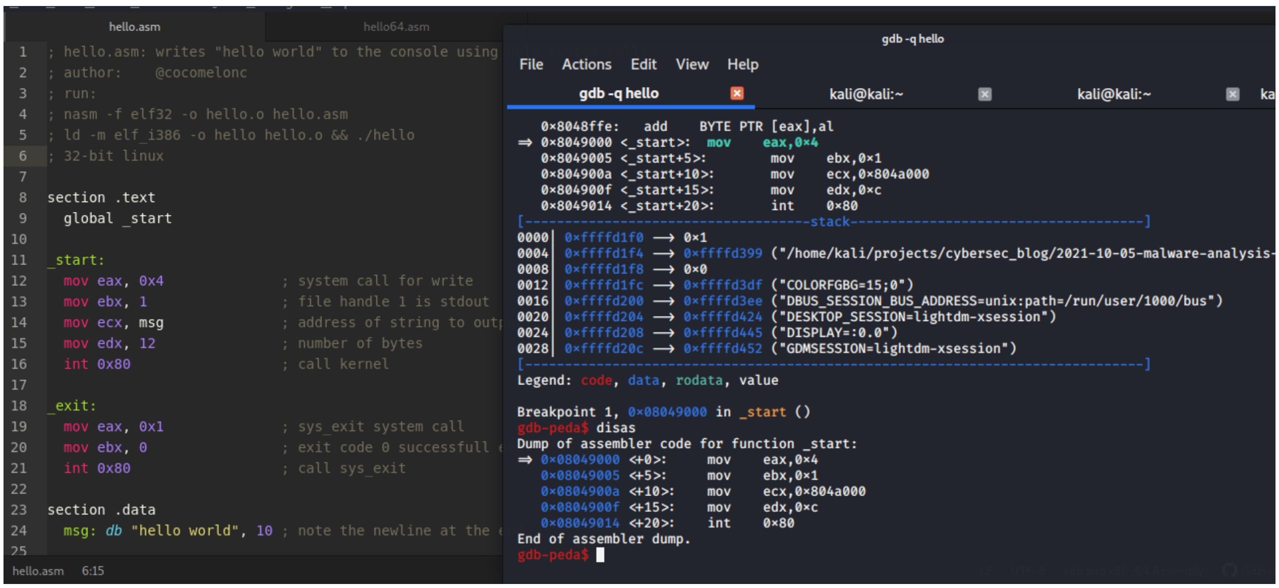
Malware analysis - part 2: My NASM tutorial.

cocomelonc.github.io/tutorial/2021/10/08/malware-analysis-2.html

October 8, 2021

18 minute read

Hello, cybersecurity enthusiasts and white hackers!



NASM tutorial

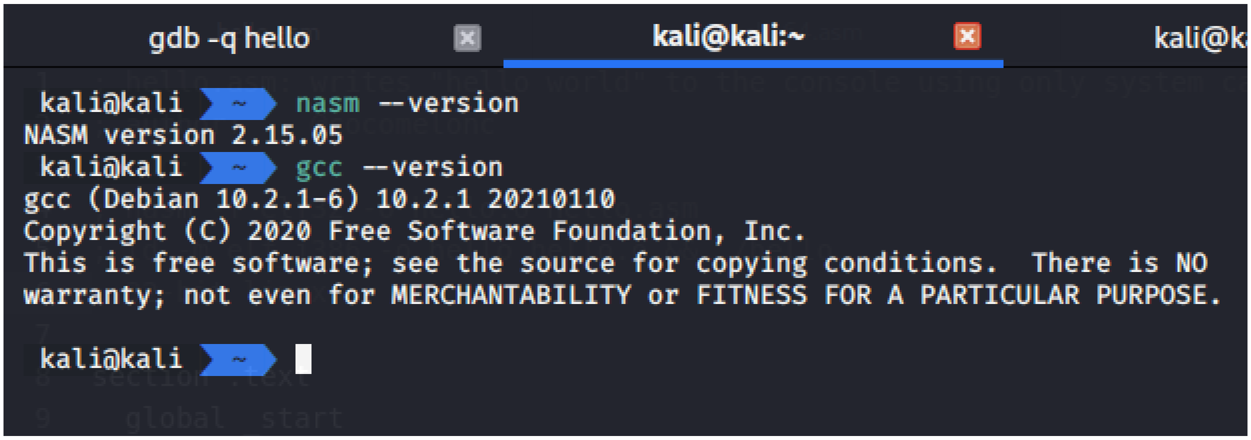
So, I am continuing a series of articles dedicated to my journey in the study of malware analysis.

In the last post in the series, I started learning examples in assembly language.

This tutorial will show you how to write assembly language programs on the x86 architecture, but now I will also provide code examples that integrate with C language.

Once again, make sure we have both nasm and gcc installed:

```
nasm --version
gcc --version
```



Let's go to repeat some instructions:

```
mov a, b   ; copy b to a
and a, b   ; copy "a logical AND b" to a
or  a, b   ; copy "a logical OR b" to a
xor a, b   ; copy "a logical XOR b" to a
add a, b   ; copy a + b to a
sub a, b   ; copy a - b to a
inc a      ; increment a (copy a + 1 to a)
dec a      ; decrement a (copy a - 1 to a)
db         ; pseudo-instruction that declares bytes
           ; will be in memory when the program runs
```

As i wrote earlier, in fact, most of the basic instructions have only the following forms:

```
mov eax, ebx      ; copy register to register
mov ebx, [123]    ; copy memory address to register
mov [123], eax    ; copy register to memory address
mov eax, 0x12     ; copy immediate to register
mov [151], 0x55   ; copy immediate to memory address
```

Pseudo-instructions are things which, though not real x86 machine instructions, are used in the instruction field anyway because that's the most convenient place to put them:

```
db    0x55                 ; just the byte 0x55
db    0x55,0x56,0x57       ; three bytes in succession
db    'a',0x55             ; character constants are OK
db    'hello',13,10,'$'    ; so are string constants
dw    0x1234               ; 0x34 0x12
dw    'a'                  ; 0x61 0x00 (it's just a number)
dw    'ab'                 ; 0x61 0x62 (character constant)
dw    'abc'                ; 0x61 0x62 0x63 0x00 (string)
dd    0x12345678           ; 0x78 0x56 0x34 0x12
dd    1.234567e20          ; floating-point constant
dq    0x123456789abcdef0   ; eight byte constant
dq    1.234567e20          ; double-precision float
dt    1.234567e20          ; extended-precision float
```

To reserve space (without initializing), you can use the following pseudo instructions. They should go in a section called `.bss` (you'll get an error if you try to use them in a `.text` section):

```
buffer:     resb  64  ; reserve 64 bytes
wordvar:    resw  1   ; reserve a word
realarray:  resq  10  ; array of ten reals
```

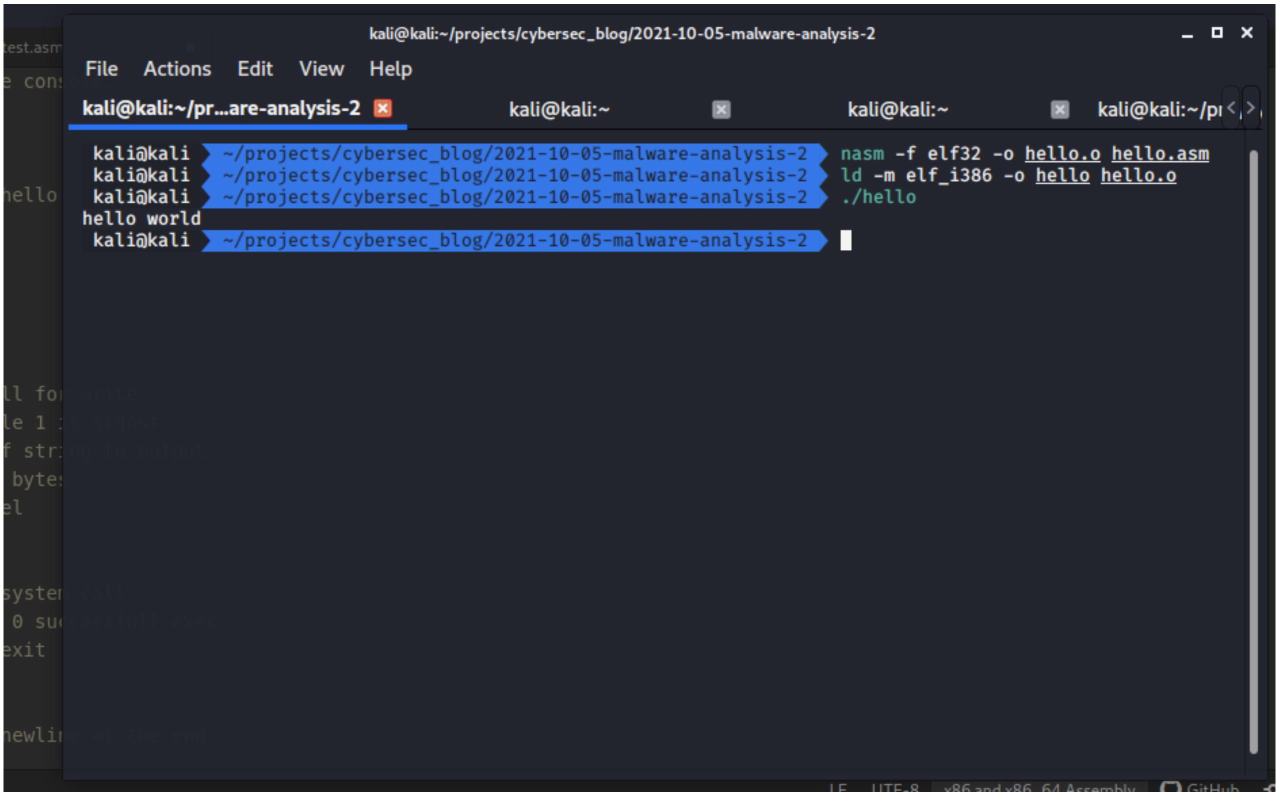
hello world

So what about our first practical example? Let's start with the classic "Hello world" program:

```
; hello.asm: writes "hello world" to the console.
; author:    @cocomelonc
; run:
; nasm -f elf32 -o hello.o hello.asm
; ld -m elf_i386 -o hello hello.o && ./hello
; 32-bit linux

section .text
  global _start

_start:
  mov eax, 0x4              ; system call for write
  mov ebx, 1                ; file handle 1 is stdout
  mov ecx, msg              ; address of string to output
  mov edx, 12               ; number of bytes
  int 0x80                  ; call kernel

_exit:
  mov eax, 0x1              ; sys_exit system call
  mov ebx, 0                ; exit code 0 successfull exec
  int 0x80                  ; call sys_exit

section .data
  msg: db "hello world", 10 ; note the newline at the end
```

Compile and run:

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



As you can see everything work as expected. Our program writes “hello world” to the console using only system calls. Let’s examine lines 12-16:

```
11  _start:
12    mov eax, 0x4              ; system call for write
13    mov ebx, 1                ; file handle 1 is stdout
14    mov ecx, msg              ; address of string to output
15    mov edx, 12               ; number of bytes
16    int 0x80                  ; call kernel
17
```

Everything is written in the comments to my code:
line 12: system call for write.
line 13: file descriptor (stdout).
line 14: message “hello world”.
line 15: number of bytes.
line 16: system interrupt call.

