第38卷第6期
2010年6月

同济大学学报(自然科学版)
JOURNAL OF TONGJI UNIVERSITY(NATURAL SCIENCE)

Vol.38 No.6
Jun. 2010

文章编号：0253-374X(2010)06-0917-08

DOI:10.3969/j.issn.0253-374x.2010.06.024

基于 k 循环随机序列的动态缓冲区溢出防御

江建慧[1]，章力源[1]，金　涛[2]，陈　川[2]

(1.同济大学 计算机科学与技术系，上海 201804；2.上海轨道交通信息管理中心，上海 201103)

摘要：面向 Intel 80×86 体系结构和 C/C++ 语言，介绍了栈缓冲区溢出攻击的基本原理及攻击模式，分析了现有的动态防御典型方案的优点与不足.结合基于随机地址空间与签名完整性的防御思想，提出了一种基于 k 循环随机序列的动态缓冲区溢出防御方案.该方案能够在极大概率下防御多种模式的缓冲区溢出攻击，解决了“连续猜测攻击”的问题，并使软件具有一定的容侵能力.

关键词：缓冲区溢出；栈溢出；软件漏洞；动态检测；容侵

中图分类号：TP 393.08　　**文献标识码**：A

Dynamic Buffer Overflow Prevention Based on k Circular Random Sequence

JIANG Jianhui[1], *ZHANG Liyuan*[1], *JIN Tao*[2], *CHEN Chuan*[2]

(1. Department of Computer Science and Technology, Tongji University, Shanghai 201804, China; 2. Shanghai Rail-Transit Information Management & Administration Centre, Shanghai 201103, China)

Abstract: The paper presents an analysis of the principle of stack buffer overflow attacks and basic attack patterns for Intel 80×86 architecture and C/C++. Then, the merits and drawbacks of the existing dynamic buffer overflow prevention methods are discussed. On the basis of the address obfuscation and integrity checking, this paper presents a new dynamic buffer overflow prevention method based on k circular random sequence. This improved prevention method can defend attacks of multiple patterns with high probability and enhance the intrusion-tolerance capability of the vulnerable software.

Key words: buffer overflow; stack overflow; software vulnerability; dynamic prevention; intrusion tolerance

缓冲区溢出是最为常见的软件漏洞之一，对于 Intel 80×86 体系结构而言，进程的内存分布结构和 C/C++ 灵活的内存操作是造成缓冲区溢出漏洞的主要原因[1-2].栈缓冲区溢出是最主要的缓冲区溢出方法，攻击者通过溢出栈上的数据可以改变进程的执行路径，取得进程的控制权并执行恶意代码[3].相比其他攻击手段，栈缓冲区溢出攻击的形式更加灵活多变，检测难度更大.因此，研究栈缓冲区溢出的防御方案具有现实意义.

本文介绍了缓冲区溢出的基本原理及栈缓冲区溢出的攻击模式.分析了基于签名完整性和随机算法的防御方案，并简要介绍了其他主流的动态防御模型.针对现有方案的不足，提出了基于 k 循环随机序列的动态缓冲区溢出防御方案.对新方案进行了有效性分析及性能测试，证明了该方案能在极大概率下防御各种攻击模式，且不造成过大性能负载.

1　栈缓冲区溢出攻击

1.1　基本原理

缓冲区溢出是指：写入缓冲区的数据量超过该缓冲区的设计大小，溢出的数据覆盖了相邻存储单元中的数据.攻击者通过溢出缓冲区将恶意代码注入内存，使进程运行时跳转并执行恶意代码来进行攻击[1].

在 Intel 80×86 体系结构中栈位于高地址区域，且向低地址方向增长，这种结构给栈缓冲区溢出攻击提供了基本条件.栈帧在内存中的分布结构如图1所示，其中，调用方 EBP(extended base pointer，栈基地址指针)存放了函数调用者的栈底地址[2]，函数的返回值和调用方 EBP 统称为与控制相关的关键数据.由于新创建的数据总是位于内存的低地址区域，当用户向低地址缓冲区写入过多数据时，就会造成

收稿日期：2009-08-07

基金项目：国家“八六三”高技术研究发展计划资助项目(2007AA01Z142)；上海申通地铁集团有限公司项目

作者简介：江建慧(1964—)，男，教授，博士生导师，工学博士，主要研究方向为可信计算、软件可靠性工程、处理器体系结构、计算机辅助设计、测试、评估.E-mail：jhjiang@tongji.edu.cn

数据溢出,从而篡改高地址内存中的关键数据,改变进程的行为.

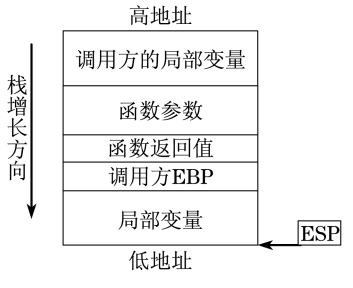

图1 内存中栈帧的结构

Fig.1 Memory layout of a stack frame

尽管实施缓冲区溢出攻击的方法多种多样,但其基本步骤可概括如下[1,4]:注入恶意数据、控制流重定向、执行恶意代码.

