|  |
| --- |
| 1. 实验目的及实验内容   （本次实验所涉及并要求掌握的知识；实验内容；必要的原理分析） |
| **需求掌握的知识**：汇编的基础知识，虚拟内存的理解，以及gdb的使用  **原理分析：**缓冲区溢出主要是因为动态变量在程序运行时定位于堆栈之中. 溢出, 说白了就是灌满, 使内容物超过顶端, 边缘, 或边界. 此次实验只关心动态缓冲区的溢出问题, 即基于堆栈的缓冲区溢出. |
| 1. 实验环境及实验步骤   （本次实验所使用的器件、仪器设备等的情况；具体的实验步骤）  **操作系统：**Ubuntu 18.10 处理器：Intel x86 CPU  **实验步骤：**1.阅读相关文章，对缓冲区溢出有个基本概念  2.关闭操作系统对堆栈溢出的保护措施  3.编写存在缓冲区溢出的C语言程序  4.Gdb生成汇编代码，分析堆栈的使用情况，从而了解缓冲区溢出的基本原理 |
| 1. 实验过程分析   （实验分工，详细记录实验过程中发生的故障和问题，进行故障分析，说明故障排除的过程及方法。根据具体实验，记录、整理相应的数据表格等） |
| **实验过程：**  **通过堆栈溢出改变函数执行流程测试代码：**  #include <stdio.h>  void function(int a,int b,int c)  {  char buffer1[5];  char buffer2[10];  int \*ret;    ret = (int \*)buffer1 + 12;  (\*ret) += 8;  }  int main()  {  int x;  x=0;  function(1,2,3);  x=1;  printf("%d\n",x);  }  在编译时使用 gcc -z execstack -fno-stack-protector -o exam example3.c 关闭堆栈溢出保护机制 -z execstack（关闭NX）-fno-stack-protector（关闭Cannary）  然后使用命令gdb exam得到汇编代码  **汇编代码：**  Main()  0x0000000000001160 <+0>: push %rbp  0x0000000000001161 <+1>: mov %rsp,%rbp  0x0000000000001164 <+4>: sub $0x10,%rsp  0x0000000000001168 <+8>: movl $0x0,-0x4(%rbp)  0x000000000000116f <+15>: mov $0x3,%edx  0x0000000000001174 <+20>: mov $0x2,%esi  0x0000000000001179 <+25>: mov $0x1,%edi  0x000000000000117e <+30>: callq 0x1135 <function>  0x0000000000001183 <+35>: movl $0x1,-0x4(%rbp)  0x000000000000118a <+42>: mov -0x4(%rbp),%eax  0x000000000000118d <+45>: mov %eax,%esi  0x000000000000118f <+47>: lea 0xe6e(%rip),%rdi # 0x2004  0x0000000000001196 <+54>: mov $0x0,%eax  0x000000000000119b <+59>: callq 0x1030 <printf@plt>  0x00000000000011a0 <+64>: mov $0x0,%eax  0x00000000000011a5 <+69>: leaveq  0x00000000000011a6 <+70>: retq  function()  0x0000000000001135 <+0>: push %rbp  0x0000000000001136 <+1>: mov %rsp,%rbp  0x0000000000001139 <+4>: mov %edi,-0x24(%rbp)  0x000000000000113c <+7>: mov %esi,-0x28(%rbp)  0x000000000000113f <+10>: mov %edx,-0x2c(%rbp)  0x0000000000001142 <+13>: lea -0xd(%rbp),%rax  0x0000000000001146 <+17>: add $0x30,%rax  0x000000000000114a <+21>: mov %rax,-0x8(%rbp)  0x000000000000114e <+25>: mov -0x8(%rbp),%rax  0x0000000000001152 <+29>: mov (%rax),%eax  0x0000000000001154 <+31>: lea 0x8(%rax),%edx  0x0000000000001157 <+34>: mov -0x8(%rbp),%rax  0x000000000000115b <+38>: mov %edx,(%rax)  0x000000000000115d <+40>: nop  0x000000000000115e <+41>: pop %rbp  0x000000000000115f <+42>: retq  此时得到的汇编代码和文章中的代码有些许不同，因为此时函数间传递参数不再使用压栈方法，而是通过普通寄存器来传递参数，但与我们实现该实验没有太大影响，关键是通过让堆栈溢出，使得函数调用时的返回地址被改变，从而达到控制程序甚至操作系统的目的  此例展示的通过修改返回地址使得函数执行流程被改变。    堆栈中在buffer1[]之前的是SFP, SFP之前是返回地址. Ret从buffer1[]的结尾算起是4个字节. 应该记住的是buffer1[]实际上是2个字即8个字节长. 因此返回地址从buffer1[]的开头算起是12个字节. 