

Windows 内存防护机制研究

鲁婷婷¹, 王俊峰^{1,2}

(1. 四川大学计算机学院, 四川 成都 610065;
2. 四川大学空天科学与工程学院, 四川 成都 610065)

摘要: 近30年来, 攻击者利用 Windows 内存漏洞发起的攻击事件层出不穷, 其惯用手段是攻击控制数据以劫持执行流。为此, 微软在 Windows 上加注了层层防护以遏止此类攻击, 但现阶段部署的防护机制无法阻止针对非控制数据的攻击。鉴于目前研究 Windows 内存防护机制的文献寥寥无几, 对 Windows 采用的各种内存防护机制及其突破技术进行了深入研究, 并详述非控制数据防护的研究现状, 在此基础上, 分析了 Windows 内存防护面临的挑战, 并讨论了内存防护的未来之路。

关键词: Windows 内存防护; 内存防护机制; 控制数据防护; 非控制数据防护

中图分类号: TP309

文献标识码: A

doi: 10.11959/j.issn.2096-109x.2017.00208

Research on Windows memory protection mechanism

LU Ting-ting¹, WANG Jun-feng^{1,2}

(1. College of Computer Science, Sichuan University, Chengdu 610065, China;
2. School of Aeronautics and Astronautics, Sichuan University, Chengdu 610065, China)

Abstract: Over the past three decades, attacks exploiting Windows memory holes have emerged in an endless stream, and the usual means is to attack control-data to hijack the execution flow of the program. To this end, Microsoft has added layers of protection mechanisms to Windows to prevent such attacks. But at this stage, the defensive mechanisms deployed on Windows cannot prevent attacks against non-control-data. In view of the published articles lacking comprehensive study of Windows memory protection mechanism, a detailed introduction to Windows memory protection mechanism and its breakthrough techniques, and non-control-data protection research status was conducted. On this basis, the challenges of Windows memory protection were analyzed and the future of memory protection was discussed.

Key words: Windows memory protection, memory protection mechanism, control-data protection, non-control-data protection

1 引言

近30年来, 针对内存溢出漏洞发起的攻

击数不胜数, 从第一代网络蠕虫 Morris^[1]到 Heartbleed^[2], 每次攻击都造成灾难性后果。此类攻击大多通过缓冲区溢出修改影响程序执行流程

收稿日期: 2017-09-06; 修回日期: 2017-09-18。通信作者: 王俊峰, wangjf@scu.edu.cn

基金项目: 国家重点研发计划基金资助项目 (No.2016QY06X1205, No.2016YFB0800605); 国家自然科学基金资助项目 (No.91438120, No.91338107, No.91438119); 四川省重点研发计划基金资助项目 (No.17ZDYF2583)

Foundation Items: The National Key Research and Development Program (No.2016QY06X1205, No.2016YFB0800605), The National Natural Science Foundation of China (No.91438120, No.91338107, No.91438119), The Technology Research and Development Program of Sichuan Province (No.17ZDYF2583)

的控制数据,如返回地址^[3]、虚表指针^[4]等,进而劫持控制流,使其指向定制的 shellcode,执行恶意代码以达到攻击目标,如图1所示。安全界针对此类攻击的防护已有大量的研究成果,如控制流完整性(CFI, control flow integrity)^[5]、代码指针完整性(CPI, code-pointer integrity)^[6]等,多种成熟的防护技术也已部署到商业应用。但随着控制数据防护技术的大范围部署,另一类针对非控制数据的攻击应运而生。非控制数据攻击完全遵循合法的控制流,通过非法写程序的关键数据(如决策数据^[7])或者读敏感数据破坏数据的完整性和机密性^[8],以达到改变执行流、泄露安全信息或提升权限等目的,图2为覆写决策数据改变程序分支的示意。针对非控制数据攻击的防护研究也已有数十载,相继提出了很多防护技术,如数据流完整性(DFI, data flow integrity)^[9]、数据空间随机化(DSR, data space randomization)^[10]等,但由于自身的局限性,目前此类防护技术还未能广泛地部署到商业应用。

Windows 作为全球占有量最大的桌面系统,其安全性直接关乎高达 90%^[11]个人计算机的安全。面对冯·诺依曼体系的天然缺陷,接二连三的内存攻击,微软不得不在 Windows 系统上相继部署层层防护,如数据执行保护(DEP, data execution prevention)^[12]、控制流防护(CFG, control flow guard)^[13]等,以加强系统的安全性。虽然这些内存防护机制加大了攻击的难度,但由于每种机制都受限于特定的条件,因此也给攻破层层防护留下了机会。从 Windows XP 到 Windows 10,随着系统的逐步加固,利用缓冲区溢出实施控制数据攻击的事件逐渐减少,与此同时,针对非控制数据的攻击却在增长,内存防护仍任重道远。图3展示了 Windows 已部署的内存防护机制及非控制数据防护技术的研究成果。接下来对 Windows 上的每种内存防护机制进行详细介绍,并指出其局限性和突破技术;随后介绍安全界针对非控制数据攻击提出的各种防护方案;最后讨论 Windows 内存防护存在的挑战,以及未来之路。