1.2 栈缓冲区溢出攻击模式

溢出栈缓冲区的最终目的是改变内存中关键数据,使进程跳转并执行恶意代码.以下将对C/C++语言改变内存数据的方式进行分析,并总结通过改变关键数据进行栈缓冲区溢出的攻击模式[3].

C/C++语言改变内存数据的赋值表达式可分为3类:$A=B$,$^*A=B$,$A[I]=B$.

第1类,$A=B$表达式中A和B均为变量.由于在赋值时A的地址已经确定,因此无法将A的地址替换成返回值(或调用方的EBP)所在地址,且编译器不会将返回值(或调用方的EBP)存储在临时变量中,所以无法通过此类表达式发动缓冲区溢出攻击.

第2类,$^*A=B$表达式中A为指针,B为变量.该赋值表达式将B的值写入A指向的内存地址,如下攻击模式与该赋值表达式对应.

攻击模式Ⅰ:

步骤Ⅰ1 若在指针A的低地址侧存在一个可溢出缓冲区,攻击者可利用该漏洞,用函数返回值(或调用方的EBP)的地址覆盖A.

步骤Ⅰ2 攻击者使用步骤Ⅰ1所述的方法篡改B的值,将B的值改为恶意代码的起始地址.

步骤Ⅰ3 在赋值语句$^*A=B$执行前完成步骤Ⅰ1和步骤Ⅰ2,并保证步骤Ⅰ1、步骤Ⅰ2执行完毕后,其他语句不改变A及B的值.

第3类,$A[I]=B$表达式中A为数组,I为无符号整型,B为变量.该复制语句将B的值写入数组A第I个元素所在位置.有2种攻击模式与该表达式对应.

攻击模式Ⅱ:

步骤Ⅱ1 若在B的低地址侧存在一个可溢出缓冲区,攻击者可利用该漏洞,将B的值改为恶意代码的起始地址.

步骤Ⅱ2 进程在没有进行边界检查的情况下通过循环语句将B的值循环写入数组A,每次循环将I加1,攻击者通过增加循环的次数将B值写入函数返回值(或调用方的EBP)所在内存地址.

攻击模式Ⅲ:

步骤Ⅲ1 若在B的低地址侧存在一个可溢出缓冲区,攻击者可利用该漏洞,将B的值改为恶意代码的起始地址.

步骤Ⅲ2 假设返回值(或调用方的EBP)的内存地址与$A[k]$相同,攻击者使用步骤Ⅲ1所述的方法将I的值改为k,这样攻击者就能篡改函数返回值(或调用方的EBP).

虽然C/C++中还存在很多不同类型的表达式可以改变内存数据的值,但这些不同表达式都能通过上述3种基本类型进行组合或变形来表示.例如,gets(),fgets(),strcpy(),strcat(),sprintf(),vsprintf(),sscanf(),vscanf(),fscanf(),read(),等.

2 现有动态防御方案

2.1 基于签名完整性的防御方案

基于签名完整性的检测方案在函数执行前将签名插入到关键数据与局部变量之间,并在函数调用返回时检查该签名的完整性,若签名被篡改,则说明受到了栈缓冲区溢出攻击.

下面以StackGuard[5]为例介绍该方案的基本步骤.StackGuard属于编译器优化方案.如图2所示.在函数执行前首先需将当前指令寄存器EIP(extended instruction pointer)和栈顶指针ESP(extend stack pointer)写入栈,即图2中的"返回地址"和"调用方EBP";保存完寄存器的值后,在调用方的EBP相邻的低地址侧插入"canary字".函数执行过程中,若攻击者通过溢出局部变量将恶意数据写入栈,溢出过程从低地址向高地址方向增长,直至改写函数返回值或调用方EBP的值,在此溢出过程中canary字的值会被篡改.函数返回时,StackGuard检查canary字的完整性,若canary字被修改则说明进程受到了栈缓冲区溢出攻击,检测代码发出警告信息并终止进程运行;若canary字未被篡改,说明未检测到攻击,函数正常返回.

由以上分析可知,只有在攻击者无法绕过签名修改返回值及其他关键数据时,StackGuard才能有

效防御攻击，但这种假设在某些情况下未必成立，下面根据 3 类不同攻击模式对该检测方法的安全性进行分析.

