内存数据污染攻击和防御综述

马梦雨[1,2], 陈李维[1], 孟 丹[1,2]

[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 网络空间安全学院, 北京 中国 100049

摘要 内存数据被污染往往是程序漏洞被利用的本质所在, 从功能角度把内存数据划分为控制相关和非控制相关, 由此引出控制流劫持攻击和非控制数据攻击。两者危害程度相当, 前者因利用成本较低而成为主流, 但随着控制流劫持防御方法的不断完善, 非控制数据攻击逐渐被重视。研究者先后在顶级会议上提出了数据导向攻击得自动化利用框架 Data-oriented Exploits(DOE)以及图灵完备性地证明 Data-oriented Programming(DOP), 使得非控制数据攻击成为热点。本文基于这两种攻击形式, 首先简化内存安全通用模型, 并对经典内存数据污染攻击和防御的原理进行分析, 其次分别论述新型控制流劫持和非控制数据攻击与防御的研究现状, 最后探讨内存安全领域未来的研究方向, 并给出两者协作攻击和防御的可能方案。

关键词 内存数据污染；内存安全通用模型；控制流劫持攻击；非控制数据攻击；协作攻击和防御
中图法分类号 TP309.2 DOI 号 10.19363/j.cnki.cn10-1380/tn.2017.10.007

A Survey of Memory Corruption Attack and Defense

MA Mengyu[1, 2], CHEN Liwei[1], MENG Dan[1, 2]

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

Abstract Memory corruption is one of the important research about computer security, and it's the essence of programs being exploited. Memory data is divided into control-related and non-control related from the angle of function, which leads to control flow hijacking attacks and non-control data attacks. The threats of both are almost the same. The former became mainstream because of the lower costs. With the continuous improvement of control flow hijacking defense methods, non-control data attacks are valued gradually. Researchers have presented automatic generation of Data-oriented Exploits (DOE) and Turing-complete Data-oriented Programming (DOP) at the top-level meeting. This paper simplifies the general model of memory security based on the two types of attacks. We analyze the principles of classic memory corruption, and summary its research status systematically by introducing new control flow hijacking and non-control data attack and defense. Then we discuss future research direction of memory security, and give the possible schemes of collaborative attack and defense.

Key words Memory corruption; generic memory security model; control flow hijack; non-control data attack; collaborative attack and defense

1 引言

系统级别的语言如 C 和 C++, 使程序员可以更自由地控制代码, 编写出优化且高效的程序, 但也使得程序不可避免地出现漏洞, 例如内存泄露、堆栈溢出、格式化字符串、整型溢出等威胁系统和软件安全的漏洞形式[1], 恶意的使用者就会利用漏洞劫持程序并执行恶意代码或提升系统权限, 其中内存数据污染攻击一直都备受攻击者喜爱[2]。研究者曾提出一种用于分析内存安全的通用模型[3], 包括了不同层次的安全策略和攻击向量。从中可以得出, 内存安全问题的本质就在于内存数据被攻击者污染, 导致程序执行了异常的行为和逻辑。内存数据从功能的角度可以分为控制流相关和非控制流相关数据, 攻击者污染控制数据就会导致控制流劫持攻击, 污染非控制数据则会导致非控制数据攻击。两者危害程度相当, 前者因为相对容易被利用, 所以成为主流的攻击方式; 而后者因利用成本较高所以被忽视[4], 但近几年因为控制流劫持防御方法的不断完善, 非控制数据攻击逐渐被研究者重视[5-7]。

通讯作者: 陈李维, 博士, 助理研究员, Email: chenliwei@iie.ac.cn。
本课题得到国家自然科学基金(61602469)资助。
收稿日期: 2017-03-24; 修改日期: 2017-05-08; 定稿日期: 2017-08-23

控制流劫持攻击作为传统主流的攻击方式, 造成的危害极大。攻击者能够通过内存漏洞, 使用代码注入或者代码重用的方式篡改程序内存空间中的关键控制数据, 例如函数栈上的返回地址、函数堆上的存贮指针等, 从而劫持正常的控制流转向恶意控制流完成攻击。早期的代码注入攻击[8]通过外部输入的方式在内存中存储恶意代码, 然后劫持程序流转向恶意代码执行来达到目的。为了抵御此种类型的攻击, 研究者提出了几种防御方法: 栈 cookie[9], 在栈中函数返回地址前添加标记信息, 防止外部输入地溢出污染; 数据不可执行(Data Execution Prevention, DEP) [10, 11], 在内存页属性上设置不可执行权限, 限制外部输入的数据不能运行; 地址空间布局随机化(Address Space Layout Randomization, ASLR)[12], 使得虚拟内存空间的地址布局对攻击者不可见。DEP 能够从本质上防御代码注入攻击, ASLR 能够增加注入攻击的利用成本, 因此这些防御机制能够很好地抵御代码注入攻击, 但是对代码重用攻击的防御不足。代码重用攻击通过利用程序代码中已有的指令序列来绕过 DEP[13], 通过即时编译和信息泄露技术来绕过 ASLR[14, 15], 如 return-to-libc[16], Return Oriented Programming(ROP)[13, 17], Jump Oriented Programming (JOP)[18, 19]等。为了抵御此种攻击, 研究者提出了控制流完整性(Control-flow Integrity, CFI)[20]和代码指针完整性(Code-pointer Integrity, CPI)[21], 理论上可以很好地防御所有的控制流劫持攻击。但由于性能和兼容性的问题而不得不有所妥协, 这些防御机制依然被精心设计得攻击而突破[22-27]。本文总结了针对 CFI 和 CPI 的弱点而提出的新型控制流劫持攻击方案, 以及为了防御这些攻击而进一步完善的防御机制。

随着控制流劫持攻击防御方法的不断完善, 非控制数据攻击逐渐被重视。近年提出控制流弯曲(Control-flow Bending, CFB)[6]通过修改敏感函数的参数, 控制流相关的指令地址等关键数据, 绕过细粒度 CFI 的防御, 完成信息泄露或提权操作。紧接着提出了可以自动生成的数据导向攻击框架(Data-oriented Exploits, DOE)[5]和数据导向编程(Data- oriented Programming, DOP)[7], 充分表明了非控制数据攻击的有效性和完备性。目前还没有能够防御此类攻击的有效措施。曾经和 CFI 同时期提出的数据流完整性(Data-flow Integrity, DFI)[28]虽然提供了防御此类攻击的理论支持, 但存在成倍的性能开销。写完整性测试(Write Integrity Test, WIT)[29]和部分内核数据保护[30]等防御方法一定程度上提升了 DFI 的效率, 但对于以 DOP 为代表的新型非控制数据攻击的防御效果仍然不佳。本文也总结为了攻破细粒度 CFI 而转向利用非控制数据攻击的发展过程, 并讨论防御此类攻击的可行方案。

本文和其他相似文献比较, 不仅仅阐述基于内存错误的攻击和防御技术, 而是从攻击本质入手, 以内存数据表示的功能为依据, 划分为控制和非控制相关两大类别, 分别综述其攻击过程和防御思想, 并以时间轴的形式论述其研究现状。文章结构安排如下: 第二章提出简化的内存安全模型, 突出内存数据污染攻击和防御的范畴。从纵向的角度把攻击过程划分为不同的阶段, 不同层次有不同的防御方法。从横向的角度比较两类攻击方式的主要区别, 污染不同功能的内存数据, 形成不同类别的攻击方式; 第三章依据第二章提出的安全模型, 对经典的内存数据污染攻击和防御进行总结。从攻击目标, 攻击手段, 防御方法三个方面进行介绍; 第四章通过介绍新型控制流劫持攻击和防御之间地较量来说明其研究现状; 第五章对非控制数据攻击和防御的发展过程进行论述, 包括非控制数据攻击概念的提出, 系统化的实践, 以及图灵完备地证明; 第六章对内存数据污染攻击和防御进行总结和讨论; 第七章提出内存安全未来可能的研究方向, 并提出针对新型非控制数据攻击防御的优化和增强思想; 第八章对全文进行总结。

2 简化的内存安全通用模型

文献[4]提出的经典内存安全模型从五个方面阐述了基于内存错误导致的攻击类型和对应的防御措施, 总结于表 1 中。

表 1 内存安全攻击和防御

攻 \ 防	指针安全	权限	控制流	数据流	随机化
内存错误	边界(空间) 状态(时间)				
代码注入		DEP	CPI, CFI	DFI	ISR
代码重用			CPI, CFI	DFI	ASR
数据污染				DFI	DSR
信息泄露					随机化

(注: DEP(Data Execution Prevention), CPI(Code-pointer Integrity), CFI(Control-flow Integrity), DFI(Data-flow Integrity), ISR(Instruction Space Randomization), ASR(Address Space Randomization), DSR(Data Space Randomization))

根据表 1 展开叙述。第一, 攻击者从空间上利用数据指针越界操作, 造成溢出漏洞, 或从时间上利用对象释放后再次非法引用操作, 造成悬空指针漏

洞, 以此非法读写目标程序的内存数据。因为这类内存错误本质上就是安全策略的违反情况, 因此可以在底层监控指针的边界和对象的引用状态来防御, 也可以在高级语言层面设计便于标记代码和内存对象的编程语言, 然后程序利用携带的标记与操作系统和软件实现协同防御[31]。第二, 攻击者会尝试直接修改内存中的程序代码, 如果不能直接修改, 那么就破坏代码指针, 劫持程序正常的执行逻辑, 转向执行注入的恶意数据完成攻击。限制程序的代码段不可写且数据段不可执行, 可以有效地防御此类攻击。第三, 如果注入的外部指令被限制权限不能执行, 那么就利用程序中原有的指令片段替代外部注入的指令完成攻击。研究证明, 在一定大小的程序内存空间中, 能够找到图灵完备的攻击指令序列[16]。如果劫持是通过直接修改代码指针的方式, 则可以通过 CPI 来防御, 如果劫持是通过污染间接控制转移指令的操作数, 则可以通过 CFI 来防御。第四, 如果不能劫持目标程序的原有执行流程, 那么就直接修改一些安全的敏感数据, 在不改变控制流的前提下获取一定权限。DFI 能够发现恶意的数据篡改并及时阻止; 第五, 以上四种类型的攻击都可用于泄露内存的数据, 有效的防御措施不外乎两种, 一种是加密机制, 另外一种是随机化机制[32], 又细分为指令随机和地址随机。

本文对以上五个方面进行总结和归纳, 以突出内存安全的重点研究内容: 第一种内存错误是其他类型攻击的前提, 第五种信息泄露是其他类型攻击的附属效应, 第二种代码注入和第三种代码重用可以合并为控制流劫持攻击, 第四种篡改敏感数据即非控制数据攻击。因此可以简化内存安全模型如图 1 所示, 从图中可以很明显得看出内存数据污染攻击的范畴和防御此类攻击的通用缓解机制。

