第 2 卷 第 4 期　　信 息 安 全 学 报　　Vol. 2 No. 4
2017 年 10 月　　Journal of Cyber Security　　October, 2017

程序漏洞:原因、利用与缓解——以 C 和 C++语言为例

陈小全[1,2], 薛　锐[1]

[1] 中国科学院信息工程研究所信息安全国家重点实验室, 北京 中国 100093
[2] 中国科学院大学 网络空间安全学院, 北京 中国 100049

摘要　程序中存在的漏洞是针对程序的各种攻击事件的根源, 攻击者可以利用这些漏洞改变程序的行为或完全控制程序。本文以 C 语言和 C++语言为例循序渐进地阐明了程序中漏洞产生的根本原因, 并对利用这些漏洞实施的攻击进行了深入地分析和探讨, 同时也指出了当前主要的漏洞检测和漏洞阻止技术的优势和不足。最后, 我们提出了对程序进行持续的和全面的内存布局多样性的未来研究方向。

关键词　程序漏洞；利用；缓解
中图法分类号　TP309.2　DOI 号　10.19363/j.cnki.cn10-1380/tn.2017.10.004

Cause, Exploitation and Mitigation of Program Vulnerability—C and C++ language as an example

CHEN Xiaoquan[1,2], XUE Rui[1]

[1] State Key Laboratory Of Information Security, Institute of Information Engineering, Chinese Academy of Sciences, Beijing 100093, China
[2] School of Cyber Security, University of Chinese Academy of Sciences, Beijing 100049, China

Abstract　The vulnerability in the program is the source of the attacks against the program. These vulnerabilities allow the attackers to alter the behavior of the program or completely control the program. Firstly, this paper clearly explains the fundamental reason of the vulnerability in the program. Secondly, the attacks that exploit these vulnerabilities have been analyzed and discussed thoroughly. Thirdly, this paper also points out the advantages and weaknesses of the current vulnerability detection and defense technology. Finally, the future research direction —— the Continuous and Comprehensive Memory Layout Diversity is proposed.

Key words　program vulnerability；exploitation；mitigation

1　引言

现代社会是一个信息化的社会, 每天的生活离不开种类繁多、功能强大的软件, 大到用来预测天气情况的超级计算机系统, 小到时时在使用的社交应用, 如 WeChat、Twitter、Facebook 等。可以说, 人类已经离不开这些智能的生产、生活的好帮手。但是, 任何事情都有其两面性: 在你享受其带来的便利的同时, 也必将遭受其带来的困扰, 有时甚至是噩梦。当前, 随着计算机和互联网的飞速发展, 针对各种软件系统的攻击层出不穷, 与日俱增, 给社会与个人造成了严重的损失: 信息泄露、财产损坏、安全受到威胁等等。在这些让人遭受严重损失的攻击中, 绝大部分都是利用了软件系统中存在的某些软件瑕疵(Software Defect)。软件瑕疵是指在开发软件的过程中, 人类的某些错误融入进了软件中。这些错误既包括设计上的不完善, 也包括编码上的问题。如果这些瑕疵使软件处于风险之中, 那么这些软件瑕疵就被称为安全缺陷(Security Flaw)。如果这些安全缺陷被某些意图不轨、心怀叵测的人利用来获得某些利益, 那么这个安全缺陷就成为了一个软件漏洞(Software Vulnerability)。在软件开发的过程中, 软件漏洞存在于软件开发的三个

通讯作者: 薛锐, 博士, 研究员, Email: xuerui@iie.ac.cn。
本课题得到中国科学院战略性先导科技专项(批准号: XDA06010701)项目, 国家自然科学基金(No. 61472414, No. 61772514), 中国科学院信息工程研究所密码基金资助。
收稿日期: 2016-09-30; 修改日期: 2017-02-08; 定稿日期: 2017-08-23

主要阶段:架构设计, 编码实现和实际运行。

架构设计阶段是每个软件开发的核心阶段。在这一阶段需要对关于如何开发软件做出一系列的关键决策, 这些决策包括软件的高层组件、功能组件、安装部署 、性能优化等。这一阶段存在的安全缺陷主要有架构设计不合理、需求不完善等。这些安全缺陷又造成了软件漏洞。一般情况下, 架构设计阶段产生的漏洞在软件开发的过程中很难被发现, 但危害却巨大。因此, 在架构设计阶段对软件进行合理和安全的设计可以大幅度地降低软件出现漏洞的概率。并且, 在该阶段越早发现软件中存在的漏洞, 则漏洞修复的成本也就越低。软件运行阶段是三个阶段中的最后一个阶段。这一阶段需要做的工作有软件部署运行和软件运行环境配置等, 存在的安全缺陷主要有软件配置错误、认证安全错误以及系统数据安全性错误等。而由这些安全缺陷造成的漏洞有访问验证漏洞和数据机密性漏洞等。这些漏洞和架构设计阶段产生的漏洞, 通常意义下被认为是软件设计方面的漏洞。因此, 这两个阶段漏洞产生的原因与软件开发过程中处于中间的阶段——软件编码实现阶段的关系不大, 主要是由于软件开发主体自身的设计原因造成的。这就像建一座房子, 在房子的设计阶段为房子的前门加了一把锁, 而对房子的后门却没有任何的安全防护措施。在施工阶段, 工人按照设计图纸来施工。房子建成后却经常发生失窃事件, 究其原因, 原来是设计上有问题: 没有为后门加一把锁, 小偷经常从后门出入。因此, 对于架构设计和软件运行这两个阶段产生的漏洞, 主要从软件设计上来防范, 采用的措施有安全设计原则和安全设计策略。主要的设计原则有最低权限原则、深度防护原则、权限分离原则等; 主要的安全设计策略有: 完整性策略、认证机制以及机密性策略[1]。

软件编码实现阶段所产生的漏洞是传统意义上的漏洞, 也是学术界和工业界重点关注和研究的漏洞, 同时也是本文所要关注的漏洞。这一阶段产生漏洞的主要原因是由于软件开发者未能对开发语言中某些关键特性进行谨慎地使用而造成的。当前, 很多的软件系统是用 C 和 C++ 语言开发的, 此外还存在着大量的用 C 语言开发的遗留代码库。并且, 目前软件中所暴露出来的漏洞很多都和 C 语言、以及 C++语言中某些特性有着千丝万缕的联系。因此, 本文将以 C 语言和 C++语言为例, 分析和探讨软件编码实现阶段所产生的漏洞的主要原因、利用漏洞实施的攻击以及各种缓解技术, 并在本文的最后提出漏洞阻止技术未来的研究方向, 用以缓解利用软件漏洞实施的攻击。

众所周知, C 语言不是一种类型安全的语言。类型安全的语言要求对某特定类型的操作, 其结果仍然是该类型[2]。C 语言来源于两种无类型的语言, B 语言[3]和 BCPL(Basic Combined Programming Language)语言[4]。所以仍然保留着许多无类型语言的特征, 如需要类型的显式或隐式转换, 而这种转换有可能产生安全缺陷。例如, 将指向某一类型的指针转换为指向另一类型的指针, 而后对转换后的指针进行解引用时, 会产生未定义的行为[5], 进而产生一个可被利用的漏洞。因此, 这种类型安全的缺乏导致了 C 语言中安全缺陷与漏洞的产生。此外, C 语言还有一些设计方面的特性, 如不会自动进行数组边界检查、指针初始化、内存释放与回收等等的工作。如果程序员在开发的过程中, 由于无意识或遗忘而没有主动地进行上面的工作的话, 也会导致程序漏洞的出现。综上所述, C 语言在软件开发过程中的大量使用, 以及C语言的某些特性, 使其成为漏洞频发的重灾区。本文针对使用 C 语言的软件编码实现阶段所产生的漏洞, 进行了系统和深入的分析与探讨, 做了如下的工作。在第二节我们系统地分析了 C 语言程序中存在的漏洞, 以及漏洞产生的根本原因。在第三节, 对利用这些漏洞实施的攻击进行了分类探讨, 指出利用漏洞成功实施攻击的关键所在。在第四节, 对当前主要的漏洞检测和漏洞阻止技术进行了分类讨论, 分析了各种技术的特点。在第五节, 对当前最具代表性的漏洞阻止技术进行了实验对比分析, 深入讨论了这些技术的实现方式和存在的主要问题。第六节在前面几节的基础上, 对各种安全技术进行了进一步地讨论和分析, 指出了各种技术的优势所在, 以及不足之处。第七节我们提出了未来的研究方向, 并指出了该研究方向所面临的挑战和前景。最后, 在第八节总结了论文的主要内容。

2 程序中存在的漏洞

正如上面所描述的, C 语言是一种类型不安全的语言。由于开发人员有意或无意的行为, 以及 C 语言固有的一些特性, 使得所开发的程序中存在某些瑕疵或安全缺陷, 而这些瑕疵或安全缺陷就有可能造成被攻击者所利用的漏洞。当前, 在使用 C 语言开发的程序中存在的漏洞可以分为以下五种: 缓冲区溢出(Buffer Overflow)漏洞、指针覆写(Pointer Subterfuge)漏洞、内存管理错误(Memory Management Error)漏洞、整数溢出(Integer Overflow)漏洞和格式化输出(Formatted Output)漏洞。在表 1 中, 我们对程序中存

在的漏洞进行了总结。

缓冲区溢出漏洞是指向某一数据结构的内存空间中写入数据时, 写入的数据超出了分配给该数据结构的内存空间的边界, 覆盖了该数据结构相邻内存空间中的内容。这种漏洞经常产生在和数组有关的操作中, 例如, 向一个元素个数为 10 的数组里写入 11 个数据的操作。产生该漏洞主要的原因是: (1)由于开发人员的粗心大意或无意识, 在执行从源字符串向目标字符串的复制操作时, 不对目标字符串数组的边界进行检查, 这极有可能使复制的字符个数超出目标字符串数组的边界, 从而产生溢出的漏洞; (2) C 语言为编程人员提供了众多的函数调用。这些函数极大地方便了编程人员的开发工作, 使他们的开发效率大大提高。但是这些函数共有的一个缺陷是对 C 语言中的数据类型不进行强制性的边界检查。这就会造成程序运行过程中越界操作数据元素的错误, 而这种错误同样会产生溢出漏洞。缓冲区溢出是一种很严重的漏洞, 会导致针对存在该漏洞程序的控制流劫持攻击, 我们会在第三节进行详细的分析。此外, 当缓冲区溢出发生在栈上时, 会产生栈溢出(Stack Smashing)[6], 发生在堆上时, 会产生堆溢出(Heap Overflow)[7]。

