Task 1 Printing on the Screen [30%]

(a) In print_stk.asm, explain how the line “push 0x000a0d21” works. Show a screenshot from gdb to support your explanation.

(1) This push instruction first decrements esp by 4 (from 0xbffff72c to 0xbffff728)

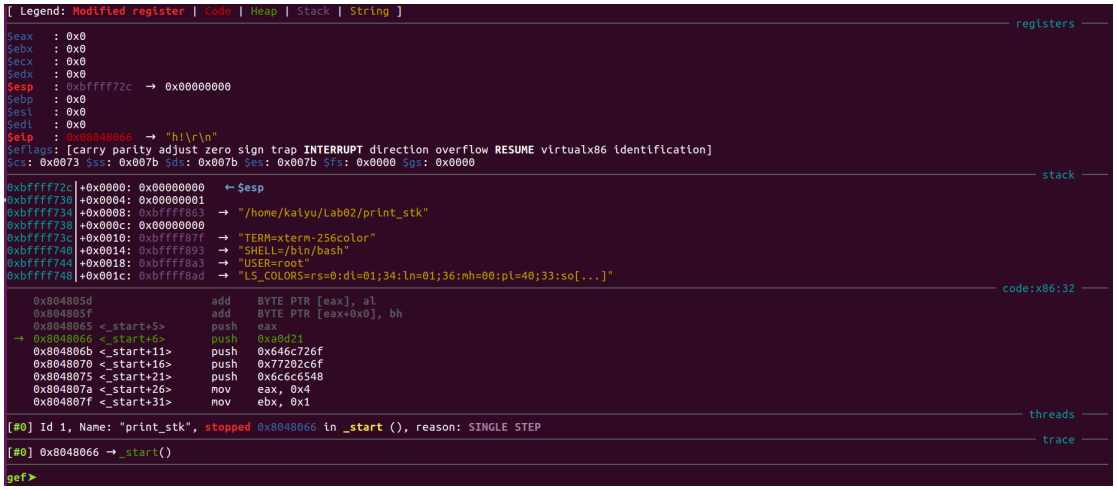
(2) Then it places its operand "0x000a0d21" into the contents of the 32-bit location at address [esp] (0xbffff728)

Actually, the single instruction "push 0x000a0d21" is equivalent to these two instructions:

sub esp, 4

mov dword [esp], 0x000a0d21

A screenshot before executing the instruction "push 0x000a0d21" :



A screenshot after executing the instruction "push 0x000a0d21" :

(b) Also, in the same file, explain how you got the string address. Show a screenshot from gdb to support your explanation.

Because the string "Hello, world!\r\n" is stored on the stack, we can get the string address by copying the value of the **esp** register into the **ecx** register.

Below is the screenshot before executing the instruction "mov ecx, esp".

We can see that now the **esp** register points the string "Hello, world!\r\n", which means the **esp** register now holds the address of the string.

So, we can copy the address the value of the **esp** register into the **ecx** register to get the address of the string "Hello, world!\r\n".

And this is the screenshot after executing the instruction "mov ecx, esp":

Task 2 Spawning a Shell [70%]

Startup Code (labsh.asm) [10%]

The process number of the **calling shell** is **2261**, and the process number of the **spawned shell** is **2456**.

The passed environment variable to the spawned shell is **"PWD=/home/kaiyu/Lab02"**.

