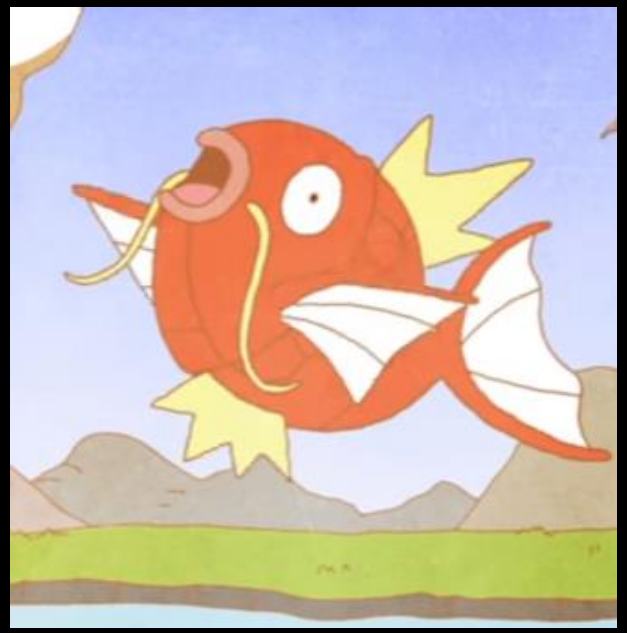
Format String Vulnerability

BAMBOOFOX

whoami

- briansp8210
- 交通大學資工系大二
- 喜歡學習 pwn 題的相關技巧 XD



What is it ?

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

int main(int argc, char **argv)
{
    char buffer[40];

    strncpy(buffer, argv[1], 40);
    printf("%s", buffer);
    printf("\n");

    return 0;
}
```

```
brian@MyUbuntuServer:~$ ./fmt abc123
abc123
brian@MyUbuntuServer:~$ ./fmt '%p %p %p %p %p %p %p %p %p %p'
%p %p %p %p %p %p %p %p %p %p
brian@MyUbuntuServer:~$
```

What is it ?

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

int main(int argc, char **argv)
{
    char buffer[40];

    strncpy(buffer, argv[1], 40);
    printf(buffer);
    printf("\n");

    return 0;
}
```

```
brian@MyUbuntuServer:~$ ./fmt2 abc123
abc123
brian@MyUbuntuServer:~$ ./fmt2 '%p %p %p %p %p %p %p %p %p %p'
0xffdb7563 0x28 0x8048301 0x2 (nil) 0x25207025 0x70252070 0x20702520 0x25207025 0x70252070
brian@MyUbuntuServer:~$
```

Looking into stack with gdb

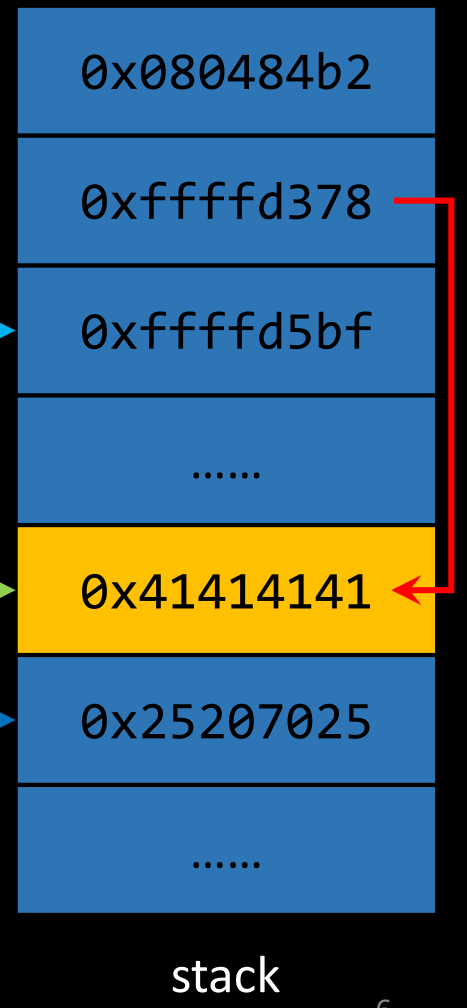
```
(gdb) r 'AAAA%p %p %p %p %p %p %p %p %p %p'
Starting program: /home/brian/fmt2 'AAAA%p %p %p %p %p %p %p %p %p %p'

Breakpoint 1, 0x08048330 in printf@plt ()
(gdb) x/16wx $esp
0xffffd35c:     0x080484b2      0xffffd378      0xffffd5bf      0x00000028
0xffffd36c:     0x08048301      0x00000002      0x00000000      0x41414141
0xffffd37c:     0x25207025      0x70252070      0x20702520      0x25207025
0xffffd38c:     0x70252070      0x20702520      0x25207025      0x00000070
(gdb) c
Continuing.
AAAA0xffffd5bf 0x28 0x8048301 0x2 (nil) 0x41414141 0x25207025 0x70252070 0x20702520 0x25207025
[Inferior 1 (process 2119) exited normally]
```

