doi 10.3969/j.issn.1671-1122.2014.12.016

基于内存的漏洞缓解关键技术研究

贺滢睿[1]，史记[1]，张涛[2]，文伟平[2]
（1. 中国公安大学网络安全保卫学院，北京 100037 ；2. 北京大学软件与微电子学院，北京 102600）

摘 要 ：随着漏洞挖掘技术日渐成熟，每年新增漏洞数量逐步增加。从操作系统以及编译器层面来说，为了提高内存保护的安全性，对抗漏洞利用的缓解措施也在不断完善。文章介绍了近年来比较成熟的基于内存的漏洞关键缓解技术，包括 GS 编译选项技术、SEH 安全校验机制、堆数据保护机制、DEP 技术以及 ASLR 技术。GS 编译选项技术和 SEH 安全校验机制能够有效遏制针对栈数据的攻击 ；堆数据保护机制为堆溢出增加了更多限制 ；DEP 技术能够对内存执行额外检查以防止恶意代码在系统中执行 ；ASLR 技术通过对关键地址的随机化使一些堆栈溢出手段失效。文章还指出了这些防护措施所存在的不足，并据此从攻击者的角度介绍了针对这几种缓解措施的攻击思路。针对漏洞缓解技术，文章指出未来必须考虑的是如何弥补在抵御复合向量攻击方面的不足，如何完善旁路保护。

关键词 ：内存安全 ；漏洞缓解 ；绕过
中图分类号 ：TP309 文献标识码 ：A 文章编号 ：1671-1122（2014）12-0076-07

中文引用格式 ：贺滢睿，史记，张涛，等 . 基于内存的漏洞缓解关键技术研究 [J]. 信息网络安全，2014，(12)：76-82.

英文引用格式 ：HE Y R, SHI J, ZHANG T, et al. The Research on Vulnerability Mitigation in Memory [J]. Netinfo Security,2014,(12):76-82.

The Research on Vulnerability Mitigation in Memory

HE Ying-rui[1], SHI Ji[1], ZHANG Tao[2], WEN Wei-ping[2]
(1. *School of Network Security, People's Public Security of China,Beijing* 100037,*China*;2. *School of Software & Microelectronics, Peking University, Beijing* 102600,*China*)

Abstract: With the technology of finding vulnerabilities in software getting more mature, the total number of bugs is increasing yearly. In order to improve the security of memory protection, in terms of operating system and compiler, measures taken by OS to mitigate exploit are getting more perfect. This article describes some of the key mitigations, including GS options, SEH, Heap protection, DEP, and ASLR. The GS compiler technology and SEH security authentication mechanism can effectively detect and prevent most stack-based overflow attacks; Heap protection provides more restrictions aiming at stack overflow; DEP can perform additional memory checks to prevent malicious code executing in the system; ASLR helps to prevent buffer overflow attacks by randomizing the key address.The article also points out the drawbacks and introduces some method to defeat these mitigations from the views of attackers. Aiming at the vulnerability mitigation technology, the article points out it must be considered how to cover the shortage on resisting the attack of composite vectors and how to improve and perfect the bypassing protection in the future.

Key words: memory security; vulnerability mitigation; bypassing

收稿日期 ：2014-11-15
基金项目 ：国家自然科学基金 [61170282]
作者简介 ：贺滢睿（1991-），女，新疆，硕士研究生，主要研究方向 ：信息对抗、网络安全与防范 ；史记（1990-），男，北京，硕士研究生，主要研究方向 ：漏洞挖掘与利用、软件安全漏洞分析 ；张涛（1987-），男，江西，硕士，主要研究方向 ：系统与网络安全、软件安全漏洞分析 ；文伟平（1976-），男，湖南，副教授，博士，主要研究方向 ：网络攻击与防范、恶意代码研究、信息系统逆向工程和可信计算技术等。
通讯作者 ：贺滢睿 heyingrui415@163.com