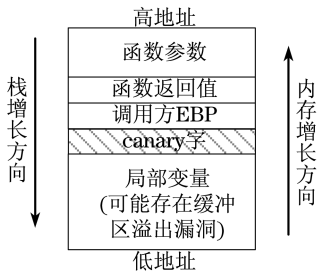

图 2　StackGuard 检测模型

Fig.2　StackGuard detection model

对于攻击模式Ⅰ，若代码中存在形如 $*A = B$ 的赋值表达式，并且攻击者可以通过溢出栈缓冲区将指针 A 重定位到函数返回值或调用方的 EBP，并将恶意代码的起始地址写入 B 对象中，这样就可以在不破坏签名完整性的情况下完成攻击. 完成这种攻击需要准确估计关键数据的线性地址.

对于攻击模式Ⅱ，在多数情况下，通过检查 canary 字的完整性能够判断是否受到模式Ⅱ的栈溢出攻击，但在以下两种情况下，该检测方法无效：

(1) 由于操作系统会根据对齐原则调整数组中对象的大小使内存出现空隙，若 canary 字恰巧位于空隙中，攻击者就能在不改变函数返回值(或调用方的 EBP)

(2) 若能够猜测出 canary 字的值，那么攻击者能够在溢出缓冲区时将正确的 canary 字写回到原有位置，从而保持 canary 字的完整性.

对于攻击模式Ⅲ，若代码中存在形如 $A[I] = B$ 的赋值表达式，且攻击者可以通过溢出栈缓冲区篡改 I 及 B 的值，那么就能通过重定位指针跳过 canary 字直接篡改返回值或调用方的 EBP.

2.2　基于随机算法的防御方案

基于随机算法的缓冲区溢出检测方案通过随机排布进程的内存分布，或是用随机生成密钥对关键数据加密来防御攻击.

一种典型的基于随机算法的防御方法是地址迷惑(address obfuscation)[6]. 该方法在编译时向目标程序插入检测代码并调整程序的内存分布，使其能够在运行时防御缓冲区溢出攻击. 如图 3 所示，其主要思想是在函数的控制数据与局部变量区域之间插入随机长度的保护空间，使函数的局部变量和关键数据保持一定“距离”. 这种方案的好处是每个函数栈帧中关键数据与局部变量被隔离，栈呈现一种稀疏的状态，使得攻击者难以猜测关键数据的地址.

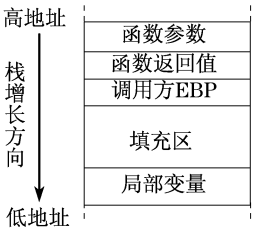

图 3　地址迷惑检测模型

Fig.3　Address obfuscation detection model

填充区的选取分为静态和动态 2 种方案. 静态方案在每次进程启动时生成一个随机数作为填充区的大小，所有函数栈帧中的填充区大小相同，这种方案的优势是性能负载较小，但是存在一定的安全风险. 动态方案在每次函数被调用时，动态生成一个随机数作为当前函数栈帧中填充区的大小，使用这种方案使得每个函数栈帧中的填充区都各不相同，增大了攻击者的猜测难度，提高了安全性，但每次动态生成会带来较大的性能负载. 为了避免过大的性能开销，地址迷惑方法选取静态方案来生成填充区.

设填充区的大小的取值空间是 $N(N \bmod 4 = 0)$，根据操作系统的自动对齐原则，变量的起始位置均为 4 的倍数，所以填充区实际取值为$\{4,8,\cdots,4i,\cdots,N\}$，攻击者单次猜测成功的概率为 $4/N$. 若每次错误猜测都将导致进程退出，由于进程重启后将再次随机生成填充区的值，那么 k 次连续猜测都是独立随机事件，所以连续 k 次猜测后成功的概率为 $p_k = 1-(1-4/N)^k$. 由 $p_k = 1-(1-4/N)^k \geqslant 50\% \Rightarrow k > 8/N$ 可知，在进行了 8/N 次攻击后，成功的概率才超出 50%. 因此，若能保证每次攻击后进程都终止运行并重新启动，地址迷惑可以获得较高的安全性. 但只要猜测方法得当，即使攻击失败，被攻击进程也不会终止运行，这就给了攻击者多次尝试的机会降低了地址迷惑的安全性[7]. 下面针对 3 种不同的攻击模式，对地址迷惑方法的安全性进行分析.

对于攻击模式Ⅰ，攻击者必须猜测函数返回值或其他关键数据的线性地址. 在加入填充区之后，由于填充区的大小是在进程启动时随机生成的，所以攻击者必须猜测填充区大小，而单次猜测成功的概率为 $4/N$. 在理想情况下，每次攻击失败都会导致进程终止，那么 k 次猜测成功的概率为$p_k = 1-(1-4/N)^k$.

对于攻击模式Ⅱ，攻击者单次猜测成功的概率也仍为 $4/N$. 但是攻击者可以通过以下 2 种方式来

增加单次猜测成功率,或是进行连续猜测:

(1) 溢出大量数据

图4中反映了内存区域为未受攻击前的正常情况和受到攻击后的内存分布(阴影部分).攻击者首先通过溢出栈上的缓冲区,在栈上写入 k 条 NOP (no operation instruction)指令,然后注入恶意代码,最后在恶意代码上方连续写入大量篡改后的地址,该篡改只要能够指向任意一条 NOP 指令,攻击者就能使进程跳转并执行恶意代码.因此攻击者的单次攻击成功率从 $4/N$ 增加到了 $4\ k/N$.