buffer

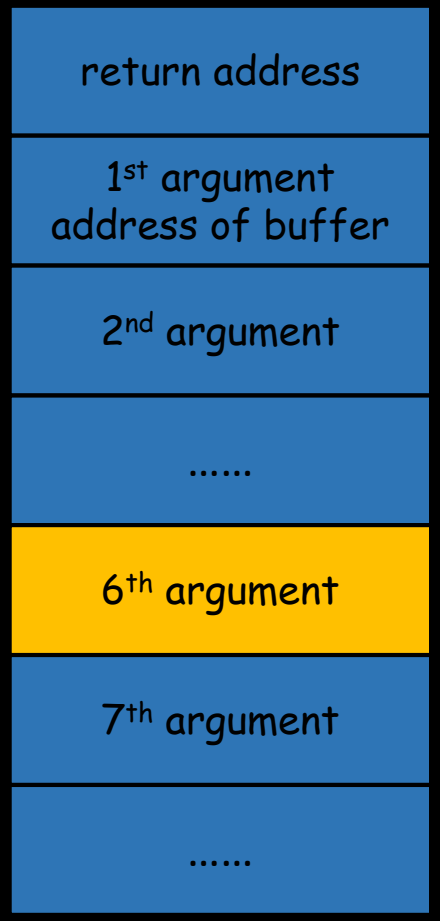
Looking into stack with gdb

```
printf("AAAA%p %p %p %p %p %p %p %p %p %p");
```



Looking into stack with gdb

- printf() can have variable number of arguments .
- For each format specifiers, it retrieves argument from stack .
- If we provide enough format specifiers, we can finally fetch our target .
- And when we get the offset of our target, we can do read / write on arbitrary address !



stack

Conversion specification

% [parameter] [flags] [width] [.precision] [length] specifier

Conversion specification

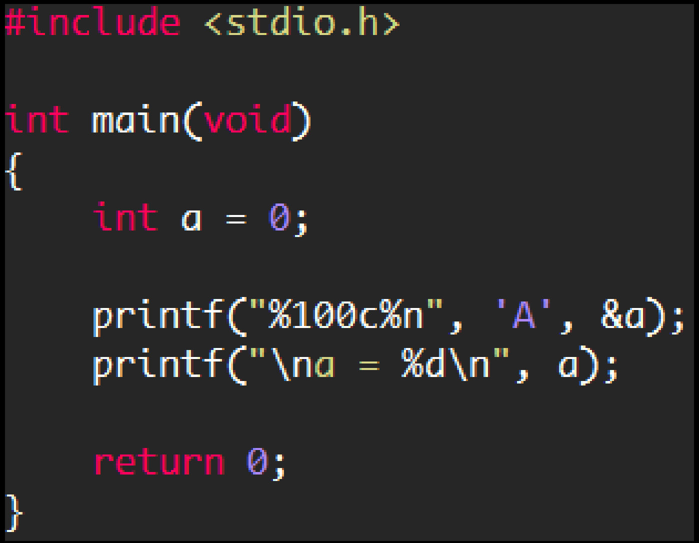
% [parameter] [flags] [width] [.precision] [length] specifier

%x : unsigned integer as hexadecimal
%s : string pointed by the argument
%p : pointer address
%n : store the number of characters written so far by current call of printf() to the location pointed by the correspond argument .

Conversion specification

% [parameter] [flags] [width] [.precision] [length] specifier

Example:

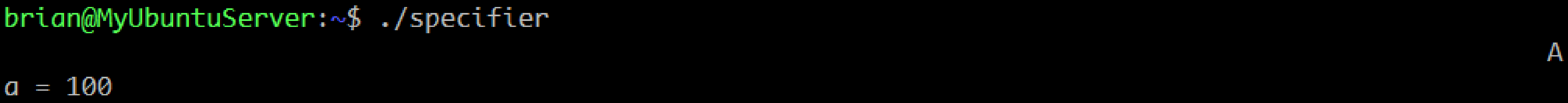


```c
#include <stdio.h>

int main(void)
{
    int a = 0;

    printf("%100c%n", 'A', &a);
    printf("\na = %d\n", a);

    return 0;
}
```



```
brian@MyUbuntuServer:~$ ./specifier
                                                                                                   A
a = 100
```

Usage:

Use with [width], we can write arbitrary value to an address .

Conversion specification

% [parameter] [flags] [width] [.precision] [length] specifier

(num)$: access the (num)th argument to manipulate

Conversion specification

% [parameter] [flags] [width] [.precision] [length] specifier

Example:

```c
#include <stdio.h>

int main(void)
{
    int a = 0;
    int b = 0;

    printf("%3$d, %2$x\n", 16, 17, 18, 19);
    printf("AAAA%2$n", &a, &b);

    printf("a = %d, b = %d\n", a, b);

    return 0;
}
```

```
brian@MyUbuntuServer:~$ ./parameter
18, 11
AAAAa = 0, b = 4
```

Usage:

when knowing the offset, we can use this to manipulate specific value on stack .

Conversion specification

% [parameter] [flags] [width] [.precision] [length] specifier

%(num)c : output at least *num* character.

Conversion specification

% [parameter] [flags] [width] [.precision] [length] specifier

Example:

```c
#include <stdio.h>

int main(void)
{
    int a = 0;

    printf("%100c%n", 'A', &a);
    printf("\na = %d\n", a);

    return 0;
}
```

```
brian@MyUbuntuServer:~$ ./specifier

                                                                                                    A

a = 100
```

Usage:

Use with %n, we can write arbitrary value to an address .