Below is a screenshot of a successful run:

```
root@kaiyu:/home/kaiyu/Lab02# nasm -f elf labsh.s
root@kaiyu:/home/kaiyu/Lab02# ld -o labsh labsh.o
root@kaiyu:/home/kaiyu/Lab02# ./labsh
# echo $$
2456
# /usr/bin/env
PWD=/home/kaiyu/Lab02
# exit
root@kaiyu:/home/kaiyu/Lab02# echo $$
2261
root@kaiyu:/home/kaiyu/Lab02#
```

Providing Arguments to `/bin/sh` [20%]

Below is a screenshot of a successful run:

```
root@kaiyu:/home/kaiyu/Lab02# ./labsh_args
total 124
drwxr-xr-x  2 root  root  4096 Jan 15 21:58 .
drwxr-xr-x 19 kaiyu kaiyu 4096 Jan 16 13:40 ..
-rw-r--r--  1 root  root   289 Jan 12 14:42 code.c
-rwxr-xr-x  1 root  root   648 Jan 12 14:36 helloworld
-rw-r--r--  1 root  root   624 Jan 12 14:36 helloworld.o
-rw-r--r--  1 root  root   388 Jan 12 14:35 helloworld.s
-rwxr-xr-x  1 root  root   512 Jan 13 17:14 labsh
-rw-r--r--  1 root  root   448 Jan 13 17:14 labsh.o
-rw-r--r--  1 root  root   630 Jan 13 16:53 labsh.s
-rwxr-xr-x  1 root  root   548 Jan 15 13:27 labsh_args
-rw-r--r--  1 root  root   496 Jan 15 13:27 labsh_args.o
-rw-r--r--  1 root  root   896 Jan 15 13:27 labsh_args.s
-rwxr-xr-x  1 root  root   576 Jan 15 16:25 labsh_env
-rw-r--r--  1 root  root   512 Jan 15 16:25 labsh_env.o
-rw-r--r--  1 root  root  1097 Jan 15 16:25 labsh_env.s
-rwxr-xr-x  1 root  root   576 Jan 15 19:51 labsh_rel
-rw-r--r--  1 root  root   512 Jan 15 19:51 labsh_rel.o
-rw-r--r--  1 root  root   545 Jan 15 19:51 labsh_rel.s
-rwxr-xr-x  1 root  root   488 Jan 12 10:48 mini
-rw-r--r--  1 root  root   432 Jan 12 10:48 mini.o
-rw-r--r--  1 root  root    77 Jan 12 10:45 mini.text
-rwxr-xr-x  1 root  root  7344 Jan 12 14:40 mini_shelltest
-rwxr-xr-x  1 root  root   604 Jan 12 15:17 print_rel
-rw-r--r--  1 root  root   544 Jan 12 15:17 print_rel.o
-rw-r--r--  1 root  root   610 Jan 15 12:49 print_rel.s
-rwxr-xr-x  1 root  root   540 Jan 15 12:49 print_stk
-rw-r--r--  1 root  root   480 Jan 15 12:49 print_stk.o
-rw-r--r--  1 root  root   589 Jan 15 12:49 print_stk.s
-rwxr-xr-x  1 root  root  7404 Jan 12 14:43 shelltest
```

