doi 10.3969/j.issn.1671-1122.2011.07.014

Windows 7 操作系统关键内存防攻击研究

周虎生[1]，文伟平[1]，尹亮[1]，傅军[2]

（1. 北京大学 软件与微电子学院，北京 102600；2. 厦门市信息技术服务中心，福建厦门 361012）

摘 要：GS Stack Protection、SafeSEH、Heap Protection、DEP、ASLR 技术是 Windows 7 操作系统内存保护机制的关键技术，这五种技术对缓解缓冲区溢出攻击起到显著作用。文章以对 Windows 7 的内存保护机制的原理分析为基础，测试了 Windows 7 环境下绕过这些保护机制的可能性；最后对 Windows 7 内存保护机制抵御缓冲区溢出攻击的整体效果进行了分析，指出 Windows 7 仍然不能完全抵御缓冲区溢出攻击，并讨论了更全面提高系统安全性的改进方案。

关键词：Windows 7；内存保护机制；GS；SafeSEH；DEP; ASLR；Heap Protection

中图分类号：TP393.08 文献标识码：A 文章编号：1671-1122（2011）07-0038-04

Research on Key Memory Attack Windows 7 Operating System

ZHOU Hu-sheng[1], WEN Wei-ping[1], YIN Liang[1], FU Jun[2]

(1. *Department of Information Security, SSM, Peking University, Beijing* 102600, *China*

2. *Xiamen Information Technology Service Center,Xiamen Fujian* 361012,*China*)

Abstract: GS Stack Protection, SafeSEH, Heap Protection, DEP, ASLR are five key technologies for the memory protection mechanism of Windows 7, these technologies significantly contribute to mitigate buffer overflow attacks. Based on the analysis of these technologies, we also tested the possibility of bypassing the protection mechanisms in Windows 7. Evaluating the effect of resisting buffer overflow of Windows 7, we pointed out that Windows 7 couldn't resist the buffer overflow attacks absolutely. Finally several measures were introduced to improve system security more comprehensively.

Key words: Windows 7; memory protection; GS; SafeSEH; DEP; ASLR; Heap Protection

0 引言

在过去的十几年里，随着信息化步伐的高速迈进，计算机已经深入到社会生活中的方方面面，成为人们生活中不可分割的重要组成部分，小到个人计算机的娱乐应用，大到贯穿于一个企业生产、经营、管理全过程的企业级应用。因此，在今天的信息环境里，如何保证信息本身以及信息系统的安全性已然变得至关重要了。操作系统是计算机必不可少的软件平台，其安全的重要性不言而喻。微软公司的 Windows 操作系统是当前最流行的操作系统，新推出的 Windows 7 系统，更是以其界面美观、功能强大、资源占用低而倍受好评，有望成为最主流的 Windows 平台版本。在安全性上 Windows 7 相较之前版本有了长足的改进，集成了内存保护机制，主要包括 GS Stack Protection、SafeSEH（Safe Structured Exception Handling）、Heap Protection、DEP（Data Executive Protection）、ASLR（Address Space Layout Randomization）、PatchGuard 和 Driver Signing 等。

GS Stack Protection、SafeSEH、Heap Protection、DEP 和 ASLR 技术是 Windows 7 内存保护机制的关键技术，本文深入分析了这五种技术的工作机制，并对它们的脆弱性进行了分析，最后对 Windows 7 的安全性进行了总结并提出了改进方案。

1 GS Stack Protection

1.1 GS 机制分析

GS 是 VC++ 编译器提供的一个编译选项，对栈中内容进行检查和保护，以阻止攻击者通过栈中缓冲区溢出运行恶意代码。

收稿时间 2011-06-10

作者简介 周虎生（1986-），男，山东，硕士研究生，主要研究方向：系统与网络安全；文伟平（1976-），男，湖南，副教授，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；尹亮（1986-），男，湖南，硕士研究生，主要研究方向：系统与网络安全；傅军（1978-），男，湖南，工程师，主要研究方向：办公自动化、数据安全等。

GS 编译选项最早是在 Visual Studio 2002 中引入的，迄今为止已经更新了多个版本。如果将 /GS 编译选项置上，会给函数增加额外的代码，在函数开始处的代码会在初始化本地变量之后，给函数栈帧中局部变量和返回地址之间增加一个 32 位的随机数，即 GS cookie，标准栈帧如图 1（左）所示，Windows 7 系统中的文件主要由 Visual Studio 2003 (VS2003) 和 Visual Studio 2005 (VS2005) 编译器编译而成，使用 VS2003 GS 的栈帧布局如图 1（右）所示。