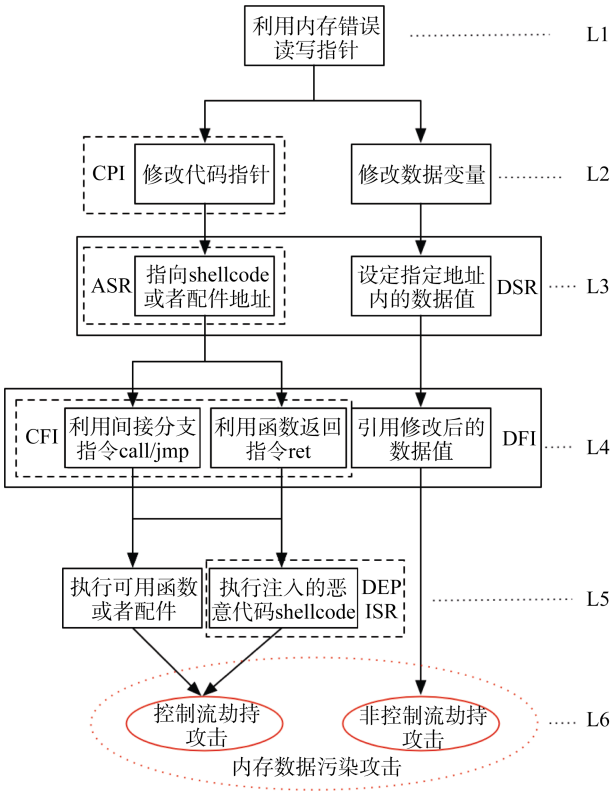

图 1　简化的内存安全通用模型示意图

表 2　控制相关数据分析

序号	描述
1	直接 Jmp 指令的操作数
2	直接 Call 指令的操作数
3	Ret 指令的操作数
4	间接 Jmp 指令的操作数
5	间接 Call 指令表示为函数指针时的操作数
6	间接 Call 指令表示为调用虚函数表时的操作数
7	间接 Call 指令表示为消息触发例程时的操作数
8	控制转移指令表示为例外触发时的操作数
9	控制转移指令表示为动态链接、独立编译等模块化特征时的操作数

3　经典的内存数据污染攻击和防御

分析第二部分的简化内存安全模型(图 1)得出, 攻击者首先利用内存错误(如溢出指针边界和悬空指针)来污染程序中和安全相关的数据(见表 2 和表 3), 然后影响正常的执行流或输出结果, 最后实现恶意代码地执行或权限地提升。本节首先分析程序中和安全相关的数据, 从功能的角度来划分为控制流相关和非控制流相关数据, 其次介绍对应的控制流劫持和非控制数据攻击以及防御。

3.1　经典的控制流劫持攻击和防御

3.1.1　控制流转移数据分析

文献[33]从控制转移指令的角度分析了能够决定程序控制流的相关数据, 整体上分为两类: 前向转移 Jmp 和 Call 指令的操作数据, 以及后向转移 Ret 指令的依赖数据。细分为五种不同情况: 第一, 直接的 Jmp 指令依赖的操作数往往是一个常量, 静态分析就可以获得它的目的地址, 例如循环和条件语句使用直接 Jmp 来控制执行流的转移; 第二, 直接的 Call 指令同样依赖于一个常量, 例如静态的函数调用; 第三, 间接的 Jmp 指令则需要通过运行时的数据计算才能获得其目的地址, 例如多条件分支语句依赖的分配表, 以及延迟绑定技术和多线程调度优化技术的具体实现等; 第四, 间接的 Call 指令同样需要动态计算才能获得操作数据, 例如 C++中虚函数的实现需要依赖于分配表指向不同的实函数, 以及回

调函数和 Smalltalk 类型的消息响应机制依赖分配表的动态查找过程; 第五, 函数返回 Ret 指令的操作数据, 要根据 Ret 指令对应 Call 指令的下一条指令所在的地址决定。控制相关数据总结于表 2 中。

其中前两行表示的数据依赖于一个常量, 而能够被攻击者利用的数据往往具备动态属性的性质, 所以排除前两行, 后面七种情况则需要依赖寄存器或内存的数据通过动态计算得到, 因此都是攻击者能够劫持程序控制流的安全关键数据。

3.1.2 控制流劫持攻击

分析图 1 的简化内存安全模型得出, 攻击者在 L4 层通过污染程序控制流相关的安全关键数据(表 2), 劫持程序控制流去执行 L5 层的代码指令。这部分恶意代码可能来源于注入的 shellcode, 可用的系统函数, 以及精心构造的功能指令配件(一段以 ret 为结尾的具有一定功能的短指令序列)。于是根据恶意代码的来源可以得出三种经典的控制流劫持攻击示例, 分别为通过缓冲区溢出的代码注入攻击[8], 返回函数库(Return-to-libc)攻击[16], 以及代码重用攻击(Code Reuse Attacks, CRAs)[13], 如图 2 所示。攻击者首先通过缓冲区溢出的方式, 污染栈中的返回地址, 并配置一些参数和变量, 在程序执行 Ret 指令时, CPU 将栈中的返回地址弹给 EIP 寄存器, 系统读取 EIP 寄存器中的值继续执行程序代码。对于代码注入攻击, 攻击者就劫持返回地址跳转到注入的 shellcode 地址, 对于返回函数库攻击, 则跳转到系统敏感的函数地址, 对于代码重用攻击, 则跳转到精心构造的配件序列地址, 然后系统就会自动地执行恶意代码完成攻击。

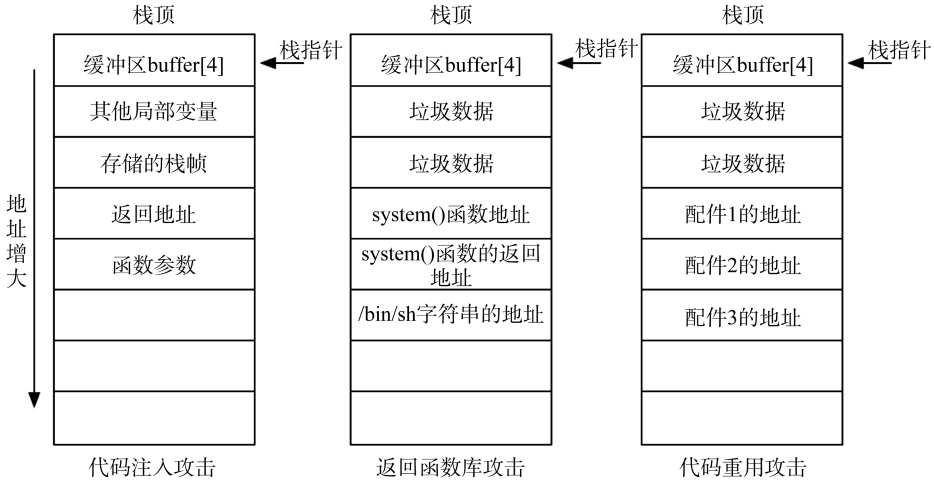

图 2 三种经典的控制流劫持攻击示例

3.1.3 控制流劫持攻击的防御

对于控制流劫持攻击防御的方法可以归纳为以下几类: 基于指令执行的特征, 从而进行动态的监控思想[34, 35]; 基于内存随机化, 从而隐藏控制相关数据的思想[36]; 以及最为贴合攻击本质的 CFI[20]和 CPI[21], CPI 在图 1 安全模型的 L2 层阻止攻击, CFI 在图 1 安全模型的 L4 层发现攻击。下面具体介绍经典 CFI 和 CPI 的防御思想。

控制流完整性 CFI 的核心思想是依据静态分析, 获得程序的控制流图(Control-Flow Graph, CFG), 计算出所有和控制流相关的数据, 即间接转移指令合法的目的地址, 然后通过二进制重写技术在 CFG 路径上添加标记, 最后严格要求程序依照 CFG 执行, 示例如图 3 所示。这种思想被称之为原始的细粒度 CFI, 虽简单但性能开销较大。因此, 研究人员进一步提出了粗粒度的 CFI, 它并不要求完全精度的 CFG, 而是仅仅对部分控制流路径添加限制规则, 极大提高了效率但同时也降低了安全性。

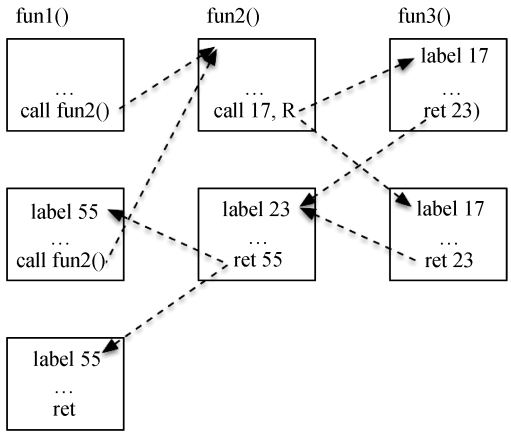

图 3 CFI 原理示意图

代码指针完整性 CPI 的核心思想是把进程的内存空间划分为安全区域和常规区域, 敏感的指针数据存放在安全区中, 只有安全的指针操作才能访问

安全区的规则, 以此达到保护敏感指针数据不被劫持的目的。CPI 的实现方法是通过源码的插桩技术, 在编译时分析出程序需要保护的敏感指针对象, 并把这些敏感指针替换为符合 CPI 要求的安全指针, 再使用安全指针去访问安全的内存区域。作者同时还提出了一种条件相对宽松的机制, 代码指针分离(Code-Pointer Separation, CPS),它减少了安全区域的大小以及敏感代码指针的数量和质量, 因此大大降低了开销。CPI 框架如图 4 所示。