0 引言

电子计算机的实现参照了图灵机模型，但是却忽略了图灵机模型中对程序和数据的明确区分，将指令和数据混淆在一起放到计算机内存中。这使得现代计算机有了先天缺陷，漏洞产生的根本原因也正是由于这一缺陷[1]。例如，软件加壳和脱壳技术、高级的变形病毒等是由于存在当程序运行时，将程序指令当成一般内存数据进行动态读写的缺陷；堆栈溢出攻击中 shellcode 的执行是由于存在计算机错误地把存放在堆栈中的普通内存数据当做程序指令来执行的缺陷；此外，跨脚本攻击、SQL 注入攻击等都是基于计算机没有明确区分数据和代码这一先天缺陷。

为了提高内存保护的安全性，操作系统以及编译器层面所提供的漏洞利用的缓解措施包括：1）GS 编译选项技术。GS 编译选项技术是检测某些覆盖函数返回地址、异常处理程序地址或特定类型参数的一种技术[2]。经过 GS 编译技术编译的程序会在函数的返回地址前加入 security cookie，在函数返回前先对 security cookie 进行检测，判断其是否被覆盖，如果被覆盖则程序不会继续执行。这项技术使得栈溢出变得相当困难。文献 [3,4] 总结了缓冲区溢出原理及其特征和攻击方法。2）SEH（structured exception handling）安全校验机制。SEH 安全校验机制提供程序设计人员一种能够对错误进行处理的机制，能够有效遏制采用改写 SEH 手段的劫持进程攻击。SEHOP 是 SEH 安全校验机制的补充，是一种更为严格的 SEH 保护措施，将 SEH 安全校验机制提升到系统级别。3）堆数据保护机制。GS 编译技术和 SEH 安全校验机制都是用于保护栈中的数据，如果堆中发生溢出，则这两种保护机制都将失效。为此堆数据保护机制中加入了 Heap Cookie、Safe Unlinking 等一系列安全机制，给堆溢出增加更多的限制。4）DEP（data execution prevention）技术。DEP 技术用于将数据和执行代码区分开，不允许在被标记为“数据”的内存区域执行代码，阻碍了栈、堆中攻击代码的执行。5）ASLR（address space layout randomization）技术。ASLR 技术通过对关键地址的随机化，使一些堆栈溢出手段失效。文献 [5,6] 探讨了 Vista 中的 DEP 以及 ASLR 技术，文献 [7] 探讨了 DEP、ASLR 技术的演化过程。

本文详细介绍了这些保护措施的进展，并指出存在的不足，同时以攻击者的角度介绍了针对这几种缓解措施的攻击思路。

1. 针对 GS 编译选项的内存对抗技术

1.1 GS编译选项

绕过 StackGuard、StackShield 检测的方法由 Bulba 最早提出。随后 VisualStudio（以下简称 VS）2002 开始引入 GS 保护选项 1.0 版本，对栈进行保护，该保护基于编译器并能够启发式识别可能存在风险的函数。在函数的 prologue 里插入 security cookie[3]，之后在函数的 epilogue 里检查原 security cookie，如果发现二者不一致，则终止程序的执行。但是这种保护方法存在缺陷，即能够利用参数的覆盖或邻接的局部变量来绕过该保护。Litchfield[8] 于 2003 年提出了利用 SEH 覆盖旁路的办法。Skape[9] 给出了猜测计算 cookie 的方法，但有了覆盖 SEH 的方法后，这种方法已基本不用。通过 GS 保护，运行的操作系统能够有效检测并阻止大部分基于栈溢出的攻击。

1.1.1 校验栈cookie

缓冲区发生溢出时，栈中的变量布局如图 1 所示。

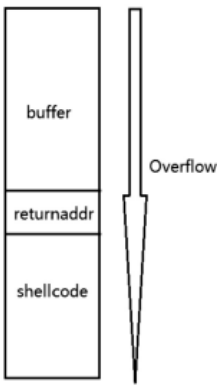

图1 缓冲区溢出时栈布局

当 GS 选项开启后，编译器会在函数的 prologue 中向栈中添加随机的 security cookie，当程序执行到函数的 epilogue 时程序会对原来的 security cookie 进行检查，如果发现两个 cookie 不符，则可以认定发生了缓冲区溢出。图 2 展示了加入 security cookie 后缓冲区发生溢出时栈的情况。