A screenshot of running the command "ls -la" in the same path:

```
root@kaiyu:/home/kaiyu/Lab02# ls -la
total 124
drwxr-xr-x  2 root  root  4096 Jan 15 21:58 .
drwxr-xr-x 19 kaiyu kaiyu 4096 Jan 16 13:40 ..
-rw-r--r--  1 root  root   289 Jan 12 14:42 code.c
-rwxr-xr-x  1 root  root   648 Jan 12 14:36 helloworld
-rw-r--r--  1 root  root   624 Jan 12 14:36 helloworld.o
-rw-r--r--  1 root  root   388 Jan 12 14:35 helloworld.s
-rwxr-xr-x  1 root  root   512 Jan 13 17:14 labsh
-rwxr-xr-x  1 root  root   548 Jan 15 13:27 labsh_args
-rw-r--r--  1 root  root   496 Jan 15 13:27 labsh_args.o
-rw-r--r--  1 root  root   896 Jan 15 13:27 labsh_args.s
-rwxr-xr-x  1 root  root   576 Jan 15 16:25 labsh_env
-rw-r--r--  1 root  root   512 Jan 15 16:25 labsh_env.o
-rw-r--r--  1 root  root  1097 Jan 15 16:25 labsh_env.s
-rw-r--r--  1 root  root   448 Jan 13 17:14 labsh.o
-rwxr-xr-x  1 root  root   576 Jan 15 19:51 labsh_rel
-rw-r--r--  1 root  root   512 Jan 15 19:51 labsh_rel.o
-rw-r--r--  1 root  root   545 Jan 15 19:51 labsh_rel.s
-rw-r--r--  1 root  root   630 Jan 13 16:53 labsh.s
-rwxr-xr-x  1 root  root   488 Jan 12 10:48 mini
-rw-r--r--  1 root  root   432 Jan 12 10:48 mini.o
-rwxr-xr-x  1 root  root  7344 Jan 12 14:40 mini_shelltest
-rw-r--r--  1 root  root    77 Jan 12 10:45 mini.text
-rwxr-xr-x  1 root  root   604 Jan 12 15:17 print_rel
-rw-r--r--  1 root  root   544 Jan 12 15:17 print_rel.o
-rw-r--r--  1 root  root   610 Jan 15 12:49 print_rel.s
-rwxr-xr-x  1 root  root   540 Jan 15 12:49 print_stk
-rw-r--r--  1 root  root   480 Jan 15 12:49 print_stk.o
-rw-r--r--  1 root  root   589 Jan 15 12:49 print_stk.s
-rwxr-xr-x  1 root  root  7404 Jan 12 14:43 shelltest
```

Providing Env. Variables to `/bin/sh` [20%]

Below is a screenshot of a successful run:

```
root@kaiyu:/home/kaiyu/Lab02# vim labsh_env.s
root@kaiyu:/home/kaiyu/Lab02# nasm -f elf labsh_env.s
root@kaiyu:/home/kaiyu/Lab02# ld -o labsh_env labsh_env.o
root@kaiyu:/home/kaiyu/Lab02# ./labsh_env
# /usr/bin/env
bbbb=5678
aaaa=1234
cccc=1234
PWD=/home/kaiyu/Lab02
#
```

Using the Relative Addressing Technique [20%]

To set the text and data sections to be readable and writable, we should use the command:

ld --**omagic** labsh_rel.o -o labsh_rel,

If we didn't add the --omagic option, it will report a segmentation fault.

Below is a screenshot of a successful run:

```
root@kaiyu:/home/kaiyu/Lab02# nasm -f elf labsh_rel.s
root@kaiyu:/home/kaiyu/Lab02# ld --omagic labsh_rel.o -o labsh_rel
root@kaiyu:/home/kaiyu/Lab02# ./labsh_rel
# echo $$
3576
# exit
root@kaiyu:/home/kaiyu/Lab02# echo $$
2246
root@kaiyu:/home/kaiyu/Lab02#
```

A detailed explanation for each line of the code: ()

```
section .text
global _start
_start:
        jmp two
one:
        pop esi
        ; esi now holds address of the string "/bin/sh*AAAABBBB"
        mov ebx, esi
        ; (complete) ebx should contain the string address
        mov eax, 0
        mov byte [ebx+7], 0x00
        ; (complete) terminate /bin/sh with 0x00 (1 byte)
        ; ebx now holds the address of the string "/bin/sh"

        ; start to construct the array argv[]
        mov [ebx+8], ebx
        ; (complete) save ebx to memory at address ebx+8
        ; replace AAAA with the address of the string "/bin/sh"
        mov [ebx+12], eax
        ; (complete) save eax to memory at address ebx+12
        ; replace BBBB with NULL bytes
        lea ecx, [ebx+8]
        ; let ecx = ebx + 8
        ; now ecx points to the array argv[]
        ; argv[0] = NULL
        ; argv[1] = the address of string "/bin/sh"

        ; For environment variable
        mov edx, 0
        ; No env variables

        ; Call execve()
        mov al, 0x0b
        ; eax = 0x0000000b
        int 0x80
two:
```

```
        call one
        ; now the string '/bin/sh*AAAABBBB' is stored on the stack
        db '/bin/sh*AAAABBBB'
```

Why this code would successfully execute the `/bin/sh` program?

