ISSN 1009-3044
Computer Knowledge and Technology 电脑知识与技术
Vol.7, No.1, January 2011,pp.90-93
E-mail: info@cccc.net.cn
http://www.dnzs.net.cn
Tel:+86-551-5690963 5690964

基于 Linux 地址空间随机化的缓冲区溢出研究

俞晓鸿,陆瑶

(同济大学,上海 201804)

摘要:缓冲区溢出攻击是一种被广泛利用并危害严重的攻击方式,已经成为计算机系统安全亟待解决的重要问题。在分析缓冲区溢出攻击原理的基础上,给出了主要的防御方法,重点探讨了 Linux 系统下地址空间随机化防御方法的实现,研究了如何绕过其防护的缓冲区溢出攻击方法,并对其防护效果进行了分析。结果表明,地址空间随机化技术能有效降低缓冲区溢出攻击成功的概率,但不能完全阻止攻击。

关键词:缓冲区溢出;Linux;地址空间随机化;攻击;防御

中图分类号:TP393 **文献标识码**:A **文章编号**:1009-3044(2011)01-0090-04

Buffer Overflow Research based on Address Space Randomization in Linux

YU Xiao-hong, LU-Yao

(Tongji University, Shanghai 201804, China)

Abstract: Buffer overflow attack has been widely used and could cause great damage. It has become an important issue of computer system security. This paper analyzes the principle of buffer overflow attack, and presents common methods for defending the attack. Then the implementation of defense method for address space randomization in Linux system is mainly discussed, and buffer overflow attack methods for bypassing the defense are proposed. Finally, the effectiveness evaluation of the defense is also given. The results show that address space randomization technique can effectively reduce the probability of successful buffer overflow attacks, but it can not prevent attacks completely.

Key words: buffer overflow; Linux; address space randomization; attack; defense

随着信息技术的高速发展,利用广泛开放的网络环境进行通信已成为时代发展的趋势。网络在提供开放和共享资源的同时,也不可避免地存在着安全风险。由于计算机网络系统自身存在漏洞,这些漏洞一旦被攻击者利用,就可能导致计算机系统的数据和资源遭到破坏。缓冲区溢出漏洞是最常见的漏洞。在 SANS 选出的威胁最大的 20 个漏洞中,有 10 个是由于缓冲区溢出所造成的[1]。对缓冲区溢出漏洞进行攻击,将会导致程序运行失败、系统崩溃以及执行任意代码等后果,对系统安全构成极大威胁。远程攻击者能利用这种攻击获取系统控制权,进而执行各种非法操作,窃取秘密信息或以该主机为跳板攻击其他主机。据统计 80%以上成功获取非法权限的攻击都是利用了缓冲区溢出漏洞[2]。因此研究缓冲区溢出的攻击和防御机制对计算机网络安全尤为重要。

本文介绍了缓冲区溢出的基本原理及主要的防御方法,其中重点分析了地址空间随机化技术在 Linux 系统中的实现,并针对这一防御技术,研究如何绕开其防护的缓冲区溢出攻击方法,并作了相关的理论分析和实际测试。

1 缓冲区溢出

1.1 缓冲区溢出攻击原理

缓冲区溢出是指:写入缓冲区的数据量超过缓冲区的设计大小,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[3]。缓冲区溢出攻击主要分为以下几类:栈溢出、堆溢出、格式化字符串溢出和整型变量溢出等,其中以栈溢出最为常见。

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。栈帧在内存中的分布结构如图 1 所示,其中,调用方 EBP(栈基地址指针)存放了函数调用方的栈底地址,函数返回值存放了函数结束后的返回地址。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,就会造成数据溢出,从而篡改高地址内存中的关键数据。如果写入的数据是攻击者精心构造的恶意代码 shellcode,返回地址等指针被覆盖成了指向 shellcode 的指针,当调用该指针时,就将跳转并执行 shellcode,从而改变进程的行为,达到入侵的目的。