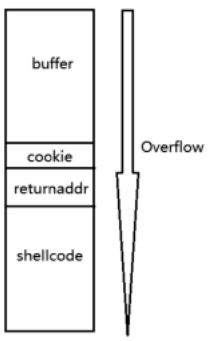

图2 添加cookie后缓冲区溢出时栈布局

相比图1，图2中的cookie保存于寄存器EBP之前，并且在prologue执行过程中将入栈的security cookie的副本保存到程序的.data区域。当缓冲区发生溢出时，原来的security cookie首先会被覆盖，然后才覆盖到返回地址。而溢出覆盖过程中攻击者并不能够预知prologue执行过程中security cookie的真实值，因此在函数的epilogue的检测过程中，系统会比较栈中的security cookie值和.data中的security cookie副本，如果发现二者不一致，则判定缓冲区发生溢出。此时程序会进入异常处理流程，而不会进行正常的函数返回。

GS选项提高了攻击者的攻击门槛，但也降低了程序的执行效率。为了使效率降低得最小，GS保护不会应用在所有的函数中。当编译器发现函数出现以下几种情况时，就不会对该函数进行保护：1）函数没有缓冲区；2）函数源代码中明确使用了无保护关键字；3）缓冲区非8字节类型且小于等于4个字节；4）函数的第一条语句包含内联汇编；5）函数包含变量参数列表。

1.1.2 变量重排

以上提到的校验栈cookie的方法可以有效检测并阻止缓冲区溢出攻击。但在某些情况下，在对security cookie检查之前一些局部变量可能会被读取使用，这就给攻击者提供了其他攻击思路：不溢出至security cookie而只是将部分局部变量的值覆盖。这样既能逃过对security cookie的检查，同时还可能破坏程序的正常执行逻辑。为了将这种攻击所造成的影响降到最低，微软引入了一个名为变量重排的技术。该技术在编译阶段会根据局部变量的类型将变量在栈中的位置重新排序，字符串类型的变量会被放到栈的高地址区，这样即便发生了溢出，也不至于影响到其他局部变量。

1.1.3 严格GS

1.1.1节中曾提到并不是所有的函数都会被GS保护，微软也考虑到了对于安全要求较高的开发需求，因此提供了#pragma strict_gs_check安全标识来对所有的函数采用GS保护。

1.2 针对GS保护的绕过方法

作为攻击者来说，他们同样会分析GS的工作原理，找出防御措施中的弱点以进行防御绕过。总结起来有以下几种方式可以有效绕过GS保护。

1.2.1 攻击SEH绕过GS保护

程序运行过程中栈帧上会保存若干个异常处理结构以便对函数中可能发生的异常进行处理。GS保护机制并没有对SEH进行保护，因此选择攻击SEH从而绕过GS是攻击者经常会选择的攻击方式[10]。异常处理结构保存于栈帧的高地址区，只要溢出数据足够长，使其能够覆盖到SEH的函数指针并触发一个异常，便可以成功劫持程序到伪造的SEH处理函数。SEH在内存中的位置如图3所示。

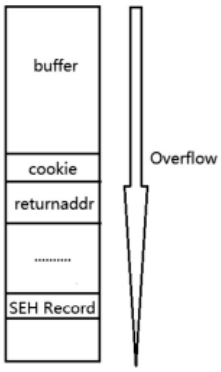

图3 SEH在栈中的位置

1.2.2 覆盖函数指针

类似于1.1.1节中所提到的在函数返回之前覆盖局部变量会破坏函数逻辑，因而当函数栈中存在函数指针时，如果能刚好覆盖虚函数指针，则有机会引导程序到指定的函数地址执行。

1.2.3 堆溢出攻击

如果溢出发生在堆上[11]，则GS保护完全失效。

2 针对SEH安全校验机制的内存对抗技术

从Windows XP SP2版本开始，微软引入了针对SEH攻击的校验机制SafeSEH[12]。并且随着攻击方式的多样化，微软从Windows Vista开始增加了对SEH更为严格的保护措施SEHOP。SafeSEH的原理并不复杂，在对异常处理函数进行一系列有效性检验之后，程序才可调用异常处理函数。如果发现异常处理函数不可靠，那么就将终止异常处理函数的调用[13]。SafeSEH的实现不仅需要操作系统的支持，还需要编译器的支持，二者缺一不可。本部分将对SafeSEH和SEHOP这两种保护技术进行讨论并介绍相关绕过措施。