How the `argv` array is constructed?

(1) We know that the system call interface: sys_execve() is:

```
asmlinkage long sys_execve(const char __user *filename,
                           const char __user *const __user *argv,
                           const char __user *const __user *envp);
```

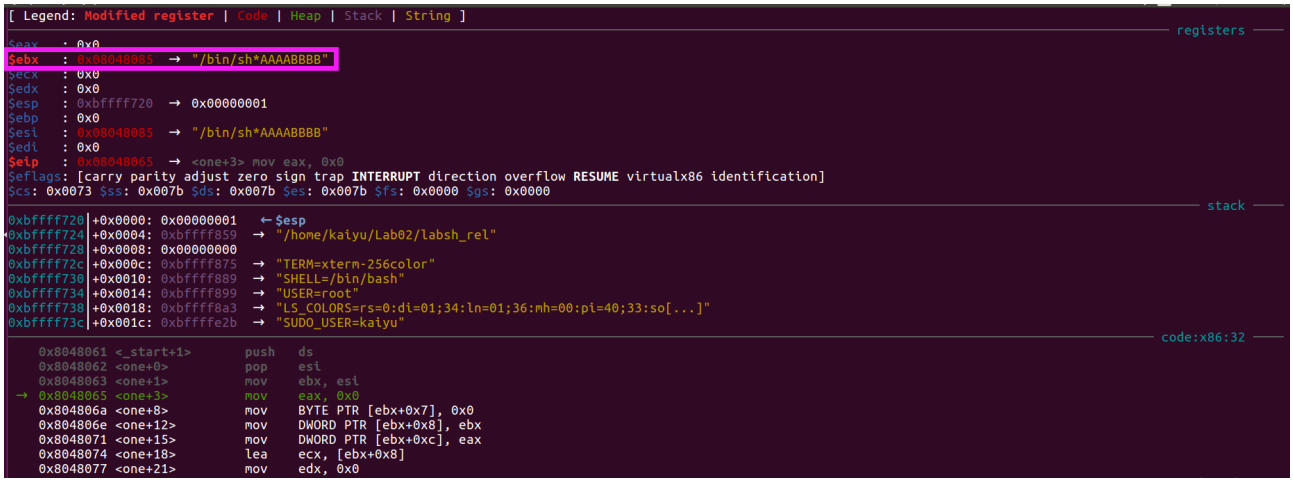
The first parameter filename should be stored in ebx register, the second parameter argv should be stored in ecx register, and the third parameter envp should be stored in edx register.

(2) To execute the `/bin/sh` program, we should move address of string "/bin/sh0" into ebx, move address of the address of "/bin/sh0" into ecx, and move address of NULL word into edx.
So, we need the null terminated string /bin/sh somewhere in memory, and we need the address of the string /bin/sh somewhere in memory, and we need we need a NULL word somewhere in memory.

(3) By using the Relative Addressing Technique, we first use jmp instruction at beginning of shelldcode to jump to the call instruction, which is right before the string '/bin/sh*AAAABBBB'.
Then we use call instruction to jump back to first instruction after jmp.
Now the address of the string '/bin/sh*AAAABBBB' is on the stack.