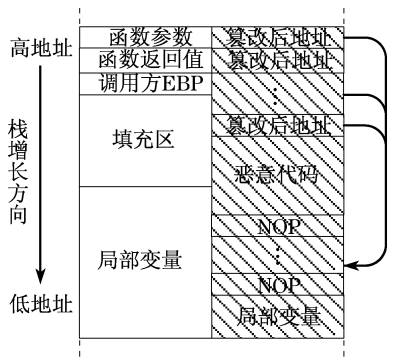

图4 溢出大量数据攻击

Fig.4 Format of an attack using a large buffer overflow

(2)连续多次猜测

当攻击者通过溢出局部变量中的数组进行攻击时,若溢出数据只修改了填充区域,并未污染函数返回值及其他关键数据,进程仍能正常返回,这就给了攻击者连续猜测的机会.进程中的一些读操作(如read()、scanf()等)很可能被多次调用,而根据地址迷惑生成填充区的方法,只要进程没有重启,这些函数被调用时栈帧中填充区的大小都是相同的.设填充区的取值范围是$\{4,8,\cdots,4i,\cdots,N\}$(其中 $N \bmod 4 = 0$),共 $N/4$ 项,攻击者只需从最小值开始,依次增大猜测值,那么含有漏洞的函数只要被运行 $N/4$ 次以上就肯定会被成功攻击,平均情况下只须 $N/8$ 就能成功实施攻击.

对于攻击模式Ⅲ,攻击利用形如"$A[I]=B$"的赋值表达式,通过篡改相对偏移量 I 及局部变量 B 的值,使 $A[I]$指向关键数据所在位置,并用 B 取代原有数据,完成该模式攻击需要准确估计数组 A 与关键数据的相对位移.与攻击模式Ⅱ类似,单次准确猜测的概率也仍是 $4/N$,但是攻击者可以通过逐步扩大相对偏移量 I,发动多次攻击来提高攻击成功率,并在 $N/4$ 次尝试后成功实施攻击.

由以上分析可知,由于攻击者通过攻击模式Ⅰ重新定位指针很容易导致进程终止并重启,因此地址迷惑方法能够以较高概率防御攻击模式Ⅰ.但对于攻击模式Ⅱ及攻击模式Ⅲ,攻击者可通过一次溢出大量数据,并注入 k 条 NOP 指令将单次猜测成功率增加到 $4k/N$.而且若猜测方法得当,可在 $N/4$ 次连续猜测后实施攻击.

2.3 其他动态防御模型

现有的动态防御方案可以大致分为3类:编译期优化法案、扩展 C/C++ 语言、硬件优化.

编译期优化方案的主要思想是在编译阶段插入漏洞检测代码,并在程序运行期间进行动态检测[8].它可细分为基于签名完整性防御的方案,包括 StackGuard[5],ProPolice[9]等;基于关键数据保护栈的防御方案,包括 RAD[10],StackShield[11];基于随机算法的防御方案,包括地址迷惑[6],PointGuard[12]等.

对 C/C++ 语言进行优化,提供安全的 C/C++ 数据类型和库函数.这类方案包括 Cyclone[13],CCured[14-15],Libsafe 和 LibVerified[16]等.

硬件优化技术通过硬件来实现上述检测方案,硬件实现方式能大大节约程序的运行时间,并且对于用户是完全透明的,这类方法包括 SmashGuard[17],SecureBit[18]等.

3 基于 k 循环随机序列的防御方案

针对使用地址迷惑的缺陷,本文结合基于随机算法与签名完整性的防御思想提出了一种改进方案.该方案通过在编译期间向目标程序插入安全防御代码,使其能够在运行时动态检测并防御缓冲区溢出攻击.新方案能在极大概率下防御多种攻击模式,消除或削弱攻击者连续猜测带来的安全隐患,并且不造成过大的运行时负载.

新方案在采用地址迷惑防御思想的同时,将 StackGuard 作为可选方案,通过使用 k 随机循环序列,在函数栈帧之间插入随机保护区域,使攻击者即使能够连续多次猜测也只能在极小概率下成功实施攻击,同时还使被攻击进程具备一定的容侵能力.

图5给出了使用改进方案后栈内存的的分布情况.与地址迷惑不同,改进方案首先定义了一个长度为 k 的全局整形数组 padding_size[k],在进程启动时动态生成 k 个各不相同的正整数,每个均为4的倍数,对 padding_size[k]进行填充.当函数被调用时,循环从该全局数组中取值作为填充区的大小.例

如，当函数 A 被调用时，将填充区 1 的大小设置为 S_1，若函数 A 调用函数 B，那么函数 B 的栈帧将 S_2 作为填充区 2 的大小，若函数 A 嵌套调用 B 那么，函数 B 的栈帧将紧接着函数 A 栈帧分配.