我们会使用这种方法修改返回地址, 跳过函数调用后面的赋值语句x=1;, 为了做到这一点我们把返回地址加上8个字节.  如下是堆栈的一些信息：  堆栈顶部  +-----------+ 内存低地址  | buffer2 |  +-----------+  | buffer1 |  +-----------+  | sfp |  +-----------+  | ret |  +-----------+  | a |  +-----------+  | b |  +-----------+  | c |  +-----------+ 内存高地址  堆栈底部  **通过堆栈溢出派生出一个shell测试代码：**  #include <unistd.h>  void main() {  char \*name[2];  name[0] = "/bin/sh";  name[1] = NULL;  execve(name[0], name, NULL);  }  编译过程同上  **汇编代码：**  Main()  0x0000000000401b6d <+0>: push %rbp  0x0000000000401b6e <+1>: mov %rsp,%rbp  0x0000000000401b71 <+4>: sub $0x10,%rsp  0x0000000000401b75 <+8>: lea 0x8d488(%rip),%rax # 0x48f004  0x0000000000401b7c <+15>: mov %rax,-0x10(%rbp)  0x0000000000401b80 <+19>: movq $0x0,-0x8(%rbp)  0x0000000000401b88 <+27>: mov -0x10(%rbp),%rax  0x0000000000401b8c <+31>: lea -0x10(%rbp),%rcx  0x0000000000401b90 <+35>: mov $0x0,%edx  0x0000000000401b95 <+40>: mov %rcx,%rsi  0x0000000000401b98 <+43>: mov %rax,%rdi  0x0000000000401b9b <+46>: callq 0x4460b0 <execve>  0x0000000000401ba0 <+51>: nop  0x0000000000401ba1 <+52>: leaveq  0x0000000000401ba2 <+53>: retq  Execve()  0x00000000004460b0 <+0>: mov $0x3b,%eax  0x00000000004460b5 <+5>: syscall  0x00000000004460b7 <+7>: cmp $0xfffffffffffff001,%rax  0x00000000004460bd <+13>: jae 0x4460c0 <execve+16>  0x00000000004460bf <+15>: retq  0x00000000004460c0 <+16>: mov $0xffffffffffffffc0,%rcx  0x00000000004460c7 <+23>: neg %eax  0x00000000004460c9 <+25>: mov %eax,%fs:(%rcx)  0x00000000004460cc <+28>: or $0xffffffffffffffff,%rax  0x00000000004460d0 <+32>: retq  首先创建两个指向字符串的指针，name[0]=“/bin/sh”name[1]=”NULL”，  然后在调用execve函数的时候，将这两个参数压入栈，然后进入内核模式，将这两个参数放到内存某处，“/bin/sh”后面跟着一个空的长字，然后执行指令int x80。此时我们已经向内存中插入一个shell了，但是我们不清楚它被放在了内存的哪个地方，只要我们找到这个地址，然后在其前面插入一个Call指令，则可以调用shell了  **堆栈信息**  内存低地址 内存高地址  +----------------------+------+------+------+------+------+  | DDDDDDDDEEEEEEEEEEEE | EEEE | FFFF | FFFF | FFFF | FFFF |  +----------------------+------+------+------+------+------+  | 89ABCDEF0123456789AB | CDEF | 0123 | 4567 | 89AB | CDEF |  +----------------------+------+------+------+------+------+  | buffer | sfp | ret | a | b | c |  +----------------------+------+------+------+------+------+  | JJSSSSSSSSSSSSSSCCss | ssss | 0xD8 | 0x01 | 0x02 | 0x03 |  +----------------------+------+------+------+------+------+  ↑ |  +-----------------------------+  堆栈顶部 堆栈底部  **遇到的问题及解决：** 在实验过程当中在进行汇编处理时未处理系统对堆栈溢出的相关保护措施，如Cannary（在堆栈返回地址附近的某个位置添加一个随机数，在程序结束时检查是否被替换，如若替换，则认定发生堆栈溢出，直接结束程序），ASLR(通过对攻击者在进行缓冲区溢出攻击时所要用到的内存布局中的偏移做了随机化，ASLR 加大了攻击成功的难度，从而增强了系统的控制流完整性),NX(将数据所在内存页标识为不可执行，当程序溢出成功转入shellcode时，程序会尝试在数据页面上执行指令，此时CPU就会抛出异常，而不是去执行恶意指令)等，导致程序未能按照预期执行。只需要在GCC进行编译时添加指令段，将其关闭即可。 |