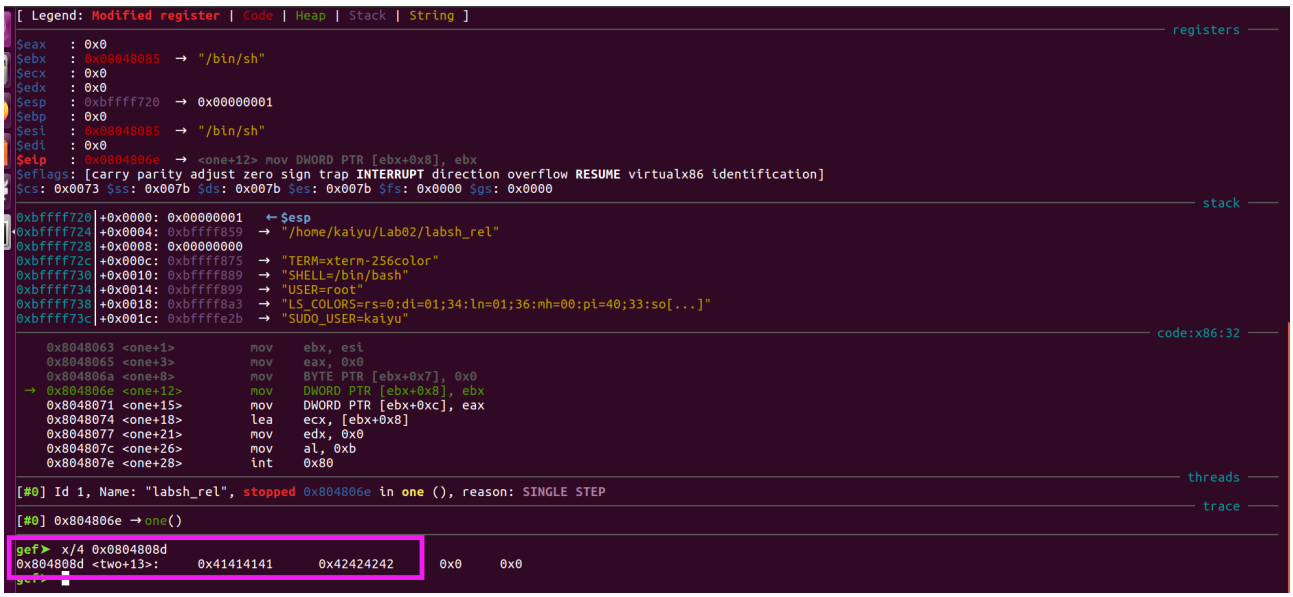
(4) Then we use the instruction "pop esi", now esi holds the address of the string '/bin/sh*AAAABBBB'.

(5) Then we use instruction "mov ebx, esi", now ebx holds the address of the string '/bin/sh*AAAABBBB'.