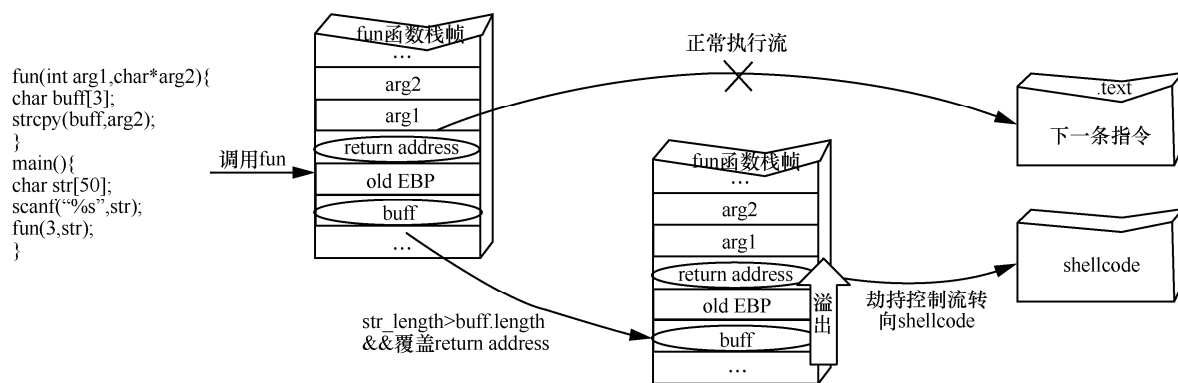


图1 覆盖返回地址劫持控制流

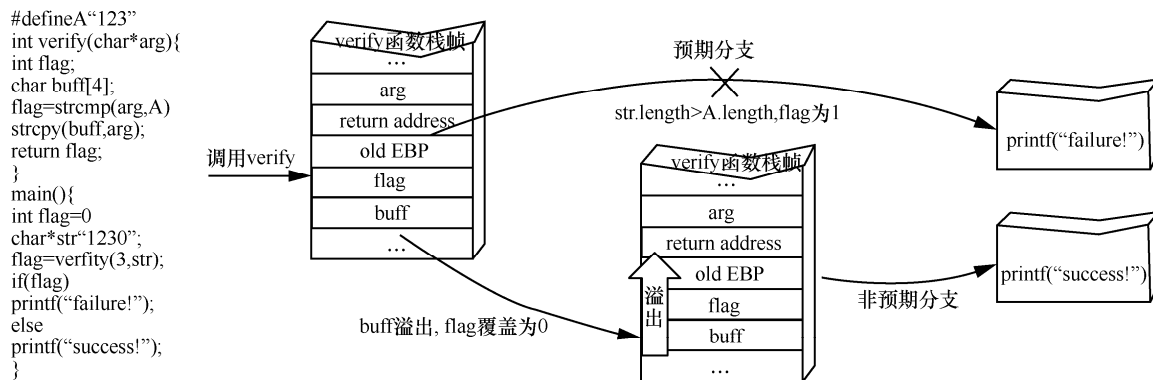


图2 覆写决策数据改变程序分支

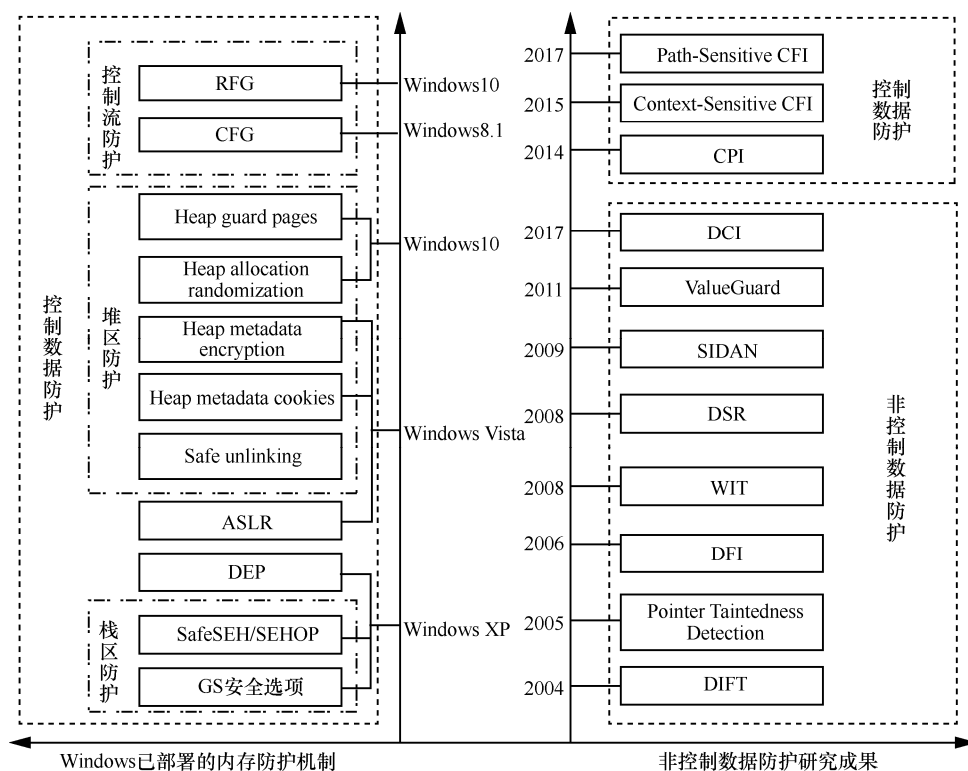


图 3 Windows 已部署的内存防护机制及非控制数据防护研究成果

2 Windows 已部署的内存防护机制

随着缓冲区溢出漏洞利用的泛滥，微软在每一次发布的 Windows 系统中都引入新的内存保护机制，这些机制成功阻扰了很多针对控制数据的攻击，使 Windows 平台下能利用的漏洞大大减少。然而每种机制都受限于特定条件，这给攻击者留下了突破的机会。

2.1 GS 安全编译选项

微软在吸收 StackGuard^[14]技术的思想上，独立开发并在 Windows XP SP2 中引入 GS^[15]编译技术，在 visual studio 2003 及以后版本中默认启用此编译选项。GS 编译选项为每个函数调用增加额外数据 Security cookie 和安全校验操作，以检测溢出攻击。在函数调用时，系统以 .data 节中的第一个双字作为 Security cookie 的种子，将其与函数栈帧栈顶指针 ESP 异或后的结果压入栈中，作为当前函数的 Security cookie；由于 Security cookie 位于函数局部变量和函数返回地址之间，当通过缓冲区溢出直接覆盖返回地址时，也会覆盖 Security cookie；在函数返回时，系统执行安全

校验操作，将栈帧中的 Security cookie 还原并与保存在 .data 节中的 Security cookie 种子进行一致性验证，若两者不一致，表明栈帧中的 Security cookie 遭到破坏，发生了溢出攻击，立即进入异常处理流程，反之函数正常返回。此过程如图 4 所示。

除加入 Security cookie 之外，微软在 visual studio 2005 及后续版本中使用了 Variables reordering^[16]技术，将字符串变量移动到栈帧高地址，同时在栈帧低地址存储指针类型和字符串类型参数的副本，避免溢出破坏局部变量和函数参数。此外，微软在 visual studio 2005 SP1 引入新的安全标识 #pragma strict_gs_check^[17]，实现对不符合 GS 保护条件的函数添加保护。

尽管 GS 能够有效遏止直接修改返回地址的溢出攻击，但其本身在设计上存在缺陷。一方面，GS 只在函数返回时进行一致性验证，在此之前没有任何的安全检查；另一方面，影响 GS 验证结果的 Security cookie 种子保存在 .data 节的固定位置；此外，若定义了安全处理函数，其指针也将存放在 .data 节。因此大部分突破 GS 防护的技术

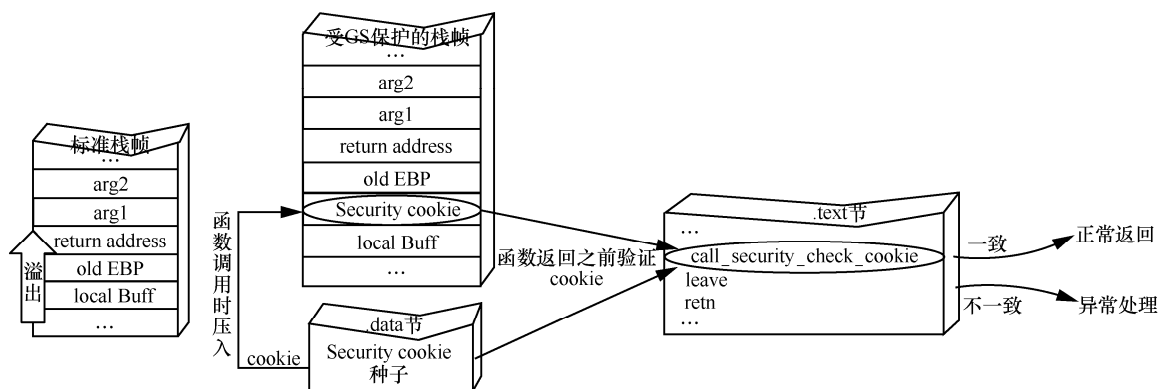


图 4 GS 防护机制原理示意

都利用这三大缺陷，设法在函数返回之前劫持控制流、泄露.data节中的种子或同时修改栈帧中的Security cookie和.data节中的种子，修改保存在.data节中的安全处理函数指针。Litchfield^[18]给出了多种突破GS防护机制的方案，如覆盖异常处理函数指针、修改cookie、重写安全处理函数指针等。

2.2 SafeSEH&SEHOP

结构化异常处理 (SEH, structured exception handling)^[19]是Windows采用的异常处理机制，其关键结构EXCEPTION_REGISTRATION包含2个重要的指针：SEH链表指针和异常处理函数指针，通过溢出覆盖异常处理函数指针是Windows平台下经典的漏洞利用方式，从前述可知攻击覆盖异常处理函数指针也是突破GS防护的重要手段。微软在Windows XP SP2及Windows Vista SP1中分别引入SafeSEH和结构化异常处理覆盖保护 (SEHOP, structured exception handling overwrite protection) 机制，以加强对SEH的保护。

SafeSEH在调用异常处理函数之前，对其进行有效性验证。启用/SafeSEH^[20]链接选项的编译器在编译程序时将所有的异常处理函数地址提取出来存放到一张SafeSEH表中；当调用异常处理函数时，将其地址与SafeSEH表中的表项进行匹配；若SafeSEH表中没有与其地址匹配的表项，表明其地址已被攻击，立即终止异常处理函数调用，反之继续调用。

SEHOP^[21]是对SafeSEH的进一步完善，其思想最初由Miller^[22]提出。程序一般有多个

EXCEPTION_REGISTRATION结构，这些结构组织成SEH链表存放在栈中，且此链末端保存系统默认的异常处理。在攻击异常处理函数指针时，也会同时覆盖指向下一个SEH结构的指针，这样SEH链就被破坏。基于此，SEHOP在进行SafeSEH校验之前，先检查SEH链的完整性。如果SEH链中的最后一个异常处理函数指针没有指向系统默认的异常处理函数ntdll!FinalExceptionHandler，则SEH链被破坏，立即终止异常处理函数调用，反之继续SafeSEH校验。整个SEH安全校验过程如图5所示。

SafeSEH和SEHOP对SEH的保护已经相当完善，能够抵御大部分的SEH攻击，但却不能从根本上消除此类攻击。SEHOP通过检查SEH链的完整性实施安全防护，事实上可以伪造SEH链^[23]，使其最后一个异常处理函数指针指向ntdll!FinalExceptionHandler，从而通过检查。对于SafeSEH，本身在实现时就有多个限制条件，Sotirov^[24]在2008年Black Hat大会上提出了多种绕过SafeSEH的技术。

2.3 DEP

微软在Windows XP SP2中引入DEP^[12]来弥补数据与代码混淆存储的缺陷，将数据所在页面标识为不可执行，当溢出成功转向shellcode时，会企图在数据页执行指令，此时抛出异常，从而遏止shellcode的执行。

硬件DEP需要CPU和操作系统的双重支持，2004年以后发行的CPU都支持DEP。要实现此功能，需在页表项中添加新的标志位NX/XD，以标识是否允许在页面上执行指令。当NX/XD设