表 1 程序中的漏洞

漏洞	原因	利用	著名的漏洞
缓冲区 溢出	开发人员粗心大意, 不对数组边界进行检查; 函数对数据类型不进行强制性的边界检查	栈溢出 代码注入攻击 返回函数攻击 返回导向编程	Morris worm W32.Blaster.Wo rm
指针覆写	对指针不正确的使用	代码注入攻击 返回函数攻击 返回导向编程	overwriting the GOT entry exception-handler hijacking
内存管理	动态内存管理技术不正确的使用	信息泄露 空间错误 时间错误 控制流劫持 非控制数据攻击	CVS Buffer Vulnerability Vulnerabilities in MIT Kerberos 5
整数溢出	丢失或者错误表示的数据	整数回绕 转换和截断 整数逻辑错误	N/A
格式化输出	格式化输出函数接受包含格式化字符串的 可变数量的参数	信息泄露 控制流劫持	Washington University FTP Daemon CDE ToolTalk Ettercap Version NG-0.7.2

指针覆写漏洞是指由于对指针没有进行很好的处理和保护, 造成指针的值被修改和利用的漏洞[8]。一般情况下, 可以通过栈上的缓冲区溢出来覆盖和修改函数的返回地址, 从而让程序的控制权转移至攻击者提供的恶意代码处, 完成其攻击的意图。如果要利用指针覆写漏洞实施攻击, 一般情况下需要满足下面的条件: (1)程序中存在缓冲区溢出漏洞, 这样可以通过缓冲区溢出来覆写指针的值; (2)发生溢出错误的缓冲区要和被覆写的指针在同一个内存段内, 且位置最好相邻。这样攻击者可以直接通过覆写与溢出缓冲区相邻的指针来实施攻击。在程序中可以被覆写利用的指针有函数返回地址、对象指针、全局偏移量表(Global Offset Table, GOT)中的函数绝对地址[9]、C++中的虚指针(Virtual Pointer)等。

C 程序中经常需要对元素个数可变的数据对象进行操作, 因此需要用动态内存管理函数为这些数据对象进行动态的内存分配。但是, 不正确地使用动态内存管理函数会产生很多的错误, 而这些错误又导致了许多和内存有关的漏洞, 使得内存管理成为程序中漏洞的主要来源之一。常见的内存管理错误有: (1)内存初始化错误。当用内存分配函数, 如 *malloc()*函数分配内存时, 所分配的内存并没有被初始化, 里面的内容并不确定, 容易产生信息泄露的问题[10]。而之所以调用类似 *malloc()*的函数分配内存, 却不自动初始化的原因主要是性能方面的考虑: 初始化大的内存块会使系统的性能下降。因此, C 标准委员会把是否初始化的决定权留给编程人员。而编程人员的素质良莠不齐, 产生了大量的内存初始化错误, 被攻击者利用来获取程序中有价值的信息。(2)空指针或无效指针解引用错误。当使用内存分配函数分配内存时, 如果分配不成功, 函数会返回一个空指针。此外, 当指针超出它所指对象的边界或所

指的对象被释放时, 它就会变成一个无效的指针。对空指针和无效指针的解引用导致所谓的时间错误(Temporal Error)或空间错误(Spatial Error), 引起程序崩溃。在某些情况下, 也会导致任意代码的执行攻击[11]。(3)引用已经被释放的内存。当分配给一个对象的内存被释放时, 指向该内存空间的指针应该被赋值为空, 或者重新指向其他的内存空间。否则, 通过该指针仍然可以访问已经被释放的内存空间, 产生信息泄露的问题。并且, 如果对该指针所指向的内存块写入攻击者所控制的数据时, 这就是一个可以被利用的漏洞了。(4)内存泄漏。当分配给对象的内存空间不再需要时却不释放, 就会产生内存泄露的错误。内存泄露的极端情况是系统没有可分配的内存空间, 产生拒绝服务的攻击。

当有符号整数运算的结果超出了该整数类型所能表达的最大值时, 就产生了整数溢出的漏洞。根据文献[12]的分类, 整数溢出可分为整数上溢、整数下溢、符号错误和截断错误。整数之间的加法、减法、乘法和移位操作导致的整数溢出和整数之间的类型转换操作导致的截断错误会使整数计算得到的结果与期望的值存在较大差异, 同时类型转换操作所导致的符号错误也会混淆负数和极大正数之间数值的正确理解。总之, 整数溢出会导致程序计算结果错误、数据丢失和数值被错误理解。但是整数溢出并不会直接对内存的内容进行修改, 所以攻击者通常并不直接利用整数溢出漏洞来实施攻击, 而是利用程序存在的整数溢出漏洞间接地实施某种攻击, 危害程序的安全。

格式化输出函数是C语言中定义的一些可以接受可变数量参数的输出函数, 其中一个参数称为格式化字符串(Format String)。用户可以通过控制格式化字符串来控制格式化输出函数的执行。对格式化输出函数的不正确使用可以产生下列的漏洞: (1)缓冲区溢出。一般情况下, 格式化输出函数会假定存在任意长度的缓冲区。在这种假设下, 如果不对目标数组进行边界检查, 则会发生溢出错误。而这个错误会被作为实施漏洞攻击的前提条件。(2)利用格式化字符串打印程序栈和内存的内容。在使用格式化输出函数时, 攻击者可以使用“显示指定地址的内存”的格式化字符串来查看栈和任意地址的内存。例如, 格式转换指示符“%*s*”可以显示参数指针所指定的地址的内存。如果攻击者能够操作这个参数指针来引用一个特定的地址, 那么格式转换指示符“%*s*”将会输出该地址内存空间的内容。通过这种方式, 攻击者就可以获得程序中指定地址内存的内容, 使得程序的信息发生泄露。

3 利用程序中的漏洞实施的攻击

在第二节中, 我们把程序中存在的漏洞分为缓冲区溢出漏洞、指针覆写漏洞、内存错误漏洞、整数溢出漏洞和格式化字符串漏洞等五种类型。这五种漏洞类型基本上涵盖了程序中通常存在的漏洞。并且大量的研究资料显示, 攻击者利用漏洞攻击程序时, 主要就是通过这五种漏洞来实施的。但是, 一般情况下攻击者不会利用某个特定的漏洞来实施攻击, 而是综合使用若干个漏洞来实施攻击。例如, 攻击者会利用内存错误漏洞、格式化字符串漏洞获取程序的内存布局信息, 然后利用缓冲区溢出漏洞、指针覆写漏洞覆盖程序栈上调用函数的返回地址, 使程序的执行转向攻击者设定好的代码(Shell Code), 最终完成攻击。当前, 根据攻击者攻击的目的和方式, 利用程序漏洞实施的攻击基本上可以分为四种类型: 信息泄漏(Information Leak)、内存错误(Memory Corruption)、控制流劫持(Control-Flow Hijack)以及非控制数据攻击(Non-control Data Attacks, NDA)[13-15]。

信息泄露攻击是指利用程序中存在的漏洞, 通过非授权的方式得到与程序有关的信息, 如程序在内存中的地址空间布局、函数的入口地址、程序所使用的寄存器的内容等。或者还可以得到与用户有关的信息, 如系统登录的密码、信用卡账号等。此外, 信息泄露攻击除了直接获取程序的信息外, 还可以作为其他攻击的前提条件, 如控制流劫持攻击。在控制流劫持攻击中, 攻击者先要通过程序存在的漏洞获得内存地址的信息(信息泄露攻击)。然后, 把设计精巧的攻击代码作为函数(该函数是一个存在安全缺陷的函数, 如 *strcpy()*)的输入参数传递到程序中。当漏洞程序运行时劫持其执行流, 使执行流转向攻击者设定好的攻击代码处, 完成控制流劫持攻击。信息泄露攻击分为两种类型, 一种是入侵型(Intrusive)的攻击方式, 这种方式在获得程序的相关信息后, 主动的去修改漏洞程序的内容(如修改程序栈), 以达到攻击的目的。还有一种称之为侧信道攻击(Side Channel Attacks, SCA)的方式, 它是一种非入侵的信息泄露攻击方式。这种攻击方式并不去主动的修改程序的内容, 它通过程序中存在的漏洞对运行的程序输入一些数据, 观察程序对输入数据的反应, 从而推断出程序具有的某些“信息”。侧信道攻击最初是针对电子设备在运行过程中的时间、功率消耗或电磁辐射之类的信息泄露而实施攻击。但有迹象表

明, 这种攻击方式逐步渗入到了对程序实施攻击[14]。在信息泄露攻击中, 可以利用的漏洞有内存错误漏洞和格式化字符串漏洞。

内存错误攻击是一种常见的攻击方式。在内存错误攻击中, 如果开发人员使用 *malloc()*函数分配所需的内存块, 但却没有对分配成功的内存块进行初始化操作(正如我们在第二节所介绍的, *malloc()*函数不会对分配的内存块进行初始化)。此时, 攻击者可以通过程序中的漏洞(如格式化字符串漏洞), 把未初始化的内存块的内容打印出来, 从而使程序产生保密或隐私泄露的危险。同样, 这种危险也会发生在攻击者引用已经被释放的内存块的情况下。此外, 如果开发人员动态申请了多块内存, 在使用完后却不进行释放, 则会产生内存泄露的危险。当这种危险被攻击者利用时, 会耗尽程序的内存, 进而使得系统没有内存可以分配。 通常情况下, 内存错误攻击可能是攻击者的最终目的, 如为了获取程序或用户的某些信息; 也可能是攻击者实施攻击的一个中间步骤, 如为了实施控制流劫持攻击需要提前了解漏洞程序的地址空间布局情况。但是, 不管是哪一种目的, 内存错误攻击在攻击者实施的攻击中都起着重要的作用。在内存错误攻击中, 可以利用的漏洞有缓冲区溢出漏洞、指针覆写漏洞、内存错误漏洞以及格式化字符串漏洞。