As for lines 19-21:

```
18  _exit:
19    mov eax, 0x1              ; sys_exit system call
20    mov ebx, 0                ; exit code 0 successfull exec
21    int 0x80                  ; call sys_exit
```

they are identical to the logic from an example from first post, it's just normal exit logic.

I hope you haven't forgotten about the instruction `int 0x80`. There is an `int 0x80` instruction in the assembler code. This is a system interrupt. When the processor receives interrupt `0x80`, it performs the requested system call in kernel mode, while getting the desired handler from the Interrupt Descriptor Table.

hello world via using C library

Let's go to code our "hello world" example with using C library. Remember how in C execution "starts" at the function `main`? That's because the C library actually has the `_start` label inside itself! The code at `_start` does some initialization, then it calls `main`, then it does some clean up, then it issues the system call for exit. So you just have to implement `main`. We can do that in assembly!

```
; hello.asm: writes "hello world" to the console by using C lib.
; author:    @cocomelonc
; run:
; nasm -f elf32 -o hello2.o hello2.asm
; gcc -static -m32 -o hello2 hello2.o && ./hello2
; 32-bit linux

section .text
global main
extern puts

main:                        ; called by C lib startup code
  push msg                   ; address of string to output
  call puts                  ; puts (msg)
  add esp, 4                 ; update stack pointer (1 argument 4 byte)
  xor eax, eax               ; a faster way of setting eax to zero
  ret                        ; return from main back into C library wrapper

msg: db "hello world", 0     ; note strings must be terminated with 0 in C
```

which is equivalent in C:

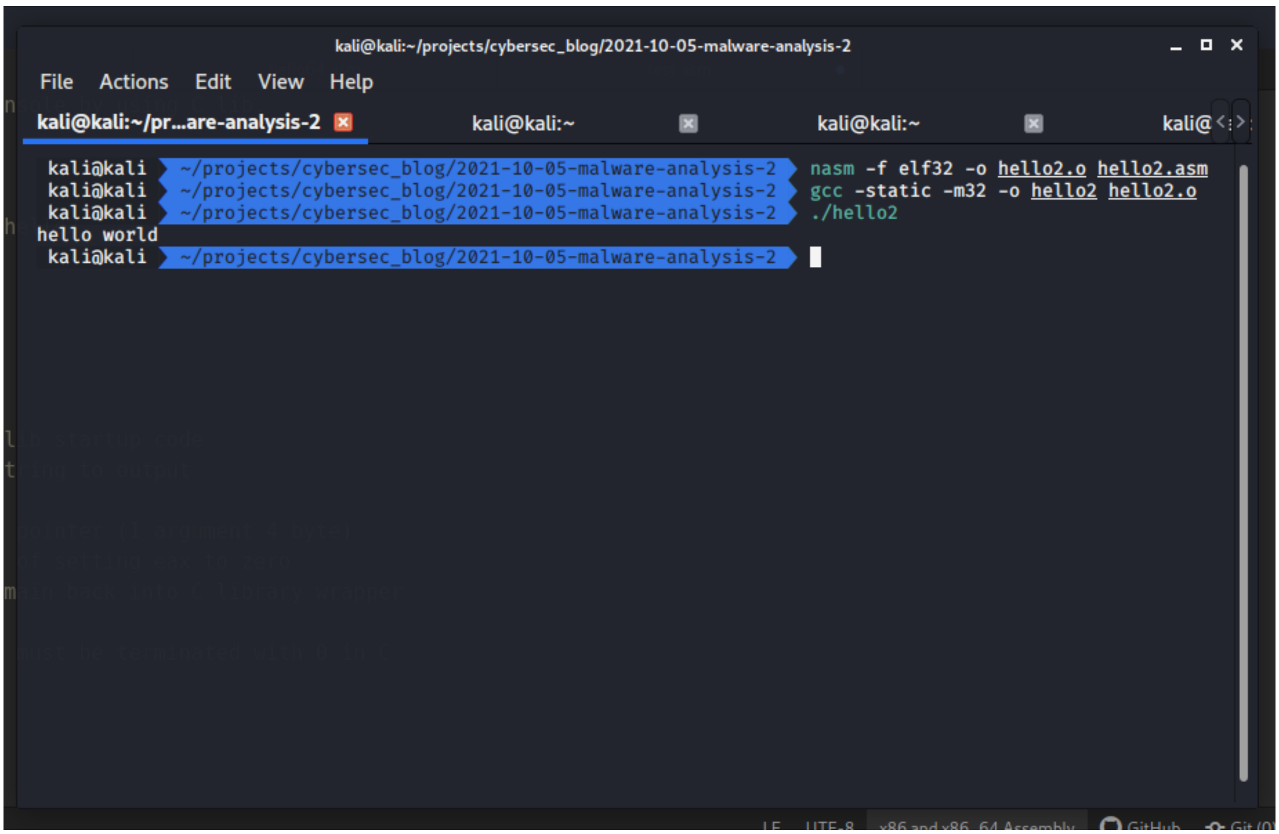
```
#include <stdio.h>

int main(void) {
  puts ("hello world");
  return 0;
}
```

I think from the comments to the code everything should be clear, this is a simplest example:

on line 14, a call to the `puts()` function: `call puts`. Before this call, the address of the string (or a pointer to it) with our "hello world" is pushed onto the stack using the `push` instruction. After the `puts()` function returns control to the `main()` function, the address of the string (or a pointer to it) is still on the stack. Since it is no longer needed, the stack pointer (`esp` register) is updated. `add esp, 4` means add 4 to the value in the `ESP` register. Why 4? Because this is 32 bit code. After calling `puts()`, the original C code states `return 0` - return 0 as the result of the `main()` function. In the generated code, this is provided by the instruction: `xor eax, eax`

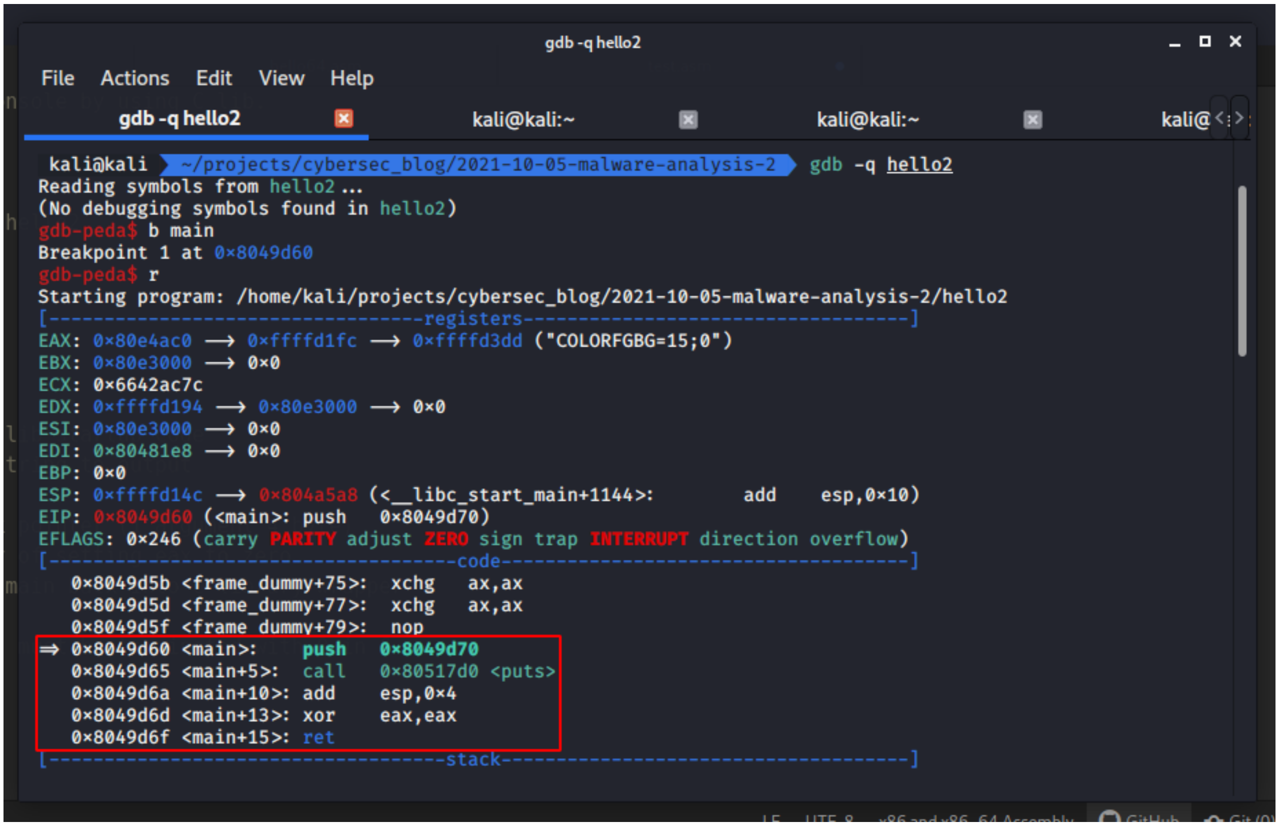
Let's go to compile and run:

```
nasm -f elf32 -o hello2.o hello2.asm
gcc -static -m32 -o hello2 hello2.o
./hello2
```



As you can see again everything is good.
Let's go to load this binary to `gdb` and debug:

```
gdb -q hello2
```

