理解栈缓冲区溢出

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。

用编辑器(vim 或其他)输入如下代码,命名为 stack_overflow.c。 建议用虚拟机的同学将该程序在 Windows 平台下保存为 stack_overflow.c,拷贝到虚拟机中。

```
#include <stdio.h>
#include <string.h>
int main(int argc, char *argv[]) {
  int value = 5;
  char buffer one[8], buffer two[8];
  strcpy(buffer one, "one");
  strcpy(buffer two, "two");
 printf("[BEFORE] buffer two is at %p and contains \'%s\'\n",
buffer two, buffer two);
  printf("[BEFORE] buffer one is at %p and contains \'%s\'\n",
buffer one, buffer one);
  printf("[BEFORE] value is at %p and is %d (0x\%08x) \n\n, &value,
value, value);
 printf("[STRCPY] copying %d bytes into buffer two\n\n",
strlen(argv[1]));
  strcpy(buffer two, argv[1]);
 printf("[AFTER] buffer two is at %p and contains \'%s\'\n",
buffer two, buffer two);
 printf("[AFTER] buffer_one is at %p and contains \'%s\'\n",
buffer one, buffer one);
 printf("[AFTER] value is at %p and is %d (0x\%08x)\n", &value, value,
value);
}
```

理解代码,在空白处简要介绍上述代码功能。

整体功能概述:定义两个连续的 buffer 与一个单独的变量,在对 buffer 进行缓冲区溢出操作后观察所有 buffer 与单独变量的变化,以此了解缓冲区溢出的方法与影响效果。 具体功能见注释进行说明。

//定义一个变量,两个大小为 8 的 char 型缓冲区

```
int value = 5;
 char buffer_one[8], buffer_two[8];
// 复制简单内容到两个 buffer 内
 strcpy(buffer one, "one");
 strcpy(buffer_two, "two");
// 输出完成初始化之后的两个 buffer 的内存位置与内容,还有 value 的内存位置与内容
 printf("[BEFORE] buffer two is at %p and contains \'%s\'\n", buffer two, buffer two);
 printf("[BEFORE] buffer one is at %p and contains \'%s\\n", buffer one, buffer one);
 printf("[BEFORE] value is at %p and is %d (0x%08x)\n\n", &value, value, value);
// 将输入参数 argv[1]中内容拷贝到 buffer2 中
 printf("[STRCPY] copying %d bytes into buffer_two\n\n", strlen(argv[1]));
 strcpy(buffer_two, argv[1]);
// 输出完成拷贝操作之后,两个 buffer 与 value 在内存中的位置与内容
 printf("[AFTER] buffer two is at %p and contains \'%s\\n", buffer two, buffer two);
 printf("[AFTER] buffer one is at %p and contains \'%s\'\n", buffer one, buffer one);
 printf("[AFTER] value is at %p and is %d (0x%08x)\n", &value, value, value);
}
```

使用 debug 选项编译代码:

gcc -g -o so stack overflow.c

按照下面命令执行程序,多执行几次该程序,比较输出结果,将某一次输出拷贝在空白处。

(本文中需要拷贝输出的地方可以直接将输出粘贴在空白处,也可以截取屏幕中输出部分的图片 粘贴到空白处。注意:如果采用截屏方式,请仅把输出部分截取出来,不要粘贴整个屏幕。)

