OverflowDungeon:一种新颖的溢出攻击防护系统*)

韩 宏 卢显良 任立勇 杨 宁
(电子科技大学计算机学院 成都 610054)

摘 要 溢出攻击是网络上威胁最大的一种攻击方式,现有的防护技术存在不同的缺陷。本文提出并实现了一种新颖的防护机制,它从阻止溢出攻击产生破坏性效果入手,抑制攻击行为。该方式不需要修改编译器,或修改操作系统,具有很高的实用性,它采用了统计方式,可有效防护 Dummy return 攻击技术。试验结果表明,系统不仅能有效地阻止溢出攻击,在效率方面相对于其他方式也有显著优势。

关键词 溢出攻击, API Hook, 函数调用 Tracing, API 调用基因校验

OverflowDungeon: A Novel Overflow Attack Prevention System

HAN Hong LU Xian-Liang REN Li-Yong YANG-Ning
(Computer Science Department of UESTC, Chengdu 610054)

Abstract Overflow attack does the broadest harm to network applications. There are techniques which can prevent the kind of attack. However they have obvious flaws. This paper presents a novel mechanism of overflow attack prevention, which prevents shell code from exploiting resources of system. In this way, we need not change compiler or operating system. It uses statistic way to prevent dummy return trick. The implementation shows that it can prevent overflow attack and has advantage of performance over other mechanisms.

Keywords Overflow attack, API Hook, Function calling tracing, API calling gene checking

1 引言

缓冲区溢出是严重威胁网络安全的攻击方式,占网络攻击的80%。而对它的防范,目前还没有特别有效的防御手段。其中主要包括四种方式:(1)修改编译器(主要是 Linux 下的 gcc);(2)源代码静态分析;(3)修改操作系统内核(比如修改 Linux);(4)网络入侵检测分析溢出攻击特征。

这些方法都有明显缺陷,第一种方式的代表有 StackGuard 和 StackShield,他们通过在 gcc 编译器上增加模块,对函数调用进行保护。StackGuard 在函数中插入代码,检查函数返回时是否因缓冲区溢出修改了某些特殊字(Canary Word),并对常用函数调用采取了利用硬件特性的高效保护措施[1]。而 StackShield 则是在函数中插入代码保存堆栈映象,当函数返回时将栈映象还原[2]。这种修改编译器的方式最大的问题是,必需编译器支持,而且需要源代码,许多商业化的编译器并不支持这种机制。同时,商业化系统也不可能得到其源代码,对于众多已存在系统这一方式也无能为力。此外,在函数中插入保护代码,开销是显著的。为了弥补没有源代码这个缺陷,有研究尝试分析二进制文件,在机器码中插入保护代码,这种方式有许多缺陷性,比如无法精准分析代码,在某些情况下也不具备插入代码的条件[3]。

对于源代码静态分析,相关研究工作很多,比如文[4,5]。这一方式的缺点是无法弥补已存在系统的漏洞,并且需要源代码。

至于第三种方式,在开源系统上研究工作比较多,最著名的例子是 Solaris 下非执行栈的补丁,而这一方式被移植到了 Linux 上,主要有 Open Wall[6],Pax[7]。该方式通过将栈的内存页属性变为非执行,从而阻止在栈上运行代码。这一方式有两个弱点,一、对于没有源代码的商业操作系统(比如 Windows)将无能为力,二、在栈或堆上运行某些代码是现有软件系统使用的技巧,比如 gcc 为了对内嵌函数支持使用了一种称为蹦床(trampoline)的技术,就是在栈上运行代码[8],而在 Windows 平台下,许多类库为了封装消息循环,运行期在堆上分配代码以充当消息处理函数,比如 VCL 和 ATL。直接阻止在栈或堆上运行代码会导致许多软件无法运行。因此折衷的办法是设定某些软件可在栈或堆上运行代码,这又留下了被攻击的可能。

关于最后一种方式,相对而言是一种辅助的手段。他们通过检查网络上的报文,分析某些特征,例如 Prelude。在 Linux 下的溢出攻击经常会采用连续多个空操作(NOP)。但这些特征并不非常有效,比如 Windows 下的攻击就不会采用这一方法。另外,现在的攻击经常会将溢出代码加密,这样就很难分析特征了。