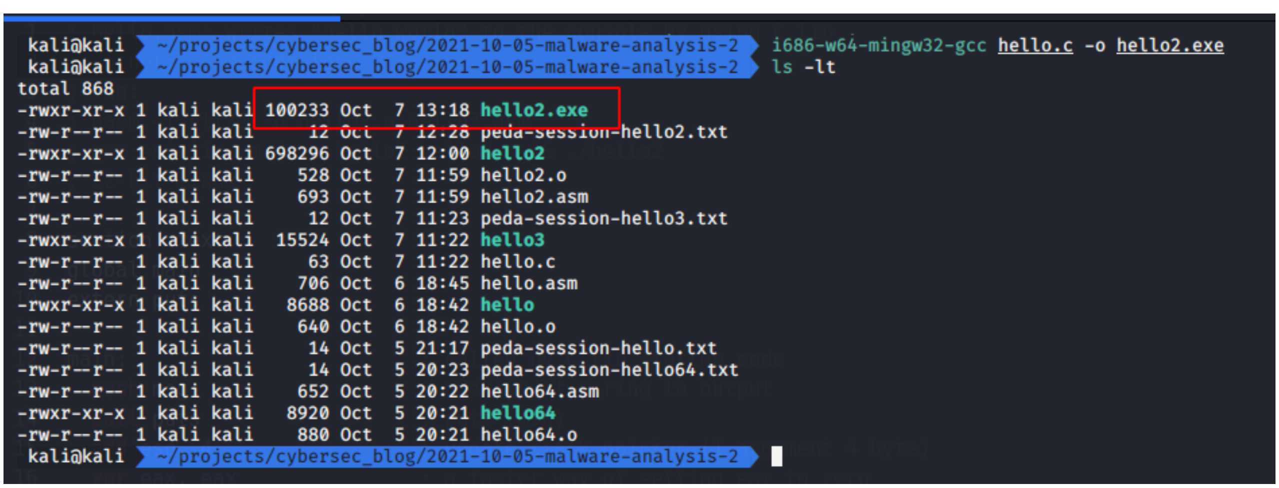


Let’s now cross-compile the C code:

```
#include <stdio.h>

int main(void) {
    puts ("hello world");
    return 0;
}
```

to an .exe file:

```
i686-w64-mingw32-gcc hello.c -o hello2.exe
```



Basic static analysis

Since I consider all my examples from the point of view of a malware analyst, let's do a little static analysis of our three files:

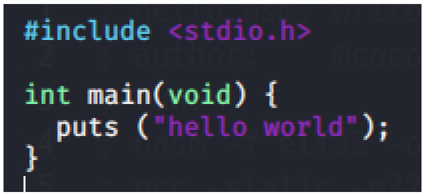
`hello` - compilation result of `hello.asm`:

```
; hello.asm: writes "hello world" to the console.
; author:    @cocomelonc
; run:
; nasm -f elf32 -o hello.o hello.asm
; ld -m elf_i386 -o hello hello.o && ./hello
; 32-bit linux

section .text
  global _start

_start:
  mov eax, 0x4                  ; system call for write
  mov ebx, 1                    ; file handle 1 is stdout
  mov ecx, msg                  ; address of string to output
  mov edx, 12                   ; number of bytes
  int 0x80                      ; call kernel

_exit:
  mov eax, 0x1                  ; sys_exit system call
  mov ebx, 0                    ; exit code 0 successfull exec
  int 0x80                      ; call sys_exit

section .data
  msg: db "hello world", 10 ; note the newline at the end
```

`hello2` - compilation result of `hello2.asm`:

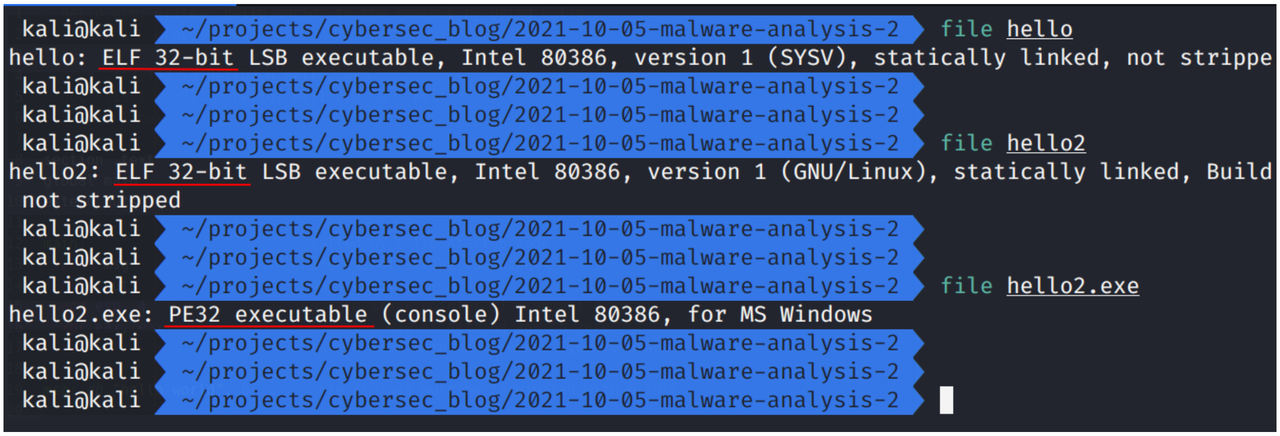
```
; hello.asm: writes "hello world" to the console by using C lib.
; author:    @cocomelonc
; run:
; nasm -f elf32 -o hello2.o hello2.asm
; gcc -static -m32 -o hello2 hello2.o && ./hello2
; 32-bit linux

section .text
global main
extern puts

main:                         ; called by C lib startup code
  push msg                    ; address of string to output
  call puts                   ; puts (msg)
  add esp, 4                  ; update stack pointer (1 argument 4 byte)
  xor eax, eax                ; a faster way of setting eax to zero
  ret                         ; return from main back into C library wrapper

msg: db "hello world", 0      ; note strings must be terminated with 0 in C
```

and `hello2.exe` - cross-compilation result of `hello.c`:



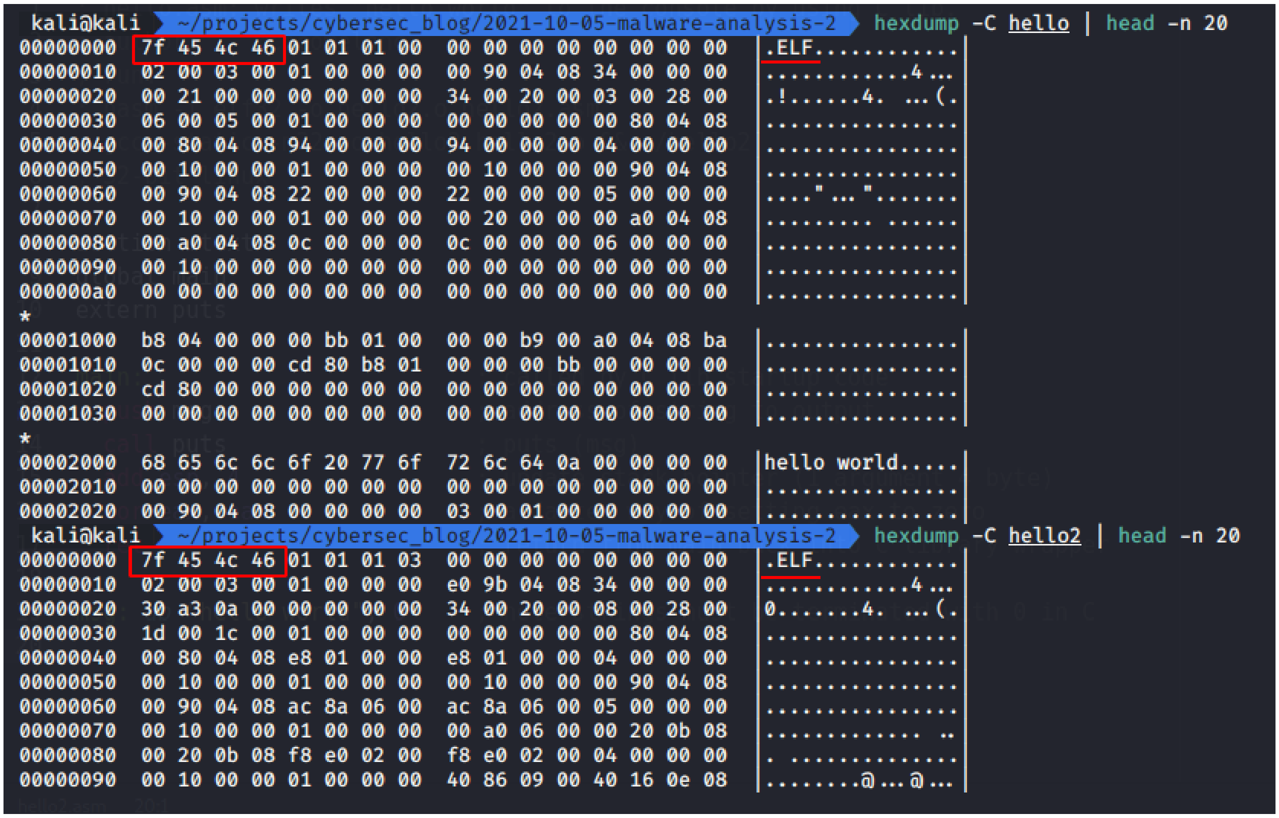
Firstly, run:

```
file hello
file hello2
file hello2.exe
```



Then, run:

```
hexdump -C hello | head 20
hexdump -C hello2 | head 20
```

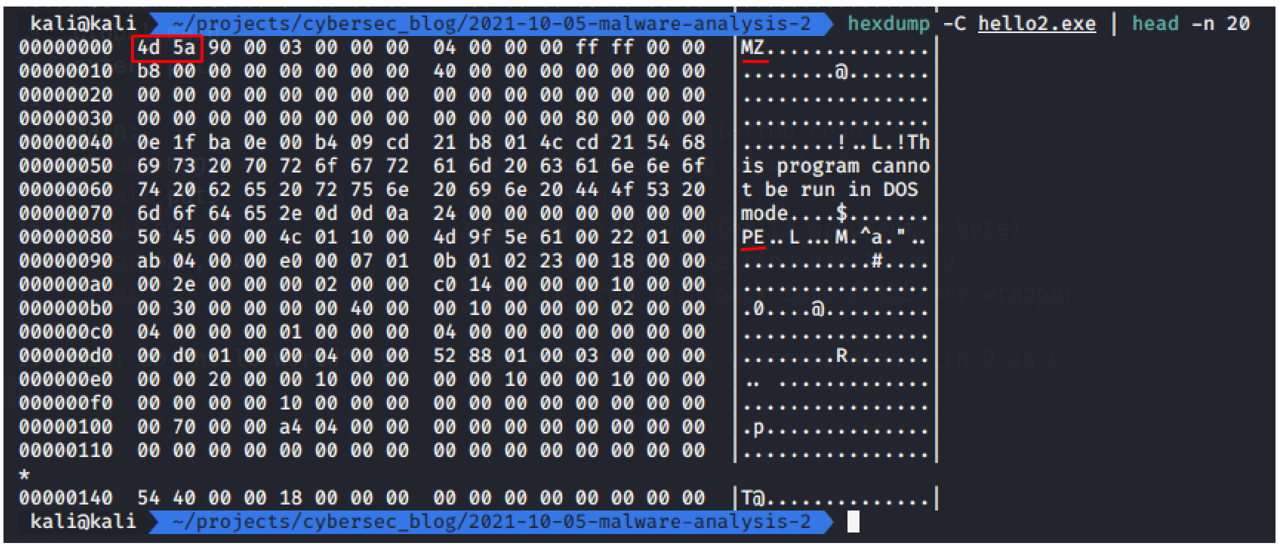


I hope you haven't forgotten that `hello` and `hello2` are ELF (**E**xecutable and **L**inkable **F**ormat) files. What we see here?
As can be seen in this screenshot, the `ELF` header starts with some magic. This ELF header magic provides information about the file. The first 4 hexadecimal parts define that this is an ELF file (**45**=E,**4c**=L,**46**=F), prefixed with the **7f** value.