控制流劫持攻击是当前危害极大的一种攻击方式, 攻击者能够通过它来获取目标机器的控制权, 进而对目标机器进行全面的控制。一般情况下, 程序在其运行过程中, 应当按照预先定义好的控制流图(Control Flow Graph, CFG)来执行, 以确保程序的控制流不被劫持或篡改, 偏离程序所设计好的控制流转移关系。但是, 如果一个攻击者掌握了程序中可以被利用的漏洞, 如缓冲区溢出漏洞, 他就可以利用该漏洞劫持或篡改程序的执行流程, 使程序向攻击者设定好的方向执行, 达到其攻击的目的。早期的控制流劫持攻击采用代码注入的方式, 称为代码注入式攻击(Code Injection Attacks, CIA)。在这种攻击方式中, 攻击者利用程序中的漏洞注入需要的恶意代码, 再篡改程序的控制流使其指向注入的代码, 使得注入代码得以执行, 达到攻击意图(如系统权限提升, 隐私窃取等)。近年来, 控制流劫持攻击逐步发展为利用漏洞程序中存在的函数和代码来实施攻击。利用程序中已经存在的函数实施攻击, 称为返回函数库攻击(Return-into-libc, RILC)[16-17]; 利用程序中已经存在的代码实施攻击, 称为返回导向编程(Return Oriented Programming, ROP)[18]。在 RILC 攻击中, 攻击者篡改程序的控制流, 使其指向程序中已有的一些标准库函数。攻击者可通过这种攻击手段来调用 *system()*函数生成新进程, 或者调用 *mprotect()* 函数创建可写可执行的内存区域用以绕过系统传统的防御方法。此类攻击方法被改进后可连续调用程序中的代码段和标准库中一系列的函数, 完成更复杂的功能。但是, 由于 RILC 攻击方法中供攻击者选择的函数数量比较少, 而且攻击的成功与否要依赖于标准库中的几种关键的函数, 因此 RILC 应用的范围不是很大。在 2007 年, 文献[18]扩展了 RILC 的思想, 提出了 ROP 技术。在 ROP 攻击技术中, 攻击者可以利用程序指令中的返回指令 ret 组织一些短小的攻击代码片段来实施攻击, 这些短小的攻击代码片段被称为 gadget。因为程序在运行过程当中, 程序本身和程序所使用的库函数将有大量的二进制代码被加载到内存中, 这些代码中有非常多的代码片段是以 ret 指令结束的。如图 1 所示。当一个代码片段执行完毕, 执行最后一条 ret 指令时, 程序栈的下一个地址将被取出, 从而完成从一个代码片段向另一个代码片段的跳转。这样, 一个一个的代码片段按顺序执行完后, 攻击者的攻击意图也就达到了。因此在 ROP 攻击中, 通过有效的构造程序栈的内容, 可以将程序中本身存在的二进制代码片段有机的组合, 构成恶意的攻击代码, 达到破坏系统或窃取信息的目的。与 RILC 攻击相比, ROP 攻击不需要程序中的标准库函数, 因为程序中存在大量的以 ret 指令结尾的代码片段, 所以攻击者可以很容易地构造用于攻击的代码片段, 进而使得攻击很容易地被实施。而且在 2007 年, Shacham 等人[18]也证明这些短小的代码片段是具有图灵完备特性的, 这使得构造用于攻击的代码片段更加便利。而且由于 ROP 攻击便利的特点, 其攻击思想已经被应用在了许多平台架构上, 如 Intel x86[18]、SPARC[19]、Atmel AVR[20]、ARM[21-22] 以及 PowerPC[23]。但是, 由于 ROP 使用以 ret 结尾的指令序列来构造攻击负载, 而这种明显的特征使得 ROP 攻击很容易被检测到。因此, 为了弥补 ROP 攻击的这个缺陷, 安全研究人员们对 ROP 攻击技术进行了改进。如在 2010 年, Checkoway 等人[24]使用含 pop-jmp 这种类似 ret 功能的指令序列代替原有的以 ret 结尾的指令序列。在 2011 年, Bletsch 等人[25]试图寻找以 jmp 结尾的指令序列构造攻击, 这类攻击技术被称作跳转导向编程(Jump-Oriented Programming, JOP)技术。在 2013 年, Snow 等人[26]利用程序的内存错误漏洞, 通过一个单独的代码指针来映射一个内存页面的地址范围, 进

而映射整个程序的内存布局, 在此基础上, 挑选和组织大量的 gadget 在程序运行的过程中实施攻击。这种攻击方法称为即时返回导向编程(Just-In-Time Return Oriented Programming, JIT-ROP)。在2014年, Carlini 等人[27]利用 ROP 攻击中, ret 指令被执行时, 目标必定是一个 call-preceded 指令的特点, 提出了 call-preceded ROP 攻击, 该攻击方法使得 ROP 攻击更具一般化。在2015年2月, Davi 等人在文献[28]中提出了一种新的即时代码复用攻击方法。在这种方法中, 攻击者可以从程序的堆或栈段的一个代码指针来映射程序的整个内存布局, 而不是使用程序代码段中直接分支的代码指针。以往攻击者都是关注了程序的代码段部分, 因为程序的代码段是程序的可执行部分, 有可以运行的指令, 而堆或栈段是程序用来临时存放数据的部分, 没有可以运行的指令。在2015年5月, Schuster 等人[29]针对用 C++ 等面向对象语言编写的程序提出了 COOP (Counterfeit Object-Oriented Programming, COOP)攻击。在这种攻击方法中, 攻击者通过劫持 C++程序中的虚拟函数来映射程序的内存布局, 进而构造若干 gadget 来实施攻击。此外, 还有 Bittau 等人[30]提出的 Blind-ROP 攻击。在控制流劫持攻击中, 可以利用的漏洞包括缓冲区溢出漏洞、指针覆写漏洞、内存错误漏洞和格式化字符串漏洞。图2显示了控制流劫持攻击的发展历程。

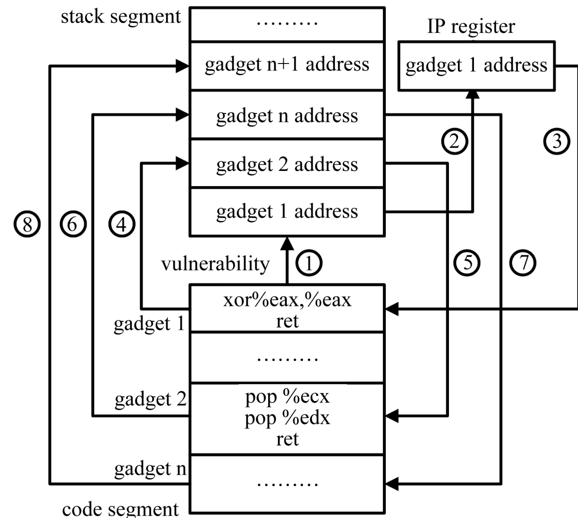

图1 ROP 攻击原理

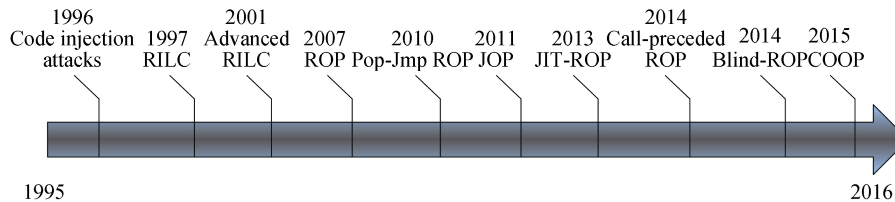

图2 控制流劫持攻击的发展历程

还有一种与控制流劫持类似的攻击, 在这种攻击中, 攻击者利用缓冲区溢出等漏洞修改的不是程序中的返回地址或函数指针(程序中的控制数据), 而是程序中的非控制数据, 如用户身份信息、配置信息以及密码等具有一定语义的变量。这种攻击我们称之为非控制数据攻击。攻击者实施这种攻击的前提条件是需要对程序的数据区域非常熟悉, 并据此构造出巧妙的攻击数据负载。代码 1 显示了一个非控制数据攻击的例子。

代码 1 非控制数据攻击

```
1 void login_client(int file){
2 bool admin=false;
3 char client_name=[128];
4 /*a buffer overflow error*/
5 read(file,client_name,256);
6 if (admin)
7 printf("welcome!");
8 }
```

在这个例子中, 攻击者可以通过由 *read()*函数引起的缓冲区溢出错误, 设计一个巧妙的攻击负载来覆盖变量 *admin* 的值, 从而以非管理员的身份登录系统。从这个例子我们不难看出, 在非控制数据攻击中, 攻击者所修改的内容并不是程序的控制数据, 而是非控制数据。并且, 这个非控制数据是一个有语义的变量, 从二进制代码的层次很难阻止其被修改。而且, 在2016年, Hu 等人[31]提出一种新的基于非控制数据的攻击方法-数据导向编程(Data-Oriented Programming, DOP)。DOP 攻击类似于 ROP, 通过在程序中构造大量的“data-oriented gadget”来实施攻击, 并且该攻击方法被证明是图灵完备的。但是, 非控制数据攻击实施的难度要比控制流劫持攻击大很多, 因为它需要对漏洞程序进行深层次的了解, 最好是能得到漏洞程序的源代码并对其进行分析, 了解程序中各变量的语义信息以及相对位置(便于利用缓冲区溢出漏洞进行信息覆盖)。而且, 通过对二进制程序实施反汇编得到的信息对攻击实施并没有多大的帮助。因为很多商业化的二进制程序加入了代码混淆技术, 使得通过反汇编得到的信息一般情况下不具有利用价值。然而, 由于这种攻击没有修改程

序中的任何控制数据, 现有的防御措施(如 DEP 和 ASLR)很难检测与阻止该攻击, 所以该攻击对系统安全也是一个很大的威胁。同样, 在非控制数据攻击中, 可以利用的漏洞包括缓冲区溢出漏洞、指针覆写漏洞、内存错误漏洞和格式化字符串漏洞。