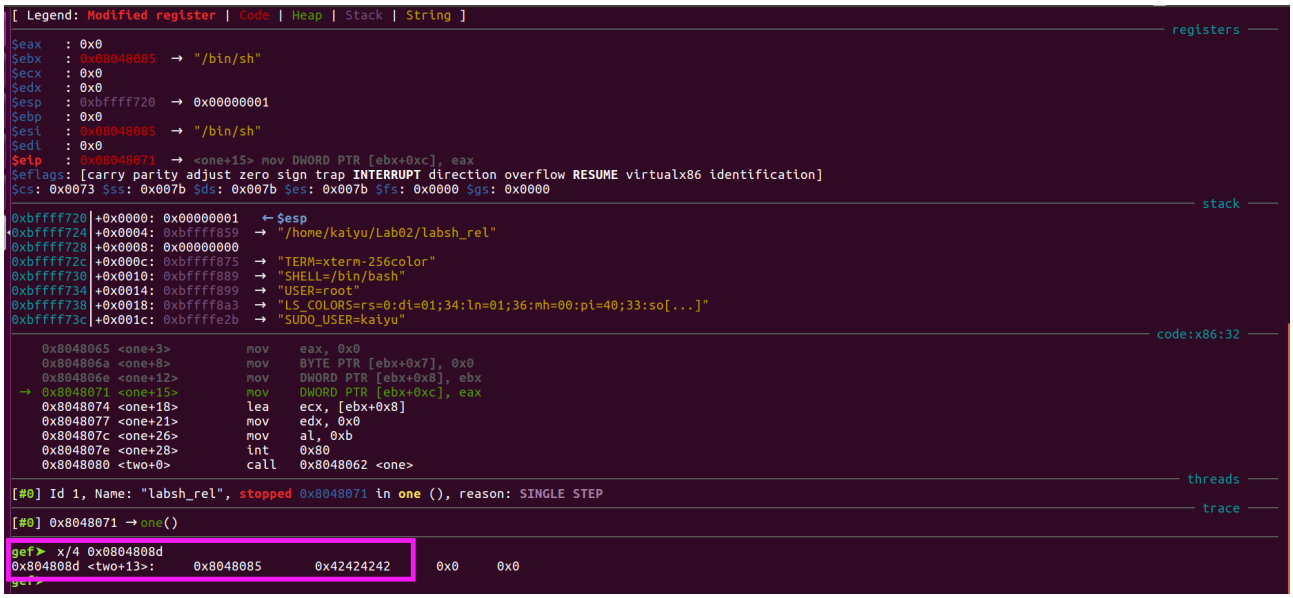


(6) Then we use the instruction "mov byte [ebx+7], 0x00" to replace the "*" of the string "/bin/sh*AAAABBBB" with a zero byte. Now ebx holds the address of the string "/bin/sh".

The whole string (include zero byte at the middle) in the memory is "/bin/sh0AAAABBBB".



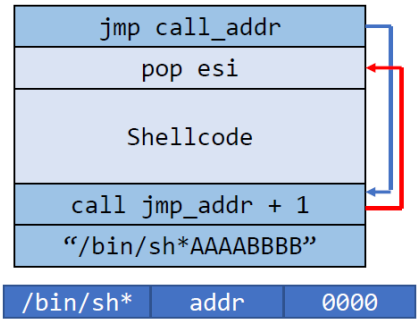
(7) Then we start to construct the array argv[]. We use the instruction "mov [ebx+8], ebx" to replace "AAAA" of the string "/bin/sh0AAAABBBB" with the address of string "/bin/sh". The whole string (include zero byte at the middle) in the memory is "/bin/sh0'addr'BBBB".



There is a mistake in the page 6 of "lab02_explainer.pdf":

Activity 2: Spawn a new Shell

- A startup code is provided that uses relative addressing, you need to:
  - Complete the missing parts
  - Answer few questions
- You need to replace:
  - * with a NULL byte
  - AAAA with the address of the address of string
  - BBBB with NULL bytes
  - Why cannot we start with /bin/sh0AAAA0000?
- Can a program modify the code segment?
  - How can we solve this issue?



We need to replace "AAAA" with the address of string "/bin/sh" instead of the address of address of string "/bin/sh".