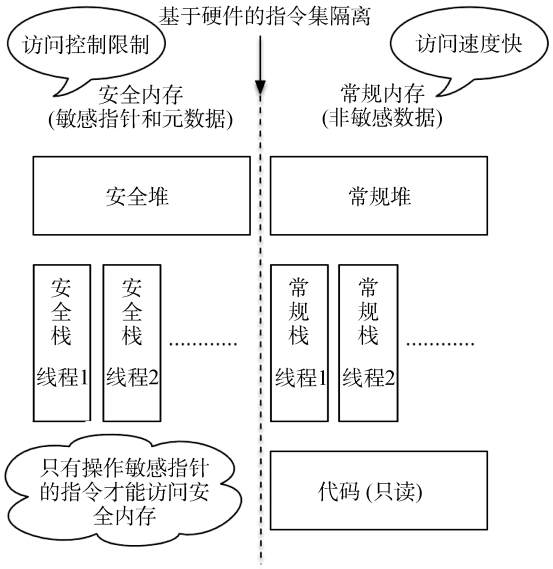

图 4 CPI 整体架构示意图

3.2 经典的非控制数据攻击和防御

3.2.1 非控制安全关键数据分析

文献[4]根据一些常见的网络服务器运行日志, 总结出了五类非控制安全关键数据: 第一, 通常用于初始化服务程序的配置文件数据, 包括用户登陆权限, 文件的访问控制权限以及其他安全相关的初始化参数等。这些数据往往控制程序的运行时行为, 而且它们几乎不会被改变, 都是设定好的特殊值; 第二, 用于识别用户身份的数据, 如用户标识和群组标识等。在远程连接服务器进行授权之前, 需要根据用户的身份认证从而选择相应的权限; 第三, 服务器安全检查机制使用的由用户外部输入的数据; 第四, 系统中一些布尔决策数据, 如用户身份决定得用于认证权限的布尔数据。第五, 其他一些安全的关键数据, 如文件描述符、内核中文件列表的索引值、系统调用对应处理例程的索引值等。非控制相关数据总结于表 3 中。

3.2.2 非控制数据攻击

传统的非控制数据攻击不同于控制流劫持攻击, 它需要对程序语义有较高了解, 才能准确地完成攻击过程。结合表 3 中的非控制相关数据, 阐述几种可能的攻击示例: 第一, 如果配置文件被篡改, 最直观的影响就是攻击者可以直接绕过初始化的访问控制策略, 从而访问一些敏感文件; 第二, 客户和服务器之间连接后会在缓存中保留相应的用户标识, 方便同一用户快速地再次发起请求, 如果攻击者篡改了缓存中的数据, 刷新后便可以获得额外的操作权限; 第三, 攻击者可以利用合法的输入绕过程序的验证检查机制, 然后再污染这部分数据, 最后利用上下文条件竞争的逻辑漏洞, 强制程序使用被篡改后的数据从而达到恶意的目的; 第四, 系统决策例程中的条件判断语句所依赖的数据往往来自寄存器或内存中, 那么攻击者就可以篡改寄存器或内存的数据, 从而影响认证结果。下面通过一个具体的实例说明非控制数据攻击的过程。如图 5 所示, 攻击者通过栈溢出的方式攻击用户的输入数据, 发送一个 GET 命令到服务器, 由于输入的字符串过长而溢出栈帧, 覆盖了调用函数(caller)栈帧中寄存器的值, 等被调用函数(callee)返回时, POP 指令还原调用函数(caller)栈帧中寄存器的值, 这时因为决策变量 *authenticated* 被污染, 导致服务器返回一个超级用户的 shell。

表 3 非控制相关数据分析

序号	描述
1	初始化系统的配置文件数据
2	用户身份识别的数据
3	部分用户外部输入的数据
4	系统决策性质的数据
5	其他一些安全的关键数据(描述符, 索引值等)

3.2.3 非控制数据攻击的防御

对于非控制数据攻击的防御, 大体上可归纳为三类: 第一, 结合具体语义的躲避防御思想, 例如准确声明安全变量的生命周期[4]以及使用特殊类型的关键字声明安全变量的访问权限[37], 以消除这些数据被污染的可能性; 第二, 数据随机化[32]思想(图 1 安全模型的 L3 层)对攻击者隐藏安全数据的内存布局或安全数据的具体含义; 第三, 污点分析[38]和 DFI[28]思想(图 1 安全模型中的 L4 层)精确保障安全数据的来源和引用是合法的。具体内容在第五章节中再详细介绍。

4 新型控制流劫持攻击和防御

第三章节中简述了传统的控制流劫持攻击[39](图 6 虚线以上的代码注入, Return-to-libc, ROP, JOP 攻击), 其促进了一系列防御思想的提出, 而后又发展出了新型的控制流劫持攻击。接下来就通过新型代

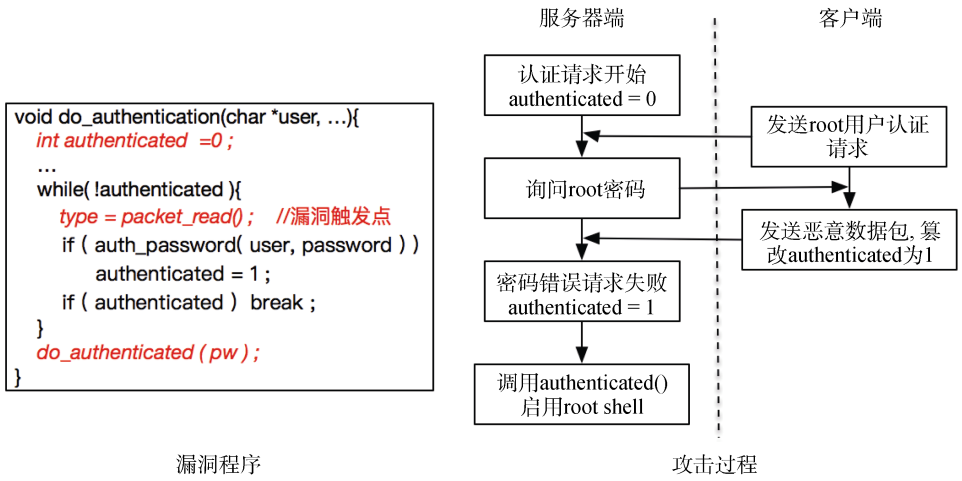

图 5 非控制数据攻击示例

码重用攻击和防御之间的较量来说明控制流劫持攻击和防御的研究现状, 其发展趋势如图 6 所示。

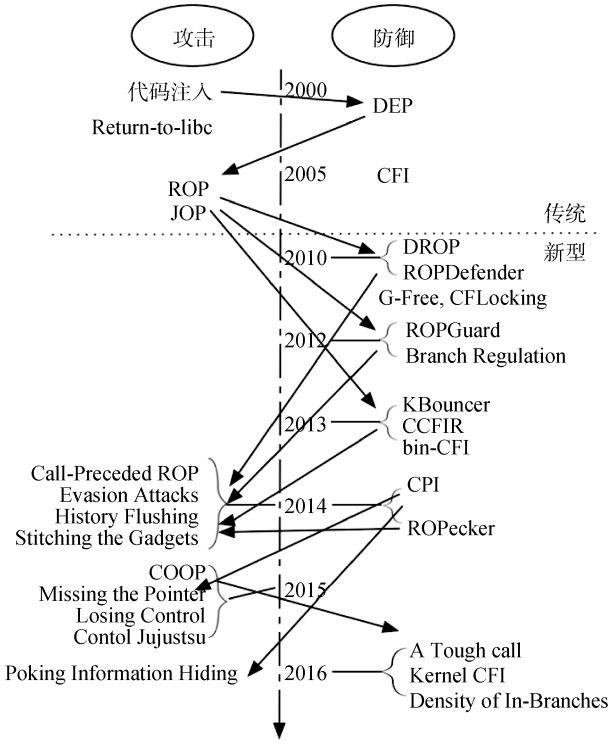

图 6 控制流劫持攻击和防御的发展

攻防交替的过程在时间抽上看得很清楚: 由于 DEP 对代码注入攻击的防御效果很好, 所以催生出以传统代码重用攻击(ROP[13, 17], JOP[18, 19])为代表的控制流劫持方案; 根据传统代码重用攻击实施的指令或行为特征, 研究者提出一系列基于特征的启发式防御技术(DROP[34], ROPDefender[35], G-Free[42]); 根据代码重用攻击污染控制相关数据的本质特征, 研究者提出了一系列可行的 CFI 实现方案(CCFIR[44], Bin-CFI[50], KBouncer[47], ROPGuard[46], ROPecker[48]), 然而为了考虑可实施性, 这些方案都是典型的粗粒度 CFI 思想, 自然存在被攻击者绕过的可能; 所以被具有针对性的新型攻击方式所攻破(Call-preceded ROP, Evasion Attacks, History Flushing)[24], 这些攻击方式能够有效绕过基于特征的启发式防御和基于粗粒度 CFI 的防御; 而后不得不寻找更有效且更安全的防御机制(CPI[21], HA-CFI[65]), 虽然 CPI 面临着侧信道攻击的威胁, 但控制流劫持攻击的利用成本大大增加了, 其危害性自然也就降低了, 因此对于控制流劫持攻击的防御研究也渐渐放慢了节奏。

整体上从三个方面来阐述控制流劫持攻击和防御的研究情况: 传统的代码重用攻击和基于特征的启发式防御之间的对抗; CFI 的发展和新型代码重用攻击之间的对抗; 以及 CPI 和信息泄露攻击之间的对抗。

4.1 传统代码重用攻击 VS 基于特征的启发式防御

由于代码重用攻击中需要一些短小的指令序列来构成具有一定功能的配件, 配件相连形成 shellcode 完成恶意功能, 因此有着以下几个区别于正常程序的显著特征: 依赖两个重要的寄存器(程序指针 EIP 和栈顶指针 ESP); 每个配件都是以 Ret 指令结束, 并且 Ret 指令的目的地址是前一条 pop 指令的操作数; 程序运行过程中会执行超过常规数值的配件数量。正是因为这些规律和特征, 衍生出了基于指令执行特征检测的防御思想。

文献[34]基于统计规律和启发式学习来判断程序运行过程中的配件数量是否超过一定的阈值, 以此区分程序的正常或攻击行为。文章的核心思想是首先识别 Ret 指令, 然后判断它的目的地址是否指向系统中的 libc 库, 除此之外还要记录从 Ret 指令目的

地址开始的连续配件数量以及每个配件包含的指令条数, 用上面的统计结果来检测程序是否运行了 ROP 恶意代码。作者对大量的实验数据进行分析得出, 虽然正常程序执行 Ret 指令后也可能执行连续的配件代码, 但是和恶意代码还是有一定地区分。结果表示单个功能配件中包含的指令条数通常不超过 5 条, 并且连续的配件序列形成的配件链长度不超过 20, 以此为检测标准判断是否发生攻击行为。此方法主要问题在于误报率较高, 后来被文献[22]中提出的 NOP 配件绕过, 攻击者可以插入没有实际作用的 NOP 指令, 让配件的长度超过被认为是配件的要求, 稀释了功能配件出现的频率, 所以能够逃脱监测。