4 阻止利用漏洞实施攻击的缓解技术

漏洞缓解(Mitigation)技术是用于防止漏洞被利用的解决方案。本文为了说明问题的方便性, 将缓解技术分为漏洞检测技术和漏洞阻止技术。但是, 在这里我们需要说明的是, 漏洞检测技术具有双面性: 攻击者可以用它来检测出程序中存在的漏洞, 进而利用该漏洞来实施攻击; 安全人员却可以用它来检测出程序中存在的漏洞, 进而修补该漏洞, 阻止攻击者利用该漏洞来实施攻击。而漏洞阻止技术则是专门用来阻止利用漏洞来实施攻击的技术。

4.1 漏洞检测技术

漏洞检测的过程可以静态地进行(此时的程序处于非运行的状态), 检测的对象是程序的源代码或者二进制代码; 也可以动态的进行(程序处于运行的状态), 检测的对象是运行中的程序和外部输入的不受信任的数据, 然后采用状态重现的方法, 确定程序中漏洞的位置。漏洞检测技术如表 2 所示。

表 2 漏洞检测技术

类型	技术	流程	工具
静态检测	1. 数据流分析 2. 符号执行	1. 创建代码模型 2. 建立漏洞数据库 3. 执行分析引擎 4. 判定漏洞 5. 分析结果	Fortify SCA Coverity Prevent KLEE
动态检测	1. 模糊测试 2. 污点分析	1. 提供特别构造的数据或测试用例作为程序的输入 2. 观察程序的运行过程	SPIKE TaintCheck Flayer Dytan

4.1.1 静态漏洞检测技术

程序漏洞静态检测是通过扫描程序的源代码或者二进制代码, 找到符合漏洞特征的代码片段的过程。静态漏洞检测的首要任务是根据各种漏洞的特点, 建立相应的漏洞特征库, 如缓冲区溢出漏洞特征库、 格式化字符串漏洞特征库以及整数漏洞特征库等, 然后根据建立的特征库对程序进行静态地分析, 检测出程序中存在的漏洞。检测的工作流程包括程序源代码模型创建、漏洞特征库建立、漏洞判定, 以及最后的结果分析等几个步骤。检测使用的技术有数据流分析(Data-Flow Analysis)和符号执行(Symbolic Execution)等。主要的检测工具有 Fortify SCA[32]、Coverity Prevent[33]、KLEE[34-35]等。其中, Fortify SCA 是一个针对程序源代码的静态漏洞检测工具, 它通过分析程序可能会执行的各种路径, 从源代码层面上识别程序中的漏洞, 并对识别出的漏洞提供完整的分析报告。在检测的过程中, Fortify SCA 首先将程序的源代码转化成中间表达形式, 然后利用内置的五种分析引擎(数据流引擎、语义引擎、结构引擎、控制流引擎、配置引擎)和相关的漏洞特征库对程序的中间表达式进行静态分析, 从而将程序中符合漏洞特征(如缓冲区溢出漏洞)的代码段挖掘出来, 辅以人工分析最终确定漏洞的位置。Coverity Prevent 是由 Coverity 公司开发的源代码静态漏洞检测工具, 最初源于 Engler 和学生共同开发的 xGCC 系统。它采用的基本分析模型是将程序漏洞建立为一个状态机模型。状态机模型中的状态表示程序中数据的相关安全性质。在分析过程中, Coverity Prevent 的分析引擎对程序的各种执行路径进行静态地遍历, 根据当前运行语句的语义驱动状态机不断的运行, 当程序的状态处于一个预先定义好的非安全状态时, 就认为发现了一个疑似的漏洞, 分析引擎将漏洞的特征、触发路径等信息报告给用户, 用户在漏洞特征库的帮助下, 最终确认漏洞。KLEE 是斯坦福大学的 Cristian Cadar、Daniel Dunbar、Dawson Engler 三人合作开发的运行在 Linux 系统上的一个符号执行工具[35]。它可以通过符号执行技术自动生成测试用例。其在分析程序生成测试用例的同时, 也利用符号执行和约束求解技术在程序的关键点上对符号的取值范围进行分析, 检查符号的取值范围是否在规定的安全范围之内。如果发现符号的取值范围不能满足预先定义好的安全规则, 则可以判定程序中存在漏洞。

4.1.2 动态漏洞检测技术

动态漏洞检测技术通过在真实的环境或虚拟的环境中实际运行程序, 记录程序运行时的轨迹, 然后再进一步解析程序运行过程中的数据操作和函数之间的调用关系等运行时的信息, 用以检测出程序中存在的漏洞。动态漏洞检测代表性技术有模糊测试(Fuzzing)和动态污点分析(Dynamic Taint Analysis, DTA)。模糊测试技术最初被称为随机测试(Random Testing)[36], 后来在 B.Miller 教授的课堂教学中被称之为模糊测试。模糊测试的基本思想是: 向待测的程序提供大量的经过特殊构造的或是随机的数据作为程序的输入, 监视程序在运行过程中发生的异常情

况并记录导致发生异常情况的输入数据, 在人工分析的帮助下, 定位程序中漏洞的位置。模糊测试技术的典型工具是 D.Aitel 等人于 2002 年发布的 SPIKE[37]模糊测试框架。SPIKE 模糊测试框架实现了一种基于块(Block-based)的测试方法, 用来测试基于网络的应用程序。SPIKE 除了可以自己产生随机的数据作为程序的输入外, 本身还带有一个包含了各种特殊数据的库。库中的数据可以使得含有错误的程序发生故障, 从而触发存在于程序中的漏洞。此外, SPIKE 框架还内置了一些预定义的函数, 这些函数可以帮助生成常见的各种格式的数据用以作为程序的输入。动态污点分析技术[38]是近年来逐渐流行的一种动态漏洞分析技术。该分析技术在程序运行的过程中, 对程序的数据流或控制流进行实时地监控, 从而实现对数据在内存中的传播过程进行跟踪和检测, 以此来发现程序中存在的漏洞。动态污点分析技术分为基于数据流的分析和基于控制流的分析。基于数据流的分析主要是通过把来自外部的数据标记为污点数据, 并跟踪这些污点数据在内存中显式传播的过程, 以此来检测程序中的漏洞, 主要的检测工具有 TaintCheck[39] 和 Flayer[40]。基于控制流的动态污点分析是对数据流分析的有力补充。在把外部数据标记为污点、污点数据显式传播跟踪的基础上, 通过分析程序的控制流来建立程序的控制流图, 并设计特定的算法实现对程序的隐式流(Implicit Flows)传播过程的监控和分析, 用以发现程序中的漏洞。主要的工具有 DTA++[41] 和 Dytan[42]。

4.2 漏洞阻止技术

不管是漏洞的静态检测还是动态检测, 当程序实际投入运行时, 还是存在很多的漏洞被攻击者所利用, 这也从另一个侧面说明漏洞检测也不是完美的, 并不能解决所有的问题。而且, 漏洞检测所采用的技术本身也存在一定的局限。如符号执行技术存在路径爆炸的问题[43], 因此检测时不可能覆盖到所有的路径, 存在一定的漏报(False Negatives)。模糊测试技术需要构造大量的异质数据来触发程序中存在的漏洞, 但实际情况是不可能完全地构造出所有的异质数据触发程序中的漏洞, 这同样存在着漏报的问题。除此而外, 漏洞动态检测过程中的性能问题也困扰着这些技术的实际部署。因此, 众多的研究人员把目光投向了在程序的实际运行过程中阻止漏洞被利用的研究方向。漏洞阻止技术指的是在程序的运行过程中, 运用某种策略或措施阻止程序中存在的漏洞被利用来实施攻击的技术。这里需要注意的是, 所谓的程序运行过程是包括程序的编译、链接、装载和运行这几个阶段的。在程序的运行过程中阻止漏洞的利用, 具有漏洞检测所不具备的优点。首先, 这一过程是一个实际运行的过程, 会产生许多意想不到的问题。而这些曝露出来的问题使得漏洞阻止技术逐步变得更加完善和实用。其次, 真实的运行过程可以检验漏洞阻止技术运行时真实的性能消耗, 为漏洞阻止技术的不断优化提供不同的途径。最后, 真实的运行过程也可以检验漏洞阻止技术的兼容性能力, 使之适合运行于多种平台和不同的环境。当前, 程序运行过程中的漏洞阻止技术主要有基于程序控制流完整性的技术和基于程序多样性的技术。表 3 显示了主要的漏洞阻止技术。

表 3 漏洞阻止技术

类型	技术	流程	工具
控制流完整性	1. 粗粒度的控制流完整性 2. 细粒度的控制流完整性	1. 创建控制流图 2. 根据控制流图检查跳转指令	Ropecker CCFIR BinCFI
程序多样性	1. 指令层次多样性 2. 基本快层次多样性 3. 函数层次多样性 4. 程序级别多样性	重新排序程序中的指令和内存布局	ICR STIR ASLR ASLP TASR

4.2.1 基于程序控制流完整性的技术

控制流完整性(Control Flow Integrity, CFI)[44]要求程序严格按照程序的控制流图来运行, 不允许出现偏离控制流图的情况出现, 若出现偏离控制流图的执行流程, 可以据此断定程序受到了攻击, 出现了异常情况。实现程序控制流完整性的主要方式是创建程序完整的控制流图。然后通过分析程序的控制流图, 对程序中的直接跳转指令和间接跳转指令进行监控, 观察程序运行过程中是否出现异常的跳转现象。控制流完整性可以分为细粒度(Fine-grained)和粗粒度(Coarse-grained)两类。细粒度的控制流完整性要求严格控制每一个跳转指令的目的地, 不允许出现任何偏离既定执行流程的情况。粗粒度的控制流完整性则比较宽松, 是把一组类型相似的跳转指令归为一类进行检查, 用以降低程序的性能开销, 但这种方法会导致较低的安全性。从理论上来说, 控制流完整性是一种确定性的(Deterministic)阻止攻击的方法[13], 只要程序的执行流程偏离了预定的轨道, 就可以判定程序受到了攻击。然而, 在实现实用的利用控制流完整性阻止攻击的过程中, 面临着诸多挑战: (1)获取程序完整的控制流图是一个很困难的问题。程序的运行过程是一个不断判断与跳转的过程。对一些实施简单计算的程序来说, 判断与跳转关系简单, 可以通过分析源代码静态的获取完整的控制