本文提出了一种新的思路,它允许代码溢出发生,但是遏制溢出行为造成破坏性结果。因为溢出攻击最终必须使用操作系统的资源,比如生成超级用户,或形成远程 Shell 等[9]。而系统资源的使用是通过 API 调用,那么只要防止攻击对系统 API 调用就可阻止溢出攻击的最终目标。基于该思路,我们提出并实现了 Windows 下的溢出攻击防护系统 OverflowDungeon。

2 基于系统调用基因校验的溢出攻击防护系统

2.1 堆栈溢出攻击

Buffer Overflow 攻击分为基于栈和基于堆两种。基于栈

*)本文受信息产业部生产发展基金项目网络安全集成防护系统支持项目代号信运部[2002]546。**韩 宏** 博士,主要研究方向:网络安全、软件工程。**卢显良** 教授,博士生导,主要研究方向:计算机网络、操作系统。**任立勇** 博士,主要研究方向:计算机网络、操作系统。

的攻击危害和威胁都是最大的。其原理是，利用一些有缺陷的函数调用，比如 C 语言中的 strcpy 函数，将函数的返回地址覆盖，使函数返回时跳转到攻击者给出的代码上。图 1 给出了基于栈的溢出攻击原理：该图是发生溢出攻击时的栈映象，返回地址被攻击者选择的地址覆盖并指向攻击者的代码入口，当被调用函数返回时，该返回地址被弹入程序指令寄存器，从而溢出代码获得了执行权限。

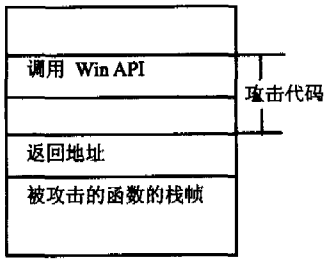
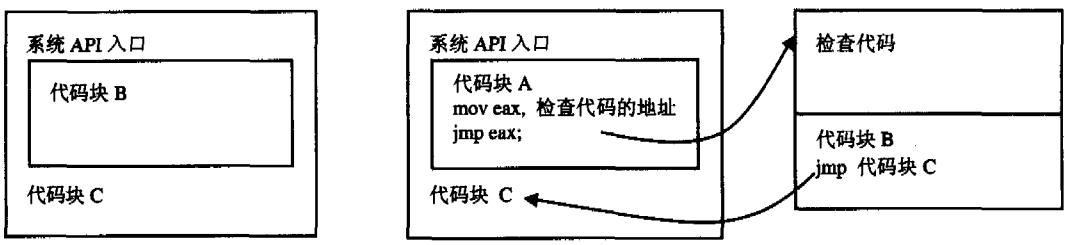


图 1　基于栈的溢出攻击

溢出攻击主要分为两个阶段：attack vector 和 exploit payload。获得执行能力的阶段称为 AttackVector 阶段。当溢出代码获得执行权限后，必须作相关后续工作才能完成攻击目标，这一后续工作称为 Exploit payload 阶段[9]。而现有防止攻击的方式都是从阻止攻击代码获得执行能力入手，如果我们能遏制溢出攻击的 Exploit payload 阶段，也能阻止其最终目标。从文[9]我们发现，不论是简单地生成超级用户，还是生成远程 Shell，不论是初级的采用被动模式的远程 Shell，还是以绕过防火墙为目的的反向远程 Shell 等，入侵代码必须使用系统资源来完成其攻击目标，也就是调用系统 API。比如生成超级用户必须使用 CreateProcess 调用 Cmd 程序，而远程 Shell 必须使用 socket 函数。如果攻击者想获取被攻击主机的文件信息还必须使用系统的文件调用接口。如果我们能防止非法调用系统 API，那么就能最终遏制攻击行为。本文提出的技术就是从这一思路出发：即使攻击代码取得了 CPU 的执行权限，但他无法造成任何有危害的结果，因此该方式被成为 Overflow 的 Dungeon。

