of Fine-Grained Address Space Layout Randomization” in Proc. the 2013 IEEE Symposium on Security and Privacy.(SP’13), pp. 574-588, 2013.

[50] J. Hiser, M. Co, M. Hall, A. Nguyen-Tuong, and J. Davidson. “ILR: Where’d My Gadgets Go?” in *Proc. the 2012 IEEE Symposium on Security and Privacy.(SP’12)*, pp. 571-585, 2012.

[51] V. Pappas, M. Polychronakis, and A. Keromytis. “Smashing the Gadgets: Hindering ROP Programming Using In-Place Code Randomization” in *Proc. the 2012 IEEE Symposium on Security and Privacy (SP’12)*, pp. 601-615, 2012.

[52] R. Wartel, V. Mohan, K. Halen, and Z. Lin. “Binary Stirring: Self-randomizing Instruction Addresses of Legacy x86 Binary Code” in *Proc. the 19th ACM Conference on Computer and Communication Security(CCS ’12)*, pp. 157-168, 2012.

[53] L. Davi, A. Dmitrienko, A. Sadeghi, and S. Nurnberger. “Gadge Me If You Can: Secure and Efficient Ad-hoc Instruction-Level Randomization for X86 and ARM” in *Proc. the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security. (ASIA CCS’13)*, pp. 299-310, 2013.

[54] L. Davi, C. Liebchen, A. Sadeghi, K. Snow, and F.Monrose. “Isomeron: Code Randomization Resilient to (Just-In-Time) Return-Oriented Programming” in *Proc. the 2015 Network and Distributed System Security Symposium.(NDSS’15)* , 2015

[55] L. Xiao. “The Design and Implementation of a Function Lever Randomization Defensing Method against ROP Attack [Master Degree Thesis],” Nanjing:Software School,Nanjing University, 2013.
(肖亮.一种针对 ROP 攻击的函数粒度随机化防御方法的设计与实现[硕士学位论文]. 南京: 南京大学软件学院, 2013.)

[56] X. Zhan. “The Research of Defending ROP Attacks Using Basic Block Level Randomization [Master Degree Thesis],” Nanjing: Software School, Nanjing University, 2014.
(詹珣. 针对 ROP 攻击的块粒度地址空间随机化防御技术的研究[硕士学位论文]. 南京: 南京大学软件学院, 2014)

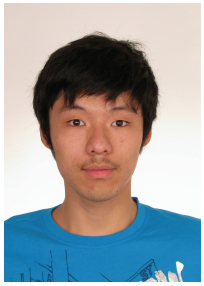
[57] S. Crane, C. Lirbchen, A. Homescu, L. Davi, P. Larsen, A. Sadeghi, S. Brunthaler, and M. Franz. “Readactor: Practical Code Randomization Resilient to Memory Disclosure” in *Proc of the 2015 IEEE Symposium on Security and Privacy(SP’15)*. 2015

[58] M. Backes, T. Holz, B. Kollenda, P. Koppe, S. Nurnberger, and J. Pewny. “You Can Run but You Cant Read Preventing Disclosure Exploits in Executable Code” in *Proc. the 2014 ACM SIGSAC Conference on Computer and Communications Security.(CCS’14)*, pp. 1342-1353, 2014.

[59] A. Tang, S. Sethumadhavan, S. Stolfo, “Heisenbyte: Thwarting Memory Disclosure Attacks using Destructive Code Reads” in *Proc, the 22nd ACM Conference on Computer and Communication Security (CCS’15)*, pp. 256-267, 2015.

[60] K. Lu, C. Song, W. Lee, S. Chung, T. Kim, and W. Lee. “ASLR-Guard:Stopping Address Space Leakage for Code Reuse Attacks” in *Proc of the 22nd ACM Conference on Computer and Communication Security (CCS’15)*, pp. 280-291, 2015.

[61] C. Yang, Q.X. Wang, Q. Wei, “Code Reuse Attack Mitigation Based on Unreadable Property of Executable Memory”, Journal of Information Engineering University. Vol. 17, No. 1, 2016.
(杨超, 王清贤, 魏强, “基于可执行内存不可读属性的防代码重用技术”, *信息工程大学学报*, 2016,17(1):59-64.)



柳童　于 2013 年在西安电子科技大学计算机科学与技术专业获得学士学位。现在中国科学院信息工程研究所系统结构专业攻读博士学位。研究领域为计算机系统安全。研究兴趣包括代码重用攻击、系统防御等。Email: liutong9017@iie.ac.cn

孟丹　于 1995 年在哈尔滨工业大学获得计算机体系结构专业博士学位。现任中国科学院信息工程研究所所长。研究领域包括计算机系统安全, 大数据与云计算等。研究兴趣包括计算机系统安全, 云计算安全等。Email: mengdan@iie.ac.cn

史岗　于 2004 年在中国科学院计算技术研究所计算机体系结构专业获得博士学位。现任中国科学院信息工程研究所第五研究室高级工程师。研究领域为计算机系统安全。研究兴趣包括计算机架构, 计算机芯片安全等。Email: shigang@iie.ac.cn