Conversion specification

% [parameter] [flags] [width] [.precision] [length] specifier

%h : interpret the argument as *short int or short int**

%hh : interpret the argument as *char or char**

Conversion specification

% [parameter] [flags] [width] [.precision] [length] specifier

Example:

```c
#include <stdio.h>

int main(void)
{
    int num = 0xabcd1234;

    printf("printf with %%x   : %x\n", num);
    printf("printf with %%hx  : %hx\n", num);
    printf("printf with %%hhx : %hhx\n", num);

    return 0;
}
```

```
brian@MyUbuntuServer:~$ ./length
printf with %x   : abcd1234
printf with %hx  : 1234
printf with %hhx : 34
```

Usage:

write only one or two bytes of data in the specified location.

modify the value of variable

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

char modifyme='X';

int main(void)
{
    char buf[100];

    printf("try to modify the value of variable !\n");

    gets(buf);
    printf(buf);

    if(modifyme=='O')
        printf("good!\n");

    return 0;
}
```

What else can we do ?

- **Bypass stack canary**
- **Leak libc address**
- **GOT hijacking**

What else can we do ?

- **Bypass stack canary**
- **Leak libc address**
- **GOT hijacking**

Stack canary

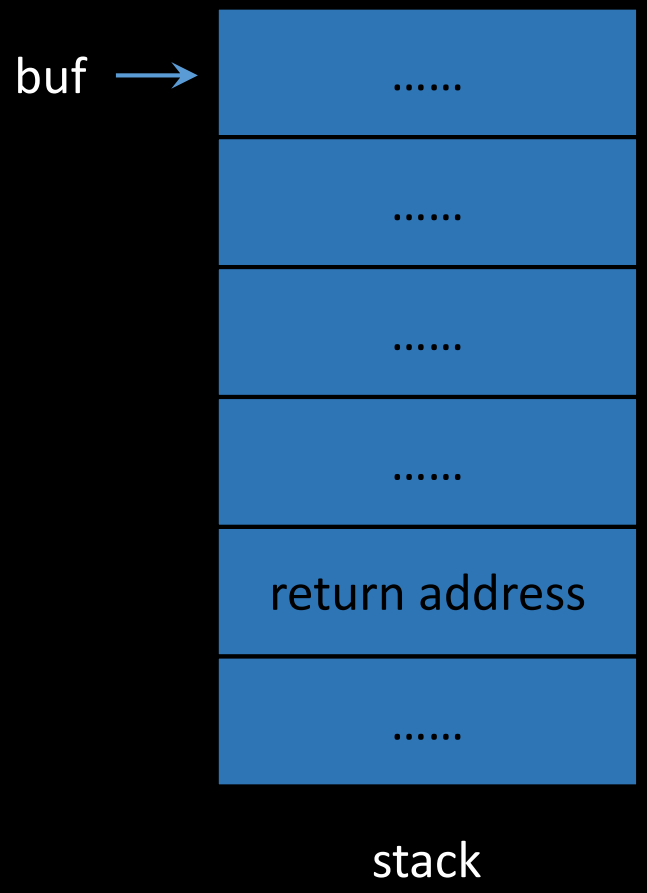
- Store a random number on stack .
- Before return, check whether the value is the same with the original one .
- Prevent the occurrence of stack overflow .
- gcc option:
  - `-fstack-protector` (enable)
  - `-fno-stack-protector` (disable)

```
Arch:     i386-32-little
RELRO:    Partial RELRO
Stack:    Canary found
NX:       NX enabled
PIE:      No PIE
```

How canary prevent us from exploiting

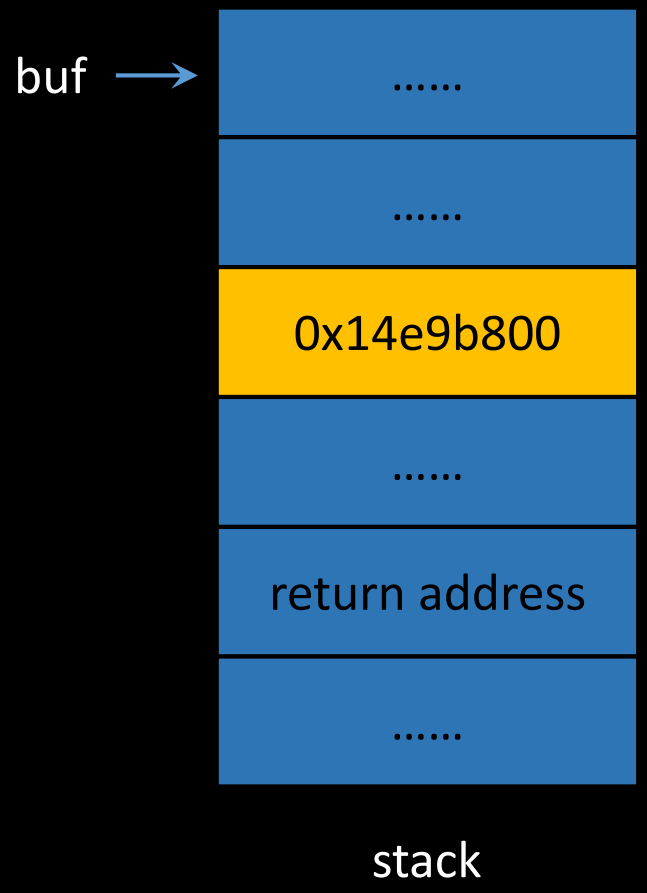
```
.....
mov     eax,gs:0x14
mov     DWORD PTR [esp+0x2c],eax
.....
call    0x8048330 <gets@plt>
.....
mov     edx,DWORD PTR [esp+0x2c]
xor     edx,DWORD PTR gs:0x14
je      0x80484a5 <main+56>
call    0x8048340 <__stack_chk_fail@plt>
leave
ret
```

eax = 0x14e9b800



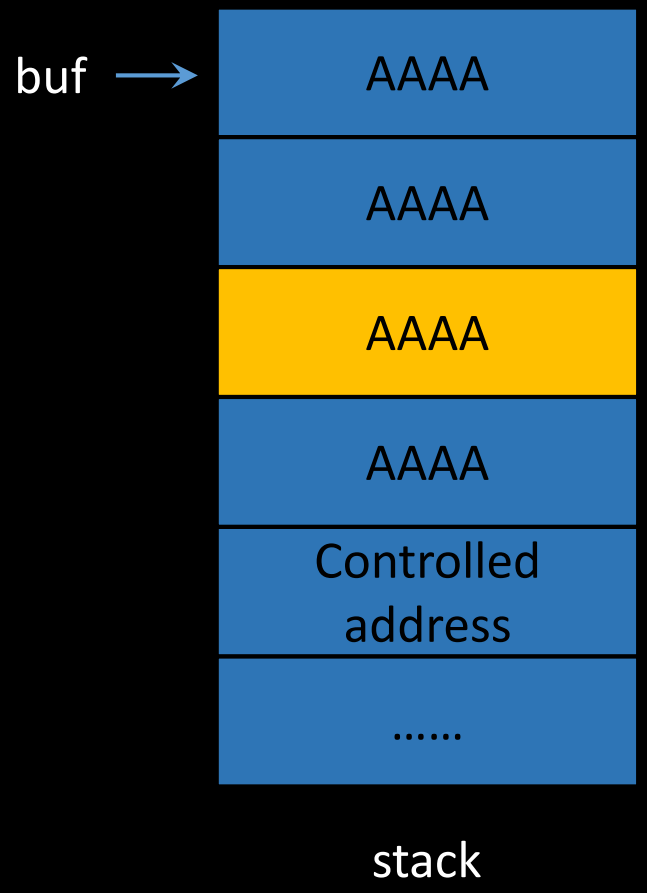
How canary prevent us from exploiting