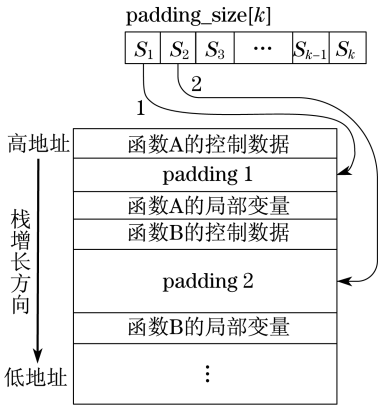

图 5　基于 k 循环随机序列防御的栈结构

Fig.5　Stack layout for prevention based on k circular random sequence

如图 6 所示，若函数 B 调用完毕后退栈，A 函数又嵌套调用了函数 C，那么继续取用 S_3 作为填充区 3 的大小，以此类推. 在 k 个函数被调用后，即图 6 中函数 k 调用完成之后，padding_size[k]数组中的所有数值都已被使用，下一个函数被调用时循环使用 padding_size[k]数组中的值，即函数 $k+1$ 次循环使用 S_1 作为其填充区的大小.

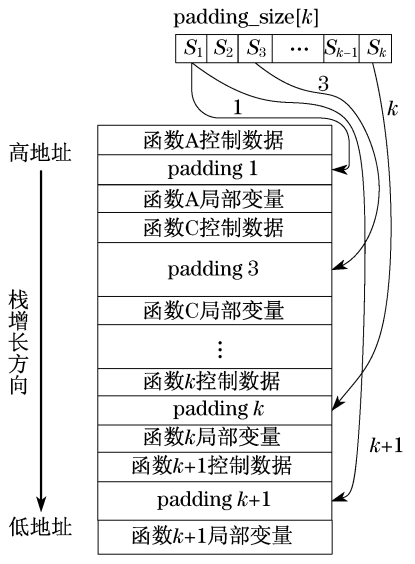

图 6　$k+1$ 次函数调用后的栈结构

Fig.6　Stack layout after $k+1$ function call

为了弥补地址迷惑对于攻击模式Ⅱ可能被一次溢出大量数据的缺陷，改进方案引入基于签名完整性的检测防御方法，但是考虑到程序的执行效率仅将签名检测作为可选方案，在安全要求较高的情况下，可以选用签名检测. 图 7 显示了增加签名后一个函数栈帧的分布，该方案首先在调用方 EBP 的低地址侧插入 canary 字，并在 canary 字与局部变量之间插入随机大小的填充区. 通过加入 canary 字，即使攻击者采用攻击模式Ⅱ一次溢出大量数据，改进后的方案也能通过在函数返回时检查 canary 字的完整性来判断是否受到攻击，若 canary 被修改，就终止进程运行.

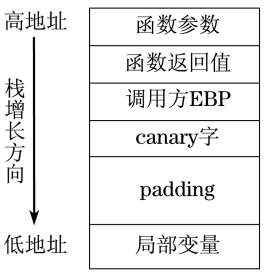

图 7　基于 k 循环随机序列和 canary 字的栈帧结构

Fig.7　Stack frame layout based on k circular random sequence and canary word

由以上分析可知，改进方案采用 k 个循环随机序列作为填充区大小，其好处是改进方案在 k 步内函数栈帧上插入填充区的大小均不相同，这样依照进程执行路径的情况的不同，当调用了 n 个函数后，填充区的组合情况有 kn 种不同的情况，而地址迷惑方法中无论函数调用序列如何变化都只有一种组合方式，更多的组合方式增加了攻击者成功猜测的难度. 其次，考虑到每次函数调用时随机的生成填充区的大小会极大降低性能，所以改进方案在进程启动时随机生成 k 个不同的随机整形变量，并将其存放在全局整形数组 padding_size[k]中，为防止攻击者修改，可以将这些数据设置为只读（因为每个进程值会执行一遍该过程，所以不会大幅度降低性能），这样就避免了在每次函数调用时动态生成填充区的大小，提高了进程的运行效率.

攻击者通过溢出栈缓冲区发动攻击的一个前提是充分分析栈上数据的分布，并能准确计算出函数返回值、调用方的 EBP 等关键数据所在的位置. 通过在函数栈帧的控制数据和局部变量之间插入大小不同随机保护区域，改进方案进一步增加了猜测难度. 首先，由于当前函数的控制数据和临时数据中存在随机保护区域，所以攻击者无法准确计算控制数据与被溢出缓冲区之间的相对偏移. 其次，即使进程栈的起始地址相对固定，且攻击者通过程序分析能得到进程执行路径，由于栈底到当前函数栈帧之间

插入了若干了随机保护区域,所以攻击者将无法根据栈的起始地址及调用关系来计算当前栈帧中关键数据的地址.最后,新方案结合了基于签名完整性的防御方案,通过增加 canary 字进一步降低攻击者单次猜测及连续攻击的成功率.