文献[35]基于 ROP 配件都是以 Ret 指令为结尾的特点, 提出一种影子栈(Shadow Stack)的思想, 也是控制流完整性中后向安全(backward control-flow)的重要保障。影子栈利用 call 指令和 Ret 指令的一致性防御 ROP 攻击, 即正常程序每次执行 call 指令后陷入被调用函数, 等待函数返回时应该跳转到 call 指令的下一条指令所在的地址, 这条指令我们称之为 call-preceded 指令, 也就是说 Ret 指令的目的地址应该是 call-preceded 指令的地址, 而恶意行为会破坏这一原则, 利用 Ret 指令跳转到非 call-preceded 指令, 从而执行恶意行为。影子栈的核心思想是每次程序执行 call 指令时, 除了把函数返回地址压栈外, 额外再复制一份返回地址存入可信的影子栈中, 每次栈顶执行 Ret 指令时, 需要先比较常规栈和影子栈的栈顶值是否相同, 如果不相同则表示为异常行为并终止程序执行。很明显, 影子栈主要问题在于无法防御 JOP 攻击, 原因在于 JOP 使用的攻击配件是以 Jmp 指令为结尾的指令片段。除此之外, 还必须保证影子栈作为可信基的前提, 一旦影子栈中的内容被篡改, 攻击者也就能够绕过检测机制了。文献[41]使用加锁(locking)结合 CFI[20] 的思想来弥补影子栈的不足同时又降低 CFI 的性能开销。在编译时添加锁信息相关的段(section), 在链接时构造函数调用关系图, 在执行间接分支指令前根据锁的状态值和函数调用关系图来决定是否违反策略。虽然这种防御思想能够防御 JOP 类型的攻击, 但是为了适用于编译的标准, 需要大量的人工来修改 libc 库的代码。

除了以上两种经典的基于特征启发的防御方法之外, 还有研究者提出其他比较有代表性的思想: 文献[42]则认为程序控制流被劫持的部分原因在于 x86 指令集较为紧密, 没有进行对齐处理, 就导致了从不同的位置读取字节, 得到的指令不一样, 于是就产生了无意识配件, 攻击者可以利用它执行非法的意图。因此文章通过修改编译器, 强制将指令进行对齐, 从而使生成的二进制代码中不包括无意识配件。此方法虽然效率还可以, 但只能适合于源码分析, 并且只能消除无意识配件, 并不能阻止攻击者利用程序中存在的配件进行攻击; 文献[43]介绍了一种通过计算间接分支指令的密度来检测 ROP 攻击的方法, 结果显示 ROP 攻击程序中确实存在较高的间接分支指令密度, 就文章中的 benchmarks 来说, 可以确定一个通用的阈值为每 32 条指令中存在 13 条以上的间接指令, 则认定为恶意攻击的发生。

基于特征的启发式防御, 并没有从攻击本质的角度去阻止控制流劫持的发生, 而是通过检测的方式判断程序中是否发生了类似攻击的行为。下面就从 CFI 和 CPI 的角度, 探讨控制流劫持攻击和防御的发展。

4.2 CFI 发展 VS 新型重用攻击

由于原始 CFI[20]性能和通用性的问题, 研究者提出了许多优化的方案, 如粗粒度 CFI 等, 不可避免的需要在安全性上做出妥协, 因此也催生出了一些具有针对性的攻击方案。

CCFIR[44]是一种对粗粒度 CFI 思想的软件实现方案。首先通过重定位表来分析每个模块中所有间接分支指令的可能目的地址集合, 然后重写二进制文件增加一个跳转代码段(segment), 最后在程序运行时, 要求所有的间接转移指令转向这个跳转代码段, 再完成控制流的转移, 跳转代码段会去判断这些间接转移指令的目标地址是否在合法的集合内, 如果不在就认为是异常情况。系统策略从控制转移指令的角度分为三类: 第一, 间接 Call 和 Jmp 指令强制规定只能跳转到函数的开头, 并且目标地址都使用 8 字节对齐, 以此最大可能地消除无意识配件的存在; 第二, 常规函数不能调用系统敏感函数的地址, 但敏感函数可以调用常规函数的地址; 第三, 跳转代码段需要通过随机化的方式, 在载入内存时隐藏地址布局。此方案优点是性能开销有所降低得同时, 对传统 ROP 攻击的防御效果也很乐观。为了进一步提高性能, 文献[45]则通过一种轻量级的硬件支持来实现粗粒度 CFI 的思想, 与 CCFIR 一致, 该原型系统也是通过规定了间接分支指令合法的转移集合, 从而避免构造细粒度 CFG 带来得严重性能开销。总结这类粗粒度 CFI 的思想如图 7 所示。

随着粗粒度 CFI 思想的兴起, 各种完善的防御机制也随之被提出。ROPGuard[46]提出在敏感系统调用(例如 *mprotect(), execve(), bind()*等)触发时, 基于一些启发式算法, 动态监控 ROP 的攻击特征, 从而

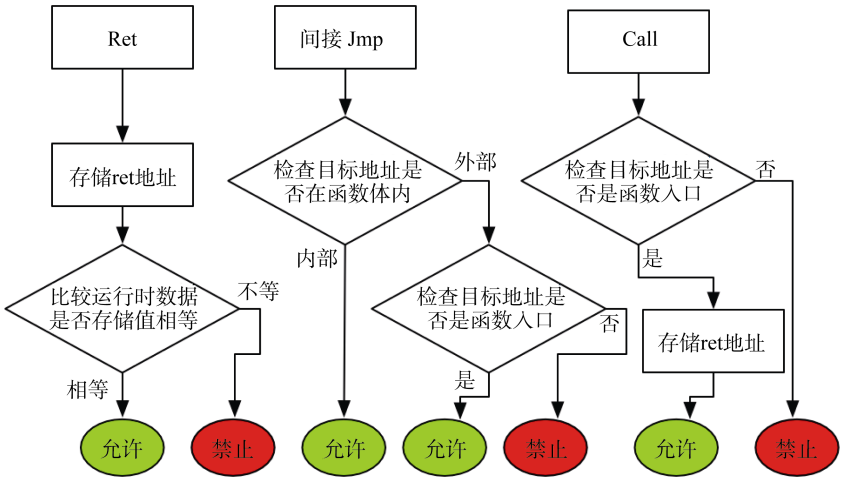

图 7　粗粒度 CFI 核心思想示意图

判断是否存在异常情况。KBouncer[47]是首次提出使用 intel 处理器自带硬件机制的防御思想, 这个硬件机制就是最近分支记录(Last Branch Recording, LBR), 它是一种循环寄存器组, 可以记录程序最近的十六条跳转指令信息, 并且它的定制性很强, 可以过滤掉不重要的分支而只关注重点的跳转。KBouncer 只检查最近十六个执行系统调用的分支信息, 然后设定两个规则: 其一是 Ret 指令的返回地址必须是 call-preceded 地址, 其二是检查最近 8 个间接分支指令是否具备配件的特征, 违反任意一种都会被认为是攻击行为。ROPecker[48]是一种通用, 不需要源码, 且非常高效的防御机制, 它不仅能防御 ROP 形式的攻击, 还能抵御 JOP 攻击。ROPecker 首先对程序进行离线的分析, 并使用硬件 LBR 寄存器记录执行流的分支信息, 然后依据攻击行为通常会在代码段进行大幅度跳转的特性, 在程序运行时使用滑动窗口(sliding window)的思想限制分支指令的跳转距离, 即不在当前窗口范围内的代码变为不可执行状态, 一旦分支指令跳转到窗口以外的地址便会触发分析例程, 而分析例程会基于离线的配件统计结果, 对有风险的指令片段(十一个连续的配件被认为是危险的)进行报警处理。

攻防相生相克, 不少研究者就针对粗粒度 CFI 的弱点, 提出了具有针对性的攻击方案。文献[24]提出了三种有效绕过粗粒度 CFI 的 ROP 攻击方法: 第一, call-preceded ROP 在不违反 call-preceded 原则的情况下仍然可以实现 ROP 攻击, 它只用 call-preceded 配件, 并且让配件序列变得复杂冗长, 使得可以满足执行 Ret 指令之后跳转到一个 call-preceded 指令的地址; 第二, 躲避攻击(Evasion Attacks)在配件序列中添加一些 NOP 指令, 从而伪装成正常的指令串, 那么就可以构造长短结合的配件来绕过基于配件长度特征的防御检测; 第三是刷新历史记录攻击(History Flushing), 既然一些防御方法是依据程序中分支指令执行的历史信息来检查未来执行流是否异常, 那么就可以通过添加 NOP 指令(无关的间接 Jmp)使得防御机制无法维持有效的历史记录, 从而绕过检查策略。结合以上三种攻击方法, 就可以有效绕过 CCFIR[44]、DROP[34]、ROPDfender[35]、KBouncer[47]、ROPekcer[48]等基于启发式或粗粒度 CFI 思想的防御机制。文献[22]同样对提出得各种 CFI 可行的解决方案进行安全性度量, 结合新型的 ROP 攻击讨论了这些技术的有效性。结果表明, 即使把上面提到得几种防御机制整合起来, 同样存在能够绕过它们且图灵完备的攻击方式, 因此需要设计更苛刻和更健壮有效的 CFI 策略。

随后, 文献[49]提出了一种控制流敏感的防御思想 PathArmor, 即 CFI 的思想结合上下文敏感的关系来增强防御能力。根据程序上下文敏感的静态分析和二进制插桩技术, 在目标文件的控制流路径中添加上下文敏感的控制流标记, 把控制流标记和 CFG 中的控制流关系联系在一起, 展示了一个可应用于实际程序高效且上下文敏感的 CFI 方案。文献[50]提出了二进制级别的细粒度, 模块化, 动态 CFI 原型系统 Lockdown, 使用共享库的符号表和 CFG, 使用更为细致的前向控制流转移规则以及影子栈的后向转移保护, 以此来防御所有类型的代码重用攻击。

虽然研究者提出了几个更为苛刻的 CFI 策略, 但是攻击者还是发现了细粒度 CFI 的弱点, 并提出了攻击理论。文献[27]提出使用 CFG 之外的控制流劫持目标程序。作者从三个方面揭露了 CFI 的弱点: 第一, 当发生函数调用时, 调用函数会将当前使用

到的寄存器值压入栈中, 以便函数返回时还原现场, 于是就给攻击者提供了机会, 如果他们能够篡改栈上保存的寄存器的值, 那么系统还原现场后就会导致 CFI 的检测失效; 第二, 用户态的 CFI 仅仅插桩用户模式的函数调用, 而不涉及系统内核的函数调用, 当一个系统调用返回时, 内核会从用户栈上读取返回地址, 然后跳转到用户代码的执行位置, 但是 CFI 并不会去检测这个返回值, 因此存在被攻击得可能。一条线程连续地进行系统调用, 另外一条线程持续地修改系统调用的返回地址, 就可以完成劫持的目的; 第三, 影子栈和常规栈之间存在一个常数偏移量, 因此攻击者可以通过任意地址写漏洞, 在常规栈基址的基础上, 计算得出影子栈的地址, 从而篡改影子栈的数据来绕过 CFI 检测。文献[26] 介绍了一种控制流劫持攻击的新型配件, 称之为 ACICS (argument corruptible indirect call sites)配件, 它是由成对的间接函数调用指令以及能够启用远程执行代码的函数组成, 同时又符合源程序的 CFG 逻辑。文章认为同时满足健壮性和精确性的 CFG 难以构建, 因此使用不完整指针分析构造出的 CFG 存在漏洞, 间接证明了 ACICS 配件的攻击有效性。