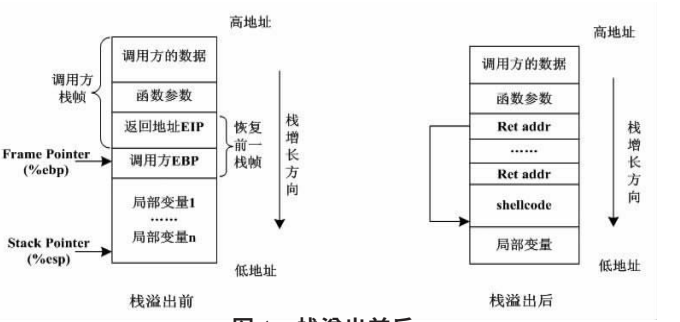

图 1 栈溢出前后

1.2 缓冲区溢出攻击防御

针对缓冲区溢出攻击的防御方法归纳起来有:编译保护、安全库函数、缓冲区不可执行和随机化方法等。

编译保护是通过扩展编译器以防止函数的返回地址被修改来实现栈保护。通常是在返回地址前放置一个标志字或建立一个影

收稿日期:2010-10-23
作者简介:俞晓鸿(1986-),女,福建宁德人,同济大学,硕士研究生,主要研究方向为信息安全。

子栈,在函数返回前查看是否有溢出存在。目前已有不少的补丁可供使用,如 StackGuard 和 StackShield。但这种方法只局限于对栈的保护。

C 标准库中的函数一般不进行参数验证,通过重写这些函数使其进行参数验证,增加一些安全检查机制可以有效阻止缓冲区溢出攻击。如 Libsafe 对常见的字符串操作函数进行封装[4],在调用这些函数时先进行边界检查,一旦目标缓冲区不能容纳源字符串则终止程序。针对 Libsafe 只保护栈的不足,Avi-jit 等提出 LibsafePlus[5],通过抽取程序的调试信息,实现对全局变量、栈变量和堆变量的边界检查。

缓冲区不可执行包括栈不可执行和数据段不可执行。攻击者通常是将恶意代码植入溢出的缓冲区中,只要让这些可写的缓冲区不可执行,就能防止恶意代码控制程序。采用这种技术的典型例子有 Linux 内核补丁 PaX(Page Exec)[6]。它通过不可执行页实现了不可执行栈和不可执行堆。不可执行缓冲区对于需要注入可执行代码这类攻击的防范是高效的,但无法阻止像 return-into-libc 这种无须注入代码的攻击。

现有的用于防范针对内存攻击的技术或者系统只能防御某一特定方式的攻击, 而且要将这些技术或者系统集成在一起很困难。操作系统的安全防护机制对缓冲区溢出漏洞利用有很大影响。如果能在操作系统层检测和阻止缓冲区溢出攻击,将大大提升系统的安全性。因此研究人员提出了随机化方法对系统进行保护,主要有指令随机化和地址空间随机化。

指令随机化是指编译时对可执行程序的机器码进行加密,在指令执行前,对每条加密的指令进行解码。这个过程使得攻击者植入的恶意代码不能完成预定的功能,通常会导致程序崩溃。但这种方法存在因使用虚拟机导致负荷过重等问题[7]。

地址空间随机化 ASR(Address Space Randomization)技术是近几年发展起来的一项比较有效的防御缓冲区溢出攻击的技术,由于它的独到之处,Linux、FreeBSD 和 Windows Vista 等主流操作系统都已采用了该技术。

2 Linux 地址空间随机化技术的实现

基于缓冲区溢出漏洞的攻击需要事先熟悉进程的地址空间,从而将程序的执行流程跳转到攻击代码的位置。ASR 技术通过对操作系统内核或 C 库的修改,使进程加载到内存的地址随机化,使得攻击者无法将程序的执行流程跳转到预期位置,从而阻止进程执行攻击者预先设置好的攻击代码。攻击者要想控制系统,需要修改返回地址或指定地址为指定值。地址随机化后,这些指定地址或者指定值都将无法事先确定,攻击者只能猜测该地址,所以不能保证能正确地跳转到攻击代码的位置,从而有效地降低攻击者攻击成功的概率。