4 评估

本节将分析改进方案针对3种基本攻击模式的有效性.由于改进方案主要通过 k 个循环随机序列来防御栈缓冲区溢出攻击,所以分析方法主要是概率证明.

在验证该方案的有效性后,再进一步分析改进方案如何通过在函数栈帧中插入随机保护区域来达到一定程度的容侵,最后使用 micro-bench 对改进方案的性能进行测试.

4.1 有效性分析

假设填充区的取值范围为$\{4,8,\cdots,4i,\cdots,N\}$(其中 $N \bmod 4=0$),共 $N/4$ 项.每次从 $4/N$ 项中随机的取出各不相同的k 项放入 padding_size[k]数组中,$k \leqslant 4/N$.攻击者单次成功猜中填充区大小的概率为 $4/N$.

4.1.1 攻击模式Ⅰ的有效性分析

通过攻击模式Ⅰ发动攻击需要准确估计关键数据的绝对地址.如图6所示,在改进方案中,要猜测当前函数与控制相关的数据(图中的函数 $k+1$ 控制数据)必须猜测之前所有填充区之和.假设攻击者通过分析程序得出了进程的执行路径和函数栈帧的分布结构,攻击者还需猜测填充区的大小,设 $L_i(1 \leqslant i \leqslant k)$ 表示大小为 S_i 的填充区个数,那么当前栈上填充区大小的总和 $S_{\text{padding}}=\sum_{i=1}^{k} L_i S_i$.设 $L=\sum_{i=1}^{k} L_i$,从攻击者角度来看,每个栈帧中填充区的大小可能是$\{4,8,\cdots,4i,\cdots,N\}$中的任意一个,因此 S_{padding} 共有 $LN/4$ 种不同的取值,攻击者单次猜测成功的概率为 $4/LN$.当进程栈上的函数嵌套深度 L 达到一定数量之后,攻击者单次猜测成功的概率大大降低.即使在最坏情况下,当前进程栈上只有一个函数栈帧,攻击者的单次猜测成功率为 $4/N$,与地址迷惑相同,实际应用中只存在一层函数调用的情况非常少.

4.1.2 攻击模式Ⅱ的有效性分析

通过对地址迷惑的分析可知,在攻击失败未导致进程退出的情况下,攻击者只要从小到大逐次猜测填充区就能在尝试 $N/4$ 次后成功完成攻击.如果每次函数调用时都动态生成随机数作为当前填充区的大小,那么攻击者每次猜测都是独立的,连续猜测 m 次成功的概率为 $P_{\text{m}}=1-(1-4/N)^m$.下面证明采用 k 循环随机序列方案在连续猜测攻击情况下的防御成功率.

首先讨论未选用签名完整性检查方案的情况,设攻击者连续猜测 m 次,当 $m \leqslant k$ 时(k 为 padding_size 的大小),那么 m 次猜测都是独立的,所以攻击者连续 m 次猜测成功的概率为 $P_m=1-(1-4/N)^m$.当 $k<m \leqslant 2k$ 时,由于攻击者已经进行了 k 次失败的尝试,因此在新一轮猜测中,至少可以排除一个错误答案,因此攻击者单次猜测成功的概率变为 $4/N-4$,那么 m 次猜测后成功的概率为 $P_m=1-(1-4/N)^k(1-4/N-4)^{m-k}$,$k<m \leqslant 2k$.由归纳法可知,在最坏情况下,即每次攻击失败的攻击都未导致程序终止运行,连续猜测 m 次后成功的概率为 $P_m=1-\prod_{i=0}^{m \bmod k}[1-4/(N-4i)]^{m-ik}$.相比地址迷惑,改进方案使得攻击者无法在连续被攻击 $N/4$ 次后成功猜测到填充区的大小;相比每次在函数调用时动态生成随机大小的方法,改进方案在前 k 次猜测攻击中,成功防御的概率与动态生成填充区完全相同,当连续猜测次数 $m>k$ 时,每 k 次猜测只能排除一种可能,攻击者的成功猜测的概率上升极为缓慢,因此改进方案能够较好地防御连续猜测攻击.

对于攻击者通过大量溢出数据,并在恶意代码前加入若干条 NOP 指令来增加单词猜测成功率的攻击,改进方案可以通过引入 canary 字来进一步提高安全性.canary 字在进程启动时随机生成,假设 canary 字的取值范围为$\{0,1,2,3,\cdots,N_1-1\}$ 共 N_1 种,攻击者插入了 N_2 条 NOP 指令来提高攻击成功率,那么单次成功猜测的概率为 $4N_2/NN_1$,通常情况下 N_1 为16位随机数,即有 2^{16} 种不同取值,远大于攻击可能插入的 NOP 数目,因此,通过引入 canary 可以进一步降低单词猜测成功率.