```
.....
mov     eax,gs:0x14
mov     DWORD PTR [esp+0x2c],eax
.....
call    0x8048330 <gets@plt>
.....
mov     edx,DWORD PTR [esp+0x2c]
xor     edx,DWORD PTR gs:0x14
je      0x80484a5 <main+56>
call    0x8048340 <__stack_chk_fail@plt>
leave
ret
```



How canary prevent us from exploiting

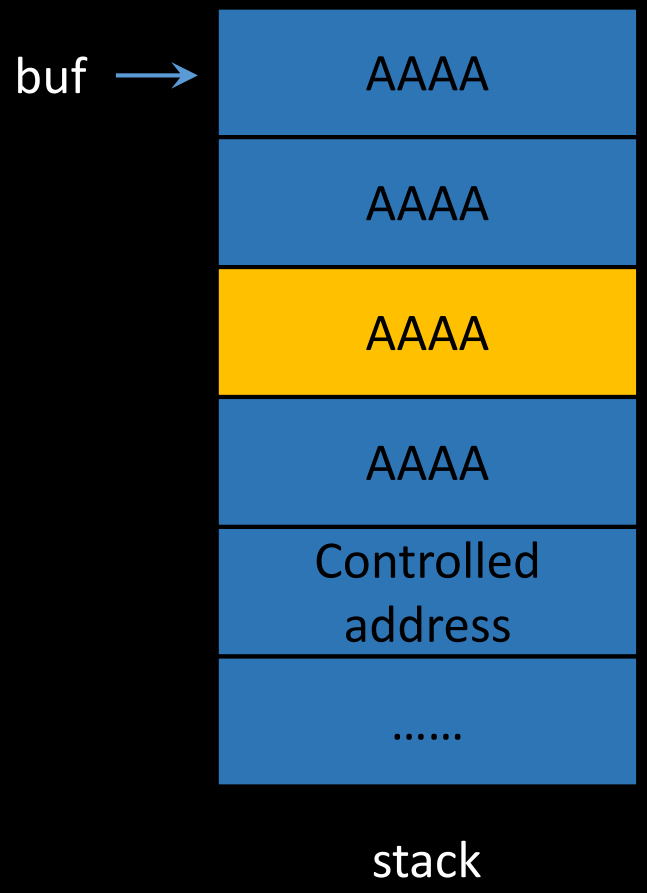
```
.....
mov     eax,gs:0x14
mov     DWORD PTR [esp+0x2c],eax
.....
call    0x8048330 <gets@plt>
.....
mov     edx,DWORD PTR [esp+0x2c]
xor     edx,DWORD PTR gs:0x14
je      0x80484a5 <main+56>
call    0x8048340 <__stack_chk_fail@plt>
leave
ret
```



How canary prevent us from exploiting

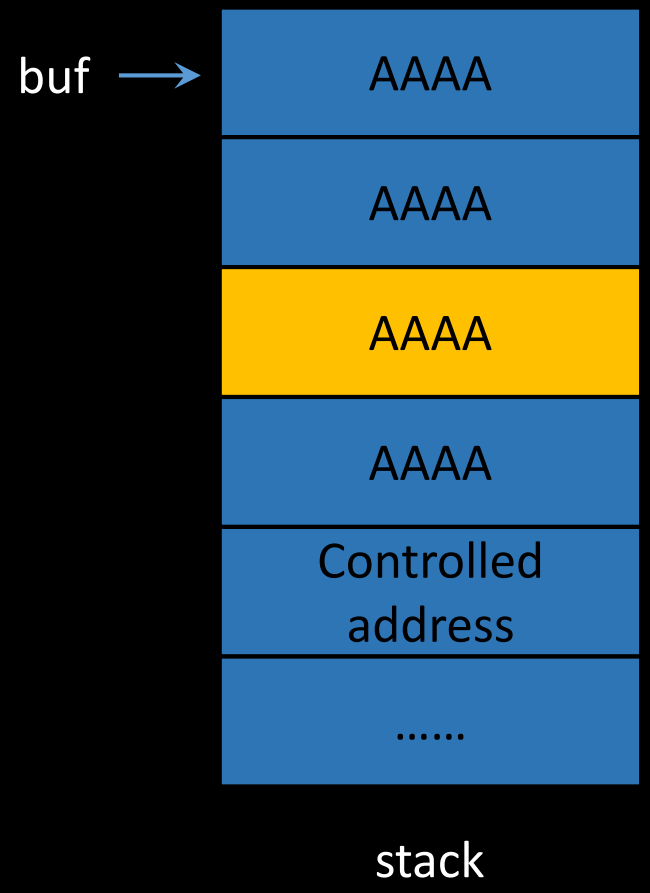
```
.....
mov     eax,gs:0x14
mov     DWORD PTR [esp+0x2c],eax
.....
call    0x8048330 <gets@plt>
.....
mov     edx,DWORD PTR [esp+0x2c]
xor     edx,DWORD PTR gs:0x14
je      0x80484a5 <main+56>
call    0x8048340 <__stack_chk_fail@plt>
leave
ret
```

edx = 0x41414141



How canary prevent us from exploiting