This ELF header is mandatory. It ensures that data is correctly interpreted during linking or execution. To better understand the inner working of an ELF file, it is useful to know this header information is used.

Let's see an `hello2.exe`:

```
hexdump -C hello2.exe | head 20
```



All the valid PE files contain the value of the first two-byte as **4D** and **5A** (**"MZ"** in ASCII), named after **M**ark **Z**bikowsky, a well-known architect of MS-DOS. Under this header, includes a list of structure.

Also all the valid PE files contain **"PE"** (**P**ortable **E**xecutable).

Then, run:

```
strings -n 6 hello | head
strings -n 6 hello2 | head
strings -n 6 hello2.exe | head
```

As you can see all three files contain “hello world” string.

And then run:

```
objdump -D -M intel hello | head
```

```
kali@kali ~/projects/cybersec_blog/2021-10-05-malware-analysis-2 objdump -D -M intel hello

hello:     file format elf32-i386


Disassembly of section .text:

08049000 <_start>:
 8049000:       b8 04 00 00 00          mov    eax,0x4
 8049005:       bb 01 00 00 00          mov    ebx,0x1
 804900a:       b9 00 a0 04 08          mov    ecx,0x804a000
 804900f:       ba 0c 00 00 00          mov    edx,0xc
 8049014:       cd 80                   int    0x80

08049016 <_exit>:
 8049016:       b8 01 00 00 00          mov    eax,0x1
 804901b:       bb 00 00 00 00          mov    ebx,0x0
 8049020:       cd 80                   int    0x80

Disassembly of section .data:

0804a000 <msg>:
 804a000:       68 65 6c 6c 6f          push   0x6f6c6c65
 804a005:       20 77 6f                and    BYTE PTR [edi+0x6f],dh
 804a008:       72 6c                   jb     804a076 <__bss_start+0x6a>
 804a00a:       64                      fs
 804a00b:       0a                      .byte 0xa
 kali@kali ~/projects/cybersec_blog/2021-10-05-malware-analysis-2
```

then run:

```
objdump -D -M intel hello2 | head
```

```
kali@kali ~/projects/cybersec_blog/2021-10-05-malware-analysis-2 objdump -D -M intel hello2 | head

hello2:     file format elf32-i386


Disassembly of section .note.gnu.build-id:

08048134 <__ehdr_start+0x134>:
 8048134:       04 00                   add    al,0x0
 8048136:       00 00                   add    BYTE PTR [eax],al
 8048138:       14 00                   adc    al,0x0
 kali@kali ~/projects/cybersec_blog/2021-10-05-malware-analysis-2
```

and for exe file, run:

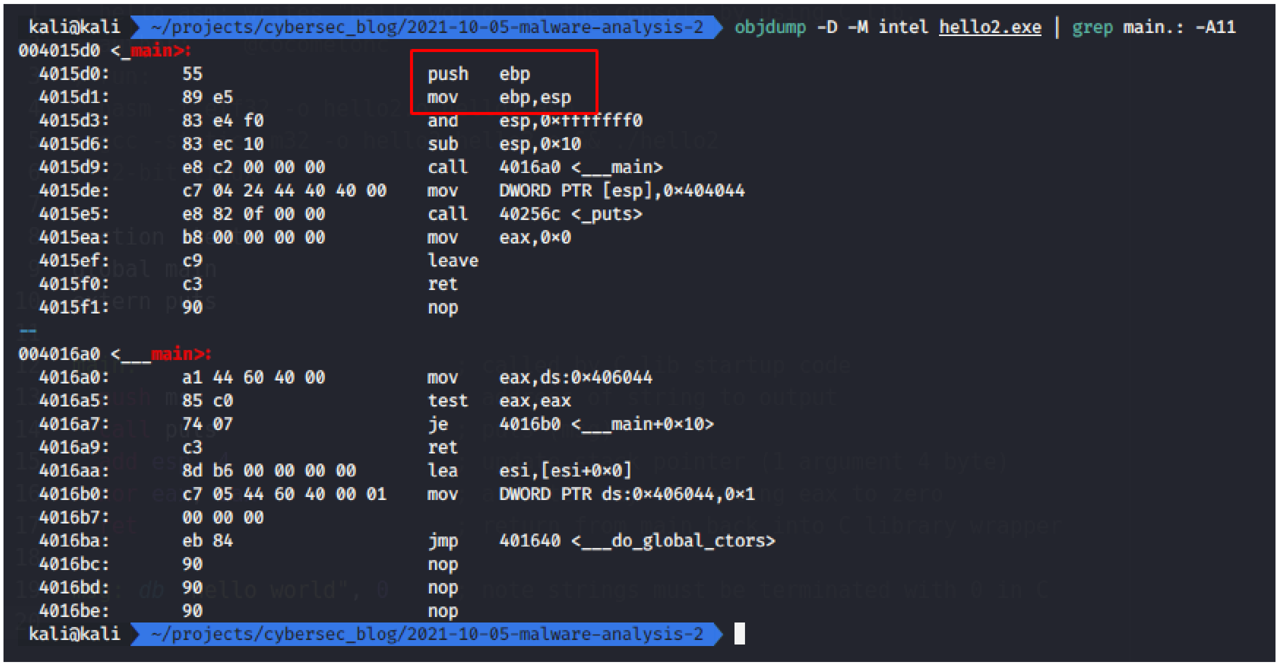
```
objdump -D -M intel hello2.exe | head
```

```
kali@kali ~/projects/cybersec_blog/2021-10-05-malware-analysis-2 objdump -D -M intel hello2.exe | head -n 12

hello2.exe:     file format pei-i386


Disassembly of section .text:

00401000 <___mingw_invalidParameterHandler>:
  401000:       c3                      ret
  401001:       8d b4 26 00 00 00 00    lea    esi,[esi+eiz*1+0x0]
  401008:       8d b4 26 00 00 00 00    lea    esi,[esi+eiz*1+0x0]
  40100f:       90                      nop
```

As you can see in this way you can also understand the file type by its headers.

If you run:

```
objdump -D -M intel hello2.exe | grep main.: -A11
```



I want to draw your attention to these instructions that I indicated in the screenshot. These 2 instructions save the previous base pointer `ebp` and set `EBP` to point at that position on the stack (right below the return address). This sets up `EBP` as a frame pointer.

Some compilers may subtract the required space from the stack pointer after this two instructions, then write each argument directly, see below:

```
push ebp
mov ebp, esp
sub esp, 12   ; if 3 arguments (4*3 bytes)
```

These 3 lines are known as the assembly **function prologue**. Now let's look at an example and you will immediately understand what does it mean. Let's consider this C code:

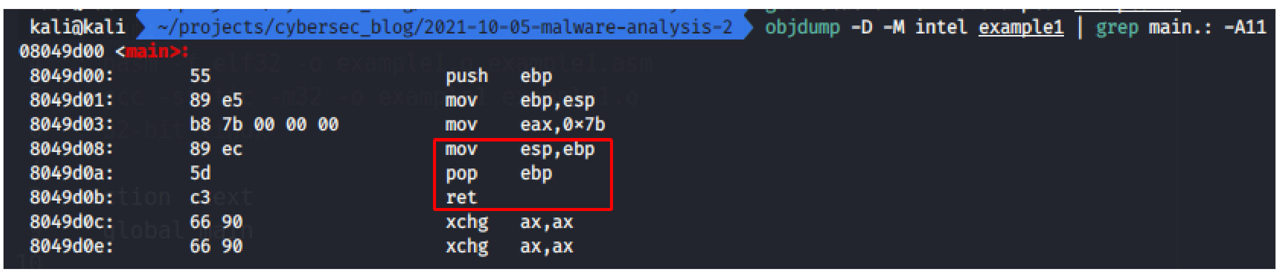
```
#include <stdlib.h>

int main(void) {
    return 123;
}
```

This code in assembler will look like this:

```
; example1.asm
; author: @cocomelonc
; run:
; nasm -f elf32 -o example1.o example1.asm
; gcc -static -m32 -o example1 example1.o
; 32-bit linux

section .text
  global main

main:
  push ebp
  mov ebp, esp
  mov eax, 123
  mov esp, ebp
  pop ebp
  ret

section .data
```

Let’s check. Firstly, compile, then run `objdump`:

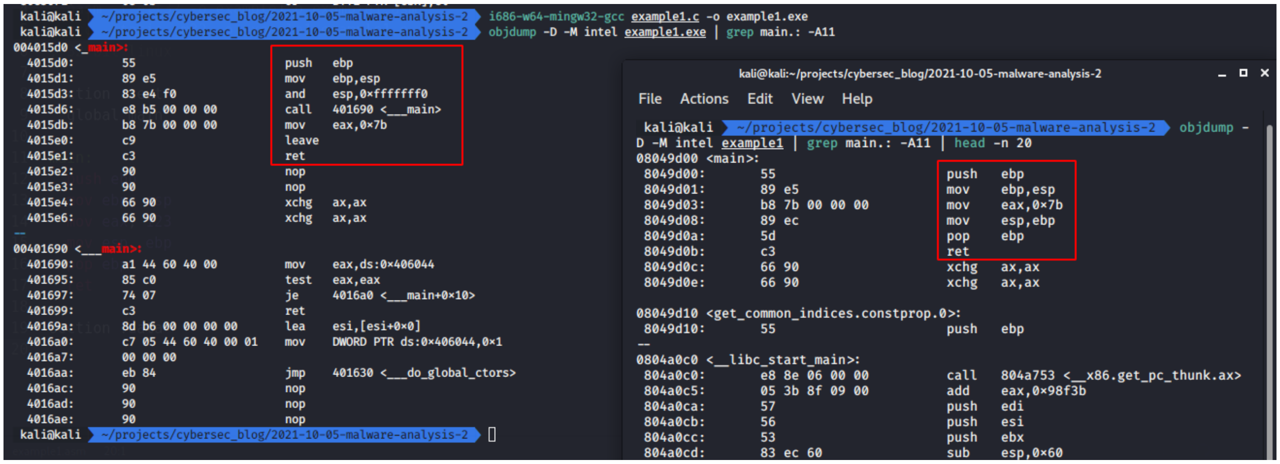
```
nasm -felf32 -o example1.o example1.asm
gcc -static -m32 -o example1 example1.o
objdump -D -M intel example1 | grep main.: -A11
```



I want to draw your attention to these instructions that I indicated in the screenshot. This is called the assembly **function epilogue**. The function epilogue invalidates the allocated stack space, restores the EBP value to the old one, and returns control to the calling function.

