|  |
| --- |
| 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） |
| 实验目的：  了解格式化字符串漏洞的产生原理和实现方法  实验内容：  学习论文《Scut\_2001\_Exploiting Format String Vulnerabilities》，了解漏洞的原理，并自主实现代码利用漏洞  在windows和Linux环境下分别实现代码，利用格式化字符串漏洞。 |
| 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） |
| 实验环境：   1. IDE: CodeBlocks 2. 系统：Win10 家庭版 3. ubuntu-18.10-desktop   实验步骤：   1. 阅读论文，并结合函数原型，了解漏洞的机理    1. 阅读原文之——典型场景   第一类。漏洞存在于syslog的第二个参数中。格式化字符串部分是用户提供。  char tmpbuf[512];  snprintf (tmpbuf, sizeof (tmpbuf), "foo: %s", user);  tmpbuf[sizeof (tmpbuf) - 1] = ’\0’;  syslog (LOG\_NOTICE, tmpbuf);  第二类 部分由用户提供的字符串直接传给了格式化函数。  int Error (char \*fmt, ...);  int someotherfunc (char \*user) {  Error (user);  }  第一类漏洞能够由自动化工具安全监测（例如 pscan 或 TESOgcc）。  只有工具被告知函数Error用作格式化函数，第二类漏洞才能检测出来。   * 1. 阅读原文之——格式化字符串漏洞的用途   一：使程序崩溃  几乎所有 UNIX 系统中，CORE都会检测非法指针访问，并且进程会接收到SIGSEGV信号。通常程序会终止并转到CORE。  通过利用格式化字符串，我们可以轻易触发一些无效指针访问：  printf ("%s%s%s%s%s%s%s%s%s%s%s%s");  由于%s展示某个地址中的内存，这个地址位于栈上，栈上也储存了大量其他数据。我们就有很大机会来从非法地址访问数据。同时，多数格式化函数的实现提供了%n参数的功能，可以用于向栈上的地址写入。如果它执行了几次，也一定会产生崩溃。  二：查看进程内存  查看栈：  printf ("%08x.%08x.%08x.%08x.%08x\n");  这是栈内存的部分转储，从当前的栈底一直到栈顶。取决于格式化字符传缓冲区的大小，以及输出缓冲区的大小，使用这种技巧，你可以或多或少重构栈内存的一部分。在一些情况下，你甚至可以获取整个栈内存。  查看任何地址的内存：  为此，我们需要让格式化函数从我们可以提供的某个地址展示内存。这就有两个问题：首先，我们需要找到一个格式化字符串，它将某个地址（传值）用作栈的参数，并且展示其中的内存，并且我们需要提供这个地址。  足够幸运的是，由于%s参数就是干这个的，它展示内存 – 通常是 ASCII 字符串 – 从栈上提供的地址。所以剩下的问题是，如何将这个栈上的地址放到正确的位置上。  格式化函数在内部维护一个指针，指向当前格式化参数的栈区域。如果我们能够将这个指针指向一块可控的内存区域，我们就能向%s参数提供一个地址。为了修改栈指针，我们可以仅仅使用假的参数，它会通过打印垃圾来挖掘栈区。  printf ("AAA0AAA1\_%08x.%08x.%08x.%08x.%08x");  %08x 参数使格式化函数内部的栈指针向栈顶方向增加。将这个参数增加之后，栈指针就指向了我们的内存：格式化字符串本身。格式化函数总是维护最低的栈帧，所以如果我们的缓冲区完全在栈上，它一定会在当前栈指针的上面。如果我们正确选择了%08x的数值，我们就能够展示任意地址的内存，通过向我们的字符串附加%s。  address = 0x08480110  // address (encoded as 32 bit le string): "\x10\x01\x48\x08"  printf ("\x10\x01\x48\x08\_%08x.%08x.%08x.%08x.%08x|%s|");  就会转储0x08480110的内存，直到到达了空字符。  三：任意内存覆盖  可以采取类似于常见的缓冲区溢出的方法  char outbuf[512];  char buffer[512];  sprintf (buffer, "ERR Wrong command: %400s", user);  sprintf (outbuf, buffer);  也可以只利用格式化字符串  strcpy (canary, "AAAA");  printf ("%16u%n%16u%n%32u%n%64u%n", 1, (int \*) &foo[0], 1, (int \*) &foo[1], 1, (int \*) &foo[2], 1, (int \*) &foo[3]);  printf ("%02x%02x%02x%02x\n", foo[0], foo[1], foo[2], foo[3]);  printf ("canary: %02x%02x%02x%02x\n", canary[0], canary[1], canary[2], canary[3]);   * 1. 阅读原文之——爆破方式   基于响应的爆破：  返回地址retaddr和返回地址位置retloc，并使用它来构建完整的偏移独立的利用程序。  为了爆破这个距离，你应该像这样使用格式化字符串。   * "AAAABBBB|stackpop|%08x|"   stackpop取决于我们打算猜测的距离。距离在每次尝试中都会增加：  while (distance > 0) {  strcat (stackpop, "%u");  distance -= 4;  }  如果我们探测距离 32，格式化字符串为：  "AAAABBBB|%u%u%u%u%u%u%u%u|%08x|"  我们从栈上弹出了 32 个字节（8 个%u），并以十六进制，打印了第 32 字节位置的四个字节。