2.1 SafeSEH机制

设计SafeSEH缓解机制的最初目的是为了对抗攻击者利用缓冲区溢出覆盖异常处理结构，从而劫持程序逻辑。

微软在 Windows XP SP2 及之后版本中均引入了 SafeSEH。SafeSEH 同样是在编译环节被设置，因此 Windows XP SP2 的本地文件不支持 SafeSEH，只有当编译阶段添加 SafeSEH 选项后镜像文件才能够支持该缓解措施。从 Windows Vista 开始这种情况有所改善，Vista 本地系统文件开始支持 SafeSEH，也是到此时，这种机制才真正开始发挥作用。SafeSEH 机制的核心思想是在调用异常处理程序时先对该处理程序的正确性进行验证，如果验证失败，则不继续执行该处理函数。具体来说：

1) 在编译过程中，如果开启了 SafeSEH 选项，编译器在生成二进制文件时，将正确的 SEH 处理函数地址提取并生成一张 SEH 函数表，用于后续的检查。

2) 当程序开始运行时，系统将正确的 SEH 函数表地址和共享内存中的一个随机数进行加密。加密后系统将 SEH 函数表地址以及其他部分信息一并存入 NTDLL 的数据内存中。在程序运行过程中，如果发生了异常，系统会进行以下判断：

(1) SEH 记录是否在栈中。

(2) SEH 处理函数地址是否在栈中。

(3) 验证 SEH 处理函数的有效性是否正确。具体来说会进行以下判断：

a) 判断处理程序地址是否在一个镜像空间内。

b) 判断镜像是否忽略 SEH 标志。

c) 判断镜像是否有 SEH 函数表。

d) 判断处理函数地址是否在 SEH 函数表中。

e) 判断程序是否是一个.NET 程序并且拥有 ILFlag 标志。

f) 判断 SEH 处理函数地址是否在一个不可执行页面上并且 DEP 是否开启。

g) 判断 SEH 处理函数地址是否在一个模块之外并且设置了 ImageDispatchEnable。

整个验证过程如图 4 所示。

2.2 SEHOP保护机制

SEHOP 是微软从 Windows Vista 开始引入的针对 SEH 验证的一个更为严格的 SEH 保护措施，全称为 structured exception handler overwrite protection。SEH 攻击通过软件漏洞或栈溢出，覆盖结构化异常的指针函数等，从而控制 EIP，执行恶意代码。SEHOP 通过一系列的对 SEH 结构的