```
.....
mov     eax,gs:0x14
mov     DWORD PTR [esp+0x2c],eax
.....
call    0x8048330 <gets@plt>
.....
mov     edx,DWORD PTR [esp+0x2c]
xor     edx,DWORD PTR gs:0x14
je      0x80484a5 <main+56>
call    0x8048340 <__stack_chk_fail@plt>
leave
ret
```



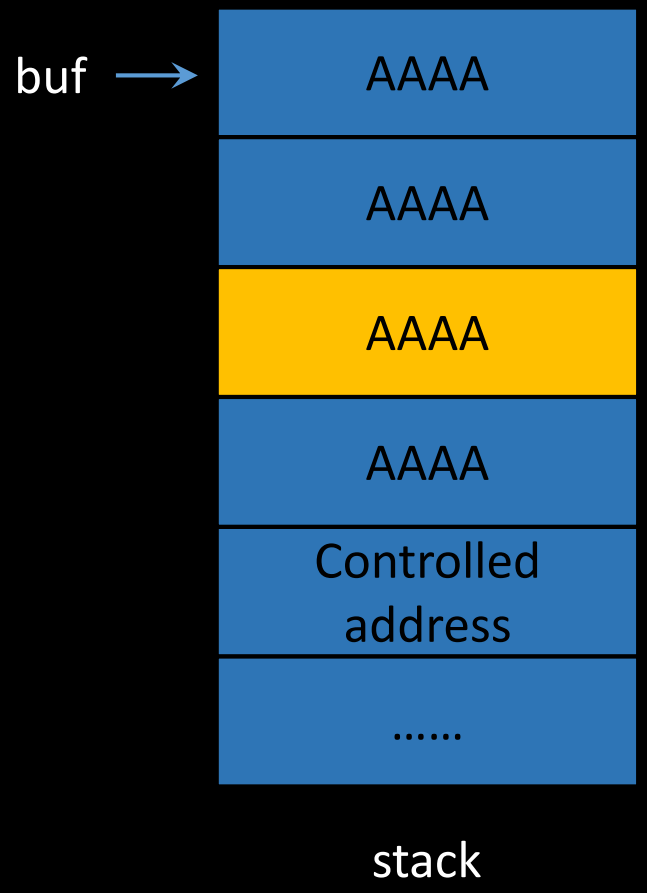
Because what we retrieve from stack now is not the original value, the result will not be 0 .

How canary prevent us from exploiting

```
.....
mov     eax,gs:0x14
mov     DWORD PTR [esp+0x2c],eax
.....
call    0x8048330 <gets@plt>
.....
mov     edx,DWORD PTR [esp+0x2c]
xor     edx,DWORD PTR gs:0x14
je      0x80484a5 <main+56>
call    0x8048340 <__stack_chk_fail@plt>
leave
ret
```

edx = 0x41414141

Since the result ≠ 0, it will not jump to <main+56>



How canary prevent us from exploiting

```
.....
mov     eax,gs:0x14
mov     DWORD PTR [esp+0x2c],eax
.....
call    0x8048330 <gets@plt>
.....
mov     edx,DWORD PTR [esp+0x2c]
xor     edx,DWORD PTR gs:0x14
je      0x80484a5 <main+56>
call    0x8048340 <__stack_chk_fail@plt>
leave
ret
```

```
brian@MyUbuntuServer:~$ ./with_canary
AAAAAAAAAAAAAAAAAAAAAAAAABBBB
*** stack smashing detected ***: ./with_canary terminated
Aborted (core dumped)
```

Bypass stack canary

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

void canary_protect_me(void)
{
    system("/bin/sh");
}

int main(void)
{
    setvbuf(stdout, 0LL, 2, 0LL);
    setvbuf(stdin, 0LL, 1, 0LL);
    char buf[40];

    gets(buf);
    printf(buf);

    gets(buf);

    return 0;
}
```

- We can use fmt vulnerability to leak the canary .
- When doing overflow, putting the leaked canary at the original position .