2.1 Linux 可执行目标文件的加载过程

在 Linux 系统中可以通过 execve 系统调用启动加载器。加载器为新进程创建一组新的代码、数据、堆和栈段。新的堆和栈段被初始化为零。通过将虚拟地址空间中的页映射到可执行文件的页大小的组块(chunks),新的代码和数据段被初始化为可执行文件的内容。最后,加载器跳转到 _start 地址,它最终会调用应用的 main 函数。除了一些头部信息,在加载过程中没有任何从磁盘到存储器的数据拷贝,直到 CPU 引用一个被映射的虚拟页时,操作系统才会利用页面调度机制将页面从磁盘传送到存储器。

当加载器运行时,它创建的存储器映像如图 2 所示。在 Linux 系统中,代码段总是从 0x08048000 处开始。数据段是在接下来的下一个 4KB 对齐的地址处。运行时堆在接下来的读/写段之后的第一个 4KB 对齐的地址处,并通过 malloc 调用向高地址增长。开始于地址 0x40000000 处的段为共享库保留。用户栈从地址 0xbfffffff 处开始,并向低地址增长。从栈的上部开始于地址 0xc0000000 处的段是为内核的代码和数据保留的。

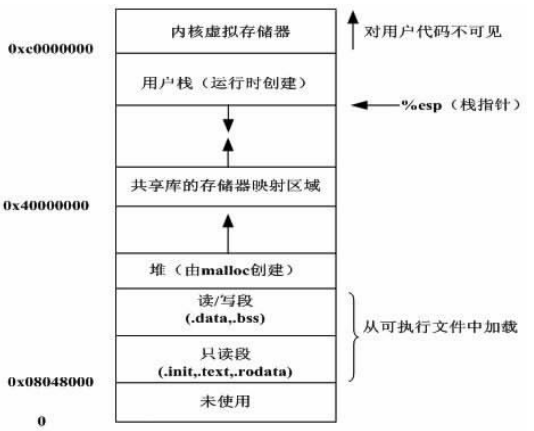

图 1 栈溢出前后

2.2 Linux 中的地址随机化

自从 Linux 2.6.12 版本后, 地址随机化技术已经默认在 Linux 内核中启用。在 Linux 2.6.27 下的测试结果如图 3 所示。

```
yuxh@yuxh-desktop:~/test$ cat /proc/self/maps|egrep '(libc|heap|stack)'
    09a2c000-09a4d000 rw-p 09a2c000 00:00 0          [heap]
    b7df1000-b7f49000 r-xp 00000000 08:01 409505     /lib/tls/i686/cmov/libc-2.8.90.so
    b7f49000-b7f4b000 r--p 00158000 08:01 409505     /lib/tls/i686/cmov/libc-2.8.90.so
    b7f4b000-b7f4c000 rw-p 0015a000 08:01 409505     /lib/tls/i686/cmov/libc-2.8.90.so
    bff6d000-bff82000 rw-p bffeb000 00:00 0          [stack]
yuxh@yuxh-desktop:~/test$ cat /proc/self/maps|egrep '(libc|heap|stack)'
    09b25000-09b46000 rw-p 09b25000 00:00 0          [heap]
    b7f15000-b806d000 r-xp 00000000 08:01 409505     /lib/tls/i686/cmov/libc-2.8.90.so
    b806d000-b806f000 r--p 00158000 08:01 409505     /lib/tls/i686/cmov/libc-2.8.90.so
    b806f000-b8070000 rw-p 0015a000 08:01 409505     /lib/tls/i686/cmov/libc-2.8.90.so
    bf990000-bf9a5000 rw-p bffeb000 00:00 0          [stack]
```

图 3 /proc/self/maps

从图 3 中可以看出,栈、堆和共享库的加载地址是随机的。通过比较多次运行的结果可以判断,对于 32 位系统,栈基地址的第 4-23 位共 20 位是随机的,堆基地址的第 12-27 共 16 位是随机的,共享库基地址的第 12-27 位共 16 位是随机的。