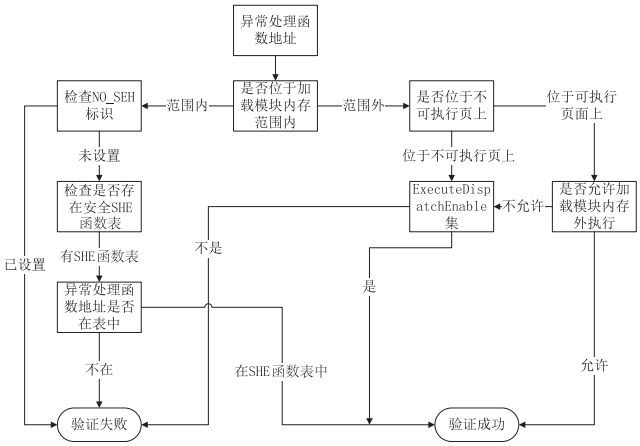

图4 SafeSEH验证流程

检测来检查 SEH 是否遭到攻击，其核心思想为：对所有 SEH 结构进行检查，尤其是对最后一个 SEH 结构，因为最后一个 SEH 结构中的异常处理函数地址指向了 NTDLL 中的一个函数。相比于 SafeSEH，这种保护措施不需要镜像文件自身的支持，只需要操作系统支持即可。这是因为当异常发生时，程序执行权限会转移到操作系统，因此不需要对程序本身做修改就可以实现对 SEH 的验证。下面的伪代码展示了 SEHOP 开启之后系统对 SEH 链的验证逻辑。

```
    BOOL RtlIsValidHandler(handler)
    {
    if (handler is in an image) {
    if (image has the IMAGE_DLLCHARACTERISTICS_NO_SEH flag
set)
    return FALSE;
    if (image has a SafeSEH table)
    if (handler found in the table)
    return TRUE;
    else
    return FALSE;
    if (image is a .NET assembly with the ILonly flag set)
    return FALSE;
  // fall through
 }
    if (handler is on a non-executable page) {
    if (ExecuteDispatchEnable bit set in the process flags)
    return TRUE;
    else
  // enforce DEP even if we have no hardware NX
    raise ACCESS_VIOLATION;
 }
    if (handler is not in an image) {
    if (ImageDispatchEnable bit set in the process flags)
    return TRUE;
    else
    return FALSE; // don't allow handlers outside of images
 }
    // everything else is allowed
    return TRUE;
```

```
    }
    // Skip the chain validation if the
    DisableExceptionChainValidation bit is set
    if (process_flags& 0x40 == 0) {
 // Skip the validation if there are no SEH records on the
 // linked list
    if (record != 0xFFFFFFFF) {
  // Walk the SEH linked list
    do {
   // The record must be on the stack
    if (record <stack_bottom || record >stack_top)
    goto corruption;
   // The end of the record must be on the stack
    if ((char*)record + sizeof(EXCEPTION_REGISTRATION) >stack_
top)
    goto corruption;
   // The record must be 4 byte aligned
    if ((record & 3) != 0)
    goto corruption;
    handler = record->handler;
   // The handler must not be on the stack
    if (handler >= stack_bottom&& handler <stack_top)
    goto corruption;
    record = record->next;
  } while (record != 0xFFFFFFFF);
  // End of chain reached
  // Is bit 9 set in the TEB->SameTebFlags field?
  // This bit is set in ntdll!RtlInitializeExceptionChain,
  // which registers FinalExceptionHandler as an SEH handler
  // when a new thread starts.
    if ((TEB->word_at_offset_0xFCA & 0x200) != 0) {
   // The final handler must be ntdll!FinalExceptionHandler
    if (handler != &FinalExceptionHandler)
    goto corruption;
  }
 }
    }
```

由以上代码可以总结出 SEHOP 的检测逻辑为 ：

1) 所有 SEH Record 必须要在栈中。

2) 所有 SEH Record 都必须是 4 字节对齐。

3) 所有 SEH Record 中的处理函数地址不能够在栈中。

4) 最后一个 SEH Record 的处理函数必须是 ntdll!FinalExceptionHandler。

5) 最后一个 SEH Record 的下一个指针必须为 0xFFFFFFFF。

2.3 针对SEH保护的绕过方法

虽然针对 SEH 的验证措施在不断增强，但是其中也存在一定的缺陷使得攻击者可以绕过这些保护措施。下面例举了几个绕过思路 ：

1) 通过覆盖 SEH 使其跳转到堆中执行 shellcode。这种方式在禁用 DEP 的环境中可以使用，并且只针对 SafeSEH 进行绕过。禁用 DEP 后，SEH handler 就可以位于除栈空间以外的非映像页面。这时如果将 shellcode 放置在堆中，通过覆盖 SEH 跳转至堆中执行 shellcode，就可以绕过 SafeSEH 保护。

2) 利用加载模块之外的地址绕过 SafeSEH，因为 SafeSEH 并不对加载模块之外的地址进行有效性检测。

3) 如果存在没有启用 SafeSEH 的模块，则可以利用该模块执行 shellcode。从 Windows XP SP2 版本开始，微软引入 SafeSEH 保护。由于用户并不时刻更新自己的操作系统版本，因此现阶段仍有大部分 Windows 操作系统的系统模块没有受到 SafeSEH 保护,攻击者可以利用这些模块。此外，并不是所有 VC6 编译的软件都受到 SafeSEH 保护，这时就可以使用这些软件中的指令作为跳板来绕过 SafeSEH。