What else can we do ?

- Bypass stack canary
- **Leak libc address**
- **GOT hijacking**

Lazy binding

- Solve the waste of linking all function
  - binding a function when it was called the first time
  - without binding unused function
  - Implemented by PLT (Procedure Linkage Table)

- _dl_runtime_resolve()
  - resolve a function which is called the first time

How it exactly works

```c
#include <stdio.h>

int main(void)
{
    int i;

    for(i=0; i<5; i++)
        printf("%d\n", i);

    return 0;
}
```

How it exactly works

```
0x8048425 <main+26>          sub    esp,0x8
0x8048428 <main+29>          push   DWORD PTR [ebp-0xc]
0x804842b <main+32>          push   0x80484d0
0x8048430 <main+37>          call   0x80482e0 <printf@plt>
0x8048435 <main+42>          add    esp,0x10
```

```
080482d0 <printf@plt-0x10>:
 80482d0:       ff 35 04 a0 04 08       push   DWORD PTR ds:0x804a004
 80482d6:       ff 25 08 a0 04 08       jmp    DWORD PTR ds:0x804a008
 80482dc:       00 00                   add    BYTE PTR [eax],al
        ...

080482e0 <printf@plt>:
 80482e0:       ff 25 0c a0 04 08       jmp    DWORD PTR ds:0x804a00c
 80482e6:       68 00 00 00 00          push   0x0
 80482eb:       e9 e0 ff ff ff          jmp    80482d0 <_init+0x24>
```

When we call the function, it jumps to the .plt entry . This is why we can use .plt entry of a function as the address of the function in ret2libc exploit .

How it exactly works

```
0x8048425 <main+26>          sub    esp,0x8
0x8048428 <main+29>          push   DWORD PTR [ebp-0xc]
0x804842b <main+32>          push   0x80484d0
0x8048430 <main+37>          call   0x80482e0 <printf@plt>
0x8048435 <main+42>          add    esp,0x10
```

```
080482d0 <printf@plt-0x10>:
 80482d0:       ff 35 04 a0 04 08       push   DWORD PTR ds:0x804a004
 80482d6:       ff 25 08 a0 04 08       jmp    DWORD PTR ds:0x804a008
 80482dc:       00 00                   add    BYTE PTR [eax],al
        ...

080482e0 <printf@plt>:
 80482e0:       ff 25 0c a0 04 08       jmp    DWORD PTR ds:0x804a00c
 80482e6:       68 00 00 00 00          push   0x0
 80482eb:       e9 e0 ff ff ff          jmp    80482d0 <_init+0x24>
```

If the function is called for the first time, its GOT entry just stores the next instruction's address .

```
(gdb) x/wx 0x804a00c
0x804a00c:      0x080482e6
```

How it exactly works

```
0x8048425 <main+26>          sub    esp,0x8
0x8048428 <main+29>          push   DWORD PTR [ebp-0xc]
0x804842b <main+32>          push   0x80484d0
0x8048430 <main+37>          call   0x80482e0 <printf@plt>
0x8048435 <main+42>          add    esp,0x10
```

```
080482d0 <printf@plt-0x10>:
 80482d0:       ff 35 04 a0 04 08       push   DWORD PTR ds:0x804a004
 80482d6:       ff 25 08 a0 04 08       jmp    DWORD PTR ds:0x804a008
 80482dc:       00 00                   add    BYTE PTR [eax],al
        ...

080482e0 <printf@plt>:
 80482e0:       ff 25 0c a0 04 08       jmp    DWORD PTR ds:0x804a00c
 80482e6:       68 00 00 00 00          push   0x0
 80482eb:       e9 e0 ff ff ff          jmp    80482d0 <_init+0x24>
```

How it exactly works

```
0x8048425 <main+26>          sub    esp,0x8
0x8048428 <main+29>          push   DWORD PTR [ebp-0xc]
0x804842b <main+32>          push   0x80484d0
0x8048430 <main+37>          call   0x80482e0 <printf@plt>
0x8048435 <main+42>          add    esp,0x10
```

```
080482d0 <printf@plt-0x10>:
 80482d0:       ff 35 04 a0 04 08       push   DWORD PTR ds:0x804a004
 80482d6:       ff 25 08 a0 04 08       jmp    DWORD PTR ds:0x804a008
 80482dc:       00 00                   add    BYTE PTR [eax],al
        ...

080482e0 <printf@plt>:
 80482e0:       ff 25 0c a0 04 08       jmp    DWORD PTR ds:0x804a00c
 80482e6:       68 00 00 00 00          push   0x0
 80482eb:       e9 e0 ff ff ff          jmp    80482d0 <_init+0x24>
```

```
0xf7ff04e0 <_dl_runtime_resolve>        push   eax
0xf7ff04e1 <_dl_runtime_resolve+1>      push   ecx
0xf7ff04e2 <_dl_runtime_resolve+2>      push   edx
0xf7ff04e3 <_dl_runtime_resolve+3>      mov    edx,DWORD PTR [esp+0x10]
0xf7ff04e7 <_dl_runtime_resolve+7>      mov    eax,DWORD PTR [esp+0xc]
0xf7ff04eb <_dl_runtime_resolve+11>     call   0xf7fea76a <_dl_fixup>
0xf7ff04f0 <_dl_runtime_resolve+16>     pop    edx
0xf7ff04f1 <_dl_runtime_resolve+17>     mov    ecx,DWORD PTR [esp]
0xf7ff04f4 <_dl_runtime_resolve+20>     mov    DWORD PTR [esp],eax
0xf7ff04f7 <_dl_runtime_resolve+23>     mov    eax,DWORD PTR [esp+0x4]
0xf7ff04fb <_dl_runtime_resolve+27>     ret    0xc
```