If you compile and disassembly C code:

```
i686-w64-mingw32-gcc example1.c -o example1.exe
objdump -D -M intel example1.exe | grep main.: -A11
```



Stop! But we see `leave` instruction. The `leave` instruction does exactly what these two instructions do, and is used by some compilers to save code size. (enter 0,0 is very slow and never used; `leave` is about as efficient as `mov` + `pop`.)

Prologue and epilogue are usually found in disassemblers to separate functions from each other.

memory addressing modes

Let's go to examine another example:

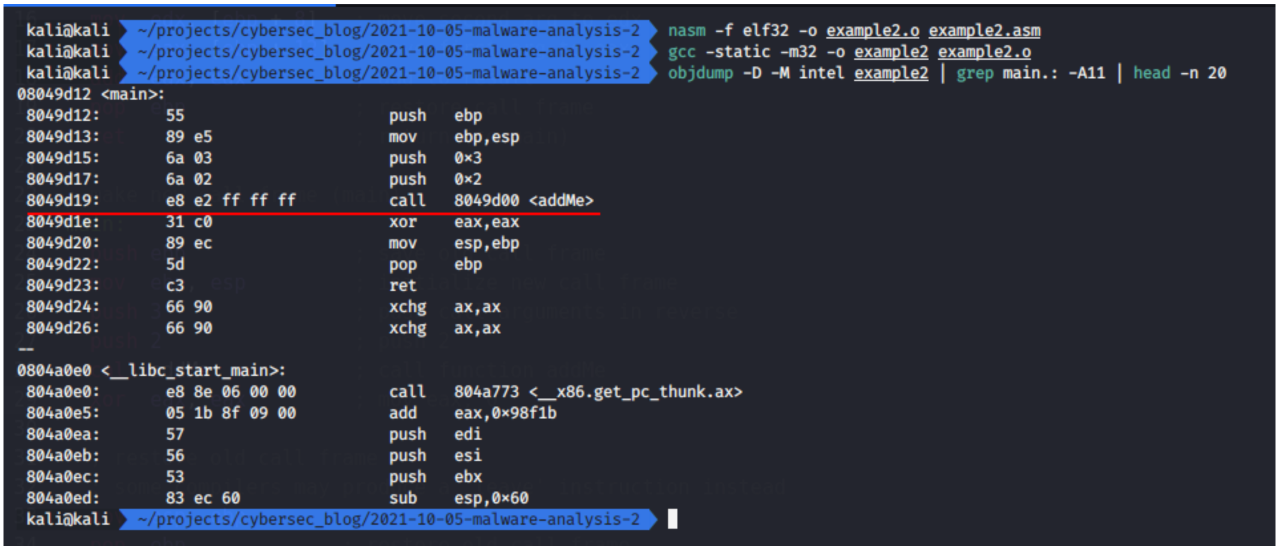
```
#include <stdlib.h>

int addMe(int a, int b) {
    return a + b;
}

int main(void) {
    addMe(2, 3);
    return 0;
}
```

Let's see how it'll be look on x86 assembly language:

```
; example2.asm
; author: @cocomelonc
; run:
; nasm -f elf32 -o example2.o example2.asm
; gcc -static -m32 -o example2 example2.o
; 32-bit linux

section .text
  global main

; make new call frame (addMe)
addMe:
  push ebp              ; save old call frame
  mov  ebp, esp         ; initialize new call frame
  mov  eax, 0           ; move 0 to eax
  mov  edx, [ebp + 8]   ; move second arg to edx
  mov  eax, [ebp + 12]  ; move first arg to eax
  add  eax, edx         ; add to result
  pop  ebp              ; restore call frame
  ret                   ; return (to main)

; make new call frame (main)
main:
  push ebp              ; save old call frame
  mov  ebp, esp         ; initialize new call frame
  push 3                ; push call arguments in reverse
  push 2                ; push 2
  call addMe            ; call function addMe
  xor  eax, eax         ; mov eax, 0

  ; restore old call frame
  ; some compilers may produce a 'leave' instruction instead
  mov  esp, ebp
  pop  ebp             ; restore old call frame
  ret

section .data
```

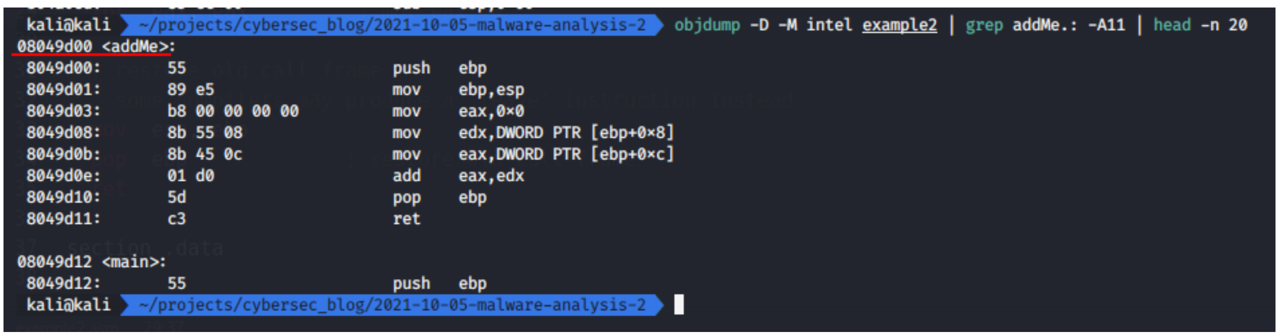
Let’s go to compile and run `objdump`:

```
nasm -f elf32 -o example2.o example2.asm
gcc -static -m32 -o example2 example2.o
objdump -D -M intel example2 | grep main.: -A11 | head -n 20
```



and if we run:

```
objdump -D -M intel example2 | grep addMe.: -A11 | head -n 20
```



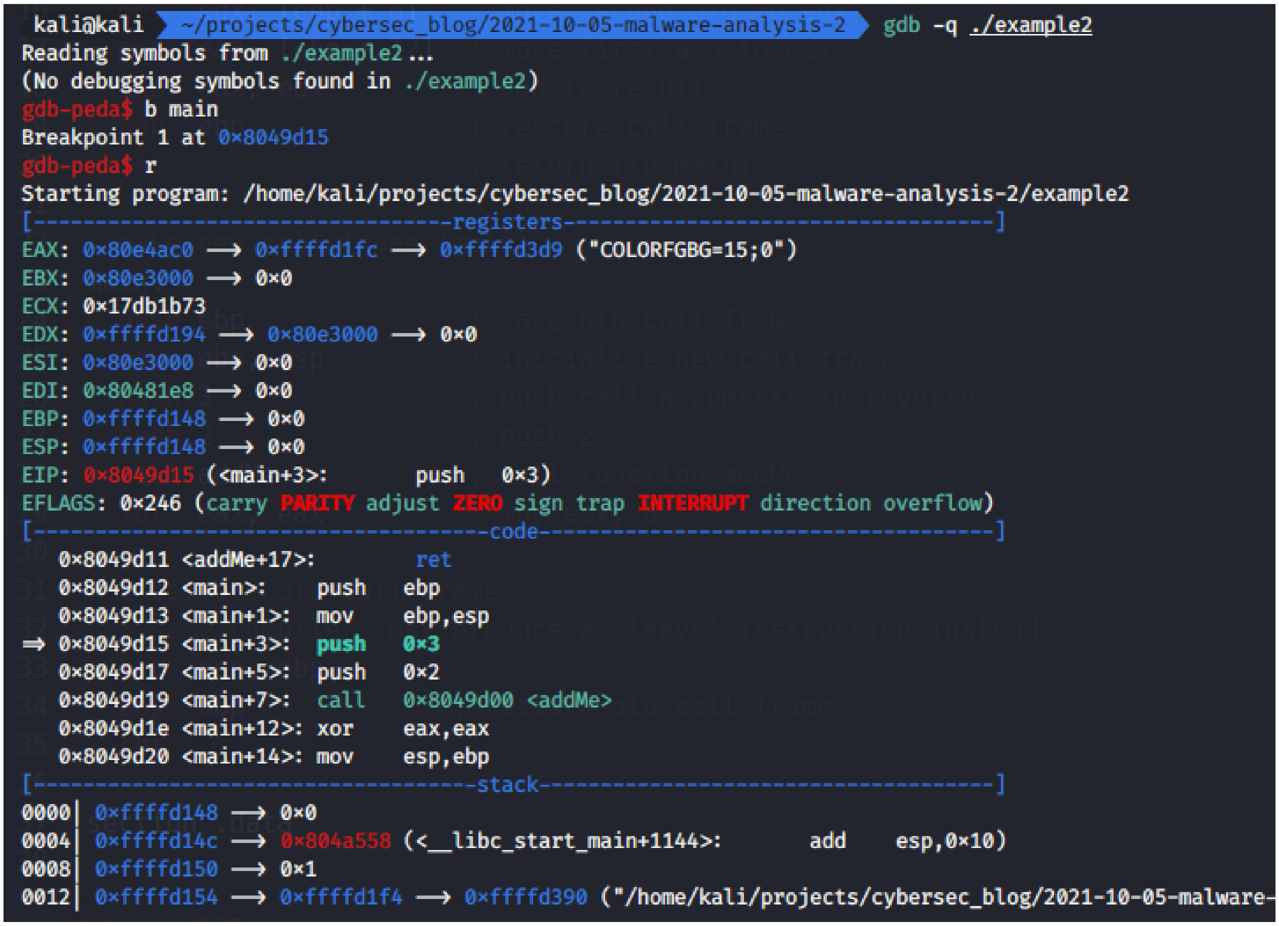
as you can see after insructions:

```
push 3
push 2
```

we go to function `addMe` in address `08049d00`.