在引入签名完整性检查后,由于攻击者只有在同时猜中 canary 字及填充区大小的情况下才可能成功实施攻击,那么攻击者连续猜测 m 次成功猜中的概率为 $P_{\text{m}}=1-\prod_{i=0}^{m \bmod k}[1-4/(NN_1-4i)]^{m-ik}$(不考虑 NOP 指令的影响).由此可见,即使能够反复尝

试，攻击者也只有在极小的概率下才能成功发动一次攻击.

4.1.3　攻击模式Ⅲ的有效性分析

在攻击模式Ⅲ中，攻击者利用代码中存在形如“$A[i]=B$”的赋值表达式，通过篡改相对偏移 i 及 B 的值，使 $A[i]$指向关键数据所在位置，并用 B 中数据取代原有数据，完成该模式攻击需要准确估计数组 A 与关键数据的相对位移. 与攻击模式Ⅱ类似，攻击者可以逐步调整 i 的偏移量，从小到大依次猜测填充区的大小.

通过对攻击模式Ⅱ的分析可知，采用 k 个循环随机序列作为填充区大小后，在最坏情况下，即每次攻击失败程序都未终止运行，连续猜测 m 次后，成功实施攻击的概率为 $P_{\mathrm{m}}=1-\prod_{i=0}^{m \bmod k}[1-4/(N-4i)]^{m-ik}$.

4.1.4　容侵性

在 StackGuard 方案中，即使攻击者溢出局部变量区 1 个字节也会破坏 canary 字，并导致进程被迫终止. 在现实中，程序使用者往往会由于疏忽，将过多的数据复制到缓冲区中，在这种情况下终止进程运行会降低程序的可用性. 客观上，地址迷惑具备一定容侵性，只要攻击者没有篡改关键数据，进程一般不会终止运行，但地址迷惑能够以高概率防御攻击的前提是每次攻击失败后进程都会终止运行，否则攻击者能再多次尝试后猜测出 padding 值并实施攻击. 因此，地址迷惑的设计与容侵思想相悖，其容侵能力反而成为被攻击者利用的漏洞.

本文提出的改进方案通过使用 k 循环随机序列，在关键数据和局部变量之间插入随机保护区域，从而有效地解决了上述问题. 通过在一定范围内变化填充区大小，改进方案保证了即使攻击者能够进行连续猜测，也只有在极小概率下才能成功实施攻击，这就给容侵提供了基本条件. 当用户将过多数据复制到栈上并造成溢出时，随机保护就起到了缓冲的作用，即使用户的溢出行为是有意识的攻击，保护区域也能一定程度消除这种攻击带来的影响. 只要攻击者未篡改函数返回值、调用方的 EBP 等关键数据，进程返回后仍能保证正确的执行路径，这给了函数调用者处理函数异常的机会，若调用者设置了有效的异常处理机制，那么进程就不需要因这类错误而终止运行，这在一定程度上提高了软件的可用性. 若攻击者以攻击模式Ⅱ的方式复制了大量恶意数据，并超出了保护区域的大小，改进方案也能通过检测 canary 字的完整性及终止被污染进程来保证系统的安全性.

4.2　性能分析

基于 k 循环随机序列的检测方案在每次函数调用时都需要根据 padding_size[k]中的数据动态插入填充区；若选用了签名完整性检查，在函数运行前必须插入 canary 字，并在函数返回时检查其完整性，这些动态检测势必会给函数调用带来一定性能开销. 为了测试改进方案对一次函数调用带来的性能开销，本文采用文献[5－6，8]的测试方法，测试分为 4 个部分：

(1) i ++

在 main 函数中执行对全局变量 i 执行 500 000 000次 i ++ 操作.

(2) void inc ()

在 void inc()中对全局变量进行 i ++ 操作，并在 main 函数中对 void inc()函数进行 500 000 000 次循环调用.

(3) void inc (int*)

在 void inc (int * p) 中对 int* 类型参数 p 进行 p ++ 操作，并在 main 函数中对 void inc (int * p)函数进行 500 000 000 次循环调用.

(4) int inc (int)

在 void inc (int value)中返回 value＋1，并在 main 函数中对 void inc (int value)函数进行500 000 000次循环调用.

表 1 为各防御方法相对于未采取防御措施时的函数调用性能开销.

表 1　函数调用性能开销

Tab.1　Performance overhead of function calls %

测试函数	静态生成法	基于 k 循环随机序列	基于 k 循环随机序列(选用签名)
i ++	无性能损失	无性能损失	无性能损失
void inc()	108	151	185
void inc(int*)	117	158	174
int inc(int)	110	152	184

由表 1 中的数据可知，采用静态方案生成填充区大小时，单次函数调用开销为 108%～117%. 在未选用签名完整性检查情况，采用 k 循环随机序列的性能开销为 151%～158%. 若选用签名检查，单次调用函数的性能开销在 174%～185%之间. 由此可见，采用 k 循环随机序列动态生成填充区方法在提高安全性的同时，并未造成过大性能开销.