After setting arguments,
call `_dl_runtime_resolve()`

How it exactly works

```
0x8048425 <main+26>          sub    esp,0x8
0x8048428 <main+29>          push   DWORD PTR [ebp-0xc]
0x804842b <main+32>          push   0x80484d0
0x8048430 <main+37>          call   0x80482e0 <printf@plt>
0x8048435 <main+42>          add    esp,0x10
```

```
080482d0 <printf@plt-0x10>:
 80482d0:       ff 35 04 a0 04 08       push   DWORD PTR ds:0x804a004
 80482d6:       ff 25 08 a0 04 08       jmp    DWORD PTR ds:0x804a008
 80482dc:       00 00                   add    BYTE PTR [eax],al
        ...

080482e0 <printf@plt>:
 80482e0:       ff 25 0c a0 04 08       jmp    DWORD PTR ds:0x804a00c
 80482e6:       68 00 00 00 00          push   0x0
 80482eb:       e9 e0 ff ff ff          jmp    80482d0 <_init+0x24>
```

```
0xf7ff04e0 <_dl_runtime_resolve>        push   eax
0xf7ff04e1 <_dl_runtime_resolve+1>      push   ecx
0xf7ff04e2 <_dl_runtime_resolve+2>      push   edx
0xf7ff04e3 <_dl_runtime_resolve+3>      mov    edx,DWORD PTR [esp+0x10]
0xf7ff04e7 <_dl_runtime_resolve+7>      mov    eax,DWORD PTR [esp+0xc]
0xf7ff04eb <_dl_runtime_resolve+11>     call   0xf7fea76a <_dl_fixup>
0xf7ff04f0 <_dl_runtime_resolve+16>     pop    edx
0xf7ff04f1 <_dl_runtime_resolve+17>     mov    ecx,DWORD PTR [esp]
0xf7ff04f4 <_dl_runtime_resolve+20>     mov    DWORD PTR [esp],eax
0xf7ff04f7 <_dl_runtime_resolve+23>     mov    eax,DWORD PTR [esp+0x4]
0xf7ff04fb <_dl_runtime_resolve+27>     ret    0xc
```

```
0xf7e81406 <__printf>          push   ebx
0xf7e81407 <__printf+1>        sub    esp,0x18
0xf7e8140a <__printf+4>        call   0xf7f5397a <__x86.get_pc_thunk.bx>
0xf7e8140f <__printf+9>        add    ebx,0x153bf1
0xf7e81415 <__printf+15>       lea    edx,[esp+0x24]
0xf7e81419 <__printf+19>       mov    eax,DWORD PTR [ebx-0x78]
0xf7e8141f <__printf+25>       mov    eax,DWORD PTR [eax]
0xf7e81421 <__printf+27>       mov    DWORD PTR [esp+0x8],edx
0xf7e81425 <__printf+31>       mov    edx,DWORD PTR [esp+0x20]
0xf7e81429 <__printf+35>       mov    DWORD PTR [esp+0x4],edx
0xf7e8142d <__printf+39>       mov    DWORD PTR [esp],eax
0xf7e81430 <__printf+42>       call   0xf7e77325 <_IO_vfprintf_internal>
0xf7e81435 <__printf+47>       add    esp,0x18
0xf7e81438 <__printf+50>       pop    ebx
0xf7e81439 <__printf+51>       ret
```

How it exactly works

```
0x8048425 <main+26>          sub    esp,0x8
0x8048428 <main+29>          push   DWORD PTR [ebp-0xc]
0x804842b <main+32>          push   0x80484d0
0x8048430 <main+37>          call   0x80482e0 <printf@plt>
0x8048435 <main+42>          add    esp,0x10
```

```
080482d0 <printf@plt-0x10>:
 80482d0:       ff 35 04 a0 04 08       push   DWORD PTR ds:0x804a004
 80482d6:       ff 25 08 a0 04 08       jmp    DWORD PTR ds:0x804a008
 80482dc:       00 00                   add    BYTE PTR [eax],al
        ...

080482e0 <printf@plt>:
 80482e0:       ff 25 0c a0 04 08       jmp    DWORD PTR ds:0x804a00c
 80482e6:       68 00 00 00 00          push   0x0
 80482eb:       e9 e0 ff ff ff          jmp    80482d0 <_init+0x24>
```

When the function is called again, it will still jump to .plt entry .