流图。而对于那些复杂的、巨大的程序的来说, 静态获取的控制流图并不完整, 不能反映程序真实的运行过程。而在程序运行的过程中去分析程序获取完整的控制流图, 则需要考虑程序运行过程中出现的所有情况, 而且性能消耗巨大, 这些问题都给获得完整的控制流图带来了巨大的挑战。这样, 由于没有完整的控制流图, 细粒度的控制流完整性技术几乎不可能实现; (2)因为不能获得完整的控制流图, 所以粗粒度的控制流完整性就使用了一个简化的、精确度不高的粗糙控制流图。这种折中方案虽然在性能消耗上有所缓解[45-46], 但阻止攻击的能力不是很强, 而且已经被很多的攻击所绕过[27, 47-48]。在这种情况下, 广大研究人员把研究的兴趣转向了程序多样性技术。与控制流完整性相比, 程序多样性是一种非确定性的(Probabilistic)阻止攻击的方法[13]。该方法通过多次改变程序的内存空间布局使得攻击者不能通过程序中的漏洞获取确切的攻击信息, 或者获取的信息由于内存空间布局的改变而失去利用价值, 进而阻止利用漏洞来实施攻击。程序多样性的实现门槛较低, 而且已经有成熟实用的技术被广泛使用[49]。因此, 本文将以程序多样性为主介绍漏洞阻止技术。

4.2.2 基于程序多样性的技术

程序多样性(Program Diversity)来源于程序进化的理论[50]。在密码学里,“信息论之父”香农(C. E. Shannon)提出了设计密码体制的两种基本思想: 扩散(Diffusion)和混淆(Confusion)[51]。这两种思想的根本目的就是为了增加密码分析的难度, 使密码分析的成本远远大于攻击成功后所获得的利益, 从而阻止对密码的攻击。在程序进化理论里也应用了这两种思想。通过对程序实施指令替换、指令重新排序、变量替换、垃圾代码插入、指令编码加密等等措施, 为程序加入大量的干扰信息, 使程序以不同的形态出现, 从而加大攻击者攻击的难度。而程序多样性则是对程序进化理论进一步的具体实现。程序多样性是指通过某种手段使程序的组成部分, 如指令、基本块、函数等在程序运行过程中呈现不同的状态, 以此来增加攻击者实施攻击的难度, 进而阻止攻击。程序多样性是一种非确定性的阻止漏洞被利用的方法。这种方法通过改变程序组成部分的状态, 使得攻击者先前通过某种手段(如利用内存错误漏洞和格式化字符串漏洞)获得的关于程序的信息变得不确定, 从而不能实施攻击。在对程序实施多样性的过程中, 一个很重要的原则是: 源程序和实施多样性之后的程序对于相同的输入必定会有相同的输出结果。也就是, 多样性不应破坏程序的语义信息。程序多样性按照实施对象的粒度来划分, 可以分为指令级别、基本块级别、函数级别以及程序级别。

在程序内实现指令级别的多样性: (1)用一条指令代替另一条指令或一个指令序列后重新排列指令的顺序[52-54]; (2)随机化程序中寄存器的分配[53-54]; (3)指令中插入一些垃圾代码, 如空操作指令(NOPs)[53, 55-56]。这种多样性的影响程度在一个程序的基本块内部, 可以阻止细粒度的利用漏洞实施的攻击, 如 CIA, 但前提是不能泄露这种多样性的实现细节。

基本块作为程序执行的一个基本单位, 包含着一个顺序执行的语句序列, 其中只有一个入口和一个出口, 入口就是其中的第一条语句, 出口就是其中的最后一条语句。因此, 基本块级别的多样性可以采用重新排列基本块的顺序方式进行。重新排列基本块需要在有衔接顺序的基本块之间加入一条跳转指令[55-57], 或一个分支函数[52, 57], 用以保证随机排列后的基本块仍能按照程序控制流图的正确顺序被执行。因此, 这种情况下也需要保证实现的细节不被泄漏。

在函数的级别上实行多样化, 首先从栈的实现上来进行。这很容易理解, 因为很多的攻击都是从覆盖栈上函数的返回地址开始的。对栈进行多样化, 可以从对程序中存放于栈上的变量重新排序、栈的增长方向反转以及创建不相邻的栈帧来实现。总之, 目的就是打破传统的栈结构, 使得攻击者不能通过预测栈的位置来实施攻击[55, 58-59] 。其次, 可以通过把一个函数分成若干个函数, 或者对已经存在的函数进行重新排序来实现函数级别的多样化[60-61]。这样, 通过程序栈的重新创建和打乱函数的排列顺序, 就可以阻止攻击者通过利用内存错误等漏洞而实施的攻击。

在程序的级别上实施多样化, 主要有地址空间随机化(Address Space Layout Randomization, ASLR)的技术[49]。 这也是目前唯一的具有实用价值的多样性技术, 当前部署在多个操作系统上。在 ASLR 技术中, 程序编译之后生成位置无关的代码(Position-independent Code, PIC)。这样, 每次当程序被装载时, 就可以把程序的某些段, 如栈段、堆段等放在随机的虚拟地址空间位置内, 以此来阻止 CIA、RILC 和 ROP 攻击。另外两种程序级别上的多样性技术是程序编码随机化技术(Program Encoding Randomization, PER)[62] 和数据随机化技术(Data Randomization, DR)[59]。在程序编码随机化技术中, 用一个编码之后的程序代替源程序。所采用的最简单的编码方法可

以是对源程序实施异或操作生成一个多样性的程序。当程序运行时, 对其进行相反的操作即可。此外, 也可以对程序的代码布局和代码地址实施编码随机化。在数据随机化技术中, 随机化的对象不是程序中的指令或者地址, 而是具有一定语义的变量或常量, 主要是为了阻止非控制数据攻击。可以采用的随机化方式有:程序中的变量重新排序、程序中的结构类型对象或类类型对象的布局随机化以及堆布局随机化。

5 当前主要的程序多样性技术对比分析

我们在第四节介绍过, 漏洞阻止技术中的控制流完整性技术由于程序完整控制流图的缺乏, 因此使用了一个简化的控制流图, 虽然性能上有所提升, 但安全能力却不足。而程序多样性技术通过对程序实施多样性可以有力地阻止攻击者实施的攻击。而且, 程序多样性技术中的 ASLR 技术已经被广泛的应用到当今的多种操作系统上。然而, 这些程序多样性技术却有一个共同的弱点, 那就是在程序运行的过程中只对程序实施一次多样性。这样, 对于那些运行时间较长的 Daemon Server 类型的程序来说(如 Apache 和 Nginx Web Server), 几乎没有能力阻止 JIT(Just-In-Time)类型的控制流劫持攻击(如 JIT-ROP 和 Blind-ROP)。因此, 研究人员开始对程序进行运行过程中的多次多样性的研究工作。到目前为止, 最具代表性的多次多样性的技术有 ASR[56]、TASR[63]、RUNTIMEASLR[64]和 Morula[65]。下面我们就从这四种技术的多样性实现方式、实施多次多样性的时机选择、技术应用的平台以及性能消耗等方面对它们进行对比分析, 并指出它们存在的主要问题。

就我们所知, Giuffrida 等人提出的技术 ASR 可以说是第一次真正意义上的多次多样性的随机化技术: 在程序运行的过程中, 对程序的内存布局进行多次的改变。在 ASR 技术中, 把操作系统不同的子系统隔离成事件驱动的组件, 以一个个独立运行的进程来表示。这样, 可以对不同的子系统进行可选次数的随机化操作, 避免由于随机化次数过于频繁而造成的性能消耗的增加。而且在实际的测试中, 采用这种设计方式的 ASR 技术的性能消耗不超过文献[13]的性能消耗标准, 平均为 5%。ASR 技术在具体的实施过程中, 每一个用户都将接收到一个完全相同的二进制版本的操作系统和操作系统中各个子系统组件的 LLVM 字节码文件。这些子系统的字节码文件被存储在用户磁盘上受保护的分区里, 任何用户都不能访问。在操作系统运行的过程中, 这些字节码文件由一个后台进程周期性的进行处理, 生成不同的随机化变体(Variant)。然后, 一个随机化管理器组件使用这些变体来随机化每一个子系统组件, 达到随机化的目的。随机时机的选择是在这些子系统组件进行状态转变时开始的, 对用户完全透明, 并且不需要系统或进程重新启动。该技术第一次提出了对进程内存布局进行运行中的多次多样性的随机化, 为利用程序多样性技术阻止控制流劫持攻击提出了新的研究思路。但是, 该技术也存在一些问题。主要的问题在于该技术适用于基于组件的操作系统架构, 如微内核架构的嵌入式操作系统 L4 和 QNX, 以及基于软件的组件隔离方案的研究型操作系统 Singularity, 并不适用于当今整体架构的商业操作系统(如 Windows 和 Linux)。因为在整体架构的操作系统中, 各个系统组件之间缺少明确的界限, 这使得只随机化单个组件的方案根本行不通, 限制了该技术在这些商业操作系统上的使用。

在 Bigelow 等人提出的多次多样性的内存布局随机化技术 TASR 中, 首先通过修改 GCC 编译器, 实现程序相关信息的自动化收集, 然后把收集到的信息以 Segment 的形式注入到程序的 ELF(Executable and Linkable Format, ELF)文件中。程序在运行的过程中, 随机化组件根据这些信息来对程序的内存布局实施多样性随机化。收集到的信息中, 最主要的是一些代码指针的内容。当对程序实施随机化时, 这些指针需要被更新为新的内容, 否则就会阻止程序的正常运行。同时, TASR 技术的随机化时机选择在程序的一次输入和紧跟着的输出之间进行。因为程序中的输入与输出之间最容易被攻击者所利用。此外, TASR 技术完全以普通程序为研究对象, 所以该技术完全适用于普通的应用程序, 而由于随机化的实施带来的性能消耗却很小, 经过测试平均只有 2.1%。但是, TASR 技术也存在一定的问题。首先从它的设计初衷来说, TASR 技术被设计用来保护编译执行的二进制代码。所以对那些解释执行的代码并不具有保护能力, 从而也就不能阻止针对这些代码实施的 JIT 类型的攻击。其次, TASR 技术不能自动处理不符合 C 标准(C Standard)的代码。所以如果程序中含有此类代码, 则需要开发人员采用手动处理的方式对这些代码进行转化。这无形中增加了开发人员的工作量, 也不利于该技术的推广。最后, 由于 TASR 技术对程序的内存布局实施整体搬迁方式的多样性随机化, 因此它不能阻止利用相对寻址方式实施的攻击, 这也从另一个侧面说明 TASR 是一个粗糙的多样性技术。