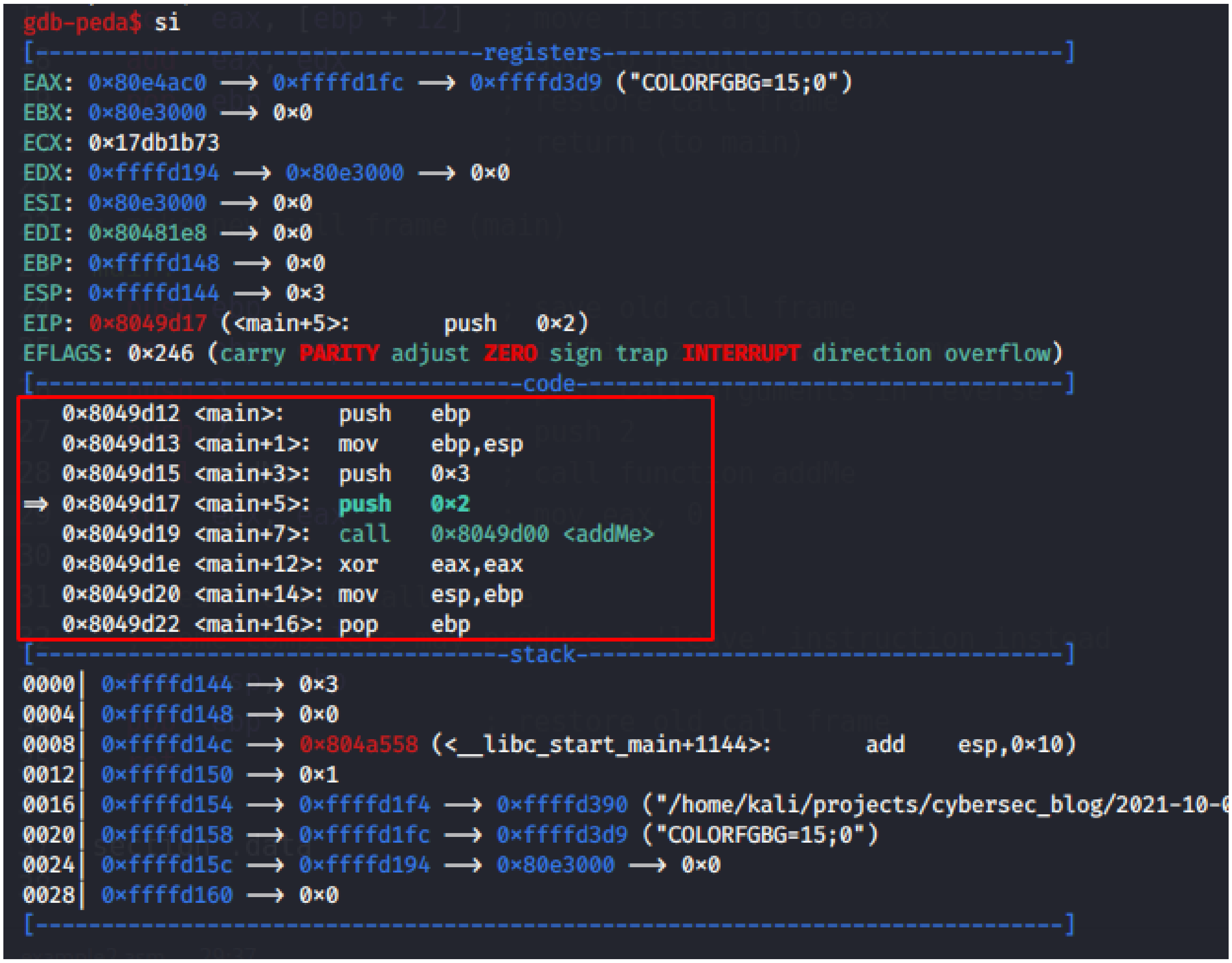
Let’s go to debug with `gdb`:

```
gdb -q ./example2
gdb-peda$ b main
gdb-peda$ r
```



next steps:

```
gdb-peda$ si
gdb-peda$ disas
```



as you can see, push arguments, and we are in function `main` now. Then next steps:

```
gdb-peda$ si
gdb-peda$ disas
```

```
gdb-peda$ si
[----------------------------------registers-----------------------------------]
EAX: 0x80e4ac0 --> 0xffffd1fc --> 0xffffd3d9 ("COLORFGBG=15;0")
EBX: 0x80e3000 --> 0x0
ECX: 0x17db1b73
EDX: 0xffffd194 --> 0x80e3000 --> 0x0
ESI: 0x80e3000 --> 0x0
EDI: 0x80481e8 --> 0x0
EBP: 0xffffd148 --> 0x0
ESP: 0xffffd140 --> 0x2
EIP: 0x8049d19 (<main+7>:       call   0x8049d00 <addMe>)
EFLAGS: 0x246 (carry PARITY adjust ZERO sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x8049d13 <main+1>:  mov    ebp,esp
   0x8049d15 <main+3>:  push   0x3
   0x8049d17 <main+5>:  push   0x2
=> 0x8049d19 <main+7>:  call   0x8049d00 <addMe>
   0x8049d1e <main+12>: xor    eax,eax
   0x8049d20 <main+14>: mov    esp,ebp
   0x8049d22 <main+16>: pop    ebp
   0x8049d23 <main+17>: ret
Guessed arguments:
arg[0]: 0x2
arg[1]: 0x3
arg[2]: 0x0
arg[3]: 0x804a558 (<__libc_start_main+1144>:    add    esp,0x10)
[------------------------------------stack-------------------------------------]
```

and repeat once again:

```
ESP: 0xffffd13c --> 0x8049d1e (<main+12>:       xor    eax,eax)
EIP: 0x8049d00 (<addMe>:        push   ebp)
EFLAGS: 0x246 (carry PARITY adjust ZERO sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x8049cfb <frame_dummy+75>:  xchg   ax,ax
   0x8049cfd <frame_dummy+77>:  xchg   ax,ax
   0x8049cff <frame_dummy+79>:  nop
=> 0x8049d00 <addMe>:   push   ebp
   0x8049d01 <addMe+1>: mov    ebp,esp
   0x8049d03 <addMe+3>: mov    eax,0x0
   0x8049d08 <addMe+8>: mov    edx,DWORD PTR [ebp+0x8]
   0x8049d0b <addMe+11>:        mov    eax,DWORD PTR [ebp+0xc]
[------------------------------------stack-------------------------------------]
0000| 0xffffd13c --> 0x8049d1e (<main+12>:      xor    eax,eax)
0004| 0xffffd140 --> 0x2
0008| 0xffffd144 --> 0x3
0012| 0xffffd148 --> 0x0
0016| 0xffffd14c --> 0x804a558 (<__libc_start_main+1144>:       add    esp,0x10)
0020| 0xffffd150 --> 0x1
0024| 0xffffd154 --> 0xffffd1f4 --> 0xffffd390 ("/home/kali/projects/cybersec_blog/202
0028| 0xffffd158 --> 0xffffd1fc --> 0xffffd3d9 ("COLORFGBG=15;0")
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
0x08049d00 in addMe ()
gdb-peda$
```

and we are call subroutine `addMe`. And a few more steps:

```
[----------------------------------code----------------------------------]
   0x8049d00 <addMe>:    push   ebp
   0x8049d01 <addMe+1>:  mov    ebp,esp
   0x8049d03 <addMe+3>:  mov    eax,0x0
=> 0x8049d08 <addMe+8>:  mov    edx,DWORD PTR [ebp+0x8]
   0x8049d0b <addMe+11>:        mov    eax,DWORD PTR [ebp+0xc]
   0x8049d0e <addMe+14>:        add    eax,edx
   0x8049d10 <addMe+16>:        pop    ebp
   0x8049d11 <addMe+17>:        ret
[---------------------------------stack----------------------------------]
```

we are push arguments and add to result (eax).

The x86-32 instruction set supports using up to four separate components to specify a memory operand. The four components are a fixed displacement value, a base register, an index register, and a scale factor. An effective address is calculated as follows:

effective address = base register + index register * scale factor + displacement

The base register can be any general-purpose register; the index register can be any general-purpose register except `ESP`; Displacement values are constant offsets that are encoded within the instruction; valid scale factors include 1,2,4, and 8. The size of the final effective address is always 32 bits.
For example:

```
mov eax, [MyVal]                ; displacement
mov eax, [ebx]                  ; base register
mov eax, [ebx + 12]             ; base register + displacement
mov eax, [MyArray + esi * 4]    ; displacement + index register * scale factor
mov eax, [ebx + esi]            ; base register + index register
mov eax, [ebx + esi + 12]       ; base register + index register + displacement
mov eax, [ebx + esi * 4]        ; base register + index register * scale factor
mov eax, [ebx + esi * 4 + 20]   ; base register + index register * scale factor +
displacement
```

In our case we push call arguments, in reverse:

```
mov edx, [ebp + 8]
mov eax, [ebp + 12]
add eax, edx
```

If your function has 3 arguments, in reverse:

```
mov  edx, [ebp + 8]   ; move third arg to edx
add  eax, edx         ; add to result
mov  edx, [ebp + 12]  ; move second arg to eax
add  eax, edx         ; add to result
mov  edx, [ebp + 16]  ; move first arg
add  eax, edx         ; add to result
```

If your function has 4 arguments, add:

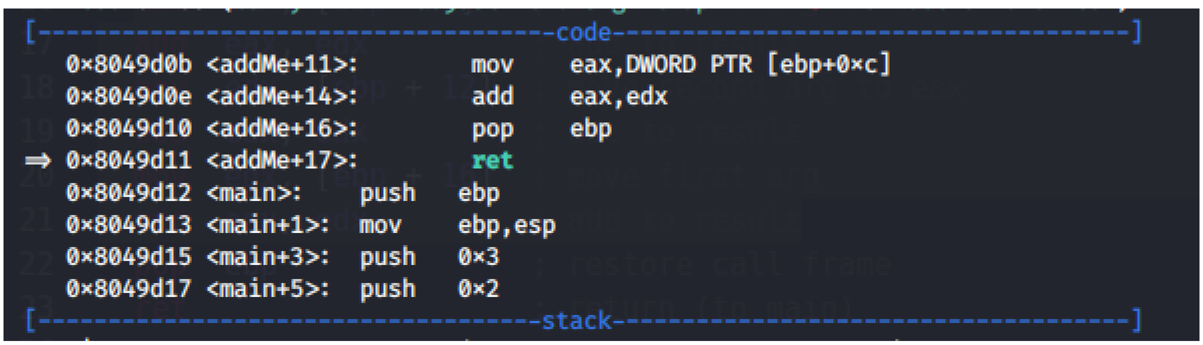
```
mov edx, [ebp + 20]   ; first arg
add eax, edx          ; add to result
; ...
```

etc… I think your got the main idea.