5 结论

本文面向 Intel 80×86 体系结构和 C/C++ 语言,在分析现有动态缓冲区溢出防御方案特点的基础上,结合基于随机算法与签名完整性的防御思想,提出了基于 k 循环随机序列的栈缓冲区溢出防御方案.该方案在编译期间向目标程序插入安全防御代码,使程序能够在运行时动态检测并防御缓冲区溢出攻击.通过有效性分析可知,该方案能够在极大概率下防御多种模式的栈缓冲区溢出攻击,有效解决了溢出大量数据攻击和连续猜测攻击问题,并使软件具备一定容侵能力.性能测试结果表明改进方案不会造成过大的运行时负载.

参考文献:

[1] Forst J C, Osipov V, Bhalla N, et al. Buffer overflow attacks: detect, exploit, prevent[M]. Rockland: Syngress Press, 2005.

[2] Bovet D P, Cesati M. 深入理解 Linux 内核[M]. 3 版. 陈莉君, 张琼声, 张宏伟, 译. 北京: 中国电力出版社, 2007.
Bovet D P, Cesati M. Understanding the Linux kernel[M]. 3rd ed. Translated by CHEN Lijun, ZHANG Qiongsheng, ZHANG Hongwei. Beijing: China Power Press, 2007.

[3] AlephOne. Smashing stack for fun and profit[EB/OL]. (1996-11-08)[2009-06-15]. http://phrack.com/issues.html?issue=49&id=14#article.

[4] Wagner D, Foster J, Brewer E, et al. A first step towards automated detection of buffer overrun vulnerabilities[C]// Proceedings of the Network and Distributed Systems Security Symposium. San Diego: Internet Society, 2000: 1-14.

[5] Cowan C, Pu C, Maier D, et al. StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks[C]// Proceedings of the 7th conference on USENIX Security Symposium. San Antonio: USENIX Association, 1998: 63-78.

[6] Bhatkar S, DuVarney D C, Sekar R. Address obfuscation: an efficient approach to combat a board range of memory error exploits[C]// Proceedings of the 12th Conference on USENIX Security Symposium. Washington D C: USENIX Association, 2003: 8-23.

[7] Strackx R, Younan Y, Philippaerts P, et al. Breaking the memory secrecy assumption[C]// Proceedings of the Second European Workshop on System Security. Nuremburg: ACM New York Press, 2009: 1-8.

[8] Pozza D, Sisto R. A lightweight security analyzer inside GCC[C]// Proceedings of IEEE Third International Conference on Availability, Reliability and Security. Barcelona: IEEE Computer Society, 2008: 851-858.

[9] Etoh H. ProPolice: GCC extension for protecting applications from stack-smashing attacks[EB/OL]. (2005-08-22)[2009-06-15]. http://www.trl.ibm.com/projects/security/ssp/.

[10] Chiueh T, Hsu F H. RAD: a compile-time solution to buffer overflow attacks[C]// Proceedings of the International Conference on Distributed Computing Systems. Phoenix: IEEE Computer Society, 2001: 409-417.

[11] Vendicator. Stack Shield: a stack smashing technique protection tool for Linux[EB/OL]. (2000-01-08)[2009-06-15]. http://www.angelfire.com/sk/stackshield/.

[12] Cowan C, Beattie S, Johansen J, et al. PointGuard TM: protecting pointers from buffer overflow vulnerabilities[C]// Proceedings 12th USENIX Security Symposium. Washington D C: USENIX Association, 2003: 359-389.

[13] Jim T, Morrisett G, Grossman D, et al. Cyclone: a safe dialect of C[C]// Proceedings of the USENIX Annual Technical Conference. Monterey: USENIX Society, 2002: 275-288.

[14] Necula G C, Condit J, Harren M, et al. CCured: type-safe retrofitting of legacy software[J]. ACM Transactions on Programming Languages and Systems, 2005, 27(3): 477.

[15] Condit J, Harren M, McPeak S, et al. CCured in the real world[C]// Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation. San Diego: ACM Broadway Press, 2003: 232-244.

[16] Baratloo A, Singh N, Tsai T. Transparent run-time defense against stack smashing attacks[C]// Proceedings of the USENIX Technical Conference. San Diego: USENIX Association, 2000: 251-262.

[17] Vijaykumar T N, Brodley C E, Kuperman B A, et al. SmashGuard: a hardware solution to prevent security attacks on the function return address[R]. West Lafayette: Purdue University, 2003.

[18] Piromsopa K, Enbody R J. Secure Bit: transparent hardware buffer-overflow protection[J]. IEEE Transactions on Dependable and Secure Computing, 2006, 3(4): 365.