./so hi

```
正常执行(未溢出):
 /root/Documents/code/netSec >./so hi
[BEFORE] buffer_two is at 0x7fff86cb7f10 and contains 'two'
[BEFORE] buffer_one is at 0x7fff86cb7f20 and contains 'one'
[BEFORE] value is at 0x7fff86cb7f2c and is 5 (0x00000005)
[STRCPY] copying 2 bytes into buffer_two
[AFTER] buffer_two is at 0x7fff86cb7f10 and contains 'hi'
[AFTER] buffer_one is at 0x7fff86cb7f20 and contains 'one'
[AFTER] value is at 0x7fff86cb7f2c and is 5 (0x00000005)
                 溢出(较短字符串):
 /root/Documents/code/netSec >./so hello_world_abcdefg
[BEFORE] buffer_two is at 0x7ffe253164f0 and contains 'two'
[BEFORE] buffer_one is at 0x7ffe25316500 and contains 'one'
[BEFORE] value is at 0x7ffe2531650c and is 5 (0x000000005)
[STRCPY] copying 19 bytes into buffer_two
[AFTER] buffer_two is at 0x7ffe253164f0 and contains 'hello_worl
d_abcdefg'
[AFTER] buffer_one is at 0x7ffe25316500 and contains 'efg'
[AFTER] value is at 0x7ffe2531650c and is 5 (0x000000005)
                  溢出(较长字符串)
 /root/Documents/code/netSec >./so hello_world_abcdefg_23333333
33333
[BEFORE] buffer_two is at 0x7ffc8da76e40 and contains 'two'
[BEFORE] buffer_one is at 0x7ffc8da76e50 and contains 'one'
[BEFORE] value is at 0x7ffc8da76e5c and is 5 (0x00000005)
[STRCPY] copying 33 bytes into buffer_two
[AFTER] buffer_two is at 0x7ffc8da76e40 and contains 'hello_worl
[AFTER] buffer_one is at 0x7ffc8da76e50 and contains 'efg_233333
3333333'
[AFTER] value is at 0x7ffc8da76e5c and is 858993459 (0x33333333)
```

执行如下命令,将输出拷贝到空白处。

某些操作系统提供栈溢出保护,gcc 在编译时默认对缓冲区边界进行检查,如果你遇到了相关输出,请输入下面指令将源代码编译成不检查边界的模式。如果系统没有提示栈溢出保护,忽略这部分。

gcc -fno-stack-protector -g -o so stack overflow.c

通过这个程序,将你得到的结论写在空白处。

- 1、c/c++栈区中定义的变量是自底向上的(从内存低地址向高地址增长)
- 2、栈区中的所有变量空间连续, 且可以通过偏移访问
- 3、当栈区中某个变量溢出时会向上存储,占用之后的存储单元,即发生缓冲区溢出
- 4、strcpy 函数是没有对溢出的监测与防护,只会按从低地址到高地址依次赋值
- 5、栈区中较为容易使用缓冲区溢出方式进行攻击

用编辑器(vim 或其他)输入如下代码,命名为 ans_check.c。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int check answer(char *ans) {
  int ans_flag = 0;
 char ans buf[16];
  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");
}
```

理解代码,在空白处简要介绍上述代码功能。

输入一个字符串,仅当其为"forty-two"时输出"Right answer",否则为"Wrong ansser" 其中 check_ans 函数,先对传入的 ans 进行复制,然后再进行判断返回结果。 编译代码:

```
gcc -g -fno-stack-protector -o ans check ans check.c
按如下方式执行该程序,将控制台信息(含输入和输出)复制在空白处。
./ans check
./ans check yes
./ans check forty-two
                              #注:10 个 1 (32 位计算机)
./ans check 1111111111
                           #注:16 个 1 (32 位计算机)
./ans check 1111111111111111
                              #注:17 个 1 (32 位计算机)
./ans_check 11111111111111111
                           ./ans_check
   /root/Documents/code/netSec >./ans_check
   Usage: ./ans_check <answer>
                         ./ans_check yes
     /root/Documents/code/netSec >./ans_check yes
   Wrong answer!
                       ./ans check forty-two
   /root/Documents/code/netSec >./ans_check forty-two
  Right answer!
                      ./ans check 1111111111
   /root/Documents/code/netSec >./ans_check 1111111111
  Wrong answer!
                   ./ans check 11111111111111111
    /root/Documents/code/netSec >./ans_check 1111111111
  111111
  Wrong answer!
                   ./ans check 1111111111111111111
    /root/Documents/code/netSec >./ans_check 1111111111
  1111111
  Wrong answer!
```

./ans check 11111111111111111111111111111111(30 个 1,64 位系统)

在 30 个 1 之后, 出现 Right answer, 在 33 个 1 时, 出现段错误。

静静地思考 2 分钟,简要解释一下最后一个命令的结果,填写在空白处。

在 check_ans 函数内,首先对传入的 ans 进行拷贝,拷贝结果存于 ans_buf[16]内,当字符串长度小于等于 16(32 位系统)时,strcpy 函数仍可以进行比较判断,修改 ans_flag 的值。但当 ans 的长度超过 16 时,产生了缓冲区溢出,strcpy 在栈区内向上赋值,修改了 ans flag 的值,使 ans flag 的值不为 0,故在返回时出现错误的结果。

为了具体分析发生结果的细节,使用 gdb 进行检查,输入如下命令:

```
gdb ans_check -q
list 1
list
list
break 10 #注: 断点放在 strcpy 所在行,根据程序行号调整
break 15 #注: 断点放在该函数 return 语句所在行,根据程序行号调整
run 11111111111111 #注:17 个 1
```

将最后一行命令的结果填写到空白处。

```
在 64 位操作系统下,在小于等于 31 个"1"时,可以正常结束程序且"1"的个数在 30、31 时可以显
示为"Right answer",但"1"的个数大于等于 32 个时,程序会出错。截图如下:
/root/Documents/code/netSec >gdb ans_check -q
Reading symbols from /root/Documents/code/netSec/ans_check...done.
(gdb) break 10
Breakpoint 1 at 0x400663: file ./ans_check.c, line 10.
(gdb) break 15
Breakpoint 2 at 0x400692: file ./ans_check.c, line 15.
Breakpoint 1, check_answer (ans=0x7fffffffe5dc '1' <repeats 32 times>)
   at ./ans_check.c:10
10
        strcpy(ans_buf, ans);
(gdb) c
Continuing.
Breakpoint 2, check_answer (ans=0x7fffffffe5dc '1' <repeats 32 times>)
   at ./ans_check.c:15
15
        return ans_flag;
(gdb) c
Continuing.
Right answer!
auit
^C
rogram received signal SIGINT, Interrupt.
x000007fffff7a91bc3 in _IO_new_file_overflow (f=0x0, ch=-7360) at fileops.c:875
375
      if (_IO_do_write (f, f->_IO_write_base,
(gab)
```

大家熟知,断点位置是程序尚未执行的位置。因此,程序当前在 strcpy 所在行,但并未执行该行指令。检查下面两个变量的值,将结果粘贴到空白处。

```
x/s ans_buf
x/x &ans flag
```

```
The program is not being run.

(gdb) run 111111111111111111111111111

Starting program: /root/Documents/code/netSec/ans_check 11111111111111111111111

Breakpoint 1, check_answer (
    ans=0x7fffffffe5dc '1' <repeats 32 times>)
    at ./ans_check.c:10

10    strcpy(ans_buf, ans);

(gdb) x/s ans_buf

0x7fffffffe220: "\001"

(gdb) x/x &ans_flag

0x7fffffffe23c: 0x00

(gdb) |
```

在 gdb 中输入命令 c <enter>,继续执行程序。

在这个断点, strcpy 已经执行完毕, 再次检查变量并将输出结果拷贝在空白处。

```
x/s ans_buf
x/x &ans flag
```

```
(gdb) x/s ans_buf
0x7fffffffe220: '1' <repeats 32 times>
(gdb) x/x &ans_flag
0x7fffffffe23c: 0x31
```

由此可见,栈中的缓冲区溢出改变了条件变量的值。C语言中,任何非零值都是 true,因此,覆盖后的字符'1'(0x31)表示 true。

(输入 quit 退出 gdb)

简单修改代码,将 ans flag 和 ans buf 两个变量声明交换位置,编译后运行:

./ans_check 111111111111111 #注:17 个 1

将输出结果和解释写到空白处。

此处为好更好的契合实验,使用 32 位 ubuntu 系统以完成实验

joker@joker-virtual-machine:~/Desktop\$./ans_check \$(python -c "print '1'*17") d 1 1 100 Right answer!

结果不变,但据理论分析,缓冲区溢出但是不会修改 flag 的结果。

这里经查找发现是编译器对这个进行了优化,故无法显示正确结果。

这类溢出与程序设计部分相关,但这不是一个通用的方法:

在栈区连续的情况下是一个通用的方法,但不同的编译器有不同的优化,无法保证所有的栈区都是这样连续分布,故不是一个通用的方法。

这类栈溢出属于"溢出(后覆盖其他)变量"类型。

思考 1 分钟: 这类栈溢是否与程序设计有关? 这类栈溢出的利用是否是通用方法?

下面,我们将介绍一种普遍存在于所有程序中的栈溢出问题和利用,这种方法通过**修改函数返回地址**实现漏洞的利用。("溢出地址"类型)

重新启动 gdb:

```
gdb ans check -q
```

在第 25 行(调用 check_answer()函数的地方)设断点。

反编译 main 函数。

disass main

在 main 函数的反编译代码中,找到断点所在位置。将断点所在指令到 main 函数结尾的汇编代码 复制在下面。

```
0x080485b0 <+9>:
                       cmpl
                              $0x1,0x8(%ebp)
  0x080485b4 <+13>:
                       jg
                              0x80485d7 <main+48>
  0x080485b6 <+15>:
                              0xc(%ebp),%eax
                       mov
                              (%eax),%eax
  0x080485b9 <+18>:
                       MOV
                     mov
movl
  0x080485bb <+20>:
                              %eax,0x4(%esp)
                              $0x80486b7,(%esp)
  0x080485bf <+24>:
                              0x80483c0 <printf@plt>
  0x080485c6 <+31>:
                      call
  0x080485cb <+36>: movl
                              $0x0,(%esp)
                     call
mov
add
  0x080485d2 <+43>:
                              0x8048400 <exit@plt>
  0x080485d7 <+48>:
                              0xc(%ebp),%eax
  0x080485da <+51>:
                              $0x4,%eax
  0x080485dd <+54>:
                      mov
                              (%eax),%eax
                       mov
  0x080485df <+56>:
                              %eax,(%esp)
  0x080485e2 <+59>:
                     call
                              0x804852d <check_answe
  0x080485e7 <+64>:
                      test
                              %eax,%eax
                     je
                              0x80485f9 <main+82>
  0x080485e9 <+66>:
  0x080485eb <+68>:
                              $0x80486cb,(%esp)
                     movl
                      call
  0x080485f2 <+75>:
                              0x80483e0 <puts@plt>
                              0x8048605 <main+94>
  0x080485f7 <+80>:
                       jmp
                       movl
                              $0x80486d9,(%esp)
  0x080485f9 <+82>:
  0x08048600 <+89>:
                       call
                              0x80483e0 <puts@plt>
  0x08048605 <+94>:
                       leave
  0x08048606 <+95>:
                       ret
End of assembler dump.
```

再将第 10 行和 15 行设为断点(strcpy 所在行和所在函数 return)。这样,该程序共有 3 个 断点: 25 行 (pre-call), 10 行 (pre-strcpy)和 15 行 (pre-return).

在 gdb 中执行如下命令

run 111111111111111 #注:17 个 1

用以下指令检查栈寄存器和栈内容:

i r esp
x/32xw \$esp

将输出粘贴在空白处A。

(空白处 A) 第 25 行断点信息 Breakpoint 1, main (argc=2, argv=0xbffff094) at ans_check.c:25 warning: Source file is more recent than executable. if (check_answer(argv[1])) { (gdb) i r esp 0xbfffefe0 esp 0xbfffefe0 (gdb) x/32xw \$esp 0xbfffefe0: 0xb7fc33c4 0xb7fff000 0x0804861b 0xb7fc3000 0xbfffeff0: 0x08048610 0x00000000 0x00000000 0xb7e31a83 0xbffff000: 0xbffff094 0xbffff0a0 0x00000002 0xb7feccea 0xbffff010: 0x00000002 0xbffff094 0xbfffff034 0x0804a024 0xbffff020: 0x0804826c 0xb7fc3000 0x00000000 0x00000000 0xbffff030: 0x00000000 0x71c9ac78 0x481d2868 0x00000000 0xbffff040: 0x00000000 0x00000000 0x00000002 0x08048430 0xbffff050: 0x00000000 0xb7ff2500 0xb7e31999 0xb7fff000

上述输出是栈在调用 check answer 之前的内容和位置

输入 c <enter> 到下一个断点

使用下面指令再次检查栈信息,并粘贴输出到空白处 B。

i r esp
x/32xw \$esp

(空白处B)

第10行断点信息