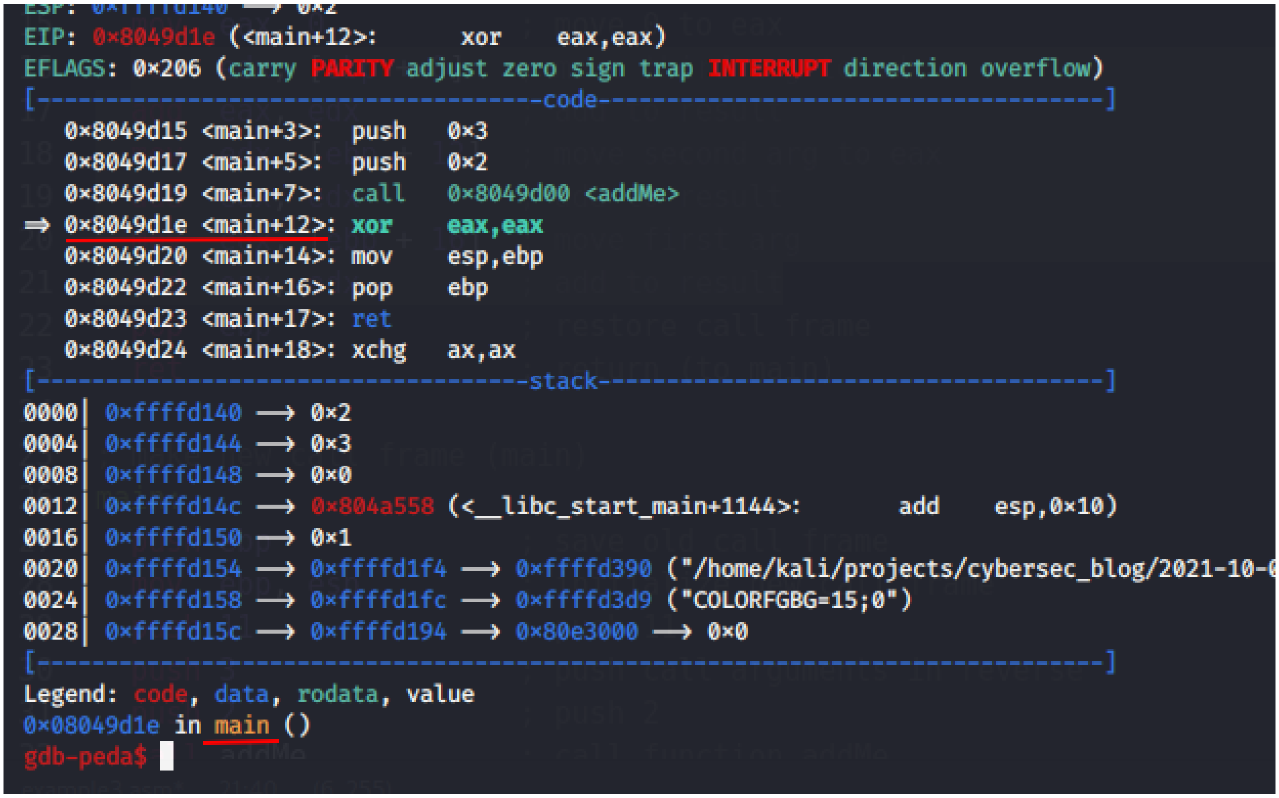
As I wrote earlier, some compilers may subtract the required space from the stack pointer, something like this:

```
sub esp, 16       ; 16 bytes (4 arguments * 4 bytes)
mov edx, [ebp + 8]
add eax, edx
mov edx, [ebp + 12]
add eax, edx
mov edx, [ebp + 16]
add eax, edx
mov edx, [ebp + 20]
add eax, edx
add esp, 16       ; remove call arguments from frame (16 bytes)
```

Continue to examine our debug. And a few more steps:



we are return to function `main(void)`:



```
EIP: 0x8049d1e (<main+12>:      xor    eax,eax)
EFLAGS: 0x206 (carry PARITY adjust zero sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x8049d15 <main+3>:  push   0x3
   0x8049d17 <main+5>:  push   0x2
   0x8049d19 <main+7>:  call   0x8049d00 <addMe>
=> 0x8049d1e <main+12>: xor    eax,eax
   0x8049d20 <main+14>: mov    esp,ebp
   0x8049d22 <main+16>: pop    ebp
   0x8049d23 <main+17>: ret
   0x8049d24 <main+18>: xchg   ax,ax
[------------------------------------stack-------------------------------------]
0000| 0xffffd140 --> 0x2
0004| 0xffffd144 --> 0x3
0008| 0xffffd148 --> 0x0
0012| 0xffffd14c --> 0x804a558 (<__libc_start_main+1144>:       add    esp,0x10)
0016| 0xffffd150 --> 0x1
0020| 0xffffd154 --> 0xffffd1f4 --> 0xffffd390 ("/home/kali/projects/cybersec_blog/2021-10-0
0024| 0xffffd158 --> 0xffffd1fc --> 0xffffd3d9 ("COLORFGBG=15;0")
0028| 0xffffd15c --> 0xffffd194 --> 0x80e3000 --> 0x0
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
0x08049d1e in main ()
gdb-peda$
```

I think now you understand better why we needed to understand stacks. Suppose we have a function `f1` that calls function `f2`, and function `f2`, in turn, calls function `f3`. When the function `f1` is called, it is assigned a certain place on the stack for local data. This space is allocated by subtracting from the `ESP` register a value equal to the size of the required memory. The minimum size of the allocated memory is 4 bytes, i.e. even if the procedure needs 1 byte, it should take 4 bytes.

The `f1` function does some things and then calls the `f2` function. The `f2` function also makes space on the stack by subtracting some value from the `ESP` register. In this case, the local data of the functions `f1` and `f2` are located in different memory areas. Next, the function `f2` calls the function `f3`, which also allocates space for itself on the stack. The `f3` function does not call any other functions and at the end of its work it must free up space on the stack by adding to the `ESP` register the value that was subtracted when the function was called. If the function `f3` does not restore the value of the `ESP` register, then the function `f2`, continuing to work, will not access its data, since it looks for them based on the value of the `ESP` register. Similarly, the function `f2` must restore the value of the `ESP` register upon exiting, which was before its call.

Thus, at the level of procedures, it is necessary to follow the rules for working with the stack - the procedure that took up space on the stack last must free it first. If this rule is not followed, the program will not work correctly. But each procedure can access its own stack area in an arbitrary way. If we were forced to follow the rules for working with the stack inside each procedure, we would have to transfer data from the stack to another memory area, and this would be extremely inconvenient and would extremely slow down the program execution.

Each program has a data area where global variables are located. Why is local data stored on the stack? This is done to reduce the amount of memory occupied by the program. If the program calls several procedures sequentially, then at each moment of time space will be allocated only for the data of one procedure, since the stack is occupied and released. The data area exists all the time the program is running. If local data were located in the data area, it would be necessary to allocate space for local data for all program procedures.

Let’s update our function `addMe`:

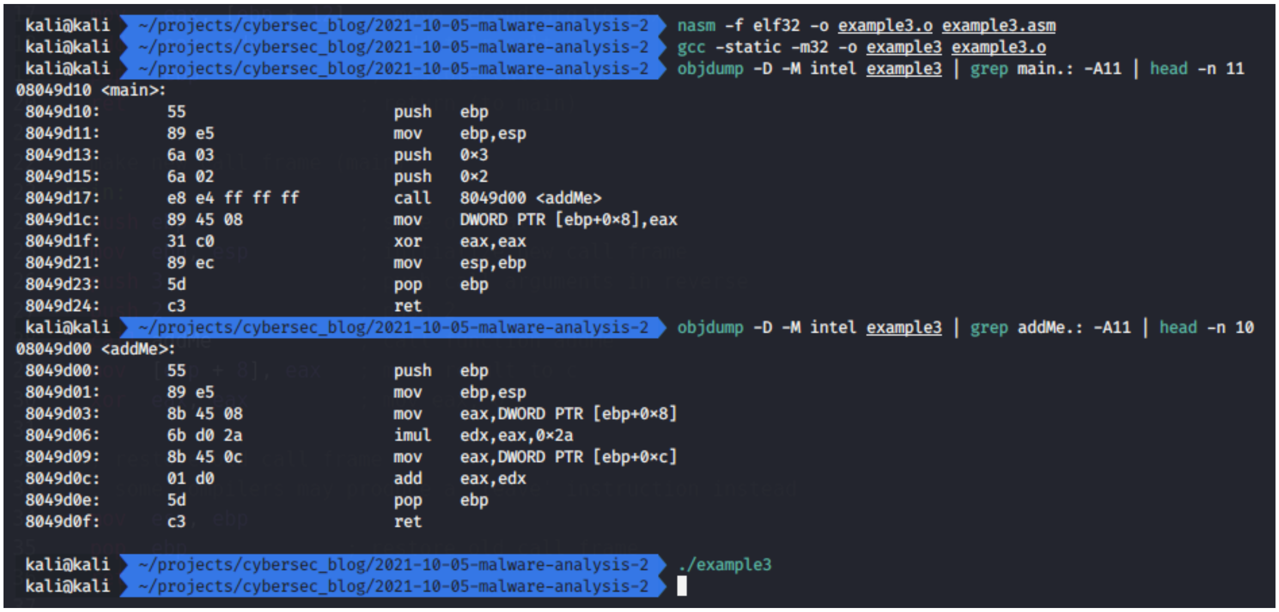
```
#include <stdlib.h>

int addMe(int a, int b) {
  return 42 * a + b;
}

int main(void) {
  int c;
  c = addMe(3, 5);
  return 0;
}
```

which is equivalent this x86 assembly code:

```
; example2.asm
; author: @cocomelonc
; run:
; nasm -f elf32 -o example3.o example3.asm
; gcc -static -m32 -o example3 example3.o
; 32-bit linux

section .text
  global main

; make new call frame (addMe)
addMe:
  push  ebp              ; save old call frame
  mov   ebp, esp         ; initialize new call frame
  mov   eax, [ebp + 8]   ; move a to eax
  imul  edx, eax, 42     ; calculate result
  mov   eax, [ebp + 12]  ; move second arg to eax
  add   eax, edx         ; add to result
  pop   ebp              ; restore call frame
  ret                   ; return (to main)

; make new call frame (main)
main:
  push ebp              ; save old call frame
  mov  ebp, esp         ; initialize new call frame
  push 3                ; push call arguments in reverse
  push 2                ; push 2
  call addMe            ; call function addMe
  mov  [ebp + 8], eax   ; move result to c
  xor  eax, eax         ; mov eax, 0

  ; restore old call frame
  ; some compilers may produce a 'leave' instruction instead
  mov  esp, ebp
  pop  ebp             ; restore old call frame
  ret

section .data
```

let’s go to compile and analyze:

```
nasm -f elf32 -o example3.o example3.asm
gcc -static -m32 -o example3 example3.o
objdump -D -M intel example3 | grep main.: -A11 | head -n 11
objdump -D -M intel example3 | grep addMe.: -A11 | head -n 10
```



As you already understood, the `imul` instruction is used for multiplication.

win32 programming

Ok. Everything is good. But since most malware written for windows, the malware analyst often encounters win32 applications when analyzing.
So, let's go to code win32 example (let's call it `hello3.asm`):

```
; hello3.asm: pop-up "hello world" to the window by using win32 API.
; author:    @cocomelonc
; run:
; nasm -f win32 -o hello3.o hello3.asm
; i686-w64-mingw32-ld -o hello3.exe hello3.o -lkernel32 -luser32
; 32-bit windows

[BITS 32]

section .text
global  _start
extern _MessageBoxA@16
extern  _ExitProcess@4

_start:

  ; MessageBoxA(HWND hWnd, LPCSTR lpText, LPCSTR lpCaption, UINT uType);
  push dword 0            ; push arguments reverse: 0
  push caption            ; push arguments reverse: caption
  push msg                ; push arguments reverse: msg
  push dword 0            ; push arguments reverse: hWnd
  call _MessageBoxA@16    ; call MessageBoxA

  ; ExitProcess(0)
  push dword 0            ; push arguments: 0
  call _ExitProcess@4     ; call ExitProcess

section .data:
  msg: db "hello world", 0
  caption: db "hello", 0
```

This application is simplest, just pop-up message box with `hello world`. Let's examine this code. It uses only plain Win32 system calls from `kernel32.dll`, so it is very instructive to study since it does not make use of a C library. Because system calls from `kernel32.dll` are used, you need to link with an import library. You also have to specify the starting address yourself.