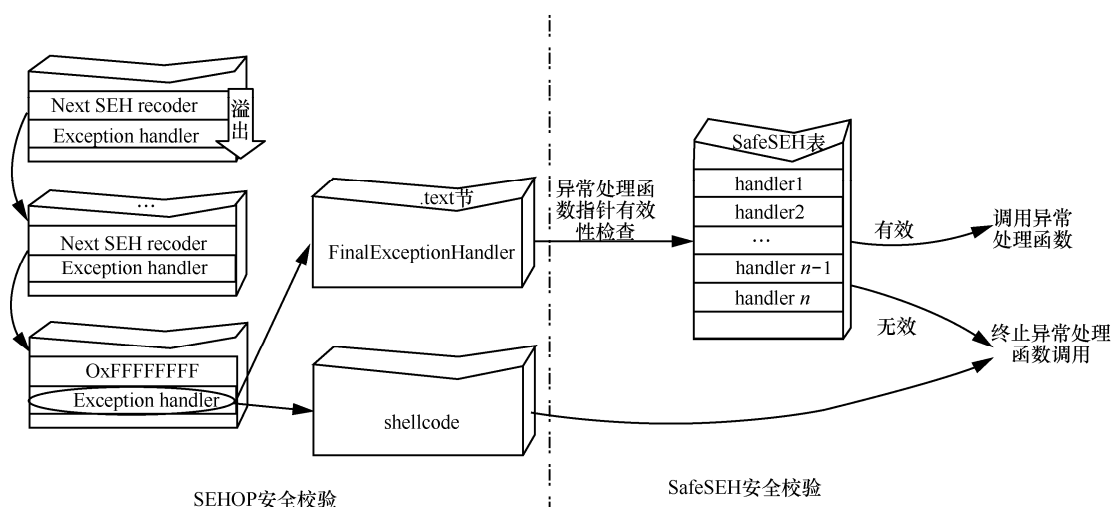


图 5 SEH 安全校验机制原理示意

置为 1 时,表示页面不允许执行指令;当为 0 时,表示页面允许执行指令。在 CPU 不支持硬件 NX/XD 时,软件 DEP 通过检查 SEH 异常处理函数指针,确保其位于可执行页的 .text 节。

出于兼容性考虑,Windows 并不能对所有进程都开启 DEP 保护,尤其是无法确定一些第三方插件 DLL 是否支持 DEP,对涉及这些 DLL 的程序一般都不会开启 DEP。DEP 将数据页标识为不可执行,以防止攻击,但对于可写可执行页,DEP 不起作用,典型的例子是 JVM 为 JAVA 对象或其他数据分配的是可写可执行页面,便可利用 heap spray 布局 shellcode 实施攻击。另一种突破思路是代码重用,在 .text 节中找到可重用的 gadget 组成具有一定攻击目的的 shellcode,面向返回的编程 (ROP, return-orient programming)^[25,26]是代码重用的成熟技术,ret2libc^[27]技术也可看作 ROP 的特例。

2.4 ASLR

前述的所有防护机制都是针对控制数据的攻击,这类攻击都需要一个明确的跳转地址,用来覆盖控制数据。地址空间布局随机化 (ASLR, address space layout randomization)^[28]是一种地址随机化技术,在加载程序时不再使用固定的基址加载,从而增大定位跳转地址的难度。

Whitehouse^[29]详细介绍了映像随机化、堆栈随机化、进程环境块 (PEB, process environment block) 与线程环境块 (TEB, thread environment block) 随机化。映像随机化在系统每次重启时实

现可执行文件 (PE, portable executable) 加载基址随机化,使通用跳板指令的地址不再固定。堆栈随机化在程序每次运行时随机地选择堆栈基值,增大攻击 SHE 结构、虚表指针等堆栈数据的难度。PEB 和 TEB 不再使用固定的加载基址,增加攻击其中的函数指针难度。

ASLR 给定位 shellcode 的跳转地址带来了巨大挑战,固定的跳转地址只在系统重启前甚至是程序的本次运行中才有效。但 ASLR 仍然留下了可利用的缺陷。映像随机化只对模块加载基址的前 2 个字节做了随机处理,后 2 个字节保持不变,加之并没有对指令序列进行随机化,故通用跳板、虚表指针等数据相对于加载基址的位置不变,如此便可利用“相对寻址”的概念动态地进行部分覆盖^[24,30]以定位跳转地址。堆栈基址随机化使程序每次运行时变量地址不同,但只能用来防止精准攻击,无法防御只需大概的地址范围就可实施攻击的技术,如 heap spray^[31]。PEB 和 TEB 即使完全随机化,也可以通过 FS 寄存器确定其地址,防护能力很有限。

2.5 CFG

微软在 Windows 8.1 update 3 和 Windows 10 中引入 CFG^[13],进一步加固了内存防护堡垒。CFG 是目前针对控制数据攻击的最强防护,主要通过检验间接跳转地址的有效性来防范攻击。

编译程序时启动/guard:cf^[32]选项,编译器会分析出程序中所有间接调用地址,将其保存在 Guard CF function table 中,并在所有函数调用之

前插入一段校验代码；当程序加载运行时，将进程空间中的每8个字节映射到 CFGBitmap 中的一位，如果8个字节中包含一个函数地址，对应位置为1，如此 Guard CF function table 中的每个地址都会转换成 CFGBitmap 中的对应比特位。间接调用之前，将先以目标地址为参数调用校验函数 `ntdll!LdrpValidateUserCallTarget`（由函数指针 `_guard_check_icall_fptr` 指向），访问 CFGBitmap，若以目标地址索引到的 CFGBitmap 位为1，表明目标是一个有效的函数地址，进行间接调用；若为0，表明目标无效，或许已被攻击，抛出异常，终止间接调用。此过程如图6所示。

尽管 CFG 是强有力的防护措施，但 CFG 并不是完美无缺的。Tang^[33]已详细分析了 CFG 机制的缺陷，如 CFGBitmap 空间基址固定且能获取、调用 `NtAllocVirtualMemory` 函数分配的可执行虚拟内存对应的 CFGBitmap 位全为1等。有缺陷就有突破的机会，可以通过获取 CFGBitmap 基址改写特定位状态，使间接目标地址合法；借助虚拟内存操作 `NtAllocVirtualMemory` 定制位图，绕过 CFG 检验。此外，CFG 只对函数指针进行有效性检查，并没有检查返回地址，因此另一种绕过 CFG 的方法是设法攻击返回地址。Theori^[34]在其博士论文中详细介绍了基于 Chakra JIT 绕过 CFG 机制的方法。2017年 Black Hat ASIA 大会上，Sun^[35]介绍了利用目标地址指针可写并且不受 CFG 保护的 memory-based 间接调用来绕过 CFG。

2.6 RFG

微软部署的 CFG 防护并没有执行完全的 CFI^[5]，CFG 只检查函数指针，而没有检查返回地址，也就无法防御针对返回地址^[3]的攻击或 ROP^[25,26]攻击。为了弥补这一缺陷，2016年微软

又引入了返回流防护(RFG, return flow guard)^[36]。

RFG 将返回地址保存到 `fs:[rsp]`，并在函数返回前将其与栈上返回地址进行比较，从而有效阻止了针对返回地址的攻击。开启 RFG 需要操作系统和编译器的双重支持，在编译阶段，编译器会以 `nop` 指令的形式在目标函数的首尾预留出指令空间。当目标可执行文件在支持并开启 RFG 的系统上运行时，预留的指令空间会在加载阶段被替换为 RFG 指令，首部替换的指令用来读取函数的返回地址，并将该地址存放到 `fs:[rsp]`，尾部替换的指令在函数返回时检查栈中的返回地址是否与 `fs:[rsp]` 中的一致，若一致，函数正常返回；若不一致，表明可能受到攻击，将跳转到 `_guard_ss_verify_failure`，引发 int29 异常，进程崩溃，从而阻止恶意攻击的进行。在不支持 RFG 的操作系统上运行时，这些 `nop` 指令不会影响程序的执行流程。RFG 防护原理如图7所示。RFG 将返回地址保存到攻击者不可控制的 Thread Control Stack，极大增加了后向转移攻击的难度与代价，其与 CFG 的结合可实现对控制流比较完善的防护。

2.7 堆区防护技术

除了以上防护机制外，微软专门针对堆区增加了很多安全校验机制。Sotirov^[24]在2008年 Black Hat 大会上详细介绍了 Safe unlinking、Heap metadata cookies 和 Heap metadata encryption 堆区防护技术。Safe unlinking 是在分配堆块时，先验证堆块前向指针 `fblink` 和后向指针 `blink` 的完整性，若完整性被破坏，将进行异常处理，反之才分配堆块，此过程如图8所示。Heap metadata cookies 类似于 GS 的 Security cookie，主要在堆块块首加入安全 cookie，以防止利用溢出覆盖块首中的 `fblink` 和 `blink`。Heap