```
Breakpoint 2, check_answer (ans=0xbffff2aa '1' <repeats 17 times>) at ans_check.c:10
          strcpy(ans_buf, ans);
(gdb) iresp
               0xbfffefa0
                                0xbfffefa0
(gdb) x/32xw $esp
0xbfffefa0:
                0xffffffff
                                0xbfffefce
                                               0xb7e24bf8
                                                               0xb7e4b493
0xbfffefb0:
                0x00000000
                                0x00c30000
                                               0x00000001
                                                               0x0804833d
0xbfffefc0:
                0xbffff28c
                                0x0000002f
                                               0x0804a000
                                                               0x00000000
0xbfffefd0:
                0x00000002
                                0xbffff094
                                               0xbfffeff8
                                                               0x0804855f
0xbfffefe0:
                0xbffff2aa
                                0xb7fff000
                                               0x0804858b
                                                               0xb7fc3000
0xbfffeff0:
                                               0x00000000
                0x08048580
                                0x00000000
                                                               0xb7e31a83
0xbffff000:
             0x00000002
                                               0xbffff0a0
0xbffff034
                                0xbffff094
                                                               0xb7feccea
0xbffff010:
                0x00000002
                                0xbffff094
                                                               0x0804a024
(gdb) x/w ans_buf
              0x0804833d
0xbfffefbc:
(gdb) x/w &ans_flag
0xbfffefcc:
                0x00000000
```

Breakpoint 2, check_answer (ans=0xbffff2aa '1' <repeats 17 times>) at ans_check.c:10 strcpy(ans_buf, ans);

(gdb) i r esp

esp 0xbfffefa0 0xbfffefa0

(gdb) x/32xw \$esp

0xbfffefa0:	0xfffffff	0xbfffefce	0xb7e24bf8	0xb7e4b493
0xbfffefb0:	0x00000000	0x00c30000	0x0000001	0x0804833d
0xbfffefc0:	0xbffff28c	0x0000002f	0x0804a000	0x00000000
0xbfffefd0:	0x00000002	0xbffff094	0xbfffeff8	0x0804855f
0xbfffefe0:	0xbffff2aa	0xb7fff000	0x0804858b	0xb7fc3000
0xbfffeff0:	0x08048580	0x00000000	0x00000000	0xb7e31a83
0xbffff000:	0x00000002	0xbffff094	0xbffff0a0	0xb7feccea
0xbffff010:	0x00000002	0xbffff094	0xbffff034	0x0804a024

(gdb) x/w ans buf

0xbfffefbc: 0x0804833d

(gdb) x/w &ans_flag

0xbfffefcc: 0x00000000

检查如下两个静态变量的地址和内容

x/s ans_buf
x/x &ans flag

在空白处 B 记录的栈中找到上面两个变量内容,将对应地址的字体加粗。同时,将返回地址的字体加粗。

最后,继续执行程序到下一个断点 c <enter>,使用如下命令检查堆栈,将内容粘贴到空白处 C,将 ans_buf、ans_flag 和返回地址的字体加粗。

i r esp x/32xw \$esp

/京台県 (2)					
(空白处 C)		<i>}</i>	三人叫 上户 白		
		—	5 行断点信息	,	
(gdb) i r					
esp	0xbfffe	fa0 0x	bfffefa0		
(gdb) x/32		- 5	L 66660	0	0
0xbfffefa0 0xbfffefb0			bffff2aa :00c30000	0xb7e24bf8 0x0000001	0xb7e4b493 0x31313131
0xbfffefc0			31313131	0x31313131	0x00000031
0xbfffefd0			bffff094	0xbfffeff8	0x0804855f
0xbfffefe0	e Oxbfff	f2aa 0x	b7fff000	0x0804858b	0xb7fc3000
0xbfffeff0			00000000	0x00000000	0xb7e31a83
0xbffff000			bffff094	0xbffff0a0	0xb7feccea
0xbffff010		0002 0x	bffff094	0xbffff034	0x0804a024
(gdb) x/w 0xbfffefbc	_	3131			
(gdb) x/w		3131			
0xbfffefcc		0031			
(adb)					
(adh) i r con					
(gdb) i r esp esp 0:	xbfffefa0 0xbfffe	ofo0			
(gdb) x/32xw		siau			
0xbfffefa0:	0xbfffefbc	0xbffff2aa	0xb7e24bf8	0xb7e4b493	
0xbfffefb0:	0x00000000	0x00c30000	0x00000001	0x31313131	
0xbfffefc0:	0x31313131	0x31313131	0x31313131	0x00000031	
0xbfffefd0:	0x00000002	0x51515151 0xbffff094	0x51515151 0xbfffeff8	0x0804855f	
0xbfffefe0:	0xbffff2aa	0xb1111094 0xb7fff000	0x0804858b	0xb7fc3000	
0xbfffeff0:	0x08048580	0x00000000	0x000000000	0xb7ic3000 0xb7e31a83	
0xbffff000:	0x00046560	0x00000000 0xbffff094	0xbffff0a0	0xb7e31a63 0xb7feccea	
0xbffff010:	0x00000002	0xbffff094	0xbffff034	0x0804a024	
(gdb) x/w ans	_				
0xbfffefbc:	0x31313131				
(gdb) x/w &a					
0xbfffefcc:	0x00000031				

这里可以看到, ans_flag 变量被改写了,但返回地址没有变化。

重复 Gate 5 步骤,但这次找到覆盖返回地址的最小字符串长度。在 gdb 中使用下面命令输出你覆盖后的堆栈,并用加粗你覆盖的返回地址。

i r esp
x/32xw \$esp