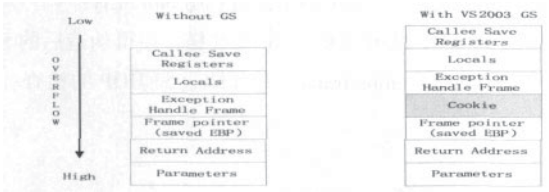

图1 无/GS(左)、visual studio 2003 /GS(右)

GS 编译选项在函数结束处也会增加代码。当攻击者利用缓冲区溢出对返回地址进行覆盖的时候，会同时改变 GS cookie 的值，这时编译器在函数返回时额外增加的代码会调用 _security_check_cookie() 函数检查 stack cookie 是否被修改，如果发现 cookie 被改变，将会终止当前程序并报错。一个典型的 GS 保护的反汇编代码如下 [1] ：

```
; prologue
      push  ebp
      mov  ebp, esp
      sub   esp, 214h
      mov eax,      __security_cookie   ; random value, initialized at module startup
      xor   eax, ebp
      mov  [ebp+var4], eax              ; store the cookie
      …
; epilogue
      mov   ecx, [ebp+var_4]            ; get the cookie from the stack
      xor    ecx, ebp
      call   __security_check_cookie    ; check the cookie
      leave
      retn   0Ch
```

GS cookie 是一个伪随机数，在进程启动时，这个值由系统调用 __security_init_cookie 函数计算出来，保存在加载模块的 __security_cookie variable 中。它的原始存储地址和程序映像基址的相对位移是一定的，由于系统中启用 ASLR 技术，GS cookie 的存储地址也是随机的。

为了防止攻击者通过溢出函数需要用到的覆盖局部变量和参数，新的编译器改变了栈帧中变量的布局，它将字符串缓冲区放在栈帧中的高地址，以确保字符串溢出时不会覆盖其他的变量。同时在栈帧地址处申请额外的空间，存放函数参数的备份，返回地址之后的原函数参数数据将不会再被使用。

额外代码的加入与运行导致运行效率的下降，在极端情况下效率降幅达到 42%[1]，为了优化使用 GS 编译选项的效率，编译器只对包含字符串和使用 _alloca 申请内存的函数加入 GS cookie[2]。C 编译器将元素大小为 1 或 2 字节且数组大小不小于 4 字节，定义为字符串类型，所以元素大小过大或者数组大小过小的变量将不受 GS 的保护。VS2005 SP1 版本之后的编译器加入了更有效的 GS 策略，即 strict_gs_check，如果将 #pragma strict_gs_check(on) 加入到需要 GS 保护的模块中，编译器将会在所有使用变量的函数中加入 GS cookie[2]。

2.2 GS 脆弱性分析

GS 是 VC++ 编译选项，许多基于 Windows 7 平台的应用程序都没有加入 GS 检查，应用程序中的任何一个组件没有被 GS 保护且存在缓冲区溢出漏洞的话，攻击者就可以利用该组件运行恶意代码。即使是 Windows 7 的系统文件(%systemroot%)中自带的文件仍有许多未被 GS 保护，这些文件必将成为系统安全性的短板，导致系统整体的保护效果降低。

通过前面的栈帧结构可以看出，即使是使用了 VS2005 之后的版本启用 GS 编译，缓冲区溢出仍然会覆盖其他的字符串缓冲区、结构化异常处理句柄、当前调用函数上层的数据。当函数的参数是对象指针或结构指针时，这些对象或结构存在于调用者的堆栈中，这也能导致 GS 被绕过，覆盖对象和虚函表指针，如果把这个指针指向一个用于欺骗的虚函数表，就可以重定向这个虚函数的调用，并执行恶意的代码。

至关重要的一点是，GS 机制并不保护结构化异常处理句柄，函数只在即将返回之前才进行 security_cookie 检查，所以可以利用溢出覆盖该句柄，在函数返回进行 __security_cookie_check 前触发异常，系统会遍历 SEH 链表，调用到已经被覆盖的异常处理历程的地址，从而获得进程运行的控制权。