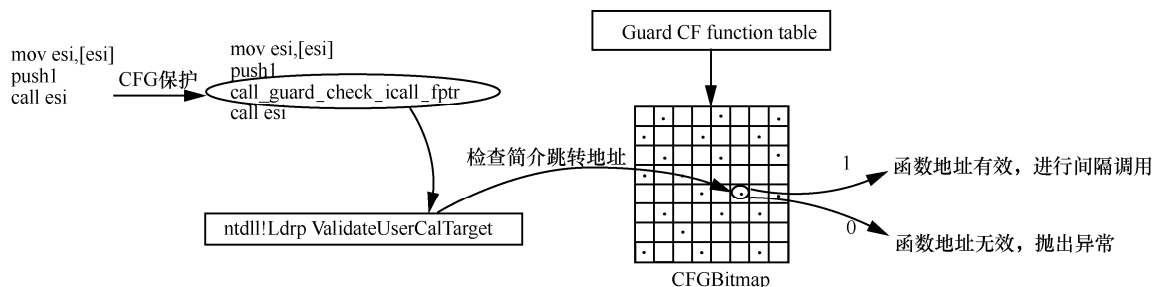


图6 CFG 防护原理示意

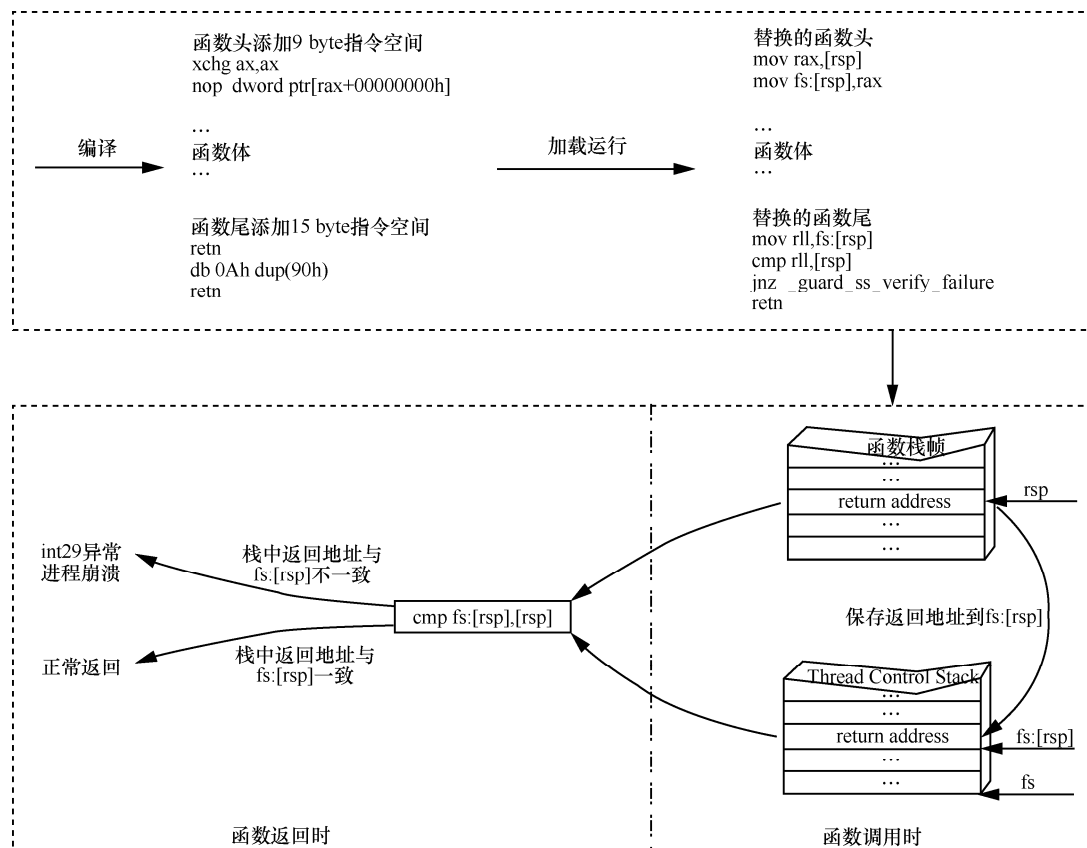


图 7 RFG 防护原理示意

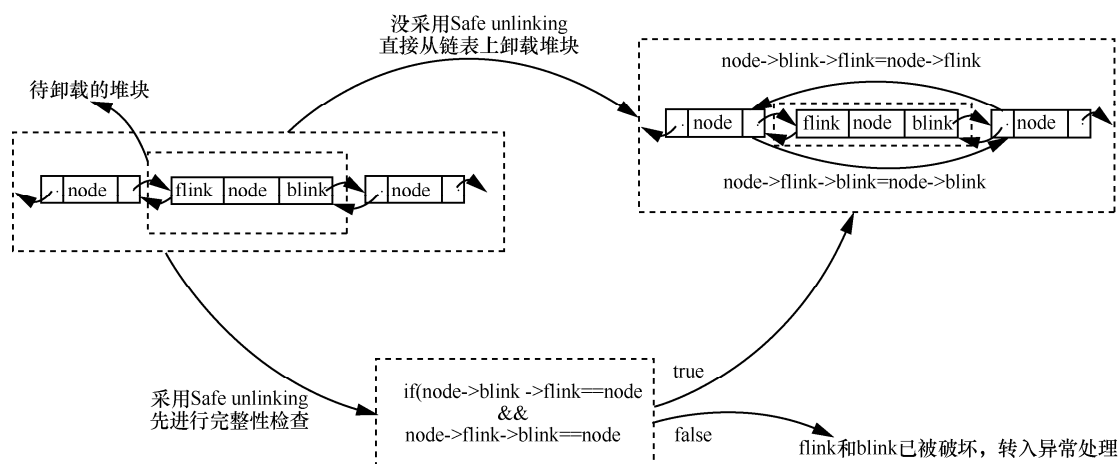


图 8 Safe unlinking 防护示意

metadata encryption 将堆块块首的重要数据与一个 4 byte 随机数异或后进行存储，以防止直接破坏这些数据。虽然这些技术可以减少堆区攻击，但不难看出，大多都是对堆块结构的防护，而没有对堆区存储的数据进行保护，如果攻击堆区存储的函数指针等关键数据，其后果不堪设想，因此这些机制的局限性很大。在

Windows 10 中，微软又加入了 Heap allocation randomization^[37]，每个新分配的堆块都会加上一个随机的偏移量，从而增大定位堆溢出目标的难度；此外，在堆块前后分别加入了 Heap guard pages^[37]，凡企图通过溢出攻击堆块，都会引起保护页的修改，而修改保护页的行为被视为内存破坏操作，此时系统就会立即终止程序。

3 非控制数据防护方案

上述都是微软采用的针对控制数据攻击的防护机制，目前微软还没有部署针对非控制数据攻击的系统级防护措施，但此类攻击却频频出现，而且危害极其严重。例如，攻击者利用 2014 年披露的 Heartbleed^[2]漏洞实施攻击，导致大量 Web 服务器上的私钥、cookie 及密码泄露，造成巨大的经济损失。由于非控制数据攻击完全遵循合法的控制流，所以现有的防护机制无法阻止非控制数据攻击；而且随着控制数据防护的逐步加固，可以预见，针对非控制数据的攻击将纷沓而至；但就在此类防护还没有大范围部署的情况下，研究人员却已提出多种利用方案，如 Data-Oriented Exploits^[38]和面向数据的编程（DOP, data-oriented programming）^[39]等，因此对于非控制数据的防护迫在眉睫。

自 Chen 等^[7]首次分析了非控制数据攻击的严重危害之后，安全界陆陆续续提出多种针对非控制数据攻击的防护技术，如 DFI^[9]、DSR^[10]、写完整性检查（WIT, write integrity testing）^[40]和数据机密性与完整性（DCI, data confidentiality and integrity）^[8]等，其中也不乏能同时防护控制数据和非控制数据的方案。

3.1 DIFT

Suh 等^[41]在 2004 年 ASPLOS 会议上提出动态信息流追踪（DIFT, dynamic information flow tracking），尽管 DIFT 最初实现并不是为了防护非控制数据攻击，但可以将其应用到非控制数据防护。DIFT 通过追踪并监控 I/O 数据的使用来实现

防护，主要有 3 部分工作：首先，操作系统来自 I/O 的潜在恶意数据标记为不可信数据；运行时，处理器根据每条指令的操作数和操作类型来传播不可信数据，即动态信息流追踪；当处理器检测到一个可疑的不可信数据操作，立即生成一个陷阱，并交由操作系统的陷阱处理器处理，若陷阱处理器检测到该操作违背了安全策略，将立即终止程序。防护原理如图 9 所示。

DIFT 防护对应用完全透明，不需要对程序进行任何修改，由操作系统和处理器实现，而且执行开销相当小，仅有 1.1%。但 DIFT 在实现之初存在一个严重的问题，采用宽松的不可信数据追踪规则，这导致引入了不必要的陷阱。