```
通过"二分查找"的方式,得到发生段错误的字符串长度如下:
                                                        (为28)
joker@joker-virtual-machine:~/Desktop$ ./ans_check $(python -c "print '1'*27")
Right answer!
joker@joker-virtual-machine:~/Desktop$ ./ans_check $(python -c "print '1'*28")
Right answer!
Segmentation fault (core dumped)
下为返回地址:
Breakpoint 1, main (argc=2, argv=0xbffff094) at ans_check.c:25
25
       if (check answer(argv[1])) {
(gdb) i r esp
         0xbfffefe0 0xbfffefe0
esp
(gdb) x/32xw $esp
0xbfffefe0:
             0xb7fc33c4
                          0xb7fff000
                                       0x0804858b
                                                    0xb7fc3000
            0x08048580
                         0x00000000
0xbfffeff0:
                                       0x00000000
                                                    0xb7e31a83
0xbffff000:
            0x00000002 0xbffff094
                                       0xbffff0a0
                                                    0xb7feccea
0xbffff010:
            0x00000002 0xbffff094
                                       0xbffff034
                                                    0x0804a024
0xbffff020:
             0x0804825c 0xb7fc3000
                                       0x00000000
                                                    0x00000000
                                                    0x00000000
0xbffff030:
            0x00000000
                         0xdf464dfc
                                       0xe692c9ec
0xbffff040:
             0x00000000
                         0x00000000
                                       0x00000002
                                                    0x080483e0
0xbffff050:
             0x00000000
                         0xb7ff2500
                                       0xb7e31999
                                                    0xb7fff000
(gdb) n
Breakpoint 2, check answer (ans=0xbffff29f '1' <repeats 28 times>) at ans check.c:10
10
       strcpy(ans buf, ans);
(gdb) i r esp
esp
         0xbfffefa0 0xbfffefa0
(gdb) x/32xw $esp
0xbfffefa0:
            0xffffffff
                          0xbfffefce
                                       0xb7e24bf8
                                                    0xb7e4b493
0xbfffefb0:
             0x00000000
                          0x00c30000
                                       0x0000001
                                                    0x0804833d
0xbfffefc0:
            0xbffff281
                          0x0000002f
                                       0x0804a000
                                                    0x0000000
0xbfffefd0:
             0x00000002
                         0xbffff094
                                       0xbfffeff8
                                                    0x0804855f
0xbfffefe0:
             0xbffff29f
                          0xb7fff000
                                       0x0804858b
                                                    0xb7fc3000
0xbfffeff0:
             0x08048580
                         0x00000000
                                      0x00000000
                                                    0xb7e31a83
```