此外，当利用任意地址可写漏洞时，可以将任意地址写入返回地址而不会破坏 GS cookie 的完整性，从而绕过 GS 的检查。

3 Heap Protection

3.1 堆保护机制分析

Windows 对于堆的保护，主要包含以下三个方面 ：

1) 双向链表安全摘除 ：空堆块结构如下所示 [3] ：

```
 FREE HEAP BLOCK
_HEAP_ENTRY
 +0x000 Size
 +0x002 PreviousSize
 +0x004 SmallTagIndex
 +0x005 Flags
 +0x006 UnusedBytes
 +0x007 SegmentIndex
_LIST_ENTRY
 +0x000 Flink
 +0x004 Blink
```

以下是从空堆块链表中摘除堆块时的代码 ：

```
mov dword ptr [ecx],eax
mov dword ptr [eax+4],ecx
EAX = Flink, EBX = Blink
```

攻击者借助于将 flink、blink 指向任意地址，控制 EAX、ECX 寄存器，就能写入任意 4 字节数据。从 Windows XP SP2 开始，在从空堆块链表中摘除堆块时，堆分配器实现了安全摘除。在使用 flink 和 blink 指针之前，堆分配器验证 flink->blink、blink->flink 是否都指向当前堆块，这样就阻止了攻击者控制 flink、blink 的企图。

2) 堆头 cookie ：堆分配器在堆块的头部储存了 1 字节的 cookie 值，如图 2 所示 ：

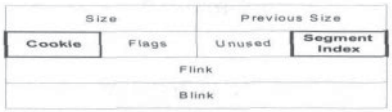

图2 堆头结构示意图

在从空堆块链表中摘除堆块时，堆分配器将检查这个 cookie 值，如果堆块被覆盖了，cookie 值无法匹配，堆分配器就认为堆被破坏了。

3) 元数据加密 ：在 Windows 7 中更进一步，堆头部所有比较重要的成员数据都和一个 32 位随机数进行异或（相当于加密），在使用这些成员数据时，再异或一次（相当于将原数据解密出来）。

这三项保护措施有效地阻止了攻击者覆盖堆块头部、在堆中创建虚假堆块，实现了对堆的强有力的保护。

3.2 堆保护脆弱性分析

在 Windows 7 中，一些通用的堆溢出方法不再有效，但攻击者可以覆盖堆中存储的一些数据，如函数、虚函数地址表、对象指针等，当程序调用了一个被改写的指针时，攻击者就可能控制程序的运行流程，引发攻击。图 3 是一个典型的类和类实例在堆中的布局 ：

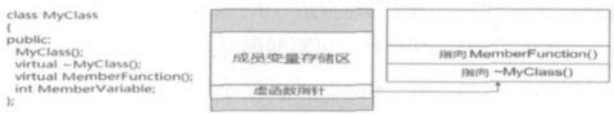

图3 类在堆中的布局示意图

攻击者如果修改了 MemberFunction() 或者 ~MyClass() 函数指针，那么就将执行攻击者指定的 shellcode 代码。

4 SafeSEH

4.1 SafeSEH机制分析

SafeSEH 是另一种安全机制，它可以阻止利用 SEH 的 exploit。SafeSEH 包含两种机制 SEH 句柄验证和 SHE 链验证。

从 vs2003 开始，增加了一个编译选项 /SAFESEH，如果启用该编译选项，编译器会在生成的文件头部包含一个异常处理句柄表，当异常发生时，异常处理机制会在调用异常处理函数前检查异常处理链是否被修改过，系统会从头到尾遍历异常处理链表，并逐个验证它们的有效性。

SEHOP ，SEH 链验证，又叫 Dynamic SafeSEH，是 SafeSEH 机制的扩展。SEHOP 作为 Structured Exception Handling 的扩展，用于针对程序中使用的 SEH 结构进行进一步的安全检测。SEHOP 的核心特性是用于检测程序栈中的所有 SEH 结构链表，特别是最后一个 SEH 结构 [4]，它拥有一个特殊的异常处理函数指针，指向一个位于 ntdll 中的函数。如果 SEH 记录被覆盖，那么 SEH 链就会被破坏，SEH 链的末尾将不再是特定的指纹。

4.2 SafeSEH脆弱性分析

如果程序编译的时候没有启用 SafeSEH 并且至少存在一个没启用 SafeSEH 的加载模块（系统模块或程序私有模块）。这样就可以用这些模块中的 pop/pop/ret 指令或者其它等价指令地址来绕过保护。事实上，建议寻找一个程序私有模块（没有启用 /Safeseh），因为它可以使 exploit 稳定地运行在各种系统版本中。如果找不到这样的模块地址也可以使用系统模块中的地址，它也可以工作（如果没有启用 SafeSEH）。