3.2 Pointer Taintedness Detection

Chen 等^[42]继首次分析非控制数据攻击的危害后，在 2005 年 DSN 会议上提出了 Pointer Taintedness Detection 技术。该技术主要基于指针污染的概念实现攻击检测与防护，一个污染指针是指其值直接或间接地来自用户输入的数据，对污染指针的非法解引用被视为一起恶意攻击。为实现污染，首先扩展了内存模型，为每个字节都增加了额外的污染位；当外部数据存入内存时，污染位被设置；处理器运行时，污染位将随着每条指令对污染数据的操作而传播；一旦包含污染位的数据被用作内存访问的地址，便认为发生了污染指针的非法解引用，处理器将抛出异常，操作系统处理异常并终止进程，从而遏止攻击。

Pointer Taintedness Detection 是一个处理器级的防护技术，对应用完全透明，因此不需要重编译/链接，并且能同时防护控制数据攻击和非控

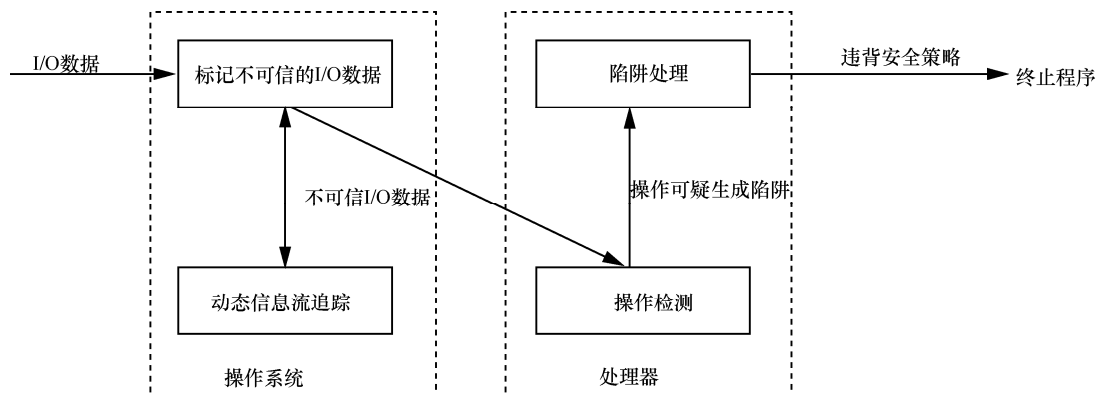


图9 DIFT 防护原理

制数据攻击。从实现原理看, Pointer Taintedness Detection 与 DIFT 都要对来自外部的数据进行标记, 只是指针污染引入污染位进行标记, 而 DIFT 是软件标记方式, 因此指针污染分析的执行开销更小, 但加入污染位增加了硬件开销。此外, Chen 等^[42]的实验结果表明, Pointer Taintedness Detection 尚没有发现已知的误报, 换言之, 误报率比较低, 逼近 0。尽管指针污染防护能检测大范围的内存破坏, 但并不能保证 100% 的覆盖, 因此仍存在漏报。

3.3 DFI

Castro 等^[9]在 2006 年的 OSDI 会议上提出了针对非控制数据攻击的防护技术数据流完整性 (DFI, data flow integrity)。DFI 通过检查变量操作是否符合数据流来防范针对非控制数据的攻击。首先, 通过静态的定义可达性分析获得程序的数据流图 (DFG, data flow graph), 并在每个变量写操作之前插入检测指令; 运行时, 对每个变量进行实际的定义可达性分析, 将其最后一条写指令的 identifier 存入运行时定义表 (RDT, runtime definitions table); 当读一个变量时, 先以变量地址索引 RDT 表中的 identifier, 再检查 identifier 是否存在于静态 DFG, 若不存在, 数据流完整性被破坏, 抛出异常。此过程如图 10 所示。DFI 通过流敏感的过程内分析和上下文敏感的过程间分析生成可达性定义, 流敏感的过程内分析计算在函数内声明的局部变量的可达性定义, 上下文敏感的过程间分析计算在函数外声明的变量的可达性定义。

DFI 是一个比较通用的防御技术, 可以同时防

护控制数据攻击和非控制数据攻击, 可自动应用到 C/C++ 程序而不需要任何的修改。但 DFI 的实现相对保守, 为变量的每个读操作都包含了所有的可达性定义, 这些可达性定义对某些读操作来说可能是多余的, 因此 DFI 不精确, 存在漏报; 此外 DFI 是一个全局防护技术, 对所有变量进行检查而没有考虑变量的安全敏感性, 因此执行开销大。

3.4 DSR

继 ASLR^[28]和指令集随机化 (ISR, instruction set randomization)^[43,44]之后, 2008 年 Bhatkar 等^[10]提出了另一种随机化技术数据空间随机化 (DSR, data space randomization)。DSR 对内存数据进行随机化处理, 同一等价类中的变量使用相同掩码进行加密存储, 不同等价类使用不同掩码, 读变量时再使用该掩码异或得到实际数据。DSR 的工作主要分为 3 步: 首先, 利用 Steensgaard^[45]指针分析法决定不同数据的掩码; 随后, 生成程序的指针指向图 (points-to graph), 用来计算等价类, 以便为其分配掩码; 最后对程序中的变量进行转换, 即用变量与其掩码的异或值替换变量本身。

相比于 ASLR 和 ISR 只能防御控制数据攻击, DSR 亦能防护非控制数据攻击。相对于 DFI 运行时平均代价 44%~103%^[9], DSR 性能开销更低, 平均只有 15%。鉴于 C 语言中指针的使用, 很难为每个变量分配唯一的掩码, 因此 DSR 使用了等价类的概念为指针解引用相同的变量分配同一个掩码, 计算等价类依赖于 points-to graph 生成算法, DSR 使用 Steensgaard 算法, 而 Steensgaard 算法将相同指针指向的变量合成一个节点, 因此算法的效率较高, 但同时导致分析不精确。

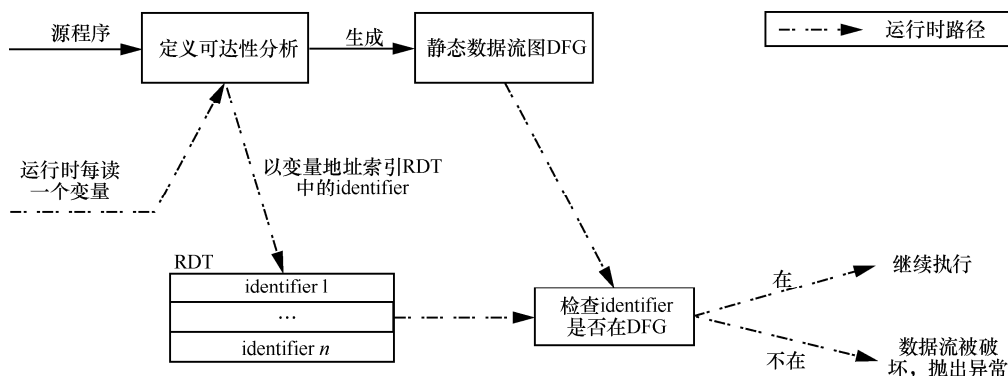


图 10 DFI 防护流程

3.5 WIT

Akritidis 等^[40]在 2008 年的 IEEE S&P 会议上提出了一种可以同时防护控制数据攻击和非控制数据攻击的技术写完整性检查 (WIT, write integrity testing)。在编译阶段, WIT 利用过程间的 points-to 分析计算程序的控制流图和程序中可写的对象集; 运行时, WIT 执行写完整性检查, 阻止指令对非静态分析的可写对象集之外的其他对象进行写操作, 并执行控制流完整性, 确保控制转移遵循静态控制流图。具体来说, WIT 通过 points-to 分析为可写的对象和其相应的写指令分配相同的颜色, 并将其颜色保存到颜色表中, 执行写完整性检查时, 再在颜色表中查找写对象的颜色并将其与写指令的颜色进行比较, 不一致表明违背了写完整性, 将抛出异常; 对于控制流防护也一样, 为间接调用指令和其对应的目标地址分配相同的颜色值, 执行间接调用检查时, 比较目标地址与间接调用指令的颜色是否相同, 不同表明发生了控制流劫持, 抛出异常。此过程如图 11 所示。

WIT 利用静态分析和动态插桩检测确保写完整性和控制流完整性, 有力防护针对控制数据和非控制数据的攻击; 通过静态分析决定安全对象 (不会违背写完整性的对象), 只对不安全的可写对象执行写完整性, 减少不必要的安全检查, 降