在 2016 年, Lu 等人提出了针对 Daemon Server 类型程序的 RUNTIMEASLR 多样性随机化方案。一般情况下, Daemon Server 类型的程序开始运行时, 父进程通过克隆自身的方式创建若干个子进程来处理用户的请求。由于子进程是通过克隆的方式被创建的, 所以子进程之间或子进程与父进程之间具有相同的状态信息和内存布局。这就给攻击者创造了攻击的机会。攻击者可以通过 Blind-ROP 攻击重复不断地探测子进程的内存布局, 进而获得需要的攻击信息来实施攻击。为了阻止这种类型的攻击, 在 RUNTIMEASLR 多样性随机化方案中, 每当父进程创建一个子进程时, RUNTIMEASLR 就随机化子进程的内存布局, 这就使得子进程之间和子进程与父进程之间的内存布局不相同, 攻击者也就不能获得完整的内存布局信息来实施 Blind-ROP 攻击。在具体的实施过程中, RUNTIMEASLR 把进程中的指针当作污点数据, 设计若干污点扩散策略。当 RUNTIMEASLR 对进程内存布局实施多样性随机化时, 就根据污点扩散策略来更新这些指针, 从而保证多样性后的进程能顺利的继续运行。经过测试, 采用这种多样性方式的 RUNTIMEASLR 的性能消耗只有 0.51%, 在四种多样性技术中是最小的。但是, RUNTIMEASLR 多样性技术存在的问题也很明显, 该技术在子进程运行的过程中, 不会对子进程的内存布局进行持续不断地多样性, 因此它不能阻止针对单个进程的 JIT-ROP 攻击。

Lee 等人提出的多样性随机化方案 Morula 类似 RUNTIMEASLR, 但 Morula 刚好与 RUNTIMEASLR 相反, Morula 随机化的是父进程的内存布局, 而不是子进程的内存布局。这是由 Morula 应用的场景决定的。Morula 应用于 Android 系统。Android 系统启动时, 由 init 进程创建一个名为 Zygote 的父进程。系统中若干服务进程和应用程序进程都是通过克隆 Zygote 进程来产生的, 这样就加快了 Android 系统的启动速度, 符合移动设备的快速启动要求。但是, 这种情况非常容易被攻击者所利用。因为由 Zygote 克隆而来的子进程和 Zygote 进程具有相同的内存布局。攻击者只要知道一个进程的内存布局, 就可以知道几乎所有进程的内存布局。从而很容易的把从一个进程上获得的信息应用到其他的进程上, 开始实施攻击。为了应付这种情况, Morula 提出了 Zygote 内存布局池(Pool of Memory Layout, PML)的设计思想。PML 中存放着 Zygote 进程不同的内存布局。当需要通过克隆 Zygote 进程的方式创建子进程时, Morula 随机地从池中选择一个内存布局, 以该内存布局为模板创建子进程。这样系统中的各个进程将会具有不同的内存布局, 从而也就阻止了上面描述的攻击。经过测试, 采用这种设计方式的多样性技术的性能消耗率为 2%。该性能消耗率对于移动应用平台来说显得有点大, 因此需要对该技术实施进一步的优化, 降低其性能消耗率。最后, 这种方案同样具有类似 RUNTIMEASLR 多样性方案的问题, 不能对进程实施持续不断地多样性操作, 因此同样也不能抵御 JIT-ROP 攻击。

表 4 实时的程序多样性技术(2012-2016)

技术	特点	随机时机	应用平台	性能消耗	局限
ASR	随机子系统组件	进程状态被改变的时候	基于组件的操作系统架构	5%	不能应用于整体架构的商业操作系统
Morula	随机化父进程	子进程被创建的时候	Android 系统	2%	不能进行持续的多样性
TASR	随机化每一个进程	程序输入和输出的时候	Linux 系统	2.1%	不能阻止 JIT-ROP 和相对寻址方式的攻击
RUNTIMEASLR	随机化子进程	子进程被创建的时候	Linux 系统	0.51%	不能进行持续的多样性

表 4 对这种四种技术进行了对比总结: (1)四种技术实施多样性的对象不同。其中, ASR 对系统的组件实施多样性, TASR 对任何进程实施多样性, Morula 对父进程实施多样性, RUNTIMEASLR 对子进程实施多样性; (2)四种技术实施多样性的时机不同。ASR 是在进程状态被改变的时候, TASR 是在程序的输入和输出之间进行, Morula 与 RUNTIMEASLR 相同, 是在进程被创建的时候进行; (3)四种技术应用的平台不同。ASR 应用于基于组件的操作系统, TASR 与 RUNTIMEASLR 应用于 Linux 操作系统, Morula 应用于 Android 系统; (4)四种技术的性能消耗率中, 最大的是 ASR, 有 5%。其次是 TASR 和 Morula, 分别是 2.1%和 2%。最后是 RUNTIMEASLR, 消耗率为 0.51%; (5)四种技术分别具有不同的局限。如 ASR 不能应用于整体架构的商业操作系统, Morula 与 RUNTIMEASLR 不能对进程实施持续的多样性, 而 TASR 则不能阻止 JIT-ROP 攻击和相对寻址方式的攻击。从这些不同之处我们可以看出, 实现程序多样性

的技术是多种多样的, 但它们的目的却是相同的, 那就是通过各种各样的程序多样性技术来阻止针对程序实施的各种类型的攻击。但是, 我们也不要忘记了一点, 在实现多样性技术的时候, 要充分考虑技术的适用范围和性能消耗的问题。只有这样, 程序多样性技术才能被广泛地应用于实践当中。

6 分析与讨论

程序中存在的漏洞是各种攻击事件的根源, 并且这种根源由于程序本身固有的特性而无法避免。我们在第四节已经提及, 针对漏洞的缓解措施有两种: 漏洞检测技术和漏洞阻止技术。这两种技术在漏洞缓解的过程中扮演着极其重要的角色, 可以说, 没有这些技术的帮助, 我们的程序都不可能安全的运行。然而, 这些技术本身也各自有优缺点。

静态的漏洞检测过程中使用的数据流分析技术虽然具有较强的分析能力, 但其分析效率较低, 代码中的过程间的分析以及优化算法非常复杂, 使得编码人员的工作量巨大, 容易出错且效率较低; 符号执行技术生成的测试用例具有很强的针对性, 具有较高的覆盖率, 可以发现程序中使用其他技术不容易发现的问题。但是符号执行技术有一个固有的顽疾, 就是路径爆炸问题, 在对复杂的程序进行符号执行时, 需要覆盖的路径几乎呈指数级的增长, 使得对问题路径的求解几乎不可能。除此而外, 静态的漏洞检测技术需要程序的源代码, 因此在程序源代码不可得的情况下, 几乎不可能执行静态检测。

动态的漏洞检测技术克服了静态检测技术需要程序源代码的缺点, 以程序的二进制可执行程序为分析对象, 大大地提高了漏洞分析技术应用的范围。此外, 动态漏洞检测技术中的污点分析还可以通过对污点数据扩散传播的跟踪监控, 快速地找到和输入数据有关的程序漏洞; 模糊测试技术的计算量较小, 实施的门槛较底, 目前已被推广应用到软件测试行业用来检测程序中存在的漏洞。但是, 动态的漏洞检测技术也有不足之处。如(1)动态污点分析容易受到编译器优化的干扰, 使得输入的数据不能起到相应的作用; (2)动态污点分析在对程序进行污点分析时, 还需要构造程序对应的污点传播树。这是一种结构复杂的树, 一般情况下需要人工的干预, 无形中增加了编码人员的工作强度; (3)动态污点的实现需要动态插桩或虚拟技术的支持, 给其实现提高了门槛; (4)由于动态污点分析中动态监测的方法不能覆盖程序所有的执行路径, 因此漏洞检测的漏报率比较高; (5)动态污点分析由于需要结合虚拟机监控以及程序分析等辅助技术, 从而加大了程序运行时的开销, 降低了运行时的效率; (6)模糊测试技术的测试用例覆盖率较低, 程序中测试用例覆盖不到的部分的漏洞很难检测出来, 也容易产生漏报的问题。然而, 尽管动态漏洞检测技术存在这些不足之处, 但由于其不需要程序源代码的特点, 使得研究人员对其抱有很大的研究热情, 最近的一篇文献[64]就对动态漏洞检测技术中的污点分析给予了充分的应用。

漏洞阻止中的细粒度控制流完整性技术需要完整的控制流图, 在实现上需要在程序运行时, 检测程序的执行流程是否在程序控制流图的范围之内, 据此识别是否受到了攻击。具体的做法是采用代码插桩的方法, 在代码中每一个控制转移指令前插入对转移目的地的检验代码, 判断目的地的合法性。这是一种非常严格的做法, 为了做到这一点, 需要对程序中每条控制转移指令都要进行检查, 并确保每条控制转移指令只能转移到它合法的目的地。为了达到更强的防御效果, 则还需要对程序的上下文进行检查, 考虑相关的语义信息。但是, 这种检查尽管增强了系统的安全性, 但其性能消耗实在过大, 而且还需要完整的程序控制流图, 这就使得细粒度的控制流完整性难以实现。粗粒度的控制流完整性技术采用宽松的策略, 在技术上得以实现控制流完整性的思想, 如CCFIR(Compact Control Flow Integrity and Randomization, CCFIR)[66] 和BinCFI[67]。但是这种实现, 正如我们前面所提到的, 是一种折中的方案, 并不能达到非常强的安全程度, 但是在性能消耗上却有不俗的表现, 平均的性能消耗都不超过10%, 完全符合实际部署的性能标准。