Firstly, we have

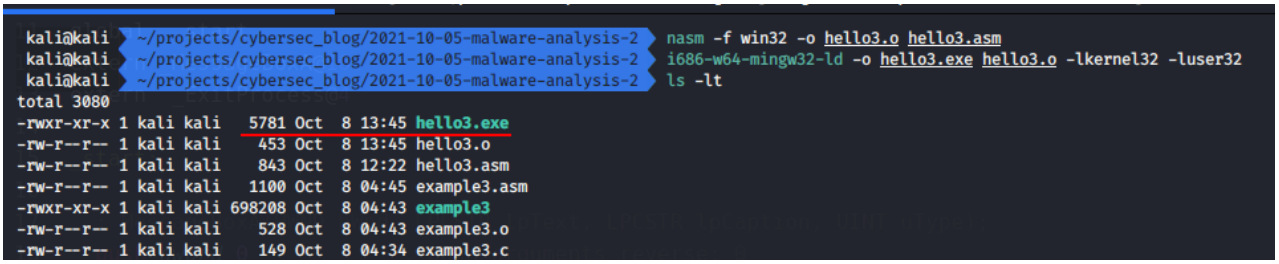
```
extern _MessageBoxA@16
extern  _ExitProcess@4
```

This is external Win32 API functions. The number after `@` is the number of bytes that the function pops from the stack before the function returns. This should be the number of PUSH instructions before the call multiplied by 4. In most cases, this will also be the number of arguments passed to the function multiplied by 4.

Then we push arguments (reverse order) to MessageBoxA, call it, then push arguments (also reverse order) to ExitProcess and call it.

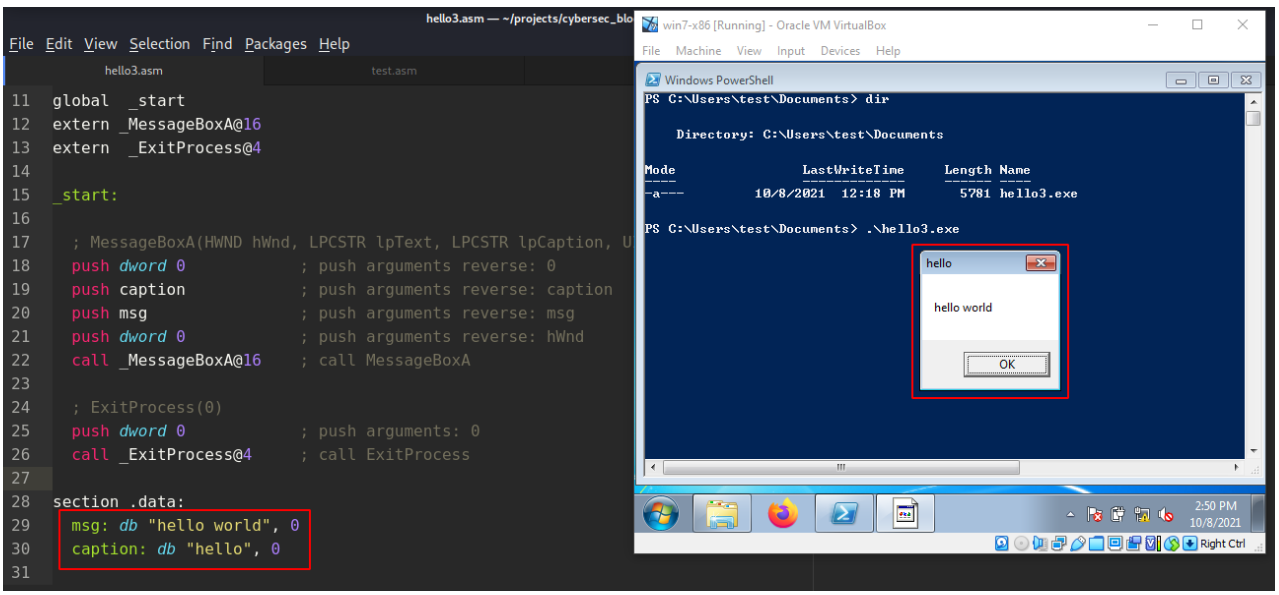
Let’s go to compile:

```
nasm -f win32 -o hello3.o hello3.asm
i686-w64-mingw32-ld -o hello3.exe hello3.o -lkernel32 -luser32
```



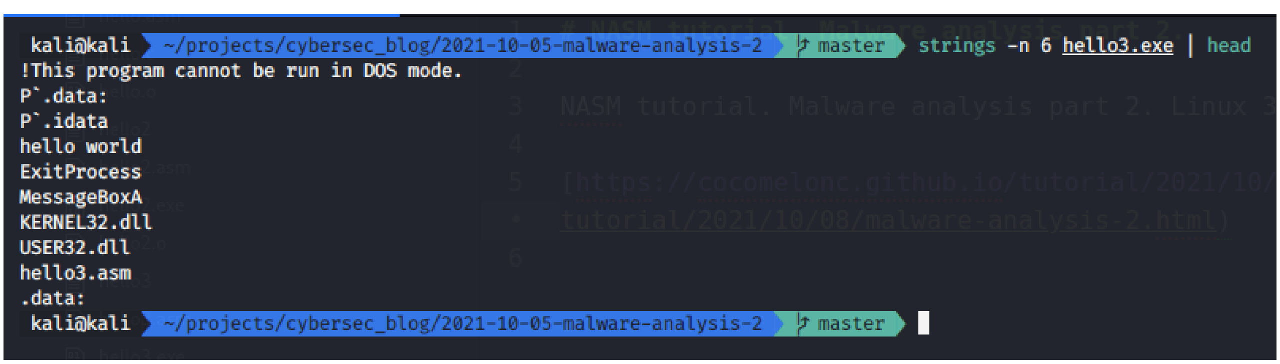
and run:

```
.\hello3.exe
```



If we go to do some static analysis:

```
strings -n 6 hello3.exe | head
```




```
hexdump -D hello3.exe | head -n 64
```

```
kali@kali  ~/projects/cybersec_blog/2021-10-05-malware-analysis-2  master  hexdump -C hello3.exe | head -n 64
00000000  4d 5a 90 00 03 00 00 00  04 00 00 00 ff ff 00 00  |MZ..............|
00000010  b8 00 00 00 00 00 00 00  40 00 00 00 00 00 00 00  |........@.......|
00000020  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000030  00 00 00 00 00 00 00 00  00 00 00 00 80 00 00 00  |................|
00000040  0e 1f ba 0e 00 b4 09 cd  21 b8 01 4c cd 21 54 68  |........!..L.!Th|
00000050  69 73 20 70 72 6f 67 72  61 6d 20 63 61 6e 6e 6f  |is program canno|
00000060  74 20 62 65 20 72 75 6e  20 69 6e 20 44 4f 53 20  |t be run in DOS |
00000070  6d 6f 64 65 2e 0d 0d 0a  24 00 00 00 00 00 00 00  |mode....$.......|
00000080  50 45 00 00 4c 01 03 00  1b f7 5f 61 00 0a 00 00  |PE..L....._a....|
00000090  78 00 00 00 e0 00 07 03  0b 01 02 23 00 04 00 00  |x..........#....|
000000a0  00 02 00 00 00 00 00 00  00 10 00 00 00 10 00 00  |................|
000000b0  00 00 00 00 00 00 40 00  00 10 00 00 00 02 00 00  |......@.........|
000000c0  04 00 00 00 01 00 00 00  04 00 00 00 00 00 00 00  |................|
000000d0  00 40 00 00 00 04 00 00  85 9c 00 00 03 00 00 00  |.@..............|
000000e0  00 00 20 00 00 10 00 00  00 00 10 00 00 10 00 00  |.. .............|
000000f0  00 00 00 00 10 00 00 00  00 00 00 00 00 00 00 00  |................|
00000100  00 30 00 00 9c 00 00 00  00 00 00 00 00 00 00 00  |.0..............|
00000110  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000150  00 00 00 00 00 00 00 00  4c 30 00 00 10 00 00 00  |........L0......|
00000160  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000170  00 00 00 00 00 00 00 00  2e 74 65 78 74 00 00 00  |.........text...|
00000180  3c 00 00 00 00 10 00 00  00 02 00 00 00 04 00 00  |<...............|
00000190  00 00 00 00 00 00 00 00  00 00 00 00 20 00 50 60  |............ .P`|
000001a0  2e 64 61 74 61 3a 00 00  12 00 00 00 00 20 00 00  |.data:....... ..|
000001b0  00 02 00 00 00 06 00 00  00 00 00 00 00 00 00 00  |................|
000001c0  00 00 00 00 20 00 50 60  2e 69 64 61 74 61 00 00  |.... .P`.idata..|
000001d0  9c 00 00 00 00 30 00 00  00 02 00 00 00 08 00 00  |.....0..........|
000001e0  00 00 00 00 00 00 00 00  00 00 00 00 40 00 30 c0  |............@.0.|
000001f0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
```

and then:

```
objdump -D -M intel hello3.exe | head -n 32
```

```
kali@kali  ~/projects/cybersec_blog/2021-10-05-malware-analysis-2  master  objdump -D -M intel hello3.exe | head -n 32

hello3.exe:     file format pei-i386


Disassembly of section .text:

00401000 <__rt_psrelocs_end>:
  401000:       6a 00                   push   0x0
  401002:       68 0c 20 40 00          push   0x40200c
  401007:       68 00 20 40 00          push   0x402000
  40100c:       6a 00                   push   0x0
  40100e:       e8 11 00 00 00          call   401024 <_MessageBoxA@16>
  401013:       6a 00                   push   0x0
  401015:       e8 02 00 00 00          call   40101c <_ExitProcess@4>
  40101a:       66 90                   xchg   ax,ax

0040101c <_ExitProcess@4>:
  40101c:       ff 25 4c 30 40 00       jmp    DWORD PTR ds:0x40304c
  401022:       90                      nop
  401023:       90                      nop

00401024 <_MessageBoxA@16>:
  401024:       ff 25 54 30 40 00       jmp    DWORD PTR ds:0x403054
  40102a:       90                      nop
  40102b:       90                      nop
```

Sometimes, in order to understand what a particular function does, you don’t have to disassemble it, but just look at its inputs and outputs. This way you can save time. But at the same time you still have to look inside.

I will write about this in the next post and I will try to consider real examples of simple malware.

I will write malware in C/C++ like in this, this or this post and then analyze it.

I hope this post was useful for entry level malware analysts or red team members like me, who want to develop skills in the art of reverse engineering.

Reverse engineering for beginners
CS5138 free course materials
Practical Malware Analysis Book
GDB
pefile
intel 64 and IA-32 arch software developer’s manual
Source code in Github

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