2.2　函数调用 Tracing

要判断是否为非法函数调用，我们首先必须能跟踪对系统 API 的调用。Windows 平台的系统调用是通过几个动态链接库提供的，比如 Kernel32. dll 就包含了最基本的系统调用。如果能在程序调用 API 之前截获它们，就能对 API 调用进行分析。我们采用了两段跳跃的方式来完成对 API 调用的截获。首先当 Kernel32 加载到内存后，修改我们要截获的 API 的入口地址的代码内容，替换成一段转跳代码 A，并把被替换的代码 B 拷贝到我们的检查函数的尾部，然后程序转移到我们的检查函数，检查函数分析完毕后，如果没有问题则执行代码块 B 然后返回代码 A 的后部继续执行，否则结束进程。整个过程的原理如图 2 所示。



a) 正常的系统 API 代码映象　　b) 被 Hook 后系统 API 代码映象

图 2

因为 Windows 下每个进程都要加载 Kernel32. dll，只要我们修改这些进程的 Kernel32. dll 的映象就能完成对系统 API 的截获。现在的问题是如何进入到被保护进程的地址空间修改该空间中 Kernel32 的代码内容，以及如何将自己的检测代码注入到被保护的地址空间中。关于这个问题文[10]给出了详细描述，我们采用了远线程调用 LoadLibrary 方式完成了该过程。通过这一方式，搭建了运行时截获指定进程的系统 API 调用的框架。下一小节讨论了如何检查系统调用合法性的原理。

2.3　基于 API 调用基因校验的系统调用合法性验证

关于系统调用是否合法，应该从调用发起的地址加以验证。基本的观点是：如果调用从栈或堆上发起，那么调用是可疑和危险的。如何分析调用发起的位置呢？这必须从函数调用的过程分析入手。当代码进入被调用的函数时，栈顶此时指向的内容保存的是返回地址，因此查看该地址，如果在栈底和栈顶之间，就说明调用来自栈，那么调用是危险的。栈底的地址可以用如下汇编语言得到：mov eax, fs:[0x4]; mov EspBase, eax; 对于判定系统调用是否来自堆，我们只需要从分析 PE 格式出发，遍历程序当前所有模块（包括执行文件和动态链接库）的代码段，如果调用不是来自栈又不是来自所有模块的代码段，那么它就是来自堆上的。

以上可以基本推断系统调用点的安全性。但是，存在两个问题没有解决。一、正常软件可能在栈或堆上运行代码，二、黑客可以容易地绕过以上的简单判断。一种称为 Dummy return 的技术可以绕过简单检查。其工作方式如下：

（1）先在执行文件的代码段寻找 ret 指令，这非常容易做到，只要在执行文件或 Kernel32. dll 中寻找，其地址在每次加载时是固定的。例如该 ret 指令的地址是 0x00432071。而调用系统 API 后，攻击代码真正希望返回到的是栈上的地址 0x00125677。

（2）构造代码如下：push 0x00125677; push 0x00432071; jmp 系统 API 地址；代码中第三个语句使攻击代码转跳到系统的 API 地址。当 API 返回时，如果简单判定返回地址 0x00432071 来自代码段，那么就被攻击者欺骗了。因为 API 在返回后会执行 0x00432071 处的代码也就是 ret，而该代码将返回到栈顶指向的内存中保存的地址，也就是 0x00125677。该地址指向攻击者栈上的代码。要破解这种技巧还比较困难，如果连续检查多层次，将会带来效率问题。举例讲，正常的调用，其返回总是在代码段，难道需要一直跟踪下去吗？而文中给出的例子只是简单的 Dummy return 技巧。如果不直接返回到 ret 所在地址，那么就更难判定了。

根据以上两个原因，我们必须寻找更为有效的方式解决

简单检测带来的问题。本文提出了一种称为 API 调用基因校验(ACGC)的方法。我们发现,正常系统 API 调用总是从程序的固定地址发出。因此,这些调用地址就是判定在某个具体程序中具体 API 调用合法性的依据。另外,API 调用时,栈顶距栈底的偏移量也是一个重要的特征。这个偏移量实际是当时函数调用序列决定的。因此,我们定义 API 调用基因为

ACG={CallingAddress, StackOffset}

其中 CallingAddress 为调用 API 的地址,StackOffset 为调用时栈顶距栈底的偏移量。

对于一个 API 而言,其调用基因特征为 ACGS,是 ACG 的集合:

ACGS = $\{ACG_i\}$ i 从 0 到 n。

API 调用基因校验算法 ACGC 工作分为两步。

(1)抽样被保护的程序,获取指定系统 API 调用的基因库

在实现的程序中,我们主要抽样了几个对溢出攻击最为重要的系统调用,比如 CreateProcess, socket, CreateRemoteThread, CreateFile 等。因为对于溢出攻击而言,其利用系统资源主要有以下几种途径:直接调用某个进程,比如 Cmd.exe;构造远程 Shell;在被攻击的进程中感染文件;通过远程注入到其他未被保护的进程中完成攻击行为。通过对相关 API 的监视,我们能阻止相应的攻击行为,同时,提高系统运行的效率。抽样得到的基因库数据被保护在被监测的执行程序所在的目录。以下是一个样本数据:

CreateProcess ID(1)

FunCallingCount = 4096

CallingInfoCount = 1

Module = Performtest. exe RelativeReturnAddress = 00050209 EspOffset = 00000AE4 instr 4B Count = 4096

……

(2)运行时检查

当收集到足够信息后,我们就可对抽样程序实施保护。其方法是将我们的监测库远程注入到被保护系统中,当系统 API 被调用时,和采样数据比对,如果 EspOffset 或 RelativeReturnAddress 不相同,则认为是攻击。

3 实验结果

实验对系统的两个部分进行了测试,第一部分是系统是否成功阻止溢出攻击,第二部分针对保护系统的效率进行了分析。我们对有问题的程序构造了溢出攻击的 ShellCode,该代码调用了 CreateProcess 去激活 Cmd. exe,在加入保护后,保护模块关闭了被攻击的进程。

对于系统的效率,我们分两个方面进行了测试。第一,对单个 API 调用测试;第二,对一个网络程序在注入保护模块前后执行效率进行对比,网络程序选择了 FTP 服务器 Server-U。以下是对 CreateProcess, socket 和 GetProcAddress 三个 API 测试的结果。CreateProcess 以连续调用 4096 次为一组测试了 5 组,socket 以连续调用 53247 次为一组测试了 5 组,GetProcAddress 以连续调用 96437183 次为一组测试了 5 组。a%为加入保护系统消耗时间和无保护系统消耗时间的比值。表 1 为测试结果,表 2 为用 Server-U 传输文件的结果,每个文件传输了 5 次。

表 1 有保护和无保护时 API 调用效率比较

API	无保护(ms)	有保护(ms)	a%
CreateProcess	5135	4791	93
socket	4522	4625	102.2
GetProcAddress	5218	7587	145.4

表 2 有保持和无保护的 API 调用效率比较

文件大小(kB)	无保护(s)	有保护(s)	a%
28,419	29	29	100
54,987	61	61	100
249,586	282	282	100

对于重量级 API 调用,即需陷入内核的比如 socket,CreateProcess 影响很小,甚至还出现了效率提高的现象,而对于 GetProcAddress,因不陷入内核而只是简单的地址转移,所以效率下降明显。但是,溢出攻击者都会根据 PE 格式自己编写 GetProcAddress 的替代函数,因此我们并不需要钩挂这个函数。从 Server-U 测试的结果分析,对实际程序没有影响。

结论 对缓冲区溢出攻击,本文采用的方法在成功阻止攻击的情况下,系统执行效率没有明显降低。同时,相对现有方式,它不需要修改已有系统,不需要修改源代码和编译器,不需要修改操作系统,同时能够容忍正常程序在栈和堆上运行代码,并且能防止攻击程序对保护系统的"欺骗",是有效而实用的。

参 考 文 献

1 StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow. Cowan C, Pu C, Maier D, et al. In: Proc. of 7th USENIX Security Symposium, 1998

2 StackShield. http://www.angelfire.com/sk/stackshield.

3 A Binary Rewriting Defense against Stack Based Overflow attacks. Manish Prasad Tzi-cker Chiueh

4 Ghosh K. Analyzing Programs for Vulnerability to Buffer Overrun Attacks

5 An automated approach for identifying potential vulnerability in software. In: Proc. of the 1998 IEEE Symposium on security and privacy

6 OpenWall Project. Linux Kernel Patch from the Openwall Project. http://www.openwall.com/linux/

7 PaX. http://pageexec.virtualave.net/pageexec.txt.

8 Breuel T M. Lexical Closures for C++. In: Proc. USENIX C++ Technical Conferenc, Denver, CO, October 1988

9 Levy E. The Shellcode Generation. IEEE security & privacy Sept./Oct. 2004

10 Richter J. Windows 核心编程. 机械工业出版社,2000