除了攻击细粒度 CFI 的方案, 还有一类专门针对 C++应用程序中虚函数表的劫持攻击思想, 文献[25]提出了针对 C++虚函数调用方式的攻击方法 COOP(Counterfeit Object-oriented Programming)。文章通过建立虚假对象, 劫持已经存在的 C++虚函数表, 利用漏洞程序中的主循环配件(具备循环功能的指令片段)反复调用以函数为粒度的功能配件, 直到完成攻击目的。随后文献[51]就提出了针对 COOP 的防御原型 TypeArmor。使用 use-def 数据流分析算法, 在二进制级别构造间接函数调用目的地址的合法集合, 从而阻止使用以函数为粒度的功能配件。另外经典的 CPI 也提供了防御 COOP 攻击的方法, 它通过保护虚函数表指针不被恶意修改, 从源头上防御了针对 C++虚函数表劫持的攻击。

4.3 CPI VS 信息泄露攻击

在第三章第一节中介绍了 CPI 的核心原理, 作为控制流劫持攻击经典的防御机制之一, 低性能开销得同时又提供了非常强的保护, 近两年也备受研究者的青睐。文献[52]找出了 CPI 潜在的安全隐患, 指出性能和安全性的两难问题还远远没有解决, 攻防之战还在继续。在 CPI 的论文中, 通过形式化的手段论证了对控制流劫持攻击百分之百的防御, 甚至比细粒度 CFI 更加完善, 但是要建立在 CPI 安全区域不被攻击者修改的前提下, 这在 x86-32 架构中由于段式内存管理的硬件隔离机制可以得到保证, 而在 x86-64 架构中则要通过软件的方式(类似 ASLR 的地址随机化思想)实现安全区域的保护, 这种基于信息隐藏的保护手段, 并不能从根本上阻止被攻击得可能。目前最新的时间侧信道攻击可以在内存崩溃次数很低的情况下绕过明显异常行为的检测[54, 55], 并得到程序任意内存地址的信息, 也就可以获得安全区域的内容并推测出其基地址。

随后 CPI 的作者在一篇报告中[53]解释文献[52]并没有打破 CPI 的防御, 强调 CPI 本身是可以通过形式化的方式来证明其安全性, 只是在特定的 Bugs 或漏洞存在的情况下才会导致安全性不足的问题。报告中讨论了基于强制硬件段隔离机制, 软件故障隔离机制, 以及基于信息隐藏的隔离机制三种不同的实施方案的安全性和性能开销, 总结出 CPI 在整体设计上没有安全漏洞。

然而最新的研究成果[54]表明, 基于信息隐藏的技术确实存在着安全漏洞, 包括 CPI 的安全栈模型, 攻击者可以找到被隐藏的信息, 因为移除目标程序中所有的敏感信息非常困难, 类似指针的地址和敏感的内存区域, 所以攻击者可以利用这些残余的敏感信息绕过基于信息隐藏的防御机制。文章是通过线程喷射和堆喷射技术, 申请大量的线程, 使之占满进程空间, 这可以提高发现隐藏区域的概率, 在这个申请的过程中通过系统返回的信息就可以泄露一些敏感地址, 足以让攻击者利用完成攻击。另外作者还提出了一个增强的防御方案, 通过 APM (Authenticating Page Mapper)技术建立用户级别的页错误处理例程来认证虚拟地址空间的内存读写操作, 维护一个最小尺寸的安全区域从而加强信息隐藏的有效性, 但是在可接受的范围内会适当增加一些性能开销。同时文献[55]也介绍了类似的思想, 通过攻击者指定的输入, 在目标程序进程空间中分配任意大小的空间, 关键程序的回复信息, 探测虚拟内存空间中敏感信息存放的位置, 实现对信息隐藏的发现。

5 非控制数据攻击和防御的发展

控制流劫持攻击因为防御机制的不断完善变得越来越难以利用, 因此攻击者的目标转向了非控制数据攻击的利用方式。在安全顶会中, 近两年就出现了三篇关于非控制数据攻击的文章, 足以说明研究者的重视程度。非控制流数据攻击和防御发展趋势如图 8 所示。

对于攻击, 大致经历了从思想的形成(Non-Control Attacks[4])到真实应用的实践(Dynamic Hooks[56]), 以

及最新系统的利用框架(Control-flow Bending[6], Auto Data-oriented Exploits[5])和图灵完备地证明(Data-oriented Programming[7]), 共四个阶段。前两个阶段在前面章节已有介绍, 本章重点介绍后面两个阶段的研究内容。

对于防御, 整体上划分为躲避, 随机化, 以及数据流完整性三个方面。虽然 DFI[28]给出了理论上的防御支撑, 但因性能开销大且需要过多地人工干预, 因此无法被实际应用。细粒度的随机化思想(DSR[32])性能开销也难以接受, 而躲避的防御机制(SIDAN[58], Module protection[37], ValueGuard[57])却不具备通用性。硬件支持的数据流隔离思想 HDFI[60]和内核级的数据流完整性(Kernel with DFI[30])虽然减少了性能开销, 但也减少了保护数据的范围, 牺牲了一定的安全性。总体上看, 目前没有一种有效的安全机制能够防御新提出地 DOP 攻击。

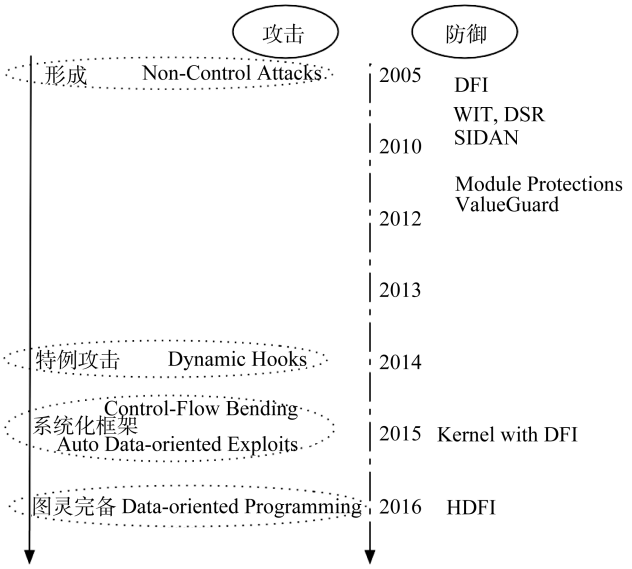

图 8 非控制数据攻击和防御的发展

5.1 非控制数据攻击的发展

5.1.1 系统化攻击框架的形成

文献[6]提出的控制流弯曲(Control-Flow Bending, CFB), 证明了即使最理想的细粒度 CFI 也不能完全抵御 CFB 的攻击方式。所谓细粒度 CFI, 即间接分支指令的目的地址严格依据程序的 CFG, 毫无例外。传统的非控制数据攻击利用内存漏洞覆盖数据而非代码指针, 但是 CFB 会通过污染内存数据间接地修改一些控制流相关的数据, 例如函数返回地址和间接分支指令的目的地址, 但修改后仍然满足细粒度 CFI 的规则。为了更改程序控制流, 攻击者往往需要劫持一些中转函数, 即程序中能够通过调用参数设置自己返回地址的函数, 借此完成攻击的意图。类似 CFB, 文献[56]打破了传统钩子系统会明显改变控制流的缺陷, 在内存中污染非控制数据, 然后程序运行时通过临时引用预先污染的内存数据, 间接污染控制流相关数据, 使得控制流转移看似合法, 以此来绕过基于控制流监控的检测机制。该原型系统通过静态切片分析和符号执行自动地筛选出能够被钩子系统利用的安全关键路径, 通过非控制数据隐藏控制流劫持的思想绕过现存的钩子检测技术。文献[5]则系统化地提出了数据导向攻击(Data-oriented Exploits, DOE)的整体框架。文章提出的数据流缝合技术(Data-flow Stitching),能够自动化地实施非控制数据攻击的整体过程, 通过自动筛选出安全相关的数据流片段, 然后通过引用其他内存中的数据来污染安全数据完成攻击目的。整个攻击过程可以绕过细粒度 CFI 和 DEP, 以及大多数 ASLR 防御体系。

5.1.2 图灵完备攻击框架的形成

类似 ROP 证明代码重用攻击的图灵完备性, 文献[7]首次证明了非控制数据攻击的图灵完备性, 构建了不依赖于特定数据或者函数的攻击框架(Data-oriented Programming, DOP), 能够在不具备地址泄露的前提下绕过 ASLR, 能够更改内存页的权限标志位使得代码注入攻击再次生效。DOP 模拟了一种最小限度的语言 MINDOP(Minimal Language DOP), 可以由 x86 配件集成六种基本操作, 能够满足 MINDOP 图灵完备的特性, 因此 DOP 是一种图灵完备的攻击方式。其核心思想是通过内存错误劫持参与运算的数据变量, 同时劫持用于循环或条件判断的数据, 保证不违反控制流逻辑的情况下完成恶意攻击。其配件构造由于无法像 ROP 一样随心所欲地使用代码和寄存器, 利用传统的配件会导致很大的副作用, 如果使用的寄存器被其他无关语句使用, 那么存储的值就会被修改, 从而导致攻击失败, 所以 DOP 用到的数据都是存在内存中, 在每次运算之前才会加载到寄存器中, 运算完成后又会马上写回内存, 确保数据不被其他无关操作破坏。DOP 的攻击实现需要依赖一个主循环配件, 这个循环配件中的变量可控, 循环体内有足够的功能配件, 如果某个函数能够完成某些功能, 并且这个函数的参数可以被控制, 那么这个函数就是一个配件。由于所有 DOP 攻击所用配件从 C 语言的角度来看都是对指针的操作, 所以可以通过赋值的方式读取攻击所需要的函数地址, 因此随机化方法是无效的。