低了执行开销 (平均 7%); 同时 WIT 编译 C/C++ 程序时不需要对其进行任何修改。但 WIT 并不是很完美, 静态分析时利用 points-to 计算不安全对象, 只对其进行完整性检查, 虽然降低了执行开销, 但限于 points-to 分析的精度, 无法保证 100% 的覆盖, 可能会存在漏报的情况; 此外, WIT 只对写操作进行了完整性检查, 并没有对读操作进行安全检查, 因此仍然无法阻止信息泄露; 另外在实现 WIT 时需要维护颜色表, 引入了额外的空间开销。

3.6 SIDAN

2009 年, Demay 等^[46]构建了一个面向非控制数据攻击的检测模型 SIDAN (software instrumentation for detecting attacks on non-control-data)。SIDAN 主要通过验证系统调用所依赖的数据项 (系统调用参数及影响系统调用执行的条件变量) 状态的一致性来检测内存错误, 从而检测非控制数据攻击。首先进行静态分析, 利用程序依赖图 (PDG, program dependency graph) 获取每个系统调用所依赖的变量集, 导出这些变量之间的约束, 从而建立起一个数据行为模型, 同时在系统调用之前加入合理性检查; 运行时, 检测系统调用所依赖的变量是否符合静态分析的约束, 如不符合, 说明发生了非法系统调用。

鉴于在系统调用点并不是所有变量都可见, SIDAN 并不能对所有的系统调用都进行检查。检

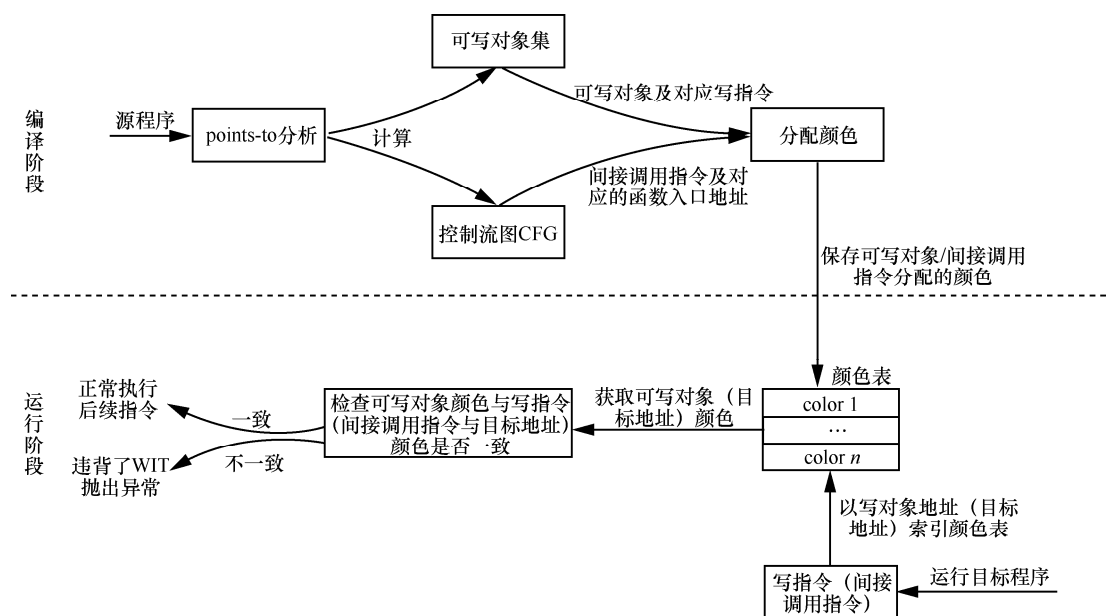


图 11 WIT 防护过程

查时在每个函数调用中插入断言来检测执行路径上可见并影响系统调用的变量,所以 SIDAN 的检测精度依赖于生成的断言数量和质量,但无法保证所有的断言都被发现,因此 SIDAN 存在漏报。但如果违背了变量约束,说明发生了非控制数据攻击,因此没有误报。

3.7 ValueGuard

ValueGuard^[47]的原理十分简单,主要借鉴 StackGuard^[14]思想,只是将其防护对象从返回地址扩展到了所有变量,所以可防御非控制数据攻击。ValueGuard 在每个变量前加入 canary,并将 canary 及变量封装成一个结构体,在使用变量时,先检查 canary 的完整性,若被破坏,表明发生了溢出攻击,立即终止程序。ValueGuard 对所有变量进行保护,代价相当大;而且由其实现原理可知,ValueGuard 只能检测通过溢出覆盖非控制数据的攻击;而且实现 ValueGuard 需要进行重编译,因此 ValueGuard 并不适用于商业应用。

3.8 DCI

前述的非控制数据防护技术大部分都是对程序的所有数据进行检查,而这些数据中有大量非安全敏感数据,所以引入了不必要的开销,致使其无法应用于商业软件。为了应对这一情况,Carr 等^[8]在 2017 年 ASIA CCS 会议上提出了数据机密性与完整性(DCI, data confidentiality and integrity)^[8]。DCI 按照类型将数据分为安全敏感数据和不敏感数据,只对敏感数据进行精确的空间和时间安全

检测;对于不敏感数据,DCI 只进行粗粒度的边界检查,从而降低执行开销。空间安全检测通常涉及越界的指针解引用,时间安全检测通常涉及未初始化或已释放的指针解引用。DCI 在 C/C++ 语言中引入注解,允许程序员定义保护类型(安全敏感数据类型),运行时,DCI 禁止对此类型数据的非法读或写,从而保证了数据的机密性和完整性。具体流程如下。首先程序员向源代码中加入注解定义程序中的敏感数据类型。编译时,识别敏感变量及相关数据流,加入安全检测程序。运行时,遵循随后策略进行安全检查。1) 对于敏感数据:检查指向敏感数据的指针解引用是否在相关的内存对象边界内或者是否是有效的内存对象,若不在边界内或不是有效的对象,表明发生了越界读或非法写,破坏了数据机密性或完整性,立即终止程序;2) 对于非敏感数据:检查其指针解引用是否在敏感数据内存区,若是,终止程序;3) 敏感数据和非敏感数据之间禁止任何的数据流。此过程如图 12 所示。

DCI 将数据分为安全敏感数据和非敏感数据,对非敏感数据进行粗粒度的安全检测,从而降低了开销,同时对敏感数据进行精确的安全检查,实现对非控制数据的防护,兼顾了性能与效能。但 DCI 需要程序员定义敏感数据类型,如此 DCI 的防护效能就依赖于程序员的专业素质;同时由于人工干预,DCI 无法实现自动分析敏感数据类型。另外,由于人工定义敏感数据类型难免有疏漏,若将敏感数据误判为

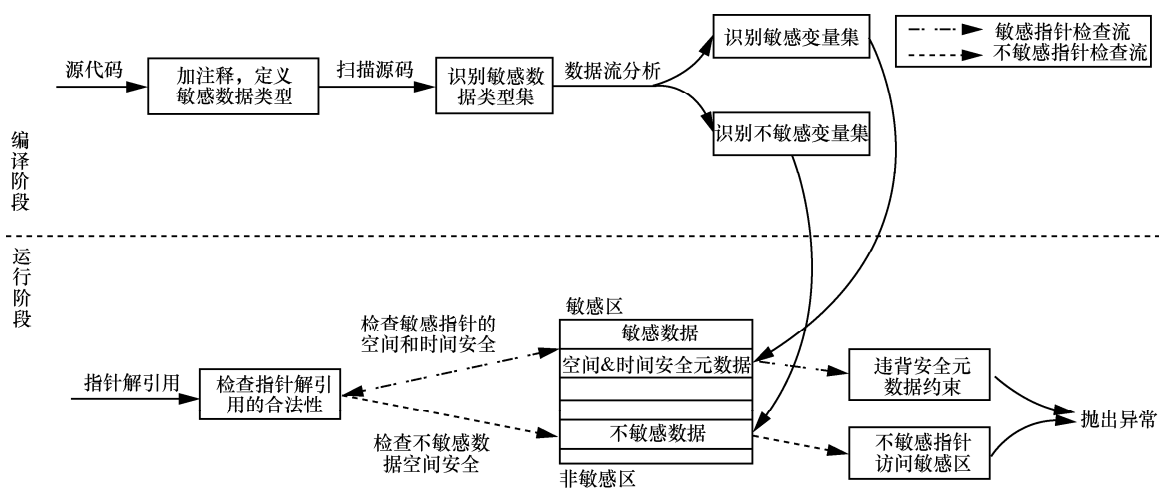


图 12 DCI 防护流程

非敏感数据，而 DCI 又对非敏感数据只进行粗粒度安全检测，最终还是可以通过修改非敏感数据实施攻击。

无论是 Windows 已部署的针对控制数据的防护机制还是未部署的非控制数据防护技术，在实现上或多或少都存在问题，但就效能来说，都起到很好的防护作用，表 1 对各类防护方案的优缺点进行了分析。