程序多样性阻止利用漏洞实施攻击的思想源自于低概率事件, 所以它是一种非确定性的漏洞阻止技术。对程序实施多样性, 可以降低攻击者通过漏洞获得程序信息的能力, 或者获取的信息无助于攻击的实施。在程序多样性的实施过程中, 如果“熵”足够大, 那么攻击成功的概率可以很低, 如2^{-64}[64]。在这种情况下, 程序多样性几乎成为一种确定性的漏洞阻止技术。另外, 在实现的过程中需要重点关注代码中指令的问题, 特别是动态的程序多样性。在对程序实施多样性之前, 需要把程序中相关的指令, 如跳转指令以某种形式保存起来, 在进行多样性时进行适当的转化以保证程序运行流畅[68]。最后, 在多样性过程中, 对指针引用关系的处理也是一个重点。因为, 程序多样性之后, 程序中某些指令的顺序, 或内存空间布局都要发生变化。在这种情况下, 原有的指针引用关系势必要发生变化, 这些引用关系需要进

行重新定位后才能保证程序的正常运行。因此, 在某种意义上对指针引用关系的处理决定了程序动态多样性的成功与否。这是程序多样性中一个极具挑战性的问题。因为尽管各种体系架构指令集中的指令条数是有限的。但是, 如果包含指针的指令之间进行了算术运算, 则会产生类似于污点数据传播的指针扩散, 进而指针之间的引用关系也会扩散, 变得非常复杂。因此, 为了顺利地对程序实施多样性就需要对这种指针扩散进行完整的追踪, 在追踪的过程中不能有指针漏报或误报的情况发生, 否则任何一种情况出现, 程序都会发生错误, 最终停止运行。然而, 尽管存在这样的挑战, 程序多样性仍不失为一种非常有效的阻止漏洞攻击的优秀技术, 被广大的研究人员进行不断地研究和改进。

7 未来的研究方向

当前, 漏洞缓解技术从最初被动的漏洞检测到主动的漏洞阻止, 走过了一个不断前进的过程: 从静态的漏洞检测到动态的漏洞检测; 从程序运行前的一次多样性到程序运行过程中的多次多样性。但是, 漏洞利用和漏洞阻止却像两个参加比赛的运动员一样, 相互不断地超越彼此: 新的漏洞利用出现, 就会紧接着出现新的防御方法; 而新的防御方法的出现, 不久又会被新的漏洞利用所绕过。因此, 漏洞缓解技术的研究是一个永恒的过程, 只有起点而没有终点。

在前面分析的漏洞阻止技术中, 基于程序控制流完整性的技术由于存在一些困难(如程序完整控制流图的获取), 其发展的速度不是很快, 但基于程序多样性的技术却发展迅速, 涌现出了很多新型的技术。然而, 尽管程序的多样性发展的很迅速, 但还是有一些不足之处。一方面, 很多的程序多样性实现方案只是对程序进行了一次多样性, 在程序运行的过程中却不再进行多样性, 这对类似于 JIT-ROP 的攻击几乎没有任何抵御能力, 如 ASLR。另一方面, 最近的程序多样性实现方案只是对程序的代码部分进行了多样性改变, 而对程序的数据部分、动态共享库部分却没有涉及, 如 TASR。这使得程序不能抵御针对非控制数据的攻击, 也不能抵御攻击者利用程序动态共享库中的漏洞实施的攻击。因此, 鉴于以上的分析, 我们非常有必要进行程序运行过程中持续的和全面的内存布局多样性技术研究, 进一步完善程序多样性技术。但是, 实现持续的和全面的程序内存布局多样性面临着如下挑战: (1)程序两次内存布局多样性之间的时间间隔的选取, 即第一次内存布局多样性后, 何时再开始第二次的内存布局多样性; (2)全面的程序内存布局多样性的实现方式。对于面临的第一个挑战, 大家可能认为很容易解决: 随机选取时间间隔即可。但是, 通过对现存的多样性技术进行研究, 我们不难发现, 对程序进行的每一次的多样性, 都会或多或少地带来程序性能消耗的增加。而且, 对程序内存布局进行的多样性越频繁, 则性能消耗的增加也就越大, 反之亦然。但是, 如果对程序内存布局进行不是很频繁的多样性, 却又不能获取足够的安全保障。因为攻击者完全可以利用两次多样性之间的空隙来实施攻击。因此, 两次多样性之间时间间隔的选取是一个值得研究的问题, 这涉及程序的性能消耗和安全能力的全面考量, 同时也极大地影响着多样性技术能否在实践中被部署应用。对于第二个挑战, 现今的多样性技术通常都是部分多样性进程的内存布局, 例如 TASR 只是多样性了程序的代码部分。通常情况下, 一个运行中的程序基本上包含以下五个部分: 代码段、数据段、堆段、栈段和内存映射段(映射程序运行过程中需要的动态链接库)。对程序实施全面的内存布局多样性, 意味着每一次的多样性, 都需要对这五个部分进行多样性改变。这是一个很困难的问题。在 TASR 中, 之所以只对程序的代码部分进行多样性, 作者给出的理由是: 性能, 以及精确地跟踪数据引用位置的难度(Performance, and The Difficulties in Precisely Tracking Data Reference Locations)[63]。在程序的内存布局多样性中, 当程序的代码部分被多样性时, 其中的代码引用指针需要被更新以指向新的位置。这个问题很容易解决, 一方面程序的代码在运行过程中是不再发生变化的(注意, 这里限于文章篇幅的原因, 我们只讨论预编译的二进制程序(Precompiled Binary), 而对于解释型的代码(Interpreted Code)不涉及)。另一方面, 代码部分中的代码引用指针的数量也不是很大。所以, 可以进行精确的跟踪并进行更新。这虽然对性能有所影响, 但还是在可以接受的范围之内。而对于其中的数据引用指针(访问数据段内的内容)难度却是很大。一般情况下, 程序中数据引用指针的数量有几万个, 有的程序可能包含上百万个。此外, 这些数据引用指针有时候需要进行算术运算以得出所引用数据的位置。这无形中增加了跟踪这些指针的难度。因此, TASR 从性能和跟踪难度方面的考虑, 而没有对程序的数据部分进行多样性。而在这里, 如果要对程序进行全面的内存布局多样性, 这不仅要涉及到程序的代码部分, 还要涉及程序的数据部分、堆段部分、栈段部分以及内存映射部

分。可以想到这是一个具有很大难度的问题。但是, 这也是一个非常值得研究的问题。因为如果实现了对程序实施全面的内存布局多样性, 那么基于程序的漏洞而实施的攻击完全可以被阻止。从这个角度看, 这是一个非常诱人的前景和目标, 是值得我们付出任何的努力去实现它的。

8 总结

几十年来, 学术界和工业界提出了众多的漏洞缓解措施。然而, 到目前为止还没有一种最终可行的解决方案。在这种情况下, 我们只能一方面不断地完善现有的技术, 使之能用于实际的应用环境中。另一方面, 我们还需要不断地开创新的研究方向, 以应对未知的漏洞攻击。当前, 漏洞阻止的两种技术: 控制流完整性和程序多样性作为漏洞防御主要的研究方向, 得到了学术界和工业界普遍的重视。控制流完整性作为一种确定性的防御技术, 在克服自身的缺点, 并在当前科学技术的支持下, 可以接近于做到对程序的每一次的控制流转移进行检查, 阻止控制流劫持方面的攻击。程序多样性作为一种非确定性的漏洞阻止技术, 充分体现了"善攻者, 敌不知其所守, 善守者, 敌不知其所攻"的军事思想。通过不断地改变程序的组成部分或内存布局, 让攻击者无从下手实施攻击, 最终阻止攻击者通过程序中的漏洞实施攻击。在程序多样性中, 最重要的一点是要增加程序的"熵", 只要"熵"足够大, 程序多样性技术也就是一种确定性的漏洞阻止技术。目前的程序多样性技术已经克服了传统 ASLR 的限制, 被应用到 64 位的系统上, 使得程序的"熵"很大。但是, 根据我们的预测, 在计算能力不断前进的今天, 不久的未来, 这样的"熵"也会变得很低。此外, 阻止漏洞缓解技术应用于实践的另一个障碍是性能方面的考虑。当前许多安全能力很强的漏洞缓解技术由于实施中的性能消耗太大, 而不能被部署到实际的应用环境中。所以, 我们需要在漏洞缓解技术的效果和效率之间做一个全面的权衡, 达到一个合适的平衡点。

参考文献

[1] S. Wu, *Software vulnerability analysis technology*. Science Press, 2014.

[2] F. Pfenning, "Lectures notes on type safety:foundations of programming languages," Carnegig Mellon University, Pittsburgh, Lecture 6, 2004. [Online]. Available: www.cs.cmu.edu/~fp/courses/15312-f04/handouts/06-safety.pdf

[3] S. C. Johnson and B. W. Kernighan, "The programming language b," NJ:Bell Labs, Murray Hill, Computing Science Technical 8, 1973.

[4] M. Richards and C. Whitby-Strevens, *BCPL: the language and its compiler.* Cambridge University Press, 1981.

[5] ISO/IEC, *Programming Languages-C,Third Edition(ISO/IEC 9899: 2011).* Geneva,Switzerland:International Organization for Standardizarion, 2011.

[6] A. One, "Smashing the stack for fun and profit," *Phrack magazine,* vol. 7, no. 49, pp. 14–16, 1996.

[7] J. Viega and G. McGraw, *Building Secure Software: How to Avoid Security Problems the Right Way, Portable Documents.* Pearson Education, 2001.

[8] J. Pincus and B. Baker, "Beyond stack smashing: Recent advances in exploiting buffer overruns," *IEEE Security & Privacy,* vol. 2, no. 4, pp. 20–27, 2004.

[9] T. Committee *et al.,* "Tool interface standard (tis) executable and linking format (elf) specification version 1.2," *TIS Committee,* 1995.

[10] M. Graff and K. R. Van Wyk, *Secure coding: principles and practices.* "O'Reilly Media, Inc.", 2003.