针对 SEHOP 的方法是伪造 SEH 链，使得伪造后的 SEH 链末端能指向 validation frame，从而绕过 SEHOP 的检查。

5 DEP

5.1 DEP机制分析

DEP（Data Executive Protection）数据执行保护，是一项阻止代码在标有 non-executable 标志位的内存页上执行的保护机制。DEP 有两种模式，如果 CPU 支持内存页 NX 属性，就是硬件支持的 DEP。如果 CPU 不支持，那就是软件支持的 DEP 模式，这种 DEP 不能阻止在数据页上执行代码，但可以防止其他的 exploit（SEH 覆盖）[5]。

在 Windows 中，系统默认 EXE 和 DLL 中 .text 段的页面为可执行页面，其他的页面不包含执行代码。在 Windows XP 及其之前的操作系统版本中，没有对代码执行区域进行限制。DEP 机制采用硬件或软件的方法将栈、堆等敏感区域设置为不可执行后，即使攻击者溢出成功，并跳转到恶意代码的入口地址，但由于该页面不可执行，将触发系统异常而使程序终止，从而有效地阻止了恶意代码的运行。

5.2 DEP脆弱性分析

1) ret2libc/ROP。可以在库中找到一段执行系统命令的代码，用这段库代码的地址覆盖返回地址，这样可以不直接跳转到栈中的 shellcode 去执行，而是去执行程序外部的库中的代码，这些库中被执行的代码也可以看做是植入的 shellcode 代码。所以，虽然 DEP 技术禁止在堆栈上执行代码，但库中的代码依然是可以执行的，可以利用这点，组合库中的一部分代码来达到特定的目的 [6]。2) 闭进程的 DEP - -NtSetInformationProcess、SetProcess DEPPolicy。DEP 可以设置不同的模式，进程的 DEP 设置标志保存在内核结构 KPROCESS 中，这个标志可以用函数 NtQueryInformationProcess 和 NtSetInformationProcess 通过设置 ProcessExecuteFlags 类来查询和修改。当启用 DEP 时，ExecuteDisable 被置位 ；当禁用 DEP 时，ExecuteEnable 被置位。当 Permanent 标志置位时，这些设置是最终设置，不可以被改变。绕过 DEP 保护的原理在于调用函数 NtSetInformationProcess 来关闭 DEP 保护，在调用的时候指定信息类 ProcessExecuteFlags(0x22) 和 MEM_EXECUTE_OPTION

_ENABLE（0x2）标志，然后返回到 shellcode。为了初始化 NtSetInformationProcess 函数调用，需要借助于 ret2libc 技术来完成参数的设置，并且需要对栈的布局进行针对性的设计，并把控制传回到用户控制的缓冲区中。3）使 Shellcode 所在内存可执行。可以配合 ret2libc 使用 VirtualAlloc、VirtualProtect 函数，使内存可执行；或者通过 WriteProcessMemory 将 shellcode 拷贝到 .code 段中，从而具有可执行的权限。

6 ASLR

6.1 ASLR 机制分析

加载地址随机（ASLR）是一种预防性的安全机制，在 Windows Xp 及 2003 系统中，只能实现对 PEB，TEB 等关键结构的地址随机化，但是在 Windows 7下，则几乎使其应用到所有的结构、程序映像、堆栈等，ASLR 对系统关键地址的随机化主要包含四个方面：

1）堆地址的随机化；2）栈基址的随机化；3）PE 文件映像基址的随机化；4）PEB(Process Environment Block，简称 PEB) 地址的随机化。线程栈、进程堆、PEB 的地址随着进程每一次运行而不同，EXE、DLL 等则是随系统每一次启动而不同。

ASLR 使得溢出不再能利用固定的程序位置、结构位置来定位执行溢出攻击代码。如经典的栈、堆溢出中可利用系统 DLL 中的指令对程序流程进行控制，如利用 JMP、ESP 等。当其位置不再固定，跳转到目标指令就变得困难，许多攻击代码不能成功执行。ASLR 和其他安全机制互补互足，例如数据执行保护（DEP）等，共同为抵抗内存漏洞提供了强有力的保护。