3 针对堆数据保护机制的内存对抗技术

3.1 堆中元数据保护

3.1.1 元数据cookie

堆中元数据的 cookie 与栈中的 security cookie 相似，其主要用于检测堆溢出的发生。堆中元数据的 cookie 被放置在堆首部分原堆块的 segment table 位置，占一个字节。由于元数据的 cookie 只占一个字节，其变化区间为 0~256，因此在研究它生成的随机算法之后，攻击者仍旧有可能破解 cookie。

3.1.2 元数据加密

微软在 Windows Vista 及其后续版本的操作系统中开始使用堆中元数据加密这种安全措施。为了达到保护堆的目的，保存块首中一些重要数据的时候，会将这些数据与一个随机数进行异或运算，此随机数的大小为 4 字节。当要使用这些数据时，需要再进行一次异或运算还原出原数据，这样就实现了不能直接破坏这些数据。

3.2 针对堆保护的绕过方法

3.2.1 利用chunk重设大小攻击堆

从 FreeList上拆卸 chunk 时，Safe Unlink 可对双向链表的有效性进行验证。但是把一个 chunk 插入到 FreeList 时是没有进行验证的。在 FreeList 中插入 chunk 有两种情况，一是释放内存后不再使用 chunk ；二是当申请的空间小于 chunk 的内存空间时。第二种情况提供了可利用的机会，第二种情况下由于 chunk 的内存空间大于申请的空

间，剩余的空间会被建立成一个新的 chunk，并被插入到链表中。在从 FreeLink上申请空间的过程中，有一步骤是从 FreeLink 的第一个 chunk 开始依次检测，直到发现第一个需要的 chunk，将其从链表中拆卸下来。这使得若覆盖 chunk 结构，就会被 Safe Unlink 检测出来。但是 Safe Unlink 即使检测到了chunk 结构已被破坏，它也不会完全阻止如重设 chunk 大小这样的后续操作的执行。

3.2.2 利用Lookaside表进行堆溢出

Safe Unlink 对快表中的单链表没有进行验证。在拆卸链表的过程中可伪造指针，基于这一思路就可控制 Lookaside 表。当用户再次申请空间时，系统就会将这个伪造的地址作为申请空间的起始地址返回给用户，用户一旦向该空间写入数据，就会留下溢出的隐患。

4 针对 DEP 技术的内存对抗技术

现代计算机所存在的不能清晰区别数据和代码的天然缺陷是溢出攻击的根源。现阶段基本不可能去重构计算机体系结构，所以只能考虑通过前向兼容的修补来降低溢出带来的危害。DEP 的全称是 data execution prevention，即数据执行保护，该技术能够对内存执行额外检查以防止在系统中执行恶意代码。DEP 的主要思想就是将数据和执行代码分离开来，不允许在被标记为“ 数据 ”的内存区域执行代码。例如，当缓冲区溢出发生时，栈中的 shellcode 是不能够被执行的。为了达到这样的保护效果，需要一个标志对内存页面是否允许直行进行标记。具体来说，当应用程序试图从标记为“ 不可执行 ”的内存位置执行代码时，无论代码是否为恶意的，Windows 利用这个 DEP 标志立即发生异常并阻止代码的执行。DEP 技术并不是防止恶意软件的安装，而是监视已安装的程序，确定它们是否在安全地使用系统内存。

4.1 NX 支持与DEP的保护类型

NX/XD 标志存在于内存中的页面表中，该标志用于标识当前内存页面是否可以被执行。当该标志设置为 0，表示这个页面允许被执行 ；设置为 1，表示该页面不允许被执行。

微软自 Windows XP SP2 开始引入这种数据执行保护措施。根据实现机制的不同可以分成软件 DEP 和硬件 DEP。软件 DEP 就是 SafeSEH，硬件 DEP 才是真正意义上的 DEP，需要 CPU 的支持。为了配合 DEP 技术，AMD 公司开发了 No-Execute Page-Protection(NX)，Intel 公司开发了 Execute Disable Bit(XD)。对于操作系统来说具体有以下几种选项 ：

1) Optin。仅仅对 Windows 系统组件和服务应用 DEP 保护，对于其他程序则不会采用此保护。此选项 DEP 可以被应用程序动态关闭，多用于普通用户版的操作系统。