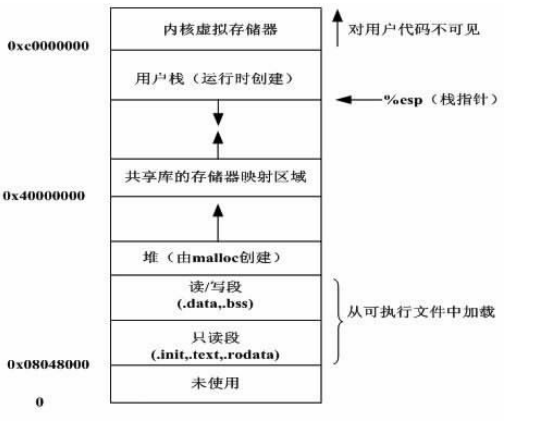

图 2 Linux 运行时存储器映像

PaX 是较成熟的 ASR 技术,它通过为 Linux 内核打补丁,在进程加载时,对栈基地址的 4-27 位共 24 位进行随机化,对包括主

程序映象、静态数据区、堆这一连续区域的基地址的 12–27 位共 16 位进行随机化,对共享库加载地址的 12–27 位共 16 位进行随机化。再加上可写页不可执行技术,PaX 能大大降低攻击成功的概率。TRR 通过修改进程加载器来实现 ASR,而无需修改系统内核。与 PaX 相比,TRR 增加了对全局偏移表 GOT 的随机化,在进程加载时将 GOT 移动到新的位置,并用二进制代码修改工具修改代码段中的过程链接表 PLT,以阻止修改 GOT 函数指针指向恶意代码的攻击。这一类 ASR 实现方案对栈、堆、主程序代码段以及共享库等在进程空间的基地址进行了随机化,粒度较粗、实现简单、额外开销较小,目前得到了广泛的应用。它们的缺陷主要在于只随机化了基地址以及使用的随机空间不大,可以被暴力破解和实施相对地址攻击。

2.3 栈随机化的实现

加载 ELF 可执行文件的函数是 load_elf_binary()[8],它把 page 指针数组指向的内核页面以及可执行文件、解释器的部分区段映射到用户空间,并设置用户空间栈上的 argc、argv、envp 变量和解释器将用到的辅助向量,它调用的 setup_arg_pages()函数形式为 setup_arg_pages (bprm, randomize_stack_top (STACK_TOP), executable_stack); 其中,STACK_TOP 为用户空间的顶端, 一般等于 0xc0000000。 randomize_stack_top()的主要代码实现如下:

```
    if ((current->flags & PF_RANDOMIZE) &&
       ! (current->personality & ADDR_NO_RANDOMIZE)) {
        random_variable = get_random_int() & STACK_RND_MASK;
        random_variable <<= PAGE_SHIFT;
    }
#ifdef CONFIG_STACK_GROWSUP
        return PAGE_ALIGN(stack_top) + random_variable;
#else
        return PAGE_ALIGN(stack_top) - random_variable;
#endif
```

首先判断内核是否开启了 ASR 保护,如果开启则调用 get_random_int()获得一个随机数,并和 STACK_RND_MASK(0x7ff)相与后左移 PAGE_SHIFT(12)得到 random_variable,最后将 stack_top 按页边界对齐后减去 random_variable,得到最终的 stack_top 值。由此可以推出 stack_top 可能的最小值为 0xc0000000–0x7ff000=0xbf801000。

stack_base 的值是按以下方式确定的:

```
stack_base = arch_align_stack(stack_top - MAX_ARG_PAGES*PAGE_SIZE);
stack_base = PAGE_ALIGN(stack_base);
```

而 arch_align_stack()的主要实现如下:

```
if (! (current->personality & ADDR_NO_RANDOMIZE) && randomize_va_space)
        sp -= get_random_int() % 8192;
    return sp & ~0xf;
```

其中,MAX_ARG_PAGES 和 PAGE_SIZE 的值分别为 32 和 4096,即参数的总长度不得超过 32 个页面。通过将 sp 指针减去一个随机数除 8192 的余数后,末尾四位取 0,再进行 PAGE_ALIGN,得到最终的 stack_base 值。最后 setup_arg_pages 函数将 page 指针数组指向的内核页面映射到用户空间中 stack_base 开始的区域。

上述分析可以得出:1) 栈地址的第 4–23 位是随机的,页内偏移(0–11 位)也是随机的。2) 当向程序传递相同的环境变量时,即使 stack_base 不是固定值,但环境变量字串在页内的偏移是一个固定值。3) 因为 stack_base 的最小值不会小于 0xbf7df000,所以环境变量字串的地址总是高于 0xbf7df000。