为了测试 Windows 7 ASLR 的随机化程度，进行 4 项测试，记录每一次它运行时的相关数据：输出代码中某一函数的地址，测试进程的映象文件加载地址随机化程度；输出一个变量的存储地址，测试栈地址的随机化程度；输出 PEB 地址，测试 PEB 地址随机化程度；输出 malloc、HeapAlloc、HeapCreate 三个函数的返回地址，测试堆地址的随机化程度。

6.2 ASLR脆弱性分析

1）利用静态加载的 DLL 与 EXE 映像。Windows 7 系统默认只随机加载使用 /DynamicBase 连接的程序。而众多的第三方软件，组件都没采用这种连接。特别是浏览器中大量第三方组件的存在，使系统安全性大大降低。2）部分覆盖。通过对溢出大小的精确控制，只覆盖关键指针的几个字节。分析 ASLR 可知，随机加载只是地址的部分高位是动态的，程序中的相对地址是不变的。则通过修改指针的低位可得到稳定的定位。3）暴力方式。当溢出只发生在一个线程中且不会造成整个进程的崩溃时。此时如果以暴力猜测的方法不断尝试不同的跳转地址，将可能触发异常处理例程，进程不会崩溃，甚至从外部无法获得攻击的任何迹象。由于系统 DLL 只对基址的 8 比特进行了随机化 [7]，攻击者有 1/28 的几率得到正确地址 (PEB 基址的随机化只用到了 4 比特)。4）内存信息泄漏。如结合信息泄漏获得一些内存信息，特别是指针地址，就可算出许多信息，很可能就使下一步攻击成功。比如可算出映像是否加载，其附近结构的地址，如栈地址、堆块地址。甚至由于 ASLR 的局限性，程序加载地址只在启动时随机。从而得到一个程序的加载地址，可算出本次系统中其他程序的加载地址，使 ASLR 形同虚设。

7 总结与建议

Windows 7 在安全性方面相对于之前的系统有了相当大的提升，尤其是在内存防护机制上下了很多功夫，但是其有效性仍不乐观，尽管提高了攻击的门槛，但是攻击者仍然有很多机会绕过这些安全机制：

较为先进的攻击手段，如 ROP，已经有一套成熟的理论和方法，能够轻易绕过没有同时启用 ASLR 的 DEP 的模块，如果能读取进程空间的内存，或者能寻找到堆栈保护百密中的一疏，哪怕只有单独的一条地址，都足够获取到 DLL 的加载基址从而绕过 ASLR 和 DEP 的组合。

针对前面几个章节提到的 Windows 7 的内存保护机制的脆弱性，提出一些改进升级建议：

1）开发新的安全防护机制。如微软应提供一种机制，开启所有 EXE 和 DLL 随机加载时还能提供例外控制，类似于 OptOut 启动项下的 DEP 例外。虽然现有机制可通过修改注册表开启全程序随机加载，减少攻击者利用空间以增加安全性。但使执行第三方程序不稳定，实用性不高。如通过例外不稳定第三方程序，可以更好地应用高级防护，提高系统安全性。2）加强对危险程序及组件的监控。对于不能支持安全防护机制的程序，系统在加载、受与权限时应另加处理减小其被攻击利用的危害性。3）微软公司应加强与第三方软件的合作，推广代码安全防护机制。文章中的很多攻击手段都是利用没有采用安全编译连接的软件或组件。而 Visual Studio 默认是不使用安全连接 /NXCOMPAT 与 /DynamicBase，不利于第三方软件兼容 Windows 安全机制，更不用说别的语言开发的程序。微软公司应帮助、刺激主流软件加入到其防护机制中，用更高效安全的代码保护功能引导第三方软件的开发更新，以更好地保护系统安全。（责编 杨晨）

参考文献：
[1] SOTIROV A, DOWD M. Bypassing browser memory protections[C]. 2008.
[2] Michael Howard. Protecting Your Code with Visual C++ Defenses. http://msdn.microsoft.com/en-us/magazine/cc337897.aspx#S1.
[3] Richard Johnson. Windows Vista Exploitation Countermeasures. Microsoft. 2006.
[4] Matt Miller. Preventing the Exploitation of SEH Overwrites. http://www.uninformed.org/?v=5&a=2&t=pdf, September 2006.
[5] Wikipedia. http://en.wikipedia.org/wiki/Data_Execution_Prevention
[6] skape, Skywing. Bypassing Windows Hardware-enforced Data Execution Prevention. http://www.uninformed. org/?v=2&a=4&t=txt.
[7] Ollie Whitehouse. An Analysis of Address Space Layout Randomization on Windows Vista. Symantec, 2007.