5.2 非控制数据防御的发展

非控制数据攻击的防御目前还处于比较被动的状态。攻击方法逐渐完善起来, 然而防御思想只是简

单地优化, 并没有创新性的突破。防御方式如第三章所述, 整体总结为三类。

5.2.1 躲避的防御思想

第一, 躲避的防御思想[4]。这里以图 5 所示的非控制数据攻击实例来说明, 针对它的防御机制如图 9 所示, 把关键安全数据变量 *authenticated* 的生命周期缩至最短, 以至于攻击者无法在漏洞触发点之前污染它, 也就成功抵御了此类型的攻击。它的优点自然是针对性极强, 性能开销几乎为零, 但无法防御所有的非控制数据攻击, 通用性不够并且需要熟知程序语义。类似方法还有文献[57]提出的 ValueGuard, 与在函数返回地址前添加 Canary 一样, 本文则是在安全关键变量前添加哨兵, 如验证用户身份的布尔变量等, 等系统引用该变量时会检查其对应的标记是否被污染, 从而保护变量的安全性。其简单的思想没有给程序造成额外的开销, 但是需要在源码的基础上重新编译程序。文献[37]提出了一种基于语言的安全防御策略, 把程序中的关键安全变量声明为特殊类型, 在动态引用时要求只有类型匹配的读写操作才能访问。文献[58]则以系统调用为切入点, 检测程序的异常行为。由于传统以系统调用序列为基础的检测模型欠缺准确性和完整性, 所以作者提出了增添对系统调用参数的监控, 如系统调用所需要的系统调用号, 配置参数, 以及上下文相关的信息等, 保护这些数据不被恶意篡改来防御非控制数据攻击。原型首先通过源码级别地静态分析得出系统调用所依赖的变量, 然后通过插桩技术检测程序触发系统调用时的数据流是否合法, 从而判断是否发生攻击。

```
void do_authentication(char *user, ...){
  int authenticated =0 ;
  ...
  while( !authenticated ){
    type = packet_read() ;   //漏洞触发点
    if ( auth_password( user, password ) )
        authenticated = 1 ;
    if ( authenticated ) break ;
  }
  do_authenticated ( pw ) ;
}
```

漏洞程序

```
void do_authentication(char *user, ...){
  ...
  while( !authenticated ){
    type = packet_read() ;
    int authenticated =0 ;
    if ( auth_password( user, password ) )
        authenticated = 1 ;
    if ( authenticated ) break ;
  }
  do_authenticated ( pw ) ;
}
```

非漏洞程序

图 9 躲避的防御思想示例

5.2.2 随机化的防御思想

第二, 数据随机化的思想[32]。就是将内存中数据以随机的方式存储, 通过掩码异或操作来隐藏真实的数据含义, 因此攻击者无法污染正确的数据而无法完成攻击。基本的转换思想是把关键的变量通过掩码异或操作后再存储在内存中, 在使用时再次异或还原, 不同的变量使用不同的随机掩码, 这样攻击者就无法猜测掩码, 也就无法还原具体的数据值, 即使污染也是写入无法识别的数据, 从而引发系统报错。其中有两种特例需要单独考虑, 第一就是间接引用的数据, 因为静态分析会得到多个可能的值, 所以使用不同的掩码则会造成动态运行时地还原错误, 因此使用指针分析算法为同一个间接调用的所有可能变量分配同一个掩码。第二就是变量同名问题, 那么这些同名的不同变量会使用相同的掩码, 也就意味着溢出其中一个变量到另外同名的变量合法且无法检测, 这对于攻击者就是一个潜在的利用点, 为了解决同名问题, 把使用相同掩码的变量对象通过页映射机制, 分配到相互隔离的内存区域, 那么以上潜在的利用点就会触发页错误而中断系统。数据空间随机化相比较地址空间随机化的优点是更大的随机空间(32 位的数据则有 2^{32} 的空间可能性), 但也存在最坏百分之三十的性能损耗且需要编译器的支持。

5.2.3 数据流完整性的防御思想

第三, 是从此类攻击的本质上进行防御, 严格检查对关键安全数据的读写是否合法。其实现方式可以给内存添加标记位[38, 59], 然后通过污点跟踪技术保障数据的安全, 也可以通过数据流分析[28], 确定合法的数据来源, 添加访问策略或者构造数据流图 DFG 来保障数据的安全。

文献[59]提出了指针污点分析的方法, 传统的污点分析仅仅标记污点位, 而本文却同时使用污点位和指针位两个标志位, 一方面对于外部输入这类不可信的数据进行标记, 同时对于指针引用是否合法也标记出来。程序在运行的时候同时传播两个标志位的信息, 当外部数据被用作指针且解引用不合法时, 则认定为恶意行为。虽然作者提出的想法是有效的, 但需要一个数据结构维护标志位的信息, 并且需要为这个数据结构新开辟一个安全的影子内存, 其复杂地实现使得效率不是很理想, 而且指针分析过程也不可避免存在误报的情况, 同时如果攻击者

不采取覆盖指针污染内存数据的攻击方式, 就会绕过以上的防御方法。

经典的数据流完整性 DFI[28]首先通过静态分析标记所有变量的赋值和引用, 生成每个变量引用的合法集合, 构建数据流图 DFG。其次动态监控程序的每次赋值操作, 更新对应的标志且检查是否合法。若发现不合法的赋值操作, 则触发警报处理例程。

由于传统的 DFI 实现起来性能开销极大, 后来就提出了写完整性测试 WIT[29], 它本质上是从粗粒度的角度重新划分了变量赋值的合法策略。在静态分析时, 对于每个内存写操作和每个间接控制转移操作, 计算出其可能的目的地址集合, 然后使用一种着色表结构对写操作和转移操作以及所有静态分析出的有效目的地址分配不同颜色, 最后在动态分析时基于代码插桩, 检测每个写操作和转移操作的目的地址是否对应正确的颜色。同时为了再次降低效率, 在静态分析时就筛选出一些具备安全特性的内存写指令, 那么在运行时就可以不做检查。除此之外还提供了额外的保护, 改变栈的布局以隔离安全和不安全的本地变量, 在不安全的对象周围布置一些 Guards 或 Canaries, 对内存敏感的函数如 *malloc(), free()*进行封装处理。

除了优化安全策略, 文献[30]则是通过减小保护数据的集合来降低效率, 它仅仅防止提权攻击, 使用改进的 DFI 思想保护内核数据的安全性, 使得性能开销在可接受的范围之内。首先发现相关的数据, 如收集函数返回值, 条件分支数据以及这些数据的依赖关系, 其余不相关的数据均不在保护范围之内。其次使用优化的 DFI 思想保护数据的完整性, 因为读指令比写指令多, 所以检查写指令来替代读指令, 且大多数是写指令不相关, 所以使用隔离机制替代内联检查, 另外大多数写指令是安全的, 所以使用静态分析核对即可。

还有利用硬件机制提高效率的防御思想, 文献[60]提出了硬件协助的数据流隔离思想(Hardware-Assisted Data-flow Isolation, HDFI), 用一个比特位来标记敏感读写操作是否可信, 例如函数的返回地址, 用新指令 sdset1 存数据时设置标记位, 然后用新指令 ldchk1 读数据时检查标记位, 对比前后是否一致。正常的 sd 指令并不会去设置标记位, 所以如果 sd 之后再用 ldchk1 引用时, 就会触发标记位不匹配而报错。对于大多数内存安全的防御机制都存在性能开销过大的问题, HDFI 则在策略上使用隔离机制减少要保护数据的范畴, 且通过筛选机制仅仅保护敏感的数据如代码指针和重要的内核数据, 在实现上又采用 CPI 中的安全栈[21]和硬件协助的技术, 因此很多程度上提高了原型系统的效率。

6 讨论和总结

6.1 控制流劫持攻击和防御

首先, 综合分析经典的控制流劫持防御机制, 在最理想的情况下得它们的评估结果如图 10 所示, y 轴表示不同防御机制的性能开销, x 轴表示不同防御机制的安全性度量。从结果中可以得出, CPI 在安全性和效率上都优于细粒度的 CFI 策略, 而 CPS 更是首次提出性能开销低于 2%且安全保障程度相当可观的实用性防御机制[61]。目前在主流编译器 Clang 中已经实现了对安全栈的支持[62]以及对粗粒度 CFI 的支持[63], 只需要在编译时添加相应的参数就可以实现对目标程序的保护, 但也只是在开源的层次, 离工业部署还需再更多实践的认可。

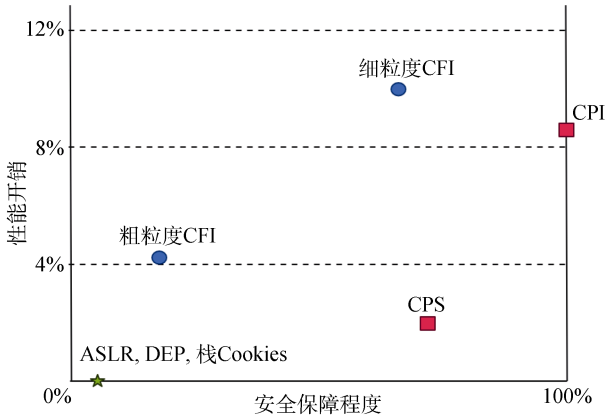

图 10 经典控制流劫持攻击的防御机制效率分析

其次, 从启发式防御、粗粒度 CFI, 以及 CPI 三方面总结不同控制流劫持攻击防御机制的评估结果于表 4 中。其中 ROP 类型一栏表示此类防御机制能够防御的 ROP 类型, ret-based 意味着仅仅只能防御以 Ret 指令结尾的配件类型, 而 ALL 表示除此之外还能防御以 Jmp 指令和 Call 指令结尾的配件类型。第二栏表示防御是否需要源代码。第三栏表示是否需要对文件进行二进制重写。二进制开销一栏表示额外添加的防御代码对二进制文件大小的影响程度。运行时开销表示添加防御机制后, 应用程序产生的额外时间。对于防御效果, 因为不同的方法有不同的衡量标准, 因此这里仅仅表示在作者论述的保护集合范围内, 被移除的攻击配件或被检测出的攻击配件占全部恶意配件的比例, 对于没有给出比例的方法, 这里给出其评估方法的描述。因为有一些作者并没提供全部的实验结果, 所以部分内容为空。分析

表 4 得出: 启发式的防御方法因为需要动态地检测目标程序, 所以它们的性能开销往往较大, 即使防御效果不错, 但却不能防御多种类型的攻击手段; CFI 的具体实现方案几乎不依赖程序源码, 但需要静态分析二进制文件, 分析算法的准确度直接影响了防御效果的程度, 同时需要对二进制文件进行重写, 因此其二进制开销较大。CPI 的实现主要依赖于程序源码的分析, 准确地识别程序中的敏感指针, 不需要对二进制文件进行分析, 因此防御效果较好且性能开销理想。