How it exactly works

```
0x8048425 <main+26>          sub    esp,0x8
0x8048428 <main+29>          push   DWORD PTR [ebp-0xc]
0x804842b <main+32>          push   0x80484d0
0x8048430 <main+37>          call   0x80482e0 <printf@plt>
0x8048435 <main+42>          add    esp,0x10
```

```
080482d0 <printf@plt-0x10>:
 80482d0:       ff 35 04 a0 04 08       push   DWORD PTR ds:0x804a004
 80482d6:       ff 25 08 a0 04 08       jmp    DWORD PTR ds:0x804a008
 80482dc:       00 00                   add    BYTE PTR [eax],al
        ...

080482e0 <printf@plt>:
 80482e0:       ff 25 0c a0 04 08       jmp    DWORD PTR ds:0x804a00c
 80482e6:       68 00 00 00 00          push   0x0
 80482eb:       e9 e0 ff ff ff          jmp    80482d0 <_init+0x24>
```

Since the address of the function in libc has been written on its GOT entry, it can directly jump and execute it .

```
0xf7e81406 <__printf>          push   ebx
0xf7e81407 <__printf+1>        sub    esp,0x18
0xf7e8140a <__printf+4>        call   0xf7f5397a <__x86.get_pc_thunk.bx>
0xf7e8140f <__printf+9>        add    ebx,0x153bf1
0xf7e81415 <__printf+15>       lea    edx,[esp+0x24]
0xf7e81419 <__printf+19>       mov    eax,DWORD PTR [ebx-0x78]
0xf7e8141f <__printf+25>       mov    eax,DWORD PTR [eax]
0xf7e81421 <__printf+27>       mov    DWORD PTR [esp+0x8],edx
0xf7e81425 <__printf+31>       mov    edx,DWORD PTR [esp+0x20]
0xf7e81429 <__printf+35>       mov    DWORD PTR [esp+0x4],edx
0xf7e8142d <__printf+39>       mov    DWORD PTR [esp],eax
0xf7e81430 <__printf+42>       call   0xf7e77325 <_IO_vfprintf_internal>
0xf7e81435 <__printf+47>       add    esp,0x18
0xf7e81438 <__printf+50>       pop    ebx
0xf7e81439 <__printf+51>       ret
```

Leak libc address

- For a dynamically linked program, foo() has been called at least once .
  - GOT entry of foo() will be address of foo() in libc .

- Use the arbitrary read ability of format string vulnerability to leak this address.
  - If we know the offset of foo() in libc
  - Base address = address of foo() - offset

GOT hijacking

- To implement lazy binding, GOT is writable .
- ➢ overwrite foo()'s GOT entry to bar()'s address .
- ➢ next time program want to call foo(), will actually call bar() .

```
08048410 <printf@plt>:
 8048410:       ff 25 10 a0 04 08       jmp    DWORD PTR ds:0x804a010
 8048416:       68 08 00 00 00          push   0x8
 804841b:       e9 d0 ff ff ff          jmp    80483f0 <_init+0x2c>

08048440 <system@plt>:
 8048440:       ff 25 1c a0 04 08       jmp    DWORD PTR ds:0x804a01c
 8048446:       68 20 00 00 00          push   0x20
 804844b:       e9 a0 ff ff ff          jmp    80483f0 <_init+0x2c>
```

```
0xf764a020 <printf>          call   0xf771e0d9
0xf764a025 <printf+5>        add    eax,0x166fdb
0xf764a02a <printf+10>       sub    esp,0xc
0xf764a02d <printf+13>       mov    eax,DWORD PTR [eax-0x68]
```

```
0xf763b940 <system>          sub    esp,0xc
0xf763b943 <system+3>        mov    eax,DWORD PTR [esp+0x10]
0xf763b947 <system+7>        call   0xf771e0dd
0xf763b94c <system+12>       add    edx,0x1756b4
```

GOT hijacking

- To implement lazy binding, GOT is writable .
- ➢ overwrite foo()'s GOT entry to bar()'s address .
- ➢ next time program want to call foo(), will actually call bar() .

```
08048410 <printf@plt>:
 8048410:       ff 25 10 a0 04 08       jmp    DWORD PTR ds:0x804a010
 8048416:       68 08 00 00 00          push   0x8
 804841b:       e9 d0 ff ff ff          jmp    80483f0 <_init+0x2c>

08048440 <system@plt>:
 8048440:       ff 25 1c a0 04 08       jmp    DWORD PTR ds:0x804a01c
 8048446:       68 20 00 00 00          push   0x20
 804844b:       e9 a0 ff ff ff          jmp    80483f0 <_init+0x2c>
```

```
0xf764a020 <printf>          call   0xf771e0d9
0xf764a025 <printf+5>        add    eax,0x166fdb
0xf764a02a <printf+10>       sub    esp,0xc
0xf764a02d <printf+13>       mov    eax,DWORD PTR [eax-0x68]
```

```
0xf763b940 <system>          sub    esp,0xc
0xf763b943 <system+3>        mov    eax,DWORD PTR [esp+0x10]
0xf763b947 <system+7>        call   0xf771e0dd
0xf763b94c <system+12>       add    edx,0x1756b4
```

GOT hijacking

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

void noShell(void)
{
    system("QAQ");
}

int main(void)
{
    setvbuf(stdout, 0LL, 2, 0LL);
    setvbuf(stdin, 0LL, 1, 0LL);

    char msg[128];

    puts("====================");
    puts("  Echo Server v1.0");
    puts("====================");

    for(;;)
    {
        if(!strcmp(msg, "Exit"))
            exit(0);
        fgets(msg, 128, stdin);
        printf(msg);
    }
}
```

Reference

- 程式設計師的自我修養 --連結. 載入. 程式庫
- Format String Vulnerability
- Introduction to format string exploits
- Execution of ELF
- Runtime Symbol Resolution