2) Optout。除了排除列表里面的程序和服务，其他所有程序和服务启用 DEP 保护。此选项 DEP 可以被应用程序动态关闭，多用于服务器版的操作系统。

3) AlwaysOn。对所有的进程启用 DEP 保护。此选项 DEP 不可以被关闭，现阶段只用于 64 位的操作系统。

4) AlwaysOff。对所有进程均关闭 DEP 保护，且 DEP 不能被动态开启，只用于特定场合。

4.2 针对DEP保护机制的绕过方法

目前针对 DEP 保护机制最为流行的绕过技术是 ROP（return oriented programming）技术 [14]，该技术利用已有的代码进行复用攻击。攻击者将函数中返回指令前面的若干条指令提取出来构造为 ROP 链，利用 ROP 链上的每一个 ROP gadget 将寄存器初始化，初始化完成后调用 VirtuallAlloc、VirtualProtect 等函数改变内存页面的属性值从而使该内存页面拥有可执行的属性 ；或者直接调用 API 关闭 DEP 保护。

5 针对 ASLR 技术的内存对抗技术

ASLR（address space layout randomization）概念在 Windows XP 时代被提出，但当时它只是用于对 PEB 和 TEB 进行简单的随机化处理，功能很有限。直到 Windows Vista 出现，该技术才真正发挥出作用 [15,16]。

缓冲区溢出攻击成功的一个重要条件是正确预测并发现某个固定地址，然后运行这个固定地址的指令来改变程序流程，只有这样才能够得到程序的控制权，否则只是造成程序崩溃。ASLR 技术针对缓冲区溢出，在加载程序的过程中不再使用固定的基址加载，而是对映像、堆栈、PEB、TEB 等随机化，干扰 shellcode 定位，阻止攻击者在堆栈溢出后利用固定地址定位到恶意代码并运行。干扰 shellcode 定位包括使攻击者猜测目的地址更加困难、阻止攻击者直接定位攻击代码位置等。

5.1 地址随机化的实现机制

1) 程序映像的随机化。系统启动时，对 PE 文件映射

到内存时加载的虚拟地址进行随机化处理并确定此虚拟地址，系统重启时此地址会发生变化。需要注意的是，映像随机化只是对基址的前两个字节进行随机处理。

2）堆栈地址的随机化。程序运行时，堆栈基址的选择是随机的。即同一个程序随意两次运行时的堆栈基址是不同的，因此各变量在内存中的位置就不确定。

3）PEB与TEB地址随机化。微软在Windows XP SP2之前，使用固定的PEB基址0x7FFDF000和TEB基址0x7FFDE000。在Windows XP SP2之后就开始使用具有随机性的基址,通过这种方法来增加攻击PEB中函数指针的难度。

5.2 针对ASLR保护机制的绕过方法

针对ASLR保护机制，攻击者往往会采取以下几种方式进行绕过：

1）攻击未启用ASLR的模块。虽然系统提供了映像地址随机化机制，但是由于兼容性问题，一些模块不得不抛弃ASLR而使用固定地址作为基址，这样攻击者通过这些未启用ASLR的模块便可以进行攻击。为了找到未启用的ASLR模块，可以使用OllyDbg的OllyFindAddr插件。

2）堆喷射技术[17]。攻击者利用堆喷射技术将恶意代码布局到固定地址，并且不会受到堆栈地址随机化的影响[18]。

3）利用信息泄露漏洞获取地址。攻击者通过信息泄漏漏洞计算出映像的基址，从而精确获得映像空间内任意指令的地址。

4）覆盖部分返回地址。2.1节中提到，映像随机化只对基址前两个字节进行随机化处理，其他字节不变。通过覆盖返回地址，使覆盖后的地址与基地址的距离相对固定，从而可以在基地址的附近寻找有用的跳转指令。但是由于覆盖返回地址时栈中的cookie会被破坏，因而这种方法并不被广泛使用。