6.2 非控制数据攻击和防御

从躲避、随机化、通用三个方面总结不同非控制数据攻击防御机制的评估结果于表 5 中。通用性一栏表示该防御机制是否依赖程序的源码。部署一栏表示该防御思想是否被应用于实际的系统, 看来对于非控制数据攻击的防御研究在实践方面还有待商榷。主要威胁一栏表示该防御方法受到的主要攻击向量有哪些。保护对象一栏表示该防御方法主要保护的数据范畴。分析表 5 得出: 躲避的防御方法往往针对传统的非控制数据攻击, 对于完备的 DOP 攻击没有防御能力, 它们仅仅在熟知程序语义的情况下, 保护传统的安全变量而已, 而且过多依赖程序源码, 不具备对商业软件的通用性; 随机化的思想受到信息泄露的严重威胁; 针对攻击本质的通用防御思想几乎都依赖于目标程序的静态分析结果, 所以不准确的指针分析会导致防御机制存在被攻击者绕过得可能。

表 4 控制流劫持攻击防御机制综合分析

类别	防御名称	ROP 类型	源码	二进制重写	二进制开销(平均)	性能开销(平均)	防御效果
启发式	DROP[34]	Ret-based	×	√	-	530%	100%
	ROPDefender[35]	Ret-based	×	√	0%	217%	100%
	G-Free[42]	ALL	√	×	25.9%	3.1%	100%
	CCFIR[44]	ALL	×	√	30%	3.6%	100%
粗粒度 CFI	Bin-CFI[50]	ALL	×	√	139%	4.29%(C)8.54%(C++)	92.68%
	KBouncer[47]	ALL	×	√	—	1%	93.6%
	ROPGuard[46]	Ret-based	×	√	—	0.48%	四种不同方式检测一种 ROP 攻击
	ROPecker[48]	ALL	×	×	19M	2.6%	100%
CPI	CPI[21]	ALL	√	×	—	8.4%	100%
	CPS[21]	ALL	√	×	—	1.9%	对可交换的代码指针无效

表 5 非控制数据攻击防御机制综合分析

类别	防御名称	性能开销(平均)	通用性	部署	主要威胁	保护对象
躲避	ValueGuard[57]	>100%	源码	×	DOP	数据缓冲区
	SIDAN[58]	0.55%	源码	×	DOP	系统调用参数
随机化	MoudulePro[37]	160%	源码	×	DOP	传统安全变量(表 2)
	DSR[32]	15%	二进制	×	信息泄露	所有敏感数据变量
通用	DFI[28]	104%	二进制	×	不准确的指针分析	所有敏感数据变量
	WIT[29]	10%	二进制	×	不准确的指针分析	敏感数据变量以及敏感函数
	Kernel with DFI[30]	11%	二进制	×	不准确的指针分析	敏感内核数据
	HDFI[60]	<2%	源码	×	代理人攻击	敏感代码指针和敏感内核数据

6.3 讨论

对于理论上的防御思想, 都需要对程序进行一定地事先分析, 例如 CFI 需要构造 CFG, CPI 需要分析敏感指针, DFI 需要构造 DFG, WIT 需要分析敏感数据等。要么从源码分析出精确的 CFG, 稳定的敏感指针关系, 以及完整的 DFG; 要么就是从二进制角度通过反汇编分析出次精度的 CFG, 不稳定的指针关系, 以及不完整的 DFG。依赖源码进行分析的典型防御如 CPI, 通过静态分析找出程序中所有需要保护的内存对象, 借此来达到保证所有代码指针安全的目的。从二进制角度进行分析的典型防御如 CFI, 因为难以构造同时满足稳定性(Soundness)且完整(Completeness)性的 CFG, 所以导致使用特殊配件的控制流劫持攻击继续有效[26], 这是由间接分支指令操作数需要依赖运行时信息的本质特征所决定。文献[66]提出的二进制分析平台 angr 表明, CFG 恢复的基础挑战就是如何确定间接跳转指令的合法目的地址, 在 CFG 的处理上采用保守的静态分析方法, 得出

接近于精确 CFG 的超集或尽可能多的交集的结果。

大多数学术界提出的保护措施没有得到工业界的应用, 主要有三方面原因: 性能损失过大, 与现有系统兼容性不够, 防御效果不佳。以表 5 中的防御机制为例: 如 DFI, 虽然防御效果很好, 但平均性能开销超过了百分之百; 如 HDFI, 其性能开销比较低, 但引入了新的指令集, 使其兼容性较差; 如 SIDAN, 仅仅监控系统调用参数的合法性, 其准确性和有效性还有待探讨。除此之外, 主流编译器 Clang 支持的粗粒度 CFI 和安全栈防御思想, 因为需要对源码重新编译, 因此也停留在开源的层面并没有广泛部署。Intel 最新提出的 Control-Flow Enforcement Technology(CET), 即将在硬件层面实现影子栈和间接转移指令跟踪技术, CET 之所以即将获得工业界的部署, 是因为影子栈的提出和软件层面的实现都得到研究者广泛的认可, 因此 Intel 进一步在硬件层面实现, 使其性能和兼容性可以得到保障。

7 未来研究方向

主流的控制流劫持攻击逐渐发展为新型的代码重用攻击, 其防御机制也在逐步完善, 虽然仍然面临信息泄露的威胁, 但整体上直接的控制流劫持利用成本越来越高。危害程度相当的非控制数据攻击发展为自动化且图灵完备的攻击框架, 但是目前还没有一种有效的防御机制出现, 巨大的性能开销使得防御暂时落后于攻击。综合来看, 在攻击方面, 从主流的控制流劫持转向非控制数据或两者的协作攻击方式。在防御方面, 逐步转向以 DFI 为基本思想, 提出一系列的优化方案, 同时相对完善的 CFI 在一定程度上有借鉴意义, 但仍需要抓住 DFI 的核心思想作为研究重点。

图灵完备的非控制数据攻击框架 DOP 的提出, 势必会掀起一阵针对 DOP 防御的研究热潮, 目前还没有完全有效且能与之对抗的防御机制。因为 DOP 利用大量的内存错误实施数据导向的配件, 因此需要完整的强制内存安全机制才能防御, 但是完整的防御也会造成大量的性能开销, 为了降低开销而减少要保护的数据集合, 那么防御机制的安全性也会随之降低; 另外因为 DOP 的成功实施仍然需要部分的非控制数据指针, 所以细粒度的数据随机化策略能够防御它, 但随机化粒度和性能的平衡仍然是一个需要研究的问题; 使用基于硬件或者软件隔离的机制来防御 DOP, 需要精确地识别所有可辨认的指针数据, 但是程序中存在各种各样指针数据, 所以这是个相当大的挑战。因此, 未来对于 DOP 的防御一定会成为安全领域的研究热点之一。目前有几种可行的思想: 减少保护的数据或使用更高效的硬件机制来减少经典防御思想的性能开销, 或者提出针对攻击本质的更为简洁的防御思想。

DOP 的出现, 能够有效的协助控制流劫持攻击绕过较为完善的 CFI 防御体系。例如 CFB 利用数据攻击隐藏控制流劫持的真实目的, 文献[64]实现了只修改少量函数指针和数据的前提下, 在内核中注入恶意 ROP 且实现常驻的攻击过程。因此未来提出更多利用非控制数据攻击联合控制流劫持攻击的协作式攻击方案也是值得研究的。同时类似 ROP 自动化攻击框架 Q[40], 研究 DOP 自动化利用框架也是具有意义的。

协作攻击的思想已然出现, 那么协作防御的思想也是能够预测到的。例如 DFI 结合 CFI 中影子栈的思想, 首先静态分析所有敏感数据的合法依赖路径, 将信息存储在一个影子内存中, 其次在数据被操作时, 依据影子内存中的信息判断操作的合法性, 检查数据流是否位于其合法的依赖路径集合中, 若不合法则报警处理, 当然这只是一个粗略的设想, 其可行性还需要进一步论证和实验。

8 结束语

一切皆数据, 哪个层次出了问题, 也就造就了不同的漏洞形式, 形成不同的攻击方式。内存是存储数据的重要载体, 利用污染内存数据形成控制流劫持攻击和非控制数据攻击, 控制程序甚至控制系统, 造成的危害不言而喻。

本文就以这两种攻击形式为基础, 综述内存安全的研究现状。首先提出了简化的内存安全模型, 突出了内存数据污染攻击的范畴和本质。然后对内存中的安全关键数据进行分析, 得出九类控制流转移数据和五类非控制安全数据, 攻击者污染以上数据进行控制流劫持攻击和非控制数据攻击, 防御者保护以上数据进而抵御这两类攻击。其次按照时间发展的顺序总结了内存数据污染攻击的发展脉络, 着重论述了新型的控制流劫持攻击及其防御之间的交互过程, 和非控制数据攻击图灵完备的研究现状。最后对不同防御机制的效率进行评估, 并提出未来内存安全可能的研究方向。

致谢 在此向对本文工作提出指导的各位同门以及提出建议的评审专家表示感谢。

参考文献

[1] Ú. Erlingsson, Y. Younan, and F. Piessens. “Low-level software

security by example." Springer Berlin Heidelberg, *Handbook of Information and Communication Security*, pp. 633-658, 2010.

[2] "CWE/SANS Top 25 Most Dangerous Software Errors," MITRE, http://cwe.mitre.org/top25, 2011.

[3] L. Szekeres, M. Payer, T. Wei, and D. Song, "SoK: Eternal War in Memory," IEEE Computer Society, *IEEE Symposium on Security and Privacy*(*SP'13*), Vol.12, pp. 48-62., 2013.

[4] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer, "Non- control-data attacks are realistic threats," USENIX Association, in Proc of *the 14th Usenix Security Symposium*(*Usenix Security'05*), Vol.53, pp. 12-12. 2005.

[5] H. Hu, Z. L. Chua, S. Adrian, P. Saxena and Z. Liang, "Automatic Generation of Data-Oriented Exploits," USENIX Association, in Proc of *the 24th Usenix Security Symposium*(*Usenix Security'15*), pp. 177-192. 2015.

[6] 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 Security'15*), pp. 161-176, 2015.

[7] H. Hu, S. Shinde, S. Adrian, Z. Leong Chua, P. Saxena, and Z. Liang, "Data-Oriented Programming: On the Expressiveness of Non-control Data Attacks," *IEEE Symposium on Security and Privacy (SP'16)*, pp. 969-986, 2016.

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

[9] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton, "StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks," *In: USENIX Security Symposium*(*Usenix Security'98*), San Antonio, Texas, pp. 63-78, January, 1998.

[10] S. Andersen, "Changes to Functionality in Microsoft Windows XP Service Pack 2 Part 2: Network Protection Technologies." Microsoft Technical Document, 2004.

[11] Molnar, Ingo. "METHOD AND APPARATUS FOR CREATING AN EXECUTION SHIELD." US, WO/2004/095275. 2004.