```
0xbffff000:
                                         0xbffff0a0
                                                       0xb7feccea
             0x00000002
                           0xbffff094
0xbffff010:
             0x00000002
                           0xbffff094
                                         0xbffff034
                                                       0x0804a024
(gdb) x/w ans_buf
0xbfffefbc:
             0x0804833d
(gdb) x/w &ans flag
             0x00000000
0xbfffefcc:
(qdb) n
12
        if (strcmp(ans_buf, "forty-two") == 0)
(gdb) i r esp
esp
          0xbfffefa0 0xbfffefa0
(gdb) x/32xw $esp
0xbfffefa0:
             0xbfffefbc
                           0xbffff29f
                                         0xb7e24bf8
                                                       0xb7e4b493
             0x00000000
0xbfffefb0:
                           0x00c30000
                                         0x00000001
                                                       0x31313131
0xbfffefc0:
             0x31313131
                           0x31313131
                                         0x31313131
                                                       0x31313131
0xbfffefd0:
             0x31313131
                           0x31313131
                                         0xbfffef00
                                                       0x0804855f
0xbfffefe0:
             0xbffff29f
                           0xb7fff000
                                         0x0804858b
                                                       0xb7fc3000
                                         0x00000000
0xbfffeff0:
             0x08048580
                           0x00000000
                                                       0xb7e31a83
0xbffff000:
             0x00000002 0xbffff094
                                         0xbffff0a0
                                                       0xb7feccea
0xbffff010:
             0x00000002
                                         0xbffff034
                                                       0x0804a024
                           0xbffff094
(gdb) x/w ans buf
0xbfffefbc:
             0x31313131
(gdb) x/w &ans flag
             0x31313131
0xbfffefcc:
(gdb) n
Breakpoint 3, check answer (ans=0xbffff29f '1' <repeats 28 times>) at ans check.c:15
15
        return ans flag;
(gdb) i r esp
          0xbfffefa0 0xbfffefa0
esp
(gdb) x/32xw $esp
0xbfffefa0:
             0xbfffefbc
                           0x08048610 0xb7e24bf8
                                                       0xb7e4b493
0xbfffefb0:
             0x00000000
                           0x00c30000
                                         0x0000001
                                                       0x31313131
0xbfffefc0:
             0x31313131
                           0x31313131
                                         0x31313131
                                                       0x31313131
0xbfffefd0:
             0x31313131
                           0x31313131
                                         0xbfffef00
                                                       0x0804855f
0xbfffefe0:
             0xbffff29f
                           0xb7fff000
                                         0x0804858b
                                                       0xb7fc3000
0xbfffeff0:
             0x08048580
                           0x00000000
                                         0x00000000
                                                       0xb7e31a83
0xbffff000:
             0x00000002
                           0xbffff094
                                         0xbffff0a0
                                                       0xb7feccea
0xbffff010:
             0x00000002
                           0xbffff094
                                         0xbffff034
                                                       0x0804a024
(gdb) x/w ans_buf
0xbfffefbc:
             0x31313131
(gdb) x/w &ans flag
0xbfffefcc:
             0x31313131
(gdb) n
```

北京理工大学网络信息安全课程 http://www.bit-netlab.org

下课了!