理想情况下的输出为：  AAAABBBB|983217938177639561760134608728913021|41414141|  41414141是AAAA是十六进制形式，我们越过了 32 字节的距离。如果你不能通过增加距离到达该模式串，这有两个原因：一是距离太大无法到达，例如如果格式化字符串位于堆上，二是不是以四字节对齐。在后者的情况中，我们仅仅需要在格式化字符串前面插入一个到三个伪造字节。之后我们可以滑动字符串的位置，以便模式串42414141变为正确的模式串41414141。  一旦你设置了对齐和距离，你就可以爆破格式化字符串的缓冲区地址了。因此你使用这样的格式化字符串：  addr|stackpop|\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_%%|%s|  格式化字符串从左到右处理，addr和\_\_\_序列没有任何害处。stackpop将栈指针向上移动，直到它指向addr地址。最后%s打印出addr处的 ASCII 字符串。  在理想情况下，addr会指向我们格式化字符串的\_\_\_序列。这里输出为：  garbage|\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_%|\_\_\_\_\_\_%%|%s||  其中garbage由addr和stackpop输出组成。之后处理的\_\_\_%%字符串会转换为\_\_\_%，因为%%被格式化字符处理器转换为%。之后字符串\_\_\_\_\_\_%%|%s|被插入，因为我们提供的格式化字符串的%s被处理。要注意在我们尝试处理不同的addr值，它是唯一会发生变化的值。在我们的理想情况下，我们让addr直接指向我们的缓冲区。你可以看到，通过观察%%，我们可以分辨出指向我们的格式化字符串的地址（带有两个%字符），以及偶然指向目标缓冲区（只有一个%字符，由于被格式化函数转义）的指针。  如果addr指向了目标缓冲区，输出为：  garbage|\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_%|\_\_\_\_\_\_%||  你可以看到，只有一个%字符。这让我们能够精确预测目标缓冲区，对于格式化字符串在堆中的情况，这会非常有用。  由于我们知道了，我们的%s相对于格式化字符串的起始位于哪里，并且我们拥有了指向缓冲区的地址，我们就可以将地址重定向，以便精确了解我们的格式化字符串在哪里开始。由于你通常希望将 shellcode 放在格式化字符串中，你可以准确计算出相对于格式化字符串地址的retaddr。  盲爆破：  盲爆破不像基于响应的爆破那样直接。基本的理念是，我们可以测量出远程计算机处理格式化字符串的所需时间。类似%.9999999u比简单的%u花费的时间要长。同样，通过在未映射的地址上使用%m，我们可以可靠地产生段错误。   1. 函数分析   \_Check\_return\_opt\_ \_CRTIMP int \_\_cdecl printf(\_In\_z\_ \_Printf\_format\_string\_ const char \*\_Format, ...);  涉及的关键控制符  %n功能  是将%n之前printf已经打印的字符个数赋值给传入的指针。通过%n我们就可以修改内存中的值了。例如： printf("aaaaaaa%n\n",&a);  printf("%d\n",a);  可以发现a的值被printf函数修改为了7。这就是%n的功效了。这是一个不常用到的参数.  它的功能是将%n之前printf已经打印的字符个数赋值给传入的指针。通过%n我们就可以  修改内存中的值了。和%sleak内存一样，只要栈中有我们需要修改的内存的地址就可以使用格式化字符串的漏洞修改它。当然，如果需要修改的数据是相当大的数值时，我们可以使用%02001d或者%2001x这种形式。  %p功能  格式控制符“%p”中的p是pointer（指针）的缩写。指针的值是语言实现（编译程序）相关的，但几乎所有实现中，指针的值都是一个表示地址空间中某个存储器单元的整数。  printf函数族中对于%p一般以十六进制整数方式读取输出指针的值，附加前缀0x。  %$功能  格式化字符串的“$”操作符，其允许我们从格式化字符串中选取一个作为特定的参数。  例如：printf("%3$s", 1, "b", "c", 4);  最终会显示结果“c”。这是因为格式化字符串“%3$s”，它告诉计算机“把格式化字  符串后面第三个参数读取告诉我，然后将参数解释为字符串”。所以，也可以这样做  printf("AAAA%3$n");  printf函数将值“4”（输入的A的数量）写入第三个参数指向的地址。  由此可见printf相关的格式化字符串漏洞主要是：  1.读取任意地址的数值，泄露内存  2.给任意地址写入数据，修改内存  主要实现方式是利用格式化串本身也处于栈中，去用直接参数访问找到这个栈中的格式化串。  这个格式化串可以使用一个要写入的内存地址。也就是直接参数访问+%n格式符+长度表示=向任意地址写入值，这个写入是不能一次写入4字节的，所以可以分两次写入2字节和分四次写入1字节。其中hhn是写一个字节，hn是写两个字节。n是写四个字节。   1. 代码实现   简单起见，采用printf函数演示。  Windows测试： String.cpp  利用%p和%s输出地址。  这里设置两个字符数组str\_test1，str\_test2对比，相同的输入。  输出str\_test1时正确操作，和输入一致。  输出str\_test2时由于没有限制%s，故认为%p不是字符串而是控制符。导致输出了不该输出的地址。    Linux测试： Linuxstring.cpp  可以看到其实gcc编译时已经检查出了错误。  -g为查看汇编代码准备。  这里在printf中多增加了一个控制符，那么输出时就会读取相邻地址的数据。    下面是汇编代码：     1. **具体代码**   **见附件中的源文件** |
| 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） |
| 遇到的问题及解决：   1. 英文原文阅读困难   耐心阅读，并参考了一部分网络上的翻译。  实验结果记录： |