[12] S. Bhatkar, D. C. Duvarney, and R. Sekar, "Address obfuscation: an efcient approach to combat a broad range of memory error exploits," Proceedings of *Usenix Security Symposium*(*Usenix Security'03*), pp. 105-120, 2003.

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

[14] 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.

[15] 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.

[16] 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.

[17] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-oriented programming: Systems, languages, and applications," *Acm Transactions on Information & System Security* 15(1), 2012.

[18] 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.

[19] 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.

[20] 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.

[21] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, "Code-Pointer Integrity," USENIX Association, *the 11th USENIX Symposium on Operating Systems Design and Implementation*(*OSDI'14*), pp. 147-163, 2014.

[22] 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 Security'14*), pp. 401-416, 2014.

[23] 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.

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

[25] 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.

[26] 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.

[27] C. Liebchen, M. Nergro, P. Larsen, M. Conti, S. Crane, L. Davi, M. Franz, M. Qunaibit, and 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.

[28] C. Miguel, C. Manuel, W. Harris, and T. Kim, “Securing Software by Enforcing Data-flow Integrity,” Proceedings of *the 7th Symposium on Operating Systems Design and Implementation*(*OSDI’06*), pp. 147—160, 2006.

[29] A. Periklis, C. Cristian, R. Costin, C. Manuel and C. Miguel, “Preventing Memory Error Exploits with WIT”, *Proceedings of the 2008 IEEE Symposium on Security and Privacy*(*SP’08*), pp. 263—277, 2008.

[30] C. Song, B. Lee, K. Lu, W. Harris, T. Kim, and W. Lee, “Enforcing Kernel Security Invariants with Data Flow Integrity,” *Network and Distributed System Security Symposium* (*NDSS '16*), 2016.

[31] Z. Huanguo, H. Wenbao, L. Xuejia, L. Dongdai, M. Jianfeng, L. Jiahua, “Survey on cyberspace security,” Scientia Sinca Information, vol. 46, no. 2pp. 125-146, 2016.
(张焕国, 韩文报, 来学嘉, 林东岱, 马建峰, 李建华, “网络空间安全综述”, 中国科学: 信息科学, 第 46 卷, 第 2 期, 125-164, 2016。)

[32] S. Bhatkar, and R. Sekar, “Data Space Randomization,” Detection of Intrusions and Malware, and Vulnerability Assessment, Lecture Notes in Computer Science, vol 5137, Springer Berlin Heidelberg, pp. 1-22, 2008.

[33] N. Burow, S. A. Carr, S. Brunthaler, M. Payer, J. Nash, and P. Larsen et al, “Control-flow integrity: precision, security, and performance,” arXiv:1602.04056 [cs.CR], 2016.

[34] P. Chen, H. Xiao, X. Shen, X. Yin, B. Mao, and L. Xie, “Drop: detecting return-oriented programming malicious code,” Proceedings of *the 5th International Conference on Information Systems Security*(*ICISS’09*), pp. 163-177, 2009.

[35] L. Davi, A. R. Sadeghi, and M. Winandy, “ROPdefender: A Detection Tool to Defend Against Return-oriented Programming Attacks,” Proceedings of *the 6th ACM Symposium on Information, Computer and Communications Security* (*ASIACCS’11*), pp. 40-51, 2011.

[36] G. S. Kc, A. D. Keromytis, and V. Prevelakis, “Countering Code-injection Attacks with Instruction-set Randomization,” Proceedings of the 10th ACM Conference on Computer and Communications Security(CCS’03), pp. 272-280, 2003.

[37] C. Schlesinger, K. Pattabiraman, N. Swamy, D. Walker, and B. Zorn, “Modular Protections Against Non-control Data Attacks,” Proceedings of the 2011 IEEE 24th Computer Security Foundations Symposium (CSF '11), pp. 131-145, 2011.

[38] J. NEWSOM, “Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software,” Network and Distributed System Security Symposium Conference Proceedings(NDSS’05), 2005.

[39] L. Tong, S. Gang, and M. Dan, “A Survey of Code Reuse Attack and Defense Mechanisms,” *Journal of Cyber Security*, vol(2), pp. 15-27, 2016.
(柳童, 史岗, 孟丹, “代码重用攻击与防御机制综述”, 信息安全学报, 第二期, 15-27, 2016)

[40] E. J. Schwartz, T. Avgerinos, and D. Brumley, “Q: Exploit Hardening Made Easy,” *Proceedings of the 20th USENIX Conference on Security*(*SEC’11*), pp. 1-16, 2011.

[41] T. Bletsch, X. Jiang, and V. Freeh, “Mitigating Code-reuse Attacks with Control-flow Locking,” Proceedings of *the 27th Annual Computer Security Applications Conference*(*ACSAC’11*), pp. 353-362, 2011.

[42] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda, “G-Free: Defeating Return-oriented Programming Through Gadget-less Binaries,” Proceedings of the 26th Annual Computer Security Applications Conference(ACSAC’10), pp. 49-58, 2010.

[43] Tymburib, Mateus, REA Moreira, Quint, and FM O Pereira, “Inference of peak density of indirect branches to detect ROP attacks,” *International Symposium on Code Generation and Optimization*(*CGO’16*), pp. 150-159, 2016.

[44] 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.

[45] M. Kayaalp, M. Ozsoy, N. Abu-Ghazaleh, and D. Ponomarev, “Branch Regulation: Low-overhead Protection from Code Reuse Attacks”, Proceedings of *the 39th Annual International Symposium on Computer Architecture*(*ISCA’12*), pp. 94-105, 2012.

[46] I. Fratric, “ROPGuard: Runtime Prevention of Return-Oriented Programming Attacks,” https://github.com/ivanfratric/ropguard/blob/master/doc/ropguard.pdf, 2012.

[47] V. Pappas, M. Polychronakis, and A. D. Keromytis, “Transparent ROP exploit mitigation using indirect branch tracing,” *Proceedings of the 22Nd USENIX Conference on Security*(*SEC’13*), pp. 447-462, 2013.

[48] 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*), 2014.

[49] 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.

[50] M. Payer, A. Barresi, and T. R. Gross, “Fine-Grained Control-Flow Integrity Through Binary Hardening,” Detection of Intrusions and

Malware, and Vulnerability Assessment, pp. 144-164, 2015.

[51] V. V. D. Veen, C. Giuffrida, E. Goktas, M. Contag, A. Pawoloski, and X. Chen, et al, "A Tough Call: Mitigating Advanced Code-Reuse Attacks at the Binary Level," *2016 IEEE Symposium on Security and Privacy* (*SP'16*), pp. 934-953, 2016.

[52] I. Evans*, S. Fingeret†, J. Gonza´Lez†, U. Otgonbaatar†, T. Tang†, and H. Shrobe†, et al, "Missing the point(er): on the effectiveness of code pointer integrity," *2015 IEEE Symposium on Security and Privacy* (*SP'15*) pp. 781-796, 2015.

[53] G. Candea, V. Kuznetsov, M. Payer, L. Szekeres, and D. Song, "Poster: Getting The Point(er): On the Feasibility of Attacks on Code-Pointer Integrity," Proceedings *Kuznetsov2015PosterGT*, 2015.

[54] S, E. Gökta¸Sgökta, R. Gawlik, B. Kollenda, , E. Athanasopoulos, G. Portokalidis, and C. Giuffrida, et al, "Undermining Information Hiding (and What to Do about It)," *25th USENIX Security Symposium* (*USENIX Security'16*), pp. 105-119. 2016.

[55] A. Oikonomopoulos, E. Athanasopoulos, H. Bos, and C. Giuffrida, "Poking Holes in Information Hiding," *25th USENIX Security Symposium* (*USENIX Security'16*), pp. 121—138, 2016.

[56] S. Vogl, R. Gawlik, B. Garmany, T. Kittel, J. Pfoh, and C. Eckert, et al, "Dynamic hooks: hiding control flow changes within non-control data," *Proceedings of the 23rd USENIX Security Symposium*(*USENIX Security'14*), 2014.

[57] S. Van Acker, N. Nikiforakis, P. Philippaerts, Y. Younan, and F. Piessens, "ValueGuard: Protection of Native Applications Against Data-only Buffer Overflows," *Proceedings of the 6th International Conference on Information Systems Security* (*ICISS'10*), pp. 156-170, 2010.

[58] J. C. Demay and E. Totel and F. Tronel, "SIDAN: A tool dedicated to software instrumentation for detecting attacks on non-control-data," *Fourth International Conference on Risks and Security of Internet and Systems* (*CRiSIS'09*), pp. 51-58, 2009.

[59] L. Xiaolong, and Z. Tao, "Improved defense method against non-control-data attacks," Application Research Computers, vol. 30, no. 12, pp. 3762-3766, Dec, 2013.
(刘小龙, 郑滔, "一种针对非控制数据攻击的改进防御方法," 计算机应用研究, 第 30 卷第 12 期, 3762-3766, 2013 年 12 月

[60] C. Song, H. Moon, M. Alam, I. Yun, B. Lee, T. Kim, W. Lee, and Y. Paek, "HDFI: Hardware-Assisted Data-Flow Isolation," *IEEE Symposium on Security and Privacy* (*SP'16*), pp. 1-17, 2016.

[61] "Code-Pointer Integrity Fast and precise control-flow hijack protection," Dependable Systems Lab, http://dslab.epfl.ch/proj/cpi/, 2016.

[62] "Clang with SafeStack", Clang 5 documentation, http://clang.llvm.org/docs/SafeStack.html, 2016.

[63] "Clang with Control-flow Intergrity", Clang 5 dicumentation, http://clang.llvm.org/docs/ControlFlowIntegrity.html, 2015

[64] S. Vogl, J. Pfoh, T. Kittel, and C. Eckert, "Persistent Data-only Malware: Function Hooks without Code," *Network and Distributed System Security Symposium*.(*NDSS'14*), Feb, 2014.

[65] L. Davi, P. Koeberl, and A. R. Sadeghi, "Hardware-Assisted Fine-Grained Control-Flow Integrity," *Design Automation Conference* (*DAC'14*), pp. 1-6, 2

[66] S. Yan, R. Wang, C. Salls, S. Nick, P. Mario, D. Andrew, G. John, F. Siji, C. Hauser, C. Kruegel, and G. Vigna, "SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis," *2016 IEEE Symposium on Security and Privacy* (*SP'16*), pp. 138-157, 2016

马梦雨 于 2014 年在西南民族大学计算机科学与技术学院网络工程专业获得学士学位。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: 内存安全, 恶意行为分析等。Email: mamengyu@iie.ac.cn

陈李维 于 2014 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安全研究所第五研究室助理研究员。研究领域为计算机系统安全。研究兴趣包括: 信息安全, 系统安全, 计算机架构和 VLSI 设计。Email: chenliwei@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