3 针对 ASR 防护的缓冲区溢出攻击方法

3.1 暴力破解

由于程序加载的位置不再固定,因此使用缓冲区溢出攻击程序成功的概率将大大降低。但由于使用的随机空间不大,通过一定次数的暴力破解后仍有可能成功。此时栈溢出攻击成功的概率计算如下。其中,L(v)表示 v 的长度。

1) shellcode 在栈内变量 buf 中

p=(L(buf)–L(shellcode))/(L(random_stack_range)–L(shellcode)),

若 L(buf)=256,L(shellcode)=25,L(random_stack–range)=1M,则 p=0.02%。

2) shellcode 在环境变量中

环境变量地址头 8 位是固定值 bf,末 12 位通过 getenv()函数再微调后获得,中间 12 位的范围是 7df~fff,这是一个不大的区间。

p=1/(L(random_env_range)–L(shellcode))),

若 L(random_env_range)=432,L(shellcode)=25,则 p=0.24%。

3.2 转向非随机化区域

Linux 操作系统默认采用了 ASR 机制后,它的栈,堆,共享库的加载地址是随机的,但是仍有些区域是非随机的,如 text 段、data 段和 bss 段。因此可以通过覆盖栈中的返回地址使程序跳转到这些地方。

text 段存放着程序代码,它是只读的,任何向其写入的操作都会被禁止,因此无法将 shellcode 放在其中。但可以用指向 text 段的另一个合法地址的指针覆盖返回地址,从而控制程序流程。bss 段中存放的是未初始化的全局变量和静态变量。data 段存放的是已初始化的全局变量和静态变量。如果程序中存在 strcpy(localbuf, input); 及 strcpy(globalbuf, localbuf)语句,那么输入字符串 input 会覆盖局部变量 localbuf,并最终覆盖全局变量 globalbuf,又因为 globalbuf 的地址可以通过 gdb 调试确定,所以通过构造如图 4 所示的输

入串就会控制程序执行 shellcode 代码。

shellcode	\x41\x41\x41\x41..	ret (&globalbuf)

图 4 攻击代码字符串

3.3 return-into-libc

当设定栈不可执行后,shellcode 无论放在栈的什么地方都基本无用,因此这种防御方法可以阻止大部分的攻击。在 Linux 中,可以通过 PaX 设定栈不可执行,而 return-into-libc 方法可以避开这样的防护对策。return-into-libc 主要从两方面突破 PaX 的防护:1) 通过伪造栈帧和利用连续的系统调用和库函数调用来构造溢出环境,这是因为系统调用和库函数所在的内存区是可执行的,然后在进程的内存空间中寻找一个非数据段的可执行空间来存放 shellcode,最后使进程跳转到这一地址执行 shellcode。2) 针对库函数的随机定位,通过使用动态连接器 dl-resolve()的接口即动态地址解析函数 rtld,在溢出执行的过程中动态地决定函数的真正地址。

3.4 相对地址攻击

现有的 ASR 技术一般只随机化了基地址,可以被实施相对地址攻击。Tyler Durden 针对 PaX 只修改了基地址而没有修改主程序代码中语句的相对地址的漏洞[9],提出了一种攻击方法。因为 PaX 只对主程序代码地址中 12-27 位进行了随机修改,所以代码段中特定代码的页内地址与没有采用 PaX 的系统中的页内地址相同,因此可以通过覆盖存在溢出漏洞的函数的返回地址的最后一个字节(页内地址的低 8 位),使其跳转到本页内 call 或 printf 之类的指令,再通过格式化字符串攻击的方法,逐步获取共享库加载的基地址,实现 return-into-libc 攻击。

利用栈内数据间的相对偏移量不变这一特点也可实施攻击。如果知道进程的栈底地址,利用相对偏移量就可以计算出栈内其它变量的地址。由于 stat 文件对每个用户都是可读的,因此栈底地址可以从/proc/<pid>/stat 文件的第 28 项记录获得。一些守护进程或是需要等待用户输入的交互式进程很容易被这种攻击技术利用。一个简单的示例如图 5 所示。