6 结束语

现阶段的漏洞缓解技术可以防御大多数的漏洞攻击行为[19],但不可否认漏洞缓解技术还是存在一定的局限性。1）毕竟这只是一种缓解技术，不能阻止漏洞攻击的发生。2）目前的缓解技术几乎不能对基于数据流的攻击进行缓解与防御，一般只是对控制流劫持类攻击进行保护。3）旁路保护不完善、防护水平不高，不能很好地应对复合向量的攻击[20]。漏洞缓解技术也在不断进步中，未来必须考虑的是如何弥补在抵御复合向量攻击方面的不足，如何完善旁路保护。（责编 马珂）

参考文献：
[1] 徐有福，文伟平，万正苏. 基于漏洞模型检测的安全漏洞挖掘方法研究 [J]. 信息网络安全，2011，(8): 72-75.
[2] Microsoft. /GS（缓冲区安全检查）[EB/OL]. http://msdn.microsoft.com/zh-cn/library/8dbf701c.aspx,2014-11-01.
[3]Cowan C, Wagle P, Pu C, et al. Buffer overflows: Attacks and defenses for the vulnerability of the decade[C]//DARPA Information Survivability Conference and Exposition, 2000. DISCEX'00.Proceedings. IEEE, 2000, (2): 119-129.
[4] 蒋卫华，李伟华，杜君. 缓冲区溢出攻击：原理，防御及检测 [J]. 计算机工程，2003, (10):5-7.
[5]Whitehouse O. GS and ASLR in Windows Vista[C]//Black Hat DC, 2007.
[6]Litchfield D. Buffer Underruns, DEP, ASLR and improving the Exploitation Prevention Mechanisms (XPMs) on the Windows platform[EB/OL]. http://www.ngssoftware.com/papers/xpms.pdf，2005.
[7] 魏强，韦韬，王嘉捷. 软件漏洞利用缓解及其对抗技术演化 [J]. 清华大学学报：自然科学版，2011, 51(10): 1274-1280.
[8]Litchfield D. Defeating the stack based buffer overflow prevention mechanism of microsoft windows 2003 server[EB/OL]. https://www.blackhat.com/presentations/bh-asia-03/bh-asia-03-litchfield.pdf，2003.
[9] Skape. Reducing the Effective Entropy of GS Cookies [EB/OL]. http://www.leviathansecurity.com/wp-content/uploads/uninformed_v7a2.pdf, 2007-03.
[10] 余俊松，张玉清，宋杨，等. Windows下缓冲区溢出漏洞的利用 [J]. 计算机工程，2007, 33(17): 162-164. [17] 刘磊，王轶骏，薛质. 漏洞利用技术Heap Spray检测方法研究 [J]. 信息安全与通信保密，2012，(6): 70-72.
[11]Younan Y, Joosen W, Piessens F. Efficient protection against heap-based buffer overflows without resorting to magic[J].Information and Communications Security, 2006,(4307):379-398.
[12]Kimball W B, Perugin S. Software Vulnerabilities by Example: A Fresh Look at the Buffer Overflow Problem-Bypassing SafeSEH[J]. Journal of Information Assurance & Security, 2012, 7(1):1.
[13]XU Y, ZHANG J, WEN W. Windows Security: The gradual improvement of SEH mechanism [J]. Netinfo Security, 2009, (5): 47-50.
[14]Sotirov A, Dowd M. Bypassing browser memory protections in Windows Vista[C]// Blackhat USA, 2008.
[15]PENG J, WU H. Research of the Key Technology for the Windows Vista Memory Protection Mechanism [J]. Computer Engineering & Science, 2007, (12): 11.
[16] 彭建山，吴灏. Windows Vista内存保护关键技术研究 [J]. 计算机工程与科学，2007, (12):33-36.
[17] 刘磊，王轶骏，薛质. 漏洞利用技术Heap Spray检测方法研究 [J]. 信息安全与通信保密，2012 (6): 70-72.
[18]Shah S. Browser exploits-attacks and defense[EB/OL].https://eusecwest.com/esw08/esw08-shah.pdf,2008.
[19] Hanebutte N, Oman P W. Software vulnerability mitigation as a proper subset of software maintenance[J]. Journal of Software Maintenance and Evolution: Research and Practice, 2005, 17(6): 379-400.
[20]Sotirov A. Bypassing Memory Protections: The Future of Exploitation[C]//USENIX Security,2009.