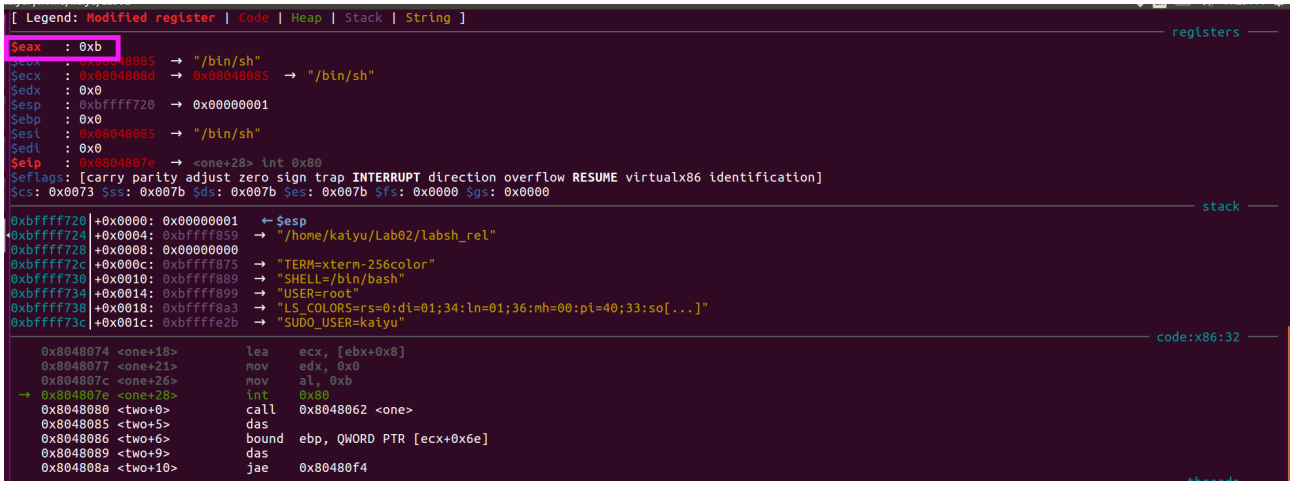
(8) Then we use the instruction "mov [ebx+12], eax" to replace "BBBB" of the string "/bin/sh0'addr'BBBB" with NULL bytes.
Now we finished constructing the argv[]. The address of the array argv[] is ebx+8.
argv[0] = NULL
argv[1] = the address of string "/bin/sh"

```
[ Legend: Modified register | Code | Heap | Stack | String ]
registers
$eax   : 0x0
$ebx   : 0x08048085  →  "/bin/sh"
$ecx   : 0x0
$edx   : 0x0
$esp   : 0xbffff720  →  0x00000001
$ebp   : 0x0
$esi   : 0x08048085  →  "/bin/sh"
$edi   : 0x0
$eip   : 0x08048074  →  <one+18> lea ecx, [ebx+0x8]
$eflags: [carry parity adjust zero sign trap INTERRUPT direction overflow RESUME virtualx86 identification]
$cs: 0x0073 $ss: 0x007b $ds: 0x007b $es: 0x007b $fs: 0x0000 $gs: 0x0000
stack
0xbffff720|+0x0000: 0x00000001   ←$esp
0xbffff724|+0x0004: 0xbffff859   →  "/home/kaiyu/Lab02/labsh_rel"
0xbffff728|+0x0008: 0x00000000
0xbffff72c|+0x000c: 0xbffff875   →  "TERM=xterm-256color"
0xbffff730|+0x0010: 0xbffff889   →  "SHELL=/bin/bash"
0xbffff734|+0x0014: 0xbffff899   →  "USER=root"
0xbffff738|+0x0018: 0xbffff8a3   →  "LS_COLORS=rs=0:di=01;34:ln=01;36:mh=00:pi=40;33:so[...]"
0xbffff73c|+0x001c: 0xbffffe2b   →  "SUDO_USER=kaiyu"
code:x86:32
    0x804806a <one+8>          mov    BYTE PTR [ebx+0x7], 0x0
    0x804806e <one+12>         mov    DWORD PTR [ebx+0x8], ebx
    0x8048071 <one+15>         mov    DWORD PTR [ebx+0xc], eax
 →  0x8048074 <one+18>         lea    ecx, [ebx+0x8]
    0x8048077 <one+21>         mov    edx, 0x0
    0x804807c <one+26>         mov    al, 0xb
    0x804807e <one+28>         int    0x80
    0x8048080 <two+0>          call   0x8048062 <one>
    0x8048085 <two+5>          das
threads
[#0] Id 1, Name: "labsh_rel", stopped 0x8048074 in one (), reason: SINGLE STEP
trace
[#0] 0x8048074 →one()
gef➤  x/4 0x0804808d
0x804808d <two+13>:     0x8048085       0x0     0x0     0x0
```

(9) Then we use the instruction "lea ecx, [ebx+8]" to store the address of the array argv[] in the ecx register. Now ecx points to the array argv[].

(10) Then we use the instruction "mov edx, 0" because we don't need to pass environment variables.

(11) Then we use the instruction "mov al, 0x0b" to set eax = 0x0000000b because the system call number of execve is 11 (0x0000000b).



(12) Then we use the instruction "int 0x80" to invoke kernel actions from a userland process to do the system call.