4 结束语

随着 Windows 每个版本的发布，都会引进大量的内存防护机制，系统的安全性在不断加强。虽然微软已为堆栈等内存区加注了层层防护，增大了攻击此区域内控制数据的难度，也成功遏止了大部分的控制流劫持攻击。但每种防护机制，鉴于实现时的限制条件，多多少少都存在可以绕

表 1 各类防护方案分析

	防护方案	防护对象	优点	缺点
非控制数据防护方案	DIFT	决策数据&控制数据	对应用程序完全透明 执行开销小仅有 1.1% 无误报	采用宽松的不可信数据追踪规则，导致引入不必要的陷阱
	Pointer Taintedness Detection	指针解引用	对应用程序完全透明 执行开销小 误报率低逼近于 0	存在漏报 引入硬件开销
	DFI	程序的所有变量	不需要修改源码 无误报	执行开销大 存在漏报
	DSR	内存数据	较 DFI 执行开销小，平均 15%	需修改源码 等价类掩码计算不精确
	WIT	可写对象&控制数据	不需要修改源码 较 DFI、DSR 开销小，平均 7% 无误报	存在漏报 引入空间开销 未检查读操作，无法阻止越界读
	SIDAN	系统调用参数及影响其调用的变量	执行开销小 无误报	需修改源码 存在漏报 只能检测非法系统调用
	ValueGuard	程序的所有变量	无误报	需修改源码 全局防护执行开销很大 只能检测堆栈攻击
	DCI	安全敏感数据	较 DFI、DSR 大范围防护，DCI 执行开销小，最大 16.34% 较 WIT 只能检查读操作，DCI 可同时检查读操作和写操作	需修改源代码 需人工干预 存在漏报
Windows 已部署的内存防护机制	GS	返回地址	遏止直接修改返回地址	Security cookie 易泄露
	SafeSEH	异常处理函数指针	保证了异常处理函数数据的有效性	对指向堆区的异常处理函数指针无效
	SEHOP	SEH 链	在 SafeSEH 前检查 SEH 链的完整性	可通过伪造 SEH 链绕过检查
	DEP	数据页	阻止执行部署到数据页的 shellcode	多数第三方应用的 DLL 不支持 对可写可执行页不起作用
	ASLR	控制数据地址	加大定位通用跳板等跳转地址的难度	映像随机化只对映射基值的前 2 个字节进行了随机化 堆栈随机化只能防止精准攻击
	CFG	间接跳转地址	可防范对间接跳转目标地址的攻击	位图映射基址固定 调用 NtAllocVirtualMemory 分配的可执行虚拟内存对应位图位全为 1 进程加载不支持 CFG 的模块，将不受 CFG 保护
	RFG	返回地址	返回地址保存在攻击者不可控的 Thread Control Stack	只能防护针对返回地址的攻击，验证失败后执行的处理可以通过其他方式修改
	Safe unlinking	堆块块首	在分配堆块之前验证 flink 和 blink 的有效性	只考虑了堆块分配时指针的完整性，没有处理堆块回收时的情况
	Heap metadata cookies	堆块块首	可防止直接溢出覆盖块首中的 flink 和 blink	和 GS 一样都存在 cookie 泄露的风险
	Heap metadata encryption	堆块块首	对堆块块首数据加密存储，可防止对其进行直接修改	只对块首元素进行加密存储，并没有加密存储堆块数据
	Heap guard pages	堆块	堆块首尾加入防护页，有效防止跨堆块攻击块首	对同堆块的堆数据还不能起到保护作用，仍可覆盖同一堆块中的函数指针实施攻击

过的缺陷,甚至激发出成熟的突破技术。对于 GS^[15]防护,关键的校验数据 Security cookie 可以通过各种方式被泄露;伪造 SEH 链亦能通过 SEH 安全校验机制^[23];而 DEP^[12]和 ASLR^[28]又分别促进 ROP^[25,26]和 heap spray^[31]技术的发展,而且由于兼容性问题,很多第三方应用的 DLL 并不支持 DEP,ASLR 也只对加载基址进行随机化,并没有实施 ISR^[43,44]对指令序列进行随机化,仍无法阻止部分覆盖攻击^[24];CFG^[13]和 RFG^[36]尽管对控制流进行了强有力的防护,但并没有实现完全的 CFI^[5],只是通过对目标程序进行静态分析获得所有间接跳转地址,在运行时将间接跳转地址与静态分析的结果进行简单的匹配来检查间接跳转地址的有效性,并没有实施 Path-sensitive CFI^[48]、Context-sensitive CFI^[49]和 CPI^[6],仍无法完全遏止控制流劫持攻击(如 call-oriented programming^[50]),因此对已部署的控制数据防护机制仍需进一步完善。

此外,微软目前还没有在 Windows 上部署针对非控制数据攻击的防护措施,而此类攻击却正在泛滥,尽管现阶段安全界提出的非控制防护技术不乏可以有效防御此类攻击的方案,但大多都因其限制条件无法应用于商业软件,因此 Windows 的内存防护任务仍很艰巨。DFI^[9]、DSR^[10]尽管可以检测大范围的非控制数据破坏,同时也能检测控制数据攻击,但因全局防护带来了很大的执行开销;DIFT^[41]和 Pointer Taintedness Detection^[42]虽执行开销小,也能防护大部分的攻击,但由于污染数据的标记算法不精确,会造成很多漏报,而且增加污染位也引入了空间开销;SIDAN^[46]和 ValueGuard^[47]需要对源码进行修改,而且防护范围很有限,SIDAN 只对系统调用进行检测,ValueGuard 只能阻止堆栈攻击;WIT^[40]算是一个比较高效的防护方案,比 DFI 和 DSR 的执行开销都小,并且能同时防护控制数据攻击和非控制数据攻击,但 WIT 只对写操作进行了完整性检查,并没有对读操作进行检查,因此仍无法阻止越界读而引起的信息泄露;DCI^[8]对非敏感数据进行粗粒度的安全检测,对敏感数据进行精确的安全检查,较 DFI、DSR 的大范围防护降低了执行开销,较 WIT 只对写操作进行检查,DCI 可同

时对写操作与读操作进行检查,但 DCI 需要程序员定义敏感数据类型,存在漏报情况,而且由于 DCI 对非敏感数据只进行粗粒度安全检测,仍无法阻止通过修改非敏感数据实施攻击。因此仍需要研究兼顾防护效能与代价的非控制数据防护方案。

面对如此严峻的内存防护任务,微软应从多个方面入手,加强 Windows 的安全性,将针对内存攻击的危害最小化。对于已部署的控制数据防护方案应进一步完善,如可考虑利用路径敏感分析和上下文敏感分析增强 CFG 的防护效能、实施 ISR 阻挠针对 ASLR 的部分覆盖攻击等;其次,将现有的多种防护机制配合使用,以发挥最大的防护效力;另外,出于兼容性考虑,现有的防护机制并不全都默认开启,因此,微软应加强与第三方软件合作,引导其用安全的代码保护功能开发第三方软件,以解决兼容性问题,最大化安全机制的防护范围。而随着控制数据攻击代价的与日俱增,攻击者必会将视角转移到非控制数据攻击,未来针对非控制数据的攻击将会泛滥,部署系统级非控制数据防护机制刻不容缓;同时由于现存的非控制数据防护方案很难取得效能与代价的平衡,因此仍需研究兼顾两者且适应商用的方案。

参考文献:

- [1] Morris worm[EB/OL]. https://en.wikipedia.org/wiki/Morris_worm.
- [2] DURUMERIC Z, LI F, KASTEN J, et al. The matter of heartbleed[C]//2014 Conference on Internet Measurement. 2014: 475-488.
- [3] ZATKO P. How to write buffer overflows[EB/OL]. http://insecure.org/stf/mudge_buffer_overflow_tutorial.html.
- [4] HOGLUND G. Advanced buffer overflow technique[EB/OL]. <http://www.blackhat.com/presentations/bh-asia-00/greg/greg-asia-00-stalking.ppt>.
- [5] ABADI M, BUDIU M, ERLINGSSON U, et al. Control-flow integrity[C]//The 12th ACM Conference on Computer and Communications Security. 2005: 340-353.
- [6] KUZNETSOV V, SZEKERES L, PAYER M, et al. Code-pointer integrity[C]//The 11th USENIX Symposium on Operating Systems Design and Implementation. 2014: 147-163.
- [7] CHEN S, XU J, SEZER E C, et al. Non-control-data attacks are realistic threats[C]//The 14th USENIX Security Symposium. 2005:

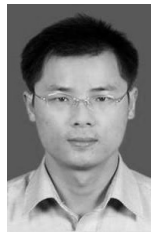
- 177-191.
- [8] CARR S A, PAYER M. DataShield: configurable data confidentiality and integrity[C]//2017 ACM on Asia Conference on Computer and Communications Security. 2017:193-204.
- [9] CASTRO M, COSTA M, HARRIS T. Securing software by enforcing data-flow integrity[C]//The 7th USENIX Symposium on Operating Systems Design and Implementation. 2006:147-160.
- [10] BHATKAR S, SEKAR R. Data space randomization[C]//The 5th International Conference on Detection of Intrusion and Malware, and Vulnerability Assessment. 2008:1-22.
- [11] Usage share of operating systems[EB/OL]. https://en.wikipedia.org/wiki/Usage_share_of_operating_systems#Desktop_and_laptop_computers.
- [12] ANDERSEN S. Memory protection technologies[EB/OL]. [https://technet.microsoft.com/en-us/library/bb457155\(d=printer\).aspx](https://technet.microsoft.com/en-us/library/bb457155(d=printer).aspx).
- [13] Control flow guard[EB/OL]. [https://msdn.microsoft.com/library/Windows/desktop/mt637065\(v=vs.85\).aspx](https://msdn.microsoft.com/library/Windows/desktop/mt637065(v=vs.85).aspx).
- [14] COWAN C, PU C, MAIER D, et al. StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks[C]//The 7th USENIX Security Symposium. 1998:63-77.
- [15] GS compiler option documentation for Visual Studio 2003[EB/OL]. [https://msdn.microsoft.com/en-us/library/8dbf701c\(v=vs.71\).aspx](https://msdn.microsoft.com/en-us/library/8dbf701c(v=vs.71).aspx).
- [16] RICHARTE G. Four different tricks to bypass StackShield and StackGuard protection[EB/OL]. <https://www.cs.purdue.edu/homes/xyzhang/fall07/Papers/defeat-stackguard.pdf>.
- [17] strict_gs_check pragma documentation for Visual Studio 2005[EB/OL]. [https://msdn.microsoft.com/en-us/library/bb507721\(v=vs.80\).aspx](https://msdn.microsoft.com/en-us/library/bb507721(v=vs.80).aspx).
- [18] LITCHFIELD D. Defeating the stack based buffer overflow prevention mechanism of Microsoft Windows 2003 server[EB/OL]. <http://www.blackhat.com/presentations/bh-federal-03/bh-fed-03-lite-hfield.pdf>.
- [19] PIETREK M. A crash course on the depths of Win32 structured exception handling[EB/OL]. <https://www.microsoft.com/msj/0197/exception/exception.aspx>.
- [20] SAFESEH linker option documentation for Visual Studio 2003[EB/OL]. [https://msdn.microsoft.com/en-us/library/9a89h429\(v=vs.71\).aspx](https://msdn.microsoft.com/en-us/library/9a89h429(v=vs.71).aspx).
- [21] Preventing the exploitation of structured exception handler (SEH) overwrites with SEHOP[EB/OL]. <https://blogs.technet.microsoft.com/srd/2009/02/02/preventing-the-exploitation-of-structured-exception-handler-seh-overwrites-with-schop/>.
- [22] Miller M. Preventing the exploitation of SEH overwrites[EB/OL]. <http://www.uninformed.org/?v=5&a=2&t=pdf>.
- [23] BERRE S L, CAUQUIL D. Bypassing SEHOP[EB/OL]. <https://repo.zenk-security.com/Reversing%20.%20cracking/Bypassing%20SEHOP.pdf>.
- [24] SOTIROV A, DOWD M. Bypassing browser memory protections[EB/OL]. http://www.blackhat.com/presentations/bh-usa-08/Sotirov_Dowd/bh08-sotirov-dowd.pdf.
- [25] SHACHAM H. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86)[C]//The 14th ACM CCS. 2007:552-561.
- [26] BUCHANAN E, ROEMER R, SAVAGE S, et al. Return-oriented programming: exploitation without code Injection[EB/OL]. http://www.blackhat.com/presentations/bh-usa-08/Shacham/BH_US_08_Shacham_Return_Oriented_Programming.pdf.
- [27] Bypassing Windows hardware-enforced data execution prevention[EB/OL]. <http://www.uninformed.org/?v=2&a=4&t=pdf>.
- [28] PaX ASLR[EB/OL]. <http://pax.grsecurity.net/docs/aslr.txt>.
- [29] WHITEHOUSE O. An analysis of address space layout randomization on Windows VistaTM[EB/OL]. http://www.symantec.com/av-center/reference/Address_Space_Layout_Randomization.pdf.
- [30] SOTIROV A. Windows ANI header buffer overflow[EB/OL]. <http://www.phreedom.org/research/vulnerabilities/ani-header/>.
- [31] Heap spraying[EB/OL]. https://en.wikipedia.org/wiki/Heap_spraying#cite_note-8.
- [32] Guard compiler option documentation for Visual Studio 2015[EB/OL]. [https://msdn.microsoft.com/en-us/library/8dbf701c\(v=vs.71\).aspx](https://msdn.microsoft.com/en-us/library/8dbf701c(v=vs.71).aspx).
- [33] TANG J. Exploring control flow guard in Windows 10[EB/OL]. <http://sjc1-te-ftp.trendmicro.com/assets/wp/exploring-control-flow-guard-in-Windows10.pdf>.
- [34] THEORI. Chakra jit cfg bypass[EB/OL]. <http://theori.io/research/chakra-jit-cfg-bypass>.
- [35] SUN K, OU Y, ZHAO Y H, et al. Never let your guard down: finding unguarded gates to bypass control flow guard with big data[EB/OL]. <https://www.blackhat.com/docs/asia-17/materials/asia-17-Sun-Never-Let-Your-Guard-Down-Finding-Unguarded-Gates-To-Bypass-Control-Flow-Guard-With-Big-Data.pdf>.
- [36] Return flow guard[EB/OL]. <http://xlab.tencent.com/en/2016/11/02/return-flow-guard/>.
- [37] Mitigate threats by using Windows 10 security features[EB/OL]. <https://docs.microsoft.com/en-us/Windows/threat-protection/overview-of-threat-mitigations-in-Windows-10>.
- [38] HU H, CHUA Z L, ADRIAN S, et al. Automatic generation of data-oriented exploits[C]//The 24th USENIX Security Symposium. 2015: 177-192.
- [39] HU H, SHINDE S, ADRIAN S, et al. Data-oriented programming: on the expressiveness of non-control data attacks[C]//2016 IEEE Symposium on Security and Privacy. 2016: 969-986.
- [40] AKRITIDIS P, CADAR C, RAICIU C, et al. Preventing memory error exploits with WIT[C]//2008 IEEE Symposium on Security and privacy. 2008:263-277.
- [41] SUH G E, LEE J W, ZHANG D, et al. Secure program execution via dynamic information flow tracking[C]//The 11th International

- Conference on Architectural Support for Programming Languages and Operating Systems. 2004:85-96.
- [42] CHEN S, XU J, NAKKA N, et al. Defeating memory corruption attacks via pointer taintedness detection[C]//2005 International Conference on Dependable Systems and Networks. 2005:378-387.
- [43] BARRANTES E G, ACKLEY D, PALMER T S, et al. Randomized instruction set emulation to disrupt binary code injection attacks[C]//The 10th ACM Conference on Computer and Communications Security. 2003:281-289.
- [44] KC G S, KEROMYTIS A D, PREVELAKIS V. Countering code injection attacks with instruction-set randomization[C]//The 10th ACM Conference on Computer and Communications Security. 2003:272-280.
- [45] STEENSGAARD B. Points-to analysis in almost linear time[C]//The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. 1996:32-41.
- [46] DEMAY J C, TOTEL E, TRONEL R. SIDAN: a tool dedicated to software instrumentation for detecting attacks on non-control-data[C]//2009 Fourth International Conference on Risks and Security of Internet and Systems. 2009:51-58.
- [47] ACKER S V, NIKIFORAKIS N, PHILIPPAERTS P, et al. Value-guard: protection of native applications against data-only buffer overflows[C]//The 6th International Conference on Information Systems Security. 2010:156-170.
- [48] DING R, QIAN C X, SONG C Y, et al. Efficient protection of path-sensitive control security[C]//The 26th USENIX Security Symposium. 2017:131-148.
- [49] VEEN V V D, ANDRIESSE D, GÖKTAS E, et al. Practical context-sensitive CFI[C]//The 22nd ACM SIGSAC Conference on Computer and Communications Security. 2015: 927-940.
- [50] GÖKTAS E, ATHANASOPOULOS E, BOS H, et al. Out of control: overcoming control-flow integrity[C]//2014 IEEE Symposium on Security and Privacy. 2014: 575-589.

作者简介:



鲁婷婷(1994-),女,甘肃正宁人,四川大学博士生,主要研究方向为信息安全。



王俊峰(1976-),男,安徽芜湖人,博士,四川大学教授、博士生导师,主要研究方向为空间信息网络、网络与信息安全、大数据处理在医学和交通等领域应用。