[11] B. Jack, "Vector rewrite attack: Exploitable null pointer vulnerabilities on arm and xscale architectures," *White paper, Juniper Networks*, 2007.

[12] D. Brumley, T.-c. Chiueh, R. Johnson, H. Lin, and D. Song, "Rich: Automatically protecting against integer-based vulnerabilities," *Department of Electrical and Computing Engineering*, p. 28, 2007.

[13] L. Szekeres, M. Payer, T. Wei, and D. Song, "Sok: Eternal war in memory," in *Security and Privacy (SP), 2013 IEEE Symposium on.* IEEE, 2013, pp. 48–62.

[14] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, "Sok: Automated software diversity," in *2014 IEEE Symposium on Security and Privacy.* IEEE, 2014, pp. 276–291.

[15] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer, "Non-control-data attacks are realistic threats." in *Usenix Security,* vol. 5, 2005.

[16] R. Wojtczuk, "The advanced return-into-lib (c) exploits: Pax case study," *Phrack Magazine, Volume 0x0b, Issue 0x3a, Phile# 0x04 of 0x0e,* 2001.

[17] P. Baecher and M. Koetter, "Getting around non-executable stack (and fix)."

[18] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proceedings of the 14th ACM conference on Computer and communications security.* ACM, 2007, pp. 552–561.

[19] E. Buchanan, R. Roemer, H. Shacham, and S. Savage, "When good instructions go bad: Generalizing return-oriented programming to risc," in *Proceedings of the 15th ACM conference on Computer and communications security.* ACM, 2008, pp. 27–38.

[20] A. Francillon and C. Castelluccia, "Code injection attacks on harvard-architecture devices," in *Proceedings of the 15th ACM conference on Computer and communications security.* ACM, 2008, pp. 15–26.

[21] C. Miller and V. Iozzo, "Fun and games with mac os x and

iphone payloads," *BlackHat Europe,* 2009.

[22] T. Kornau, "Return oriented programming for the arm architecture," Ph.D. dissertation, Master 锟斤拷 s thesis, Ruhr-Universität Bochum, 2010.

[23] F. Lindner, "Cisco ios router exploitation," *Black Hat USA,* 2009.

[24] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, "Return-oriented programming without returns," in *Proceedings of the 17th ACM conference on Computer and communications security.* ACM, 2010, pp. 559–572.

[25] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump- oriented programming: a new class of code-reuse attack," in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security.* ACM, 2011, pp. 30–40.

[26] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi, "Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization," in *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE, 2013, pp. 574–588.

[27] N. Carlini and D. Wagner, "Rop is still dangerous: Breaking modern defenses," in *23rd USENIX Security Symposium (USENIX Security 14),* 2014, pp. 385–399.

[28] L. Davi, C. Liebchen, A.-R. Sadeghi, K. Z. Snow, and F. Monrose, "Isomeron: Code randomization resilient to (just-in-time) return-oriented programming." in *NDSS*, 2015.

[29] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz, "Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications," in *Security and Privacy (SP), 2015 IEEE Symposium on. IEEE,* 2015, pp. 745–762.

[30] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières, and D. Boneh, "Hacking blind," in *2014 IEEE Symposium on Security and Privacy.* IEEE, 2014, pp. 227–242.

[31] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang, "Data-oriented programming: On the expressiveness of non-control data attacks," 2016.

[32] drdobbs, "Fortify, cert/cc team up for secure C, C++," *Dr Dobbs Journal,* 2007.

[33] A. Almossawi, K. Lim, and T. Sinha, "Analysis tool evaluation: Coverity prevent," *Pittsburgh, PA: Carnegie Mellon University,* 2006.

[34] Y. Kim, M. Kim, Y. J. Kim, and Y. Jang, "Industrial application of concolic testing approach: A case study on libexif by using crest-bv and klee," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 1143–1152.

[35] D. Cristian Cadar and D. Dunbar, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," *OSDI, San Diego, CA, USA (December 2008),* 2008.

[36] J. W. Duran and S. C. Ntafos, "An evaluation of random testing," *IEEE transactions on software engineering,* no. 4, pp. 438–444, 1984.

[37] D. Aitel, "An introduction to spike, the fuzzer creation kit," *presentation slides),* Aug, vol. 1, 2002.

[38] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda, "Panorama: capturing system-wide information flow for malware detection and analysis," in *Proceedings of the 14th ACM conference on Computer and communications security.* ACM, 2007, pp. 116–127.

[39] J. Newsome and D. Song, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," 2005.

[40] W. Drewry and T. Ormandy, "Flayer: Exposing application internals." *WOOT,* vol. 7, pp. 1–9, 2007.

[41] M. G. Kang, S. McCamant, P. Poosankam, and D. Song, "Dta++: Dynamic taint analysis with targeted control-flow propagation." in NDSS, 2011.

[42] J. Clause, W. Li, and A. Orso, "Dytan: a generic dynamic taint analysis framework," in *Proceedings of the 2007 international symposium on Software testing and analysis.* ACM, 2007, pp. 196–206.

[43] G. Yang, S. Person, N. Rungta, and S. Khurshid, "Directed incremental symbolic execution," *ACM Transactions on Software Engineering and Methodology (TOSEM),* vol. 24, no. 1, p. 3, 2014.

[44] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proceedings of the 12th ACM conference on Computer and communications security.* ACM, 2005, pp. 340–353.

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

[46] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Transparent rop exploit mitigation using indirect branch tracing," in *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13),* 2013, pp. 447–462.

[47] L. Davi, A.-R. Sadeghi, D. Lehmann, and F. Monrose, "Stitching the gadgets: On the ineffectiveness of coarse- grained control-flow integrity protection," in *23rd USENIX Security Symposium (USENIX Security 14),* 2014, pp. 401–416.

[48] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis, "Out of control: Overcoming control-flow integrity," in *2014 IEEE Symposium on Security and Privacy.* IEEE, 2014, pp. 575–589.

[49] P. Team, "Homepage of the pax team," 2007.

[50] F. B. Cohen, "Operating system protection through program evolution, 1992."

[51] C. E. Shannon, "Communication theory of secrecy systems," *Bell system technical journal,* vol. 28, no. 4, pp. 656–715, 1949.

[52] B. De Sutter, B. Anckaert, J. Geiregat, D. Chanet, and K. De Bosschere, "Instruction set limitation in support of software diversity," in *International Conference on Information Security and Cryptology.* Springer, 2008, pp. 152–165.

[53] T. Jackson, B. Salamat, A. Homescu, K. Manivannan, G. Wagner, A. Gal, S. Brunthaler, C. Wimmer, and M. Franz, "Compiler-generated software diversity," in *Moving Target Defense.*

Springer, 2011, pp. 77–98.

[54] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Smashing the gadgets: Hindering return-oriented programming using in-place code randomization," in *2012 IEEE Symposium on Security and Privacy.* IEEE, 2012, pp. 601–615.

[55] S. Forrest, A. Somayaji, and D. H. Ackley, "Building diverse computer systems," in *Operating Systems, 1997., The Sixth Workshop on Hot Topics in.* IEEE, 1997, pp. 67–72.

[56] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum, "Enhanced operating system security through efficient and fine-grained address space randomization," in *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12),* 2012, pp. 475–490.

[57] B. Coppens, B. De Sutter, and K. De Bosschere, "Protecting your software updates," *IEEE Security & Privacy,* vol. 11, no. 2, pp. 47–54, 2013.

[58] M. Chew and D. Song, "Mitigating buffer overflows by operating system randomization," 2002.

[59] S. Bhatkar, D. C. DuVarney, and R. Sekar, "Address obfuscation: An efficient approach to combat a broad range of memory error exploits." in *Usenix Security,* vol. 3, 2003, pp. 105–120.

[60] ——, "Efficient techniques for comprehensive protection from memory error exploits." in *Usenix Security,* 2005.

[61] A. Gupta, S. Kerr, M. S. Kirkpatrick, and E. Bertino, "Marlin: A fine grained randomization approach to defend against rop attacks," in *International Conference on Network and System Security.* Springer, 2013, pp. 293–306.

[62] G. S. Kc, A. D. Keromytis, and V. Prevelakis, "Countering code-injection attacks with instruction-set randomization," in *Proceedings of the 10th ACM conference on Computer* and *communications security.* ACM, 2003, pp. 272–280.

[63] D. Bigelow, T. Hobson, R. Rudd, W. Streilein, and H. Okhravi, "Timely rerandomization for mitigating memory disclosures," in *ACM Sigsac Conference on Computer and Communications Security,* 2015, pp. 268–279.

[64] K. Lu, S. Nürnberger, M. Backes, and W. Lee, "How to make aslr win the clone wars: Runtime re-randomization," 2016.

[65] B. Lee, L. Lu, T. Wang, T. Kim, and W. Lee, "From zygote to morula: Fortifying weakened aslr on android," in *2014 IEEE Symposium on Security and Privacy.* IEEE, 2014, pp. 424–439.

[66] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou, "Practical control flow integrity and randomization for binary executables," in *Security and Privacy (SP), 2013 IEEE Symposium on.* IEEE, 2013, pp. 559–573.

[67] M. Zhang and R. Sekar, "Control flow integrity for cots binaries," in *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13),* 2013, pp. 337–352.

[68] M. Backes and S. Nürnberger, "Oxymoron: Making fine-grained memory randomization practical by allowing code sharing," in *23rd USENIX Security Symposium (USENIX Security 14),* 2014, pp. 433–447.

陈小全 2017 年在中国科学院信息工程研究所信息安全国家重点实验室获得博士学位。研究方向为信息安全,网络安全,密码学等。研究兴趣包括: 软件漏洞利用与阻止,密码学技术应用,系统可生存性技术，以及人工智能技术等。Email: chenxiaoquan@iie.ac.cn

薛锐 1999 年在北京师范大学大学获得博士学位。现任中国科学院信息工程研究所信息安全国家重点实验室副主任。研究方向包括: 密码学, 安全协议, 计算复杂性理论、信息安全理论、密码协议的形式化方法、数理逻辑。Email: xuerui@iie.ac.cn