服务端代码 server.c 中存在可利用的漏洞:strcpy (writebuf, str); 首次运行程序时,通过 gdb 获得 writebuf 的地址,并通过 cat/proc/`pidof server`/stat|awk '{print $28}' 命令获得 stack_bottom 的地址,然后计算 offset=&stack_bottom - &writebuf=1496,最后再次运行程序,结果如图 6 所示。在客户端向服务端发起连接后,攻击者就能获得服务端的 root shell,实现权限由普通用户到 root 用户的提升。

```
// server.c:
int main(int argc,char* argv[]){
    ......
    listenfd=socket(AF_INET,SOCK_STREAM,0);
    ......
    servaddr.sin_port=htons(8888);
    ......
    for(;;){
    connfd=accept(listenfd,(struct sockaddr*)NULL,NULL);
    write(connfd,"> ",2);
    n=read(connfd,line,sizeof(line)-1);    line[n]=0;
    strcpy(writebuf, line);
    write(connfd,writebuf,strlen(writebuf));
    close(connfd);
  }
}
```

```
// exploit.c
char shellcode[]="\x31\xc0\x50\x68\x2f\x2f\x73\x68"
"\x68\x2f\x62\x69\x6e\x89\xe3\x50\x89\xe2\x53\x89"
"\xe1\xb0\x0b\xcd\x80";

int main(int argc, char** argv){
    ......
    unsigned long stkptr = strtoul(argv[1], NULL, 10)-1496;
    buf=(char *)malloc(265);
    addrptr=(long *)buf;
    for(i=0;i<264;i+=4)
      *(addrptr++)=stkptr;
    for(i=0;i<strlen(shellcode);i++)
      *(buf++)=shellcode[i];
    buf[264]='\0';
    printf("%s",buf);
}
```

图 5 服务端代码和攻击代码

图 6 测试结果

4 小结

本文重点介绍了地址空间随机化技术在 Linux 系统中的实现,以及如何绕过该技术保护的缓冲区溢出攻击方法。地址空间随机化技术是一种试图阻止各种内存攻击的有效技术。尽管它能有效降低缓冲区溢出攻击成功的概率,但由于随机化空间不够大、随机化粒度不够细等原因,使其不能完全阻止攻击。而细粒度的地址空间随机化又需要对程序内部结构进行较大的修改,这将导致兼容性差而得不到广泛应用。找到易于实现的细粒度随机化方法可能是 ASR 技术未来的研究重点。同时把 ASR 技术同其它的安全防护机制有效结合,可以构建出一套完整的缓冲区溢出防御体系,提高系统的安全强度。

参考文献:

[1] SANS Institute.Top 20 security vulnerabilities [EB/OL].http://www.sans.org/top20/.
[2] 阎雪.黑客就这么几招[M].北京:北京科海电子出版社,2002.
[3] Forst J. C., Osipov V., Bhalla N., et al. Buffer overflow attacks: detect, exploit, prevent [M].Rockland: Syngress Press,2005.
[4] A. Baratloo, N. Singh, T. Tsai. Transparent run-time defense against stack smashing attacks [C].Proceedings of 2000 USENIX Annual Technical Conference (USENIX'00). California: Usenix Association,2000:251-262.
[5] Kumar Avijit, Prateek Gupta, Deepak Gupta. TIED, LibsafePlus: Tools for runtime buffer overflow protection [C]. Proceedings of the 13th Conference on USENIX Security Symposium. Berkeley: Usenix Association, 2002:191-206.
[6] PaX [EB/OL]. http://pageexec.virtualave.net.
[7] Gaurav S. K., Angelos D. K., Vassilis Prevelakis. Countering code-injection attacks with instruction-set randomization [C]. Proceedings of the 10th ACM Conference on Computer and Communications Security. New York: ACM Press, 2003:272-280.
[8] Hackisle. 突破 ASLR 保护和编译器栈保护[EB/OL].http://blog.chinaunix.net/u3/102108/showart_2025891.html.
[9] Tyler Durden. Bypassing PaX ASLR protection [J/OL].Phrack Magazine,2002,11(59).